

Perl 5 就像现实世界中其余解决问题的语言那样，是一门庞大的语言。高效的 **Perl** 程序不仅要求对语法的深刻理解；你也必须开始体会 **Perl** 的特性间互动的方式，以及那些用 **Perl** 解决已被充分理解问题的常见方法。

请为 **Perl** 的第二条学习曲线做好准备：通过对常见行为模式和内置捷径的高效使用，以 **Perl** 的方式思考，这些惯用语在恰当使用时能让你写出清晰、强大的代码。

惯用语

任何语言——无论编程或是自然语言——都将发展出 惯用语，或者说表达的通用模式。地球每天自转，但我们称之为太阳东升西落。我们也谈论聪明的 *hack*、下流的 *hack* 和不靠谱的代码。

随着你在学习中更加清楚地了解到 **Perl 5** 的本质，你将碰到并理解常见惯用语。它们并非语言特性——你不是 必须 使用它们——而且它们也没有庞大到使你能用函数和方法封装它们。相反，它们是一种习惯。它们是用 **Perl** 的腔调来编写 **Perl** 代码的方式。

将对象用作 `$self`

Perl 5 的对象系统 (*moose*) 将方法的调用者视作一个俗套的参数。类方法的调用者——一个包含类名称的字符串——就是该方法的第一个参数。对象或实例方法的调用者——对象自身——就是该方法的第一个参数。你可以随意按需使用或忽略它。

Perl 5 惯用语将 `$class` 用作类名而将 `$self` 用做实例方法的调用者。这不是一个被语言自身强制的惯例，但这是一个强大到连诸如 `MooseX::Method::Signatures` 这类有用的扩展都默认假设你将 `$self` 用作调用者的惯例。

具名参数

不谈 *signatures* 和 `MooseX::Multimethods` 这类模块，**Perl 5** 的参数传递机制还是比较简单的：所有参数展开为可以通过 `@_` (*function parameters*) 访问的单个列表。然而这种简单偶尔也会显得太过简单——具名参数时常很有用——它没有排除使用惯用语来提供具名参数。

列表上下文求值和 `@_` 赋值允许你用自然且 **Perl** 味十足的方式将具名参数一对对展开。即便这种函数调用方式等同于传入逗号分隔或由 `qw//` 创建的列表，把参数像真正的键值对排列使得函数调用方看起来像是支持具名参数一般：

```
make_ice_cream_sundae(  
    whipped_cream => 1,  
    sprinkles     => 1,  
    banana       => 0,  
    ice_cream     => 'mint chocolate chip',  
);
```

被调用方则可以将这些参数解开到一个哈希里，并将此哈希作为单个参数对待：

```
sub make_ice_cream_sundae  
{  
    B<my %args = @_;>  
  
    my $ice_cream = get_ice_cream( $args{ice_cream} ) );  
    ...  
}
```

Perl 最佳实践 (**Perl Best Practices**) 建议传入哈希引用代替。这样 **Perl** 就可以在调用方检查所构造的哈希是否合法。它比其他方式也少用些内存。

这个技巧能很好地和 `import()` (`importing`) 一起工作；在将余下部分吸入哈希之前你可以处理任意数量的参数：

```
sub import
{
    B<my ($class, %args) = @_;>
    my $calling_package = caller();

    ...
}
```

Schwartzian 转换

作为表达式求值基本成分的列表和列表处理，Perl 新人通常会忽视其重要性。说得更加简单些，Perl 程序员串联求值得变长列表表达式的能力提供了无数高效操作数据的机会。

Schwartzian 转换 (Schwartzian transform) 是一个展示上述原则的重要惯用语，是从 Lisp 语言家族中随手借鉴过来的。

假设你有一个将同事名及其分机号关联起来的 Perl 哈希：

```
my %extensions =
(
    4 => 'Jerryd',
    5 => 'Rudy',
    6 => 'Juwan',
    7 => 'Brandon',
    10 => 'Joel',
    21 => 'Marcus',
    24 => 'Andre',
    23 => 'Martell',
    52 => 'Greg',
    88 => 'Nic',
);
```

又假设你想打印出按名字而非分机号排序的同事-分机号列表。换句话说，你需要对此哈希按值排序。按字符串顺序给哈希的值排序还是很容易的：

```
my @sorted_names = sort values %extensions;
```

