

Perl 中的 函数（或者 子过程），是行为的一个离散又封闭的单元。它有可能没有名称。它处理也可能不处理外来信息。它产出也可能不产出信息。它代表了一种类型的控制流程，即程序的执行过程转到源代码中的另一个点上。

函数是 *Perl 5* 中抽象和封装以及可重用代码的首选机制；其余不少机制是基于函数这一理念建立的。

## 函数

可以使 `sub` 关键字定义一个函数：

```
B<sub> greet_me { ... }
```

现在起 `greet_me()`

可以在程序的其他任何地方被调用，假如本符号——即函数名——是 可见的。

你不必在声明函数时 定义 它。你可以使用这种 预先声明 的方式告知 Perl 你希望此函数存在的意图，接着延后它的定义：

```
sub greet_sun;
```

你不必在使用 Perl 5 函数之前声明它，除非它修改了语法分析器分析它的方式。参见 `attributes`。

## 函数调用

要调用一个函数，请在源代码中提及它的名称并将可选的参数列表传递给它：

```
greet_me( 'Jack', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

如果你的程序在启用 `strict` 后运行正常，则你 通常 可以忽略用于分组参数的括号，但是它们会将代码意图清晰地展现给给语法分析器——以及更重要的——阅读者。

当然，你可以将多种 类型 的参数传递给函数：

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
```

.....但也请参考引用 `references` 一小节以获取更详细的内容。

## 函数参数

在函数内部，所有参数存在于一个数组中，即 `@_`。如果 `$_` 对应英语单词 `it`（“它”），那么 `@_` 对应的就是单词 `them`（“它们”）。Perl 将所有传入的参数 展开 为单个列表。函数必须自行将参数解开为所需变量或着直接操作 `@_`：

```
sub greet_one
{
    B<my ( $name ) = @_>;
```

```

        say "Hello, $name!";
    }

    sub greet_all
    {
        say "Hello, B<$_" for @_>;
    }

```

@\_ 表现得和 Perl 中其他数组一样。你可以用下标引用单个元素：

```

sub greet_one_indexed
{
    B<my $name = $_[0]>;
    say "Hello, $name!";

    # or, less clear
    say "Hello, $_[0]!";
}

```

你也可以 `shift`、`unshift`、`push`、`pop`、`splice` 并且在 @\_ 上使用列表分片。在函数内部，`shift` 和 `pop` 隐含地对 @\_ 进行操作，就像在函数外操作 @ARGV 一样：

```

sub greet_one_shift
{
    B<my $name = shift>;
    say "Hello, $name!";
}

```

虽然起初写下 `shift @_` 看上去更清晰，但利用 `shift` 的隐含操作数是 Perl 5 中的惯用语。

注意从 @\_ 中赋值一个标量变量需要 `shift`、或者对 @\_ 的下标访问、又或者左值列表上下文括号。否则，Perl 5 会高兴地替你在标量上下文中对 @\_ 求值，并将参数个数赋值给此标量：

```

sub bad_greet_one
{
    B<my $name = @_>; # buggy
    say "Hello, $name; you're looking quite numeric today!"
}

```

多参数的列表赋值通常比多行 `shift` 要来得清爽。请比较：

```

sub calculate_value
{
    # multiple shifts
    my $left_value = shift;
    my $operation = shift;
    my $right_value = shift;
    ...
}

```

.....和：

```

sub calculate_value
{
    B<my ($left_value, $operation, $right_value) = @_;>
}

```

```

    ...
}

```

偶尔会需要提取 @\_ 中的部分参数并把其余的传递给另外函数:

```

sub delegated_method
{
    my $self = shift;
    say 'Calling delegated_method()'

    $self->delegate->delegated_method( @_ );
}

```

主导用法就是仅在函数需要访问单个参数使用 shift, 访问多个参数时使用列表赋值。

说明性的参数处理方法, 请参见如下 CPAN 模块: signatures、Method::Signatures、MooseX::Method::Signatures。

## 参数展开

参数展开为 @\_ 发生在调用方。把哈希作为参数传递给函数将产生一个键值对列表:

```

sub show_pets
{
    my %pets = @_;
    while (my ($name, $type) = each %pets)
    {
        say "$name is a $type";
    }
}

my %pet_names_and_types = (
    Lucky    => 'dog',
    Rodney   => 'dog',
    Tuxedo    => 'cat',
    Petunia  => 'cat',
);

show_pets( %pet_names_and_types );

```

show\_pets() 函数的工作原理就是因为 %pet\_names\_and\_types 哈希展开为列表 'Lucky', 'dog', 'Rodney', 'dog', 'Tuxedo', 'cat', 'Petunia', 'cat'。show\_pets() 函数内的哈希赋值工作原理基本上就和更为明确的 %pet\_names\_and\_types 赋值一样。

通常很有用, 但是当你将一部分参数作为标量传递、一部分参数作为展开列表传递时, 你必须明确你的目的。如果你想编写一个 show\_pets\_of\_type() 函数, 它的一个参数是要显示的宠物的 类型, 你必须将其作为 第一个 参数 (或使用 pop 把它从 @\_ 的末尾移出):

```

sub show_pets_by_type
{
    B<my ($type, %pets) = @_;>;

    while (my ($name, $species) = each %pets)
    {
        B<next unless $species eq $type;>
        say "$name is a $species";
    }
}

```

```

my %pet_names_and_types = (
    Lucky    => 'dog',
    Rodney   => 'dog',
    Tuxedo    => 'cat',
    Petunia  => 'cat',
);

show_pets_by_type( 'dog',    %pet_names_and_types );
show_pets_by_type( 'cat',    %pet_names_and_types );
show_pets_by_type( 'moose',  %pet_names_and_types );

```

## 参数吸入

就像所有对集合的左值赋值，在函数内对 `%pets` 的赋值会吸入所有 `@_` 中剩下的值。如果 `$type` 参数位于 `@_` 的末尾，Perl 将试图将奇数个元素赋值给哈希，并会产生警告。你可以绕过它：

```

sub show_pets_by_type
{
    B<my $type = pop;>
    B<my %pets = @_;>

    ...
}

```

.....但会以清晰为代价。当然，同样的原理也适用于赋值给一个数组并将其作为参数。在传递集合参数时，避免展开和吸入，请参考“引用”（references）一节。

## 别名

一个 `@_` 的有用特性可能会使得那些粗心的人感到惊讶：在你将解开 `@_` 为属于自己的变量之前，它包含了传入参数的别名。这个行为最容易用如下例子说明：

```

sub modify_name
{
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
say $name;

# prints C<egnarO>

```

如果你直接修改了 `@_` 中的某个元素，你将同时对原始参数进行直接修改。请谨慎些。

## 名称空间

每一个函数存在于某名称空间中。位于未声明名称空间中的函数——即，没有在一条明确的 `package ...` 语句之后声明的函数——存在于 `main` 名称空间。你可以在声明时为函数指定当前之外的名称空间：

```

sub B<Extensions::Math::>add {
    ...
}

