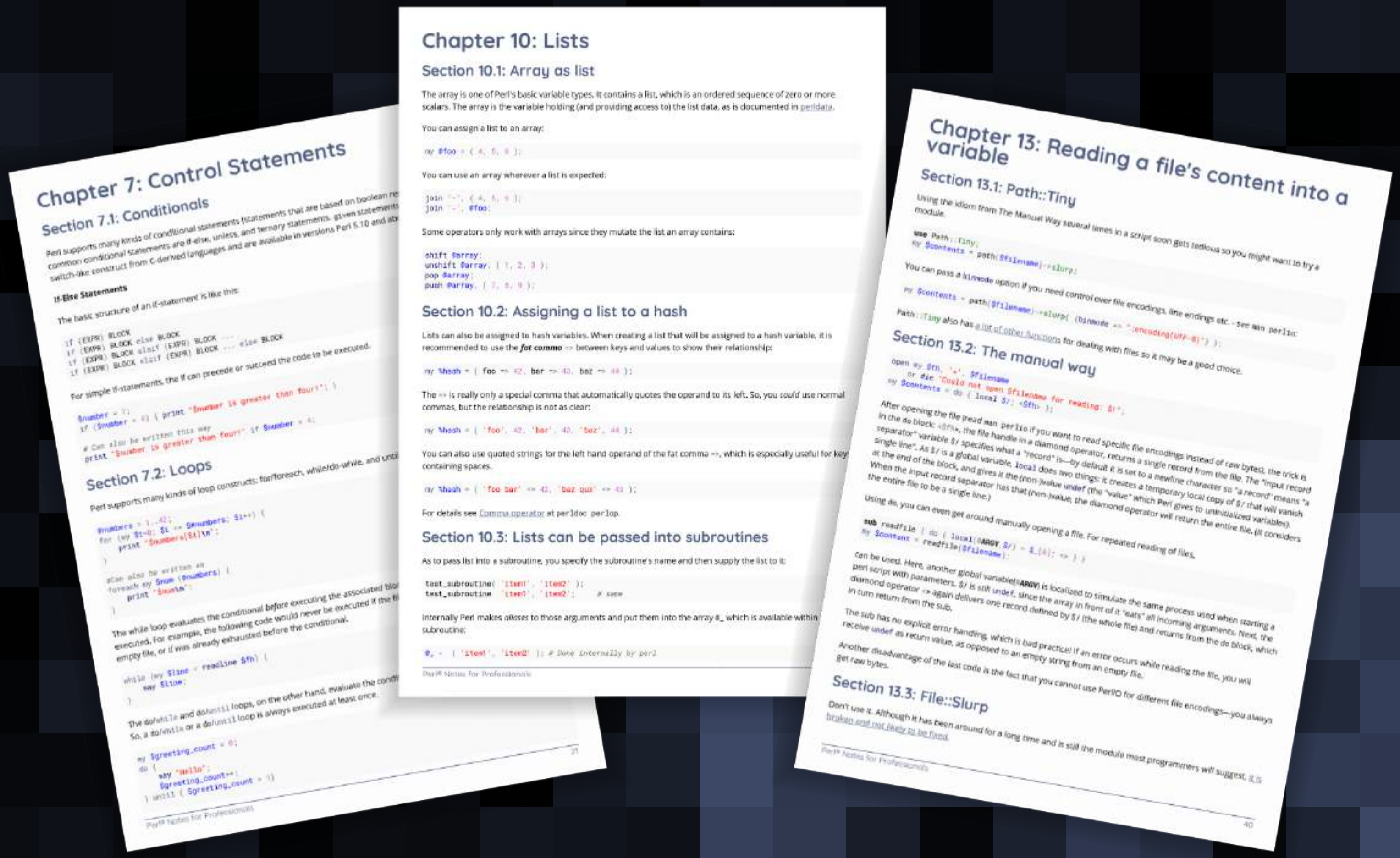
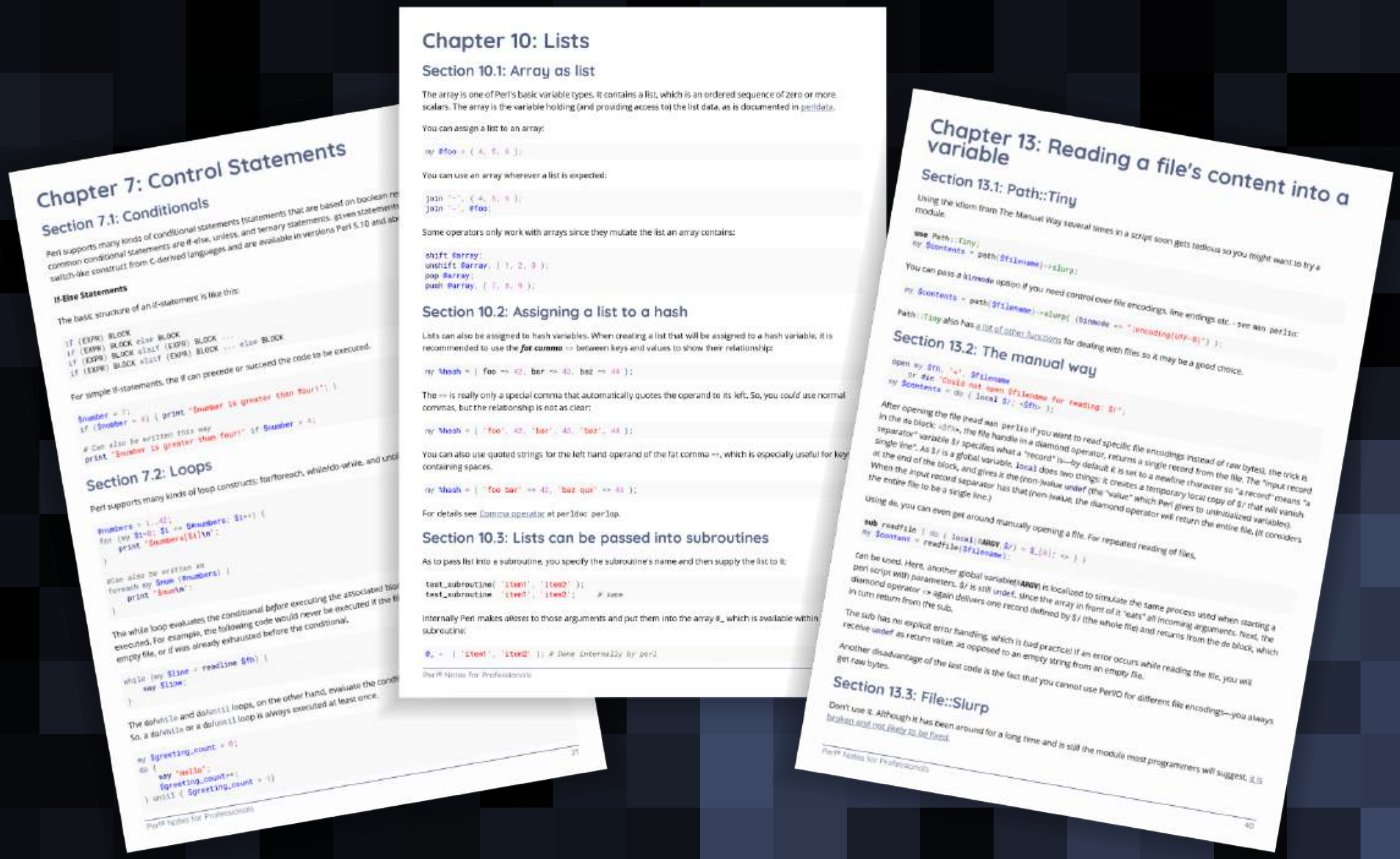
Perl®
专业人士笔记



90+ 页
专业提示和技巧

GoalKicker.com
免费编程书籍

Perl®
Notes for Professionals



90+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

目录

Contents

关于

About

第1章：开始使用Perl语言

版本	发布说明	发布日期
1.000		1987-12-18
2.000		1988-06-05
3.000		1989-10-18
4.000		1991-03-21
5.000		1994-10-17
5.001		1995-05-13
5.002		1996-02-29
5.003		1996-06-25
5.004	perl5004delta	1997-05-15
5.005	perl5005delta	1998-07-22
5.6.0	perl56delta	2000-03-22
5.8.0	perl58delta	2002-07-18
5.8.8	perl581delta, perl582delta, perl583delta, perl584delta, perl585delta, perl586delta, perl587delta, perl588delta	2006-02-01
5.10.0	perl5100delta	2007-12-18
5.12.0	perl5120delta	2010-04-12
5.14.0	perl5140delta	2011-05-14
5.16.0	perl5160delta	2012-05-20
5.18.0	perl5180delta	2013-05-18
5.20.0	perl5200delta	2014-05-27
5.22.0	perl5220delta	2015-06-01
5.24.0	perl5240delta	2016-05-09
5.26.0	perl5260delta	2017-05-30

第1.1节：Perl入门

Perl尝试实现你的意图：

```
print "Hello World";
```

两个难点是行尾的分号和
，它会添加一个换行符（换行）。如果你使用的是较新的Perl版本，可以使用say代替print，自动添加回车符：

版本 ≥ 5.10.0

```
use feature 'say';
say "Hello World";
```

使用 v5.10（或更高版本）声明时，say 功能也会自动启用：

```
use v5.10;
```

Chapter 1: Getting started with Perl Language

Version	Release Notes	Release Date
1.000		1987-12-18
2.000		1988-06-05
3.000		1989-10-18
4.000		1991-03-21
5.000		1994-10-17
5.001		1995-05-13
5.002		1996-02-29
5.003		1996-06-25
5.004	perl5004delta	1997-05-15
5.005	perl5005delta	1998-07-22
5.6.0	perl56delta	2000-03-22
5.8.0	perl58delta	2002-07-18
5.8.8	perl581delta, perl582delta, perl583delta, perl584delta, perl585delta, perl586delta, perl587delta, perl588delta	2006-02-01
5.10.0	perl5100delta	2007-12-18
5.12.0	perl5120delta	2010-04-12
5.14.0	perl5140delta	2011-05-14
5.16.0	perl5160delta	2012-05-20
5.18.0	perl5180delta	2013-05-18
5.20.0	perl5200delta	2014-05-27
5.22.0	perl5220delta	2015-06-01
5.24.0	perl5240delta	2016-05-09
5.26.0	perl5260delta	2017-05-30

Section 1.1: Getting started with Perl

Perl tries to do what you mean:

```
print "Hello World\n";
```

The two tricky bits are the semicolon at the end of the line and the `\n`, which adds a newline (line feed). If you have a relatively new version of perl, you can use `say` instead of `print` to have the carriage return added automatically:

Version ≥ 5.10.0

```
use feature 'say';
say "Hello World";
```

The say feature is also enabled automatically with a `use v5.10` (or higher) declaration:

```
use v5.10;
```

```
say "Hello World";
```

在命令行上使用 perl 并带上 -e 选项是非常常见的做法：

```
$ perl -e 'print "Hello World"'Hello Worl
d
```

添加 -l 选项是一种自动打印换行符的方法：

```
$ perl -le 'print "Hello World"'
Hello World
```

版本 ≥ 5.10.0

如果想启用 新特性，应该使用 -E 选项：

```
$ perl -E 'say "Hello World"'
Hello World
```

当然，你也可以将脚本保存在文件中。只需去掉-e命令行选项，使用脚本的文件名：perl script.pl。对于超过一行的程序，建议开启几个选项：

```
use strict;use
 warnings;print

 "Hello World";
```

这样做没有真正的缺点，只是代码稍微长一点。作为交换，strict pragma防止你使用潜在不安全的代码，warnings会提醒你许多常见错误。

注意，最后一行的行尾分号是可选的，但建议加上，以防你以后在代码末尾添加内容。

有关如何运行Perl的更多选项，请参见perlrun或在命令提示符下输入perldoc perlrun。有关Perl的更详细介绍，请参见perlintro或输入perldoc perlintro。想要一个有趣的交互式教程，请试试Try Perl。

```
say "Hello World";
```

It's pretty common to just use perl on the command line using the -e option:

```
$ perl -e 'print "Hello World\n"'
Hello World
```

Adding the -l option is one way to print newlines automatically:

```
$ perl -le 'print "Hello World"'
Hello World
```

Version ≥ 5.10.0

If you want to enable new features, use the -E option instead:

```
$ perl -E 'say "Hello World"'
Hello World
```

You can also, of course, save the script in a file. Just remove the -e command line option and use the filename of the script: `perl script.pl`. For programs longer than a line, it's wise to turn on a couple of options:

```
use strict;
use warnings;

print "Hello World\n";
```

There's no real disadvantage other than making the code slightly longer. In exchange, the strict pragma prevents you from using code that is potentially unsafe and warnings notifies you of many common errors.

Notice the line-ending semicolon is optional for the last line, but is a good idea in case you later add to the end of your code.

For more options how to run Perl, see perlrun or type `perldoc perlrun` at a command prompt. For a more detailed introduction to Perl, see perlintro or type `perldoc perlintro` at a command prompt. For a quirky interactive tutorial, Try Perl.

第2章：注释

第2.1节：单行注释

单行注释以井号#开头，直到行尾：

```
# 这是一个注释

my $foo = "bar"; # 这也是一个注释
```

第2.2节：多行注释

多行注释以=开始，以=cut语句结束。这些是称为POD（Plain Old
Documentation，纯旧文档）的特殊注释。

标记之间的任何文本都会被注释掉：

```
=begin comment

这是另一条注释。
并且它跨越多行！

=end comment

=cut
```

Chapter 2: Comments

Section 2.1: Single-line comments

Single-line comments begin with a pound sign # and go to the end of the line:

```
# This is a comment

my $foo = "bar"; # This is also a comment
```

Section 2.2: Multi-line comments

Multi-line comments start with = and with the =cut statement. These are special comments called POD (Plain Old Documentation).

Any text between the markers will be commented out:

```
=begin comment

This is another comment.
And it spans multiple lines!

=end comment

=cut
```

第3章：变量

第3.1节：标量

标量是Perl最基本的数据类型。它们以符号$标记，保存三种类型之一的单个值：

- **一个数字**（3，42，3.141等）
- **一个字符串**（'hi'，"abc"等）
- **一个变量的引用**（见其他示例）。

```
my $integer = 3;                      # 数字
my $string = "Hello World";           # 字符串
my $reference = \$string;             # 引用 $string
```

Perl 根据特定运算符的需求，动态地在数字和字符串之间转换。

```
my $number = '41';                    # 字符串 '41'
my $meaning = $number + 1;            # 数字 42
my $sadness = '20 apples';            # 字符串 '20 apples'
my $danger = $sadness * 2;            # 数字 '40'，会产生警告
```

在将字符串转换为数字时，Perl 会尽可能从字符串开头取出数字部分–因此最后一行中的20 apples被转换成了20。

根据你想将标量内容视为字符串还是数字，需要使用不同的运算符。不要混用它们。

```
# 字符串比较                 # 数字比较
'Potato' eq 'Potato';                   42 == 42;
'Potato' ne 'Pomato';                   42 != 24;
'Camel'  lt 'Potato';                   41 < 42;
'Zombie' gt 'Potato';                   43 > 42;

# 字符串连接               # 数字求和
'Banana' . 'phone';                23 + 19;

# 字符串重复                 # 数字乘法
'nan' x 3;                          6 * 7;
```

尝试对数字使用字符串操作不会引发警告；尝试对非数字字符串使用数字操作则会引发警告。请注意，某些非数字字符串如'inf'、'nan'、'0 but true'被视为数字。

第3.2节：数组引用

数组引用是标量（$），它们引用数组。

```
my @array = ("Hello"); # 创建数组，从列表赋值
my $array_reference = \@array;
```

这些可以更简洁地创建，如下所示：

```
my $other_array_reference = ["Hello"];
```

修改/使用数组引用需要先对其进行解引用。

Chapter 3: Variables

Section 3.1: Scalars

Scalars are Perl's most basic data type. They're marked with the sigil $ and hold a single value of one of three types:

- **a number** (3, 42, 3.141, etc.)
- **a string** ('hi', "abc", etc.)
- **a reference** to a variable (see other examples).

```
my $integer = 3;                      # number
my $string = "Hello World";           # string
my $reference = \$string;             # reference to $string
```

Perl converts between numbers and strings on the fly, based on what a particular operator expects.

```
my $number = '41';                    # string '41'
my $meaning = $number + 1;            # number  42
my $sadness = '20 apples';            # string '20 apples'
my $danger = $sadness * 2;            # number '40', raises warning
```

When converting a string into a number, Perl takes as many digits from the front of a string as it can – hence why 20 apples is converted into 20 in the last line.

Based on whether you want to treat the contents of a scalar as a string or a number, you need to use different operators. Do not mix them.

```
# String comparison                  # Number comparison
'Potato' eq 'Potato';                 42 == 42;
'Potato' ne 'Pomato';                 42 != 24;
'Camel'  lt 'Potato';                 41 < 42;
'Zombie' gt 'Potato';                 43 > 42;

# String concatenation               # Number summation
'Banana' . 'phone';                   23 + 19;

# String repetition                  # Number multiplication
'nan' x 3;                            6 * 7;
```

Attempting to use string operations on numbers will not raise warnings; attempting to use number operations on non-numeric strings will. Do be aware that some non-digit strings such as 'inf', 'nan', '0 but true' count as numbers.

Section 3.2: Array References

Array References are scalars ($) which refer to Arrays.

```
my @array = ("Hello"); # Creating array, assigning value from a list
my $array_reference = \@array;
```

These can be created more short-hand as follows:

```
my $other_array_reference = ["Hello"];
```

Modifying / Using array references require dereferencing them first.

```
my @contents = @{ $array_reference };                # 前缀表示法
my @contents = @$array_reference;                    # 大括号可省略
```
版本 ≥ 5.24.0

新的后缀解引用语法，v5.24起默认可用

```
use v5.24;
my @contents = $array_reference->@*; # 新的后缀表示法
```

访问数组引用的内容时，可以使用->语法糖。

```
my @array = qw(one two three);      my $arrayref = [ qw(one two three) ]
my $one = $array[0];                my $one = $arrayref->[0];
```

与数组不同，数组引用可以嵌套：

```
my @array = ( (1, 0), (0, 1) )  # 一个包含四个元素的数组：(1, 0, 0, 1)
my @matrix = ( [1, 0], [0, 1] ) # 一个包含两个数组引用的数组
my $matrix = [ [0, 1], [1, 0] ] # 一个数组引用的数组引用
# 标量、数组和哈希之间没有命名空间冲突
# 因此此时@matrix和$matrix都存在且保存不同的值。

my @diagonal_1 = ($matrix[0]->[1], $matrix[1]->[0])     # 使用@matrix
my @diagonal_2 = ($matrix->[0]->[1], $matrix->[1]->[0]) # 使用$matrix
# 由于链式的[]和{}访问只能作用于引用，你可以
# 省略部分箭头符号。
my $corner_1 = $matrix[0][1];   # 使用@matrix;
my $corner_2 = $matrix->[0][1]; # 使用$matrix;
```

作为布尔值使用时，引用总是为真。

第3.3节：标量引用

引用（reference）是一个标量变量（前面带有$的变量），它"指向"某个其他数据。

```
my $value     = "Hello";
my $reference = \$value;
print $value;     # => Hello
print $reference; # => SCALAR(0x2683310)
```

要获取被引用的数据，你需要对其进行解引用（de-reference）。

```
say ${$reference};                  # 显式前缀语法
say $$reference;                    # 大括号可以省略（可能会引起混淆）
```
版本 ≥ 5.24.0

新的后缀解引用语法，v5.24起默认可用

```
use v5.24;
say $reference->$*; # 新的后缀表示法
```

这个"解引用的值"随后可以像原始变量一样被修改。

```
${$reference} =~ s/Hello/World/;
print ${$reference};  # => World
print $value;         # => World
```

```
my @contents = @{ $array_reference };                # Prefix notation
my @contents = @$array_reference;                    # Braces can be left out
```
Version ≥ 5.24.0

New postfix dereference syntax, available by default from v5.24

```
use v5.24;
my @contents = $array_reference->@*; # New postfix notation
```

When accessing an arrayref's contents by index you can use the `->` syntactical sugar.

```
my @array = qw(one two three);      my $arrayref = [ qw(one two three) ]
my $one = $array[0];                my $one = $arrayref->[0];
```

Unlike arrays, arrayrefs can be nested:

```
my @array = ( (1, 0), (0, 1) )  # ONE array of FOUR elements: (1, 0, 0, 1)
my @matrix = ( [1, 0], [0, 1] ) # an array of two arrayrefs
my $matrix = [ [0, 1], [1, 0] ] # an arrayref of arrayrefs
# There is no namespace conflict between scalars, arrays and hashes
# so @matrix and $matrix _both_ exist at this point and hold different values.

my @diagonal_1 = ($matrix[0]->[1], $matrix[1]->[0])     # uses @matrix
my @diagonal_2 = ($matrix->[0]->[1], $matrix->[1]->[0]) # uses $matrix
# Since chained []- and {}-access can only happen on references, you can
# omit some of those arrows.
my $corner_1 = $matrix[0][1];   # uses @matrix;
my $corner_2 = $matrix->[0][1]; # uses $matrix;
```

When used as Boolean, references are always true.

Section 3.3: Scalar References

A **reference** is a scalar variable (one prefixed by $) which "refers to" some other data.

```
my $value     = "Hello";
my $reference = \$value;
print $value;     # => Hello
print $reference; # => SCALAR(0x2683310)
```

To get the referred-to data, you **de-reference** it.

```
say ${$reference};                  # Explicit prefix syntax
say $$reference;                    # The braces can be left out (confusing)
```
Version ≥ 5.24.0

New postfix dereference syntax, available by default from v5.24

```
use v5.24;
say $reference->$*; # New postfix notation
```

This "de-referenced value" can then be changed like it was the original variable.

```
${$reference} =~ s/Hello/World/;
print ${$reference};  # => World
print $value;         # => World
```

引用总是真值（truthy）——即使它所指向的值是假值（如0或""）。

如果你想要一个标量引用（Scalar Reference），可能是因为：

- 你想将一个字符串传递给函数，并让函数修改该字符串，而不通过返回值来实现。
- 您希望明确避免 Perl 在函数传递过程中隐式复制大字符串的内容（尤其是在不支持写时复制字符串的旧版 Perl 中相关）。
- 您希望区分具有特定含义的类字符串值与传递内容的字符串，例如：
  - 区分文件名和文件内容
  - 区分返回的内容和返回的错误字符串您希望实现一个轻量级的
- 内部对象模型，其中传递给调用代码的对象不携带用户可见的元数据：

```
our %objects;
my $next_id = 0;
sub new {
    my $object_id = $next_id++;
    $objects{ $object_id } = { ... }; # 为对象分配数据
    my $ref = \$object_id;
    return bless( $ref, "MyClass" );
}
```

第3.4节：数组

数组存储有序的值序列。您可以通过索引访问内容，或对其进行迭代。值将保持您填充时的顺序。

```
my @numbers_to_ten = (1,2,3,4,5,6,7,8,9,10); # 更方便的写法: (1..10)
my @chars_of_hello = ('h','e','l','l','o');
my @word_list = ('Hello','World');

# 注意符号：用 $array[index] 访问 @数组的元素
my $second_char_of_hello = $chars_of_hello[1]; # 'e'

# 使用负数索引从末尾计数（-1 表示最后一个）
my $last_char_of_hello = $chars_of_hello[-1];

# 将数组赋值给标量变量以获取数组长度
my $length_of_array = @chars_of_hello; # 5

# 你可以用 $# 获取数组的最后一个索引，但这会让 Stack Overflow 感到困惑
my $last_index_of_array = $#chars_of_hello; # 4

# 你也可以同时访问数组的多个元素
# 这称为"数组切片"
# 由于这会返回多个值，右侧应使用的符号是 @
my @some_chars_of_hello = @chars_of_hello[1..3]; # ('H', 'e', 'l')
my @out_of_order_chars = @chars_of_hello[1,4,2]; # ('e', 'o', 'l')

# 在 Python 中你可以用 array[1:-1] 来获取除第一个和最后一个之外的所有元素
# Perl 中不行：(1..-1) 是空列表。请使用 $# 代替
my @empty_list = @chars_of_hello[1..-1];                           # ()
my @inner_chars_of_hello = @chars_of_hello[1..$#chars_of_hello-1]; # ('e','l','l')
```

A reference is always **truthy** – even if the value it refers to is falsy (like `0` or `""`).

You may want a Scalar Reference If:

- You want to pass a string to a function, and have it modify that string for you without it being a return value.
- You wish to explicitly avoid Perl implicitly copying the contents of a large string at some point in your function passing (especially relevant on older Perls without copy-on-write strings)
- You wish to disambiguate string-like values with specific meaning, from strings that convey content, for example:
  - Disambiguate a file name from file content
  - Disambiguate returned content from a returned error string
- You wish to implement a lightweight inside out object model, where objects handed to calling code don't carry user visible metadata:

```
our %objects;
my $next_id = 0;
sub new {
    my $object_id = $next_id++;
    $objects{ $object_id } = { ... }; # Assign data for object
    my $ref = \$object_id;
    return bless( $ref, "MyClass" );
}
```

Section 3.4: Arrays

Arrays store an ordered sequence of values. You can access the contents by index, or iterate over them. The values will stay in the order you filled them in.

```
my @numbers_to_ten = (1,2,3,4,5,6,7,8,9,10); # More conveniently: (1..10)
my @chars_of_hello = ('h','e','l','l','o');
my @word_list = ('Hello','World');

# Note the sigil: access an @array item with $array[index]
my $second_char_of_hello = $chars_of_hello[1]; # 'e'

# Use negative indices to count from the end (with -1 being last)
my $last_char_of_hello = $chars_of_hello[-1];

# Assign an array to a scalar to get the length of the array
my $length_of_array = @chars_of_hello; # 5

# You can use $# to get the last index of an array, and confuse Stack Overflow
my $last_index_of_array = $#chars_of_hello; # 4

# You can also access multiple elements of an array at the same time
# This is called "array slice"
# Since this returns multiple values, the sigil to use here on the RHS is @
my @some_chars_of_hello = @chars_of_hello[1..3]; # ('H', 'e', 'l')
my @out_of_order_chars = @chars_of_hello[1,4,2]; # ('e', 'o', 'l')

# In Python you can say array[1:-1] to get all elements but first and last
# Not so in Perl: (1..-1) is an empty list. Use $# instead
my @empty_list = @chars_of_hello[1..-1];                           # ()
my @inner_chars_of_hello = @chars_of_hello[1..$#chars_of_hello-1]; # ('e','l','l')
```

```
# 访问数组末尾之外的元素会返回 undef，而不是错误
my $undef = $chars_of_hello[6]; # undef
```

数组是可变的：

```
use utf8; # 必须，因为这段代码是 utf-8 编码
$chars_of_hello[1] = 'u';              #       ('h','u','l','l','o')
push @chars_of_hello, ('!', '!');      #       ('h','u','l','l','o','!','!')
pop @chars_of_hello;                   #       ('h','u','l','l','o','!')
shift @chars_of_hello;                 #           ('u','l','l','o','!')
unshift @chars_of_hello, ('¡', 'H');   # ('¡','H','u','l','l','o','!')
@chars_of_hello[2..5] = ('O','L','A'); # ('¡','H','O','L','A',undef,'!') 哎呀！
delete $chars_of_hello[-2];            # ('¡','H','O','L','A',       '!')

# 在数组末尾之外设置元素不会导致错误
# 数组会根据需要用 undef 扩展。这称为“自动生存化（autovivification）”。
my @array;           # ()
my @array[3] = 'x';  # (undef, undef, undef, 'x')
```

最后，你可以遍历数组的内容：

```
use v5.10; # 必须用于 'say'
for my $number (@numbers_to_ten) {
  say $number ** 2;
}
```

当作为布尔值使用时，数组如果非空则为真。

第3.5节：类型符号（typeglobs）、类型符号引用、文件句柄和常量

类型符号 *foo 保存对同名全局变量内容的引用：$foo、@foo、$foo、&foo 等。你可以像访问哈希表一样访问它，并赋值以直接操作符号表（很危险！）。

```
use v5.10; # 必须用于 say
our $foo = "foo";
our $bar;
say ref *foo{SCALAR};      # SCALAR
say ${ *foo{SCALAR} };     # bar
*bar = *foo;
say $bar;                  # bar
$bar = 'egg';
say $foo;                  # egg
```

处理文件时，类型符号表（typeglobs）更为常见。例如，open 在被要求创建非全局文件句柄时，会生成对类型符号表的引用：

```
use v5.10; # 需要用于 say
open(my $log, '> utf-8', '/tmp/log') or die $!; # 以编码方式打开写入
say $log '日志已打开';

# 你可以对这个符号表引用进行解引用，但它用处不大。
say ref $log;                      # GLOB
say (*{$log}->{IO} // 'undef'); # undef

close $log or die $!;
```

```
# Access beyond the end of the array yields undef, not an error
my $undef = $chars_of_hello[6]; # undef
```

Arrays are mutable:

```
use utf8; # necessary because this snippet is utf-8
$chars_of_hello[1] = 'u';              #     ('h','u','l','l','o')
push @chars_of_hello, ('!', '!');      #     ('h','u','l','l','o','!','!')
pop @chars_of_hello;                   #     ('h','u','l','l','o','!')
shift @chars_of_hello;                 #         ('u','l','l','o','!')
unshift @chars_of_hello, ('¡', 'H');   # ('¡','H','u','l','l','o','!')
@chars_of_hello[2..5] = ('O','L','A'); # ('¡','H','O','L','A',undef,'!') whoops!
delete $chars_of_hello[-2];            # ('¡','H','O','L','A',      '!')

# Setting elements beyond the end of an array does not result in an error
# The array is extended with undef's as necessary. This is "autovivification."
my @array;           # ()
my @array[3] = 'x';  # (undef, undef, undef, 'x')
```

Finally, you can loop over the contents of an array:

```
use v5.10; # necessary for 'say'
for my $number (@numbers_to_ten) {
  say $number ** 2;
}
```

When used as booleans, arrays are true if they are not empty.

Section 3.5: Typeglobs, typeglob refs, filehandles and constants

A typeglob *foo holds references to the contents of *global* variables with that name: $foo, @foo, $foo, &foo, etc. You can access it like an hash and assign to manipulate the symbol tables directly (evil!).

```
use v5.10; # necessary for say
our $foo = "foo";
our $bar;
say ref *foo{SCALAR};      # SCALAR
say ${ *foo{SCALAR} };     # bar
*bar = *foo;
say $bar;                  # bar
$bar = 'egg';
say $foo;                  # egg
```

Typeglobs are more commonly handled when dealing with files. open, for example, produces a reference to a typeglob when asked to create a non-global filehandle:

```
use v5.10; # necessary for say
open(my $log, '> utf-8', '/tmp/log') or die $!; # open for writing with encoding
say $log 'Log opened';

# You can dereference this globref, but it's not very useful.
say ref $log;                      # GLOB
say (*{$log}->{IO} // 'undef'); # undef

close $log or die $!;
```

类型符号表也可以用来创建全局只读变量，尽管 use constant 更为广泛使用。

```
# 创建全局常量
*TRUE = \('1');
our $TRUE;
say $TRUE;  # 1
$TRUE = ''; # 终止，"尝试修改只读值"

# 使用常量代替定义一个无参数函数，因此它不是全局的，
# 可以在没有符号的情况下使用，可以被导入，但不易插值。
use constant (FALSE => 0);
say FALSE;        # 0
say &FALSE;       # 0
say "${\FALSE}";  # 0 (ugh)
say *FALSE{CODE}; # CODE(0xMA1DBABE)

# 当然，当你可以操作符号表时，这两者都不是真正的常量……
*TRUE = \('');
use constant (EVIL => 1);
*FALSE = *EVIL;
```

第3.6节：符号前缀（Sigils）

Perl 有许多符号前缀：

```
$scalar = 1; # 单个值
@array = ( 1, 2, 3, 4, 5 ); # 值的序列
%hash = ('it', 'ciao', 'en', 'hello', 'fr', 'salut'); # 无序的键值对
&function('arguments'); # 子程序
*typeglob; # 符号表条目
```

这些看起来像符号标记，但其实不是：

```
\@array; # | 返回右侧内容的引用（所以，是 @array 的引用）
$#array; # 这是 @array 最后一个元素的索引
```

如果你愿意，可以在符号标记后使用大括号。有时，这样做可以提高可读性。

```
say ${value} = 5;
```

虽然你用不同的符号标记来定义不同类型的变量，但同一个变量可以根据你使用的符号标记以不同的方式访问。

```
%hash;            # 我们用 % 是因为我们查看的是整个哈希表
$hash{it};        # 但如果我们只想要单个值，因为它是单数，所以用 $
$array[0];        # 数组也是如此。注意括号的变化。
@array[0,3];      # 我们想要数组的多个值，所以用 @
@hash{'it','en'}; # 哈希表同理（这会得到值：'ciao', 'hello'）
%hash{'it','fr'}; # 我们只想要哈希表中的部分键，所以用 %
                  # （这会得到键值对：'it', 'ciao', 'fr', 'salut'）
```

这对于引用尤其适用。为了使用引用的值，你可以将符号组合在一起。

```
my @array = 1..5;                    # 这是一个数组
my $reference_to_an_array = \@array; # 数组的引用是一个单一值
push @array, 6;                      # push 期望的是一个数组
push @$reference_to_an_array, 7;     # @ 符号表示右边的是一个数组
                                     # 而右边的是 $reference_to_an_array
```

Typeglobs can also be used to make global read-only variables, though **use** constant is in broader use.

```
# Global constant creation
*TRUE = \('1');
our $TRUE;
say $TRUE;  # 1
$TRUE = ''; # dies, "Modification of a read-only value attempted"

# use constant instead defines a parameterless function, therefore it's not global,
# can be used without sigils, can be imported, but does not interpolate easily.
use constant (FALSE => 0);
say FALSE;        # 0
say &FALSE;       # 0
say "${\FALSE}";  # 0 (ugh)
say *FALSE{CODE}; # CODE(0xMA1DBABE)

# Of course, neither is truly constant when you can manipulate the symbol table...
*TRUE = \('');
use constant (EVIL => 1);
*FALSE = *EVIL;
```

Section 3.6: Sigils

Perl has a number of sigils:

```
$scalar = 1; # individual value
@array = ( 1, 2, 3, 4, 5 ); # sequence of values
%hash = ('it', 'ciao', 'en', 'hello', 'fr', 'salut'); # unordered key-value pairs
&function('arguments'); # subroutine
*typeglob; # symbol table entry
```

These look like sigils, but aren't:

```
\@array; # \ returns the reference of what's on the right (so, a reference to @array)
$#array; # this is the index of the last element of @array
```

You can use braces after the sigil if you should be so inclined. Occasionally, this improves readability.

```
say ${value} = 5;
```

While you use different sigils to define variables of different types, the same variable can be accessed in different ways based on what sigils you use.

```
%hash;            # we use % because we are looking at an entire hash
$hash{it};        # we want a single value, however, that's singular, so we use $
$array[0];        # likewise for an array. notice the change in brackets.
@array[0,3];      # we want multiple values of an array, so we instead use @
@hash{'it','en'}; # similarly for hashes (this gives the values: 'ciao', 'hello')
%hash{'it','fr'}; # we want an hash with just some of the keys, so we use %
                  # (this gives key-value pairs: 'it', 'ciao', 'fr', 'salut')
```

This is especially true of references. In order to use a referenced value you can combine sigils together.

```
my @array = 1..5;                    # This is an array
my $reference_to_an_array = \@array; # A reference to an array is a singular value
push @array, 6;                      # push expects an array
push @$reference_to_an_array, 7;     # the @ sigil means what's on the right is an array
                                     # and what's on the right is $reference_to_an_array
# 因此：先是 @，然后是 $
```

这里有一种可能不那么令人困惑的思考方式。正如我们之前看到的，你可以用大括号包裹符号右边的内容。所以你可以把 @{} 看作是接受一个数组引用并返回被引用的数组。

```
# pop 不喜欢数组引用
pop $reference_to_an_array; # Perl 5.20+ 中的错误
# 但是如果我们使用 @{}，那么...
pop @{ $reference_to_an_array }; # 这样就可以了！
```

事实证明，@{} 实际上接受一个表达式：

```
my $values = undef;
say pop @{ $values };        # 错误：不能将未定义值用作数组引用
say pop @{ $values // [5] } # undef // [5] 返回 [5]，所以这行打印 5
```

...同样的技巧也适用于其他符号。

```
# 这不是良好 Perl 代码的示例，仅仅是对该语言特性的演示
my $hashref = undef;
for my $key ( %{ $hashref // {} } ) {
  "这不会崩溃";
}
```

...但如果符号的“参数”很简单，可以省略大括号。

```
say $$scalar_reference;
say pop @$array_reference;
for keys (%$hash_reference) { ... };
```

事情可能变得非常复杂。这是可行的，但请负责任地使用 Perl。

```
my %hash = (it => 'ciao', en => 'hi', fr => 'salut');
my $reference = \%hash;
my $reference_to_a_reference = \$reference;

my $italian = $hash{it};                              # 直接访问
my @greets = @$reference{'it', 'en'};                 # 解引用后作为数组访问
my %subhash = %$$reference_to_a_reference{'en', 'fr'} # 解引用两次后作为哈希访问
```

对于大多数常规使用，您可以直接使用子程序名称而不加符号。（没有符号的变量通常称为“裸词”。）&符号仅在有限的情况下有用。

- 引用子程序：

  ```
  sub many_bars { 'bar' x $_[0] }
  my $reference = \&many_bars;
  say $reference->(3); # barbarbar
  ```

- 调用函数时忽略其原型。
- 结合goto，作为一种稍显奇怪的函数调用，当前调用帧被调用者替换。
  类似于linux的 exec() API调用，但用于函数。

```
# hence: first a @, then a $
```

Here's a perhaps less confusing way to think about it. As we saw earlier, you can use braces to wrap what's on the right of a sigil. So you can think of @{} as something that takes an array reference and gives you the referenced array.

```
# pop does not like array references
pop $reference_to_an_array; # ERROR in Perl 5.20+
# but if we use @{}, then...
pop @{ $reference_to_an_array }; # this works!
```

As it turns out, @{} actually accepts an expression:

```
my $values = undef;
say pop @{ $values };       # ERROR: can't use undef as an array reference
say pop @{ $values // [5] } # undef // [5] gives [5], so this prints 5
```

...and the same trick works for other sigils, too.

```
# This is not an example of good Perl. It is merely a demonstration of this language feature
my $hashref = undef;
for my $key ( %{ $hashref // {} } ) {
  "This doesn't crash";
}
```

...but if the "argument" to a sigil is simple, you can leave the braces away.

```
say $$scalar_reference;
say pop @$array_reference;
for keys (%$hash_reference) { ... };
```

Things can get excessively extravagant. This works, but please Perl responsibly.

```
my %hash = (it => 'ciao', en => 'hi', fr => 'salut');
my $reference = \%hash;
my $reference_to_a_reference = \$reference;

my $italian = $hash{it};                              # Direct access
my @greets = @$reference{'it', 'en'};                 # Dereference, then access as array
my %subhash = %$$reference_to_a_reference{'en', 'fr'} # Dereference ×2 then access as hash
```

For most normal use, you can just use subroutine names without a sigil. (Variables without a sigil are typically called "barewords".) The & sigil is only useful in a limited number of cases.