.....但是这样就失去了姓名和分机号之间的联系。相反，采用 Schwartzian 转换对数据在排序前后

进行处理以保留所需信息。第一步，将该哈希转化为一个数据结构列表，此数据结构按可排序的方式保留重要信息。此例子中，哈希转化为一个二元素匿名数组：

```
my @pairs = map { [ $_, $extensions{$_} ] } keys %extensions;
```

将此哈希就地反转仅在没有重名的情况下才是可行的。这个特别的数据集并没有这个问题，但还是带防御性地编写代码。

`sort` 接受一个匿名数组列表并照对第二个元素（姓名）的字符串比较结果对其排序：

```
my @sorted_pairs = sort { $a->[1] cmp $b->[1] } @pairs;
```

得到 `@sorted_pairs`，又一个 `map` 操作将此数据结构转化为一个更可用的形式：

```
my @formatted_exts = map { "$_->[1], ext. $_->[0]" } @sorted_pairs;
```

.....现在你可以把它整个打印出来了:

```
say for @formatted_exts;
```

当然, 这些步骤用到了很多(命名不佳的)临时变量。这是一个值得花时间来好好理解的技巧, 但真正的魔法在于组合:

```
say for
  map { " $_->[1], ext. $_->[0]" }
  sort { $a->[1] cmp $b->[1] }
  map { [ $_ => $extensions{$_} ] }
  keys %extensions;
```

从右往左按求值顺序阅读整个表达式。对分机号哈希中的每一个键, 把来自原哈希的键值组合为一个两个元素的匿名数组。对这个由匿名数组构成的列表按第二个元素, 就是原哈希的值, 排序。把排序后的数组格式化为一个个输出字符串。

Schwartzian 转换就是 **map-sort-map** 的管道, 用于将一种数据结构转换为易于排序的另一种形式接着将排序后的结果转换回用于进一步操作的形式。

这种转换是简单的。考虑一种情况, 计算用于排序的值既费时又费内存, 比方说, 对一个个大文件计算密码学散列值。这种情况下 **Schwartzian** 转换也很有用, 因为这样的昂贵操作可以只进行一次(位于最右边的 **map** 内), 在用 **sort** 排序时, 实际上是从缓存中取出值比较, 随后可以在最左边的 **map** 中去掉用于比较的中间散列值。

简易文件吸入

Perl 5

神奇的全局变量在很多情况下是真正全局的。在其他地方搞砸它们的值真是太容易了, 除非你在每一处使用 **local**。这种要求还衍生出若干有趣的惯用语。举例来说, 你可以用单个表达式将文件吸入某个标量中:

```
my $file = do { local $/ = <$fh> };
```

或

```
my $file = do { local $/; <$fh> };
```

\$/ 是输入记录分隔符。用 **local** 将其局部化即将其值设为 **undef**, 并等候赋值。该 **local** 操作在赋值之前发生。由于分隔符的值未定义, **Perl** 高兴地将文件句柄的所有内容用一句话读入并将此值赋给 **\$/**。因为 **do** 代码块求值为该代码块内最后求值语句的值, 也即赋值操作的值, 或者说, 文件的内容。即便在代码块末尾 **\$/** 立刻恢复为先前的状态, **\$file** 现在包含文件的所有内容。

第二个例子是类似的, 除了它不执行赋值操作而仅返回从文件句柄读这一句代码的结果。你可能见过这些例子中的一个, 它们在此情况下一相同的方式工作。

如果你没有从 **CPAN** 安装 **File::Slurp** 时, 此惯用语非常有用(并且, 诚然, 对不熟悉这个 **Perl 5** 功能特定组合的人来说尤其恼火)。

控制程序执行

程序和模块之间的显著差异在于它们预计的用途。用户直接地调用程序, 而程序在执行流程开始之后才加载模块。程序和模块技术上的区别就是对其直接调用是否有意义。

在希望使用 **Perl** 测试工具 (**testing**) 来测试独立程序中的函数或创建用户可以直接的

模块时碰到这个问题。你要做的全部就是了解 Perl 如何 开始执行一段代码。对于这个问题，使用 `caller`。

`caller` 的单个可选参数就是要报告的调用帧的数目。（调用帧（call frame） 就是代表一次函数的调用的簿记信息。）你可以用 `caller(0)` 得到当前调用帧的信息。要允许一个模块正确地作为程序 或 模块运行，你可以编写一个合适的 `main()` 函数并在模块开始处添加这样一行：

```
main() unless caller(0);
```