```

任何符合包命名规则的函数名前缀会创建此函数并将其插入合适的名称空间中，而非当前的名称空间。由于 Perl 5 中包可以在任何时刻被修改，你甚至可以在名称空间未存在前、或该函数已在目标名称空间中声明时这样做。

你仅可以在同一名称空间下为某名称声明一个函数。否则，Perl 5 将就子过程重定义发出警告。如果你确实想 替换 一个已经存在的函数，可以通过 `no warnings 'redefine'` 来禁用这类警告。

通过使用完全限定名称，你可以在其他名称空间内调用函数：

```
package main;

Extensions::Math::add( $scalar, $vector );
```

名称空间中的函数对外部是 可见的，因为它们可以被直接地引用，但仅在它们可以在所定义的名称空间内使用其短名时称其为 可调用——除非你通过导入导出机制使这些函数在当前名称空间内可用（`exporting`）。

## 导入

当使用 `use` 关键字加载模块时（`modules`），Perl 自动调用所提供模块的 `import()` 方法。带有过程式接口的模块可以提供它们自己的 `import()`，以使部分或全部经过定义的符号在调用者的名称空间中可用。所有 `use` 语句内模块名后的参数会传递给模块的 `import()` 方法。即：

```
use strict;
```

.....加载 `strict.pm` 模块并不带参数地调用 `strict->import()`，且：

```
use strict 'refs';
use strict qw( subs vars );
```

.....加载 `strict.pm` 模块，调用 `strict->import( 'refs' )`，接着再调用 `strict->import( 'subs', vars' )`。

你可以直接调用某模块的 `import()` 方法。前面的代码示例等价于：

```
BEGIN
{
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

注意，`use` 关键字向这些语句添加了一个隐式的 `BEGIN` 块使得 `import()` 调用在语法分析器编译完整个语句后 立即 执行。这样便保证在编译后续部分时所有导入符号可见。否则，任何从其他模块导入而未在当前文件定义的函数将看上去像是为声明的裸字 使 `strict` 发出抱怨。

## 报告错误

在函数之内，你可以通过 `caller` 操作符得到有关本次调用的上下文信息。如果加参数，它返回三个元素的列表，包括调用包的名称、包含本次调用的文件名、本次调用发生的包内行号：

```
package main;
```

```

main();

sub main
{
    show_call_information();
}

sub show_call_information
{
    my ($package, $file, $line) = caller();
    say "Called from $package in $file at $line";
}

```

你可以向 `caller()` 传递单个可选的整数参数。如此，Perl 将按给出的层数回查调用者的调用者的调用者，并提供该次调用的相关信息。换句话说，如果 `show_call_information()` 中用到 `caller(0)`，它将收到来自 `main()` 的调用信息。如果它使用 `caller(1)`，则会收到程序开始之初的调用信息。

提供此可选参数可以让你检查调用者的调用者，同时它也会返回更多值，包括函数的名称，和调用的上下文：

```

sub show_call_information
{
    my ($package, $file, $lineB<, $func>) = caller(B<0>);
    say "Called B<$func> from $package in $file at $line";
}

```

标准 `Carp` 模块使用这个技巧来增强错误报告和函数警告的效果，其中的 `croak()` 抛出由调用者在某文件某行报告的异常。当在库代码中代替 `die` 使用，`croak()` 会因不正确的使用方法而就地抛出异常。`Carp` 中的 `carp()` 函数从文件中发出警告并报告有问题的行号（`producing_warnings`）。

此行为在验证参数或函数先决条件时候最为有用，例如意图指出调用代码的错误时：

```

use Carp 'croak';

sub add_two_numbers
{
    croak 'add_two_numbers() takes two and only two arguments'
        unless @_ == 2;

    ...
}

```

## 验证参数

防御性编程通常得益于针对参数类型和值的检查，以便做出后续处理。Perl 5 为此提供了一些默认内置机制（别指望 `prototypes` 对此有所帮助）。你可以通过在标量上下文中对 `@_` 求值来检查传递给函数的参数个数是否正确：

```

sub add_numbers
{
    croak "Expected two numbers, but received: " . @_
        unless @_ == 2;

    ...
}

```

```
}
```

Perl 面向操作符的类型转换（参见“上下文哲学”一节 `context_philosophy`）使得类型检查更为困难。如果你确实想变得更加严格，请考虑 CPAN 模块 `Params::Validate`。

## 高级函数

函数也需看上去简单，但你可以做的还有很多很多（参见闭包 `closures` 和匿名函数 `anonymous_functions` 以获得更多细节）。

## 上下文认知

Perl 5 内置函数了解你是在空、标量还是列表上下文中调用它们。你编写的函数也可以知晓它们各自的调用上下文。被错误命名的 参见 `perldoc -f wantarray` 以确认 `wantarray` 关键字返回 `undef` 表明空上下文，假值表明标量上下文，真值则表明列表上下文。

```
sub context_sensitive
{
    my $context = wantarray();
    return qw( Called in list context ) if $context;
    say 'Called in void context' unless defined $context;
    return 'Called in scalar context' unless $context;
}

context_sensitive();
say my $scalar = context_sensitive();
say context_sensitive();
```

这在避免可能产生昂贵返回值的函数在空上下文中返回结果时很有用。一些惯用语函数在列表上下文中返回列表并在标量上下文中返回数组引用（或列表中的第一个元素）。即便如此，也没有什么针对使用 `wantarray` 与否的好建议；一些时候还是单独编写返回所需类型值的函数来得清楚。

话虽如此，CPAN 上的 Robin Houston 的 `Want` 和 Damian Conway 的 `Contextual::Return` 发行版为编写强大而可用的接口提供了种种可能。

## 递归

在 Perl 中每一次函数调用都创建一个新的 调用帧。这是一种代表调用本身的内部数据结构：实际上就是，传入参数、返回点、步入此调用点前的所有程序控制流程。它同样捕获了本次函数调用的特定词法环境。这意味着一个函数可以 递归，它可以调用自己。

递归是一个带迷惑性但又简单的概念，如果之前你没有碰到过，它看上去令人望而却步。考虑你如何在一个经过排序的数组中查找某个元素。你可以 迭代整个数组中的每个元素，在其中查找目标，但是平均起来，你每次只需查找数组中半数元素。

另一中方法就是将数组分开两半。选取中间值，比较，看接下去是否应该将较小的那部分进行折半还是对较大部分，接着继续。你可以用循环来编写上述算法，或者，你可以通过递归 让 Perl 接管所有状态和跟踪所需变量。看上去可能会像这样：

```
use Modern::Perl;

use Test::More tests => 8;

my @elements = ( 1, 5, 6, 19, 48, 77, 997, 1025, 7777, 8192, 9999 );

ok elem_exists( 1, @elements ), 'found first element in array';
ok elem_exists( 9999, @elements ), 'found last element in array';
```

```

    ok ! elem_exists( 998, @elements ), 'did not find element not in
array';
    ok ! elem_exists( -1, @elements ), 'did not find element not in
array';
    ok ! elem_exists( 10000, @elements ), 'did not find element not in
array';

    ok elem_exists( 77, @elements ), 'found midpoint element';
    ok elem_exists( 48, @elements ), 'found end of lower half
element';
    ok elem_exists( 997, @elements ), 'found start of upper half
element';

sub elem_exists
{
    my ($item, @array) = @_;