- Making a reference to a subroutine:

  ```
  sub many_bars { 'bar' x $_[0] }
  my $reference = \&many_bars;
  say $reference->(3); # barbarbar
  ```

- Calling a function ignoring its prototype.
- Combined with goto, as a slightly weird function call that has the current call frame replaced with the caller. Think the linux exec() API call, but for functions.

第3.7节：哈希引用

哈希引用是包含指向哈希数据内存位置指针的标量。因为标量直接指向哈希本身，当它被传递给子程序时，对哈希所做的更改不像普通哈希那样局限于子程序内部，而是全局生效。

首先，让我们看看当你将一个普通哈希传递给子程序并在其中修改时会发生什么：

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my %hash = @_;

    $hash{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper(\%hash);

    return;
}

my %example_hash = (
    old_value   => 1,
);

modify(%example_hash);

print Dumper("After exiting the subroutine");
print Dumper(\%example_hash);
```

Which results in:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'old_value' => 1
        };
```

注意，当我们退出子程序后，哈希保持不变；对它的所有更改都是局部于 modify 子程序的，因为我们传递的是哈希的副本，而不是哈希本身。

相比之下，当你传递一个哈希引用时，你传递的是原始哈希的地址，因此在子程序内所做的任何更改都会作用于原始哈希：

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my $hashref = shift;
```

Section 3.7: Hash References

Hash references are scalars which contain a pointer to the memory location containing the data of a hash. Because the scalar points directly to the hash itself, when it is passed to a subroutine, changes made to the hash are not local to the subroutine as with a regular hash, but instead are global.

First, let's examine what happens when you pass a normal hash to a subroutine and modify it within there:

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my %hash = @_;

    $hash{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper(\%hash);

    return;
}

my %example_hash = (
    old_value   => 1,
);

modify(%example_hash);

print Dumper("After exiting the subroutine");
print Dumper(\%example_hash);
```

Which results in:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'old_value' => 1
        };
```

Notice that after we exit the subroutine, the hash remains unaltered; all changes to it were local to the modify subroutine, because we passed a copy of the hash, not the hash itself.

In comparison, when you pass a hashref, you are passing the address to the original hash, so any changes made within the subroutine will be made to the original hash:

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my $hashref = shift;
    # 解引用哈希以添加新值
    $hashref->{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper($hashref);

    return;
}

# 创建一个哈希引用
my $example_ref = {
    old_value   => 1,
};

# 将哈希引用传递给子程序
modify($example_ref);

print Dumper("退出子程序后");
print Dumper($example_ref);
```

这将导致：

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
```

第3.8节：哈希

哈希可以理解为查找表。你可以通过为每个哈希指定一个键来访问其内容。键
必须是字符串。如果不是，它们将被转换为字符串。

如果你给哈希一个已知的键，它将返回对应的值。

```
# 元素按 (键, 值, 键, 值) 顺序排列
my %inhabitants_of = ("伦敦", 8674000, "巴黎", 2244000);

# 你可以使用"胖逗号"
# 语法糖来减少输入并提高清晰度。它的行为类似逗号，并对左边的内容加引号。
my %translations_of_hello = (spanish => 'Hola', german => 'Hallo', swedish => 'Hej');
```

在下面的例子中，注意括号和符号：你使用 $hash{key} 来访问 %hash 的元素，因为你想要的
*值*是一个标量。有些人认为给键加引号是良好习惯，而另一些人则觉得这种风格视觉上较为杂乱。
只有当键可能被误认为类似 $hash{'some-key'} 这样的表达式时，才需要加引号

```
my $greeting = $translations_of_hello{'spanish'};
```

虽然 Perl 默认会尝试将裸词作为字符串使用，+ 修饰符也可以用来告诉 Perl 键
不应被插值，而是执行，执行结果作为键使用：

```
my %employee = ( name => '约翰·多伊', shift => '夜班' );
# 这个例子将打印 '夜班'
```

```
    # De-reference the hash to add a new value
    $hashref->{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper($hashref);

    return;
}

# Create a hashref
my $example_ref = {
    old_value   => 1,
};

# Pass a hashref to a subroutine
modify($example_ref);

print Dumper("After exiting the subroutine");
print Dumper($example_ref);
```

This will result in:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
```

Section 3.8: Hashes

Hashes can be understood as lookup-tables. You can access its contents by specifiying a key for each of them. Keys must be strings. If they're not, they will be converted to strings.

If you give the hash simply a known key, it will serve you its value.

```
# Elements are in (key, value, key, value) sequence
my %inhabitants_of = ("London", 8674000, "Paris", 2244000);

# You can save some typing and gain in clarity by using the "fat comma"
# syntactical sugar. It behaves like a comma and quotes what's on the left.
my %translations_of_hello = (spanish => 'Hola', german => 'Hallo', swedish => 'Hej');
```

In the following example, note the brackets and sigil: you access an element of %hash using $hash{key} because the *value* you want is a scalar. Some consider it good practice to quote the key while others find this style visually noisy. Quoting is only required for keys that could be mistaken for expressions like $hash{'some-key'}

```
my $greeting = $translations_of_hello{'spanish'};
```

While Perl by default will try to use barewords as strings, + modifier can also be used to indicate to Perl that key should not be interpolated but executed with result of execution being used as a key:

```
my %employee = ( name => 'John Doe', shift => 'night' );
# this example will print 'night'
```

```
print $employee{shift};

# 但这个将执行 [shift][1]，从 @_ 中提取第一个元素，
# 并将结果作为键使用
print $employee{+shift};
```

和数组一样，你可以同时访问多个哈希元素。这称为哈希切片。结果值是一个列表，所以使用@符号：

```
my @words = @translations_of_hello{'spanish', 'german'}; # ('Hola', 'Hallo')
```

遍历哈希的键时，keys keys 会以随机顺序返回项目。如果需要，可以结合 sort 使用。

```
for my $lang (sort keys %translations_of_hello) {
  say $translations_of_hello{$lang};
}
```

如果你实际上不需要像前面例子中那样使用键，values 会直接返回哈希的值：

```
for my $translation (values %translations_of_hello) {
  say $translation;
}
```

你也可以使用带有 each 的 while 循环来遍历哈希。这样，你可以同时获得键和值，而无需单独查找值。不过不推荐使用，因为 each 可能会以令人困惑的方式中断。

```
# 不推荐使用
while (my ($lang, $translation) = each %translations_of_hello) {
  say $translation;
}
```

访问未设置的元素会返回 undef，而不是错误：

```
my $italian = $translations_of_hello{'italian'}; # undef
```

map 和列表扁平化可以用来从数组创建哈希。这是一种常用的方法来创建值的“集合”，例如快速检查一个值是否在@elems 中。这个操作通常需要 O(n) 时间（即与元素数量成正比），但通过将列表转换成哈希，可以在常数时间（O(1)）内完成：

```
@elems = qw(x y x z t);
my %set = map { $_ => 1 } @elems;   # (x, 1, y, 1, t, 1)
my $y_membership = $set{'y'};       # 1
my $w_membership = $set{'w'};       # undef
```

这需要一些解释。数组 @elems 的内容被读入一个列表，由 map 处理。 map 接受一个代码块，该代码块会对输入列表的每个值调用；元素的值可通过 $_ 使用。我们的代码块为每个输入元素返回两个列表元素：$_，即输入元素，和 1，仅仅是某个值。考虑到列表扁平化，结果是 map { $_ => 1 } @elems 将 qw(x y x z t) 转换为 (x => 1, y => 1, x => 1, z => 1, t => 1)。

当这些元素被赋值到哈希时，奇数位置的元素成为哈希键，偶数位置的元素成为哈希值。当一个键在赋值给哈希的列表中出现多次时，最后的值生效。这实际上会丢弃重复项。

```
print $employee{shift};

# but this one will execute [shift][1], extracting first element from @_,
# and use result as a key
print $employee{+shift};
```

Like with arrays, you can access multiple hash elements at the same time. This is called a *hash slice*. The resulting value is a list, so use the @ sigil:

```
my @words = @translations_of_hello{'spanish', 'german'}; # ('Hola', 'Hallo')
```

Iterate over the keys of an hash with keys keys will return items in a random order. Combine with sort if you wish.

```
for my $lang (sort keys %translations_of_hello) {
  say $translations_of_hello{$lang};
}
```

If you do not actually need the keys like in the previous example, values returns the hash's values directly:

```
for my $translation (values %translations_of_hello) {
  say $translation;
}
```

You can also use a while loop with each to iterate over the hash. This way, you will get both the key and the value at the same time, without a separate value lookup. Its use is however discouraged, as each can break in mistifying ways.

```
# DISCOURAGED
while (my ($lang, $translation) = each %translations_of_hello) {
  say $translation;
}
```

Access to unset elements returns undef, not an error:

```
my $italian = $translations_of_hello{'italian'}; # undef
```

map and list flattening can be used to create hashes out of arrays. This is a popular way to create a 'set' of values, e.g. to quickly check whether a value is in @elems. This operation usually takes O(n) time (i.e. proportional to the number of elements) but can be done in constant time (O(1)) by turning the list into a hash:

```
@elems = qw(x y x z t);
my %set = map { $_ => 1 } @elems;   # (x, 1, y, 1, t, 1)
my $y_membership = $set{'y'};       # 1
my $w_membership = $set{'w'};       # undef
```

This requires some explanation. The contents of @elems get read into a list, which is processed by map. map accepts a code block that gets called for each value of its input list; the value of the element is available for use in $_. Our code block returns *two* list elements for each input element: $_, the input element, and 1, just some value. Once you account for list flattening, the outcome is that map { $_ => 1 } @elems turns qw(x y x z t) into (x => 1, y => 1, x => 1, z => 1, t => 1).

As those elements get assigned into the hash, odd elements become hash keys and even elements become hash values. When a key is specified multiple times in a list to be assigned to a hash, the *last* value wins. This effectively discards duplicates.

将列表转换为哈希的更快方法是使用哈希切片赋值。它使用 x 操作符将单元素列表 (1) 乘以 @elems 的大小，因此左侧切片中的每个键都有一个对应的 1 值：

```
@elems = qw(x y x z t);
my %set;
@set{@elems} = (1) x @elems;
```

下面对哈希的应用也利用了哈希和列表在实现命名函数参数时经常可以互换的事实：

```
sub hash_args {
  my %args = @_;
  my %defaults = (foo => 1, bar => 0);
  my %overrides = (__unsafe => 0);
  my %settings = (%defaults, %args, %overrides);
}

# 该函数可以这样调用：
hash_args(foo => 5, bar => 3); # (foo => 5, bar => 3, __unsafe ==> 0)
hash_args();                   # (foo => 1, bar => 0, __unsafe ==> 0)
hash_args(__unsafe => 1)       # (foo => 1, bar => 0, __unsafe ==> 0)
```

当作为布尔值使用时，哈希如果非空则为真。

A faster way to turn a list into a hash uses assignment to a hash slice. It uses the x operator to multiply the single-element list (1) by the size of @elems, so there is a 1 value for each of the keys in the slice on the left hand side:

```
@elems = qw(x y x z t);
my %set;
@set{@elems} = (1) x @elems;
```

The following application of hashes also exploits the fact that hashes and lists can often be used interchangeably to implement named function args:

```
sub hash_args {
  my %args = @_;
  my %defaults = (foo => 1, bar => 0);
  my %overrides = (__unsafe => 0);
  my %settings = (%defaults, %args, %overrides);
}

# This function can then be called like this:
hash_args(foo => 5, bar => 3); # (foo => 5, bar => 3, __unsafe ==> 0)
hash_args();                   # (foo => 1, bar => 0, __unsafe ==> 0)
hash_args(__unsafe => 1)       # (foo => 1, bar => 0, __unsafe ==> 0)
```

When used as booleans, hashes are true if they are not empty.

第4章：Perl中的插值

第4.1节：什么是插值

Perl会对变量名进行插值：

```
my $name = 'Paul';
print "Hello, $name!"; # Hello, Paul!

my @char = ('a', 'b', 'c');print "$
char[1]"; # bmy %map = (a

 => 125, b => 1080, c => 11);print "$map{a}"; #
125
```

数组可以整体插值，它们的元素之间用空格分隔：

```
my @char = ('a', 'b', 'c');print "M
y chars are @char"; # My chars are a b c
```

Perl 不会整体插值哈希：

```
my %map = (a => 125, b => 1080, c => 11);print
 "My map is %map"; # My map is %map
```

以及函数调用（包括常量）：

```
use constant {
    PI => '3.1415926'
};
print "I like PI";          # I like PIprint "I like
" . PI . ""; # I like 3.1415926
```

Perl 插值以 \\ 开头的转义序列：

```
                    水平制表符
                    换行符
\r                       返回
\f                    换页符
\b                    退格符
\a                       警报(铃声)
\e                    转义符
```

对

的插值取决于程序运行的系统：它将根据当前系统的约定生成换行符字符。

Perl 不对\v进行插值，\v在 C 语言及其他语言中表示垂直制表符。

字符可以使用它们的代码来表示：

```
\x{1d11e}     通过十六进制代码表示的符号
\o{350436}    通过八进制代码表示的符号
\N{U+1d11e}   通过 Unicode 码点表示的符号
```

或者通过 Unicode 名称表示：

Chapter 4: Interpolation in Perl

Section 4.1: What is interpolated

Perl interpolates variable names:

```
my $name = 'Paul';
print "Hello, $name!\n"; # Hello, Paul!

my @char = ('a', 'b', 'c');
print "$char[1]\n"; # b

my %map = (a => 125, b => 1080, c => 11);
print "$map{a}\n"; # 125
```

Arrays may be interpolated as a whole, their elements are separated by spaces:

```
my @char = ('a', 'b', 'c');
print "My chars are @char\n"; # My chars are a b c
```

Perl does *not* interpolate hashes as a whole:

```
my %map = (a => 125, b => 1080, c => 11);
print "My map is %map\n"; # My map is %map
```

and function calls (including constants):

```
use constant {
    PI => '3.1415926'
};
print "I like PI\n";          # I like PI
print "I like " . PI . "\n"; # I like 3.1415926
```

Perl interpolates *escape sequences* starting with `\`:

```
\t                  horizontal tab
\n                  newline
\r                  return
\f                  form feed
\b                  backspace
\a                  alarm (bell)
\e                  escape
```

Interpolation of `\n` depends on the system where program is working: it will produce a newline character(s) according to the current system conventions.

Perl does *not* interpolate `\v`, which means vertical tab in C and other languages.

Character may be addressed using their codes:

```
\x{1d11e}     ???? by hexadecimal code
\o{350436}    ???? by octal code
\N{U+1d11e}   ???? by Unicode code point
```

or Unicode names:

```
\N{MUSICAL SYMBOL G CLEF}
```

在本地编码中，代码从0x00到0xFF的字符可以用更短的形式表示：

```
\x0a     十六进制表示
\012     八进制表示
```

控制字符可以使用特殊的转义序列来处理：

```
\c@      chr(0)
\ca      chr(1)
\cb      chr(2)
...
\cz      chr(26)
\c[      chr(27)
\c\      chr(28) # 不能用于字符串末尾
                 # 因为反斜杠会插入终止引号
|c]      chr(29)
\c^      chr(30)
\c_      chr(31)
\c?      chr(127)
```

大写字母具有相同含义："\cA" == "\ca"。

除 \N{...} 之外，所有转义序列的解释可能依赖于平台，因为它们使用平台和编码相关的代码。

第4.2节：基本插值

插值意味着 Perl 解释器会将变量的值替换变量名，并将一些符号（这些符号直接输入是不可能或困难的）替换为特殊字符序列（这也称为转义）。最重要的区别在于单引号和双引号：双引号会对包含的字符串进行插值，而单引号则不会。

```
my $name = 'Paul';
my $age = 64;
print "我的名字是 $name。我今年 $age。"; # 我的名字是 Paul。
                                        # 我今年 64 岁。
```

但是：

```
print '我的名字是 $name。我今年 $age。'; # 我的名字是 $name。我今年 $age。
```

你可以使用 q{}（任意分隔符）代替单引号，使用 qq{}代替双引号。例如，q{我今年 64 岁} 允许在非插值字符串中使用撇号（否则会终止字符串）。

语句：

```
print qq{$name 说: "我今年 $age".}; # Paul 说: "我今年 64 岁"。
print "$name 说: \"我今年 $age\"."  # Paul 说: "我今年 64 岁"。
```

做同样的事情，但在第一个中你不需要对字符串中的双引号进行转义。

如果你的变量名与周围文本冲突，可以使用语法 ${var} 来消除歧义：

```
my $decade = 80;
```

```
\N{MUSICAL SYMBOL G CLEF}
```

Character with codes from 0x00 to 0xFF in the *native* encoding may be addressed in a shorter form:

```
\x0a     hexadecimal
\012     octal
```

Control character may be addressed using special escape sequences:

```
\c@      chr(0)
\ca      chr(1)
\cb      chr(2)
...
\cz      chr(26)
\c[      chr(27)
\c\      chr(28) # Cannot be used at the end of a string
                 # since backslash will interpolate the terminating quote
\c]      chr(29)
\c^      chr(30)
\c_      chr(31)
\c?      chr(127)
```

Uppercase letters have the same meaning: "\cA" == "\ca".

Interpretation of all escape sequences except for \N{...} may depend on the platform since they use platform- and encoding-dependent codes.

Section 4.2: Basic interpolation

Interpolation means that Perl interpreter will substitute the values of variables for their name and some symbols (which are impossible or difficult to type in directly) for special sequences of characters (it is also known as escaping). The most important distinction is between single and double quotes: double quotes interpolate the enclosed string, but single quotes do not.

```
my $name = 'Paul';
my $age = 64;
print "My name is $name.\nI am $age.\n"; # My name is Paul.
                                         # I am 64.
```

But:

```
print 'My name is $name.\nI am $age.\n'; # My name is $name.\nI am $age.\n
```

You can use q{} (with any delimiter) instead of single quotes and qq{} instead of double quotes. For example, q{I'm 64} allows to use an apostrophe within a non-interpolated string (otherwise it would terminate the string).

Statements:

```
print qq{$name said: "I'm $age".}; # Paul said: "I'm 64".
print "$name said: \"I'm $age\"."  # Paul said: "I'm 64".
```

do the same thing, but in the first one you do not need to escape double quotes within the string.

If your variable name clashes with surrounding text, you can use the syntax ${var} to disambiguate:

```
my $decade = 80;
```

```
print "我喜欢${decade}s的音乐！"  # 我喜欢80年代的音乐！
```

```
print "I like ${decade}s music!"  # I like 80s music!
```

第5章：真与假

第5.1节：真值和假值列表

```
use feature qw( say );

# 数字如果不等于0则为真。
say 0             ? '真' : '假'; # 假
say 1             ? '真' : '假'; # 真
say 2             ? '真' : '假'; # 真
say -1            ? '真' : '假'; # 真
say 1-1           ? '真' : '假'; # 假
say 0e7           ? '真' : '假'; # 假
say -0.00         ? '真' : '假'; # 假

# 字符串如果非空则为真。
say 'a'           ? '真' : '假'; # 真
say 'false'       ? '真' : '假'; # 真
say ''            ? '真' : '假'; # 假

# 即使字符串在数值上下文中被视为0，只要非空就为真。
# 唯一的例外是字符串 "0"，它是 false。
# 要强制数值上下文，可以给字符串加上 0
say '0'           ? '真' : '假'; # 假
say '0.0'         ? '真' : '假'; # 真
say '0e0'         ? '真' : '假'; # 真
say '0 but true'  ? '真' : '假'; # 真
say '0 whargarbl' ? '真' : '假'; # 真
say 0+'0 argarbl' ? '真' : '假'; # 假

# 在标量上下文中变成数字的东西被当作数字处理。
my @c = ();
my @d = (0);
say @c            ? '真' : '假'; # 假
say @d            ? '真' : '假'; # 真

# 任何未定义的值都是假。
say undef         ? '真' : '假'; # 假

# 引用总是真，即使它们指向的是假值
my @c = ();
my $d = 0;
say \@c            ? '真' : '假'; # 真
say \$d            ? '真' : '假'; # 真
say \0             ? '真' : '假'; # 真
say \''            ? '真' : '假'; # 真
```

Chapter 5: True and false

Section 5.1: List of true and false values

```
use feature qw( say );

# Numbers are true if they're not equal to 0.
say 0             ? 'true' : 'false'; # false
say 1             ? 'true' : 'false'; # true
say 2             ? 'true' : 'false'; # true
say -1            ? 'true' : 'false'; # true
say 1-1           ? 'true' : 'false'; # false
say 0e7           ? 'true' : 'false'; # false
say -0.00         ? 'true' : 'false'; # false

# Strings are true if they're not empty.
say 'a'           ? 'true' : 'false'; # true
say 'false'       ? 'true' : 'false'; # true
say ''            ? 'true' : 'false'; # false

# Even if a string would be treated as 0 in numeric context, it's true if nonempty.
# The only exception is the string "0", which is false.
# To force numeric context add 0 to the string
say '0'           ? 'true' : 'false'; # false
say '0.0'         ? 'true' : 'false'; # true
say '0e0'         ? 'true' : 'false'; # true
say '0 but true'  ? 'true' : 'false'; # true
say '0 whargarbl' ? 'true' : 'false'; # true
say 0+'0 argarbl' ? 'true' : 'false'; # false

# Things that become numbers in scalar context are treated as numbers.
my @c = ();
my @d = (0);
say @c            ? 'true' : 'false'; # false
say @d            ? 'true' : 'false'; # true

# Anything undefined is false.
say undef         ? 'true' : 'false'; # false

# References are always true, even if they point at something false
my @c = ();
my $d = 0;
say \@c            ? 'true' : 'false'; # true
say \$d            ? 'true' : 'false'; # true
say \0             ? 'true' : 'false'; # true
say \''            ? 'true' : 'false'; # true
```

第6章：日期和时间

第6.1节：日期格式

Time::Piece 在 perl 5 版本 10 之后可用

```
use Time::Piece;

my $date = localtime->strftime('%m/%d/%Y');
print $date;
```

```
输出
07/26/2016
```

第 6.2 节：创建新的 DateTime

在您的电脑上安装DateTime，然后在 perl 脚本中使用：

`use DateTime;`

创建新的当前日期时间

`$dt = DateTime->now( time_zone => 'Asia/Ho_Chi_Minh');`

然后你可以访问日期和时间元素的值：

```
$year   = $dt->year;
$month  = $dt->month;
$day    = $dt->day;
$hour   = $dt->hour;
$minute = $dt->minute;
$second = $dt->second;
```

只获取时间：

`my $time = $dt->hms; #返回格式为 hh:mm:ss的时间`

只获取日期：

`my $date = $dt->ymd; #返回格式为 yyyy-mm-dd的日期`

第6.3节：操作日期时间元素

设置单个元素：

`$dt->set( year => 2016 );`

设置多个元素：

`$dt->set( year => 2016, 'month' => 8);`

向日期时间添加持续时间

`$dt->add( hour => 1, month => 2)`

日期时间相减：

Chapter 6: Dates and Time

Section 6.1: Date formatting

Time::Piece is available in perl 5 after version 10

```
use Time::Piece;

my $date = localtime->strftime('%m/%d/%Y');
print $date;
```

```
Output
07/26/2016
```

Section 6.2: Create new DateTime

Install `DateTime` on your PC and then use it in perl script:

`use DateTime;`

Create new current datetime

`$dt = DateTime->now( time_zone => 'Asia/Ho_Chi_Minh');`

Then you can access elements's values of date and time:

```
$year   = $dt->year;
$month  = $dt->month;
$day    = $dt->day;
$hour   = $dt->hour;
$minute = $dt->minute;
$second = $dt->second;
```

To get only time:

`my $time = $dt->hms; #return time with format hh:mm:ss`

To get only date:

`my $date = $dt->ymd; #return date with format yyyy-mm-dd`

Section 6.3: Working with elements of datetime

Set single element:

`$dt->set( year => 2016 );`

Set many elements:

`$dt->set( year => 2016, 'month' => 8);`

Add duration to datetime

`$dt->add( hour => 1, month => 2)`

Datetime subtraction:

```
my $dt1 = DateTime->new(
      year      => 2016,
      month     => 8,
      day       => 20,
);

my $dt2 = DateTime->new(
      year      => 2016,
      month     => 8,
      day       => 24,
);

my $duration = $dt2->subtract_datetime($dt1);
print $duration->days
```

你将得到结果是4天

第6.4节：计算代码执行时间

```
use Time::HiRes qw( time );

my $start = time();

#计算执行时间的代码
sleep(1.2);

my $end = time();

printf("执行时间: %0.02f 秒", $end - $start);
```

这将打印代码的执行时间（秒）

```
my $dt1 = DateTime->new(
      year      => 2016,
      month     => 8,
      day       => 20,
);

my $dt2 = DateTime->new(
      year      => 2016,
      month     => 8,
      day       => 24,
);

my $duration = $dt2->subtract_datetime($dt1);
print $duration->days
```

You will get the result is 4 days

Section 6.4: Calculate code execution time

```
use Time::HiRes qw( time );

my $start = time();

#Code for which execution time is calculated
sleep(1.2);

my $end = time();

printf("Execution Time: %0.02f s\n", $end - $start);
```

This will print execution time of Code in seconds

第7章：控制语句

第7.1节：条件语句

Perl支持多种条件语句（基于布尔结果的语句）。最常见的条件语句是if-else、unless和三元语句。given语句作为类似C语言派生语言中的switch结构被引入，并且在Perl 5.10及以上版本中可用。

if-else语句

if语句的基本结构如下：

```
if (表达式) 代码块
if (表达式) 代码块 else 代码块
if (表达式) 代码块 elsif (表达式) 代码块 ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

对于简单的 if 语句，if 可以放在要执行的代码之前或之后。

```
$number = 7;
if ($number > 4) { print "$number 大于四！"; }

# 也可以这样写
print "$number 大于四！" if $number > 4;
```

第7.2节：循环

Perl 支持多种循环结构：for/foreach、while/do-while 和 until。

```
@numbers = 1..42;
for (my $i=0; $i <= $#numbers; $i++) {print "
    $numbers[$i]";
}

# 也可以写成
foreach my $num (@numbers) {p
    rint "$num";
}
```

while 循环在执行相关代码块之前先判断条件。因此，有时代码块可能根本不会被执行。例如，如果文件句柄 $fh 是一个空文件的文件句柄，或者在条件判断之前已经读取完毕，下面的代码块将永远不会被执行。

```
while (my $line = readline $fh) {
    say $line;
}
```

另一方面，do/while 和 do/until 循环是在每次执行代码块之后才判断条件。因此，do/while 或 do/until 循环至少会执行一次。

```
my $greeting_count = 0;
do {
    say "Hello";
    $greeting_count++;
} until ( $greeting_count > 1)
```

Chapter 7: Control Statements

Section 7.1: Conditionals

Perl supports many kinds of conditional statements (statements that are based on boolean results). The most common conditional statements are if-else, unless, and ternary statements. `given` statements are introduced as a switch-like construct from C-derived languages and are available in versions Perl 5.10 and above.

If-Else Statements

The basic structure of an if-statement is like this:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

For simple if-statements, the if can precede or succeed the code to be executed.

```
$number = 7;
if ($number > 4) { print "$number is greater than four!"; }

# Can also be written this way
print "$number is greater than four!" if $number > 4;
```

Section 7.2: Loops

Perl supports many kinds of loop constructs: for/foreach, while/do-while, and until.

```
@numbers = 1..42;
for (my $i=0; $i <= $#numbers; $i++) {
    print "$numbers[$i]\n";
}

#Can also be written as
foreach my $num (@numbers) {
    print "$num\n";
}
```

The while loop evaluates the conditional *before* executing the associated block. So, sometimes the block is never executed. For example, the following code would never be executed if the filehandle `$fh` was the filehandle for an empty file, or if was already exhausted before the conditional.

```
while (my $line = readline $fh) {
    say $line;
}
```

The do/`while` and do/`until` loops, on the other hand, evaluate the conditional *after* each time the block is executed. So, a do/`while` or a do/`until` loop is always executed at least once.

```
my $greeting_count = 0;
do {
    say "Hello";
    $greeting_count++;
} until ( $greeting_count > 1)
```

```
# Hello
# Hello
```

```
# Hello
# Hello
```

第8章：子程序

第8.1节：创建子程序

子程序通过使用关键字 sub，后跟标识符和用大括号括起来的代码块来创建。

可以使用特殊变量@_访问参数，该变量包含所有参数的数组。

```
sub function_name {
    my ($arg1, $arg2, @more_args) = @_;
    # ...
}
```

由于函数 shift在子程序内部默认对@_进行移位，因此在子程序开始时按顺序将参数提取到局部变量中是一种常见的模式：

```
sub function_name {
    my $arg1 = shift;
    my $arg2 = shift;
    my @more_args = @_;
    # ...
}

# 模拟命名参数（而非位置参数）
sub function_name {
    my %args = (arg1 => 'default', @_);
    my $arg1 = delete $args{arg1};
    my $arg2 = delete $args{arg2};
    # ...
}

sub {
    my $arg1 = shift;
    # ...
}->($arg);
```

版本 ≥ 5.20.0

或者，可以使用实验性功能"signatures"来解包参数，这些参数是按值传递的（*不是*按引用传递）。

```
use feature "signatures";

sub function_name($arg1, $arg2, @more_args) {
    # ...
}
```

参数可以使用默认值。

```
use feature "signatures";

sub function_name($arg1=1, $arg2=2) {
    # ...
}
```

你可以使用任何表达式为参数赋默认值–包括其他参数。

```
sub function_name($arg1=1, $arg2=$arg1+1) {
```

Chapter 8: Subroutines

Section 8.1: Creating subroutines

Subroutines are created by using the keyword **sub** followed by an identifier and a code block enclosed in braces.

You can access the arguments by using the special variable `@_`, which contains all arguments as an array.

```
sub function_name {
    my ($arg1, $arg2, @more_args) = @_;
    # ...
}
```

Since the function `shift` defaults to shifting `@_` when used inside a subroutine, it's a common pattern to extract the arguments sequentially into local variables at the beginning of a subroutine:

```
sub function_name {
    my $arg1 = shift;
    my $arg2 = shift;
    my @more_args = @_;
    # ...
}

# emulate named parameters (instead of positional)
sub function_name {
    my %args = (arg1 => 'default', @_);
    my $arg1 = delete $args{arg1};
    my $arg2 = delete $args{arg2};
    # ...
}

sub {
    my $arg1 = shift;
    # ...
}->($arg);
```

Version ≥ 5.20.0

Alternatively, the experimental feature "signatures" can be used to unpack parameters, which are passed by value (*not* by reference).

```
use feature "signatures";

sub function_name($arg1, $arg2, @more_args) {
    # ...
}
```

Default values can be used for the parameters.

```
use feature "signatures";

sub function_name($arg1=1, $arg2=2) {
    # ...
}
```

You can use any expression to give a default value to a parameter – including other parameters.

```
sub function_name($arg1=1, $arg2=$arg1+1) {
    # ...
}
```

请注意，您不能引用在当前参数之后定义的参数–因此以下代码
的运行结果不会完全符合预期。

```
sub function_name($arg1=$arg2, $arg2=1) {
    print $arg1;  # => <nothing>
    print $arg2;  # => 1
}
```

第8.2节：子程序

子程序包含代码。除非另有说明，否则它们是全局定义的。

```
# 函数不必指定它们的参数列表
sub returns_one {
  # 函数默认返回最后一个表达式的值
  # 这里的return关键字不是必须的，但有助于提高可读性。
  return 1;
}

# 其参数可通过@_访问
sub sum {
  my $ret = 0;
  for my $value (@_) {
    $ret += $value
  }
  return $ret;
}

# Perl 尽力使参数列表的括号可选
say sum 1..3;      # 6

# 如果将函数视为变量，& 符号是必须的。
say defined &sum; # 1
```

一些内置函数如 print 或 say 是关键字，不是函数，所以例如 &say 是未定义的。这也意味着你可以定义它们，但调用时必须指定包名

```
# 这在默认包 'main' 下定义函数
sub say {
  # 这是 say 关键字的用法
  say "我说，@_" ;
}

# ...所以你可以这样调用它：
main::say('wow'); # 我说，哇。
```

版本 ≥ 5.18.0

自 Perl 5.18 起，你也可以使用非全局函数：

```
use feature 'lexical_subs';
my $value;
{
  # 糟糕的代码来了
  my sub prod {
    my $ret = 1;
    $ret *= $_ for @_;
```

```
    # ...
}
```

Note that you can't reference parameters which are defined after the current parameter – hence the following code doesn't work quite as expected.

```
sub function_name($arg1=$arg2, $arg2=1) {
    print $arg1;  # => <nothing>
    print $arg2;  # => 1
}
```

Section 8.2: Subroutines

Subroutines hold code. Unless specified otherwise, they are globally defined.

