

第六章 子过程

- ↓ 第六章 子过程
 - ↓ 1.0 语法
 - ↓ 2.0 语意
 - ↓ 2.1 参数列表的技巧
 - ↓ 2.2 错误指示
 - ↓ 2.3 范围问题
 - ↓ 3.0 传入引用
 - ↓ 4.0 函数原型
 - ↓ 4.1 内联常量函数
 - ↓ 4.2 谨慎使用函数原型
 - ↓ 5.0 子过程属性
 - ↓ 5.1 Locked 和 method 属性
 - ↓ 5.3 左值属性

象其他的语言一样，Perl 也支持自定义的子过程。（注：我们也把它们叫做函数，不过函数和子过程在 Perl 里是一样的东西。有时候我们甚至叫它们方法，方法和函数或子过程是同样的方式定义的，只是调用方式不同。）这些子过程可以在主程序中的任何地方定义，也可以用 **do**、**require** 或 **use** 关键字从其他文件中加载。或者直接使用 **eval** 在运行的时候产生。你甚至可以使用第十章"包"中"自动装载"一节描述的机制在运行时加载它们。你可以间接调用子过程，使用一个包含该子过程名字或包含指向该子过程引用的变量来调用，或者通过对象，让对象决定调用哪个子过程。你可以产生只能通过引用使用的匿名子过程，如果必要，你还可以通过闭合，用匿名子过程克隆几乎相同的函数。我们将在第八章"引用"中的相关小节中讲述。

1.0 语法

声明一个命名子过程，但不定义它，使用下面的形式：

```
sub NAME
sub NAME PROTO
sub NAME      ATTRS
sub NAME PROTO ATTRS
```

声明并且定义一个命名子过程，加上一个 **BLOCK**：

```
sub NAME          BLOCK
sub NAME PROTO    BLOCK
sub NAME      ATTRS BLOCK
sub NAME PROTO ATTRS BLOCK
```

创建一个匿名子过程或子句，把 **NAME** 去掉就可以：

```
sub          BLOCK
sub      PROTO    BLOCK
sub      ATTRS BLOCK
sub      PROTO ATTRS BLOCK
```

PROTO 和 **ATTRS**表示原型和属性，分别将在本章下面的章节中讨论。相对于 **NAME** 和 **BLOCK** 它们并不很重要。**NAME** 和 **BLOCK** 是基本部分，甚至有时候它们也可以省略。

对于没有 **NAME** 的形式，你还必须提供调用子过程的方法。因此你必须保存返回值，因为这种形式的 **sub** 声明方法不但在编译的时候编译，同时也产生一个运行时的返回值，所以我们可以保证保存它：

```
$subref = sub BLOCK;
```

可以用下面的方法引入在另一个模块中定义的子过程:

```
use MODULE qw(NAME1 NAME2 NAME2...)
```

直接调用子过程可以用下面的方法:

```
NAME (LIST)      # 有圆括弧时 & 是可选的
NAME LIST        # 如果预声明/输入了子过程, 那么圆括弧是选的
&NAME            # 把当前的 @_ 输出到该子过程
                  # (并且绕开原型)。
```

间接调用子过程(通过名字或引用), 可以使用下面的任何一种方法:

1. `&$subref(LIST)` # 在间接调用的时候, `&` 不能忽略
2. `$subref->(LIST)` # (除非使用中缀表示法)
3. `&$subref` # 把当前的 `@_` 输出到该子过程

正式情况下, 一个子过程的名字包括 `&` 前缀, 一个子过程可以使用 `&` 前缀调用, 但通常情况下 `&` 是可选的, 如果预先定义了子过程, 那么圆括弧也是可选的. 但是, 在只使用子过程名字的时候, `&` 不能省略, 例如当子过程名字被用做一个参数来判断是否它已经定义过的时候, 或者当你使用 `$subref = \&name` 来获取一个命名子过程的引用的时候. 同样, 当你使用 `&$subref()` 或 `&{$subref()}` 进行一个间接子过程调用的时候也不能省略 `&`. 不过, 如果使用一种更方便的形式 `$subref->()`, 则不需要 `&`. 参看第八章, 那里有更多有关子过程引用的内容.