如果此模块 没有 调用者，用户就可以作为程序直接调用它（使用 `perl path/to/Module.pm` 而非 `use Module;`）。

检查 `caller` 在列表上下文返回的列表中的第八个元素也许在多数情况下更加精确，但很少见。这个值在调用帧表示 `use` 或 `require` 为真，否则为 `undef`。

处理 Main 函数

Perl 在创建闭包（closures）时不要求任何特殊语法；你可能不经意间使闭包闭合于某词法变量之上。在实践中，这通常 很少 会导致问题，除一些特定于 `mod_perl` 的情形以及 `main()` 函数。

许多程序通常会在将执行流程交付其他函数前设置若干文件作用域的词法变量。相比向其他函数传递或从中返回值，直接使用这些变量是一种诱惑，特别是当程序扩展以提供更多功能时。更糟糕的是，这些程序也许会依赖于发生在 Perl 5 编译期间的隐晦作用；一个你 本以为会赋为某特定值的变量也许要到很晚才能得到初始化。

这里有一个简单的解决办法。用一个简单的函数，`main()`，来包装程序的主要代码。将所有你不需要的变量封装为真正的全局变量。接着在你使用完所有需要的模块和编译命令后，向程序的开头加上如下行：

```
#!/usr/bin/perl

use Modern::Perl;
use autodie;

...

B<main( @ARGS );>
```

在执行程序其他部分 之前 调用 `main()` 强制你显式安排编译的初始化顺序。同时也有助于提醒你程序行为封装为函数和模块。（这样可以和既是程序又是库的代码工作得很好 —— `controlled_execution`。）

后缀参数验证

即使不使用诸如 `Params::Validate` 和 `MooseX::Params::Validate` 等 CPAN 模块来验证函数接受的参数是否正确，你仍可以从对偶尔的正确性检查中获益。`unless` 控制流程修饰符是一个在函数开始处断言你期望参数的简单易读的方式。

假设你的函数接受两个参数，不多也不少。你 原可以 这样写：

```
use Carp;

sub groom_monkeys
{
    if (@_ != 2)
    {
```

```

        croak 'Monkey grooming requires two monkeys!';
    }
}

```

.....但从语言学角度来看，后果通常比检查更为重要，因而值得放在表达式的 开头：

```
croak 'Monkey grooming requires two monkeys!' if @_ != 2;
```

.....这种形式，按你阅读后缀条件的偏好，可以简化为：

```
croak 'Monkey grooming requires two monkeys!' unless @_ == 2;
```

如果你专注于消息的文字内容 ("You need to pass two parameters!") 和测试条件 (@_ 应包含两个元素)，则这样读起来更加顺口。这在真值表中几乎占了一行。

Regex En Passant

许多 Perl 5 惯用语依赖于表达式求值得值这一语言设计，就像在：

```
say my $ext_num = my $extension = 42;
```

这样编写代码不太好，但它阐述如下观点：你可以在其他表达式中使用另一个表达式的值。这也不是什么新鲜事；你很可能早就在列表中使用过函数的返回值，或者把它传递给另一个函数。你也许并没有意识到这一隐含点。

假设你有一个全名并且你想提取名字部分。用正则表达式来做很简单：

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

.....其中 `$first_name_rx` 是一个预编译的正则表达式。在列表上下文中，一个成功正则表达式匹配返回由所有捕获组成的列表，并且 Perl 将第一个元素赋值给 `$first_name`。

现在想像你打算修改这个名字，就说去掉所有非单词字符以作为系统帐号的用户名。你可以 这样写：

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

不像前一个例子，此例子从右往左读。第一，将 `$name` 赋值给 `$normalized_name`。接着，对 `$normalized_name` 进行直译这里的括号影响优先级使得赋值先行发生。。赋值 表达式求值为 `$normalized_name` 变量。这个技巧对所有就地改动操作符都是可用的：

```
my $age = 14;
(my $next_age = $age)++;

say "Next year I will be $next_age";
```

一元强制类型转换

Perl 5

的类型系统通常能正确完成任务，至少在你选对操作符的情况下是如此。要拼接字符串，使用字符串拼接操作符，Perl 将两个标量都作字符串对待。把两个数相加，使用加法操作符，Perl 将两个标量都作数值对待。