```
# Functions do not (have to) specify their argument list
sub returns_one {
  # Functions return the value of the last expression by default
  # The return keyword here is unnecessary, but helps readability.
  return 1;
}

# Its arguments are available in @_, however
sub sum {
  my $ret = 0;
  for my $value (@_) {
    $ret += $value
  }
  return $ret;
}

# Perl makes an effort to make parens around argument list optional
say sum 1..3;     # 6

# If you treat functions as variables, the & sigil is mandatory.
say defined &sum; # 1
```

Some builtins such as `print` or `say` are keywords, not functions, so e.g. `&say` is undefined. It also does mean that you can define them, but you will have to specify the package name to actually call them

```
# This defines the function under the default package, 'main'
sub say {
  # This is instead the say keyword
  say "I say, @_";
}

# ...so you can call it like this:
main::say('wow'); # I say, wow.
```

Version ≥ 5.18.0

Since Perl 5.18, you can also have non-global functions:

```
use feature 'lexical_subs';
my $value;
{
  # Nasty code ahead
  my sub prod {
    my $ret = 1;
    $ret *= $_ for @_;
    $ret;
  }
  $value = prod 1..6; # 720
  say defined &prod; # 1
}
say defined &prod; # 0
```
版本 ≥ 5.20.0

自5.20版本起，你也可以使用命名参数。

```
使用 feature 'signatures';
sub greet($name) {
  说 "Hello, $name";
}
```

这不应与原型混淆，原型是Perl提供的一种功能，允许你定义行为类似内置函数的函数。
函数原型必须在编译时可见，其效果可以通过指定&符号来忽略。
原型通常被认为是高级功能，最好谨慎使用。

```
# 这个原型使得用非数组调用此函数成为编译错误。
# 此外，数组会自动转换为数组引用
sub receives_arrayrefs(\@\@) {
    my $x = shift;
    my $y = shift;
}

my @a = (1..3);
my @b = (1..4);
receives_arrayrefs(@a, @b);     # 正确, $x = |@a, $y = |@b, @_ = ();
receives_arrayrefs(\@a, \@b);  # 编译错误, "类型 ... 必须是数组 ..."
BEGIN { receives_arrayrefs(\@a, \@b); }

# 指定符号以忽略原型。
&receives_arrayrefs(\@a, \@b); # 正确, $x = |@a, $y = |@b, @_ = ();
&receives_arrayrefs(@a, @b);    # 正确, 但 $x = 1, $y = 2, @_ = (3,1,2,3,4);
```

第8.3节：子程序参数通过引用传递（签名中的除外）

Perl中的子程序参数是通过引用传递的，除非它们在签名中。这意味着子程序内部的@_数组成员只是实际参数的别名。在下面的例子中，主程序中的$text在子程序调用后被修改，因为子程序内部的$_[0]实际上只是同一变量的另一个名字。第二次调用会抛出错误，因为字符串字面量不是变量，因此不能被修改。

```
use feature 'say';

sub edit {
    $_[0] =~ s/world/sub/;
}

my $text = "Hello, world!";
edit($text);
say $text;      # Hello, sub!

edit("Hello, world!"); # 错误：尝试修改只读值
```

为了避免覆盖调用者的变量，因此将@_复制到局部作用域变量（my ...）中非常重要，正如"创建子程序"中所述。

```
    $ret;
  }
  $value = prod 1..6; # 720
  say defined &prod; # 1
}
say defined &prod; # 0
```
Version ≥ 5.20.0

Since 5.20, you can also have named parameters.

```
use feature 'signatures';
sub greet($name) {
  say "Hello, $name";
}
```

This should *not* be confused with prototypes, a facility Perl has to let you define functions that behave like built-ins. Function prototypes must be visible at compile time and its effects can be ignored by specifying the & sigil. Prototypes are generally considered to be an advanced feature that is best used with great care.

```
# This prototype makes it a compilation error to call this function with anything
# that isn't an array. Additionally, arrays are automatically turned into arrayrefs
sub receives_arrayrefs(\@\@) {
   my $x = shift;
   my $y = shift;
}

my @a = (1..3);
my @b = (1..4);
receives_arrayrefs(@a, @b);    # okay,   $x = \@a, $y = \@b, @_ = ();
receives_arrayrefs(\@a, \@b);  # compilation error, "Type … must be array …"
BEGIN { receives_arrayrefs(\@a, \@b); }

# Specify the sigil to ignore the prototypes.
&receives_arrayrefs(\@a, \@b); # okay,   $x = \@a, $y = \@b, @_ = ();
&receives_arrayrefs(@a, @b);   # ok, but $x = 1,   $y = 2,   @_ = (3,1,2,3,4);
```

Section 8.3: Subroutine arguments are passed by reference (except those in signatures)

Subroutine arguments in Perl are passed by reference, unless they are in the signature. This means that the members of the @_ array inside the sub are just *aliases* to the actual arguments. In the following example, $text in the main program is left modified after the subroutine call because $_[0] inside the sub is actually just a different name for the same variable. The second invocation throws an error because a string literal is not a variable and therefore can't be modified.

```
use feature 'say';

sub edit {
    $_[0] =~ s/world/sub/;
}

my $text = "Hello, world!";
edit($text);
say $text;      # Hello, sub!

edit("Hello, world!"); # Error: Modification of a read-only value attempted
```

To avoid clobbering your caller's variables it is therefore important to copy @_ to locally scoped variables (my ...) as

正如“创建子程序”中所述。

described under "Creating subroutines".

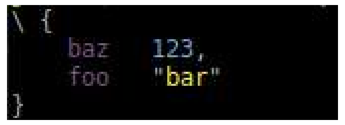
第9章：调试输出

第9.1节：风格化转储

有时Data::Dumper不够用。想检查一个Moose对象？大量相同结构？想要排序？彩色显示？Data::Printer是你的好帮手。

```
use Data::Printer;

p $data_structure;
```



```
\ {
    baz   123,
    foo   "bar"
}
```

Data::Printer像warn一样写入标准错误输出（STDERR）。这使得查找输出更容易。默认情况下，它会对哈希键进行排序并查看对象。

```
use Data::Printer;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
p $ua;
```

它会查看对象的所有方法，并列出内部结构。

```
LWP::UserAgent  {
父类         LWP::MemberMixin
公共方法 (45) : add_handler, agent, clone, conn_cache, cookie_jar, credentials,
default_header, default_headers, delete, env_proxy, from, get, get_basic_credentials,
get_my_handler, handlers, head, is_online, is_protocol_supported, local_address, max_redirect,
max_size, mirror, new, no_proxy, parse_head, post, prepare_request, progress, protocols_allowed,
protocols_forbidden, proxy, put, redirect_ok, remove_handler, request, requests_redirectable,
run_handlers, send_request, set_my_handler, show_progress, simple_request, ssl_opts, timeout,
use_alarm, use_eval
私有方法 (4) : _agent, _need_proxy, _new_response, _process_colonic_headers
    内部变量: {
def_headers            HTTP::Headers,
        handlers                {
response_header    HTTP::Config
        },
local_address           undef,
        max_redirect            7,
        max_size                undef,
        no_proxy                [],
        protocols_allowed       undef,
        protocols_forbidden     undef,
        proxy                   {},
        requests_redirectable   [
            [0] "GET",
            [1] "HEAD"
        ],
show_progress           undef,
        ssl_opts                {
verify_hostname    1
        },
timeout             180,
        use_eval                1
```

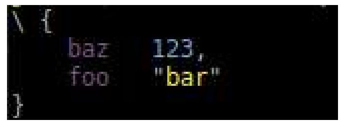
Chapter 9: Debug Output

Section 9.1: Dumping with Style

Sometimes Data::Dumper is not enough. Got a Moose object you want to inspect? Huge numbers of the same structure? Want stuff sorted? Colored? Data::Printer is your friend.

```
use Data::Printer;

p $data_structure;
```



```
\ {
    baz   123,
    foo   "bar"
}
```

Data::Printer writes to STDERR, like `warn`. That makes it easier to find the output. By default, it sorts hash keys and looks at objects.

```
use Data::Printer;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
p $ua;
```

It will look at all the methods of the object, and also list the internals.

```
LWP::UserAgent  {
    Parents       LWP::MemberMixin
    public methods (45) : add_handler, agent, clone, conn_cache, cookie_jar, credentials,
default_header, default_headers, delete, env_proxy, from, get, get_basic_credentials,
get_my_handler, handlers, head, is_online, is_protocol_supported, local_address, max_redirect,
max_size, mirror, new, no_proxy, parse_head, post, prepare_request, progress, protocols_allowed,
protocols_forbidden, proxy, put, redirect_ok, remove_handler, request, requests_redirectable,
run_handlers, send_request, set_my_handler, show_progress, simple_request, ssl_opts, timeout,
use_alarm, use_eval
    private methods (4) : _agent, _need_proxy, _new_response, _process_colonic_headers
    internals: {
        def_headers             HTTP::Headers,
        handlers                {
            response_header    HTTP::Config
        },
        local_address           undef,
        max_redirect            7,
        max_size                undef,
        no_proxy                [],
        protocols_allowed       undef,
        protocols_forbidden     undef,
        proxy                   {},
        requests_redirectable   [
            [0] "GET",
            [1] "HEAD"
        ],
        show_progress           undef,
        ssl_opts                {
            verify_hostname    1
        },
        timeout                 180,
        use_eval                1
    }
}
```

您可以进一步配置它，使其以特定方式序列化某些对象，或包含任意深度的对象。完整配置请参见文档。

不幸的是，Data::Printer 并未随 Perl 一起发布，因此您需要从 CPAN 或通过您的包管理系统安装它。

第9.2节：转储数据结构

```
use Data::Dumper;

my $data_structure = { foo => 'bar' };
print Dumper $data_structure;
```

使用 Data::Dumper 是在运行时查看数据结构或变量内容的简便方法。它随 Perl 一起发布，您可以轻松加载。 Dumper 函数返回以类似 Perl 代码的方式序列化的数据结构。

```
$VAR1 = {
        'foo' => 'bar',
}
```

这使得快速查看代码中的某些值非常有用。它是你工具库中最方便的工具之一。请阅读关于metacpan的完整文档。

第9.3节：Data::Show

当执行use Data::Show; 时，函数show会被自动导出。该函数以一个变量作为唯一参数，并输出：

1. 变量名
2. 该变量的内容（以可读格式）
3. 3. 运行show所在文件的行号
4. 运行show所在的文件

假设以下代码来自文件 example.pl：

```
use strict;
use warnings;
use Data::Show;

my @array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

my %hash  = ( foo => 1, bar => { baz => 10, qux => 20 } );

my $href = \%hash;

show @array;
show %hash;
show $href;
```

perl example.pl 输出如下：

```
======(  @array  )=======================[ 'example.pl', 第11行 ]======
```

```
    }
}
```

You can configure it further, so it serializes certain objects in a certain way, or to include objects up to an arbitrary depth. The full configuration is available in the documentation.

Unfortunately Data::Printer does not ship with Perl, so you need to install it from CPAN or through your package management system.

Section 9.2: Dumping data-structures

```
use Data::Dumper;

my $data_structure = { foo => 'bar' };
print Dumper $data_structure;
```

Using Data::Dumper is an easy way to look at data structures or variable content at run time. It ships with Perl and you can load it easily. The `Dumper` function returns the data structure serialized in a way that looks like Perl code.

```
$VAR1 = {
        'foo' => 'bar',
}
```

That makes it very useful to quickly look at some values in your code. It's one of the most handy tools you have in your arsenal. Read the full documentation on metacpan.

Section 9.3: Data::Show

The function `show` is automatically exported when `use Data::Show;` is executed. This function takes a variable as its sole argument and it outputs:

1. the name of the variable
2. the contents of that variable (in a readable format)
3. the line of the file that `show` is run from
4. the file `show` is run from

Assuming that the following is code from the file `example.pl`:

```
use strict;
use warnings;
use Data::Show;

my @array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

my %hash  = ( foo => 1, bar => { baz => 10, qux => 20 } );

my $href = \%hash;

show @array;
show %hash;
show $href;
```

`perl example.pl` gives the following output:

```
======(  @array  )=======================[ 'example.pl', line 11 ]======
    [1 .. 10]


======(  %hash  )=======================[ 'example.pl', 第12行 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }


======(  $href  )=======================[ 'example.pl', 第13行 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }
```

请参阅 Data::Show 的文档。

第9.4节：转储数组列表

```
my @data_array = (123, 456, 789, 'poi', 'uyt', "rew", "qas");
print Dumper @data_array;
```

使用 Data::Dumper 可以轻松获取列表值。Dumper 会以一种看起来像 Perl 代码的方式序列化列表值。

输出：

```
$VAR1 = 123;
$VAR2 = 456;
$VAR3 = 789;
$VAR4 = 'poi';
$VAR5 = 'uyt';
$VAR6 = 'rew';
$VAR7 = 'qas';
```

正如用户 @dgw 所建议的，当转储数组或哈希时，最好使用数组引用或哈希引用，这样显示会更符合输入。

```
$ref_data = [23,45,67,'mnb','vcx'];
print Dumper $ref_data;
```

输出：

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

你也可以在打印时引用数组。

```
my @data_array = (23,45,67,'mnb','vcx');
print Dumper \@data_array;
```

输出：

```
$VAR1 = [
          23,
```

```
    [1 .. 10]


======(  %hash  )=======================[ 'example.pl', line 12 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }


======(  $href  )=======================[ 'example.pl', line 13 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }
```

See the documentation for Data::Show.

Section 9.4: Dumping array list

```
my @data_array = (123, 456, 789, 'poi', 'uyt', "rew", "qas");
print Dumper @data_array;
```

Using Data::Dumper gives an easy access to fetch list values. The Dumper returns the list values serialized in a way that looks like Perl code.

Output:

```
$VAR1 = 123;
$VAR2 = 456;
$VAR3 = 789;
$VAR4 = 'poi';
$VAR5 = 'uyt';
$VAR6 = 'rew';
$VAR7 = 'qas';
```

As suggested by user @dgw When dumping arrays or hashes it is better to use an array reference or a hash reference, those will be shown better fitting to the input.

```
$ref_data = [23,45,67,'mnb','vcx'];
print Dumper $ref_data;
```

Output:

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

You can also reference the array when printing.

```
my @data_array = (23,45,67,'mnb','vcx');
print Dumper \@data_array;
```

Output:

```
$VAR1 = [
          23,
        45,
        67,
        'mnb',
        'vcx'
    ];
```

```
        45,
        67,
        'mnb',
        'vcx'
    ];
```

第10章：列表

第10.1节：数组作为列表

数组是Perl的基本变量类型之一。它包含一个列表，即零个或多个标量的有序序列。数组是保存（并提供访问）列表数据的变量，如perldata中所述。

你可以将一个列表赋值给数组：

```
my @foo = ( 4, 5, 6 );
```

你可以在期望列表的任何地方使用数组：

```
join '-', ( 4, 5, 6 );
join '-', @foo;
```

有些运算符只适用于数组，因为它们会改变数组所包含的列表：

```
shift @array;
unshift @array, ( 1, 2, 3 );
pop @array;
push @array, ( 7, 8, 9 );
```

第10.2节：将列表赋值给哈希

列表也可以赋值给哈希变量。当创建将赋值给哈希变量的列表时，建议在键和值之间使用fat comma=>来表示它们的关系：

```
my %hash = ( foo => 42, bar => 43, baz => 44 );
```

=>实际上只是一个特殊的逗号，会自动为其左侧的操作数加引号。所以，你可以使用普通逗号，但关系就不那么清晰了：

```
my %hash = ( 'foo', 42, 'bar', 43, 'baz', 44 );
```

你也可以为fat comma=>的左操作数使用带引号的字符串，这对于包含空格的键特别有用。

```
my %hash = ( 'foo bar' => 42, 'baz qux' => 43 );
```

详情请参见 Comma operator 于 perldoc perlop。

第10.3节：列表可以传递给子程序

要将列表传递给子程序，您需要指定子程序的名称，然后向其提供列表：

```
test_subroutine( 'item1', 'item2' );
test_subroutine  'item1', 'item2';     # 相同
```

Perl 内部为这些参数创建了别名，并将它们放入数组@_，该数组在子程序内可用：

```
@_ =  ( 'item1', 'item2' ); # 由 perl 内部完成
```

Chapter 10: Lists

Section 10.1: Array as list

The array is one of Perl's basic variable types. It contains a list, which is an ordered sequence of zero or more scalars. The array is the variable holding (and providing access to) the list data, as is documented in perldata.

You can assign a list to an array:

```
my @foo = ( 4, 5, 6 );
```

You can use an array wherever a list is expected:

```
join '-', ( 4, 5, 6 );
join '-', @foo;
```

Some operators only work with arrays since they mutate the list an array contains:

```
shift @array;
unshift @array, ( 1, 2, 3 );
pop @array;
push @array, ( 7, 8, 9 );
```

Section 10.2: Assigning a list to a hash

Lists can also be assigned to hash variables. When creating a list that will be assigned to a hash variable, it is recommended to use the ***fat comma*** => between keys and values to show their relationship:

```
my %hash = ( foo => 42, bar => 43, baz => 44 );
```

The => is really only a special comma that automatically quotes the operand to its left. So, you *could* use normal commas, but the relationship is not as clear:

```
my %hash = ( 'foo', 42, 'bar', 43, 'baz', 44 );
```

You can also use quoted strings for the left hand operand of the fat comma =>, which is especially useful for keys containing spaces.

```
my %hash = ( 'foo bar' => 42, 'baz qux' => 43 );
```

For details see Comma operator at `perldoc perlop`.

Section 10.3: Lists can be passed into subroutines

As to pass list into a subroutine, you specify the subroutine's name and then supply the list to it:

```
test_subroutine( 'item1', 'item2' );
test_subroutine  'item1', 'item2';     # same
```

Internally Perl makes *aliases* to those arguments and put them into the array @_ which is available within the subroutine:

```
@_ =  ( 'item1', 'item2' ); # Done internally by perl
```

您可以这样访问子程序参数：

```
sub test_subroutine {
    print $_[0]; # item1
    print $_[1]; # item2
}
```

别名使您能够更改传递给子程序的参数的原始值：

```
sub test_subroutine {
    $_[0] +=  2;
}

my $x =  7;
test_subroutine( $x );
print $x; # 9
```

为了防止对子程序中传入的原始值进行无意的修改，应该先复制它们：

```
sub test_subroutine {
    my( $copy_arg1, $copy_arg2 ) =  @_;
    $copy_arg1 += 2;
}

my $x =  7;
test_subroutine $x; # 在这种情况下 $copy_arg2 的值将是 `undef`
print $x; # 7
```

要测试传入子程序的参数个数，可以检查 @_ 的大小

```
sub test_subroutine {
    print scalar @_, ' 个参数传入子程序';
}
```

如果将数组参数传入子程序，它们都会被展平：

```
my @x =  ( 1, 2, 3 );
my @y =  qw/ a b c /; # ( 'a', 'b', 'c' )
test_some_subroutine @x, 'hi', @y; # 7 个参数传入子程序
# @_ =  ( 1, 2, 3, 'hi', 'a', 'b', 'c' ) # 这是该调用内部完成的
```

如果你的 test_some_subroutine 包含语句 $_[4] = 'd'，对于上述调用，它将导致 $y[0] 之后的值为 d：

```
print "@y"; # d b c
```

第10.4节：从子程序返回列表

当然，你可以从子程序返回列表：

```
sub foo {
    my @list1 =  ( 1, 2, 3 );
    my @list2 =  ( 4, 5 );

    return    ( @list1, @list2 );
}
```

You access subroutine arguments like this:

```
sub test_subroutine {
    print $_[0]; # item1
    print $_[1]; # item2
}
```

Aliasing gives you the ability to change the original value of argument passed to subroutine:

```
sub test_subroutine {
    $_[0] +=  2;
}

my $x =  7;
test_subroutine( $x );
print $x; # 9
```

To prevent inadvertent changes of original values passed into your subroutine, you should copy them:

```
sub test_subroutine {
    my( $copy_arg1, $copy_arg2 ) =  @_;
    $copy_arg1 += 2;
}

my $x =  7;
test_subroutine $x; # in this case $copy_arg2 will have `undef` value
print $x; # 7
```

To test how many arguments were passed into the subroutine, check the size of `@_`

```
sub test_subroutine {
    print scalar @_, ' argument(s) passed into subroutine';
}
```

If you pass array arguments into a subroutine they all will be *flattened*:

```
my @x =  ( 1, 2, 3 );
my @y =  qw/ a b c /; # ( 'a', 'b', 'c' )
test_some_subroutine @x, 'hi', @y; # 7 argument(s) passed into subroutine
# @_ =  ( 1, 2, 3, 'hi', 'a', 'b', 'c' ) # Done internally for this call
```

If your `test_some_subroutine` contains the statement `$_[4] = 'd'`, for the above call it will cause `$y[0]` to have value `d` afterwards:

```
print "@y"; # d b c
```

Section 10.4: Return list from subroutine

You can, of course, return lists from subs:

```
sub foo {
    my @list1 =  ( 1, 2, 3 );
    my @list2 =  ( 4, 5 );

    return    ( @list1, @list2 );
}
```

```
my @list =  foo();
print @list;          # 12345
```

但除非你知道自己在做什么，否则这不是推荐的做法。

当结果处于 LIST 上下文时，这样做是可以的，但在 SCALAR 上下文中情况就不明确了。让我们看看下一行：

```
print scalar foo();  # 2
```

为什么是2？发生了什么？

1. 因为foo()在SCALAR上下文中求值，这个列表(@list1,@list2)也在SCALAR上下文中求值
2. 在SCALAR上下文中，LIST返回它的最后一个元素。这里是@list2
3. 同样在SCALAR上下文中，数组@list2返回它的元素数量。这里是2。

在大多数情况下，正确的策略是返回对数据结构的引用。
所以在我们的例子中，应该改为如下操作：

```
 return    (\@list1,\@list2);
```

然后调用者会这样做来接收两个返回的arrayrefs：

```
 my ($list1, $list2) = foo(...);
```

第10.5节：作为列表的哈希

在列表上下文中，哈希会被展平。

```
my @bar =  ( %hash, %hash );
```

数组 @bar 由两个 %hash 哈希列表初始化

- 两个 %hash 都被展平
- 新列表由展平的元素创建
- @bar 数组由该列表初始化

保证键值对是成对出现的。键总是偶数索引，值是奇数索引。但不保证
键值对总是以相同顺序展平：

```
my %hash =  ( a => 1, b => 2 );
print %hash; # 可能是 'a1b2' 或 'b2a1'
```

第10.6节：使用数组引用传递数组给子程序

数组引用 @foo 是 \@foo。如果你需要将数组和其他内容传递给子程序，这非常方便。传递
@foo 就像传递多个标量。但传递 \@foo 是单个标量。在子程序内部：

```
xyz(\@foo, 123);
...
sub xyz {
    my ($arr, $etc) = @_;
    print $arr->[0]; # 使用 $arr 中的第一个元素。它就像 $foo[0]
```

```
my @list =  foo();
print @list;          # 12345
```

But it is not the recommended way to do that unless you know what you are doing.

While this is OK when the result is in **LIST** context, in **SCALAR** context things are unclear. Let's take a look at the next line:

```
print scalar foo();  # 2
```

Why 2? What is going on?

1. Because `foo()` evaluated in *SCALAR* context, this list `( @list1, @list2 )` also evaluated in *SCALAR* context
2. In *SCALAR* context, LIST returns its last element. Here it is `@list2`
3. Again in *SCALAR* context, **array** `@list2` returns the number of its elements. Here it is 2.

In most cases the **right strategy will return references to data structures**.
So in our case we should do the following instead:

```
 return    ( \@list1, \@list2 );
```

Then the caller does something like this to receive the two returned *arrayrefs*:

```
 my ($list1, $list2) = foo(...);
```

Section 10.5: Hash as list

In list context hash is flattened.

```
my @bar =  ( %hash, %hash );
```

The *array* `@bar` is initialized by list of two `%hash` hashes

- both `%hash` are flattened
- new list is created from flattened items
- `@bar` array is initialized by that list

It is guaranteed that key-value pairs goes together. Keys are always even indexed, values - odd. It is not guaranteed that key-value pairs are always flattened in same order:

```
my %hash =  ( a => 1, b => 2 );
print %hash; # Maybe 'a1b2' or 'b2a1'
```

Section 10.6: Using arrayref to pass array to sub

The arrayref for `@foo` is `\@foo`. This is handy if you need to pass an array and other things to a subroutine. Passing `@foo` is like passing multiple scalars. But passing `\@foo` is a single scalar. Inside the subroutine:

```
xyz(\@foo, 123);
...
sub xyz {
    my ($arr, $etc) = @_;
    print $arr->[0]; # using the first item in $arr. It is like $foo[0]
```

第11章：排序

对于排序列表，Perl 只有一个函数，不出所料叫做 sort。它足够灵活，可以排序各种项目：数字、任意编码的字符串、嵌套数据结构或对象。然而，由于其灵活性，使用时有许多技巧和惯用法需要学习。

第11.1节：基本词汇排序

```perl
@sorted = sort @list;

@sorted = sort { $a cmp $b } @list;

sub compare { $a cmp $b }
@sorted = sort compare @list;
```

上面三个例子做的事情完全相同。如果你没有提供任何比较函数或代码块，sort会假设你想对其右侧的列表进行字典序排序。如果你只是需要数据按某种可预测的顺序排列且不在意语言上的正确性，这通常是你想要的形式。

sort 会将 @list 中的元素成对传递给比较函数，该函数告诉 sort 哪个元素更大。对于字符串，cmp操作符完成此功能，而对于数字，则使用 <=>。比较函数被调用的频率很高，平均调用次数为 n * log(n)，其中 n 是待排序元素的数量，因此速度很重要。这也是 sort 使用预定义的包全局变量（$a 和 $b）将要比较的元素传递给代码块或函数，而不是使用正式的函数参数的原因。

如果你 use locale，cmp 会考虑特定语言环境的排序规则，例如在丹麦语环境下它会将 Å 排序得像 A 一样，但在英语或德语环境下则排在 Z 之后。然而，它不考虑更复杂的 Unicode 排序规则，也不提供对排序顺序的控制——例如电话簿的排序方式通常与词典不同。对于这些情况，推荐使用 Unicode::Collate，特别是 Unicode::Collate::Locale 模块。

第11.2节：Schwartzian变换

这可能是利用Perl函数式编程特性进行排序优化的最著名示例，适用于排序顺序依赖于昂贵函数的情况。

```perl
# 你通常会这样做
@sorted = sort { slow($a) <=> slow($b) } @list;

# 你为了加快速度会这样做
@sorted =
map { $_->[0] }
sort { $a->[1] <=> $b->[1] }
map { [ $_, slow($_) ] }
@list;
```

第一个例子的问题在于比较器被频繁调用，并且不断使用一个缓慢的函数重复计算值。一个典型的例子是按文件大小对文件名进行排序：

```perl
use File::stat;
@sorted = sort { stat($a)->size <=> stat($b)->size } glob "*";
```

这可以工作，但最多会在每次比较时产生两次系统调用的开销，最糟糕的情况是每次比较都必须两次访问磁盘，而该磁盘可能位于地球另一端的过载文件服务器上。

Chapter 11: Sorting

For sorting lists of things, Perl has only a single function, unsurprisingly called sort. It is flexible enough to sort all kinds of items: numbers, strings in any number of encodings, nested data structures or objects. However, due to its flexibility, there are quite a few tricks and idioms to be learned for its use.

Section 11.1: Basic Lexical Sort

```perl
@sorted = sort @list;

@sorted = sort { $a cmp $b } @list;

sub compare { $a cmp $b }
@sorted = sort compare @list;
```

The three examples above do exactly the same thing. If you don't supply any comparator function or block, sort assumes you want the list on its right sorted lexically. This is usually the form you want if you just need your data in some predictable order and don't care about linguistic correctness.

sort passes pairs of items in @list to the comparator function, which tells sort which item is larger. The cmp operator does this for strings while <=> does the same thing for numbers. The comparator is called quite often, on average *n* * log(*n*) times with *n* being the number of elements to be sorted, so it's important it be fast. This is the reason sort uses predefined package global variables ($a and $b) to pass the elements to be compared to the block or function, instead of proper function parameters.

If you **use** locale, cmp takes locale specific collation order into account, e.g. it will sort Å like A under a Danish locale but after Z under an English or German one. However, it doesn't take the more complex Unicode sorting rules into account nor does it offer any control over the order—for example phone books are often sorted differently from dictionaries. For those cases, the Unicode::Collate and particularly Unicode::Collate::Locale modules are recommended.

Section 11.2: The Schwartzian Transform

This is probably the most famous example of a sort optimization making use of Perl's functional programming facilities, to be used where the sort order of items depend on an expensive function.

```perl
# What you would usually do
@sorted = sort { slow($a) <=> slow($b) } @list;

# What you do to make it faster
@sorted =
map { $_->[0] }
sort { $a->[1] <=> $b->[1] }
map { [ $_, slow($_) ] }
@list;
```

The trouble with the first example is that the comparator is called very often and keeps recalculating values using a slow function over and over. A typical example would be sorting file names by their file size:

```perl
use File::stat;
@sorted = sort { stat($a)->size <=> stat($b)->size } glob "*";
```

This works, but at best it incurs the overhead of two system calls per comparison, at worst it has to go to the disk, twice, for every single comparison, and that disk may be in an overloaded file server on the other side of the planet.

介绍 Randall Schwartz 的技巧。

Schwartzian 变换基本上是将 @list 通过三个函数从下到上处理。第一个 map 将每个条目转换为一个包含原始项和作为排序键的慢速函数结果的两元素列表，因此到此为止，我们对每个元素只调用了一次 slow()。接下来的 sort 可以通过查看列表中的排序键来访问排序键。由于我们不关心排序键，只需要排序后的原始元素，最后的 map 会丢弃从 @sort 接收到的已排序的两元素列表，只返回它们的第一个成员组成的列表。

第11.3节：不区分大小写排序

使 sort 忽略大小写的传统方法是将字符串传递给 lc 或 uc 进行比较：

```
@sorted = sort { lc($a) cmp lc($b) } @list;
```

这在所有 Perl 5 版本中都有效，并且对于英语完全足够；无论使用 uc 还是 lc 都无关紧要。然而，对于希腊语或土耳其语等语言来说，这会带来问题，因为大写和小写字母之间没有一一对应关系，因此根据使用 uc 还是 lc 会得到不同的结果。
因此，Perl 5.16 及更高版本引入了一个称为 fc 的 case folding 函数，避免了这个问题，因此现代多语言排序应使用它：

```
@sorted = sort { fc($a) cmp fc($b) } @list;
```

第11.4节：数值排序

```
@sorted = sort { $a <=> $b } @list;
```

使用 <=> 运算符比较 $a 和 $b 确保它们按数值而非默认的文本方式进行比较。

第11.5节：反向排序

```
@sorted = sort { $b <=> $a } @list;
@sorted = reverse sort { $a <=> $b } @list;
```

通过在比较块中交换 $a 和 $b，可以简单地实现降序排序。
不过，有些人更喜欢使用单独的 reverse 来表达清晰，尽管它稍微慢一些。

Enter Randall Schwartz's trick.

The Schwartzian Transform basically shoves @list through three functions, bottom-to-top. The first map turns each entry into a two-element list of the original item and the result of the slow function as a sort key, so at the end of this we have called slow() exactly once for each element. The following sort can then simply access the sort key by looking in the list. As we don't care about the sort keys but only need the original elements in sorted order, the final map throws away the two-element lists from the already-sorted list it receives from @sort and returns a list of only their first members.

Section 11.3: Case Insensitive Sort

The traditional technique to make sort ignore case is to pass strings to lc or uc for comparison:

```
@sorted = sort { lc($a) cmp lc($b) } @list;
```

This works on all versions of Perl 5 and is completely sufficient for English; it doesn't matter whether you use uc or lc. However, it presents a problem for languages like Greek or Turkish where there is no 1:1 correspondence between upper- and lowercase letters so you get different results depending on whether you use uc or lc. Therefore, Perl 5.16 and higher have a *case folding* function called fc that avoids this problem, so modern multi-lingual sorting should use this:

```
@sorted = sort { fc($a) cmp fc($b) } @list;
```

Section 11.4: Numeric Sort

```
@sorted = sort { $a <=> $b } @list;
```

Comparing $a and $b with the <=> operator ensures they are compared numerically and not textually as per default.

Section 11.5: Reverse Sort

```
@sorted = sort { $b <=> $a } @list;
@sorted = reverse sort { $a <=> $b } @list;
```

Sorting items in descending order can simply be achieved by swapping $a and $b in the comparator block. However, some people prefer the clarity of a separate reverse even though it is slightly slower.

第12章：文件输入输出（读取和写入文件）

模式	解释
`>`	写入（截断）。将覆盖现有文件。如果未找到文件，则创建新文件
`>>`	写入（追加）。不会覆盖文件，而是在文件末尾追加新内容。如果用于打开不存在的文件，也会创建该文件
`<`	读取。以只读模式打开文件。
`+<`	读/写。不会创建或截断文件。
`+>`	读/写（截断）。将创建并截断文件。
`+>>`	读/写（追加）。将创建但不截断文件。

第12.1节：打开用于读取的文件句柄

打开通用ASCII文本文件

版本 ≥ 5.6.0

```
open my $filehandle, '<', $name_of_file or die "无法打开 $name_of_file, $!";
```

这是"默认"文件输入输出的基本用法，使$filehandle成为一个可读的字节输入流，通过默认的系统特定解码器进行过滤，该解码器可以通过open pragma在本地设置

Perl本身不处理文件打开时的错误，因此你必须通过检查open的退出状态自行处理这些错误。$!中包含导致open失败的错误信息。

在Windows上，默认的解码器是"CRLF"过滤器，它将输入中的任何"\r"序列映射为""

打开二进制文件

版本 ≥ 5.8.0

```
open my $filehandle, '<:raw', 'path/to/file' or die "无法打开 $name_of_file, $!";
```

这表示 Perl 在 Windows 上不进行CRLF转换。

打开 UTF8 文本文件

版本 ≥ 5.8.0

```
open my $filehandle, '<:raw:encoding(utf-8)', 'path/to/file'or die "无法打开
     $name_of_file, $!";
```

这表示 Perl 应该既避免CRLF转换，然后将得到的字节解码为
字符的字符串（内部实现为可以超过 255 的整数数组），而不是字节字符串

第 12.2 节：从文件读取

```
my $filename = '/path/to/file';

open my $fh, '<', $filename or die "打开文件失败：$filename";

# 然后你可以一次读取文件一行...
while(chomp(my $line = <$fh>)) {print
     $line . "";
}

# ...或一次性将整个文件读入数组
```

Chapter 12: File I/O (reading and writing files)

Mode	Explaination
`>`	**Write (trunc)**. Will overwrite existing files. Creates a new file if no file was found
`>>`	**Write (append)**. Will not overwrite files but append new content at the end of it. Will also create a file if used for opening a non existing file.
`<`	**Read**. Opens the file in read only mode.
`+<`	**Read / Write**. Will not create or truncate the file.
`+>`	**Read / Write (trunc)**. Will create and truncate the file.
`+>>`	**Read / Write (append)**. Will create but not truncate the file.

Section 12.1: Opening A FileHandle for Reading

Opening Generic ASCII Text Files

Version ≥ 5.6.0

```
open my $filehandle, '<', $name_of_file or die "Can't open $name_of_file, $!";
```

This is the basic idiom for "default" File IO and makes `$filehandle` a readable input stream of `bytes`, filtered by a default system-specific decoder, which can be locally set with the `open` pragma

Perl itself does not handle errors in file opening, so you have to handle those yourself by checking the exit condition of `open`. `$!` is populated with the error message that caused open to fail.

On Windows, the default decoder is a "CRLF" filter, which maps any "\r\n" sequences in the input to "\n"

Opening Binary Files

Version ≥ 5.8.0

```
open my $filehandle, '<:raw', 'path/to/file' or die "Can't open $name_of_file, $!";
```

This specifies that Perl should *not* perform a `CRLF` translation on Windows.

Opening UTF8 Text Files

Version ≥ 5.8.0

```
open my $filehandle, '<:raw:encoding(utf-8)', 'path/to/file'
    or die "Can't open $name_of_file, $!";
```

This specifies that Perl should both avoid `CRLF` translation, and then decode the resulting bytes into strings of *characters* (internally implemented as arrays of integers which can exceed 255), instead of strings of *bytes*

Section 12.2: Reading from a file

```
my $filename = '/path/to/file';

open my $fh, '<', $filename or die "Failed to open file: $filename";

# You can then either read the file one line at a time...
while(chomp(my $line = <$fh>)) {
    print $line . "\n";
}

# ...or read whole file into an array in one go
```

```
chomp(my @fileArray = <$fh>);
```

如果你知道输入文件是UTF-8编码，可以指定编码：

```
open my $fh, '<:encoding(utf8)', $filename or die "Failed to open file: $filename";
```

读取文件完成后，应关闭文件句柄：

```
close $fh or warn "close failed: $!";
```

另见：将文件读入变量

另一种更快的读取文件的方法是使用File::Slurper模块。如果你处理许多文件，这非常有用。

```
use File::Slurper;
my $file = read_text("path/to/file"); # 默认utf8且不转换CRLF
print $file; # 包含文件内容
```

另见：[使用slurp读取文件]

第12.3节：写入文件

这段代码打开一个文件以进行写入。如果文件无法打开，则返回错误。最后还会关闭文件。

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # 设置UTF-8为默认编码

# 以写入模式（">"）打开"output.txt"，并将其引用为变量$fh。
open(my $fh, ">", "output.txt")
# 如果操作失败，打印错误信息并退出。
or die "Can't open > output.txt: $!";
```

现在我们已经打开了一个准备写入的文件，通过$fh访问（该变量称为*filehandle*）。接下来我们可以使用print操作符将输出定向到该文件：

```
# 向$fh（"output.txt"）打印"Hello"。
print $fh "Hello";
# 完成后别忘了关闭文件！
close $fh or warn "关闭失败: $!";
```

open操作符的第一个参数是一个标量变量（此处为$fh）。由于它是在open操作符中定义的，因此被视为一个*文件句柄*。第二个参数">"（大于号）表示文件以写入模式打开。最后一个参数是要写入数据的文件路径。

要将数据写入文件，使用print操作符和*文件句柄*。注意在print操作符中，文件句柄和语句之间没有逗号，只有空格。

第12.4节："use autodie"，你就不需要检查文件打开/关闭失败

autodie允许你操作文件时无需显式检查打开/关闭失败。

自Perl 5.10.1起，autodie pragma已包含在核心Perl中。使用后，Perl会自动检查

```
chomp(my @fileArray = <$fh>);
```

If you know that your input file is UTF-8, you can specify the encoding:

```
open my $fh, '<:encoding(utf8)', $filename or die "Failed to open file: $filename";
```

After finished reading from the file, the filehandle should be closed:

```
close $fh or warn "close failed: $!";
```

See also: Reading a file into a variable

Another and **faster** way to read a file is to use File::Slurper Module. This is useful if you work with many files.

```
use File::Slurper;
my $file = read_text("path/to/file"); # utf8 without CRLF transforms by default
print $file; #Contains the file body
```

See also: [Reading a file with slurp]

Section 12.3: Write to a file

This code opens a file for writing. Returns an error if the file couldn't be opened. Also closes the file at the end.