Perl 并不强制子过程名字使用大写风格. 但是按惯例由 perl 的运行时系统间接调用的函数都是大写的 (`BEGIN`, `CHECK`, `INIT`, `END`, `AUTOLOAD`, `DESTROY`, 和所有第十四章 "捆绑变量"涉及到的函数). 因此你应该避免使用这种大写风格. (但是操作常量值的子过程通常也写成大写的).

2.0 语意

在你记住所有语法前, 你只需要记住下边这种定义子过程的普通方法:

```
sub razzle {
    print "Ok, you've been razzled.\n";
}
```

和调用子过程的正常方法就是:

```
razzle();
```

在上边的写法中, 我们省略了输入(参数)和输出(返回值). 但是 Perl 向子过程中传入数据和子过程传出数据的方法非常简单: 所有传入的参数被当成单个平面标量列表, 类似的多个返回值也被当成单个平面标量列表返回给调用者. 当使用任意 `LIST` 时也一样, 任何传入的数组或散列的值都代换到一个平面的列表里面, 同时也失去了它们的标识, 不过有几种方法可以绕开这个问题, 这种自动的列表代换在很多场合非常有用. 参数列表和返回值列表都可以根据你的需要包含任意多个标量成员(当然你可以使用原型定义来约束参数的类型). 实际上, Perl 是按照支持可变参函数(可以支持任何数量的参数)概念来设计的. C 则不同, 虽然 C 也勉强支持一些变参的函数, 例如 `printf (3)`.

现在, 如果你将设计一种可以支持不定数量的任意参数的语言, 你最好让你的语言在处理这些任意长的参数列表上容易些. 所有传入 Perl 过程的参数都是以 `@_` 身份传入的. 如果你调用一个有两个参

数的函数，它们在函数内部可以作为 `@_` 数组的前两个成员访问：`$_[0]` 和 `$_[1]`。因为 `@_` 只是一个有着奇怪名字的普通数组，所以你可以象处理普通数组一样随意处理它。（注：这个领域是 Perl 和传统的编程语言冲突得最厉害的地方。）数组 `@_` 是一个本地数组，但是它的值是实际标量参数的别名(通常称为引用传参)因而如果修改了 `@_` 中的成员那么同时也修改了对应的实际参数的值。(通常的语言中很少这么做，但是采用这种方法在 Perl 中可以很容易的返回所需要的值)。

子过程(其他的程序块也一样)的返回值是过程最后一个表达式的值。或者你可以在子过程的任何一个地方明确使用一个 `return` 语句来返回值并且退出子过程。不管是那种方法，当在一个标量或列表环境中调用子过程时，最后一个表达也将同样的标量或列表环境中求值。

2.1 参数列表的技巧

Perl 没有命名的正式参数，但是在实际中你可以将 `@_` 的值拷贝到一个 `my` 列表，这样就可以方便使用这些正式参数(不一样的是，这样拷贝就将引用传参的语义变为了传值传参，也许传值传参正是很多用户通常希望参数被处理的方法，即使他们不知道这些计算机术语)，下面是一个典型的例子：

```
sub aysetenv {
    my ($key, $value) = @_;
    $ENV{$key} = $value unless $ENV{$key};
}
```

但是没人要你一定要给你的参数命名，这就是 `@_` 数组的全部观点。例如，计算一个最大值，你可以简单直接遍历 `@_` 数组：

```
sub max {
    $max = shift(@_);
    for my $item (@_) {
        $max = $item if $max < $item;
    }
    return $max;
}
```

```
$bestday = max($mon, $tue, $wed, $thu, $fri);
```

或者你可以一次将 `@_` 填入一个散列：

```
sub configuration {
    my %options = @_;
    print "Maximum verbosity.\n" if $options{VERBOSE} == 9;
}
```

```
configuration(PASSWORD => 'xyzyz', VERBOSE => 9, SOCRE => 0);
```

下面是一个例子，这里不命名正式参数，这样你可以修改实际参数的值：

```
uppercase_in($v1, $v2);    # 这里改变 $v1 和 $v2
sub uppercase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

但是你不允许用这种方法修改常量，如果一个参数是一个象 "hobbit" 这样的实际标量值或象 `$1` 这样只读标量，当你试图修改它时，Perl 会抛出一个例外(可能的致命错误或者可能的威胁)。例如，下面的例子将不能工作：