    # 如果没有元素可以查找则跳出递归
    return unless @array;

    # 折半, 如果有奇数个元素则向下取整
    my $midpoint = int( (@array / 2) - 0.5 );
    my $miditem = $array[ $midpoint ];

    # 如果当前元素就是目标则返回真
    return 1 if $item == $miditem;

    # 如果只剩一个元素则返回假
    return if @array == 1;

    # 划分数组并用较小分支递归
    return B<elem_exists>( $item, @array[0 .. $midpoint] )
        if $item < $miditem;

    # 划分数组并用较大分支递归
    return B<elem_exists>( $item, @array[$midpoint + 1 .. $#array] );
}

```

这并非是搜索已排序列表的最好的算法，但它演示了递归。再次说明，你可以按过程式方法编写这段代码，但是某些算法采用递归时更清晰。

## 词法相关

每一次对函数的新调用创建自身词法作用域的实例。拿递归为例，虽然 `elem_exists()` 的声明为 `$item`、`@array`、`$midpoint` 和 `$miditem` 创建的单独的词法作用域，对 `elem_exists()` 的每一次调用，即便是递归，也将词法变量的值分开存放。你可以通过添加如下调试代码来展示这一特性：

```

B<use Carp 'cluck'>;

sub elem_exists
{
    my ($item, @array) = @_;

    B<cluck "[$item] (@array)">

```



```

        # other code follows
        ...
    }

```

这段代码的输出显示了 `elem_exists()` 不仅可以安全地调用自身，而且词法变量之间也不会冲突。

## 尾部调用

递归的一个 缺点 就是你必须将返回条件编写正确，否则函数将无限次调用自身。这 就是为什么 `elem_exists()` 函数拥有若干 `return` 语句。

当它检测到失控的递归时，Perl 提供了有用的警告：`Deep recursion on subroutine`。限制是 100 次递归调用，对某些情况可能太少而其他情况下又太多。你可以在递归调用作用域内通过 `no warnings 'recursion'` 来禁用这一警告。

因为对函数的每一次调用都需要新建调用帧，外加自身词法变量值的存储，高度递归的代码比迭代用去的内存更多。一个名为 尾部调用消除 的特性有助解决此问题。

尾部调用消除也许在编写递归代码时最为明确，但它在含尾部调用的任何情况下都有用处。许多编程语言实现支持自动进行这个过程。

尾部调用 就是调用一个函数然后直接返回这个函数的结果。下列几行：

```

# split the array down and recurse
return elem_exists( $item, @array[0 .. $midpoint] )
    if $item < $miditem;

# split the array up and recurse
return elem_exists( $item, @array[$midpoint + 1 .. $#array] );

```

.....直接返回递归调用 `elem_exists()` 的结果，就是尾部调用消除的候选代码。这个消除的过程避免了先将结果返回给本次调用再返回父调用。取而代之的是它直接将结果返回给父 调用。

Perl 5 支持手动进行尾部调用消除，但如果你觉得高度递归的代码可以很好地利用尾部调用消除的优势，那么 Yuval Kogman 的 `Sub::Call::Tail` 值得研究。`Sub::Call::Tail` 对非递归的尾部调用同样适用：

```

use Sub::Call::Tail;

sub log_and_dispatch
{
    my ($dispatcher, $request) = @_;
    warn "Dispatching with $dispatcher\n";

    return dispatch( $dispatcher, $request );
}

```

本例中，你可以将 `return` 改为新的 `tail` 关键字并无须做出函数层面的修改（它更加清晰，也提升了性能）：

```

B<tail> dispatch( $dispatcher, $request );

```

如果你确实 必须 手动进行尾部调用消除，则你可以使用 `goto` 关键字的一个特殊形式。不像通常导致通心粉式代码的那种形式，`goto` 的函数形式将当前函数调用替换为对另一个函数的调用。可以使用函数名称或引用。如果你想传递不同的参数，则必须手动设置 `@_:`

```
# 划分数组并递归（较小部分）
if ($item < $miditem)
{
    @_ = ($item, @array[0 .. $midpoint]);
    B<goto &elem_exists;>
}

# 划分数组并递归（较大部分）
else
{
    @_ = ($item, @array[$midpoint + 1 .. $#array] );
    B<goto &elem_exists;>
}
```

相比较而言，代码 CPAN 版本的清爽程度可见一斑。

## 缺陷和设计失误

所有 Perl 5 函数的特性并非都是有用的。特别的，原型 `prototypes` 很少完成你想让它做的事。 它们有它们的用途，但是除了个别情况下，你可以避免使用它们。

从早期 Perl 版本直接接手，Perl 5 仍支持旧式函数调用。虽然现在你可以直接通过名称调用函数， 但前期版本的 Perl 要求你用前置的 `&` 字符调用函数。Perl 1 要求你使用 `do` 关键字：

```
# 过时风格；避免使用
my $result = &calculate_result( 52 );

# Perl 1 风格
my $result = do calculate_result( 42 );

# 疯狂的混杂调用，真的应该避免
my $result = do &calculate_result( 42 );
```

相比退化的语法，除了看上去是噪音外，前置 `&` 形式的其他行为偶尔会使你惊奇。第一，它禁用原型检查（好像这常常很重要）。第二，如果你不明确地传递参数，它会 隐式地 将 `@_` 的内容不加修改地传给函数。两者都可导致意外的结果。

最后一个缺陷来自于函数调用时不写括号。Perl 5 的语法分析器使用若干启发式方法解决裸字的歧义以及传递给函数参数的个数，但是这些猜测偶尔会出错。虽然移去无关括号通常很明智，但请比较下列两行代码的可读性：

```
ok( elem_exists( 1, @elements ), 'found first element in array' );

# 警告；包含一细微的缺陷
ok elem_exists 1, @elements, 'found first element in array';
```

第二行中细微的缺陷就是对 `elem_exists()` 的调用将本应作为 `ok()` 第二个参数的测试描述贪婪地吞入。因为 `elem_exists()` 的第二个参数是吸入式的，这种情况可能一直不为人知直到 Perl 产生有关将非数字（测试描述，它不能被转换为数字）和数组中某元素进行比较的警告。

这诚然是一个特例，但说明了正确地加括号可以使代码清晰并使得细微的缺陷在阅读者面前展露无遗。

## 作用域

Perl 中的 作用域 指的是符号的生存期限和可见性。在 Perl 中，任何有名字的事物（比如：变量、函数）都有作用域。作用域的制定有助于强制 封装——将相关的概念放在一块并防止它们的泄漏。

## 词法作用域

在现代的 Perl 编程环境中，最常见的作用域形式是词法作用域。Perl 编译器在编译期解决此类作用域。这类作用域在你 阅读 一段程序时是可见的。

要创建一个新的词法作用域，可以编写一个由大括号分隔的代码块。这个代码块可以是一个裸块、或循环结构主体中的块、一个 eval 块、或是其他任何没有用引号引起的块：