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

# Open "output.txt" for writing (">") and from now on, refer to it as the variable $fh.
open(my $fh, ">", "output.txt")
# In case the action failed, print error message and quit.
or die "Can't open > output.txt: $!";
```

Now we have an open file ready for writing which we access through $fh (this variable is called a *filehandle*). Next we can direct output to that file using the print operator:

```
# Print "Hello" to $fh ("output.txt").
print $fh "Hello";
# Don't forget to close the file once we're done!
close $fh or warn "Close failed: $!";
```

The open operator has a scalar variable ($fh in this case) as its first parameter. Since it is defined in the open operator it is treated as a *filehandle*. Second parameter ">" (greater than) defines that the file is opened for writing. The last parameter is the path of the file to write the data to.

To write the data into the file, the print operator is used along with the *filehandle*. Notice that in the print operator there is no comma between the *filehandle* and the statement itself, just whitespace.

Section 12.4: "use autodie" and you won't need to check file open/close failures

autodie allows you to work with files without having to explicitly check for open/close failures.

Since Perl 5.10.1, the autodie pragma has been available in core Perl. When used, Perl will automatically check for

打开和关闭文件时的错误。

下面是一个示例，读取一个文件的所有行，然后写入到日志文件的末尾。

```
use 5.010;    # 5.010 及以后版本启用 "say"，它会打印参数，然后换行
use strict;   # 要求声明变量（避免因拼写错误导致的隐性错误）
use warnings; # 启用有用的语法相关警告
use open qw( :encoding(UTF-8) :std ); # 设置 UTF-8 为默认编码
use autodie;  # 自动处理文件打开和关闭时的错误

open(my $fh_in, '<', "input.txt"); # 自动检查失败

# 以追加模式打开文件（即使用 ">>"）
open( my $fh_log, '>>', "output.log"); # 自动检查失败

while (my $line = readline $fh_in) # 也可以写成：while (my $line = <$fh_in>)
{
     # 去除换行符
     chomp $line;

     # 写入日志文件
     say $fh_log $line or die "failed to print '$line'"; # autodie 不会检查 print
}

# 关闭文件句柄（自动检查失败）
close $fh_in;
close $fh_log;
```

顺便说一下，技术上你应该总是检查 print 语句。很多人不这样做，但 perl（Perl 解释器）不会自动检查，a
utodie 也不会。

第12.5节：回绕文件句柄

有时在阅读后需要回溯。

```
# 确定文件中的当前位置，以防第一行不是注释
my $current_pos = tell;

while (my $line = readline $fh)
{
    if ($line =~ /$START_OF_COMMENT_LINE/)
    {
        push @names, get_name_from_comment($line);
    }
    else {
        last; # 跳出 while 循环
    }
    $current_pos = tell; # 记录当前位置，以防需要回退到下一行
read
}

# 回退一行，以便稍后作为第一条数据行处理
seek $fh, $current_pos, 0;
```

第12.6节：读取和写入gzip压缩文件

写入gzip压缩文件

要写入gzip压缩文件，使用模块IO::Compress::Gzip并通过创建新实例来创建文件句柄

errors when opening and closing files.

Here is an example in which all of the lines of one file are read and then written to the end of a log file.

```
use 5.010;    # 5.010 and later enable "say", which prints arguments, then a newline
use strict;   # require declaring variables (avoid silent errors due to typos)
use warnings; # enable helpful syntax-related warnings
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding
use autodie;  # Automatically handle errors in opening and closing files

open(my $fh_in, '<', "input.txt"); # check for failure is automatic

# open a file for appending (i.e. using ">>")
open( my $fh_log, '>>', "output.log"); # check for failure is automatic

while (my $line = readline $fh_in) # also works: while (my $line = <$fh_in>)
{
     # remove newline
     chomp $line;

     # write to log file
     say $fh_log $line or die "failed to print '$line'"; # autodie doesn't check print
}

# Close the file handles (check for failure is automatic)
close $fh_in;
close $fh_log;
```

By the way, you should technically always check `print` statements. Many people don't, but `perl` (the Perl interpreter) doesn't do this automatically and neither does `autodie`.

Section 12.5: Rewind a filehandle

Sometimes it is needful to backtrack after reading.

```
# identify current position in file, in case the first line isn't a comment
my $current_pos = tell;

while (my $line = readline $fh)
{
    if ($line =~ /$START_OF_COMMENT_LINE/)
    {
        push @names, get_name_from_comment($line);
    }
    else {
        last; # break out of the while loop
    }
    $current_pos = tell; # keep track of current position, in case we need to rewind the next line
read
}

# Step back a line so that it can be processed later as the first data line
seek $fh, $current_pos, 0;
```

Section 12.6: Reading and Writing gzip compressed files

Writing a gzipped file

To write a gzipped file, **use** the module `IO::Compress::Gzip` and create a filehandle by creating a new instance of

IO::Compress::Gzip，用于所需的输出文件：

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # 设置UTF-8为默认编码

use IO::Compress::Gzip;

my $fh_out = IO::Compress::Gzip->new("hello.txt.gz");print $fh_o

ut "Hello World!";

close $fh_out;

use IO::Compress::Gzip;
```

读取gzip压缩文件

要读取gzip压缩文件，**使用**模块IO::Uncompress::Gunzip，然后通过创建新实例来创建文件句柄
IO::Uncompress::Gunzip，用于输入文件：

```
#!/bin/env perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # 设置 UTF-8 为默认编码

use IO::Uncompress::Gunzip;

my $fh_in = IO::Uncompress::Gunzip->new("hello.txt.gz");

my $line = readline $fh_in;

print $line;
```

第12.7节：设置IO的默认编码

```
# 对文件和标准输入/输出进行UTF-8编码/解码
use open qw( :encoding(UTF-8) :std );
```

该pragma将读取和写入文本（文件、标准输入、标准输出和
标准错误）的默认模式更改为UTF-8，这通常是编写新应用程序时所需的。

ASCII是UTF-8的子集，因此这不会对传统ASCII文件造成任何问题，并且有助于防止
将UTF-8文件当作ASCII处理时可能发生的意外文件损坏。

然而，了解你所处理文件的编码并相应地处理它们非常重要。（我们不应忽视Unicode的原因。）有关Unicode的更深入内容，请参见Perl Unicode主题。

IO::Compress::Gzip for the desired output file:

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Compress::Gzip;

my $fh_out = IO::Compress::Gzip->new("hello.txt.gz");

print $fh_out "Hello World!\n";

close $fh_out;

use IO::Compress::Gzip;
```

Reading from a gzipped file

To read from a gzipped file, **use** the module IO::Uncompress::Gunzip and then create a filehandle by creating a new instance of IO::Uncompress::Gunzip for the input file:

```
#!/bin/env perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Uncompress::Gunzip;

my $fh_in = IO::Uncompress::Gunzip->new("hello.txt.gz");

my $line = readline $fh_in;

print $line;
```

Section 12.7: Setting the default Encoding for IO

```
# encode/decode UTF-8 for files and standard input/output
use open qw( :encoding(UTF-8) :std );
```

This pragma changes the default mode of reading and writing text (files, standard input, standard output, and standard error) to UTF-8, which is typically what you want when writing new applications.

ASCII is a subset of UTF-8, so this is not expected to cause any problems with legacy ASCII files and will help protect you the accidental file corruption that can happen when treating UTF-8 files as ASCII.

However, it is important that you know what the encoding of your files is that you are dealing with and handle them accordingly. (Reasons that we should not ignore Unicode.) For more in depth treatment of Unicode, please see the Perl Unicode topic.

第13章：将文件内容读取到变量中

第13.1节：Path::Tiny

在脚本中多次使用《手动方式》中的惯用法很快会变得乏味，所以你可能想尝试使用一个模块。

```
use Path::Tiny;
my $contents = path($filename)->slurp;
```

如果你需要控制文件编码、行结束符等，可以传递一个binmode选项——参见man perlio：

```
my $contents = path($filename)->slurp( {binmode => ":encoding(UTF-8)"} );
```

Path::Tiny还有许多其他处理文件的函数，因此它可能是一个不错的选择。

第13.2节：手动方式

```
open my $fh, '<', $filename
    or die "无法打开 $filename 进行读取: $!";
my $contents = do { local $/; <$fh> };
```

打开文件后（如果你想读取特定的文件编码而非原始字节，请阅读man perlio），关键在于这个do块：<$fh>，文件句柄在钻石操作符中，返回文件中的一条记录。输入记录分隔符变量$/指定了"记录"的定义——默认情况下它被设置为换行符，因此"一条记录"意味着"一行"。由于$/是全局变量，local做了两件事：它创建了$/的一个临时本地副本，该副本将在块结束时消失，并赋予它undef（Perl赋予未初始化变量的"值"）这个（无）值。

当输入记录分隔符具有该（非）值时，钻石操作符将返回整个文件。（它将整个文件视为单行。）

使用 do，你甚至可以绕过手动打开文件。对于重复读取文件，

```
sub readfile { do { local(@ARGV,$/) = $_[0]; <> } }
my $content = readfile($filename);
```

可以使用。这里，另一个全局变量（@ARGV）被局部化，以模拟启动带参数的perl脚本时使用的相同过程。 $/ 仍然是 undef，因为它前面的数组"吃掉"了所有传入的参数。接下来，钻石操作符 <> 再次返回由 $/ 定义的一条记录（整个文件），并从 do 块返回，进而从子程序返回。

该子程序没有显式的错误处理，这是不好的做法！如果读取文件时发生错误，你将收到 undef 作为返回值，而不是空文件时的空字符串。

最后一段代码的另一个缺点是你无法使用 PerlIO 来处理不同的文件编码——你总是得到原始字节。

第13.3节：File::Slurp

不要使用它。尽管它存在已久，且仍是大多数程序员推荐的模块，但它有缺陷且不太可能被修复。

Chapter 13: Reading a file's content into a variable

Section 13.1: Path::Tiny

Using the idiom from The Manual Way several times in a script soon gets tedious so you might want to try a module.

```
use Path::Tiny;
my $contents = path($filename)->slurp;
```

You can pass a binmode option if you need control over file encodings, line endings etc. - see man perlio:

```
my $contents = path($filename)->slurp( {binmode => ":encoding(UTF-8)"} );
```

Path::Tiny also has a lot of other functions for dealing with files so it may be a good choice.

Section 13.2: The manual way

```
open my $fh, '<', $filename
    or die "Could not open $filename for reading: $!";
my $contents = do { local $/; <$fh> };
```

After opening the file (read man perlio if you want to read specific file encodings instead of raw bytes), the trick is in the do block: <$fh>, the file handle in a diamond operator, returns a single record from the file. The "input record separator" variable $/ specifies what a "record" is—by default it is set to a newline character so "a record" means "a single line". As $/ is a global variable, local does two things: it creates a temporary local copy of $/ that will vanish at the end of the block, and gives it the (non-)value undef (the "value" which Perl gives to uninitialized variables). When the input record separator has that (non-)value, the diamond operator will return the entire file. (It considers the entire file to be a single line.)

Using do, you can even get around manually opening a file. For repeated reading of files,

```
sub readfile { do { local(@ARGV,$/) = $_[0]; <> } }
my $content = readfile($filename);
```

can be used. Here, another global variable(@ARGV) is localized to simulate the same process used when starting a perl script with parameters. $/ is still undef, since the array in front of it "eats" all incoming arguments. Next, the diamond operator <> again delivers one record defined by $/ (the whole file) and returns from the do block, which in turn return from the sub.

The sub has no explicit error handling, which is bad practice! If an error occurs while reading the file, you will receive undef as return value, as opposed to an empty string from an empty file.

Another disadvantage of the last code is the fact that you cannot use PerlIO for different file encodings—you always get raw bytes.

Section 13.3: File::Slurp

Don't use it. Although it has been around for a long time and is still the module most programmers will suggest, it is broken and not likely to be fixed.

第13.4节：File::Slurper

这是一个极简模块，只将文件读入变量，别无其他功能。

```
use File::Slurper 'read_text';
my $contents = read_text($filename);
```

read_text() 接受两个可选参数，用于指定文件编码以及是否将行结束符在类 Unix 的 LF 和类 DOS 的 CRLF 标准之间转换：

```
my $contents = read_text($filename, 'UTF-8', 1);
```

第13.5节：将文件读入数组变量

```
open(my $fh, '<', "/some/path") or die $!;
my @ary = <$fh>;
```

当在列表上下文中求值时，钻石操作符返回由文件中所有行组成的列表（在此情况下，将结果赋给数组即提供了列表上下文）。行终止符会被保留，可以通过 chomp 来移除：

```
chomp(@ary); #移除所有数组元素的行终止符。
```

第13.6节：一行命令读取整个文件

输入记录分隔符可以通过 -0 开关指定（数字零，不是大写字母 O）。它接受八进制或十六进制数值。任何值 0400 或以上都会导致 Perl 读取整个文件，但按照惯例，用于此目的的值是
0777。

```
perl -0777 -e 'my $file = <>; print length($file)' input.txt
```

进一步简化，指定-n开关会使Perl自动将每一行（在我们的例子中是—整个文件）读入变量$_。

```
perl -0777 -ne 'print length($_)' input.txt
```

Section 13.4: File::Slurper

This is a minimalist module that only slurps files into variables, nothing else.

```
use File::Slurper 'read_text';
my $contents = read_text($filename);
```

`read_text()` takes two optional parameters to specify the file encoding and whether line endings should be translated between the unixish LF or DOSish CRLF standards:

```
my $contents = read_text($filename, 'UTF-8', 1);
```

Section 13.5: Slurping a file into an array variable

```
open(my $fh, '<', "/some/path") or die $!;
my @ary = <$fh>;
```

When evaluated in list context, the diamond operator returns a list consisting of all the lines in the file (in this case, assigning the result to an array supplies list context). The line terminator is retained, and can be removed by chomping:

```
chomp(@ary); #removes line terminators from all the array elements.
```

Section 13.6: Slurp file in one-liner

Input record separator can be specified with `-0` switch (*zero*, not *capital O*). It takes an octal or hexadecimal number as value. Any value `0400` or above will cause Perl to slurp files, but by convention, the value used for this purpose is `0777`.

```
perl -0777 -e 'my $file = <>; print length($file)' input.txt
```

Going further with minimalism, specifying `-n` switch causes Perl to automatically read each line (in our case — the whole file) into variable $_.

```
perl -0777 -ne 'print length($_)' input.txt
```

第14章：字符串和引用方法

第14.1节：字符串字面量引用

字符串字面量不进行转义或插值（引用字符串终止符除外）

```
print '这是一个字符串字面量'; # 向终端输出字面量的\和nprint '这个字面量包含一个 \'撇号 ';

# 输出'但不输出其前面的\
```

你可以使用替代的引用机制来避免冲突：

```
print q/这是一个字面量 |' <-- 2个字符 /;   # 输出|和'
print q^这是一个字面量 \' <-- 2个字符 ^;  # 也是如此
```

某些选定的引号字符是“平衡”的

```
print q{ 这是一个字面量 和 我包含 { 括号! } }; # 输出内部的{ }
```

第14.2节：双引号

双引号字符串使用**插值**和**转义**——这与单引号字符串不同。要使用双引号字符串，
可以使用双引号"或qq操作符。

```
my $greeting = "Hello!";print $
greeting;
# => Hello!（后跟换行符）

my $bush = "They misunderestimated me."prin
t qq/正如布什曾说过："$bush"/;# => 正如布什曾
说过："They misunderestimated me."（带换行符）
```

这里qq非常有用，可以避免转义引号。没有它，我们就得写成......

```
print "正如布什曾说过：\"$bush\"";
```

......这就没那么好看了。

Perl 不限制你在qq中只能使用斜杠/；你可以使用任何（可见）字符。

```
use feature 'say';

say qq/你可以使用斜杠.../;
say qq{...or 大括号...};
say qq^...or 帽子符...^;
say qq|...or 管道符...|;
# 说 qq ...但不包括空白字符。
```

你也可以将数组插入到字符串中。

```
use feature 'say';

my @letters = ('a', 'b', 'c');
say "我喜欢这些字母：@letters.";
```

Chapter 14: Strings and quoting methods

Section 14.1: String Literal Quoting

String literals imply no escaping or interpolation (with the exception of quoting string terminators)

```
print 'This is a string literal\n'; # emits a literal \ and n to terminal

print 'This literal contains a \'postraphe '; # emits the ' but not its preceding \
```

You can use alternative quoting mechanisms to avoid clashes:

```
print q/This is is a literal \' <-- 2 characters /;  # prints both \ and '
print q^This is is a literal \' <-- 2 characters ^;  # also
```

Certain chosen quote characters are "balanced"

```
print q{ This is a literal and I contain { parens! } }; # prints inner { }
```

Section 14.2: Double-quoting

Double-quoted strings use **interpolation** and **escaping** – unlike single-quoted strings. To double-quote a string, use either double quotes " or the qq operator.

```
my $greeting = "Hello!\n";
print $greeting;
# => Hello! (followed by a linefeed)

my $bush = "They misunderestimated me."
print qq/As Bush once said: "$bush"\n/;
# => As Bush once said: "They misunderestimated me." (with linefeed)
```

The qq is useful here, to avoid having to escape the quotation marks. Without it, we would have to write...

```
print "As Bush once said: \"$bush\"\n";
```

... which just isn't as nice.

Perl doesn't limit you to using a slash / with qq; you can use any (visible) character.

```
use feature 'say';

say qq/You can use slashes.../;
say qq{...or braces...};
say qq^...or hats...^;
say qq|...or pipes...|;
# say qq ...but not whitespace. ;
```

You can also interpolate arrays into strings.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
say "I like these letters: @letters.";
```

```
# => 我喜欢这些字母：a b c。
```

默认情况下，值之间用空格分隔–因为特殊变量$"默认是一个空格。当然，这可以被更改。

```
use feature 'say';

my @letters = ('a', 'b', 'c');
{local $" = ", "; say "@letters"; }     # a, b, c
```

如果你愿意，你可以选择use English并更改$LIST_SEPARATOR：

```
use v5.18; # 在旧版本的Perl中应避免使用English
use English;

my @letters = ('a', 'b', 'c');
{ local $LIST_SEPARATOR = ""; say "我最喜欢的字母：@letters" }
```

对于比这更复杂的情况，您应该使用循环。

```
say "我最喜欢的字母:";
say;
for my $letter (@letters) {
  say " - $letter";
}
```

插值不适用于哈希。

```
use feature 'say';

my %hash = ('a', 'b', 'c', 'd');
say "这不起作用: %hash"         # 这不起作用: %hash
```

有些代码滥用引用插值- 应避免。

```
use feature 'say';

say "2 + 2 == @{[ 2 + 2 ]}";            # 2 + 2 = 4 (避免这样)
say "2 + 2 == ${\( 2 + 2 )}";           # 2 + 2 = 4 (避免这样)
```

所谓的“笛卡尔操作符”使 Perl 对数组引用 @{ ... } 进行解引用，该数组引用 [...] 包含您想插值的表达式 2 + 2。当您使用这个技巧时，Perl 会构建一个匿名数组，然后解引用并丢弃它。

${\(...)} 版本稍微节省一些资源，但它仍然需要分配内存，而且更难阅读。

相反，可以考虑写成：

- 比如说 "2 + 2 == " . 2 + 2;
- my $result = 2 + 2; say "2 + 2 == $result"

第14.3节：Heredocs（多行字符串）

编写大型多行字符串很麻烦。

```
# => I like these letters: a b c.
```

By default the values are space-separated – because the special variable $" defaults to a single space. This can, of course, be changed.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
{local $" = ", "; say "@letters"; }     # a, b, c
```

If you prefer, you have the option to `use English` and change `$LIST_SEPARATOR` instead:

```
use v5.18; # English should be avoided on older Perls
use English;

my @letters = ('a', 'b', 'c');
{ local $LIST_SEPARATOR = "\n"; say "My favourite letters:\n\n@letters" }
```

For anything more complex than this, you should use a loop instead.

```
say "My favourite letters:";
say;
for my $letter (@letters) {
  say " - $letter";
}
```

Interpolation does *not* work with hashes.

```
use feature 'say';

my %hash = ('a', 'b', 'c', 'd');
say "This doesn't work: %hash"         # This doesn't work: %hash
```

Some code abuses interpolation of references – **avoid it**.

```
use feature 'say';

say "2 + 2 == @{[ 2 + 2 ]}";            # 2 + 2 = 4 (avoid this)
say "2 + 2 == ${\( 2 + 2 )}";           # 2 + 2 = 4 (avoid this)
```

The so-called "cart operator" causes perl to dereference `@{ ... }` the array reference `[ ... ]` that contains the expression that you want to interpolate, `2 + 2`. When you use this trick, Perl builds an anonymous array, then dereferences it and discards it.

The `${\( ... )}` version is somewhat less wasteful, but it still requires allocating memory and it is even harder to read.

Instead, consider writing:

- `say "2 + 2 == " . 2 + 2;`
- `my $result = 2 + 2; say "2 + 2 == $result"`

Section 14.3: Heredocs

Large Multi-Line strings are burdensome to write.

```
my $variable = <<'EOF';
这段文本块被字面解释，
引号无关紧要，它们只是文本
只有结尾左对齐的EOF才起作用。
EOF
```

注意：确保忽略stack-overflows的语法高亮，它非常错误。

插值Heredocs的工作方式相同。

```
my $variable = <<"I Want it to End";
这段文本块被解释。
引号被解释，且$插值会被插值...

但仍然，左对齐的"我希望它结束"很重要。
我希望它结束
```

在5.26.0*版本中待定的是一种"缩进Heredoc"语法，它会帮你去除左侧的填充空白

版本 ≥ 5.26.0

```
我的 $variable = <<~"MuchNicer";
    这段文本块会被解释。
引号会被解释，并且$插值     会被插入...

但仍然，左对齐的"我希望它结束"很重要。
MuchNicer
```

第14.4节：去除尾部换行符

函数chomp会从传入的每个标量中移除一个换行符（如果存在）。chomp会修改原始字符串并返回移除的字符数

```
my $str = "Hello World";my $rem
oved = chomp($str);print $str;
    # "Hello World"print $removed; #
1

# 再次使用 chomp，移除另一个换行符
$removed = chomp $str;
print $str;      # "Hello World"
print $removed; # 1

# 再次使用 chomp，但没有换行符可移除
$removed = chomp $str;
print $str;      # "Hello World"
print $removed; # 0
```

你也可以一次对多个字符串使用chomp：

```
my @strs = ("Hello", "World!"); # 第一个字符串有一个换行符，第二个有两个my $removed = chomp(@

strs); # @strs 现在是 ("Hello", "World!")
print $removed;            # 2

$removed = chomp(@strs); # @strs 现在是 ("Hello", "World!")
print $removed;           # 1

$removed = chomp(@strs); # @strs 仍然是 ("Hello", "World!")
```

```
my $variable = <<'EOF';
this block of text is interpreted literally,
no \'quotes matter, they're just text
only the trailing left-aligned EOF matters.
EOF
```

NB: Make sure you ignore stack-overflows syntax highlighter: It is very wrong.

And Interpolated Heredocs work the same way.

```
my $variable = <<"I Want it to End";
this block of text is interpreted.
quotes\nare interpreted, and $interpolations
get interpolated...
but still, left-aligned "I Want it to End" matters.
I Want it to End
```

Pending in 5.26.0* is an "Indented Heredoc" Syntax which trims left-padding off for you

Version ≥ 5.26.0

```
my $variable = <<~"MuchNicer";
    this block of text is interpreted.
    quotes\nare interpreted, and $interpolations
    get interpolated...
    but still, left-aligned "I Want it to End" matters.
MuchNicer
```

Section 14.4: Removing trailing newlines

The function `chomp` will remove *one* newline character, if present, from each scalar passed to it. `chomp` will mutate the original string and will return the number of characters removed

```
my $str = "Hello World\n\n";
my $removed = chomp($str);
print $str;      # "Hello World\n"
print $removed; # 1

# chomp again, removing another newline
$removed = chomp $str;
print $str;      # "Hello World"
print $removed; # 1

# chomp again, but no newline to remove
$removed = chomp $str;
print $str;      # "Hello World"
print $removed; # 0
```

You can also `chomp` more than one string at once:

```
my @strs = ("Hello\n", "World!\n\n"); # one newline in first string, two in second

my $removed = chomp(@strs); # @strs is now  ("Hello", "World!\n")
print $removed;              # 2

$removed = chomp(@strs); # @strs is now ("Hello", "World!")
print $removed;           # 1

$removed = chomp(@strs); # @strs is still ("Hello", "World!")
```

```
print $removed;          # 0
```

但通常，没有人会在意删除了多少个换行符，所以chomp通常在void上下文中使用，且通常是因为从文件中读取了行：

```
while (my $line = readline $fh)
{
    chomp $line;

    # 现在对$line做一些操作
}

my @lines = readline $fh2;

chomp (@lines); # 删除每行末尾的换行符
```

```
print $removed;          # 0
```

But usually, no one worries about how many newlines were removed, so chomp is usually seen in void context, and usually due to having read lines from a file:

```
while (my $line = readline $fh)
{
    chomp $line;

    # now do something with $line
}

my @lines = readline $fh2;

chomp (@lines); # remove newline from end of each line
```

第15章：在未加引号的分隔符上拆分字符串

第15.1节：parse_line()

使用Text::ParseWords的parse_line()：

```
use 5.010;
use Text::ParseWords;

my $line = q{"一个带引号的，逗号", word1, word2};
my @parsed = parse_line(',', 1, $line);
say for @parsed;
```

输出：

```
"一个带引号的，逗号"
 word1
 word2
```

第15.2节：Text::CSV 或 Text::CSV_XS

```
use Text::CSV; # 可以使用 Text::CSV，如果安装了会自动切换到 _XS
$sep_char = ",";
my $csv = Text::CSV->new({sep_char => $sep_char});
my $line = q{"一个带引号的，逗号", word1, word2};
$csv->parse($line);
my @fields = $csv->fields();print j
oin("", @fields)."";
```

输出：

```
一个带引号的，逗号
 word1
 word2
```

注意事项

- 默认情况下，Text::CSV 不会像 Text::ParseWords 那样去除分隔符周围的空白字符。但是，在构造函数属性中添加allow_whitespace=>1可以实现该效果。

  ```
  my $csv = Text::CSV_XS->new({sep_char => $sep_char, allow_whitespace=>1});
  ```

  输出：

  ```
  带引号的,逗号
  word1
  word2
  ```

- 该库支持转义特殊字符（引号、分隔符）
- 该库支持可配置的分隔符字符、引号字符和转义字符

文档：http://search.cpan.org/perldoc/Text::CSV

Chapter 15: Split a string on unquoted separators

Section 15.1: parse_line()

Using parse_line() of Text::ParseWords:

```
use 5.010;
use Text::ParseWords;

my $line = q{"a quoted, comma", word1, word2};
my @parsed = parse_line(',', 1, $line);
say for @parsed;
```

Output:

```
"a quoted, comma"
 word1
 word2
```

Section 15.2: Text::CSV or Text::CSV_XS

```
use Text::CSV; # Can use Text::CSV which will switch to _XS if installed
$sep_char = ",";
my $csv = Text::CSV->new({sep_char => $sep_char});
my $line = q{"a quoted, comma", word1, word2};
$csv->parse($line);
my @fields = $csv->fields();
print join("\n", @fields)."\n";
```

Output:

```
a quoted, comma
 word1
 word2
```

NOTES

- By default, Text::CSV does not strip whitespace around separator character, the way Text::ParseWords does. However, adding allow_whitespace=>1 to constructor attributes achieves that effect.

  ```
  my $csv = Text::CSV_XS->new({sep_char => $sep_char, allow_whitespace=>1});
  ```

  Output:

  ```
  a quoted, comma
  word1
  word2
  ```

- The library supports escaping special characters (quotes, separators)
- The library supports configurable separator character, quote character, and escape character

Documentatoin: http://search.cpan.org/perldoc/Text::CSV

第16章：面向对象的Perl

第16.1节：在现代Perl中定义类

虽然可以，但不建议在现代Perl中从零定义类。建议使用提供更多功能和便利的辅助面向对象系统。这些系统包括
：

- Moose - 受Perl 6面向对象设计启发
- Class::Accessor - Moose的轻量级替代品
- Class::Tiny - 真正极简的类构建器

驼鹿

```
package Foo;
use Moose;

has bar => (is => 'ro');                    # 只读属性
has baz => (is => 'rw', isa => 'Bool');   # 可读写的布尔属性

sub qux {
    my $self = shift;
    my $barIsBaz = $self->bar eq 'baz';   # 属性获取器
    $self->baz($barIsBaz);                  # 属性设置器
}
```

Class::Accessor（Moose 语法）

```
package Foo;
use Class::Accessor 'antlers';

has bar => (is => 'ro');                # 只读属性has baz => (is => 'rw', is
a => 'Bool');  # 可读写属性（仅支持 'is'，类型被忽略）
```

Class::Accessor（原生语法）

```
package Foo;
use base qw(Class::Accessor);

Foo->mk_accessors(qw(bar baz));  # 一些可读写属性
Foo->mk_accessors(qw(qux));      # 一个只读属性
```

Class::Tiny

```
package Foo;
use Class::Tiny qw(bar baz);  # 仅属性
```

第16.2节：创建对象

与许多其他语言不同，Perl 没有为对象分配内存的构造函数。相反，应该编写一个类方法，既创建数据结构又填充数据（你可能知道这称为工厂方法设计模式）。

Chapter 16: Object-oriented Perl

Section 16.1: Defining classes in modern Perl

Although available, defining a class from scratch is not recommended in modern Perl. Use one of helper OO systems which provide more features and convenience. Among these systems are:

- Moose - inspired by Perl 6 OO design
- Class::Accessor - a lightweight alternative to Moose
- Class::Tiny - truly minimal class builder

Moose

```
package Foo;
use Moose;

has bar => (is => 'ro');                    # a read-only property
has baz => (is => 'rw', isa => 'Bool');   # a read-write boolean property

sub qux {
    my $self = shift;
    my $barIsBaz = $self->bar eq 'baz';   # property getter
    $self->baz($barIsBaz);                  # property setter
}
```

Class::Accessor (Moose syntax)

```
package Foo;
use Class::Accessor 'antlers';

has bar => (is => 'ro');                    # a read-only property
has baz => (is => 'rw', isa => 'Bool');   # a read-write property (only 'is' supported, the type is
ignored)
```

Class::Accessor (native syntax)

```
package Foo;
use base qw(Class::Accessor);

Foo->mk_accessors(qw(bar baz));  # some read-write properties
Foo->mk_accessors(qw(qux));      # a read-only property
```

Class::Tiny

```
package Foo;
use Class::Tiny qw(bar baz);  # just props
```

Section 16.2: Creating Objects

Unlike many other languages, Perl does not have constructors that allocate memory for your objects. Instead, one should write a class method that both create a data structure and populate it with data (you may know it as the Factory Method design pattern).

```
package Point;
use strict;

sub new {
    my ($class, $x, $y) = @_;
    my $self = { x => $x, y => $y }; # 在哈希中存储对象数据
    bless $self, $class;             # 将哈希绑定到类
    return $self;
}
```

此方法可按如下方式使用：

```
my $point = Point->new(1, 2.5);
```

每当使用箭头操作符->调用方法时，其左操作数会被添加到给定的参数列表前面。因此，@_在new中将包含值('Point', 1, 2.5)。

名称new本身没有特殊含义。你可以按自己的喜好调用工厂方法。

哈希本身没有什么特别的。你也可以用以下方式实现相同的功能：

```
package Point;
use strict;

sub new {
    my ($class, @coord) = @_;
    my $self = \@coord;
    bless $self, $class;
    return $self;
}
```

一般来说，任何引用都可以是对象，甚至是标量引用。但通常情况下，哈希是表示对象数据最方便的方式。

第16.3节：定义类

一般来说，Perl中的类就是包。它们可以像普通包一样包含数据和方法。

```
package Point;
use strict;

my $CANVAS_SIZE = [1000, 1000];

sub new {
    ...
}

sub polar_coordinates {
    ...
}

1;
```

需要注意的是，在包中声明的变量是类变量，而不是对象（实例）变量。更改包级变量会影响该类的所有对象。有关如何存储对象特定数据，请参见“创建对象”。

```
package Point;
use strict;

sub new {
    my ($class, $x, $y) = @_;
    my $self = { x => $x, y => $y }; # store object data in a hash
    bless $self, $class;             # bind the hash to the class
    return $self;
}
```

This method can be used as follows:

```
my $point = Point->new(1, 2.5);
```

Whenever the arrow operator `->` is used with methods, its left operand is prepended to the given argument list. So, `@_` in **new** will contain values `('Point', 1, 2.5)`.

There is nothing special in the name **new**. You can call the factory methods as you prefer.

There is nothing special in hashes. You could do the same in the following way:

```
package Point;
use strict;

sub new {
    my ($class, @coord) = @_;
    my $self = \@coord;
    bless $self, $class;
    return $self;
}
```

In general, any reference may be an object, even a scalar reference. But most often, hashes are the most convenient way to represent object data.

Section 16.3: Defining Classes

In general, classes in Perl are just packages. They can contain data and methods, as usual packages.

```
package Point;
use strict;

my $CANVAS_SIZE = [1000, 1000];

sub new {
    ...
}

sub polar_coordinates {
    ...
}

1;
```

It is important to note that the variables declared in a package are class variables, not object (instance) variables. Changing of a package-level variable affects all objects of the class. How to store object-specific data, see in "Creating Objects".

使类包具有特定性的，是箭头操作符->。它可以用于裸词后：

```
Point->new(...);
```

或者用于标量变量（通常保存引用）后：

```
my @polar = $point->polar_coordinates;
```

箭头左侧的内容会被添加到方法的参数列表前面。例如，在调用后

```
Point->new(1, 2);
```

array @_ 在 new 中将包含三个参数：('Point', 1, 2)。

表示类的包应考虑此约定，并预期其所有方法将有
一个额外的参数。

第16.4节：继承与方法解析

要使一个类成为另一个类的子类，使用 parent 预处理指令：

```
package Point;
use strict;
...
1;

package Point2D;
use strict;
use parent qw(Point);
...
1;

package Point3D;
use strict;
use parent qw(Point);
...
1;
```

Perl 允许多重继承：

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);
...
1;
```

继承完全是关于在特定情况下调用哪个方法的决议。由于纯 Perl 并不规定用于存储对象数据的数据结构，继承与此无关。

考虑以下类层次结构：

```
package GeometryObject;
use strict;

sub transpose { ...}

1;
```

What makes class packages specific, is the arrow operator `->`. It may be used after a bare word:

```
Point->new(...);
```

or after a scalar variable (usually holding a reference):

```
my @polar = $point->polar_coordinates;
```

What is to the left of the arrow is prepended to the given argument list of the method. For example, after call

```
Point->new(1, 2);
```

array `@_` in **new** will contain three arguments: `('Point', 1, 2)`.