```
uppercase_in("fredrick");
```

如果将 `upcase_in` 函数写成返回它的参数的一个拷贝会比直接改变参数安全得多:

```
($v3, $v4) = upcase($v1, $v2);
sub upcase {
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    # 检查我们是否在列表环境中被调用的
    return wantarray ? @parms : $parms[0];
}
```

注意这个函数(没有原型)并不在意传进来的参数是真的标量还是数组。Perl 将所有的参数粉碎成一个又大又长的平面 `@_` 数组列表。这是 Perl 简单传参方式闪光的地方之一。甚至我们可以给它象下面这样的参数的时候都不需要修改 `upcase` 的定义, `upcase` 将照样工作得很棒:

```
@newlist = upcase(@list1, @list2);
@newlist = upcase( split /:/, $var);
```

但是, 如果象下边这样用, 就不会得到你想要的的结果:

```
(@a, @b) = upcase( @list1, @list3);    # 错
```

因为, 和 `@_` 一样, 返回列表同样是一个平面列表。因此所有的返回值将存储在 `@a` 中, `@b` 是空的。可以在下面"传递引用"部分看到替代的办法。

2.2 错误指示

如果你希望你的函数能够以特定的方式返回, 使调用者能够得知发生了一个错误。在 Perl 中实现这个目的最自然的一种方法就是用一个不带参数的 `return` 语句。这样当函数在标量环境中使用时, 调用者得到一个 `undef`, 如果在列表环境中使用, 调用者得到一个空列表。

特殊情况下, 你可以选者产生一个例外来指示错误, 但是必须谨慎使用这种方法。因为你的程序将被例外处理程序终结。例如, 在一个文件操作函数中, 打开文件失败几乎不是例外的事件。因此, 最好能够忽略这种失败。当你在无效的环境下调用函数时, `wantarray` 内建函数将返回 `undef`。因此如果你想忽略它, 你可以使用下面的方法:

```
if($something_went_away) {
    return if defined wantarray;    # 很好, 不是空环境
    die "Pay attention to my error, you danglesocket!!!\n";
}
```

2.3 范围问题

因为每次调用都有自己的参数数组, 因此子过程可以递归调用, 甚至可以调用它自己。如果使用 `&` 的形式调用子过程, 那么参数列表是可选的。如果使用了 `&` 并且省略了参数列表, 那么有一些特殊的规则: 调用过程中的 `@_` 数组将做为被调用子过程的参数。新用户可能不想使用这种有效的机制。

```
&foo(1,2,3)    # 传递三个参数
foo(1,2,3)     # 和上面一样

foo();         # 传递一个空列表
&foo();        # 和上面一样

&foo;          # foo() 获取当前的参数, 和 foo(@_) 一样, 但更快!
foo;           # 如果预定义了子过程 foo, 那么和 foo() 一样, 否则
               # 就是光字 "foo"
```

使用 **&** 形式调用子过程不仅可以省略掉参数列表，同时对你提供的参数也不进行任何原型检查。这种做法一部分是因为历史原因形成，另一部分原因是为了在用户清楚自己在干什么的情况下提供一个方便的办法。你可以参看本章后面的"原型"小节。

在函数中访问一个并没有定义成该函数私有的变量不一定是全局变量；它们遵循第二章 "集腋成裘"中"名字"一节中提到的块作用范围规则，这意味着他们首先在词法作用范围里面决定该变量，然后才扩展到单个包作用范围。从子过程的角度看来，任何在一个闭合的词法作用域中的 **my** 变量仍然优先使用。

例如，下面例子中的 **bumpx** 函数使用了文件作用范围中的 **\$x** 变量，这是因为 **my** 变量被定义的作用范围 --- 也就是文件本身 --- 并没有在定义子过程之前结束。