```
# 外层词法作用域
{
    package My::Class;

    # 内层词法作用域
    sub awesome_method
    {
        # 最内层词法作用域
        do {
            ...
        } while (@_);

        # 内层词法作用域的兄弟
        for (@_)
        {
            ...
        }
    }
}
```

词法作用域管理由 my 声明的变量的可见性；这些变量被称作 词法 变量。在某块内声明的词法变量对块本身及嵌套块可见，但对此块兄弟或外层块是不可见的。因此，在如下 代码中：

```
# 外层词法作用域
{
    package My::Class;

    my $outer;

    sub awesome_method
    {
        my $inner;

        do {
```

```

        my $do_scope;
        ...
    } while (@_);

    # 兄弟内层词法作用域
    for (@_)
    {
        my $for_scope;
        ...
    }
}

```

..... \$outer 对全部四个作用域都是可见的。\$inner 在方法内部、do 代码块和 for 循环内都是可见。\$do\_scope 仅在 do 代码块内可见，\$for\_scope 仅在 for 循环内可见。

在内层词法作用域里声明一个和外部词法作用域同名的词法变量将隐藏，或者说 遮盖 外层的词法变量：

```

{
    my $name = 'Jacob';

    {
        my $name = 'Edward';
        say $name;
    }

    say $name;
}

```

这段程序在 Edward 之后接着打印 Jacob。即使在单一词法作用域内重新声明一个有着相同名称相同类型的词法变量会产生一个警告，在嵌套作用域内遮盖一个词法变量则不会；这是一个词法遮盖的特性。

词法遮盖可能会意外地发生，但是通过限制变量的作用域和嵌套的层数——这是一个良好的设计——你可以减小此风险。

词法变量的声明有着自身的微妙性。例如，一个用作 for 循环迭代器变量的词法变量的作用域是循环代码块 内部。它对循环体外部是不可见的：

```

my $cat = 'Bradley';

for my $cat (qw( Jack Daisy Petunia Tuxedo ))
{
    say "Iterator cat is $cat";
}

say "Static cat is $cat";

```

类似地，given 语法结构在其块内部创建了 词法话题 （近似于 my \$\_）：

```

$_ = 'outside';

given ('inner')

```

```

{
    say;
    $_ = 'whomped inner';
}

say;

```

.....先不管块内对 `$_` 的赋值。你可以显式地词法化话题，虽然这在考虑动态作用域时 更为有用。

最后，词法作用域助于构造闭包（closures）。注意不要意外地创建闭包。

“Our” 作用域

在给出的作用域内，你可以用 `our` 关键字声明一个包变量的别名。就像 `my` 一样，`our` 强制了别名的词法作用域。完全限定名称随处可用，但是词法别名仅在自身作用 域内可见。

对 `our` 的最好使用就是声明那些你 不得不 有的变量，诸如 `$VERSION`。

动态作用域

动态作用域在可见性规则上类似于词法作用域，和在编译期确定作用域相反，确定作用域的过程沿着调用上下文发生。考虑如下例子：

```

{
    our $scope;

    sub inner
    {
        say $scope;
    }

    sub main
    {
        say $scope;
        local $scope = 'main() scope';
        middle();
    }

    sub middle
    {
        say $scope;
        inner();
    }

    $scope = 'outer scope';
    main();
    say $scope;
}

```

这段程序由声明一个 `our` 变量——`$scope`，和三个函数开始。它于赋值 `$scope` 并调用 `main()` 处结束。

在 `main()` 内，这个程序打印出 `$scope` 当前的值，即 `outer scope`，接着用 `local`

局部化了这个变量。这样，符号在当前词法作用域内的可见性 连同 该符号在此词法作用域内调用的函数内部的可见性一同被改变。因此，`$scope` 在 `middle()` 和 `inner()` 两者的代码体包含 `main()` `scope` 这个值。在 `main()` 返回后——在该点流程退出了那个代码块，块中包含 `local` 后的 `$scope`，Perl 恢复了变量的原始值。最后的 `say` 再次打印出 `outer scope`。

虽然此变量在这些作用域内都是 可见的，但变量 值 的变化却取决于用 `local` 局部化和赋值操作。这个特性既狡猾又微妙，但改变那些神奇变量时还是相当有用的。

包变量和词法变量在可见性上的区别用 Perl 5 自身存储这些变量的机制来解释，就会变得非常明显。词法变量被存放在附着于作用域的 词法板 中。每次进入到词法作用域中都需要 Perl 来创建一个包含变量值的新的专属词法板。（这就是为什么函数可以调用自身而不会弄坏现有同名变量的值。）

包变量的存储机制称为符号表。每个包都含有一个单独的符号表，并且每个包变量在其中占有一个条目。你可以用 Perl 检查并修改这个符号表；这就是导入的工作原理（`importing`）。这也是为什么你只能用 `local` 局部化全局和包变量而非词法变量。

用 `local` 局部化若干神奇变量的做法很常见。举例来说，`$/`，输入记录分隔符，决定了 `readline` 操作从一个文件句柄读入数据的量。`$!`，系统错误变量，包含了最近一次系统调用的错误号。`$@`，Perl `eval` 错误变量，包含最近一次 `eval` 操作中发生的任何错误。`$|`，自动冲洗变量，决定了 Perl 是否应该在每次写操作之后，自动 冲洗当前由 `select` 选定的文件句柄。

这些全部是特殊的全局变量；在最窄小的作用域内用 `local` 局部化它们可以避免其余代码远距离修改所用全局变量而引发的问题。

## “State”（状态）作用域

最后一种作用域类型的年纪和 Perl 5.10 一样。这是 `state` 关键字的作用域。状态作用域类似词法作用域的地方在于，它声明一个词法变量，但是该变量的值只初始化 一次，随后便一直保持：

```
sub counter
{
    B<state> $count = 1;
    return $count++;
}

say counter();
say counter();
say counter();
```

在第一次调用计数函数的地方，`$count` 从未被初始化过，因此 Perl 执行赋值操作。这个程序打印出 1、2 和 3。如果你把 `state` 改为 `my`，则会打印 1、1、1。

你也可以使用传入参数来初始化 `state` 变量的值：

```
sub counter
{
    state $count = shift;
    return $count++;
}
```

```
say counter(B<2>);
say counter(B<4>);
say counter(B<6>);
```

虽然这段代码粗粗一读让人感觉输出应该是 2、4、6，但实际上是 2、3 和 4。第一次对 `counter` 子过程的调用设置了 `$count` 变量。后续调用不会改变它的值。这个行为是意料之中并如同记载的一样，但这种实现方式会导致令人惊讶的结果：

```
sub counter
{
    state $count = shift;
    say 'Second arg is: ', shift;
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

程序中计数器按预想地打印出 2、3 和 4，但是成为 `counter()` 调用的第二个参数却依次为 `two`、4 和 6——并非因为这些整数确实变成了传递的第二个参数而是因为第一个参数的 `shift` 只发生在第一次调用 `counter()` 的时候。

