

Perl 5 并不完美。一些看起来像是个好主意的特性却难以正确使用。其他一些特性的行为不合常理。还有一部分就是些坏主意。这种种特性将持续存在——从 Perl 中删除一个特性是一个严肃的过程并且只为滔天罪行而备——然而你可以并应该避免在几乎任何地方使用它们。

裸字

Perl 到处使用印记 (sigil) 和其他标点来同时帮助语法分析器和程序员区分具名实体的类别。即便如此, Perl 仍是一门可塑的语言。你可以按偏好用最具有创意的、可维护的、混乱的、奇形怪状的风格来编写程序。可维护性是每一个优秀程序员都必须考虑的, 但 Perl 开发者本身并不会冒昧地对你认为的可维护性做出任何假设。

Perl 语法分析器理解内置的 Perl 关键字和操作符; 它知道 `bless()` 意味着你在创建对象 (blessed_references)。这些不太会产生歧义.....但是 Perl 程序员可以通过使用裸字 (barewords) 添加语法分析的复杂度。裸字是一个标识符, 它没有印记或其他用于说明其语法功用的附加消歧条件。因为 Perl 5 没有名为 `curse` 的关键字, 则出现在源代码中的字面词 `curse` 就是有歧义的。你是想操作一个变量 `$curse` 还是调用函数 `curse()`? `strict` 编译命令有充分的理由对这类带歧义裸字的使用作出警告。

即便如此, 基于充分的理由, 裸字仍允许在 Perl 5 的若干地方使用。

裸字的正当使用

Perl 5 中的哈希键是裸字。键的用法已经足够让语法分析器将其识别为单引号字符串的等价物, 因此它们不会产生歧义。还要注意意图通过对函数调用或内置操作符 (例如 `shift`) 求值而产生一个哈希键的做法并不如你想象的那样, 除非你消除歧义。

```
# 'shift' 字面值是键
my $value = $items{B<shift>};

# 由 shift 产出的值是键
my $value = $items{B<shift @_>}

# 一元加号足够用于消除歧义
my $value = $items{B<+>shift};
```

从某种意义上说, Perl 5 中的包名是裸字。良好的包命名习惯 (首字母大写) 有助于避免意外情况, 但是语法分析器使用启发式 (heuristic) 方法决定 `Package->method()` 是否意味着先调用一个名为 `Package()` 的函数接着调用结果的 `method()` 方法, 或者是否应该将 `Package` 做包名对待。你可以通过后缀包分隔符 (`::`) 对此消歧, 但这种情况比较罕见, 显然也很丑陋:

```
# 可能是类方法
Package->method();

# 一定是类方法
Package::->method();
```

特殊的具名代码块提供了独有的裸字。AUTOLOAD、BEGIN、CHECK、DESTROY、END、INIT 和 UNITCHECK 声明了函数, 但无需 `sub` 关键字。你可能熟悉了编写不带 `sub` 的 BEGIN 惯用语:

```
package Monkey::Butler;
```

```
BEGIN { initialize_simians( __PACKAGE__ ) }
```

你 可以 不在 `AUTOLOAD()` 声明时使用 `sub`，但不常见。

用 `constant` 编译命令定义的常量可以按裸字使用：

```
# 不要在正式验证时这样做
use constant NAME      => 'Bucky';
use constant PASSWORD => '|38fish!head74|';

...

return unless $name eq NAME && $pass eq PASSWORD;
```

注意这些常量 不会 在内插上下文，如双引号字符串，中内插。

常量是原型函数（`prototypes`）的特例。如果你预声明了函数的原型，你就可以将函数用作裸字；Perl 5 知道它要知道的每件事来正确地对函数的每一处出现进行语法分析。原型的其他缺点仍然适用。

裸字的欠考虑使用

裸字在现代化 Perl 代码中应该很少见；它们的歧义产生脆弱的代码。你可以在几乎所有地方避免使用它们，但是你可能会在遗留的代码中遇到若干对裸字蹩脚的使用。

在词法文件句柄之前（`lexical_filehandles`），所有文件和目录句柄使用裸字。你几乎总是可以安全地将这些代码用词法文件句柄重写；除了 `STDIN`、`STDOUT`、`STDERR` 以外。