Packages representing classes should take this convention into account and expect that all their methods will have one extra argument.

Section 16.4: Inheritance and methods resolution

To make a class a subclass of another class, use `parent` pragma:

```
package Point;
use strict;
...
1;

package Point2D;
use strict;
use parent qw(Point);
...
1;

package Point3D;
use strict;
use parent qw(Point);
...
1;
```

Perl allows for multiple inheritance:

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);
...
1;
```

Inheritance is all about resolution which method is to be called in a particular situation. Since pure Perl does not prescribe any rules about the data structure used to store object data, inheritance has nothing to do with that.

Consider the following class hierarchy:

```
package GeometryObject;
use strict;

sub transpose { ...}

1;
```

```
package Point;
use strict;
use parent qw(GeometryObject);

sub new { ... };

1;

package PlanarObject;
use strict;
use parent qw(GeometryObject);

sub transpose { ... }

1;

package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new { ... }

sub polar_coordinates { … }

1;
```

方法解析过程如下：

1. 起点由箭头操作符的左操作数定义。

   ◦ 如果它是一个裸词：

   ```
   Point2D->new(…);
   ```

   …或者是一个保存字符串的标量变量：

   ```
   my $class = 'Point2D';
   $class->new(...);
   ```

   …那么起点就是具有相应名称的包（两个例子中都是Point2D）。

   ◦ 如果左操作数是一个保存了blessed引用的标量变量：

   ```
   my $point = {…};
   bless $point, 'Point2D'; # 通常，它被封装在类方法中
   my @coord = $point->polar_coordinates;
   ```

   那么起点就是该引用的类（同样是Point2D）。箭头操作符不能用于调用未被祝福（unblessed）引用的方法。

2.如果起点包含所需的方法，则直接调用。

因此，由于Point2D::new存在，

```
Point2D->new(…);
```

将直接调用它。

```
package Point;
use strict;
use parent qw(GeometryObject);

sub new { ... };

1;

package PlanarObject;
use strict;
use parent qw(GeometryObject);

sub transpose { ... }

1;

package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new { ... }

sub polar_coordinates { ... }

1;
```

The method resolution works as follows:

1. The starting point is defined by the left operand of the arrow operator.

   ◦ If it is a bare word:

   ```
   Point2D->new(...);
   ```

   ...or a scalar variable holding a string:

   ```
   my $class = 'Point2D';
   $class->new(...);
   ```

   ...then the starting point is the package with the corresponding name (`Point2D` in both examples).

   ◦ If the left operand is a scalar variable holding a *blessed* reference:

   ```
   my $point = {...};
   bless $point, 'Point2D'; # typically, it is encapsulated into class methods
   my @coord = $point->polar_coordinates;
   ```

   then the starting point is the class of the reference (again, `Point2D`). The arrow operator cannot be used to call methods for *unblessed* references.

2. If the starting point contains the required method, it is simply called.

   Thus, since `Point2D::new` exists,

   ```
   Point2D->new(...);
   ```

   will simply call it.

3. 如果起始点不包含所需的方法，则在parent类中进行深度优先搜索
   执行。在上述示例中，搜索顺序如下：

   - Point2D
   - Point（Point2D的第一个父类）
   - GeometryObject（Point的父类）
   - PlanarObject（Point2D的第二个父类）

   例如，在以下代码中：

```
my $point = Point2D->new(...);
$point->transpose(...);
```

   将调用的方法是GeometryObject::transpose，尽管它会在
   PlanarObject::transpose中被重写。

4.你可以显式设置起始点。

   在前面的例子中，你可以显式调用PlanarObject::transpose，方式如下：

```
my $point = Point2D->new(...);
$point->PlanarObject::transpose(...);
```

5. 以类似的方式，SUPER::会在当前类的父类中进行方法搜索。

   例如，

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new {
    (my $class, $x, $y) = @_;
    my $self = $class->SUPER::new;
    ...
}

1;
```

   将在Point2D::new执行过程中调用Point::new。

第16.5节：类方法和对象方法

在Perl中，类（静态）方法和对象（实例）方法之间的区别不像某些其他语言那样明显，但仍然存在。

箭头操作符->的左操作数成为要调用方法的第一个参数。它可以是一个
字符串：

```
# new 的第一个参数在两种情况下都是字符串 'Point'
Point->new(...);

my $class = 'Point';
```

3. If the starting point does not contain the required method, depth-first search in the parent classes is performed. In the example above, the search order will be as follows:

   - Point2D
   - Point (first parent of Point2D)
   - GeometryObject (parent of Point)
   - PlanarObject (second parent of Point2D)

   For example, in the following code:

```
my $point = Point2D->new(...);
$point->transpose(...);
```

   the method that will be called is GeometryObject::transpose, even though it would be overridden in PlanarObject::transpose.

4. You can set the starting point explicitly.

   In the previous example, you can explicitly call PlanarObject::transpose like so:

```
my $point = Point2D->new(...);
$point->PlanarObject::transpose(...);
```

5. In a similar manner, SUPER:: performs method search in parent classes of the current class.

   For example,

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new {
    (my $class, $x, $y) = @_;
    my $self = $class->SUPER::new;
    ...
}

1;
```

   will call Point::new in the course of the Point2D::new execution.

Section 16.5: Class and Object Methods

In Perl, the difference between class (static) and object (instance) methods is not so strong as in some other languages, but it still exists.

The left operand of the arrow operator -> becomes the first argument of the method to be called. It may be either a string:

```
# the first argument of new is string 'Point' in both cases
Point->new(...);

my $class = 'Point';
```

```
$class->new(...);
```

或者是一个对象引用：

```
# $point 中包含的引用是 polar_coordinates 的第一个参数
my $point = Point->new(...);
my @coord = $point->polar_coordinates;
```

类方法就是那些期望其第一个参数是字符串的方法，而对象方法则是期望其第一个参数是对象引用的方法。

类方法通常不会对其第一个参数做任何操作，该参数只是类的名称。通常，它仅被 Perl 自身用于方法解析。因此，典型的类方法也可以被对象调用：

```
my $width = Point->canvas_width;

my $point = Point->new(...);
my $width = $point->canvas_width;
```

虽然允许这种语法，但它常常会引起误解，因此最好避免使用。

对象方法接收一个对象引用作为第一个参数，因此它们可以访问对象数据（不同于类方法）：

```
package Point;
use strict;

sub polar_coordinates {
    my ($point) = @_;
    my $x = $point->{x};
    my $y = $point->{y};
    return (sqrt($x * $x + $y * $y), atan2($y, $x));
}

1;
```

同一个方法可以同时处理两种情况：当它作为类方法或对象方法被调用时：

```
sub universal_method {
    my $self = shift;
    if (ref $self) {
        # 对象逻辑
        ...
    }
    else {
        # 类逻辑
        ...
    }
}
```

第16.6节：角色

Perl中的角色本质上是

- 一组方法和属性，
- 直接注入到类中。

```
$class->new(...);
```

or an object reference:

```
# reference contained in $point is the first argument of polar_coordinates
my $point = Point->new(...);
my @coord = $point->polar_coordinates;
```

Class methods are just the ones that expect their first argument to be a string, and object methods are the ones that expect their first argument to be an object reference.

Class methods typically do not do anything with their first argument, which is just a name of the class. Generally, it is only used by Perl itself for method resolution. Therefore, a typical class method can be called for an object as well:

```
my $width = Point->canvas_width;

my $point = Point->new(...);
my $width = $point->canvas_width;
```

Although this syntax is allowed, it is often misleading, so it is better to avoid it.

Object methods receive an object reference as the first argument, so they can address the object data (unlike class methods):

```
package Point;
use strict;

sub polar_coordinates {
    my ($point) = @_;
    my $x = $point->{x};
    my $y = $point->{y};
    return (sqrt($x * $x + $y * $y), atan2($y, $x));
}

1;
```

The same method can track both cases: when it is called as a class or an object method:

```
sub universal_method {
    my $self = shift;
    if (ref $self) {
        # object logic
        ...
    }
    else {
        # class logic
        ...
    }
}
```

Section 16.6: Roles

A role in Perl is essentially

- a set of methods and attributes which
- injected into a class directly.

角色提供一段功能，可以*组合*到（或*应用*于）任何类（称为*使用*该角色）。角色不能被继承，但可以被另一个角色使用。

角色还可以*要求*使用该角色的类实现某些方法，而不是自己实现这些方法（就像Java或C#中的接口）。

Perl没有内置对角色的支持，但有CPAN类提供此类支持。

Moose::Role

```
package Chatty;
use Moose::Role;

requires '引入';  # 消费类必须实现的方法sub greet {            # 角色中已实现

的方法print "嗨！";

}


包 Parrot;
use Moose;

with 'Chatty';sub

 introduce {print
     "我是巴迪。";}
```

Role::Tiny

如果您的面向对象系统不支持角色（例如`Class::Accessor`或`Class::Tiny`），请使用。 不支持属性。

```
package Chatty;
use Role::Tiny;

requires '引入';  # 消费类必须实现的方法sub greet {            # 角色中已实现

的方法print "嗨！";

}

包 Parrot;
use Class::Tiny;
use Role::Tiny::With;

with 'Chatty';

sub introduce {pri
     nt "我是巴迪。";}
```

A role provides a piece of functionality which can be *composed* into (or *applied* to) any class (which is said to *consume* the role). A role cannot be inherited but may be consumed by another role.

A role may also *require* consuming classes to implement some methods instead of implementing the methods itself (just like interfaces in Java or C#).

Perl does not have built-in support for roles but there are CPAN classes which provide such support.

Moose::Role

```
package Chatty;
use Moose::Role;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}


package Parrot;
use Moose;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

Role::Tiny

Use if your OO system does not provide support for roles (e.g. `Class::Accessor` or `Class::Tiny`). Does not support attributes.

```
package Chatty;
use Role::Tiny;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}

package Parrot;
use Class::Tiny;
use Role::Tiny::With;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

第17章：异常处理

第17.1节：eval和die

这是内置的处理“异常”的方法，无需依赖第三方库如Try::Tiny。

```
my $ret;

eval {
  $ret = some_function_that_might_die();
  1;
} or do {
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};

# use $ret
```

我们“滥用”了 die 返回假值的事实，整个代码块的返回值是代码块中最后一个表达式的值：

- 如果 $ret 成功赋值，那么 1; 表达式就是 eval 代码块中最后执行的操作。这样 eval 代码块的值为真，or do 块不会执行。
- 如果 some_function_that_might_die() 调用了 die，那么 eval 代码块中最后执行的是 die。这样 eval 代码块的值为假，or do 块会执行。
- 在 or do 块中你必须做的第一件事是读取 $@。这个全局变量将保存传递给 die 的任何参数。 || "Zombie Error" 这种保护写法很流行，但在一般情况下并不必要。

理解这一点很重要，因为并非所有代码都会通过调用 die 来失败，但无论如何都可以使用相同的结构。考虑一个数据库函数返回：

- 成功时受影响的行数
- 如果查询成功但没有行受影响，则返回 '0 but true'
- 0如果查询不成功。

在这种情况下，你仍然可以使用相同的惯用法，但必须跳过最后的 1;，并且这个函数必须是 eval 中的最后一条语句。类似这样：

```
eval {
  my $value = My::Database::retrieve($my_thing); # 失败时调用 die
  $value->set_status("Completed");
  $value->set_completed_timestamp(time());
  $value->update(); # 失败时返回假值
} or do { # 处理 die 和 0 返回值两种情况
  my $eval_error = $@ || "Zombie error!";
handle_error($eval_error);
};
```

Chapter 17: Exception handling

Section 17.1: eval and die

This is the built-in way to deal with "exceptions" without relying on third party libraries like Try::Tiny.

```
my $ret;

eval {
  $ret = some_function_that_might_die();
  1;
} or do {
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};

# use $ret
```

We "abuse" the fact that `die` has a false return value, and the return value of the overall code block is the value of the last expression in the code block:

- if `$ret` is assigned to successfully, then the `1;` expression is the last thing that happens in the `eval` code block. The `eval` code block thus has a true value, so the `or do` block does not run.
- if `some_function_that_might_die()` does `die`, then the last thing that happens in the `eval` code block is the `die`. The `eval` code block thus has a false value and the `or do` block does run.
- The first thing you *must* do in the `or do` block is read `$@`. This global variable will hold whatever argument was passed to `die`. The `|| "Zombie Error"` guard is popular, but unnecessary in the general case.

This is important to understand because some not all code does fail by calling die, but the same structure can be used regardless. Consider a database function that returns:

- the number of rows affected on success
- `'0 but true'` if the query is successful but no rows were affected
- `0` if the query was not successful.

In that case you can still use the same idiom, but you *have* to skip the final `1;`, and this function *has* to be the last thing in the eval. Something like this:

```
eval {
  my $value = My::Database::retrieve($my_thing); # dies on fail
  $value->set_status("Completed");
  $value->set_completed_timestamp(time());
  $value->update(); # returns false value on fail
} or do { # handles both the die and the 0 return value
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};
```

第18章：正则表达式

第18.1节：使用正则表达式替换字符串

```
s/foo/bar/;         # 在 $_ 中将 "foo" 替换为 "bar"
my $foo = "foo";
$foo =~ s/foo/bar/; # 使用绑定操作符 =~ 对另一个变量执行上述操作
s~ foo ~ bar ~;      # 使用 ~ 作为分隔符
$foo = s/foo/bar/r; # 非破坏性 r 标志：返回替换后的字符串而不修改绑定的变量

s/foo/bar/g;        # 替换所有匹配项
```

第18.2节：匹配字符串

=~ 操作符尝试将正则表达式（用 / 分隔）匹配到字符串上：

```
my $str = "hello world";print
 "Hi, yourself!" if $str =~ /^hello/;
```

/^hello/ 是实际的正则表达式。 ^ 是一个特殊字符，表示正则表达式必须从字符串开头开始匹配，而不是在中间某处匹配。然后正则表达式尝试按顺序匹配以下字母 h、e、l、l 和 o。

如果省略变量，正则表达式会尝试匹配默认变量（$_）：

```
$_ = "hello world";prin

t "Ahoy!" if /^hello/;
```

你也可以在正则表达式前加上m操作符来使用不同的定界符：

```
m~^hello~;
m{^hello};
m|^hello|;
```

当匹配包含/字符的字符串时，这很有用：

```
print "user directory" if m|^/usr|;
```

第18.3节：使用正则表达式解析字符串

通常，使用正则表达式解析复杂结构不是一个好主意。但这可以做到。例如，你可能想将数据加载到Hive表中，字段由逗号分隔，但复杂类型如数组由"|"分隔。文件包含的记录中所有字段由逗号分隔，复杂类型包含在方括号内。在这种情况下，这段一次性Perl代码可能足够：

```
echo "1,2,[3,4,5],5,6,[7,8],[1,2,34],5" | \
    perl -ne \
        'while( /\[[^,\]]+\,.*\]/ ){
          if( /\[([^\]\|]+)\]/){
              $text = $1;
$text_to_replace = $text;
              $text =~ s/\,/\|/g;
                 s/$text_to_replace/$text/;
```

Chapter 18: Regular Expressions

Section 18.1: Replace a string using regular expressions

```
s/foo/bar/;         # replace "foo" with "bar" in $_
my $foo = "foo";
$foo =~ s/foo/bar/; # do the above on a different variable using the binding operator =~
s~ foo ~ bar ~;     # using ~ as a delimiter
$foo = s/foo/bar/r; # non-destructive r flag: returns the replacement string without modifying the
variable it's bound to
s/foo/bar/g;        # replace all instances
```

Section 18.2: Matching strings

The =~ operator attempts to match a regular expression (set apart by /) to a string:

```
my $str = "hello world";
print "Hi, yourself!\n" if $str =~ /^hello/;
```

/^hello/ is the actual regular expression. The ^ is a special character that tells the regular expression to start with the beginning of the string and not match in the middle somewhere. Then the regex tries to find the following letters in order h, e, l, l, and o.

Regular expressions attempt to match the default variable ($_) if bare:

```
$_ = "hello world";

print "Ahoy!\n" if /^hello/;
```

You can also use different delimiters is you precede the regular expression with the m operator:

```
m~^hello~;
m{^hello};
m|^hello|;
```

This is useful when matching strings that include the / character:

```
print "user directory" if m|^/usr|;
```

Section 18.3: Parsing a string with a regex

Generally, it's not a good idea to use a regular expression to parse a complex structure. But it can be done. For instance, you might want to load data into hive table and fields are separated by comma but complex types like array are separated by a "|". Files contain records with all fields separated by comma and complex type are inside square bracket. In that case, this bit of disposable Perl might be sufficient:

```
echo "1,2,[3,4,5],5,6,[7,8],[1,2,34],5" | \
    perl -ne \
        'while( /\[[^,\]]+\,.*\]/ ){
            if( /\[([^\]\|]+)\]/){
                $text = $1;
                $text_to_replace = $text;
                $text =~ s/\,/\|/g;
                s/$text_to_replace/$text/;
```

```
}
        } print'
```

你需要抽查输出结果：

> 1,2,[3|4|5],5,6,[7|8],[1|2|34],5

第18.4节：模式匹配中 \Q 和 \E 的用法

\Q 和 \E 之间的内容被视为普通字符

```
#!/usr/bin/perl

my $str = "hello.it's.me";

my @test = (
  "hello.it's.me",
    "hello/it's!me",
    );

sub ismatched($) { $_[0] ? "MATCHED!" : "DID NOT MATCH!" }

my @match = (
      [ general_match=> sub { ismatched /$str/ } ],
      [ qe_match    => sub { ismatched /\Q$str\E/ } ],
      );

for (@test) {
    print "\String = '$_':";foreach m

y $method (@match) {my($name,$
    match) = @$method;print "  - $n
    ame: ", $match->(), "";}
```

}

输出

```
String = 'hello.it's.me':
  - general_match: MATCHED!
  - qe_match: MATCHED!
String = 'hello/it's!me':
  -general_match: MATCHED!
  -qe_match: DID NOT MATCH!
```

```
        }
    } print'
```

You'll want to spot check the output:

> 1,2,[3|4|5],5,6,[7|8],[1|2|34],5

Section 18.4: Usage of \Q and \E in pattern matching

What's between \Q and \E is treated as normal characters

```
#!/usr/bin/perl

my $str = "hello.it's.me";

my @test = (
  "hello.it's.me",
    "hello/it's!me",
    );

sub ismatched($) { $_[0] ? "MATCHED!" : "DID NOT MATCH!" }

my @match = (
      [ general_match=> sub { ismatched /$str/ } ],
      [ qe_match    => sub { ismatched /\Q$str\E/ } ],
      );

for (@test) {
    print "\String = '$_':\n";

foreach my $method (@match) {
    my($name,$match) = @$method;
    print "  - $name: ", $match->(), "\n";
}
```

}

Output

```
String = 'hello.it's.me':
  - general_match: MATCHED!
  - qe_match: MATCHED!
String = 'hello/it's!me':
  - general_match: MATCHED!
  - qe_match: DID NOT MATCH!
```

第19章：XML解析

第19.1节：使用XML::Twig进行解析

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

use strict;
use warnings 'all';

use XML::Twig;

my $twig = XML::Twig->parse( |*DATA );

#我们可以使用 'root' 方法来查找 XML 的根节点。
my $root = $twig->root;

#first_child 查找匹配某个值的第一个子元素。
my $title = $root->first_child('title');

#text 读取元素的文本内容。
my $title_text = $title->text;print "标

题是: ", $title_text, "";

#以上内容可以合并为：
print $twig ->root->first_child_text('title'), "";

## 你可以使用 'children' 方法来遍历多个项目：
my $list = $twig->root->first_child('list');

#children 方法可以选择性地接受一个元素"标签"，否则它会返回所有子元素。
foreach my $element ( $list->children ) {

    #'att' 方法用于读取属性
    print "元素ID: ", $element->att('id') // '这里没有', " 是 ", $element->text,"";

}

#如果需要做更复杂的操作，可以使用 'xpath'。
#get_xpath 或 findnodes 功能相同：
#返回匹配列表，或者如果指定第二个数字参数，则返回该编号的匹配项。

#xpath 语法相当丰富，但在这里——我们搜索：
# 树中的任意位置：//
# 节点名为 'item'
#带有id属性[@id]
#且该id属性等于"1000"。
#通过指定'0'，我们表示'只返回第一个匹配项'。

print "项目 1000 是: ", $twig->get_xpath( '//item[@id="1000"]', 0 )->text, "";

#这与`map`结合得非常好，例如可以对多个项目执行相同的操作
print "所有ID:", join ( "", map { $_ -> att('id') } $twig -> get_xpath('//item'));
#注意这也会找到"summary"下的项目，因为 //

__DATA__
<?xml 版本="1.0" 编码="utf-8"?>
<root>
  <title>一些示例 XML</title>
  <first key="value" key2="value2">
```

Chapter 19: XML Parsing

Section 19.1: Parsing with XML::Twig

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

use strict;
use warnings 'all';

use XML::Twig;

my $twig = XML::Twig->parse( \*DATA );

#we can use the 'root' method to find the root of the XML.
my $root = $twig->root;

#first_child finds the first child element matching a value.
my $title = $root->first_child('title');

#text reads the text of the element.
my $title_text = $title->text;

print "Title is: ", $title_text, "\n";

#The above could be combined:
print $twig ->root->first_child_text('title'), "\n";

## You can use the 'children' method to iterate multiple items:
my $list = $twig->root->first_child('list');

#children can optionally take an element 'tag' - otherwise it just returns all of them.
foreach my $element ( $list->children ) {

   #the 'att' method reads an attribute
   print "Element with ID: ", $element->att('id') // 'none here', " is ", $element->text,
     "\n";
}

#And if we need to do something more complicated, we an use 'xpath'.
#get_xpath or findnodes do the same thing:
#return a list of matches, or if you specify a second numeric argument, just that numbered match.

#xpath syntax is fairly extensive, but in this one - we search:
# anywhere in the tree: //
#nodes called 'item'
#with an id attribute [@id]
#and with that id attribute equal to "1000".
#by specifying '0' we say 'return just the first match'.

print "Item 1000 is: ", $twig->get_xpath( '//item[@id="1000"]', 0 )->text, "\n";

#this combines quite well with `map` to e.g. do the same thing on multiple items
print "All IDs:\n", join ( "\n", map { $_ -> att('id') } $twig -> get_xpath('//item'));
#note how this also finds the item under 'summary', because of //

__DATA__
<?xml version="1.0" encoding="utf-8"?>
<root>
  <title>some sample xml</title>
  <first key="value" key2="value2">
```

```
    <second>一些文本</second>
  </first>
  <third>
    <fourth key3="value">这里也是文本</fourth>
  </third>
  <list>
      <item id="1">项目1</item>
      <item id="2">项目2</item>
      <item id="3">项目3</item>
      <item id="66">项目66</item>
      <item id="88">项目88</item>
      <item id="100">项目100</item>
      <item id="1000">项目1000</item>
      <notanitem>实际上根本不是项目.</</notanitem>
  </list>
  <summary>
      <item id="no_id">测试</item>
  </summary>
</root>
```

第19.2节：使用XML::Rabbit处理XML

使用XML::Rabbit可以轻松处理XML文件。您可以以声明式方式并使用XPath语法定义您在XML中查找的内容，XML::Rabbit将根据给定的定义返回对象。

定义：

```
package Bookstore;
use XML::Rabbit::Root;
has_xpath_object_list books => './book' => 'Bookstore::Book';
finalize_class();

package Bookstore::Book;
use XML::Rabbit;
has_xpath_value bookid => './@id';
has_xpath_value author => './author';
has_xpath_value title => './title';
has_xpath_value genre => './genre';
has_xpath_value price => './price';
has_xpath_value publish_date => './publish_date';
has_xpath_value description => './description';
has_xpath_object purchase_data => './purchase_data' => 'Bookstore::Purchase';
finalize_class();

package Bookstore::Purchase;
use XML::Rabbit;
has_xpath_value price => './price';
has_xpath_value date => './date';
finalize_class();
```

XML 消费：

```
use strict;
use warnings;
use utf8;

package Library;
use feature qw(say);
use Carp;
use autodie;
```

```
    <second>Some text</second>
  </first>
  <third>
    <fourth key3="value">Text here too</fourth>
  </third>
  <list>
      <item id="1">Item1</item>
      <item id="2">Item2</item>
      <item id="3">Item3</item>
      <item id="66">Item66</item>
      <item id="88">Item88</item>
      <item id="100">Item100</item>
      <item id="1000">Item1000</item>
      <notanitem>Not an item at all really.</notanitem>
  </list>
  <summary>
      <item id="no_id">Test</item>
  </summary>
</root>
```

Section 19.2: Consuming XML with XML::Rabbit

With XML::Rabbit it is possible to consume XML files easily. You *define* in a declarative way and with an XPath syntax what you are looking for in the XML and XML::Rabbit will return objects according to the given definition.

Definition:

```
package Bookstore;
use XML::Rabbit::Root;
has_xpath_object_list books => './book' => 'Bookstore::Book';
finalize_class();

package Bookstore::Book;
use XML::Rabbit;
has_xpath_value bookid => './@id';
has_xpath_value author => './author';
has_xpath_value title => './title';
has_xpath_value genre => './genre';
has_xpath_value price => './price';
has_xpath_value publish_date => './publish_date';
has_xpath_value description => './description';
has_xpath_object purchase_data => './purchase_data' => 'Bookstore::Purchase';
finalize_class();

package Bookstore::Purchase;
use XML::Rabbit;
has_xpath_value price => './price';
has_xpath_value date => './date';
finalize_class();
```

XML Consumption:

```
use strict;
use warnings;
use utf8;

package Library;
use feature qw(say);
use Carp;
use autodie;
```

```
say "显示数据信息";
my $bookstore = Bookstore->new( file => './sample.xml' );

foreach my $book( @{$bookstore->books} ) {
     say "ID: " . $book->bookid;say "
     Title: " . $book->title;say "Author: "
      . $book->author, "";
}
```

备注：

请注意以下事项：

1. 第一个类必须是XML::Rabbit::Root。它会将你置于XML文档的主标签内。在我们的情况下，它会将我们置于<catalog>
2. 嵌套类是可选的。这些类需要通过try/catch（或 eval / $@检查）块访问。可选字段将简单返回null。例如，对于purchase_data，循环将是：

```
foreach my $book( @{$bookstore->books} ) {
     say "ID: " . $book->bookid;
     say "Title: " . $book->title;
     say "Author: " . $book->author;
try {
           say "购买价格: ". $book->purchase_data->price, "";} catch {

           say "无可用购买价格";}

}
```

sample.xml

```
<?xml version="1.0"?>
<catalog>
    <book id="bk101">
        <author>甘巴戴拉，马修</author>
        <title>XML开发者指南</title>
        <genre>计算机</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>深入探讨使用XML创建应用程序
      的方法。</description>
    </book>
    <book id="bk102">
        <author>拉尔斯，金</author>
        <title>午夜雨</title>
        <genre>奇幻</genre>
        <price>5.95</price>
        <publish_date>2000-12-16</publish_date>
        <description>一名前建筑师与企业僵尸、邪恶女巫以及她自己的童年抗争，最
终成为世界女王。</description>

    </book>
    <book id="bk103">
        <author>科雷茨，伊娃</author>
        <title>梅芙崛起</title>
        <genre>奇幻</genre>
        <price>5.95</price>
        <publish_date>2000-11-17</publish_date>
```

```
say "Showing data information";
my $bookstore = Bookstore->new( file => './sample.xml' );

foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author, "\n";
}
```

Notes:

Please be careful with the following:

1. The first class has to be `XML::Rabbit::Root`. It will place you inside the main tag of the XML document. In our case it will place us inside **`<catalog>`**
2. Nested classes which are optional. Those classes need to be accessed via a try/catch (or `eval / $@` check) block. Optional fields will simply return `null`. For example, for `purchase_data` the loop would be:

```
foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author;
    try {
        say "Purchase price: ". $book->purchase_data->price, "\n";
    } catch {
        say "No purchase price available\n";
    }
}
```

sample.xml

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
```

```
        <description>在英格兰纳米技术社会崩溃后，年轻的幸存者们为新社
会奠定了基础。</description>

    </book>
    <book id="bk104">
        <author>科雷茨，伊娃</author>
        <title>奥伯伦的遗产</title>
        <genre>奇幻</genre>
        <price>5.95</price>
        <publish_date>2001-03-10</publish_date>
        <description>在后末日的英格兰，神秘特工奥伯伦帮助伦敦居民创造新生
活。续集于梅芙崛起。

</description>
        <purchase_data>
          <date>2001-12-21</date>
          <price>20</price>
        </purchase_data>
    </book>
```

第19.3节：使用XML::LibXML进行解析

```
# 这使用了 XML::Twig 示例中给出的 'sample.xml' 文件。

# 模块要求（使用 load_xml 需 1.70 及以上版本）
use XML::LibXML '1.70';

# 让我们成为一个优秀的 Perl 开发者
use strict;
use warnings 'all';

# 创建 LibXML 文档对象
my $xml = XML::LibXML->new();

# 我们从哪里获取XML
my $file = 'sample.xml';

# 从文件加载XML
my $dom = XML::LibXML->load_xml(
    location => $file
);

# 获取文档根元素
my $root = $dom->getDocumentElement;

# 如果文档有子节点
if($root->hasChildNodes) {

    # getElementsByLocalName 返回所有本地名称匹配 'title' 的元素节点列表，
    # 我们想要第一个匹配项
    # （通过 get_node(1)）
    my $title = $root->getElementsByLocalName('title');

    if(defined $title) {
        # 从节点列表中获取第一个匹配的节点
        my $node = $title->get_node(1);

        # 获取目标节点的文本内容
        my $title_text = $node->textContent;
```

```
        <description>After the collapse of a nanotechnology
        society in England, the young survivors lay the
        foundation for a new society.</description>
    </book>
    <book id="bk104">
        <author>Corets, Eva</author>
        <title>Oberon's Legacy</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2001-03-10</publish_date>
        <description>In post-apocalypse England, the mysterious
        agent known only as Oberon helps to create a new life
        for the inhabitants of London. Sequel to Maeve
        Ascendant.</description>
        <purchase_data>
          <date>2001-12-21</date>
          <price>20</price>
        </purchase_data>
    </book>
</catalog>
```

Section 19.3: Parsing with XML::LibXML

```
# This uses the 'sample.xml' given in the XML::Twig example.

# Module requirements (1.70 and above for use of load_xml)
use XML::LibXML '1.70';

# let's be a good perl dev
use strict;
use warnings 'all';

# Create the LibXML Document Object
my $xml = XML::LibXML->new();

# Where we are retrieving the XML from
my $file = 'sample.xml';

# Load the XML from the file
my $dom = XML::LibXML->load_xml(
    location => $file
);

# get the docroot
my $root = $dom->getDocumentElement;

# if the document has children
if($root->hasChildNodes) {

    # getElementsByLocalName returns a node list of all elements who's
    # localname matches 'title', and we want the first occurrence
    # (via get_node(1))
    my $title = $root->getElementsByLocalName('title');

    if(defined $title) {
        # Get the first matched node out of the nodeList
        my $node = $title->get_node(1);

        # Get the text of the target node
        my $title_text = $node->textContent;
        print "名称为 'title' 的第一个节点包含：$title_text";}


    # 上述调用可以合并，但可能容易出错
    # （如果 getElementsByLocalName() 未能匹配到节点）。
    #
    # my $title_text = $root->getElementsByLocalName('title')->get_node(1)->textContent;
}

# 使用 Xpath，获取 id 为 'bk104' 的书的价格
#

# 设置我们的 xpath
my $xpath = q!/catalog/book[@id='bk104']/price!;

# 该 xpath 是否存在？
if($root->exists($xpath)) {

    # 拉入twig
    my $match = $root->find($xpath);

    if(defined $match) {
        # 从节点列表中获取第一个匹配的节点
        my $node = $match->get_node(1);

        # 获取该节点的文本内容
        my $match_text = $node->textContent;print

         "书籍ID为 bk104 的价格是: $match_text";}

}
```

```
        print "The first node with name 'title' contains: $title_text\n";
    }

    # The above calls can be combined, but is possibly prone to errors
    # (if the getElementsByLocalName() failed to match a node).
    #
    # my $title_text = $root->getElementsByLocalName('title')->get_node(1)->textContent;
}

# Using Xpath, get the price of the book with id 'bk104'
#

# Set our xpath
my $xpath = q!/catalog/book[@id='bk104']/price!;

# Does that xpath exist?
if($root->exists($xpath)) {

    # Pull in the twig
    my $match = $root->find($xpath);

    if(defined $match) {
        # Get the first matched node out of the nodeList
        my $node = $match->get_node(1);

        # pull in the text of that node
        my $match_text = $node->textContent;

        print "The price of the book with id bk104 is: $match_text\n";
    }
}
```

第20章：Unicode

第20.1节：utf8 pragma：在源代码中使用Unicode

utf8 pragma 表示源代码将被解释为UTF-8编码。当然，这只有在你的
文本编辑器也将源代码保存为UTF-8编码时才有效。

现在，字符串字面量可以包含任意Unicode字符；标识符也可以包含Unicode，但仅限于类似单词的
字符（更多信息请参见perldata和perlrecharclass）：

```
use utf8;
my $var1 = '§?§©????';        # 正常工作
my $? = 4;                   # 正常工作，因为 ? 是单词（匹配 |w）字符
my $p§2 = 3;                 # 不工作，因为 § 不是单词字符。
说 "ya" 如果 $var1 =~ /?§/; # 正常工作（打印 "ya"）
```

注意：在终端打印文本时，确保其支持 UTF-8。*

输出编码和源编码之间可能存在复杂且反直觉的关系。在
UTF-8 终端上运行时，你可能会发现添加 `utf8` pragma 似乎会导致问题：

```
$ perl -e 'print "Møøse"'Møøse

$ perl -Mutf8 -e 'print "Møøse"'M??se

$ perl -Mutf8 -CO -e 'print "Møøse"'Møøse
```

在第一种情况下，Perl 将字符串视为原始字节并直接打印。由于这些字节恰好是有效的
UTF-8，即使 Perl 并不真正知道它们代表什么字符（例如 `length("Møøse")` 会
返回 7，而不是 5），它们看起来是正确的。一旦你添加了 `-Mutf8`，Perl 会正确地将 UTF-8 源解码为字符，但默认情况下输出是 Latin-1
模式，向 UTF-8 终端打印 Latin-1 会失败。只有当你使用
-CO 将 STDOUT 切换为 UTF-8 时，输出才会正确。

use utf8 不影响标准输入输出编码和文件句柄！

第20.2节：处理无效的 UTF-8

读取无效的 UTF-8

读取 UTF-8 编码数据时，重要的是要注意 UTF-8 编码的数据可能是无效或格式错误的。通常情况下，您的程序不应接受此类数据（除非您知道自己在做什么）。
当意外遇到格式错误的数据时，可以考虑不同的处理方式：

- 打印堆栈跟踪或错误信息，并优雅地终止程序，或者
- 在出现格式错误的字节序列处插入替代字符，向标准错误输出打印警告
  信息，并继续读取，仿佛什么都没发生一样。

默认情况下，Perl 会warn你有关编码问题，但不会终止程序。你可以通过使 UTF-8 警告变为致命来让程序终止
，但请注意Fatal Warnings中的注意事项。

下面的示例将 3 个字节以 ISO 8859-1 编码写入磁盘。然后尝试将这些字节作为
UTF-8 编码的数据重新读取。其中一个字节0xE5是无效的 UTF-8 单字节序列：

Chapter 20: Unicode

Section 20.1: The utf8 pragma: using Unicode in your sources

The `utf8` pragma indicates that the source code will be interpreted as UTF-8. Of course, this will only work if your text editor is also saving the source as UTF-8 encoded.

Now, string literals can contain arbitrary Unicode characters; identifiers can also contain Unicode but only word-like characters (see perldata and perlrecharclass for more information):

```
use utf8;
my $var1 = '§?§©????';       # works fine
my $? = 4;                   # works since ? is a word (matches \w) character
my $p§2 = 3;                 # does not work since § is not a word character.
say "ya" if $var1 =~ /?§/; # works fine (prints "ya")
```

Note: When printing text to the terminal, make sure it supports UTF-8.*

There may be complex and counter-intuitive relationships between output and source encoding. Running on a UTF-8 terminal, you may find that adding the `utf8` pragma seems to break things:

```
$ perl -e 'print "Møøse\n"'
Møøse
$ perl -Mutf8 -e 'print "Møøse\n"'
M??se
$ perl -Mutf8 -CO -e 'print "Møøse\n"'
Møøse
```

In the first case, Perl treats the string as raw bytes and prints them like that. As these bytes happen to be valid UTF-8, they look correct even though Perl doesn't really know what characters they are (e.g. `length("Møøse")` will return 7, not 5). Once you add `-Mutf8`, Perl correctly decodes the UTF-8 source to characters, but output is in Latin-1 mode by default and printing Latin-1 to a UTF-8 terminal doesn't work. Only when you switch `STDOUT` to UTF-8 using `-CO` will the output be correct.