`state` 在创建默认值和准备缓存时非常有用，但在使用它前，请确信你已经理解了它的初始化行为。

## 匿名函数

匿名函数 就是没有名字的函数。它的行为和那些有名字的函数一样——你可以调用它，也可以把参数传递给它，从其中返回结果，复制引用给它——它可以做到任何具名函数可以做到的事。区别就是它没有名字。你可以用引用来处理匿名函数（参见引用 `references` 和函数引用 `function_references`）。

## 声明匿名函数

你不能独立地声明一个匿名函数；你必须在构造完成后将它复制给一个变量，或直接调用它，再就是将它作为参数传递给另一个函数，显式或是隐式地。使用 `sub` 关键字而不加命名将显式地创建一个匿名函数：

```
my $anon_sub = sub { ... };
```

一个名为 分派表 的 Perl 5 惯用语，使用哈希将输入和行为关联起来：

```
my %dispatch =
(
    plus      => sub { $_[0] + $_[1] },
    minus     => sub { $_[0] - $_[1] },
    times     => sub { $_[0] * $_[1] },
    goesinto  => sub { $_[0] / $_[1] },
    raisedto  => sub { $_[0] ** $_[1] },
);

sub dispatch
{
    my ($left, $op, $right) = @_;
```

```

        die "Unknown operation!"
        unless exists $dispatch{ $op };

        return $dispatch{ $op }->( $left, $right );
    }

```

`dispatch()` 函数以 `(2, 'times', 2)` 的形式接受参数并且返回对操作求值后的 结果。

你可以在使用函数引用的地方用匿名函数。对于 Perl 来说，两者是等价的。没有什么 迫使使用匿名函数来进行这些数学操作，但对这类短小的函数来说，写成这样也没 有什么不好。

你可以将 `%dispatch` 重写为：

```

my %dispatch =
(
    plus      => \&add_two_numbers,
    minus     => \&subtract_two_numbers,
    # .....等等
);

sub add_two_numbers      { $_[0] + $_[1] }

sub subtract_two_numbers { $_[0] - $_[1] }

```

.....相比因语言特性而做出这样的决定，到不如说是出于对代码可维护性，或是安全，再或是团队编程风格的考虑。

因间接通过分派表而带来的一个好处是，它对未经验证调用函数提供了一定的保护——调用这些函数安全多了。如果你的分派函数盲目地假设那些字符串直接对应到某操作应该调用的函数名，那么可以想象通过将 `'Internal::Functions::some_malicious_function'` 修整为操作名，一个恶意用户可以调用任何其他名称空间的任何函数。

你也可以在将匿名函数作为参数传递的过程中创建它们：

```

sub invoke_anon_function
{
    my $func = shift;
    return $func->( @_ );
}

sub named_func
{
    say 'I am a named function!';
}

invoke_anon_function( \&named_func );
invoke_anon_function( sub { say 'I am an anonymous function' } );

```

## 匿名函数名称

存在可以鉴别一个引用是指向具名函数还是匿名函数的特例——匿名函数（正常情况下）没有名称。这听上去很玄乎很傻也很明显，内省可以现实这个区别：

```

package ShowCaller;

use Modern::Perl;

```



```

sub show_caller
{
    my ($package, $filename, $line, $sub) = caller(1);
    say "Called from $sub in $package at $filename : $line";
}

sub main
{
    my $anon_sub = sub { show_caller() };
    show_caller();
    $anon_sub->();
}

main();

```

结果可能令人惊讶:

```

Called from ShowCaller::B<main> in ShowCaller at anoncaller.pl : 20
Called from ShowCaller::B<__ANON__> in ShowCaller at anoncaller.pl : 17

```

输出第二行中的 `__ANON__` 展示了匿名函数没有 Perl 可以识别的名称。即使这样会难以调试，但还是有方法可以绕过它的隐匿性。

CPAN 模块 `Sub::Identify` 提供了一系列有用的函数来对传入函数引用的名称进行检查。`sub_name()` 便是不二之选:

```

use Sub::Identify 'sub_name';

sub main
{
    say sub_name( \&main );
    say sub_name( sub {} );
}

main();

```

正如你想象的那样，名称的缺少使得调试匿名函数更加复杂。CPAN 模块 `Sub::Name` 可以帮助你。它的 `subname()` 函数允许你将名称附加在匿名函数上:

```

use Sub::Name;
use Sub::Identify 'sub_name';

my $anon = sub {};
say sub_name( $anon );

my $named = subname( 'pseudo-anonymous', $anon );
say sub_name( $named );
say sub_name( $anon );

say sub_name( sub {} );

```

这个程序产生如下输出:

```

__ANON__
pseudo-anonymous
pseudo-anonymous

```

\_\_ANON\_\_

注意这两个引用都指向同一个底层函数。用 `$anon` 调用 `subname()` 并且将结果 返回给 `$named` 将修改该函数，因此其他指向这个函数的引用将见到相同的名字，即 `pseudo-anonymous`。

## 隐式匿名函数

所有这些匿名函数声明都是显式的。Perl 5 通过原型 (prototypes) 允许隐式匿名函数。虽然这个特性的存在名义上是为了让程序员为诸如 `map` 和 `eval` 编写自己的语法，一个有趣的例子就是对 延迟 函数的使用看上去不像函数那样。考虑 CPAN 模块

`Test::Exception`:

```
use Test::More tests => 2;
use Test::Exception;

throws_ok { die "I croak!" }
           qr/I croak/, 'die() should throw an exception';

lives_ok  { 1 + 1 }
         'simple addition should not';
```

`lives_ok()` 和 `throws_ok()` 都接受一个匿名函数作为它们的第一个参数。这段代码 等价于:

```
throws_ok( B<sub { die "I croak!" },>
           qr/I croak/, 'die() should throw an exception' );

lives_ok( B<sub { 1 + 1 },>
         'simple addition should not' );
```

.....只不过更加易读罢了。

注意隐式版本中匿名函数最后的大括号后 没有 逗号。相比其他一些好用的语法来说，有时候这是一个令人疑惑的疙瘩，是 Perl 5 语法分析器古怪的好意。

这两个函数的实现都不关心你是用何种机制传递函数引用的。你也可以按引用传递一个具 名函数:

```
B<sub croak { die 'I croak!' }>

B<sub add    { 1 + 1 }>

throws_ok B<\&croak>,
           qr/I croak/, 'die() should throw an exception';

lives_ok  B<\&add>,
         'simple addition should not';
```

.....但你 不 能将他们当作标量引用传递:

```
sub croak { die 'I croak!' }

sub add    { 1 + 1 }

B<my $croak = \&croak;>
B<my $add   = \&add;>
```

```
throws_ok B<$croak>,
    qr/I croak/, 'die() should throw an exception';

lives_ok B<$add>,
    'simple addition should not';
```

.....因为原型改变了 Perl 5 语法分析器解释这段代码的方式。在对 `throws_ok()` 或是 `lives_ok()` 的调用进行求值时，它不能 100% 清楚地确定 `$croak` 和 `$add` 含 有 什么内容，因此它会产生一条错误信息：

```
Type of arg 1 to Test::Exception::throws_ok must be block or sub {}
(not private variable) at testex.pl line 13,
near "'die() should throw an exception';"
```