没有启用 `strict 'subs'` 而编写的代码可以使用裸字函数名。你可以安全地给这些函数的参数列表加上括号而不会改变代码意图使用 `perl -MO=Deparse,-p` 可以显示 Perl 如何对其进行语法分析，接着——加上括号。。

类似地，旧式代码不会费神正确地对哈希对的 值 加上引号：

```
# 不良风格；不要使用
my %parents =
(
    mother => Annette,
    father => Floyd,
);
```

因为名为 `Floyd()` 和 `Annette()` 的函数都不存在，Perl 将这些哈希值分析为字符串。`strict 'subs'` 编译命令使语法分析器在这种情况下给出错误。

最后，内置的 `sort` 操作符以第二参数的形式接受一个用于排序的函数 名。作为代替，提供一个用于排序的函数 引用 可以避免使用裸字：

```
# 不良用法；不要使用
my @sorted = sort compare_lengths @unsorted;
```

```
# 更好的风格
my $comparison = \&compare_lengths;
my @sorted      = sort $comparison @unsorted;
```

结果代码要多出一行，但是它避免了对裸字的使用。不像其他裸字示例，Perl 的语法分析器不需要对这种语法消歧义。它仅有一种解释 `compare_lengths` 的途径。然而，显式引用带来的清晰度为人类读者带来了好处。

Perl 5 的语法分析器并不理解这一单行版本：

```
# 无法工作
my @sorted = sort \&compare_lengths @unsorted;
```

这起因于对 `sort` 的特殊语法分析；你不能使用任意表达式（诸如取具名函数的引用），但一个代码块或者一个标量可以行得通。

间接对象

Perl 5 中的构造器就是任何返回对象的东西；`new` 不是一个内置的函数。出于惯例，构造器是一个名为 `new()` 的类方法，但是你仍可以按需灵活地选择不同名称。若干陈旧的 Perl 5 对象教程推行 C++ 和 Java 风格的构造器调用方式：

```
my $q = B<new> CGI; # 不要这样写
```

.....而非毫无歧义的：

```
my $q = CGI->new();
```

两种语法在行为上是等价的，除了它们不等价的时候。

第一种形式就是间接对象（indirect object）形式（更精确地说，与格（dative）形式），即动词（方法）处于其意指的名词（对象）之前。在口语中这是可行的，但是这样会向 Perl 5 引入歧义。

裸字间接调用

其中一个问题就是方法名称是裸字（barewords）。语法分析器必须进行若干启发式方法来决定正确的解释。虽然这些启发式方法经过良好测试并且几乎总是正确的，但它们出错的方式令人疑惑。更糟糕的是，在编译和模块加载顺序的面前，它们相当脆弱。

在构造器带参数时，语法分析对人类和计算机来说都是困难的。间接风格可能类似于：

```
# 不要这样写
my $obj = new Class( arg => $value );
```

.....这样便使得类名 `Class` 看上去像函数调用。Perl 5 能够对许多这样的情况消歧，但它的启发式方法取决于语法分析器在该语法分析点上见到了什么包名，它已经解析了的裸字（已经如何解析它们），以及已在当前包内声明的函数名称。

假设执行一个冲突的原型函数，它的名称正好和类名或者一个间接调用的方法冲突。这样的情况并不频繁，但是难以调试，避免这样的语法总是值得的。

间接记法的标量限制

该语法的另一危险之处就是语法分析器期望单个标量表达式是一个对象。向一个存放在集合变量（aggregate variable）文件句柄打印看上去很显然，但是事实并非如此：

```
# 代码和行为不一致
say $config->{output} "This is a diagnostic message!";
```

`print`、`close` 和 `say`

——所有操作文件句柄的关键字——都按间接方式进行操作。

当文件句柄是包全局变量时一切正常，但是词法文件句柄 (`lexical_filehandles`) 使得间接对象语法问题显而易见。在上一个例子中，Perl 会尝试调用 `$config` 对象上的 `say` 方法。解决方法就是产出调用者的表达式消歧：

```
say B<{>{$config->{output}B<}> "This is a diagnostic message!";
```