`use utf8` doesn't affect standard I/O encoding nor file handles!

Section 20.2: Handling invalid UTF-8

Reading invalid UTF-8

When reading UTF-8 encoded data, it is important to be aware of the fact the UTF-8 encoded data can be invalid or malformed. Such data should usually not be accepted by your program (unless you know what you are doing). When unexpectedly encountering malformed data, different actions can be considered:

- Print stacktrace or error message, and abort program gracefully, or
- Insert a substitution character at the place where the malformed byte sequence appeared, print a warning message to STDERR and continue reading as nothing happened.

By default, Perl will `warn` you about encoding glitches, but it will not abort your program. You can make your program abort by making UTF-8 warnings fatal, but be aware of the caveats in Fatal Warnings.

The following example writes 3 bytes in encoding ISO 8859-1 to disk. It then tries to read the bytes back again as UTF-8 encoded data. One of the bytes, `0xE5`, is an invalid UTF-8 one byte sequence:

```
use strict;
use warnings;
use warnings FATAL => 'utf8';

binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
my $bytes = "\x{61}\x{E5}\x{61}";  # ISO 8859-1 中的 3 个字节：aåa
my $fn = 'test.txt';
open ( my $fh, '>:raw', $fn ) or die "无法打开文件 '$fn': $!";
print $fh $bytes;
close $fh;
open ( $fh, "<:encoding(utf-8)", $fn ) or die "无法打开文件 '$fn': $!";
my $str = do { local $/; <$fh> };
close $fh;
print "读取字符串: '$str'";
```

程序将因致命警告而中止：

```
utf8 "å" 不映射到 Unicode，位于 ./test.pl 第 10 行。
```

第10行是倒数第二行，错误发生在尝试从文件中读取一行时的 <$fh> 部分。

如果不将上述程序中的警告设为致命，Perl 仍会打印警告。但在这种情况下，它会尝试通过在流中插入四个字符 å 来从格式错误的字节 0xE5 中恢复，然后继续处理下一个字节。结果，程序将打印：

```
读取字符串: 'aåa'
```

第20.3节：单行命令的命令行开关

启用 utf8 pragma

为了在单行命令中启用 utf8 pragma，perl 解释器应使用 -Mutf8 选项调用：

```
perl -Mutf8 -E 'my $人 = "human"; say $人'
```

使用 -C 开关处理 Unicode

-C 命令行标志让您控制 Unicode 功能。它后面可以跟一系列选项字母。

标准输入输出

- I - STDIN 将使用 UTF-8
- O - STDOUT 将使用 UTF-8
- E - STDERR 将使用 UTF-8
- S - IOE 的简写，标准输入输出流将使用 UTF-8

```
echo "Ματαιότης ματαιοτήτων" | perl -CS -Mutf8 -nE 'say "ok" if /Ματαιότης/'
```

脚本参数

- A - 将 @ARGV 视为一组 UTF-8 编码的字符串数组

```
perl -CA -Mutf8 -E 'my $arg = shift; say "anteater" if $arg eq "муравьед"' муравьед
```

```
use strict;
use warnings;
use warnings FATAL => 'utf8';

binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
my $bytes = "\x{61}\x{E5}\x{61}";  # 3 bytes in iso 8859-1: aåa
my $fn = 'test.txt';
open ( my $fh, '>:raw', $fn ) or die "Could not open file '$fn': $!";
print $fh $bytes;
close $fh;
open ( $fh, "<:encoding(utf-8)", $fn ) or die "Could not open file '$fn': $!";
my $str = do { local $/; <$fh> };
close $fh;
print "Read string: '$str'\n";
```

The program will abort with a fatal warning:

```
utf8 "\xE5" does not map to Unicode at ./test.pl line 10.
```

Line 10 is here the second last line, and the error occurs in the part of the line with <$fh> when trying to read a line from the file.

If you don't make warnings fatal in the above program, Perl will still print the warning. However, in this case it will try to recover from the malformed byte 0xE5 by inserting the four characters \xE5 into the stream, and then continue with the next byte. As a result, the program will print:

```
Read string: 'a\xE5a'
```

Section 20.3: Command line switches for one-liners

Enable utf8 pragma

In order to enable utf8 pragma in one-liner, perl interpreter should be called with -Mutf8 option:

```
perl -Mutf8 -E 'my $人 = "human"; say $人'
```

Unicode handling with -C switch

The -C command line flag lets you control Unicode features. It can be followed by a list of option letters.

Standard I/O

- I - **STDIN** will be in *UTF-8*
- O - **STDOUT** will be in *UTF-8*
- E - **STDERR** will be in *UTF-8*
- S - shorthand for IOE, standard I/O streams will be in *UTF-8*

```
echo "Ματαιότης ματαιοτήτων" | perl -CS -Mutf8 -nE 'say "ok" if /Ματαιότης/'
```

Script's arguments

- A - treats @**ARGV** as an array of *UTF-8* encoded strings

```
perl -CA -Mutf8 -E 'my $arg = shift; say "anteater" if $arg eq "муравьед"' муравьед
```

默认 PerlIO 层

- i - *UTF-8* 是输入流的默认 PerlIO 层
- o - *UTF-8* 是输出流的默认 PerlIO 层
- D - io 的简写

```
perl -CD -Mutf8 -e 'open my $fh, ">", "utf8.txt" or die $!; print $fh "개미 조심해"'
```

-M 和 -C 开关可以组合使用：

```
perl -CASD -Mutf8 -E 'say "Ματαιότης ματαιοτήτων"';
```

第20.4节：标准输入输出

标准输入输出文件句柄（STDIN、STDOUT 和 STDERR）使用的编码，可以针对每个句柄单独设置，使用 binmode：

```
binmode STDIN, ':encoding(utf-8)';
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
```

注意：读取时通常更倾向于使用 :encoding(utf-8) 而非 :utf8，更多信息请参见备注。

或者，您可以使用open编译指示。

```
# 设置使得所有随后打开的输入流将使用 ':encoding(utf-8)'
# 所有随后打开的输出流将使用 ':utf8'
# 默认情况下
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# 使（已打开的）标准文件句柄继承由 open pragma 的 IO 设置给出的设置

use open ( :std );
# 现在，STDIN 已被转换为 ':encoding(utf-8)',
# STDOUT 和 STDERR 使用 ':utf8'
```

或者，为了设置所有文件句柄（包括尚未打开的和标准句柄）使用 :encoding(utf-8):

```
use open qw( :encoding(utf-8) :std );
```

第20.5节：文件句柄

使用 open() 设置编码

打开文本文件时，可以通过带有三个参数的 open() 明确指定其编码。附加到文件句柄上的这个编解码器称为“I/O 层”：

```
my $filename = '/path/to/file';
open my $fh, '<:encoding(utf-8)', $filename or die "无法打开 $filename: $!";
```

有关 :utf8 和 :encoding(utf-8) 之间差异的讨论，请参见备注。

使用 binmode() 设置编码

或者，可以使用 binmode() 为单个文件句柄设置编码：

Default PerlIO layer

- i - *UTF-8* is the default PerlIO layer for input streams
- o - *UTF-8* is the default PerlIO layer for output streams
- D - shorthand for `io`

```
perl -CD -Mutf8 -e 'open my $fh, ">", "utf8.txt" or die $!; print $fh "개미 조심해"'
```

`-M` and `-C` switches may be combined:

```
perl -CASD -Mutf8 -E 'say "Ματαιότης ματαιοτήτων\n"';
```

Section 20.4: Standard I/O

The encoding to be used for the standard I/O filehandles (**STDIN**, **STDOUT**, and **STDERR**), can be set separately for each handle using `binmode`:

```
binmode STDIN, ':encoding(utf-8)';
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
```

Note: when reading one would in general prefer `:encoding(utf-8)` over `:utf8`, see Remarks for more information.

Alternatively, you can use the `open` pragma.

```
# Setup such that all subsequently opened input streams will use ':encoding(utf-8)'
# and all subsequently opened output streams will use ':utf8'
# by default
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Make the (already opened) standard file handles inherit the setting
# given by the IO settings for the open pragma
use open ( :std );
# Now, STDIN has been converted to ':encoding(utf-8)', and
# STDOUT and STDERR have ':utf8'
```

Alternatively, to set all filehandles (both those yet to be opened and also the standard ones) to use `:encoding(utf-8)`:

```
use open qw( :encoding(utf-8) :std );
```

Section 20.5: File handles

Setting encoding with open()

When opening a text file, you may specify it's encoding explicitly with a three-argument `open()`. This en-/decoder attached to a file handle is called an "I/O layer":

```
my $filename = '/path/to/file';
open my $fh, '<:encoding(utf-8)', $filename or die "Failed to open $filename: $!";
```

See Remarks for a discussion of the differences between `:utf8` and `:encoding(utf-8)`.

Setting encoding with binmode()

Alternatively, you may use binmode() to set the encoding for individual file handle:

```
my $filename = '/path/to/file';
open my $fh, '<', $filename or die "无法打开 $filename: $!";
binmode $fh, ':encoding(utf-8)';
```

open 预处理指令

为了避免为每个文件句柄单独设置编码，可以使用 open 预处理指令设置默认的 I/O 层，该层将在此预处理指令的词法作用域内被所有后续对 open() 函数及类似操作符的调用使用：

```
# 将输入流设置为 ':encoding(utf-8)'，输出流设置为 ':utf8'
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# 或者将所有输入和输出流设置为 ':encoding(utf-8)'
use open ':encoding(utf-8)';
```

使用命令行 -C 标志设置编码

最后，也可以使用带有 -CD 标志的 perl 解释器，该标志将 UTF-8 作为默认的输入/输出层。
但是，应避免使用此选项，因为它依赖于无法预测或控制的特定用户行为。

第20.6节：创建文件名

以下示例使用 UTF-8 编码来表示磁盘上的文件名（和目录名）。如果您想使用其他编码，应使用 Encode::encode(...)
。

```
use v5.14;
# 使 Perl 识别字面字符串中的 UTF-8 编码字符。
# 要使其工作：确保您的文本编辑器使用 UTF-8，这样
# 磁盘上的字节才是真正的 UTF-8 编码。
use utf8;

# 确保可能打印到屏幕的错误信息被转换为 UTF-8。
# 要使其工作：请检查您的终端模拟器是否使用UTF-8编码。
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

my $filename = 'æ€'; # $filename 现在是一个内部UTF-8编码的字符串。

# 注意：以下假设 $filename 已设置内部UTF-8
# 标志，如果 $filename 是纯ASCII，也能正常工作，因为其编码
# 与UTF-8重叠。但是，如果它是其他编码，如扩展ASCII，
# $filename 将以该编码写入，而不是UTF-8。
# 注意：这里不需要将 $filename 编码为UTF-8
# 因为Perl已经将 $filename 作为内部UTF-8编码使用

# 示例1 -- 使用 open()
open ( my $fh, '>', $filename ) or die "无法打开 '$filename': $!";
close $fh;

# 示例2 -- 使用 qx() 和 touch
qx{touch $filename};

# 示例3 -- 使用 system() 和 touch
system 'touch', $filename;

# 示例4 -- 使用 File::Touch
use File::Touch;
eval { touch( $filename ) }; die "无法创建文件 '$filename': $!" if $@;
```

```
my $filename = '/path/to/file';
open my $fh, '<', $filename or die "Failed to open $filename: $!";
binmode $fh, ':encoding(utf-8)';
```

open pragma

To avoid setting encoding for each file handle separately, you may use the open pragma to set a default I/O layer used by all subsequent calls to the open() function and similar operators within the lexical scope of this pragma:

```
# Set input streams to ':encoding(utf-8)' and output streams to ':utf8'
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Or to set all input and output streams to ':encoding(utf-8)'
use open ':encoding(utf-8)';
```

Setting encoding with command line -C flag

Finally, it is also possible to run the perl interpreter with a -CD flag that applies UTF-8 as the default I/O layer. However, this option should be avoided since it relies on specific user behaviour which cannot be predicted nor controlled.

Section 20.6: Create filenames

The following examples use the UTF-8 encoding to represent filenames (and directory names) on disk. If you want to use another encoding, you should use Encode::encode(...).

```
use v5.14;
# Make Perl recognize UTF-8 encoded characters in literal strings.
# For this to work: Make sure your text-editor is using UTF-8, so
# that bytes on disk are really UTF-8 encoded.
use utf8;

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that your terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

my $filename = 'æ€'; # $filename is now an internally UTF-8 encoded string.

# Note: in the following it is assumed that $filename has the internal UTF-8
#  flag set, if $filename is pure ASCII, it will also work since its encoding
#  overlaps with UTF-8. However, if it has another encoding like extended ASCII,
#  $filename will be written with that encoding and not UTF-8.
# Note: it is not necessary to encode $filename as UTF-8 here
#  since Perl is using UTF-8 as its internal encoding of $filename already

# Example1  -- using open()
open ( my $fh, '>', $filename ) or die "Could not open '$filename': $!";
close $fh;

# Example2 -- using qx() and touch
qx{touch $filename};

# Example3 -- using system() and touch
system 'touch', $filename;

# Example4 -- using File::Touch
use File::Touch;
eval { touch( $filename ) }; die "Could not create file '$filename': $!" if $@;
```

第20.7节：读取文件名

Perl 不会尝试解码内置函数或模块返回的文件名。此类表示文件名的字符串应始终显式解码，以便 Perl 将其识别为 Unicode。

```
use v5.14;
use Encode qw(decode_utf8);

# 确保可能打印到屏幕的错误信息被转换为 UTF-8。
# 为使此方法生效：请检查您的终端模拟器是否使用 UTF-8。
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

# 示例1  -- 使用 readdir()
my $dir = '.';
opendir(my $dh, $dir) or die "无法打开目录 '$dir': $!";
while (my $filename = decode_utf8(readdir $dh)) {
    # 对 $filename 进行操作
}
close $dh;

# 示例2  -- 使用 getcwd()
use Cwd qw(getcwd);
my $dir = decode_utf8( getcwd() );

# 示例3  -- 使用 abs2rel()
use File::Spec;
use utf8;
my $base = 'ø';
my $path = "$base/b/æ";
my $relpath = decode_utf8( File::Spec->abs2rel( $path, $base ) );
# 注意：如果省略 $base，则需要先对 $path 进行编码：
use Encode qw(encode_utf8);
my $relpath = decode_utf8( File::Spec->abs2rel( encode_utf8( $path ) ) );

# 示例4  -- 使用 File::Find::Rule（第1部分，匹配文件名）
use File::Find::Rule;
use utf8;
use Encode qw(encode_utf8);
my $filename = 'æ';
# File::Find::Rule 需要 $filename 进行编码
my @files = File::Find::Rule->new->name( encode_utf8($filename) )->in('.');
$_ = decode_utf8( $_ ) for @files;

# 示例5  -- 使用 File::Find::Rule（第2部分，匹配正则表达式）
use File::Find::Rule;
use utf8;
my $pat = '[æ].$'; # Unicode 模式
# 注意：在这种情况下：  File::Find::Rule->new->name( qr/$pat/ )->in('.')
#  不会生效，因为 $pat 是 Unicode，而文件名是字节
# 同时先对 $pat 进行编码也不会正确工作
my @files;
File::Find::Rule->new->exec( sub { wanted( $pat, \@files ) } )->in('.');
$_ = decode_utf8( $_ ) for @files;
sub wanted {
    my ( $pat, $files ) = @_;
    my $name = decode_utf8( $_ );
    my $full_name = decode_utf8( $File::Find::name );
    push @$files, $full_name if $name =~ /$pat/;
}
```

Section 20.7: Read filenames

Perl does not attempt to decode filenames returned by builtin functions or modules. Such strings representing filenames should always be decoded explicitly, in order for Perl to recognize them as Unicode.

```
use v5.14;
use Encode qw(decode_utf8);

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that you terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

# Example1  -- using readdir()
my $dir = '.';
opendir(my $dh, $dir) or die "Could not open directory '$dir': $!";
while (my $filename = decode_utf8(readdir $dh)) {
    # Do something with $filename
}
close $dh;

# Example2  -- using getcwd()
use Cwd qw(getcwd);
my $dir = decode_utf8( getcwd() );

# Example3  -- using abs2rel()
use File::Spec;
use utf8;
my $base = 'ø';
my $path = "$base/b/æ";
my $relpath = decode_utf8( File::Spec->abs2rel( $path, $base ) );
# Note: If you omit $base, you need to encode $path first:
use Encode qw(encode_utf8);
my $relpath = decode_utf8( File::Spec->abs2rel( encode_utf8( $path ) ) );

# Example4  -- using File::Find::Rule (part1 matching a filename)
use File::Find::Rule;
use utf8;
use Encode qw(encode_utf8);
my $filename = 'æ';
# File::Find::Rule needs $filename to be encoded
my @files = File::Find::Rule->new->name( encode_utf8($filename) )->in('.');
$_ = decode_utf8( $_ ) for @files;

# Example5  -- using File::Find::Rule (part2 matching a regular expression)
use File::Find::Rule;
use utf8;
my $pat = '[æ].$'; # Unicode pattern
# Note: In this case:  File::Find::Rule->new->name( qr/$pat/ )->in('.')
#  will not work since $pat is Unicode and filenames are bytes
#  Also encoding $pat first will not work correctly
my @files;
File::Find::Rule->new->exec( sub { wanted( $pat, \@files ) } )->in('.');
$_ = decode_utf8( $_ ) for @files;
sub wanted {
    my ( $pat, $files ) = @_;
    my $name = decode_utf8( $_ );
    my $full_name = decode_utf8( $File::Find::name );
    push @$files, $full_name if $name =~ /$pat/;
}
```

注意：如果您担心文件名中存在无效的 UTF-8，上述示例中使用的 decode_utf8(...) 可能应该替换为 decode('utf-8', ...)。这是因为 decode_utf8(...) 是 decode('utf8', ...) 的同义词，而编码 utf-8 和 utf8 之间存在差异（详见下文备注），其中 utf-8 对可接受内容的限制比 utf8 更严格。

Note: if you are concerned about invalid UTF-8 in the filenames, the use of `decode_utf8( ... )` in the above examples should probably be replaced by `decode( 'utf-8', ... )`. This is because `decode_utf8( ... )` is a synonym for `decode( 'utf8', ... )` and there is a difference between the encodings `utf-8` and `utf8` (see Remarks below for more information) where `utf-8` is more strict on what is acceptable than `utf8`.

第21章：Perl 单行命令

第21.1节：将文件上传到 mojolicious

```
perl -Mojo -E 'p("http://localhost:3000" => form => {Input_Type => "XML", Input_File => {file =>
"d:/xml/test.xml"}})'
```

文件 d:/xml/test.xml 将被上传到监听 localhost:3000 端口连接的服务器（来源）

在此示例中：

-Mmodule 在执行程序前执行 use 模块；
-E 命令行 用于输入一行程序
如果您没有 ojo 模块，可以使用 cpanm ojo 命令安装它

要了解更多关于如何运行 perl，请使用 perldoc perlrun 命令或阅读 here

第21.2节：从命令行执行一些 Perl 代码

可以使用-e开关（意为"执行"）将简单的一行代码作为命令行参数传递给perl：

```
perl -e'print "Hello, World!"'
```

由于 Windows 的引用规则，不能使用单引号字符串，必须使用以下变体之一：

```
perl -e"print qq(Hello, World!)"perl -e"pri
nt \"Hello, World!\""
```

请注意，为了避免破坏旧代码，使用 -e 时只能使用 Perl 5.8.x 及以前版本支持的语法。要使用您的 Perl 版本可能支持的更新语法，请改用 -E。例如，要使用从 5.10.0 开始提供的 say 以及从 >=v5.14.0 开始支持的 Unicode 6.0（还使用 -CO 确保 STDOUT 以 UTF-8 编码输出）：

版本 ≥ 5.14.0

```
perl -CO -E'say "\N{PILE OF POO}"'
```

第 21.3 节：在 Windows 单行命令中使用双引号字符串

Windows 只使用双引号来包裹命令行参数。为了在 Perl 单行命令中使用双引号（即打印带有插值变量的字符串），必须用反斜杠转义它们：

```
perl -e "my $greeting = 'Hello'; print \"$greeting, world!\""
```

为了提高可读性，可以使用 qq() 操作符：

```
perl -e "my $greeting = 'Hello'; print qq($greeting, world!)"
```

第21.4节：打印匹配模式的行（PCRE grep）

```
perl -ne'print if /foo/' file.txt
```

不区分大小写：

Chapter 21: Perl one-liners

Section 21.1: Upload file into mojolicious

```
perl -Mojo -E 'p("http://localhost:3000" => form => {Input_Type => "XML", Input_File => {file =>
"d:/xml/test.xml"}})'
```

File d:/xml/test.xml will be uploaded to server which listen connections on localhost:3000 (Source)

In this example:

-Mmodule executes **use** module; before executing your program
-E commandline is used to enter one line of program
If you have no ojo module you can use cpanm ojo command to install it

To read more about how to run perl use perldoc perlrun command or read here

Section 21.2: Execute some Perl code from command line

Simple one-liners may be specified as command line arguments to perl using the -e switch (think "execute"):

```
perl -e'print "Hello, World!\n"'
```

Due to Windows quoting rules you can't use single-quoted strings but have to use one of these variants:

```
perl -e"print qq(Hello, World!\n)"
perl -e"print \"Hello, World!\n\""
```

Note that to avoid breaking old code, only syntax available up to Perl 5.8.x can be used with -e. To use anything newer your perl version may support, use -E instead. E.g. to use say available from 5.10.0 on plus Unicode 6.0 from >=v5.14.0 (also uses -CO to make sure **STDOUT** prints UTF-8):

Version ≥ 5.14.0

```
perl -CO -E'say "\N{PILE OF POO}"'
```

Section 21.3: Using double-quoted strings in Windows one-liners

Windows uses only double quotes to wrap command line parameters. In order to use double quotes in perl one-liner (i.e. to print a string with an interpolated variable), you have to escape them with backslashes:

```
perl -e "my $greeting = 'Hello'; print \"$greeting, world!\n\""
```

To improve readability, you may use a qq() operator:

```
perl -e "my $greeting = 'Hello'; print qq($greeting, world!\n)"
```

Section 21.4: Print lines matching a pattern (PCRE grep)

```
perl -ne'print if /foo/' file.txt
```

Case-insensitive:

```
perl -ne'print if /foo/i' file.txt
```

第21.5节：用另一个子串替换（PCRE sed）

```
perl -pe"s/foo/bar/g" file.txt
```

或者就地替换：

```
perl -i -pe's/foo/bar/g' file.txt
```

在Windows上：

```
perl -i.bak -pe"s/foo/bar/g" file.txt
```

第21.6节：仅打印特定字段

```
perl -lane'print "$F[0] $F[-1]"' data.txt
# 打印以空格分隔记录的第一个和最后一个字段
```

CSV 示例：

```
perl -F, -lane'print "$F[0] $F[-1]"' data.csv
```

第21.7节：打印第5行到第10行

```
perl -ne'print if 5..10' file.txt
```

第21.8节：就地编辑文件

无备份副本（Windows 不支持）

```
perl -i -pe's/foo/bar/g' file.txt
```

有备份副本 file.txt.bak

```
perl -i.bak -pe's/foo/bar/g' file.txt
```

With a backup copy old_file.txt.orig in the backup subdirectory (provided the latter exists):

```
perl -i'backup/old_*.orig' -pe's/foo/bar/g' file.txt
```

第21.9节：将整个文件作为字符串读取

```
perl -0777 -ne'print "整个文件作为字符串: --->$_<---"'
```

注意：-0777 只是一个约定。任何 -0400 及以上的值效果相同。

```
perl -ne'print if /foo/i' file.txt
```

Section 21.5: Replace a substring with another (PCRE sed)

```
perl -pe"s/foo/bar/g" file.txt
```

Or in-place:

```
perl -i -pe's/foo/bar/g' file.txt
```

On Windows:

```
perl -i.bak -pe"s/foo/bar/g" file.txt
```

Section 21.6: Print only certain fields

```
perl -lane'print "$F[0] $F[-1]"' data.txt
# prints the first and the last fields of a space delimited record
```

CSV example:

```
perl -F, -lane'print "$F[0] $F[-1]"' data.csv
```

Section 21.7: Print lines 5 to 10

```
perl -ne'print if 5..10' file.txt
```

Section 21.8: Edit file in-place

Without a backup copy (not supported on Windows)

```
perl -i -pe's/foo/bar/g' file.txt
```

With a backup copy `file.txt.bak`

```
perl -i.bak -pe's/foo/bar/g' file.txt
```

With a backup copy `old_file.txt.orig` in the `backup` subdirectory (provided the latter exists):

```
perl -i'backup/old_*.orig' -pe's/foo/bar/g' file.txt
```

Section 21.9: Reading the whole file as a string

```
perl -0777 -ne'print "The whole file as a string: --->$_<---\n"'
```

Note: The `-0777` is just a convention. Any `-0400` and above would de the same.

第22章：随机性

第22.1节：随机访问数组元素

```
my @letters = ( 'a' .. 'z' );                   # 英文字母表

print $letters[ rand @letters ] for 1 .. 5;   # 随机打印5个字母
```

工作原理

- rand EXPR 期望一个标量值，因此 @letters 在标量上下文中被求值标量上下文中的数
- 组返回其包含的元素数量（此处为26）rand 26 返回一个位于区间 0 ≤ VALUE < 26 的随机小数
- 。（它永远不会是 26）数组索引总是整数，因此 $letters[rand @letters] 等价于 $letters[int rand @letters]
- Perl数组是从零开始索引的，所以 $array[rand @array] 返回 $array[0]、$array[$#array] 或者介于两者之间
- 的元素

（同样的原理适用于哈希）

```
my %colors = ( red   => 0xFF0000,
               green => 0x00FF00,
               blue  => 0x0000FF,
             );

print ( values %colors )[rand keys %colors];
```

第22.2节：生成一个0到9之间的随机整数

将你的随机浮点数转换为整数。

输入：

```
my $range = 10;

# 创建一个最低为0，最高为9的随机整数
my $random = int(rand($range));    # 最大值小于但不等于$rangeprint $random . "";
```

输出：

一个随机整数，比如……

```
0
```

另请参见 rand 的 perldoc。

Chapter 22: Randomness

Section 22.1: Accessing an array element at random

```
my @letters = ( 'a' .. 'z' );                   # English ascii-bet

print $letters[ rand @letters ] for 1 .. 5;   # prints 5 letters at random
```

How it works

- rand EXPR expects a scalar value, so @letters is evaluated in scalar context
- An array in scalar context returns the number of elements it contains (26 in this case)
- rand 26 returns a random fractional number in the interval 0 ≤ VALUE < 26. (It can never be 26)
- Array indices are always integers, so $letters[rand @letters] ≡ $letters[int rand @letters]
- Perl arrays are zero-indexed, so $array[rand @array] returns $array[0], $array[$#array] or an element in between

(The same principle applies to hashes)

```
my %colors = ( red   => 0xFF0000,
               green => 0x00FF00,
               blue  => 0x0000FF,
             );

print ( values %colors )[rand keys %colors];
```

Section 22.2: Generate a random integer between 0 and 9

Cast your random floating-point number as an int.

Input:

```
my $range = 10;

# create random integer as low as 0 and as high as 9
my $random = int(rand($range));   # max value is up to but not equal to $range

print $random . "\n";
```

Output:

A random integer, like...

```
0
```

See also the perldoc for rand.

第23章：特殊变量

23.1节：Perl中的特殊变量：

1. $_ ：默认的输入和模式搜索空间。

示例1：

```
my @array_variable = (1 2 3 4);
foreach (@array_variable){
                    print $_."";    # 如果没有提供其他变量，$_ 在循环中将依次获得值1,2,3,4。
}
```

示例2：

```
while (<FH>){
    chomp($_);    # $_ 指的是循环中迭代的行。
}
```

以下函数使用$_作为默认参数：

```
abs, alarm, chomp, chop, chr, chroot, cos, defined, eval,
evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log,
lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink,
readpipe, ref, require, reverse (in scalar context only), rmdir,
say, sin, split (for its second argument), sqrt, stat, study,
uc, ucfirst, unlink, unpack。
```

2. @_ ：该数组包含传递给子程序的参数。

示例1：

```
example_sub( $test1, $test2, $test3 );

sub example_sub {
    my ( $test1, $test2, $test3 ) = @_;
}
```

在子程序内，数组@_包含传递给该子程序的参数。在子程序内部，@_是数组操作符pop和shift的默认数组。

Chapter 23: Special variables

Section 23.1: Special variables in perl:

1. $_ : The default input and pattern-searching space.

Example 1:

```
my @array_variable = (1 2 3 4);
foreach (@array_variable){
    print $_."\n";    # $_ will get the value 1,2,3,4 in loop, if no other variable is supplied.
}
```

Example 2:

```
while (<FH>){
    chomp($_);    # $_ refers to the iterating lines in the loop.
}
```

The following functions use $_ as a default argument:

```
abs, alarm, chomp, chop, chr, chroot, cos, defined, eval,
evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log,
lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink,
readpipe, ref, require, reverse (in scalar context only), rmdir,
say, sin, split (for its second argument), sqrt, stat, study,
uc, ucfirst, unlink, unpack.
```

2. @_ : This array contains the arguments passed to subroutine.

Example 1:

```
example_sub( $test1, $test2, $test3 );

sub example_sub {
    my ( $test1, $test2, $test3 ) = @_;
}
```

Within a subroutine the array `@_` contains the **arguments** passed to that subroutine. Inside a subroutine, `@_` is the `default` array for the array operators `pop` and `shift`.

第24章：包和模块

第24.1节：使用模块

```
use Cwd;
```

这将在编译时导入Cwd模块并导入其默认符号，即使该模块的一些变量和函数可供使用它的代码调用。（另见：perldoc-fuse。）

通常这将做正确的事情。然而，有时你会想控制导入哪些符号。在模块名后添加符号列表以导出：

```
use Cwd 'abs_path';
```

如果这样做，只有你指定的符号会被导入（即默认集合不会被导入）。

导入多个符号时，惯用做法是使用qw()列表构造：

```
use Cwd qw(abs_path realpath);
```

有些模块导出其符号的子集，但可以通过:all:告诉它们导出所有符号：

```
use Benchmark ':all';
```

（注意，并非所有模块都识别或使用:all标签）。

第24.2节：在目录中使用模块

```
use lib 'includes';
use MySuperCoolModule;
```

`use lib 'includes';` 将相对目录 `includes/` 添加为 `@INC` 中的另一个模块搜索路径。假设你有一个模块文件 `MySuperCoolModule.pm` 位于 `includes/` 中，内容如下：

```
package MySuperCoolModule;
```

如果你愿意，可以将多个自定义模块放在同一个目录中，并通过一个 `use lib` 语句使它们可被找到。

此时，使用模块中的子程序需要在子程序名称前加上包名前缀：

```
MySuperCoolModule::SuperCoolSub_1("Super Cool String");
```

若想在调用时不加前缀使用子程序，需要导出子程序名，使调用程序能够识别它们。导出可以设置为自动，方法如下：

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

然后在使用该模块的文件中，这些子程序将自动可用：

Chapter 24: Packages and modules

Section 24.1: Using a module

```
use Cwd;
```

This will import the `Cwd` module at compile time and import its default symbols, i.e. make some of the module's variables and functions available to the code using it. (See also: `perldoc -f` **use**.)

Generally this is will do the right thing. Sometimes, however, you will want to control which symbols are imported. Add a list of symbols after the module name to export:

```
use Cwd 'abs_path';
```

If you do this, only the symbols you specify will be imported (ie, the default set will not be imported).

When importing multiple symbols, it is idiomatic to use the `qw()` list-building construct:

```
use Cwd qw(abs_path realpath);
```

Some modules export a subset of their symbols, but can be told to export everything with `:all`:

```
use Benchmark ':all';
```

(Note that not all modules recognize or use the `:all` tag).

Section 24.2: Using a module inside a directory

```
use lib 'includes';
use MySuperCoolModule;
```

`use lib 'includes';` adds the relative directory `includes/` as another module search path in `@INC`. So assume that you have a module file `MySyperCoolModule.pm` inside `includes/`, which contains:

```
package MySuperCoolModule;
```

If you want, you can group as many modules of your own inside a single directory and make them findable with one **use** `lib` statement.

At this point, using the subroutines in the module will require prefixing the subroutine name with the package name:

```
MySuperCoolModule::SuperCoolSub_1("Super Cool String");
```

To be able to use the subroutines without the prefix, you need to export the subroutine names so that they are recognised by the program calling them. Exporting can be set up to be automatic, thus:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

Then in the file that **use**s the module, those subroutines will be automatically available:

```
use MySuperCoolModule;
SuperCoolSub_1("Super Cool String");
```

或者你可以设置模块以有条件地导出子程序，方法如下：

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT_OK = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

在这种情况下，你需要在使用该模块的脚本中显式请求导出所需的子程序：

```
use MySuperCoolModule 'SuperCoolSub_1';
SuperCoolSub_1("Super Cool String");
```

第24.3节：在运行时加载模块

```
require Exporter;
```

这将确保如果尚未导入，Exporter 模块将在运行时被加载。（另见：perldoc -f require。）

注意：大多数用户应使用use模块而非require。与use不同，require不会调用模块的import方法，且是在运行时执行，而非编译时。

这种加载模块的方式在你无法在运行前确定需要哪些模块时非常有用，比如使用插件系统时：

```
package My::Module;
my @plugins = qw( One Two );
foreach my $plugin (@plugins) {
    my $module = __PACKAGE__ . "::Plugins::$plugin";
    $module =~ s!::!/!g;
    require "$module.pm";
}
```

这将尝试加载My::Package::Plugins::One 和My::Package::Plugins::Two。 @plugins 当然应该来自某些用户输入或配置文件，这样才有意义。注意替换操作符 s!::!/!g，它将每对冒号替换为斜杠。这是因为你只能使用熟悉的模块名称语法通过 use 加载模块，前提是模块名称是裸词。如果你传递的是字符串或变量，则它必须包含文件名。

第24.4节：CPAN.pm

CPAN.pm 是一个Perl模块，允许从CPAN站点查询和安装模块。

它支持通过以下命令调用的交互模式

```
cpan
```

或

```
perl -MCPAN -e shell
```

```
use MySuperCoolModule;
SuperCoolSub_1("Super Cool String");
```

Or you can set up the module to conditionally export subroutines, thus:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT_OK = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

In which case, you need to explicitly request the desired subroutines to be exported in the script that **use**s the module:

```
use MySuperCoolModule 'SuperCoolSub_1';
SuperCoolSub_1("Super Cool String");
```

Section 24.3: Loading a module at runtime

```
require Exporter;
```

This will ensure that the `Exporter` module is loaded at runtime if it hasn't already been imported. (See also: `perldoc -f require`.)

N.B.: Most users should **use** modules rather than `require` them. Unlike **use**, `require` does not call the module's import method and is executed at runtime, not during the compile.

This way of loading modules is useful if you can't decide what modules you need before runtime, such as with a plugin system:

```
package My::Module;
my @plugins = qw( One Two );
foreach my $plugin (@plugins) {
    my $module = __PACKAGE__ . "::Plugins::$plugin";
    $module =~ s!::!/!g;
    require "$module.pm";
}
```

This would try to load `My::Package::Plugins::One` and `My::Package::Plugins::Two`. `@plugins` should of course come from some user input or a config file for this to make sense. Note the substitution operator `s!::!/!g` that replaces each pair of colons with a slash. This is because you can load modules using the familiar module name syntax from **use** only if the module name is a bareword. If you pass a string or a variable, it must contain a file name.

Section 24.4: CPAN.pm

`CPAN.pm` is a Perl module which allows to query and install modules from CPAN sites.

It supports interactive mode invoked with

```
cpan
```

or

```
perl -MCPAN -e shell
```

查询模块

按名称查询：

```
cpan> m MooseX::YAML
```

按模块名称的正则表达式查询：

```
cpan> m /^XML::/
```

注意：要启用分页器或重定向到文件，请使用|或>的shell重定向（|和>两边必须有空格），例如：m /^XML::/ | less.