不提缺点，这个特性偶尔也有其用处。虽然通过将裸代码块提升为匿名函数带来了语法上的清晰，但请有节制地使用并小心编写好 API 文档。

## 闭包

你已经见过了函数（functions）和作用域（scope）是如何工作的。你知道了每次控制流程进入函数后，该函数将得到代表本次调用的词法作用域的新环境。你现在可以使用函数引用（references）以及匿名函数（anonymous\_functions）了。

你已经学到了理解闭包所需的全部知识。

Mark Jason Dominus 的著作 *Higher Order Perl* 是有关第一等函数、闭包和利用它们完成令人惊奇的事物的公认参考书。你可以在 <http://hop.perl.plover.com/> 在线阅读这本书。

## 创建闭包

闭包 是封闭于外部词法环境之上的函数。你也许早已创建并使用了闭包，只是没有意识到：

```
{
    package Invisible::Closure;

    my $filename = shift @ARGV;

    sub get_filename
    {
        return $filename;
    }
}
```

这段代码的行为平淡无奇。你也许并未觉察有何特殊之处。显然 函数 `get_filename()` 可以在词法层面见到 `$filename`。作用域就是这样工作的！闭包还可以封闭于 转瞬即逝词法环境上。

假设你想迭代一个列表，而不愿自行管理迭代器。你可以创建一个返回函数的函数，当它被调用时，将返回迭代过程中的下一个元素：

```
sub make_iterator
{
    my @items = @_;
    my $count = 0;

    return sub
```

```

    {
        return if $count == @items;
        return $items[ $count++ ];
    }
}

my $cousins = make_iterator(qw( Rick Alex Kaycee Eric Corey ));

say $cousins->() for 1 .. 5;

```

即使 `make_iterator()` 已经返回，但此匿名函数仍将引用词法变量 `@items` 和 `$count`。它们的值会一直保持 (`reference_counts`)。这个匿名函数，存放于 `$cousins`，在调用 `make_iterator()` 的特别词法环境中闭合于这些值上。

很容易演示词法环境是独立于对 `make_iterator()` 的调用：

```

my $cousins = make_iterator(qw( Rick Alex Kaycee Eric Corey ));
my $aunts   = make_iterator(qw( Carole Phyllis Wendy ));

say $cousins->();
say $aunts->();
say $cousins->();
say $aunts->();

```

因为对 `make_iterator()` 的每次调用都为词法量创建分离的词法环境，匿名子过程就此产生且返回唯一的词法环境。

因为 `make_iterator()` 并非按值或按引用返回这些词法量，其他闭包外的 Perl 代码无法访问它们。他们和其他词法量一样被有效地封装。

多个闭包可以闭合于同一批词法变量上，这是一个偶尔会用到的惯用语，可以为本将全局可见的变量提供一个更好的封装：

```

{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}

```

.....但注意你不可以 嵌套 具名函数。具名函数有着包全局作用域。任一在嵌套函数间共享的词法变量，在外层函数销毁它的第一层词法环境时，将变为非共享 如果你还是不太清楚，可以想像一下它的实现。

CPAN 模块 `PadWalker` 可以让你打破词法封装，但是所有利用该模块破坏你代码的人应有权在没有你帮助的情况下修复所有伴随产生的软件缺陷。

## 闭包的使用

闭包可以构造针对定长列表的高效迭代器，但是他们在迭代那些不适合直接引用其元素的列表时更胜一筹，那写列表不是一次性计算全部元素的代价过于昂贵，就是尺寸太大以至于无法直接塞进内存中去。

考虑一个按需创建 `Fibonacci` 序列的函数。不必递归地重计算这个序列，而应使用缓存并惰性按需计算出各个元素：

```

sub gen_fib
{
    my @fibs = (0, 1, 1);

```

```

return sub
{
    my $item = shift;

    if ($item >= @fibs)
    {
        for my $calc ((@fibs - 1) .. $item)
        {
            $fibs[$calc] = $fibs[$calc - 2] + $fibs[$calc - 1];
        }
    }

    return $fibs[$item];
}
}

```

每次调用由 `gen_fib()` 返回的函数需提供一个参数，即 **Fibonacci** 序列的第 *n* 个元素。该函数按需要生成序列中所有前导值，缓存起来，并且返回所需的元素。它将延后计算过程直到不得不这样做。

如果你所需的全部就是计算 **Fibonacci** 数的话，这个方法也许太过复杂。然而，考虑到函数 `gen_fib()` 可以变得惊人地通用：它初始化一个数组，用于缓存，执行一些定制的代码来填充缓存的各类值，并从缓存中返回已计算的结果。抽掉计算 **Fibonacci** 值的部分，你可以使用这段代码来实行所有带缓存的、惰性迭代器的行为。

换句话说，你可以提取出一个函数，`generate_caching_closure()`，并按该函数的方式重写 `gen_fib()`：

```

sub gen_caching_closure
{
    my ($calc_element, @cache) = @_;

    return sub
    {
        my $item = shift;

        $calc_element->($item, \@cache) unless $item < @cache;

        return $cache[$item];
    };
}

sub gen_fib
{
    my @fibs = (0, 1, 1);

    return gen_caching_closure(
        sub
        {
            my ($item, $fibs) = @_;

            for my $calc ((@$fibs - 1) .. $item)
            {
                $fibs->[$calc] = $fibs->[$calc - 2] + $fibs->[$calc -
1];
            }
        }
    );
}

```

```

    },
    @fibs
  );
}

```

该程序的行为和以往一致，但对高阶函数和闭包的使用允许从 Fibonacci 序列的计算中有效分离出缓存的初始化过程。代码行为的定制——就此而言，`gen_caching_closure()`——通过传递高阶函数，带来了极大的抽象性和灵活性。

在某种意义上，你可以将内置的 `map`、`grep` 以及 `sort` 比作高阶函数，特别是 在你拿他们和 `gen_caching_closure()` 比较时。

## 闭包与部分应用

闭包除了抽象结构化细节外还可以做更多事。它允许你定制特定的行为。从某种意义上讲， 它还可以去掉 不必要的泛化。考虑一例接受若干参数的函数：

```

sub make_sundae
{
  my %args = @_;

  my $ice_cream = get_ice_cream( $args{ice_cream} );
  my $banana    = get_banana( $args{banana} );
  my $syrup     = get_syrup( $args{syrup} );
  ...
}

```

所有这些定制也许和你那位于购物中心内商品齐全的主力店正相称，但如果你在天桥附近有一辆冰淇淋专卖车，那里只出售安在卡文迪什香蕉上的法式香草冰淇淋，那样在调用 `make_sundae()` 时，你必须每次都传递一个恒久不变的参数。

一个名为 部分应用 的技巧将一部分参数绑定给函数以便你可以在后续的调用过程中填入其他值。用闭包来模拟再简单不过了：

```

my $make_cart_sundae = sub
{
  return make_sundae( @_,
    ice_cream => 'French Vanilla',
    banana    => 'Cavendish',
  );
};