```
# 文件顶部
my $x = 10;           # 声明和初始化变量
sub bumpx { $x++ }    # 函数可以看到外层词法变量
```

C 和 **C++** 程序员很可能认为 **\$x** 是一个"文件静态"变量。它对其他文件中的函数是私有的，但是在上例中在 **my** 后面定义的函数可以透视到这个变量。那些由 **C** 程序员转变而来的 **Perl** 程序员在 **Perl** 中找不到他们熟悉的文件或函数的"静态变量"。**Perl** 程序员避免使用 **"static"**这个词。因为静态系统过时而且乏味，并且因为在历史使用中这个词被搞得一团糟。

虽然 **Perl** 语法中没有包括**"static"**这个词，但是 **Perl** 程序员同样能够创建函数的私有变量，并且保持跨函数访问。**Perl** 中没有专门的词来表述他们。利用 **Perl** 丰富的作用范围规则结合自动内存管理，就可以有很多方式实现**"static"**关键字的功能。

词法变量并不会只是因为退出了它们的作用范围后就被自动内存垃圾收集回收，它们要等到不再使用后才被回收，这个概念十分重要。为了创建一个在跨函数调用中不被重置的私有变量，你可以将整个函数用花括弧括起来，并将 **my** 定义和函数定义放入该函数块中。你甚至可以放入多个函数定义，这样该私有变量就可以被这些函数共享访问。

```
{
    my $counter = 0;
    sub next_counter { return ++$counter }
    sub prev_counter { return --$counter }
}
```

通常，对词法变量的访问被限制在同一个词法作用域中。两个函数的名字可以被全局访问（在同一个包内），并且因为它们是在 **\$counter** 的作用域中定义的，它们仍然可以访问该变量，即使其他函数访问不到也无妨。

如果这个函数是通过 **require** 或 **use** 加载的，那么也可以。如果它全在主程序中，那么你就要确保使任何运行时的 **my** 赋值要足够地早，你可以将整个程序块放在主程序的最前边，也可以使用 **BEGIN** 或 **INIT** 程序块来保证它在你的程序之前运行：

```
BEGIN {
    my @scale = ('A' .. 'G');
    my $note = -1;
    sub next_pitch { return $scale[ ($note += 1) %= @scale ] };
}
```

BEGIN 既不会影响子过程的定义，也不会影响子过程里使用的任意词法的一致性。这里它仅仅保证在子程序被调用之前变量就被初始化。想了解定义私有变量和全局变量更多的内容，请分别参考29章"函数"的 **my** 和 **our** 的说明，**BEGIN** 和 **INIT** 在第十八章"编译"中解释。

3.0 传入引用

如果你想在函数中传入或传出不止一个的数组或散列结构，同时你希望它们保持它们的一致性，那么你就需要使用一个更明确的传递引用的机制。在你使用传递引用之前，你需要懂得第八章里有关引用的细节。本小节不着重讲述引用的内容。

这里有几个简单的例子，首先，让我们定义一个函数，这个函数使用数组的引用作为参数。当这个数组非常大时，作为一个引用传递要比传入一长列值要快得多：

```
$total = sum (\@a );

sub sum {
    my ($aref) = @_;
    my ($total) = 0;
    foreach (@$aref) { $total += $_ }
    return $total;
}
```

下面让我们将几个数组传入一个函数，并且使用 `pop` 得到每个数组的最后一个元素，并返回每个数组最后一个元素组成的一个新的数组：

```
@tailings = popmany (\@a, \@b, \@c, \@d );

sub popmany {
    my @retlist = ();
    for my $aref (@_) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

下面是一个函数，能够返回一个列表，这个列表包含在每个传入的散列结构中都出现的键字。

```
@common = inter (\%foo, \%bar, \%joe );
sub inter {
    my %seen;
    for my $href (@_) {
        while (my $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

这里我们只用了普通的列表返回机制。当你想传送或返回一个散列结构时会发生什么？如果你仅用其中的一个，或者你不在意它们连在一起，那么使用普通的调用方法就行了，如果不是，那么就会稍微复杂一些。

我们已经在前面提到过，人们常会在下面的写法中遇到麻烦：

```
(@a, @b) = func(@c, @d);
```

或这里：

```
(%a, %b) = func(%c, %d);
```


这些表达式将不会正确工作，它只会设置 `@a` 或 `%a`，而 `@b` 或 `%b` 则是空的。另外函数不会得到两个分离的数组和散列结构作为参数：它和往常一样从 `@_` 中得到一个长列表。

你也许想在函数的输入和输出中都使用引用。下面是一个使用两个数组引用作为参数的函数，并且根据数组中包含元数的多少为顺序返回两个数组的引用：