按发行版查询：

```
cpan> d LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

安装模块

按名称查询：

```
cpan> install MooseX::YAML
```

按发行版查询：

```
cpan> 安装 LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

第24.5节：列出所有已安装的模块

从命令行：

```
cpan -l
```

从 Perl 脚本：

```
use ExtUtils::Installed;
my $inst = ExtUtils::Installed->new();
my @modules = $inst->modules();
```

第24.6节：执行另一个文件的内容

```
do './config.pl';
```

这将读取 config.pl 文件的内容并执行它。（另见：`perldoc -f do`。）

注意： 除非是为了代码简洁，否则避免使用 do，因为它没有错误检查。对于包含库模块，请使用 `require` 或 **use**。

Querying modules

By name:

```
cpan> m MooseX::YAML
```

By a regex against module name:

```
cpan> m /^XML::/
```

Note: to enable a pager or redirecting to a file use `|` or `>` shell redirection (spaces are mandatory around the `|` and `>`), e.g.: `m /^XML::/ | less`.

By distribution:

```
cpan> d LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Installing modules

By name:

```
cpan> install MooseX::YAML
```

By distribution:

```
cpan> install LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Section 24.5: List all installed modules

From command line:

```
cpan -l
```

From a Perl script:

```
use ExtUtils::Installed;
my $inst = ExtUtils::Installed->new();
my @modules = $inst->modules();
```

Section 24.6: Executing the contents of another file

```
do './config.pl';
```

This will read in the contents of the config.pl file and execute it. (See also: `perldoc -f do`.)

N.B.: Avoid `do` unless golfing or something as there is no error checking. For including library modules, use `require` or **use**.

第25章：通过 CPAN 安装 Perl 模块

第25.1节：cpanminus，轻量级的无配置替代cpan工具

用法

安装模块（假设已安装cpanm）：

```
cpanm Data::Section
```

cpanm（“cpanminus”）力求比cpan更简洁，但仍会将所有安装信息记录到日志文件中以备需要。它还会为你处理许多“交互式问题”，而cpan则不会。

cpanm也常用于安装项目的依赖，例如来自GitHub。典型用法是先cd到项目根目录，然后运行

```
cpanm --installdeps .
```

使用--installdeps时，它会：

1. 扫描并安装configure_requires依赖，来源于
   - META.json
   - META.yml（如果缺少META.json）
2. 构建项目（相当于perl Build.PL），生成 MYMETA 文件
3. 扫描并安装来自以下任一的requires依赖项
   - MYMETA.json
   - MYMETA.yml（如果缺少 MYMETA.json）

要指定包含依赖项的文件 'some.cpanfile'，请运行：

```
cpanm --installdeps --cpanfile some.cpanfile .
```

cpanm 安装

有多种方式安装。以下是通过cpan的安装方法：

```
cpan App::cpanminus
```

cpanm 配置

没有cpanm的配置文件。相反，它依赖以下环境变量进行配置：

- `PERL_CPANM_OPT`（通用cpanm命令行选项）
  - 在.bashrc中导出PERL_CPANM_OPT="--prompt" # 例如，启用提示功能
  - 在.tcshrc中设置环境变量PERL_CPANM_OPT "--prompt"
- `PERL_MM_OPT`（ExtUtils::MakeMaker命令行选项，影响模块安装目标）
- `PERL_MB_OPT`（Module::Build命令行选项，影响模块安装目标）

第25.2节：手动安装模块

如果您没有权限安装perl模块，仍然可以手动安装，指定一个您有写权限的自定义路径
。

Chapter 25: Install Perl modules via CPAN

Section 25.1: cpanminus, the lightweight configuration-free replacement for cpan

Usage

To install a module (assuming `cpanm` is already installed):

```
cpanm Data::Section
```

`cpanm` ("cpanminus") strives to be less verbose than `cpan` but still captures all of the installation information in a log file in case it is needed. It also handles many "interactive questions" for you, whereas `cpan` doesn't.

`cpanm` is also popular for installing dependencies of a project from, e.g., GitHub. Typical use is to first `cd` into the project's root, then run

```
cpanm --installdeps .
```

With `--installdeps` it will:

1. Scan and install *configure_requires* dependencies from either
   - META.json
   - META.yml (if META.json is missing)
2. Build the project (equivalent to `perl Build.PL`), generating MYMETA files
3. Scan and install *requires* dependencies from either
   - MYMETA.json
   - MYMETA.yml (if MYMETA.json is missing)

To specify the file 'some.cpanfile', containing the dependencies, run:

```
cpanm --installdeps --cpanfile some.cpanfile .
```

`cpanm` Installation

There are several ways to install it. Here's installation via `cpan`:

```
cpan App::cpanminus
```

`cpanm` Configuration

There is **no** config file for `cpanm`. Rather, it relies on the following environment variables for its configuration:

- `PERL_CPANM_OPT` (General cpanm command line options)
  - `export PERL_CPANM_OPT="--prompt"` # in .bashrc, to enable prompting, e.g.
  - `setenv PERL_CPANM_OPT "--prompt"` # in .tcshrc
- `PERL_MM_OPT` (ExtUtils::MakeMaker command line options, affects module install target)
- `PERL_MB_OPT` (Module::Build command line options, affects module install target)

Section 25.2: Installing modules manually

If you don't have permissions to install perl modules, you may still install them manually, indicating a custom path where you've got writing permissions.

首先，下载并解压模块压缩包：

```
wget module.tar.gz
tar -xzf module.tar.gz
cd module
```

然后，如果模块发行版包含一个Makefile.PL文件，运行：

```
perl Makefile.PL INSTALL_BASE=$HOME/perl
make
make test
make install
```

或者如果你有一个Build.PL文件而不是一个Makefile.PL文件：

```
perl Build.PL --install_base $HOME/perl
perl Build
perl Build 测试
perl Build 安装
```

你还必须在PERL5LIB环境变量中包含模块路径，以便在代码中使用它：

```
export PERL5LIB=$HOME/perl
```

第25.3节：在终端（Mac和Linux）或命令提示符（Windows）中运行Perl CPAN

命令行

你可以使用cpan直接从命令行安装模块：

```
cpan install DBI
```

接下来可能会有多页输出，详细描述安装模块的具体操作。
根据安装的模块不同，可能会暂停并询问你一些问题。

交互式Shell

你也可以这样进入“shell”：

```
perl -MCPAN -e "shell"
```

它将产生如下输出：

```
终端不支持 AddHistory。

cpan shell -- CPAN 探索 和 模块安装 (v2.00)
输入 'h' 获取 帮助。

cpan[1]>
```

然后你可以通过简单命令 install <module> 安装你想要的模块。

示例：cpan[1]> install DBI

Fist, download and unzip module archive:

```
wget module.tar.gz
tar -xzf module.tar.gz
cd module
```

Then, if the module distribution contains a `Makefile.PL` file, run:

```
perl Makefile.PL INSTALL_BASE=$HOME/perl
make
make test
make install
```

or if you have `Build.PL` file instead of a `Makefile.PL`:

```
perl Build.PL --install_base $HOME/perl
perl Build
perl Build test
perl Build install
```

You also have to include the module path in `PERL5LIB` environment variable in order to use it in your code:

```
export PERL5LIB=$HOME/perl
```

Section 25.3: Run Perl CPAN in your terminal (Mac and Linux) or command prompt (Windows)

Command line

You can use `cpan` to install modules directly from the command line:

```
cpan install DBI
```

This would be followed by possibly many pages of output describing exactly what it is doing to install the module. Depending on the modules being installed, it may pause and ask you questions.

Interactive Shell

You can also enter a "shell" thus:

```
perl -MCPAN -e "shell"
```

It will produce output as below:

```
Terminal does not support AddHistory.

cpan shell -- CPAN exploration and modules installation (v2.00)
Enter 'h' for help.

cpan[1]>
```

Then you can install the modules which you want by the easy command `install <module>`.

Example: `cpan[1]> install DBI`

安装成功后，输入退出以退出。

After installing successfully, type `exit` to quit.

第26章：在Mac和Ubuntu上检查已安装模块的简便方法

第26.1节：使用perldoc检查Perl包的安装路径

```
$ perldoc -l Time::Local
```

第26.2节：通过终端检查已安装的perl模块

输入以下命令：

```
instmodsh
```

它会显示如下指南：

```
可用命令有：
l           - 列出所有已安装的模块
   m <module>    - 选择一个模块
   q           - 退出程序
cmd?
```

然后输入1列出所有已安装的模块，你也可以使用命令m<module>来选择模块并获取其信息。

完成后，只需输入q退出。

第26.3节：如何检查Perl corelist模块

```
$ corelist -v v5.23.1
```

Chapter 26: Easy way to check installed modules on Mac and Ubuntu

Section 26.1: Use perldoc to check the Perl package install path

```
$ perldoc -l Time::Local
```

Section 26.2: Check installed perl modules via terminal

Type below command:

```
instmodsh
```

It'll show you the guild as below:

```
Available commands are:
   l            - List all installed modules
   m <module>   - Select a module
   q            - Quit the program
cmd?
```

Then type l to list all the installed modules, you can also use command m <module> to select the module and get its information.

After finish, just type q to quit.

Section 26.3: How to check Perl corelist modules

```
$ corelist -v v5.23.1
```

第27章：打包与解包

第27.1节：手动将C结构体转换为Pack语法

如果你曾经通过Perl代码使用`syscall`、`ioctl`或`fcntl`函数处理C二进制API，你需要知道如何以兼容C的方式构造内存。

例如，如果你曾经处理过某个函数期望一个timespec，你会查看/usr/include/time.h并找到：

```
struct timespec
{
__time_t tv_sec;              /* 秒。   */
    __syscall_slong_t tv_nsec;  /* 纳秒。   */
};
```

你用cpp来探查这到底意味着什么：

```
cpp -E /usr/include/time.h -o /dev/stdout | grep __time_t
# typedef long int __time_t;
cpp -E /usr/include/time.h -o /dev/stdout | grep __syscall_slong_t
# typedef long int __syscall_slong_t
```

所以它是一个（有符号）整数

```
echo 'void main(){ printf("%#lx", sizeof(__syscall_slong_t)); }' |  gcc -x c -include st
dio.h -include time.h - -o /tmp/a.out && /tmp/a.out
# 0x8
```

它占用8个字节。所以是64位有符号整数。我用的是64位处理器。=)

Perldoc `pack` 说明

>   q 一个有符号四字（64-位）值。

所以要打包一个timespec：

```
sub packtime {
    my ( $config ) = @_;
    return pack 'qq', @{$config}{qw( tv_sec tv_nsec )};
}
```

要解包一个timespec：

```
sub unpacktime {
   my ( $buf ) = @_;
   my $out = {};
   @{$out}{qw( tv_sec tv_nsec )} = unpack 'qq', $buf;
   return $out;
}
```

现在你可以直接使用这些函数了。

```
my $timespec = packtime({ tv_sec => 0, tv_nsec => 0 });
syscall(  ..., $timespec ); # 某些读取 timespec 的系统调用
```

Chapter 27: Pack and unpack

Section 27.1: Manually Converting C Structs to Pack Syntax

If you're ever dealing with C Binary API's from Perl Code, via the `syscall`, `ioctl`, or `fcntl` functions, you need to know how to construct memory in a C Compatible way.

For instance, if you were ever dealing with some function that expected a `timespec`, you'd look into `/usr/include/time.h` and find:

```
struct timespec
{
    __time_t tv_sec;            /* Seconds.  */
    __syscall_slong_t tv_nsec;  /* Nanoseconds.  */
};
```

You do a dance with `cpp` to find what that really means:

```
cpp -E /usr/include/time.h -o /dev/stdout | grep __time_t
# typedef long int __time_t;
cpp -E /usr/include/time.h -o /dev/stdout | grep __syscall_slong_t
# typedef long int __syscall_slong_t
```

So its a (signed) int

```
echo 'void main(){ printf("%#lx\n", sizeof(__syscall_slong_t)); }' |
  gcc -x c -include stdio.h -include time.h - -o /tmp/a.out && /tmp/a.out
# 0x8
```

And it takes 8 bytes. So 64bit signed int. And I'm on a 64Bit Processor. =)

Perldoc `pack` says

>   q A signed quad (64-bit) value.

So to pack a timespec:

```
sub packtime {
    my ( $config ) = @_;
    return pack 'qq', @{$config}{qw( tv_sec tv_nsec )};
}
```

And to unpack a timespec:

```
sub unpacktime {
   my ( $buf ) = @_;
   my $out = {};
   @{$out}{qw( tv_sec tv_nsec )} = unpack 'qq', $buf;
   return $out;
}
```

Now you can just use those functions instead.

```
my $timespec = packtime({ tv_sec => 0, tv_nsec => 0 });
syscall(  ..., $timespec ); # some syscall that reads timespec
```

```
稍后 ...
syscall( ..., $timespec ); # 一些写入 timespec 的系统调用
print Dumper( unpacktime( $timespec ));
```

第27.2节：构造IPv4报头

有时你需要处理用C数据类型定义的结构体，这些结构体来自Perl。其中一个应用是创建原始网络数据包，以便你想做一些比常规套接字API更复杂的事情。这正是pack()（当然还有unpack()）的用途。

IP报头的必备部分长度为20个八位字节（也称“字节”）。正如你在此链接后面看到的，源IP地址和目的IP地址构成报头中的最后两个32位值。其他字段中有些是16位，有些是8位，还有一些介于2到13位之间的小块。

假设我们有以下变量要填充到报头中：

```
my ($dscp, $ecn, $length,
    $id, $flags, $frag_off,
    $ttl, $proto,
    $src_ip,
    $dst_ip);
```

注意报头中有三个字段缺失：

- 版本号始终为4（毕竟是IPv4）
- 在我们的示例中，IHL 是 5，因为我们没有options字段；长度以4个八位字节为单位指定，所以20个八位字节对应长度为5。
- 校验和可以保持为0。实际上我们需要计算它，但相关代码在这里不涉及。

我们可以尝试使用位操作来构造例如前32位：

```
my $hdr = 4 << 28 | 5 << 24 | $dscp << 18 | $ecn << 16 | $length;
```

不过这种方法只适用于整数的大小，通常是64位，但也可能低至32位。
更糟的是，它依赖于CPU的字节序，因此在某些CPU上可行，在其他CPU上则会失败。我们试试pack()：

```
my $hdr = pack('H2B8n', '45', sprintf("%06b%02b", $dscp, $ecn), $length);
```

模板首先指定了H2，即2字符的十六进制字符串，高半字节优先。对应传给pack的参数是"45"——版本4，长度5。下一个模板是B8，即8位的位字符串，每字节内按降序排列。我们需要使用位字符串来控制布局，精确到小于半字节（4位）的块，因此使用了sprintf()来从$dscp的6位和$ecn的2位构造这样的位字符串。最后一个是n，即网络字节序的无符号16位值，也就是无论CPU的本地整数格式如何，始终是大端序，由$length填充。

这就是报头的前32位。其余部分可以类似构建：

模板	论点	备注
n	`$id`	
B16	`sprintf("%03b%013b", $flags, $frag_off)`	与DSCP/ECN相同
C2	`$ttl, $proto`	两个连续的无符号八位字节
n	`0/ $checksum`	x 可以用来插入一个空字节，但 n 让我们可以指定一个参数，如果我们选择计算校验和的话

```
later ...
syscall( ..., $timespec ); # some syscall that writes timespec
print Dumper( unpacktime( $timespec ));
```

Section 27.2: Constructing an IPv4 header

Sometimes you have to deal with structures defined in terms of C data types from Perl. One such application is the creation of raw network packets, in case you want to do something fancier than what the regular socket API has to offer. This is just what `pack()` (and `unpack()` of course) is there for.

The obligatory part of an IP header is 20 octets (AKA "bytes") long. As you can see behind this link, source and destination IP address make up the last two 32-bit values in the header. Among the other fields are some with 16 bits, some with 8 bits, and a few smaller chunks between 2 and 13 bits.

Assuming we have the following variables to stuff into our header:

```
my ($dscp, $ecn, $length,
    $id, $flags, $frag_off,
    $ttl, $proto,
    $src_ip,
    $dst_ip);
```

Note that three fields from the header are missing:

- The version is always 4 (it's IPv4 after all)
- IHL is 5 in our example as we don't have an *options* field; length is specified in units of 4 octets so 20 octets gives a length of 5.
- The checksum can be left at 0. Actually we'd have to calculate it but the code to do this doesn't concern us here.

We could try and use bit operations to construct e.g. the first 32 bits:

```
my $hdr = 4 << 28 | 5 << 24 | $dscp << 18 | $ecn << 16 | $length;
```

This approach only works up to the size of an integer though, which is usually 64 bits but can be as low as 32. Worse, it depends on the CPU's endianness so it will work on some CPUs and fail on others. Let's try `pack()`:

```
my $hdr = pack('H2B8n', '45', sprintf("%06b%02b", $dscp, $ecn), $length);
```

The template first specifies `H2`, a *2-character hex string, high nybble first*. The corresponding argument to pack is "45"—version 4, length 5. The next template is `B8`, an *8-bit bit string, descending bit order inside each byte*. We need to use bit strings to control layout down to chunks smaller than a nybble (4 bits), so the `sprintf()` is used to construct such a bit string from 6 bits from `$dscp` and 2 from `$ecn`. The last one is `n`, an *unsigned 16-bit value in Network Byte Order*, i.e. always big-endian no matter what your CPU's native integer format is, and it is filled from `$length`.

That's the first 32 bits of the header. The rest can be built similarly:

Template	Argument	Remarks
n	`$id`	
B16	`sprintf("%03b%013b", $flags, $frag_off)`	Same as DSCP/ECN
C2	`$ttl, $proto`	Two consecutive unsigned octets
n	`0 / $checksum`	x could be used to insert a null byte but n lets us specify an argument should we choose to calculate a checksum

N2	$src_ip, $dst_ip	使用 a4a4 来打包两个 gethostbyname() 调用的结果，因为它们已经是网络字节序了！

所以打包一个 IPv4 头的完整调用是：

```
my $hdr = pack('H2B8n2B16C2nN2',
    '45', sprintf("%06b%02b", $dscp, $ecn), $length,
    $id, sprintf("%03b%013b", $flags, $frag_off),
    $ttl, $proto, 0,
    $src_ip, $dst_ip
);
```

N2	$src_ip, $dst_ip	use a4a4 to pack the result of two gethostbyname() calls as it is in Network Byte Order already!

So the complete call to pack an IPv4 header would be:

```
my $hdr = pack('H2B8n2B16C2nN2',
    '45', sprintf("%06b%02b", $dscp, $ecn), $length,
    $id, sprintf("%03b%013b", $flags, $frag_off),
    $ttl, $proto, 0,
    $src_ip, $dst_ip
);
```

第28章：适用于Windows的Perl命令使用Win32::OLE模块操作Excel

参数	详情
单元格1（必填）	范围的名称。此名称必须是宏语言中的A1样式引用。它可以包含范围运算符（冒号）、交集运算符（空格）或并集运算符（逗号）。
单元格2（可选）	如果指定，单元格1对应范围的左上角，单元格2对应范围的右下角

这些示例介绍了使用Perl通过Win32::OLE模块操作Excel的最常用命令。

第28.1节：打开和保存Excel工作簿

```
#使用的模块
use Cwd 'abs_path';
use Win32::OLE;
use Win32::OLE qw(in with);
use Win32::OLE::Const "Microsoft Excel";
$Win32::OLE::Warn = 3;

#需要使用Excel文件的绝对路径
                  my $excel_file = abs_path("$Excel_path") or die "错误：未找到文件 $Excel_path";

# 打开Excel应用程序
my $Excel = Win32::OLE->GetActiveObject('Excel.Application')
    || Win32::OLE->new('Excel.Application', 'Quit');

# 打开Excel文件
my $Book = $Excel->Workbooks->Open($excel_file);

# 使Excel可见
$Excel->{Visible} = 1;

#___ 添加新工作簿
my $Book = $Excel->Workbooks->Add;
my $Sheet = $Book->Worksheets("Sheet1");
$Sheet->Activate;

#保存 Excel 文件
$Excel->{DisplayAlerts}=0; # 这将关闭“该文件已存在”的提示。
$Book->Save; # 或者 $Book->SaveAs("C:||file_name.xls");
$Book->Close; # 或者 $Excel->Quit;
```

第28.2节：工作表操作

```
# 获取活动工作表
my $Book = $Excel->Activewindow;
my $Sheet = $Book->Activesheet;

# 工作表名称列表
my @list_Sheet = map { $_->{'Name'} } (in $Book->{Worksheets});

# 访问指定工作表
my $Sheet = $Book->Worksheets($list_Sheet[0]);

# 添加新工作表
```

Chapter 28: Perl commands for Windows Excel with Win32::OLE module

Parameters	Details
Cell1 (required)	The name of the range. This must be an A1-style reference in the language of the macro. It can include the range operator (a colon), the intersection operator (a space), or the union operator (a comma).
Cell2 (optional)	If specified, *Cell1* corresponds to the upper-left corner of the range and *Cell2* corresponds to the lower-right corner of the range

These examples introduce the most used commands of Perl to manipulate Excel via Win32::OLE module.

Section 28.1: Opening and Saving Excel/Workbooks

```
#Modules to use
use Cwd 'abs_path';
use Win32::OLE;
use Win32::OLE qw(in with);
use Win32::OLE::Const "Microsoft Excel";
$Win32::OLE::Warn = 3;

#Need to use absolute path for Excel files
my $excel_file = abs_path("$Excel_path") or die "Error: the file $Excel_path has not been found\n";

# Open Excel application
my $Excel = Win32::OLE->GetActiveObject('Excel.Application')
    || Win32::OLE->new('Excel.Application', 'Quit');

# Open Excel file
my $Book = $Excel->Workbooks->Open($excel_file);

#Make Excel visible
$Excel->{Visible} = 1;

#___ ADD NEW WORKBOOK
my $Book = $Excel->Workbooks->Add;
my $Sheet = $Book->Worksheets("Sheet1");
$Sheet->Activate;

#Save Excel file
$Excel->{DisplayAlerts}=0; # This turns off the "This file already exists" message.
$Book->Save; #Or $Book->SaveAs("C:\\file_name.xls");
$Book->Close; #or $Excel->Quit;
```

Section 28.2: Manipulation of Worksheets

```
#Get the active Worksheet
my $Book = $Excel->Activewindow;
my $Sheet = $Book->Activesheet;

#List of Worksheet names
my @list_Sheet = map { $_->{'Name'} } (in $Book->{Worksheets});

#Access a given Worksheet
my $Sheet = $Book->Worksheets($list_Sheet[0]);

#Add new Worksheet
```

```
$Book->Worksheets->Add({After => $workbook->Worksheets($workbook->Worksheets->{Count})});

#更改工作表名称
$Sheet->{Name} = "工作表名称";

#冻结窗格
$Excel -> ActiveWindow -> {FreezePanes} = "True";

#删除工作表
$Sheet -> Delete;
```

第28.3节：单元格操作

```
#编辑单元格的值（两种方法）
$Sheet->Range("A1")->{Value} = 1234;
$Sheet->Cells(1,1)->{Value} = 1234;

#编辑一系列单元格中的数值
$Sheet->Range("A8:C9")->{Value} = [[ undef, 'Xyzzy', 'Plugh' ],
                                   [ 42,    'Perl',  3.1415  ]];

#编辑单元格中的公式（2种类型）
$Sheet->Range("A1")->{Formula} = "=A1*9.81";
$Sheet->Range("A3")->{FormulaR1C1} = "=SUM(R[-2]C:R[-1]C)";      # 行求和
$Sheet->Range("C1")->{FormulaR1C1} = "=SUM(RC[-2]:RC[-1])";      # 列求和

#编辑文本格式（字体）
$Sheet->Range("G7:H7")->Font->{Bold}       = "True";
$Sheet->Range("G7:H7")->Font->{Italic}     = "True";
$Sheet->Range("G7:H7")->Font->{Underline}  = xlUnderlineStyleSingle;
$Sheet->Range("G7:H7")->Font->{Size}       = 8;
$Sheet->Range("G7:H7")->Font->{Name}       = "Arial";
$Sheet->Range("G7:H7")->Font->{ColorIndex} = 4;

#编辑数字格式
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\@";                                   # 文本$Sheet -> Ran
ge("A1:H7") -> {NumberFormat} = "\$#,##0.00";                       # 货币$Sheet -> Range("G7:H7") -> {Nu
mberFormat} = "\$#,##0.00_);[Red](\$#,##0.00)";      # 货币 - 负数红色

$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00_);[Red](0.00)";                 # 带小数的数字

$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0";                             # 带千分位的数字

$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0_);[Red](#,##0)";               # 带千分位的数字 - 负数红色

$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00%";                          # 百分比
$Sheet -> Range("G7:H7") -> {NumberFormat} = "m/d/yyyy"                        # 日期

#对齐文本
$Sheet -> Range("G7:H7") -> {HorizontalAlignment} = xlHAlignCenter;            # 居中文本；
$Sheet -> Range("A1:A2") -> {Orientation} = 90;                                # 旋转文本

#激活单元格
$Sheet -> Range("A2") -> Activate;

$Sheet->Hyperlinks->Add({
   锚点         =>  $range, #包含超链接的单元格范围；例如 $Sheet->Range("A1")
   地址        =>  $adr, #文件路径，http地址等。
   TextToDisplay    =>  $txt, #单元格中的文本
   ScreenTip        =>  $tip, #鼠标悬停在超链接上时的提示
```

```
$Book->Worksheets->Add({After => $workbook->Worksheets($workbook->Worksheets->{Count})});

#Change Worksheet Name
$Sheet->{Name} = "Name of Worksheet";

#Freeze Pane
$Excel -> ActiveWindow -> {FreezePanes} = "True";

#Delete Sheet
$Sheet -> Delete;
```

Section 28.3: Manipulation of cells

```
#Edit the value of a cell (2 methods)
$Sheet->Range("A1")->{Value} = 1234;
$Sheet->Cells(1,1)->{Value} = 1234;

#Edit the values in a range of cells
$Sheet->Range("A8:C9")->{Value} = [[ undef, 'Xyzzy', 'Plugh' ],
                                   [ 42,    'Perl',  3.1415  ]];

#Edit the formula in a cell (2 types)
$Sheet->Range("A1")->{Formula} = "=A1*9.81";
$Sheet->Range("A3")->{FormulaR1C1} = "=SUM(R[-2]C:R[-1]C)";      # Sum of rows
$Sheet->Range("C1")->{FormulaR1C1} = "=SUM(RC[-2]:RC[-1])";      # Sum of columns

#Edit the format of the text (font)
$Sheet->Range("G7:H7")->Font->{Bold}       = "True";
$Sheet->Range("G7:H7")->Font->{Italic}     = "True";
$Sheet->Range("G7:H7")->Font->{Underline}  = xlUnderlineStyleSingle;
$Sheet->Range("G7:H7")->Font->{Size}       = 8;
$Sheet->Range("G7:H7")->Font->{Name}       = "Arial";
$Sheet->Range("G7:H7")->Font->{ColorIndex} = 4;

#Edit the number format
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\@";                                # Text
$Sheet -> Range("A1:H7") -> {NumberFormat} = "\$#,##0.00";                       # Currency
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\$#,##0.00_);[Red](\$#,##0.00)";    # Currency - red
negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00_);[Red](0.00)";                # Numbers with
decimals
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0";                          #     Numbers with
commas
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0_);[Red](#,##0)";              # Numbers with
commas - red negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00%";                             # Percents
$Sheet -> Range("G7:H7") -> {NumberFormat} = "m/d/yyyy"                           # Dates

#Align text
$Sheet -> Range("G7:H7") -> {HorizontalAlignment} = xlHAlignCenter;               # Center text;
$Sheet -> Range("A1:A2") -> {Orientation} = 90;                                   # Rotate text

#Activate Cell
$Sheet -> Range("A2") -> Activate;

$Sheet->Hyperlinks->Add({
   Anchor          =>  $range, #Range of cells with the hyperlink; e.g. $Sheet->Range("A1")
   Address         =>  $adr, #File path, http address, etc.
   TextToDisplay   =>  $txt, #Text in the cell
   ScreenTip       =>  $tip, #Tip while hovering the mouse over the hyperlink
```

```
});
```

注意：要获取超链接列表，请查看以下帖子使用Perl Win32::OLE从Excel工作表中获取超链接列表

第28.4节：行/列的操作

```
#在第22行之前/之后插入一行
$Sheet->Rows("22:22")->Insert(xlUp, xlFormatFromRightOrBelow);
$Sheet->Rows("23:23")->Insert(-4121,0);      #xlDown 是 -4121, xlFormatFromLeftOrAbove 是 0

#删除一行
$Sheet->Rows("22:22")->Delete();

#设置列宽和行高
$Sheet -> Range('A:A') -> {ColumnWidth} = 9.14;
$Sheet -> Range("8:8") -> {RowHeight}   = 30;
$Sheet -> Range("G:H") -> {Columns} -> Autofit;

# 获取最后一行/列
my $last_row = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious, SearchOrder
=> xlByRows})    -> {Row};
my $last_col = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious, SearchOrder
=> xlByColumns}) -> {Column};


#添加边框（方法1）
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlDouble;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThick;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {ColorIndex} = 1;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {LineStyle}  = xlDashDot
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {Weight}     = xlMedium;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {Weight}     = xlThin;

#添加边框（方法2）
my @edges = qw (xlInsideHorizontal xlInsideVertical xlEdgeBottom xlEdgeTop xlEdgeRight);
foreach my $edge (@edges)
```

```
});
```

N.B: to retrieve the list of hyperlinks, have a look at the following post Getting list of hyperlinks from an Excel worksheet with Perl Win32::OLE

Section 28.4: Manipulation of Rows / Columns

```
#Insert a row before/after line 22
$Sheet->Rows("22:22")->Insert(xlUp, xlFormatFromRightOrBelow);
$Sheet->Rows("23:23")->Insert(-4121,0);      #xlDown is -4121 and that xlFormatFromLeftOrAbove is 0

#Delete a row
$Sheet->Rows("22:22")->Delete();

#Set column width and row height
$Sheet -> Range('A:A') -> {ColumnWidth} = 9.14;
$Sheet -> Range("8:8") -> {RowHeight}   = 30;
$Sheet -> Range("G:H") -> {Columns} -> Autofit;

# Get the last row/column
my $last_row = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious, SearchOrder
=> xlByRows})    -> {Row};
my $last_col = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious, SearchOrder
=> xlByColumns}) -> {Column};


#Add borders (method 1)
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlDouble;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThick;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {ColorIndex} = 1;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {LineStyle}  = xlDashDot
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {Weight}     = xlMedium;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {Weight}     = xlThin;

#Add borders (method 2)
my @edges = qw (xlInsideHorizontal xlInsideVertical xlEdgeBottom xlEdgeTop xlEdgeRight);
foreach my $edge (@edges)
```

第29章：通过DBI模块与数据库的简单交互

列	列
`$driver`	数据库驱动，Postgresql用"Pg"，MySQL用"mysql"
`$database`	你的数据库名称
`$userid`	你的数据库ID
`$password`	你的数据库密码
`$query`	在此输入您的查询，例如："select * from $your_table"

第29.1节：DBI模块

您应确保DBI模块已安装在您的电脑上，然后按照以下步骤操作：

1. 在您的Perl脚本中使用DBI模块

```
use DBI;
```

2. 声明一些主要参数

```
my $driver = "MyDriver";
```

```
my $database = "DB_name";
```

```
my $dsn = "DBI:$driver:dbname=$database";
```

```
my $userid = "your_user_ID";
```

```
my $password = "your_password";
```

```
my $tablename = "your_table";
```

3. 连接到您的数据库

```
my $dbh = DBI->connect($dsn, $userid, $password);
```

4. 准备您的查询

```
my $query = $dbh->prepare("Your DB query");
```

示例：

```
$my_query = qq/SELECT * FROM table WHERE column1 = 2/;
```

```
my $query = $dbh->prepare($my_query);
```

我们也可以在查询中使用变量，如下所示：

```
my $table_name = "table";
```

```
my $filter_value = 2;
```

```
$my_query = qq/SELECT * FROM $table_name WHERE column1 = $filter_value/;
```

5. 执行你的查询

```
$query->execute();
```

Chapter 29: Simple interaction with database via DBI module

Column	Column
`$driver`	Driver for DB, "Pg" for Postgresql and "mysql" for MySQL
`$database`	your database name
`$userid`	your database id
`$password`	your database password
`$query`	put your query here, ex: "select * from $your_table"

Section 29.1: DBI module

You should make sure that module DBI has been installed on your pc, then follow the bellow steps:

1. use DBI module in your perl script

```
use DBI;
```

2. Declare some primary parameters

```
my $driver = "MyDriver";
```

```
my $database = "DB_name";
```

```
my $dsn = "DBI:$driver:dbname=$database";
```

```
my $userid = "your_user_ID";
```

```
my $password = "your_password";
```

```
my $tablename = "your_table";
```

3. Connect to your database

```
my $dbh = DBI->connect($dsn, $userid, $password);
```

4. Prepare your query

```
my $query = $dbh->prepare("Your DB query");
```

Ex:

```
$my_query = qq/SELECT * FROM table WHERE column1 = 2/;
```

```
my $query = $dbh->prepare($my_query);
```

We can also use variable in the query, like below:

```
my $table_name = "table";
```

```
my $filter_value = 2;
```

```
$my_query = qq/SELECT * FROM $table_name WHERE column1 = $filter_value/;
```

5. Execute your query

```
$query->execute();
```

*注意：为了避免注入攻击，您应使用占位符?，而不是将变量直接放入查询中。

例如：你想显示来自“table”中所有满足 column1=$value1 且 column2=$value2 的数据：

```
my $query = $dbh->prepare("SELECT * FROM table WHERE column1 = ? AND column2 = ?;");
```

```
$query->execute($value1, $value2);
```

6.获取你的数据

`my @row = $query->fetchrow_array();` 将数据存储为数组

或

my $ref = $sth->fetchrow_hashref(); 将数据存储为哈希引用

7.完成并断开数据库连接

```
$sth->finish;
```

```
$dbh->disconnect();
```

*Note: To avoid injection attack, you should use placeholders ? instead of put your variable in the query.

Ex: you want to show the all data from 'table' where column1=$value1 and column2=$value2:

```
my $query = $dbh->prepare("SELECT * FROM table WHERE column1 = ? AND column2 = ?;");
```

```
$query->execute($value1, $value2);
```

6. Fletch your data

`my @row = $query->fetchrow_array();` store data as array

or

`my $ref = $sth->fetchrow_hashref();` store data as hash reference

7. Finish and disconnect DB

```
$sth->finish;
```

```
$dbh->disconnect();
```

第30章：Perl测试

第30.1节：Perl单元测试示例

下面是一个简单的Perl测试脚本示例，它为测试类/包中的其他方法提供了一定的结构。该脚本输出标准的“ok”/“not ok”文本，这被称为TAP（测试任意协议）。

通常，prove 命令运行脚本并总结测试结果。

```
#!/bin/env perl
# CPAN
use Modern::Perl;
use Carp;
use Test::More;
use Test::Exception;
use Const::Fast;

# 自定义
BEGIN { use_ok('Local::MyPackage'); }

const my $PACKAGE_UNDER_TEST => 'Local::MyPackage';

# 方法 'file_type_build' 的示例测试
sub test_file_type_build {
    my %arg    = @_;
    my $label  = 'file_type_build';
    my $got_file_type;
    my $filename = '/etc/passwd';

    # 检查方法调用是否存在
lives_ok(
        sub {
            $got_file_type = $PACKAGE_UNDER_TEST->file_type_build(
                filename => $filename
            );
        },
        "$label - lives"
    );

    # 检查方法调用的结果是否符合预期结果。
like( $got_file_type, qr{ASCII[ ]text}ix, "$label - result" );
    return;
} ## 子程序 test_file_type_build 结束

# 可以在这里为方法 'file_type_build' 或其他方法添加更多测试。


MAIN: {

subtest 'file_type_build' => sub {
      test_file_type_build();
      # 可以在这里添加更多该方法的测试。
done_testing();
   };

   可以像上面一样添加其他方法的测试。
```

Chapter 30: Perl Testing

Section 30.1: Perl Unit Testing Example

The following is a simple example Perl test script, that gives some structure to allow for testing of other methods in the class/package under test. The script produces standard output with simple "ok" / "not ok" text, which is called TAP (Test Anything Protocol).