```

现在不必调用 `make_sundae()` 了，你可以直接使用 `$make_cart_sundae->()` 并只将相关的参数传入，而无需顾及忘传或错传不变量。你还可以使用来自 CPAN 的 `Sub::Install` 把这个函数直接安装到你的名称空间中。。

## 状态 VS 闭包

闭包 (closures) 是除使用全局变量外，在函数调用间保证数据持续性的简易、有效且安全的方法。如果你需要在具名函数间共享变量，你可以仅为这些函数的声明引入一个新作用域 (scope)：

```

{
  my $safety = 0;

```

```

sub enable_safety { $safety = 1 }
sub disable_safety { $safety = 0 }

sub do_something_awesome
{
    return if $safety;
    ...
}

```

对安全性开关函数进行封装让这三个函数可以共享状态而不必将词法变量直接地暴露给外部代码。在外部代码可以更改内部状态时，这个惯用语可以很好的起到作用，但在状态仅由单个函数维护时，它就显得有些笨拙了。

假设你想统计一下你的冰淇淋小摊接待了多少客人。逢百的客人可以免费加料：

```

{
    my $cust_count = 0;

    sub serve_customer
    {
        $cust_count++;

        my $order = shift;

        add_sprinkles($order) if $cust_count % 100 == 0)

        ...
    }
}

```

这个方法 行得通，但是为单个函数创建新的词法作用域没有那么得必要，它带来了意外的 复杂度。`state` 关键字允许你声明一个词法作用域变量，它的值在调用之间是连续的：

```

sub serve_customer
{
    B<state $cust_count = 0;>
    $cust_count++;

    my $order = shift;
    add_sprinkles($order) if $cust_count % 100 == 0)

    ...
}

```

你必须通过使用类似 `Modern::Perl` 的模块、`feature` 编译命令、或是要求 Perl 的版本必须新于 5.10（例如，`use 5.010;` 或 `use 5.012;`）来明确地启用这个特性。

你也可以在匿名函数内使用 `state`，就像这个计数器的典型例子一样：

```

sub make_counter
{
    return sub

```

```

    {
        B<state $count = 0;>
        return $count++;
    }
}

```

.....虽然没有什么明显的优势让人采用这种方式。

Perl 5.10 不推荐为之前版本所用的一种技术，通过它你可以有效地模拟 `state`。在 `my` 声明中使用求值得假的后缀式条件表达式可以避免将一个词法变量 重新初始化 为 `undef` 或初始化过的值。通过肆意使用这个实现方式，一个具名函数可以闭合于它 之前的词法作用域。

现在，任何对改动词法变量声明的后缀条件表达式的使用都会因不推荐而产生警告。采用这种技巧时，很容易在无意中写出错误百出的代码；请在可能时用 `state` 代替，否则使用闭包。避免使用这个惯用语，但在遇到时也应该能读懂它：

```

sub inadvertent_state
{
    # 不推荐；请勿使用
    my $counter = 1 if 0;

    ...
}

```

## 属性

Perl 中的具名实体——变量和函数——可按 属性 的形式附加额外的元信息。属性是名称（通常，也是值），它允许某种形式的元编程（`code_generation`）。

定义属性可能有点别扭，对其有效地使用是科学更是艺术。它们有充足的理由不在大多数程序中出现，虽然它们 可以 在代码维护和清晰度上带来引人注目的好处。

## 使用属性

用最简单的形式来说，属性是一个附加于变量或函数声明上的前置冒号标识符：

```

my $fortress      B<:hidden>;

sub erupt_volcano B<:ScienceProject> { ... }

```

如果合适类型（分别是标量和函数）的属性处理器存在，这些声明将引发对名为 `hidden` 和 `ScienceProject` 的属性处理器的调用。如果合适的处理器不存在，Perl 会抛出编译期异常。这些处理器可以做 任何事。

属性可以包括一个参数列表；Perl 将它们作为一个常量字符串列表，即使它们可能会类似于其他值，如，数字或变量。来自 CPAN 的 `Test::Class` 模块很好地利用了这一参数形式：

```

sub setup_tests :Test(setup) { ... }

sub test_monkey_creation :Test(10) { ... }

sub shutdown_tests :Test(teardown) { ... }

```

`Test` 属性标识了包括测试断言的方法，并且可选地标识了意图执行的断言方法的数量。虽然对这些类的内省（`reflection`）可以发现合适的测试方法，当给出经良好设计的启发 式方法，`:Test` 属性使得代码及其意图毫无歧义。

`setup` 和 `teardown` 参数允许测试类定义它们自己的支持方法而不必担心重名或其他



因继承或其余设计上的考虑而引发的冲突。你可以强制施行某种设计，其中所有测试类必须自行覆盖名为 `setup()` 和 `teardown()` 的方法，但是属性方式给与实现更多的灵活性。

Catalyst Web 框架也采用属性来决定 Web 应用内方法的可见性和行为。

## 属性的缺点

属性确实有它们的缺点：

- \* 属性的正式编译命令（即 `attributes`）已经将其接口列为“实验性质”很多年了。Damian Conway 的核心模块 `Attribute::Handlers` 简化了它们的实现。请在可能时采用它而非 `attributes`；
- \* 任何声明属性处理器的模块必须继承 `Attribute::Handlers` 以使处理器对所有用到它们的包可见。你也可以将它们存放在 `UNIVERSAL`，但这是对全局的污染，还是一种糟糕的设计。。此缺点归根于 Perl 5 自身对属性的实现方式；
- \* 属性处理器生效于 `CHECK` 代码块，使得它们不适合用于那些修改语法分析及编译顺序的项目，例如 `mod_perl`；
- \* 任何提供给属性的参数是一列常量字符串。`Attribute::Handlers` 进行部分数据转换，但偶尔你需要禁用它。

属性中最差的一点就是它们会远距离产生奇怪的语法动作。给出一段使用属性的代码，你能否预言它的行为？良好而准确的文档可以帮助你，但如果一段看上去很无辜的词法变量声明将此变量的引用存放他处，则你对销毁该变量内容的期待将是错误的，除非你非常仔细地阅读文档。类似的，处理器可能会用一个函数包装另一个函数并在你不知情的情况下在符号表里替换了它——考虑一个自动调用 `Memoize` 的 `:memoize` 属性。

复杂的特性可以产生紧凑和惯用语化的代码。Perl 允许开发者实验多种设计以便找到他们想法的最佳表示。属性和其他高级 Perl 特性可以帮助你解决复杂的问题，但是他们也会混淆原本简单的代码意图。

大多数程序绝对不会用到这个特性。

## AUTOLOAD

你没有必要为调用而定义每一个函数和方法。Perl 提供了一种机制，通过它你可以截获不存在方法的调用。这样你就可以只定义所需的函数或提供有趣的错误信息和警告。

考虑如下程序：

```
#!/perl

use Modern::Perl;

bake_pie( filling => 'apple' );
```

当运行它时，Perl 将因调用未定义的函数 `bake_pie()` 而抛出一个异常。现在添加一个名为 `AUTOLOAD()` 的函数：

```
sub AUTOLOAD {}
```