间接记法的替代

直接调用记法没有这类歧义问题。要构造一个对象，直接在类名上调用构造方法即可：

```
my $q    = CGI->new();
my $obj = Class->new( arg => $value );
```

对于文件句柄操作限制这一情况来说，与格用法太普遍了，你只要将目标调用者用大括号围起，就可以是有间接调用手法。除此之外，考虑加载 `IO::Handle` 核心模块，它允许你调用文件句柄对象（如词法文件句柄）上的方法来执行 IO 操作。

对于超级偏执狂来说，你可以通过在类名后追加 `::` 来进一步对类方法调用消歧，例如 `CGI::->new()`。然而，实践中很少有代码会这样做。

CPAN 模块 `Perl::Critic::Policy::Dynamic::NoIndirect`（一个 `Perl::Critic` 的插件）可在代码审核期间识别间接调用语法。CPAN 模块 `indirect` 可以在程序运行时识别它们并禁止使用：

```
# 对间接用法做出警告
no indirect;

# 对此用法抛出异常
no indirect ':fatal';
```

原型

原型 (prototype) 是一小块附加在函数生命上的可选元数据。新手通常假设这些原型可以作为函数签名使用；但它们不是。相反它们服务于两个不同的目的：它们给予语法分析器提示来改变对函数及其参数的语法分析，并且它们还修改了 Perl 5 处理函数参数的方式。

声明函数原型，只要在名称后加上它：

```
sub foo      (&@);
sub bar      ($$) { ... }
my $baz = sub (&&) { ... };
```

你可以向函数前置声明 (forward declaration) 添加原型。你也可以在前置声明中忽略它们。如果你使用带原型的前置声明，该原型必须完整地在函数声明中出现；如果不这样做，Perl 将给出原型不匹配的警告。其逆命题不为真：你可以在前置声明中忽略原型而在完整声明中包含它。

在前置声明中忽略原型没有什么理由，除非你有想法编写太过聪明的代码。

原型最初的目的是允许用户定义它们自己的函数，这些函数的行为和（一些）内置操作符一样。考虑 `push` 操作符的行为，它接受一个数组和列表。虽然 Perl 5 正常情况下在调用方将数组和列表展开为单个列表，Perl 5 语法分析器知道调用 `push` 必须

有效地将数组作为一个单元传入，以使 `push` 能够就地操作数组。

内置操作符 `prototype` 接受一个函数名称并返回代表其原型的字符串。要参考内置关键字的原型，使用 `CORE::` 形式：

```
$ B<perl -E "say prototype 'CORE::push';">
\@@
$ B<perl -E "say prototype 'CORE::keys';">
\%
$ B<perl -E "say prototype 'CORE::open';">
*;$@
```

一些内置操作符拥有你无法模拟的原型。在这些情况下，`prototype` 将返回 `undef`：

```
$ B<perl -E "say prototype 'CORE::system' // 'undef' ">
undef
# 你无法模拟内置函数 C<system> 的调用惯例

$ B<perl -E "say prototype 'CORE::prototype' // 'undef' ">
undef
# 内置函数 C<prototype> 没有原型
```

再看看 `push`：

```
$ B<perl -E "say prototype 'CORE::push';">
\@@
```

@ 字符代表一个列表。反斜杠强制对对应的参数进行 引用。因此这个函数接受一个数组引用（因为你无法得到列表的引用）和一系列值。`mypush` 可能为：

```
sub mypush (\@@)
{
    my ($array, @rest) = @_;
    push @$array, @rest;
}
```

合法的原型字符包括强制标量参数的 `$`，对应哈希的 `%`（通常用作引用），以及标记 代码块的 `&`。完整文档参考 `perldoc perlsub`。

原型的问题

原型可以改变后续代码的语法分析过程而且它们会对参数进行强制类型转换。它们不是函数参数类型和个数的文档，它们也不对应具名参数。

原型对参数的强制类型转换以一种隐晦的方式发生，诸如在传入参数上强制标量上下文：

```
sub numeric_equality($$)
{
    my ($left, $right) = @_;
    return $left == $right;
}

my @nums = 1 .. 10;

say "They're equal, whatever that means!" if numeric_equality @nums,
10;
```