有时候你必须就你的意图给 Perl 一点提示。若干 一元强制类型转换 (unary coercions) 惯用语可用, 通过它们你可以使用 Perl 5 操作符强制对某值在指定的方法求值。

要确保 Perl 将某个值用作数值, 加零:

```
my $numeric_value = 0 + $value;
```

要确保 Perl 将某个值作为布尔值对待, 双重否定:

```
my $boolean_value = !! $value;
```

要确保 Perl 将某个值作字符串对待, 将其和空字符串拼接:

```
my $string_value = '' . $value;
```

虽然对这些强制类型转换的需求是难以察觉得稀少, 在遇见这类惯用语时, 你应该能够正确理解它们。

全局变量

Perl 5 提供了若干 超级全局变量, 它们是全球变量, 而不仅仅局限于任何特定的包。这些超级全局变量有两个缺点。第一, 它们是全球变量; 任何直接或间接的修改就能影响到程序的其余部分。第二, 它们太过精炼。经验丰富的 Perl 5 程序员早就记住了它们中的一部分。很少有人能够记全这些变量。而且, 它们中也只有几个是常用的。perldoc perlvar 包含这类变量的详尽列表。

管理超级全局变量

要管理这些超级全局变量的全局行为, 最佳途径就是避免使用它们。当你必须用到它们时, 请在尽可能小的作用域内使用 local 来约制改动。所 调用 的代码对这些全局变量做出的修改仍然会影响到你, 但你已经降低了在你能力范围 之外 出现令人惊奇的代码 的可能。

对于这些全局变量的行为目前有一些变通的方法, 但是很多变量从 Perl 1 开始就存在了, 并将作为 Perl 5 的一部分直至其生命周期的结束。就像文件吸入惯用语 (easy_file_slurping) 所展示的那样, 通常会这样写:

```
my $file = do { B<local $/> = <$fh> };
```

用 local 本地化 \$/ 的效果仅持续到代码块的末尾。以从文件句柄中读取所有行一个 tie 文件句柄就是屈指可数的可能情况中的一种。为目的并在 do 代码块内改变 \$/ 的 值的 Perl 代码出现的机会不多。

并非所有对超级全局变量的使用管理起来都那么容易, 但是这个方法通常很有效。

其他时候你需要 读取 超级全局变量的值并期望没有代码会修改它。用 eval 代码块捕获异常的方式容易受竞争条件的感染用 Try::Tiny 代替!, 在这种情况下, 因词法变量超出作用域而调用其上的 DESTROY() 方法可能会重置 \$@:

```
local $@;
```

```
eval { ... };

if (B<my $exception = $@>) { ... }
```

立刻 复制 \$@ 可以预留它的内容。

英语名称

English 核心模块为过度使用标点的超级全局变量提供了详细的名称。用如下方式将其导入名称空间：

```
use English '-no_match_vars';
```

随后你就可以在该名称空间作用域内使用记录在 `perldoc perlvar` 中的详细名称。

三个正则表达式相关的超级全局变量（`$&`、`$`` 和 `$'`）一同降低了程序内 所有正则表达式的性能。如果你因疏忽而没有提供导入参数，则你的程序性能仍会下降即便你没有显式地从这些变量中读取。出于向后兼容考虑，这并非默认行为。

现代化的 Perl 程序应使用 `@-` 代替前三个可怕的变量。

常用超级全局变量

大部分现代化的 Perl 5 程序只用到一部分超级全局变量。有部分全局变量只是为一些你不太可能碰到的特殊情况而存在的。虽说 `perldoc perlvar` 是这些变量正规文档，有部分变量还是值得在此提一下。

超级全局变量的替代

超距作用 (Action at a distance) 的罪魁祸首与IO和异常条件有关。使用 `Try::Tiny` (`exception_caveats`) 有助于把你和正确异常处理所含狡猾的语义隔开。用 `local` 局部化并复制 `$!` 的值可以帮助你避免因 Perl 执行隐式系统调用时的古怪行为。

你可以通过 `use` 引入 `IO::Handle` 来执行词法文件句柄 (`lexical_filehandles`) 上的方法而非 `IO` 相关的超级全局变量。就地 `select` 一个文件句柄，接着修改 `$|`，即是直接调用词法文件句柄上的 `autoflush()` 方法。调用特定文件句柄上的 `input_line_number()` 方法可以得到和 `$.` 等价的结果。有关其他适用方法的信息请参见 `IO::Handle` 文档。