除了错误不再出现，没有发生任何明显改变。在某个包中，名为 `AUTOLOAD()` 函数的出现告诉 Perl 无论是正常分派或方法缺失时都调用它。可以将 `AUTOLOAD()` 稍作修改，显示一个消息来演示这一点：

```
sub AUTOLOAD { B<say 'In AUTOLOAD()!>' }
```

## AUTOLOAD 的基本功能

AUTOLOAD() 函数直接在 @\_ 中接到传递给未定义函数的参数。你可以按喜好修改 这些参数:

```
sub AUTOLOAD
{
    # 将参数美化输出
    B<local $" = ', ';>
    B<say "In AUTOLOAD(@_)!">
}
```

未定义函数的 名称 可以从伪全局变量 \$AUTOLOAD 得到:

```
sub AUTOLOAD
{
    B<our $AUTOLOAD;>

    # 将参数美化输出
    local $" = ', ';>
    say "In AUTOLOAD(@_) B<for $AUTOLOAD>!"
}
```

our 声明 (our) 将此变量的作用域限制在 AUTOLOAD() 的代码体内。这个变量包含了未定义函数的完全限定名称。就此例来说, 这个函数是 main::bake\_pie。一个常见的惯用语可以用来去掉包名:

```
sub AUTOLOAD
{
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)$/i>

    # 将参数美化输出
    local $" = ', ';>
    say "In AUTOLOAD(@_) B<for $name>!"
}
```

最终, 无论 AUTOLOAD() 返回什么, 最初的调用都会收到:

```
say secret_tangent( -1 );

sub AUTOLOAD { return 'mu' }
```

目前为止, 这些例子只是截获了对未定义函数的调用。还有其他的路可走。

## 在 AUTOLOAD() 中重分派方法

面向对象编程中一个常见的模式就是将某方法 委托 或 代理 给另一个对象, 通常包含在前者内或可以从中访问到。这是一个记录日志有趣且有效的方法:

```
package Proxy::Log;

sub new
{
    my ($class, $proxied) = @_;
    bless \ $class, $proxied;
}
```

```

sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}

```

这段 `AUTOLOAD()` 的代码记录方法调用。它的神奇之处就在于一个简单的模式；它从 经  `bless` 的标量引用中对被代理对象进行解引用，提取未定义方法的名称，然后调用被代理对象中的方法，传递参数给它。

在 `AUTOLOAD()` 中生成代码

那个双重分派的技巧很有用，但比要求的要慢一些。每一次代理的方法调用必须通过常规分派，失败，最后落入 `AUTOLOAD()`。因为程序需要它们，作为代替，你可以将新方法安装到代理类中：

```

sub AUTOLOAD
{
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)/;>

    my $method = sub
    {
        Log::method_call( $name, @_ );

        my $self = shift;
        return $self->$name( @_ );
    }

    B<no strict 'refs';>
    B<*{ $AUTOLOAD } = $method;>
    return $method->( @_ );
}

```

之前 `AUTOLOAD()` 的代码体变为了一个匿名函数。这段代码创建绑定未定义函数 名称 的闭包（`closures`）。接着它在合适的符号表内安装该闭包，使得方法的后续分派能够找到已经创建的闭包而避开 `AUTOLOAD()`。最后，它直接调用该方法，返回结果。

虽然这个方法更为清爽且几乎总是比直接在 `AUTOLOAD()` 处理调用行为来得更透明，但被 `AUTOLOAD()` 调用 的代码也许会检测到分派过程经过了 `AUTOLOAD()`。简单说来，`caller()` 将反映出目前两种双重分派的技巧。这便可能成为一个问题；诚然你可以争辩这是一种打破闭包的行为，必须被关注，但是让一个对象 如何 提供方法的内幕泄露到宽广的世界中去，也算得上是违反闭包原则了。

另一种惯用语就是使用尾部调用（`tailcalls`）从 `caller()`（调用者的）“记忆”中把当前对 `AUTOLOAD()` 的调用 替换 为目标方法：

```

sub AUTOLOAD
{
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)/;>

    my $method = sub { ... }
}

```

```

        no strict 'refs';
        *{ $AUTOLOAD } = $method;
        B<goto &$method;>
    }

```

这样做和直接调用 `$method` 等效。`AUTOLOAD()` 不会出现在 `caller()` 的调用列表中，因此看上去就像生成的方法被直接调用一般。

## AUTOLOAD 的缺点

在某些情况下，`AUTOLOAD()` 的确是一个有用的工具，但它也会变得难以正确使用。请考虑使用其他的技巧，比如 `Moose` 及其他抽象来代替。

运行时生成方法的朴素手段意味着 `can()` 方法将不能正确地报告有关类和对象能力的相关信息。你可以用几种方法解决这个问题；其中最简单的一个就是用 `subs` 编译命令预定义所有打算让 `AUTOLOAD` 处理的函数：

```
use subs qw( red green blue ochre teal );
```

这个技巧的好处是可以让你记录下你的意图，但坏处则是你必须维护一个静态的函数、方法名称表。

你也可以提供你自己的 `can()` 方法来生成合适的函数：

```

sub can
{
    my ($self, $method) = @_;

    # 是用上级 can() 方法的结果
    my $meth_ref = $self->SUPER::can( $method );
    return $meth_ref if $meth_ref;

    # 添加过滤器
    return unless $self->should_generate( $method );

    $meth_ref = sub { ... };
    no strict 'refs';
    return *{ $method } = $meth_ref;
}

sub AUTOLOAD
{
    my ($self) = @_;
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/i>

    return unless my $meth_ref = $self->can( $name );
    goto &$meth_ref;
}

```

根据需求的复杂程度，你也许会发现维护一个类似包含所缺方法名称的包作用域哈希之类的数据结构更为简单。

注意某些原本不欲提供的方法也可能通过 `AUTOLOAD()`。一个常见的罪犯便是 `DESTROY()`，对象的析构器。最简便的方法就是提供一个不含实现的 `DESTROY()` 方法；Perl 将高兴地分派此方法并将 `AUTOLOAD()` 一起忽略：

```
# 跳过 AUTOLOAD()  
sub DESTROY {}
```

如 `import()`、`unimport()` 和 `VERSION()` 等特殊方法绝不会通过 `AUTOLOAD()`。

如果你在一个继承某自行提供 `AUTOLOAD()` 的包的名称空间内混用函数和方法，你可能会得到一个怪异的错误消息：

```
Use of inherited AUTOLOAD for non-method I<slam_door>() is deprecated
```

这会在当你尝试调用一个包内不存在的方法而所继承的类有包含它自己的 `AUTOLOAD()` 时。这绝不是你的意图。这个问题由多个原因混合而成：在单个名称空间内混用函数和方法通常是一个设计缺陷，继承以及 `AUTOLOAD()` 会很快地变得复杂，并且，在你不知道对象可以执行何种方法时，推理一段代码是很困难的。