.....但请 不要 进行比一个简单表达式还复杂的操作:

```
sub mypush(\@@);

# 编译错误: 原型不匹配
# (期待数组; 得到标量赋值)
mypush( my $elems = [], 1 .. 20 );
```

这些还不算是来自原型的 最隐晦 的误导。

原型的正当使用

只要代码维护者不将其作为完整的函数签名, 原型还是有一些合法的用途的。

第一, 它们在用用户定义函数模拟并覆盖内置关键字时是必须的。你必须先通过确认 `prototype` 没有返回 `undef` 来检查你是否 可以 覆盖内置关键字。一旦知道了关键字的原型, 你就可以声明和核心关键字同名的前置定义:

```
use subs 'push';

sub push (\@@) { ... }
```

注意, 不管词法作用域与否, `subs` 编译命令对 文件 余下的部分都有效。

使用原型的第二个理由就是定义编译期常数。一个由空原型 (作为 无 原型的反面) 声明且求值得单个表达式的函数将成为常数而非函数调用:

```
sub PI () { 4 * atan2(1, 1) }
```

在处理该原型声明以后, Perl 5 优化器知道在余下的源代码中, 它应该在碰到裸字或对 `PI` 带括号的调用时用计算得到的 π 值进行替换 (同时遵循作用域和可见性)。

相对于直接定义常量, `constant` 核心编译命令会为你处理这些细节并且更加易读。如果你想将这些常量内插入字符串中, 来自 CPAN 的 `Readonly` 模块或许更加有用。

最后一个使用原型的理由是, 它将 Perl 的语法扩展以便将匿名函数操作为代码块。CPAN 模块 `Test::Exception` 很好地利用这一点提供了带延迟计算 (delayed computation) 的良好 API。其中 `throws_ok()` 函数接受三个参数: 要运行的代码块、一个匹配异常字符串的正则表达式以及一个对此测试可选的描述。假设你想测试 Perl 5 在意图在未定义值上调用方法时产生的异常消息:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok
{ my $not_an_object; $not_an_object->some_method() }
qr/Can't call method "some_method" on an undefined value/,
'Calling a method on an undefined invocant should throw
exception';
```

此导出函数 `throws_ok()` 拥有原型 `&$;$`。它的第一个参数是代码块, Perl 会将其升级为全能匿名函数。第二个要求的参数是一个标量。第三个参数则是可选的。

最最细心的读者可能发现一个语法上奇怪的缺失：在作为第一参数传递给 `throws_ok()` 的匿名函数的结尾没有逗号。这是 Perl 5 语法分析器的古怪之处。加上逗号会引发语法错误。语法分析器期待的是空白，而非逗号操作符。

“此处没有逗号”这一规则是原型语法的缺点。

不带原型你也可以使用这一 API。这可不那么吸引人：

```
use Test::More tests => 1;
use Test::Exception;

throws_okB(<>
    B<sub> { my $not_an_object; $not_an_object->some_method() }B<,>
    qr/Can't call method "some_method" on an undefined value/,
    'Calling a method on an undefined invocant should throw
exception'B<)>;
```

对函数原型的稍稍使用来替代对 `sub` 的需要还是合理的。原型其他用法很少能足够令人信服地胜过它们的缺点。

Ben Tilly 提出了第四条理由：在使用 `sort` 时定义自定义函数。定义原型为 `($$)` 的函数将会使参数传入 `@_` 而非包全局变量 `$a` 和 `$b`。这是一个罕见的情况，但可以节省你的调试时间。

方法-函数等价

Perl 5 的对象系统故意很精简 (`blessed_references`)。因为一个类就是一个包，Perl 自身不对存储于包中的函数和存储于包中的方法加以强制区分。相同的关键字，`sub`，用于同时表达两者。将第一个参数作 `$self` 对待的文档和惯例向代码的读者暗示其意图，但是如果你试着将函数作为方法调用，Perl 自身将把任何能找到的拥有合适名称、处于合适包中的函数当作方法。

相似地，你可以将方法当作一个函数——完全限定、导出、或作为引用调用——如果手动传入你自己的调用物 (`invocant`) 的话。

两种方法都有各自的问题；两种都不要用。

调用方 (Caller-side)

假设你有一个包含若干方法的类：