```
($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

如何向函数传入或传出文件句柄或目录句柄，请参阅第八章的"文件句柄引用"和"符号表句柄"小节。

4.0 函数原型

Perl 可以让你定义你自己的函数，这些函数可以象 Perl 的内建函数一样调用。例如 `push (@array, $item)`，它必须接收一个 `@array` 的引用，而不仅仅是 `@array` 中的值，这样这个数组才能够被函数改变。函数原型能够让你声明的子过程能够象很多内建函数一样获得参数，就是获得一定数目和类型的参数。我们虽然称之为函数原型，但是它们的运转更像调用环境中的自动模板，而不仅仅是 C 或 java 程序员认为的函数原型。使用这些模板，Perl 能够自动添加隐含的反斜杠或者调用 `scalar`，或能够使事情能变成符合模板的其他一些操作。比如，如果你定义：

```
sub mypush (@@);
```

那么 `mypush` 就会象 `push` 一样接受参数。为了使其运转，函数的定义和调用在编译的时候必须是可见的。函数原型只影响那些不带 `&` 方式调用的函数。换句话说，如果你象内建函数一样调用它，它就像内建函数一样工作。如果你使用老式的方法调用子过程，那么它就象老式子过程那样工作。调用中的 `&` 消除所有的原型检查和相关的环境影响。

因为函数原型仅仅在编译的时候起作用，自然它对象 `\&foo` 这样的子过程引用和象 `&{$subref}` 和 `$subref->()` 这样的间接子过程调用的情况不起作用。同样函数原型在方法调用中也不起作用。这是因为被调用的实际函数不是在编译的时候决定的，而是依赖于它的继承，而继承在 Perl 中是动态判断的。

因为本节的重点主要是让你学会定义象内建函数一样工作的子过程，下面使一些函数原型，你可以用来模仿对应的内建函数：

声明为	调用
<code>sub mylink (\$\$)</code>	<code>mylink \$old, \$new</code>
<code>sub myreverse (@)</code>	<code>myreverse \$a, \$b, \$c</code>
<code>sub myjoin (\$@)</code>	<code>myjoin ":", \$a, \$b, \$c</code>
<code>sub mypop (\@)</code>	<code>mypop @array</code>
<code>sub mysplce (\@\$\$@)</code>	<code>mysplce @array, @array, 0, @pushme</code>
<code>sub mykeys (\%)</code>	<code>mykeys %(\$hashref)</code>
<code>sub mypipe (**)</code>	<code>mypipe READHANDLE, WRITEHANDLE</code>

sub myindex (\$;\$)	myindex &getstring, "substr"
	myindex &getstring, "substr", \$start
sub mysyswrite (*\$;\$)	mysyswrite UTF, \$buf
	mysyswrite UTF, \$buf, length(\$buf)-\$off, \$off
sub myopen (*;\$@)	myopen HANDLE
	myopen HANDLE, \$name
	myopen HANDLE, "- ", @cmd
sub mygrep (&@)	mygrep { /foo/ } \$a, \$b, \$c
sub myrand (\$)	myrand 42
sub mytime ()	mytime

任何带有反斜杠的原型字符(在上表左列中的圆括弧里)代表一个实际的参数(右列中有示例) 必须以以这个字符开头.例如 **keys** 函数的第一个参数必须以 **%** 开始, 同样 **mykeys** 的第一个参数也必须以 **%** 开头.

分号将命令性参数和可选参数分开. (在 **@** 或 **%** 前是多余的, 因为列表本身就可以是空的). 非反斜杠函数原型字符有特殊的含义. 任何不带反斜杠的 **@** 或 **%** 会将实际参数所有剩下的参数都吃光并强制进入列表环境. (等同于语法描述中的 **LIST**). **\$** 代表的参数强迫进入标量环境. **&** 要求一个命名或匿名子过程的引用.