Typically the prove command runs the script(s) and summarises the test results.

```
#!/bin/env perl
# CPAN
use Modern::Perl;
use Carp;
use Test::More;
use Test::Exception;
use Const::Fast;

# Custom
BEGIN { use_ok('Local::MyPackage'); }

const my $PACKAGE_UNDER_TEST => 'Local::MyPackage';

# Example test of method 'file_type_build'
sub test_file_type_build {
    my %arg    = @_;
    my $label  = 'file_type_build';
    my $got_file_type;
    my $filename = '/etc/passwd';

    # Check the method call lives
    lives_ok(
        sub {
            $got_file_type = $PACKAGE_UNDER_TEST->file_type_build(
                filename => $filename
            );
        },
        "$label - lives"
    );

    # Check the result of the method call matches our expected result.
    like( $got_file_type, qr{ASCII[ ]text}ix, "$label - result" );
    return;
} ## end sub test_file_type_build

# More tests can be added here for method 'file_type_build', or other methods.


MAIN: {

   subtest 'file_type_build' => sub {
      test_file_type_build();
      # More tests of the method can be added here.
      done_testing();
   };

   # Tests of other methods can be added here, just like above.
done_testing();
} ## 主程序结束：
```

最佳实践

一个测试脚本应只测试一个包/类，但可以有多个脚本用于测试同一个包/类。

进一步阅读

- Test::More - 基本的测试操作。
- Test::Exception - 测试抛出的异常。
- Test::Differences - 比较具有复杂数据结构的测试结果。
- Test::Class - 基于类的测试而非脚本。与JUnit类似。
- Perl 测试教程 - 进一步阅读。

```
    done_testing();
} ## end MAIN:
```

Best Practice

A test script should only test one package/class, but there many scripts may be used to test a package/class.

Further Reading

- Test::More - The basic test operations.
- Test::Exception - Testing thrown exceptions.
- Test::Differences - Comparing test results that have complex data structures.
- Test::Class - Class based testing rather than script. Similarities to JUnit.
- Perl Testing Tutorials - Further reading.

第31章：Dancer

关于：

Dancer2（Dancer的继任者）是一个简单但强大的Perl网页应用框架。

它受Sinatra启发，由Alexis Sukrieh编写。

主要特点：••• 极简 - 直观、极简且非常富有表现力的语法。••• 灵活 - 支持PSGI、插件和模块化设计，具备强大的可扩展性。••• 依赖少 - Dancer尽可能依赖少量CPAN模块，便于安装。

第31.1节：最简单的示例

```
#!/usr/bin/env perl
use Dancer2;

get '/' => sub {
    "Hello World!"
};

dance;
```

Chapter 31: Dancer

About:

Dancer2 (the successor of Dancer) is a simple but powerful web application framework for Perl.

It is inspired by Sinatra and written by Alexis Sukrieh.

Key features: ••• Dead Simple - Intuitive, minimalist and very expressive syntax. ••• Flexible - PSGI support, plugins and modular design allow for strong scalability. ••• Few dependencies - Dancer depends on as few CPAN modules as possible making it easy to install.

Section 31.1: Easiest example

```
#!/usr/bin/env perl
use Dancer2;

get '/' => sub {
    "Hello World!"
};

dance;
```

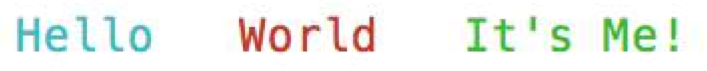
第32章：带属性的文本

第32.1节：打印彩色文本

```
#!/usr/bin/perl

use Term::ANSIColor;

print color("cyan"), "Hello", color("red"), "World", color("green"), "It's Me!",color("reset");
```



```
Hello    World    It's Me!
```

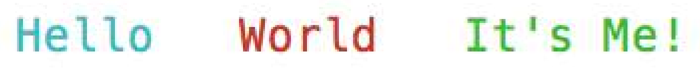
Chapter 32: Attributed Text

Section 32.1: Printing colored Text

```
#!/usr/bin/perl

use Term::ANSIColor;

print color("cyan"), "Hello", color("red"), "\tWorld", color("green"), "\tIt's Me!\n",
color("reset");
```



```
Hello    World    It's Me!
```

第33章：Perl中的GUI应用程序

第33.1节：GTK应用程序

```
use strict;
use warnings;

use Gtk2 -init;

my $window = Gtk2::Window->new();
$window->show();

Gtk2->main();

0;
```

Chapter 33: GUI Applications in Perl

Section 33.1: GTK Application

```
use strict;
use warnings;

use Gtk2 -init;

my $window = Gtk2::Window->new();
$window->show();

Gtk2->main();

0;
```

第34章：内存使用优化

第34.1节：读取文件：foreach与while

在读取可能很大的文件时，while 循环在内存使用上相比 foreach 有显著优势。以下代码将逐条读取文件记录（默认情况下，“记录”指的是由 $/ 指定的“一行”），并在读取时将每条记录赋值给 $_：

```
while(<$fh>) {
    print;
}
```

diamond operator 在这里做了一些魔法，确保循环只在文件末尾终止，而不是例如在只包含字符“0”的行上终止。

下面的循环看起来效果相同，但它在列表上下文中计算 diamond operator，导致整个文件一次性被读入：

```
foreach(<$fh>) {
    print;
}
```

如果你本来就是一次处理一条记录，这会导致大量内存浪费，因此应避免这样做。

第34.2节：处理长列表

如果你已经有一个列表在内存中，处理它的直接且通常足够的方法是使用简单的 foreach 循环：

```
foreach my $item (@items) {
    …
}
```

例如，对于常见的情况，对 $item 进行一些处理然后写入文件而不保留数据，这是可以的。然而，如果你从这些项构建其他数据结构，使用 while 循环会更节省内存：

```
my @result;
while(@items) {
    my $item = shift @items;
    push @result, process_item($item);
}
```

除非对`$item`的引用直接出现在你的结果列表中，否则你从`@items`数组中移除的项目可以被释放，当你进入下一次循环迭代时，解释器可以重用这些内存。

Chapter 34: Memory usage optimization

Section 34.1: Reading files: foreach vs. while

When reading a potentially large file, a `while` loop has a significant memory advantage over `foreach`. The following will read the file record by record (by default, "record" means "a line", as specified by $/), assigning each one to $_ as it is read:

```
while(<$fh>) {
    print;
}
```

The *diamond operator* does some magic here to make sure the loop only terminates at end-of-file and not e.g. on lines that contain only a "0" character.

The following loop seems to work just the same, however it evaluates the diamond operator in list context, causing the entire file to be read in one go:

```
foreach(<$fh>) {
    print;
}
```

If you are operating on one record at a time anyway, this can result in a huge waste of memory and should thus be avoided.

Section 34.2: Processing long lists

If you have a list in memory already, the straightforward and usually sufficient way to process it is a simple `foreach` loop:

```
foreach my $item (@items) {
    ...
}
```

This is fine e.g. for the common case of doing some processing on `$item` and then writing it out to a file without keeping the data around. However, if you build up some other data structure from the items, a `while` loop is more memory efficient:

```
my @result;
while(@items) {
    my $item = shift @items;
    push @result, process_item($item);
}
```

Unless a reference to `$item` directly ends up in your result list, items you shifted off the `@items` array can be freed and the memory reused by the interpreter when you enter the next loop iteration.

第35章：Perl脚本调试

第35.1节：以调试模式运行脚本

要以调试模式运行脚本，你应在命令行中添加-d选项：

```
$perl -d script.pl
```

如果指定了 t，表示调试器将调试的代码中会使用线程：

```
$perl -dt script.pl
```

更多信息请参见perldocperlrun

第35.2节：使用非标准调试器

`$perl -d:MOD 脚本.pl` 在安装了调试、性能分析或跟踪模块

Devel::MOD 的控制下运行程序。

例如，-d:NYTProf 使用 Devel::NYTProf 性能分析器执行程序。

请在此处查看所有 可用的 Devel 模块

推荐模块：

- `Devel::NYTProf` -- 功能强大、快速且功能丰富的 Perl 源代码性能分析器
- `Devel::Trepan` -- 类似 gdb 的模块化 Perl 调试器
- Devel::MAT -- Perl 内存分析工具
- `Devel::hdb` -- 作为网页和 REST 服务的 Perl 调试器
- `Devel::DebugHooks::KillPrint` -- 允许忘记通过 `print` 语句调试
- `Devel::REPL` -- 现代 Perl 交互式 shell
- Devel::Cover -- Perl 代码覆盖率度量工具

Chapter 35: Perl script debugging

Section 35.1: Run script in debug mode

To run script in debug mode you should add `-d` option in the command line:

```
$perl -d script.pl
```

If `t` is specified, it indicates to the debugger that threads will be used in the code being debugged:

```
$perl -dt script.pl
```

Additional info at `perldoc`perlrun

Section 35.2: Use a nonstandard debugger

`$perl -d:MOD script.pl` runs the program under the control of a debugging, profiling, or tracing module installed as `Devel::MOD`.

For example, `-d:NYTProf` executes the program using the `Devel::NYTProf` profiler.

See all available Devel modules here

Recommended modules:

- `Devel::NYTProf` -- Powerful fast feature-rich Perl source code profiler
- `Devel::Trepan` -- A modular gdb-like Perl debugger
- `Devel::MAT` -- Perl Memory Analysis Tool
- `Devel::hdb` -- Perl debugger as a web page and REST service
- `Devel::DebugHooks::KillPrint` -- Allows to forget about debugging by `print` statement
- `Devel::REPL` -- A modern perl interactive shell
- `Devel::Cover` -- Code coverage metrics for Perl

第36章：Perlbrew

Perlbrew 是一个用于管理 $HOME 目录中多个 Perl 安装的工具。

第36.1节：首次设置perlbrew

创建设置脚本 ~/.perlbrew.sh:

```
# 重置可能会干扰 `perlbrew` 的任何环境变量：
export PERL_LOCAL_LIB_ROOT=
export PERL_MB_OPT=
export PERL_MM_OPT=

# 决定你想安装 perlbrew 的位置：
export PERLBREW_ROOT=~/perlbrew
[[ -f "$PERLBREW_ROOT/etc/bashrc" ]] && source "$PERLBREW_ROOT/etc/bashrc"
```

创建安装脚本install_perlbrew.sh：

```
source ~/.perlbrew.sh
curl -L https://install.perlbrew.pl | bash
source "$PERLBREW_ROOT/etc/bashrc"

# 决定您想安装的版本：
version=perl-5.24.1
perlbrew install "$version"
perlbrew install-cpanm
perlbrew switch  "$version"
```

运行安装脚本：

```
./install_perlbrew.sh
```

添加到您的 ~/.bashrc 末尾

```
[[ -f ~/.perlbrew.sh ]] && source ~/.perlbrew.sh
```

加载 ~/.bashrc：

```
source ~/.bashrc
```

Chapter 36: Perlbrew

Perlbrew is a tool to manage multiple perl installations in your $HOME directory.

Section 36.1: Setup perlbrew for the first time

Create setup script `~/.perlbrew.sh`:

```
# Reset any environment variables that could confuse `perlbrew`:
export PERL_LOCAL_LIB_ROOT=
export PERL_MB_OPT=
export PERL_MM_OPT=

# decide where you want to install perlbrew:
export PERLBREW_ROOT=~/perlbrew
[[ -f "$PERLBREW_ROOT/etc/bashrc" ]] && source "$PERLBREW_ROOT/etc/bashrc"
```

Create installation script `install_perlbrew.sh`:

```
source ~/.perlbrew.sh
curl -L https://install.perlbrew.pl | bash
source "$PERLBREW_ROOT/etc/bashrc"

# Decide which version you would like to install:
version=perl-5.24.1
perlbrew install "$version"
perlbrew install-cpanm
perlbrew switch  "$version"
```

Run installation script:

```
./install_perlbrew.sh
```

Add to the end of your `~/.bashrc`

```
[[ -f ~/.perlbrew.sh ]] && source ~/.perlbrew.sh
```

Source `~/.bashrc`:

```
source ~/.bashrc
```

第37章：Perl的安装

我将从Ubuntu的安装过程开始，然后是OS X，最后是Windows。我没有测试所有的Perl版本，但过程应该类似。

如果您想轻松切换不同版本的Perl，请使用Perlbrew。

我想说明本教程讲的是开源版本的Perl。还有其他版本，比如`activeperl`，它们各有优缺点，但不在本教程范围内。

第37.1节：Linux

有多种方法可以做到：

- 使用包管理器：

```
sudo apt install perl
```

- 从源码安装：

```
wget http://www.cpan.org/src/5.0/perl-version.tar.gz
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

- 使用Perlbrew在你的$home目录安装（不需要sudo）：

```
wget -O - https://install.perlbrew.pl | bash
```

另见 Perlbrew

第37.2节：OS X

有几种选项：

- Perlbrew：

```
# 你需要安装 Xcode 的命令行工具
curl -L https://install.perlbrew.pl | bash
```

- 支持线程的 Perlbrew：

```
# 你需要安装 Xcode 的命令行工具
curl -L https://install.perlbrew.pl | bash
```

安装 perlbrew 后，如果你想安装支持线程的 Perl，只需运行：

```
perlbrew install -v perl-5.26.0 -Dusethreads
```

Chapter 37: Installation of Perl

I'm going to begin this with the process in Ubuntu, then in OS X and finally in Windows. I haven't tested it on all perl versions, but it should be a similar process.

Use Perlbrew if you like to switch easily beween different versions of Perl.

I want to state that this tutorial is about Perl in it's open-source version. There are other versions like `activeperl` which its advantages and disadvantages, that are not part of this tutorial.

Section 37.1: Linux

There is more than one way to do it:

- Using the package manager:

```
sudo apt install perl
```

- Installing from source:

```
wget http://www.cpan.org/src/5.0/perl-version.tar.gz
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

- Installing in your $home directory (not sudo needed) with Perlbrew:

```
wget -O - https://install.perlbrew.pl | bash
```

See also Perlbrew

Section 37.2: OS X

There are several options:

- Perlbrew:

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

- Perlbrew with thread support:

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

After the install of perlbrew, if you want to install Perl with thread support, just run:

```
perlbrew install -v perl-5.26.0 -Dusethreads
```

- 来自源代码：

```
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

第37.3节：Windows

- 如前所述，我们选择开源版本。对于Windows，你可以选择strawberry或DWIM。
  这里介绍strawberry版本，因为DWIM是基于它的。这里的简便方法是从官方可执行文件安装。

另请参见berrybrew - Windows Strawberry Perl的perlbrew

- From source:

```
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

Section 37.3: Windows

- As we said before, we go with the open-source version. For Windows you can choose `strawberry` or `DWIM`. Here we cover the `strawberry` version, since `DWIM` is based on it. The easy way here is installing from the official executable.

See also berrybrew - the perlbrew for Windows Strawberry Perl

第38章：从源代码编译Perl cpan模块sapnwrfc

我想描述在Windows 7 x64的Strawberry
Perl环境下构建Perl CPAN模块sapnwrfc的前提条件和步骤。它也应该适用于所有后续的Windows版本，如8、8.1和10。

我使用的是 Strawberry Perl 5.24.1.1 64 位版本，但它也应该适用于较旧的版本。

我花了几个小时才通过多次尝试（32位与64位Perl安装，SAP NW RFC SDK，MinGW与微软C编译器）成功。所以我希望有人能从我的发现中受益。

第38.1节：测试RFC连接的简单示例

来自 http://search.cpan.org/dist/sapnwrfc/sapnwrfc-cookbook.pod 的简单示例

```
use strict;
use warnings;
use utf8;
use sapnwrfc;

SAPNW::Rfc->load_config('sap.yml');
my $conn = SAPNW::Rfc->rfc_connect;

my $rd = $conn->function_lookup("RPY_PROGRAM_READ");
my $rc = $rd->create_function_call;
$rc->PROGRAM_NAME("SAPLGRFC");

eval {
$rc->invoke;
};
if ($@) {
    die "RFC 错误: $@";
}

print "程序名称: ".$rc->PROG_INF->{'PROGNAME'}."";my $cnt_lines_wit
h_text = scalar grep(/LGRFCUXX/, map { $_->{LINE} } @{$rc->SOURCE_EXTENDED});$conn->disconnect;
```

Chapter 38: Compile Perl cpan module sapnwrfc from source code

I'd like to describe the prerequisites and the steps how to build the Perl CPAN module sapnwrfc with the Strawberry Perl environment under Windows 7 x64. It should work also for all later Windows versions like 8, 8.1 and 10.

I use Strawberry Perl 5.24.1.1 64 bit but it should also work with older versions.

It took me some hourse to succeed with several tries (32 vs. 64 bit installation of Perl, SAP NW RFC SDK, MinGW vs. Microsoft C compiler). So I hope some will benefit from my findings.

Section 38.1: Simple example to test the RFC connection

Simple example from http://search.cpan.org/dist/sapnwrfc/sapnwrfc-cookbook.pod

```
use strict;
use warnings;
use utf8;
use sapnwrfc;

SAPNW::Rfc->load_config('sap.yml');
my $conn = SAPNW::Rfc->rfc_connect;

my $rd = $conn->function_lookup("RPY_PROGRAM_READ");
my $rc = $rd->create_function_call;
$rc->PROGRAM_NAME("SAPLGRFC");

eval {
$rc->invoke;
};
if ($@) {
    die "RFC Error: $@\n";
}

print "Program name: ".$rc->PROG_INF->{'PROGNAME'}."\n";
my $cnt_lines_with_text = scalar grep(/LGRFCUXX/, map { $_->{LINE} } @{$rc->SOURCE_EXTENDED});
$conn->disconnect;
```

第39章：最佳实践

第39.1节：使用Perl::Critic

如果你想为自己或团队开始实施最佳实践，那么Perl::Critic是最好的起点。
该模块基于Damien Conway的Perl最佳实践一书，并且相当好地实现了书中提出的建议。

> *注意：我应该提到（Conway本人在书中也说过）这些都是建议。我发现这本书在大多数情况下提供了扎实的理由，尽管我并不完全同意所有内容。重要的是要记住，无论你决定采用哪些实践，都要保持一致。你的代码越可预测，维护起来就越容易。*

你也可以通过浏览器访问perlcritic.com来试用Perl::Critic。

安装

```
cpan Perl::Critic
```

这将安装基本规则集和一个可以从命令行调用的perlcritic脚本。

基本用法

Perlcritic的CPAN文档包含完整的说明，因此我这里只介绍最常见的用例，帮助你入门。基本用法是直接对文件调用perlcritic：

```
perlcritic -1 /路径/到/脚本.pl
```

perlcritic既可以用于脚本，也可以用于模块。参数-1表示你想针对脚本运行的规则的严重级别。共有五个级别，表示Perl::Critic对代码的严格程度。

-5是最宽松的，只会警告可能导致意外结果的潜在危险问题。
-1是最严格的，甚至会抱怨代码是否整洁。根据我的经验，保持代码符合3级标准足以避免风险，同时不会过于挑剔
。

默认情况下，任何违规都会列出触发规则的原因和严重级别：

```
perlcritic -3 --verbose 8 /路径/到/脚本.pl

调试模块在第16行，第1列加载。你加载了Data::Dumper，生产环境中通常不应加载。(严重级别: 4)

私有子程序/方法'_sub_name'在第58行，第1列声明但未使用。请消除死代码。(严重级别: 3)

反引号操作符在第230行，第37列使用。请改用IPC::Open3。(严重级别: 3)
反引号操作符在第327行，第22列使用。请改用IPC::Open3。(严重级别: 3)
```

查看策略

您可以通过使用 perlcritic 的 --verbose 选项快速查看哪些规则被触发以及原因：

将级别设置为 8 会显示触发警告的规则：

Chapter 39: Best Practices

Section 39.1: Using Perl::Critic

If you'd like to start implementing best practices, for yourself or your team, then Perl::Critic is the best place to start. The module is based on the *Perl Best Practices* book by Damien Conway and does a fairly good job implementing the suggestions made therein.

> **Note:** *I should mention (and Conway himself says in the book) that these are suggestions. I've found the book provides solid reasoning in most cases, though I certainly don't agree with all of them. The important thing to remember is that, whatever practices you decide to adopt, you remain consistent. The more predictable your code is, the easier it will be to maintain.*

You can also try out Perl::Critic through your browser at perlcritic.com.

Installation

```
cpan Perl::Critic
```

This will install the basic ruleset and a **perlcritic** script that can be called from the command line.

Basic Usage

The CPAN doc for perlcritic contains full documentation, so I will only be going over the most common use cases to get you started. Basic usage is to simply call perlcritic on the file:

```
perlcritic -1 /path/to/script.pl
```

perlcritic works both on scripts and on modules. The **-1** refers to the severity level of the rules you want to run against the script. There are five levels that correspond to how much Perl::Critic will pick apart your code.

-5 is the most gentle and will only warn about potentially dangerous problems that could cause unexpected results.
-1 is the most brutal and will complain about things as small as your code being tidy or not. In my experience, keeping code compliant with level 3 is good enough to keep out of danger without getting too persnickety.

By default, any failures will list the reason and severity the rule triggers on:

```
perlcritic -3 --verbose 8 /path/to/script.pl

Debugging module loaded at line 16, column 1.  You've loaded Data::Dumper, which probably shouln't
be loaded in production.  (Severity: 4)
Private subroutine/method '_sub_name' declared but not used at line 58, column 1.  Eliminate dead
code.  (Severity: 3)
Backtick operator used at line 230, column 37.  Use IPC::Open3 instead.  (Severity: 3)
Backtick operator used at line 327, column 22.  Use IPC::Open3 instead.  (Severity: 3)
```

Viewing Policies

You can quickly see which rules are being triggered and why by utilizing perlcritic's **--verbose** option:

Setting the level to 8 will show you the rule that triggered a warning:

```
perlcritic -3 --verbose 8 /路径/到/脚本.pl

[Bangs::ProhibitDebuggingModules] 调试模块在第 16 行，第 1 列被加载。 (严重性: 4)
[Subroutines::ProhibitUnusedPrivateSubroutines] 私有子程序/方法 '_sub_name' 声明但
未使用，位于第 58 行，第 1 列。 (严重性: 3)
            [InputOutput::ProhibitBacktickOperators] 反引号操作符在第 230 行，第 37 列被使用。 (严重性:
3)
            [InputOutput::ProhibitBacktickOperators] 反引号操作符在第 327 行，第 22 列被使用。 (严重性:
3)
```

而级别 11 会显示该规则存在的具体原因：

```
perlcritic -3 --verbose 11 /path/to/script.pl

调试模块在第 16 行加载，靠近 'use Data::Dumper;'。
Bangs::禁止调试模块(严重性: 4)
    此策略禁止加载常见的调试模块，如
     Data::Dumper 手册页。

虽然这些模块在开发和调试过程中非常有用，但它们可能不应该在生产环境中加载。如果违
反了这条政策，可能意味着你忘记删除了调试时添加的`use

Data::Dumper;' 这一行。
在第58行，靠近 'sub
_sub_name {' 处声明了私有子程序/方法 '_svn_revisions_differ'，但未被使用。
Subroutines::ProhibitUnusedPrivateSubroutines（严重性：3）按照惯例，Perl
作者（如许多其他语言的作者）通过在标识符前加下划线来表示私有方法和变量。该策略会检
测在声明它们的文件中未被使用的此类子程序。



该模块将对子程序的“使用”定义为对子程序或方法的调用（子程序自身内部除外）、对子程序的引
用（即 `my $foo = \&_foo`）、子程序外部的 `goto` 跳转（例如 `goto &_foo`），或将子程序名
称作为 `use overload` 的偶数位置参数使用。


在第230行，靠近 'my $filesystem_diff = join q{}, `diff $trunk_checkout
$staging_checkout`;' 处使用了反引号操作符。
InputOutput::ProhibitBacktickOperators（严重性：3）反引号非常方
便，尤其是对于CGI程序，但我发现它们在失败时会向标准错误输出大量信息，造成噪音。我认为使用
 IPC::Open3 捕获所有输出，让应用程序决定如何处理更好。



use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

@output = `some_command`;                    #不推荐

        my ($writer, $reader, $err);
open3($writer, $reader, $err, 'some_command'); #推荐;
       @output = <$reader>;  #这里是输出
@errors = <$err>;      # 错误信息在这里，而不是控制台
在第327行附近使用了反引号操作符，靠近 'my $output = `$cmd`;'。
InputOutput::ProhibitBacktickOperators（严重性：3）反引号非常方
便，尤其是对于CGI程序，但我发现它们在失败时会向标准错误输出大量信息，造成噪音。我认为使用
 IPC::Open3 捕获所有输出，让应用程序决定如何处理更好。



use IPC::Open3 'open3';
```

```
perlcritic -3 --verbose 8 /path/to/script.pl

[Bangs::ProhibitDebuggingModules] Debugging module loaded at line 16, column 1.  (Severity: 4)
[Subroutines::ProhibitUnusedPrivateSubroutines] Private subroutine/method '_sub_name' declared but
not used at line 58, column 1.  (Severity: 3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 230, column 37.  (Severity:
3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 327, column 22.  (Severity:
3)
```

While a level of 11 will show the specific reasons why the rule exists:

```
perlcritic -3 --verbose 11 /path/to/script.pl

Debugging module loaded at line 16, near 'use Data::Dumper;'.
  Bangs::ProhibitDebuggingModules (Severity: 4)
    This policy prohibits loading common debugging modules like the
    Data::Dumper manpage.

    While such modules are incredibly useful during development and
    debugging, they should probably not be loaded in production use. If this
    policy is violated, it probably means you forgot to remove a `use
    Data::Dumper;' line that you had added when you were debugging.
Private subroutine/method '_svn_revisions_differ' declared but not used at line 58, near 'sub
_sub_name {'.
  Subroutines::ProhibitUnusedPrivateSubroutines (Severity: 3)
    By convention Perl authors (like authors in many other languages)
    indicate private methods and variables by inserting a leading underscore
    before the identifier. This policy catches such subroutines which are
    not used in the file which declares them.

    This module defines a 'use' of a subroutine as a subroutine or method
    call to it (other than from inside the subroutine itself), a reference
    to it (i.e. `my $foo = \&_foo'), a `goto' to it outside the subroutine
    itself (i.e. `goto &_foo'), or the use of the subroutine's name as an
    even-numbered argument to `use overload'.
Backtick operator used at line 230, near 'my $filesystem_diff = join q{}, `diff $trunk_checkout
$staging_checkout`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                      #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
Backtick operator used at line 327, near 'my $output = `$cmd`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
```

```
$SIG{CHLD} = 'IGNORE';

@output = `some_command`;                    #不推荐

        my ($writer, $reader, $err);
open3($writer, $reader, $err, 'some_command'); #推荐;
      @output = <$reader>;   #这里是输出
@errors = <$err>;      # 错误信息在这里，而不是控制台
```

忽略代码

有时你可能无法遵守 Perl::Critic 的某条规则。在这些情况下，你可以在代码周围使用特殊注释，"## use critic()" 和 " ## no critic"，让 Perl::Critic 忽略它们。只需在括号中添加你想忽略的规则（多个规则用逗号分隔）。

```
##no critic qw(InputOutput::ProhibitBacktickOperator)
my $filesystem_diff = join q{}, `diff $trunk_checkout $staging_checkout`;
## use critic
```

确保将整个代码块包裹起来，否则 Critic 可能无法识别忽略语句。

```
## no critic (Subroutines::ProhibitExcessComplexity)
sub no_time_to_refactor_this {
    ...
}
## use critic
```

请注意，有些策略是在文档级别运行的，不能通过这种方式豁免。然而，
它们可以被关闭...

创建永久例外

使用 ## no critic() 很方便，但当你开始采用编码标准时，可能会想对某些规则做永久例外。你可以通过创建一个. perlcriticrc配置文件来实现。

该文件不仅允许你自定义运行哪些策略，还能自定义它们的运行方式。使用方法很简单，只需将文件放在你的主目录（Linux 下，Windows 是否相同不确定）。或者，你可以在运行命令时使用--profile选项指定配置文件：

```
perlcritic -1 --profile=/path/to/.perlcriticrc /path/to/script.pl
```

同样，perlcritic CPAN 页面有这些选项的完整列表。我将列出我自己配置文件中的一些示例：

应用基本设置：

```
#非常非常严格
严重性 = 1
颜色-严重性-中等 = 粗体黄色
颜色-严重性-低 = 黄色
颜色-严重性-最低 = 粗体蓝色
```

禁用规则（注意策略名称前的短横线）：

```
# 不要求版本控制号
[-杂项::RequireRcsKeywords]
```

```
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                    #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
```

Ignoring Code

There will be times when you can't comply with a Perl::Critic policy. In those cases, you can wrap special comments, "**## use critic()**" and "**## no critic**", around your code to make Perl::Critic ignore them. Simply add the rules you want to ignore in the parentheses (multiples can be separated by a comma).

```
##no critic qw(InputOutput::ProhibitBacktickOperator)
my $filesystem_diff = join q{}, `diff $trunk_checkout $staging_checkout`;
## use critic
```

Make sure to wrap the entire code block or Critic may not recognize the ignore statement.

```
## no critic (Subroutines::ProhibitExcessComplexity)
sub no_time_to_refactor_this {
    ...
}
## use critic
```

Note that there are certain policies that are run on the document level and cannot be exempted this way. However, they can be turned off...

Creating Permanent Exceptions

Using ## no critic() is nice, but as you start to adopt coding standards, you will likely want to make permanent exceptions to certain rules. You can do this by creating a **.perlcriticrc** configuration file.

This file will allow you to customize not only which policies are run, but how they are run. Using it is as simple as placing the file in your home directory (in Linux, unsure if it's the same place on Windows). Or, you can specify the config file when running the command using the **--profile** option:

```
perlcritic -1 --profile=/path/to/.perlcriticrc /path/to/script.pl
```

Again, the perlcritic CPAN page has a full list of these options. I will list some examples from my own config file:

Apply basic settings:

```
#very very harsh
severity = 1
color-severity-medium = bold yellow
color-severity-low = yellow
color-severity-lowest = bold blue
```

Disable a rule (note the dash in front of the policy name):

```
# do not require version control numbers
[-Miscellanea::RequireRcsKeywords]
```

```
# pod 拼写检查过于严格，已禁用
[-文档::PodSpelling]
```

修改规则：

```
# 不需要检查打印失败（针对打印到标准输出而非文件句柄的误报）

[输入输出::RequireCheckedSyscalls]
    函数 = open close

# 允许moose构建器使用特定的未使用子程序
[子程序::禁止未使用的私有子程序]
private_name_regex = _(?!build_)\w+
```

结论

如果正确使用，Perl::Critic可以成为一个宝贵的工具，帮助团队保持代码的一致性和易维护性，无论你采用何种最佳实践政策。

```
# pod spelling is too over-zealous, disabling
[-Documentation::PodSpelling]
```

Modifying a rule:

```
# do not require checking for print failure ( false positives for printing to stdout, not filehandle
)
[InputOutput::RequireCheckedSyscalls]
    functions = open close

# Allow specific unused subroutines for moose builders
[Subroutines::ProhibitUnusedPrivateSubroutines]
private_name_regex = _(?!build_)\w+
```

Conclusion

Properly utilized, Perl::Critic can be an invaluable tool to help teams keep their coding consistent and easily maintainable no matter what best practice policies you employ.

鸣谢

非常感谢所有来自Stack Overflow文档的人员，他们帮助提供了这些内容，更多更改可以发送至web@petercv.com以发布或更新新内容

AbhiNickz	第23章
Al.G.	第18章
外星生命体	第13章
AntonH	第24章
asthman	第26章
Ataul Haque	第9章
badp	第3、5、8、14、16和17章
比尔·蜥蜴	第5章
布赖恩·D·福伊	第10章
卡利亚莱特	第7章
昌基·帕塔克	第31章
克里斯托弗·博托姆斯	第1、2、5、7、8、9、10、12、14、22、24和25章
datageist	第1章
丹尼斯·伊巴耶夫	第1章
digitalis_	第3章、第8章和第14章
德米特里·叶戈罗夫	第16章和第21章
dmvrtx	第3章
德拉夫·斯隆	第19章
DVK	第10章和第15章
eballes	第19章
eddy85br	第1章
尤金·孔科夫	第1章、第3章、第10章、第21章和第35章
fanlim	第26章和第37章
flamey	第37章
flotux	第38章
霍康·黑格兰	第3、12、20和36章
ikegami	第5章
interduo	第39章
伊万·罗德里格斯·托雷斯	第37章
让	第28章
杰夫·Y	第13章
约翰·哈特	第24章
乔恩·埃里克森	第1、3、18和24章
凯米	第12、13、20、21和25章
肯特·弗雷德里克	第3、5、12、14、20、24和27章
莱昂·蒂默曼斯	第1章
lepe	第24章
luistm	第25章和第37章
马特·弗里克	第12章
mbethke	第3、5、8、11、13、20、21、24、27和34章
迈克尔·卡曼	第8章
米克	第3章
msh210	第5章和第8章
穆阿兹·拉菲	第10章
纳加拉朱	第12章
nfanta	第3章
陈元	第6、25、26和29章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

AbhiNickz	Chapter 23
Al.G.	Chapter 18
Alien Life Form	Chapter 13
AntonH	Chapter 24
asthman	Chapter 26
Ataul Haque	Chapter 9
badp	Chapters 3, 5, 8, 14, 16 and 17
Bill the Lizard	Chapter 5
brian d foy	Chapter 10
callyalater	Chapter 7
Chankey Pathak	Chapter 31
Christopher Bottoms	Chapters 1, 2, 5, 7, 8, 9, 10, 12, 14, 22, 24 and 25
datageist	Chapter 1
Denis Ibaev	Chapter 1
digitalis_	Chapters 3, 8 and 14
Dmitry Egorov	Chapters 16 and 21
dmvrtx	Chapter 3
Drav Sloan	Chapter 19
DVK	Chapters 10 and 15
eballes	Chapter 19
eddy85br	Chapter 1
Eugen Konkov	Chapters 1, 3, 10, 21 and 35
fanlim	Chapters 26 and 37
flamey	Chapter 37
flotux	Chapter 38
Håkon Hægland	Chapters 3, 12, 20 and 36
ikegami	Chapter 5
interduo	Chapter 39
Iván Rodríguez Torres	Chapter 37
Jean	Chapter 28
Jeff Y	Chapter 13
John Hart	Chapter 24
Jon Ericson	Chapters 1, 3, 18 and 24
Kemi	Chapters 12, 13, 20, 21 and 25
Kent Fredric	Chapters 3, 5, 12, 14, 20, 24 and 27
Leon Timmermans	Chapter 1
lepe	Chapter 24
luistm	Chapters 25 and 37
matt freake	Chapter 12
mbethke	Chapters 3, 5, 8, 11, 13, 20, 21, 24, 27 and 34
Michael Carman	Chapter 8
Mik	Chapter 3
msh210	Chapters 5 and 8
Muaaz Rafi	Chapter 10
Nagaraju	Chapter 12
nfanta	Chapter 3
Ngoan Tran	Chapters 6, 25, 26 and 29

nslntmnx	第30章
oals	第3、4和7章
oldtechaa	第33章
奥勒·唐格	第5章
奥特拜因	第3章
pwes	第10章和第13章
兰德尔	第25章
rbennett485	第12章
丽贝卡·克洛斯	第2章和第22章
反思心智	第10章
里克·詹姆斯	第10章
鲁斯兰·巴特达洛夫	第4章和第16章
萨吉斯P	第12章、第18章和第32章
萨尔韦什·苏曼	第18章
塞比	第12章
serenesat	第15章
津巴布韦	第9章
索布里克	第19章
SREagle	第12章和第13章
斯蒂芬·莱皮克	第18章
斯瓦罗格	第14章
蒂姆·哈利伯顿	第12章
waghso	第6章和第9章
沃尔夫	第8章和第10章
xfix	第5章和第8章
xtreak	第8章
yonyon100	第12章
扎伊德	第22章
zb226	第3章和第21章

nslntmnx	Chapter 30
oals	Chapters 3, 4 and 7
oldtechaa	Chapter 33
Ole Tange	Chapter 5
Otterbein	Chapter 3
pwes	Chapters 10 and 13
Randall	Chapter 25
rbennett485	Chapter 12
Rebecca Close	Chapters 2 and 22
reflective_mind	Chapter 10
Rick James	Chapter 10
Ruslan Batdalov	Chapters 4 and 16
SajithP	Chapters 12, 18 and 32
Sarwesh Suman	Chapter 18
Sebi	Chapter 12
serenesat	Chapter 15
simbabque	Chapter 9
Sobrique	Chapter 19
SREagle	Chapters 12 and 13
Stephen Leppik	Chapter 18
svarog	Chapter 14
Tim Hallyburton	Chapter 12
waghso	Chapters 6 and 9
Wolf	Chapters 8 and 10
xfix	Chapters 5 and 8
xtreak	Chapter 8
yonyon100	Chapter 12
Zaid	Chapter 22
zb226	Chapters 3 and 21