```
package Order;

use List::Util 'sum';

...

sub calculate_price
{
    my $self = shift;
    return sum( 0, $self->get_items() );
}
```

如果你有一个 `Order` 对象 `$o`，下列对此方法调用或许看上去是等价的：

```
my $price = $o->calculate_price();

# 不正确；不要使用
```

```
my $price = Order::calculate_price( $o );
```

虽然在这一简单的例子中，它们产生相同的输出，然而后者以隐晦的方式违反了对对象的封装。它连同对象查找一起避免了。

如果 `$o` 不是一个 `Order` 对象，而是 `Order` 的子类或同质异晶体 (allomorph) (roles) 且覆盖了 `calculate_price()`，则“方法用作函数”调用将执行 错误的方法。相同地，如果 `calculate_price()` 的内部实现会改变——也许从其他地方继承或通过 `AUTOLOAD()` 委托而来——则调用者可能会失败。

Perl 有一种让此行为看上去显得必须的情况。如果你强制方法解析 (method resolution) 而不进行分派 (dispatch)，你如何调用作为结果的方法引用？

```
my $meth_ref = $o->can( 'apply_discount' );
```

这里有两种可能。一是丢弃 `can()` 方法返回的结果：

```
$o->apply_discount() if $o->can( 'apply_discount' );
```

第二则是按方法调用语法使用此引用：

```
if (my $meth_ref = $o->can( 'apply_discount' ))
{
    $o->$meth_ref();
}
```

当 `$meth_ref` 包含一个函数引用，Perl 将以 `$o` 作为调用者 (invocant) 调用该引用。这在严格 (“strict”) 要求之下同样可行，就像在包含其名称的标量上调用方法一样：

```
my $name = 'apply_discount';
$o->$name();
```

通过引用调用方法有一小小的缺点；如果程序的结构在存储引用和调用引用之间有所变动，则此引用也许不会指向当前最合适的方法。如果 `Order` 类改变导致 `Order::apply_discount` 不再是最合适的调用方法，`$meth_ref` 中的引用也不会随之更新。

如果你使用这种调用形式，请限定引用的作用域。

被调用方 (Callee-side)

因为 Perl 5 并不就声明区分函数和方法，还因为作为函数或方法调用一个给定的函数 是 可能的 (然而这是不可取的)，因而编写一个能用两种方式调用的函数也是可能 的。

CGI 核心模块是一个主要的冒犯者。它的函数手工检查 `@_` 以决定第一参数是否是一个可能的调用物 (invocant)。如果是，它们执行特别的修改来保证函数需要访问的任何对象状态都是可用的。如果第一参数不像是调用物，则函数必须参考其他地方的 全局数据。

像所有的启发式方法 (heuristics) 那样，存在一些边边角角的情况。对于给定的方法，很难确切预言什么调用物是潜在合法的，特别是考虑用户可以创建子类时。文档的负担同时也加重了——对组合了过程式和面向对象接口的解释必须反映代码的二分法 (dichotomy)——就好像原本就是被误用的。当一部分代码使用过程式接口而另一部分用对象接口时会怎么样？

为库提供隔离的过程式和对象式接口也许是正当的。一些设计使得一部分技巧比其他更加有用。将这两者合并为单一的 API 会加重维护的担子。请避免这样做。

捆绑 (Tie)

重载 (overloading) 让你就特殊类型的强制转换和访问给予类自定义行为。有一种让变量表现如同内置类型 (标量、数组和哈希) 的机制, 它还带有更多特别的行为。这种机制使用 `tie` 关键字; 它就是 捆绑 (tying)。

最初对 `tie` 的使用是为生成存储在磁盘上而非内存的哈希。这允许从 Perl 中使用 `dbm` 文件, 同时也能够访问大于内存尺寸的文件。核心模块 `Tie::File` 提供了一个相似的系统, 通过它便可以处理过大的数据文件。

用于将 `tie` 变量的类必须遵从特定数据类型的特定的、有良好文档的接口。虽然核心模块 `Tie::StdScalar`、`Tie::StdArray` 和 `Tie::StdHash` 在实践中更加有用, `perl doc perltie` 仍是这些接口的主要资源。从继承这些模块为开始, 并仅重载你需要修改的特定方法。

这些父类的文档和实现位于 `Tie::Scalar`、`Tie::Array` 和 `Tie::Hash` 模块中。因此如果你想从 `Tie::StdScalar` 继承, 还必须同时 `use Tie::Scalar`。如果 `tie()` 没有使你迷惑, 这些代码的组织可能就会。

捆绑变量

给出一个要捆绑的变量, 可以用如下语法捆绑它:

```
use Tie::File;
tie my @file, 'Tie::File', @args;
```

.....其中第一个参数是要捆绑的变量, 第二个是用于捆绑变量的类名, `@args` 是由捆绑函数要求的可选参数列表。以 `Tie::File` 为例, 就是要捆绑到数组上的文件名。

捆绑函数重组了构造器: `TIESCALAR`、`TIEARRAY()`、`TIEHASH()` 和 `TIEHANDLE()` 分别对应标量、数组、哈希和文件句柄。这些函数返回 `tie()` 关键字返回的相同对象。大多数人会忽略它。

`tied()` 操作符在用于捆绑变量时返回相同对象, 否则返回 `undef`。再一次, 很少有人会使用返回的对象。相反, 他们在使用 `tied()` 来决定一个变量是否捆绑时只检查对象的 布尔值。

实现捆绑变量

要实现一个捆绑变量的类, 从诸如 `Tie::StdScalar` 等核心模块继承, 接着为要更改的操作覆盖特定方法。就捆绑标量来说, 你很可能需要覆盖 `FETCH` 和 `STORE`, 可能需要覆盖 `TIESCALAR()`, 很可能忽略 `DESTROY()`。

你可以用很少的代码创建一个记录对某标量读写的类:

```
package Tie::Scalar::Logged;

use Modern::Perl;

use Tie::Scalar;
use parent -norequire => 'Tie::StdScalar';

sub STORE
{
    my ($self, $value) = @_;
    Logger->log("Storing <$value> (was [$$self])", 1);
    $$self = $value;
}

sub FETCH
```

```

{
    my $self = shift;
    Logger->log("Retrieving <$$self>", 1);
    return $$self;
}

1;

```

假设 `Logger` 的类方法 `log()` 接受一个字符串和一个通过调用栈帧号报告位置的 数字。注意 `Tie::StdScalar` 并没有独立的 `.pm` 文件，因此你必须使用 `Tie::Scalar` 让其可用。

在 `STORE()` 和 `FETCH()` 方法内部，`$self` 如 `bless` 后的标量一般工作。向此标量引用赋值改变标量的值而从中读取则返回它的值。

类似的，`Tie::StdArray` 和 `Tie::StdHash` 的方法对应地作用于 `bless` 后的数组和哈希引用。`perldoc perltie` 文档解释了其支持的大量方法，除其他操作外，你可以从中读取写入多个值。

`-norequire` 选项阻止 `parent` 编译命令为 `Tie::StdScalar` 加载文件的意图，因此此模块是 `Tie/Scalar.pm` 文件的一部分。

何时使用捆绑变量

捆绑变量也许像是为编写机灵的代码提供了一个有趣的机会，但是它们在几乎所有情况下会导致令人迷惑的接口，很大一部分归因于它们非常罕见。除非你有充足的理由创建行为和内置数据类型一致的对象，否则避免创建你自己的捆绑类。

充足的理由包括方便调试（使用记录标量来帮助你理解它的值在何处改变）和使得一些不可能的操作变得可能（用节省内存的方式访问大文件）。捆绑变量作为对象的主要接口时不太有用；用它配合你的整个接口来实现 `tie()` 接口通常太过困难和勉强。

最后的警句既悲哀又具说服力；相当多代码并不指望和捆绑变量一起工作。违反封装的代码也许会妨碍对小聪明的正当且合法的使用。这是一种不幸，但是违反库代码的期待常常引发缺陷，并且一般你没有能力修复这样的代码。