函数原型中的 ***** 允许子过程在该位置接受任何参数, 就像内建的文件句柄那样: 可以是一个名字, 一个常量, 标量表达式, 类型团或者类型团的引用. 值将可以当成一个简单的标量或者类型团(用小写字母的)的引用由于子过程使用. 如果你总是希望这样的参数转换成一个类型团的引用, 可以使用 **Symbol::qualify_to_ref**, 象下面这样:

```
use Symbol 'qualify_to_ref';

sub foo (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

注意上面表中的最后三个例子会被分析器特殊对待, **mygrep** 被分析成一个真的列表操作符, **myrand** 被分析成一个真的单目操作符就象 **rand** 一样, 同样 **mytime** 被分析成没有参数, 就象 **time** 一样.

也就是说, 如果你使用下面的表达式:

```
mytime +2;
```

你将会得到 **mytime()+2**, 而不是 **mytime(2)**, 这就是在没有函数原型时和使用单目函数原型时分析得到的不同结果.

mygrep 例子同样显示了当 **&** 是第一个参数的时候是如何处理的. 通常一个 **&** 函数原型要求一个象 **\&foo** 或 **sub{}** 这样参数. 当它是第一个参数时, 你可以在你的匿名子过程中省略掉 **sub**, 只在"非直接对象"的位置上传送一个简单的程序块(不带冒号). 所以 **&** 函数原型的一个重要功能就是你可以用它生成一个新语法, 只要 **&** 是在初始位置:

```
sub try (&$) {
    my ($try, $catch) = @_;
    eval { &$try };
    if ($?) {
```



```

        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
}      # 不是函数调用的结尾!
catch {
    /phooey/ and print "unphooey\n";
};

```

它打印出 "unphooey". 这里发生的事情是这样的, Perl 带两个参数调用了 **try**, 匿名函数 {**die** "phooey";} 和 **catch** 函数的返回值, 在本例中这个返回值什么都不是, 只不过是它自己的参数, 而整个块则是另外一个匿名函数. 在 **try** 里, 第一个函数参数是在 **eval** 里调用的, 这样就可以捕获任何错误. 如果真的出了错误, 那么调用第二个函数, 并且设置 **\$_** 变量以抛出例外. (注: 没错, 这里仍然有涉及 **@_** 的可视性的问题没有解决. 目前我们忽略那些问题. 但是如果我们将来把 **@_** 做成词法范围的东西, 就象现在试验的线程化 Perl 版本里已经做的那样, 那么那些匿名子过程就可以象闭合的行为一样.) 如果你觉得这些东西听起来象胡说八道, 那么你最好看看第二十九章里的 **die** 和 **eval**, 然后回到第八章里看看匿名函数和闭合. 另外, 如果你觉得麻烦, 你还可以看看 CPAN 上的 **Error** 模块, 这个模块就是实现了一个用 **try**, **catch**, **except**, **otherwise**, 和 **finally** 子句的灵活的结构化例外操作机制.

下面是一个 **grep** 操作符的重新实现(当然内建的实现更为有效):

```

sub mygrep (&@) {
    my $coderef = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$coderef;
    }
    return @result;
}

```

一些读者希望能够看到完整的字母数字函数原型. 我们有意把字母数字放在了原型之外, 为的是将来我们能够很快地增加命名的, 正式的参数. (可能) 现在函数原型的主要目的就是让模块作者能够对模块用户作一些编译时的强制参数检查.

4.1 内联常量函数

带有 () 的函数原型表示这个函数没有任何参数, 就象内建函数 **time** 一样. 更有趣的是, 编译器将这种函数当作潜在的内联函数的候选函数. 当 Perl 优化和常量值替换回合后, 得到结果如果是一个固定值或者是一个没有其他引用的语法作用域标量时, 那么这个值就将替换对这个函数的调用. 但是使用 **&NAME** 方式调用的函数不被"内联化", 然而, 只是因为它们不受其他函数原型影响. (参看第三十一章"用法模块"中的 **use constant**, 这是一种定义这种固定值的更简单的方法).

下面的两种计算 π 的函数写法都会被编译器"内联化":

```

sub pi () { 3.14159 }      # 不准确, 但接近
sub PI () { 4 * atan2(1, 1) } # 和它的一样好

```

实际上, 下面所有的函数都能被 Perl "内联化", 因为 Perl 能够在编译的时候就能确定所有的值:

```

sub FLAG_FOO () { 1 << 8 }

```

```

sub FLAG_BAR ()      { 1 << 9 }
sub FLAG_MASK ()     { FLAG_FOO | FLAG_BAR }

sub OPT_GLARCH ()    { (0x1B58 & FLAG_MASK) == 0 }
sub GLARCH_VAL ()    {
    if (OPT_GLARCH) { return 23 }
    else           { return 42 }
}

sub N () { int(GLARCH_VAL) / 3 }
BEGIN {
    # compiler runs this block at compile time
    my $prod = 1;      # persistent, private variable
    for (1 .. N) { $prod *= $_ }
    sub NFACT () { $prod }
}

```

最后一个例子中，**NFACT** 函数也将内联化，因为它有一个空的函数原型并且函数返回的变量并没有被函数修改，而且不能被其他东西改变，因为它在一个语法作用范围里面。因此编译器在编译的时候预先计算它的值，并用这个值替换所有使用 **NFACT** 的地方。

如果你重新定义已经被内联化的子过程，那么你会收到一个命令性警告(你可以使用这个警告来确认一个子过程是不是已经被内联化了)因为重新定义的子过程会用先前编译产生的值代替，因此这个警告足够的确定这个子过程是否被内联化。如果你需要重新定义子过程，你可以通过删除 **()** 函数原型(这个更改调用方法)或者重新修改函数的写法来阻挠内联化机制来避免子过程被内联化。例如：

```

sub not_inlined () {
    return 23 if $$;
}

```

参看第十八章学习更多有关程序编译和执行阶段的知识。

4.2 谨慎使用函数原型

最好在新函数中使用函数原型，而不在旧函数中使用函数原型。Perl 中函数原型是环境模板，而不象 **ANSI C** 中的函数原型，因此你必须十分注意函数原型是否将你的子过程带入了一个新的环境。例如，你想写一个只有一个参数的函数，象下面这个函数：

```

sub func ($) {
    my $n = shift;
    print "you gave my $n\n";
}

```

这将得到一个单目操作符(象 **rand** 内建函数)并且改变了编译器确定函数参数的方法。使用了新的函数原型，该函数就只使用一个标量环境下的参数，而不是在列表环境下的多个参数。如果你在以数组或者列表表达式中调用这个函数，即使这个数组或列表只包含一个元素，你可能会得到完全不同的结果：

```

func @foo;      # 计算 @foo 元素个数
func split /:;/ # 计算返回的域的个数
func "a", "b", "c"; # 只传递 "a", 抛弃 "b" 和 "c"
func("a", "b", "c"); # 马上生成一个编译器错误!

```

你已经隐含地在参数列表前面提供了一个 **scalar**，这的确令人有点吃惊。如果 **@foo** 只包含一个元素，那么传递给函数不是这个元素，而是 **1**(**@foo** 的元素个数)。并且在第二个例中，**split** 在标量环境中被调用，吞没你的整个 **@_** 参数列表。在第三个例子中，因为 **func** 已经用函数原型定义为一个

单目操作符，因此只有 "a" 传递给了 `func`；然后 `func` 返回值被丢弃，因为逗号操作符的存在因此继续处理下两个元素并返回 "c"。最后一个例子，在编译的时候用户将得到一个语法错误。

如果你想写一个新的代码得到一个只使用一个标量参数的单目操作符，而不是任何旧的标量表达式，你可以使用下面的函数原型使它使用标量引用：

```
sub func (\$) {
    my $nref = shift;
    print "you gave me $$nref\n";
}
```

现在，编译器可以让下面的例子中，参数以 `$` 开头的通过：

```
func @foo;      # 编译器错误，看见了 @，但要的是 $
func split/;/;  # 编译器错误，看见了函数，但要的是 $
func $s;        # 这个是对的 -- 获取了真的 $ 符号
func $a[3];     # 这个也对
func $h{stuff}[-1]; # 这个也对
func 2+5;       # 标量表达式也会导致编译器错误
func ${\(2+5)}; # 对，不过它是不是比病毒还糟糕？
```

如果你不小心，你可能因为使用函数原型遇到很多麻烦。但如果你非常注意，你可以使用函数原型来作很多漂亮的工作。函数原型是非常强大的，当然需要谨慎使用才能得到好的结果。

5.0 子过程属性

子过程的定义和声明能够附带一些属性。如果属性列表存在，它使用空格或者冒号分割，并等同于通过 `use attributes` 定义的一样。请阅读三十一章的 `use attributes` 获得内部细节。有三个标准的子过程属性：`locked`，`method` 和 `左值`。

5.1 Locked 和 method 属性

```
# 在这个函数里只允许一个线程
sub afunc : locked { ... }

# 在一个特定的对象上之允许一个线程进入这个函数
sub afunc : locked method { ... }
```

只有在子过程或者方法要被多个线程调用的时候，设置 `locked` 属性才有意义。当设置一个不是方法的子过程的时候，Perl 确保在进入子过程之前获得一个锁。当设置一个方法子过程时(具有 `method` 属性的子过程)，Perl 确保在执行之前锁住它的第一个参数(所属的对象)。

`method` 属性能够被它自己使用：

```
sub afunc : method { ... }
```

现在它只是用来标记子过程，使之不产生 "Ambiguous call resolved as CORE: : %s" 警告。(我们以后可以给它更多的含义)。

属性系统是用户可扩展的，Perl 可以让你创建自己的属性名。这些新的属性必须是简单的标记名字(除了 "_" 字符之外没有任何标点符号)。它们后边可以有一个参数列表用来检查它的花括弧是否匹配正确。

下面是一些正确的语法的例子(即使这些属性是未知的)：

```
sub fnord (&\%) : switch(10, foo(7,3)) : expensive;
```

```
sub plugh () : Ugly('\(') :Bad;
sub xyzzzy : _5x5 { ... }
```

下面是一些不正确语法的例子:

```
sub fnord : Switch(10, foo());    # ()-字串不平衡
sub snoid : Ugly '(');           # ()-字串不平衡
sub xyzzzy : 5x5;                # "5x5" 不是合法的标识符
sub plugh : Y2::north;           # "Y2::north"不是简单标识符
sub snurt : foo + bar;           # "+" 不是一个冒号或空格
```

属性列表作为一个常量字符串列表传递进子过程相关的代码. 它的正确工作方法是高度试验性的. 查阅 `attributes(3)` 获得属性列表的详细信息和操作方法.

5.3 左值属性

除非你定义子过程返回一个 左值, 否则你不能从子过程中返回一个可以修改的标量值:

```
my $val;
sub canmod : 左值 {
    $val;
}
sub nomod {
    $val;
}

canmod() = 5;      # 给 $val 赋值为 5
nomod()  = 5;      # 错误
```

如果你正传递参数到一个有 左值 属性的子过程, 你一般会使用圆括弧来防止歧义:

```
canmod $x = 5;     # 先给 $x 赋值 5!
canmod 42 = 5;     # 无法改变常量, 编译时错误
canmod($x) = 5;    # 这个是对的
canmod(42) = 5;    # 这个也对
```

如果你想使用省略的写法, 你可以在子过程只使用一个参数的情况下省略圆括弧. 使用 `($)` 函数原型定义一个函数可以使该函数被解释为一个具有命名的单目操作符优先级的操作符. 因为命名单目操作符优先级高于赋值, 所以你不再需要圆括弧(需不需要圆括弧只是一个代码风格的问题).

当一个子过程允许空参数时(使用 `()` 函数原型), 你可以使用下面的方法而不会引起歧义:

```
canmod = 5;
```

因为没有哪个合法项以 `=` 开头, 因此它能正确工作. 同样, 具有左值属性的方法调用在不传送任何参数时也能省略圆括弧:

```
$obj->canmod = 5;
```

我们保证在未来的 Perl 版本中不改变上面的两种方法. 当你希望在方法调用中封装对象属性时, 它们是非常简便的方法(因此它们可以象方法调用一样被继承但又象变量一样访问).

左值子过程和子过程的赋值表达式右边部分可以通过使用标量替换子过程的方法, 来确定是标量环境还是列表环境. 例如:

```
data(2,3) = get_data(3,4);
```

上边两个子过程都在标量环境中调用, 而在:

```
(data(2,3)) = get_data(3,4);
```

和:

```
(dat(3), data(3) = get_data(3,4);
```

中, 所有的子过程在列表环境中被调用.

在当前的实现中不允许从左值子过程直接返回数组和散列结构. 不过你总是可以返回一个引用来解决这个问题.

Revision: r1.5 - 22 Aug 2005 - 12:36 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [P3GoryDetail](#) > [SubRoutines](#)

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.
向为这里贡献想法, 文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)