1 NAME

perlobj – Perl 对象

2 说明

首先你必须懂得在 Perl 中，什么叫做“引用”，如果你还不懂，那么请参考 perlref。其次，如果你仍然感到下文出现的引用过于复杂的话，那么请先阅读 perltoot 和 perltooc 这两个 Perl 的面向对象编程初级教程。

首先让我们来看看有关 Perl 面向对象编程的三个基本定义：

1. 一个“对象”是指一个“有办法知道它是属于哪个类”的简单引用。
2. 一个“类”是指一个“有办法给属于它的对象提供一些方法”的简单的包。
3. 一个“方法”是指一个“接受一个对象或者类名称作为第一个参数”的简单的子程序。

我们暂时不考虑从更深一层的角度来讲，以上说法是否正确。

对象仅仅只是引用

和 C++ 不同，Perl 没有为“构造函数”提供任何特殊的语法（译者注：在 C++ 中，和类名称相同的类方法被称为“构造函数”，创建对象时被自动调用）。Perl 中，构造器（译者注：因为 Perl 不强调“函数”这个概念，因此在下文中一律译 为“构造器”）只是一个会返回一个“经过 bless 处理”的引用的子程序，这个 经过 bless 处理的引用就是人们所说的“对象”，而 bless 的作用就是用来说明 这个对象是隶属于哪个“类”。

下面就是一个典型的构造器的例子：

```
package Critter;
sub new { bless {} }
```

new 这个词并没有任何特殊的含义，如果你喜欢，你也可以写成这样：

```
package Critter;
sub spawn { bless {} }
```

这样做不会使 C++ 程序员误以为 new 有什么特殊的含义，因此或许更加合理一 些。我们建议你给你的构造器起名时尽量选择能够准确反映它在你的解决方案中的 意义的名字。而不要拘泥于 new 或者其它那些千篇一律的名字。例如在 Perl/Tk 中，组件的构造器就叫做“create”。

和 C++ 相比，Perl 的构造器有一点不同，那就是它不会自动调用基类的构造器。 因为 hash 可以轻易地表示“名字=>值”这样的属性对，因此通常我们用一个匿名 hash 引用来储存对象的各个属性。在上例中，用一对大括号 {} 可以生成一个 空的匿名 hash 引用，然后 bless() 函数给它打上一个印记，让它变成一个 Critter 类的对象，最后返回这个对象。这只是一个简便写法：因为对象自身知道 它是被 bless 过的，并且 bless {} 正好是 sub new 的最后一个语句(也是唯一 的语句)，所以可以直接做为返回值，不需要显式地 return。

实际上，sub new { bless {} } 写全了相当于下面的代码段：

```
sub new {
        my $self = {};
        bless $self;
        return $self;
}
```

有时候你经常会见到更复杂一些的构造器，比如它可能会调用另外一个方法去做一 些构造工作：

```
sub new {
        my $self = {};
        bless $self;
        $self->initialize();   # 注意这里
        return $self;
}
```

如果你小心地处理继承的话（这种情况经常碰到，参见 "模块的创建、使用和重用 in perlmodlib），那么你可以用两个参数来调用 bless，因此你的构造器就可以实现继承：

```
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

如果你希望用户不仅能够用 CLASS->new() 这种形式来调用你的构造函 数，还能够以 $obj->new() 这样的形式来调用的话，那么就这么做：

```
sub new {
    my $this = shift;
    my $class = ref($this) || $this;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

需要注意的是，这样作并不会发生任何拷贝动作。如果你希望拷贝一个对象，那 么你需要自己写代码处理。接下来 bless 的第二个参数 $class 所属的 initialize() 方法将被调用。

在类的内部，所有的方法都把对象当作一个普通的引用来使用。而在类的外部，用 户只能看到一个经过封装的对象，所有的值都是不透明的，只能通过类的方法来访 问。

虽然理论上我们可以在构造器中重新 bless 一个对象到别的类。对一个对象再次 进行 bless，将导致这个对象术语新类，而忘记原先的老类。我们应该保持一个对 象始终只属于一个，所以我们不建议这么做。但是如果有谁真的这么做，那纯粹是 自找麻烦。

澄清一下：对象是经过 bless 的，引用变量则没有。对象知道它被 bless 到了哪 个类，而引用变量不知道。bless 处理的实际上是引用指向的对象，而不是引用变 量自身。考虑下面的例子：

```
$a = {};
$b = $a;
bless $a, BLAH;
print "\$b is a ", ref($b), "\n";
```

结果显示 $b 也被 bless 到 BLAH 类了，由此可见，bless() 操作的是对象而不 是引用变量。

一个类只是一个简单的包

和 C++ 不同，Perl 并不为类定义提供任何特殊语法。实际上类只是一个包而已。你可以把一个包当作一个类用，并且把包里的函数当作类的方法来用。

不过，有一个特殊的数组，叫做 @ISA，它说明了“当 Perl 在当前包中找不到想要的方法时，应当继续从哪儿去找”。这就是 Perl 实现“继承”的关键。@ISA 中的每个元素都是一个别的包的名字。当类找不到方法时，它会从 @ISA 数组中依次寻找（深度优先）。类通过访问 @ISA 来知道哪些类是它的基类。

所有的类都有一个隐含的基类（祖先类）：UNIVERSAL。UNIVERSAL 类为它的子类提供几个通用的类方法。参见 默认的 UNIVERSAL 方法 得到更多说明。

如果在基类中找到了缺失的方法，那么为了提高效率，它会被缓存到当前类。每当修改了 @ISA 或者定义了新的子程序时，缓存会失效，这将导致 Perl 重新做一次查找。

如果在当前类、当前类所有的基类、还有 UNIVERSAL 类中都找不到请求的方法，这时会再次查找名为 AUTOLOAD() 的一个方法。如果找到了 AUTOLOAD，那么就会调用，同时设定全局变量 $AUTOLOAD 的值为缺失的方法的全限定名称。

如果还不行，那么 Perl 就宣告失败并出错。

如果你不想继承基类的 AUTOLOAD，很简单，只需要一句

```
sub AUTOLOAD;
```

就行了。然后调用 AUTOLOAD 时就会失败。

Perl 类只有方法继承。数据继承由程序员自己实现。基本上，对 Perl 来讲这不是一个什么大问题：因为我们大多数时候都用匿名 hash 来储存对象数据，而每一层的基类都可以往 hash 表中加入自己的属性，因此子类自然就可以继承基类的属性。唯一的问题发生在基类和子类使用了同一个名字作为 hash 键值时。不防假设基类已经使用了 'city' 这个键名，这时子类中也想用 'city' 这个键，那么很明显将会覆盖，由于子类在设计时无法知道父类中是否已经使用了 'city' 所以似乎这的确是一个问题。有一个变通方法就是，每一层类都优先考虑使用自己的包名称作为 hash 键的前缀：

```
sub bump {
   my $self = shift;
   $self->{ __PACKAGE__ . ".count"}++;
}
```

这样你就可以在父类和子类中访问同一个属性的不同版本。

一个方法就是一个简单的子程序

和 C++ 不同，Perl 不提供任何特殊的语法来定义方法。（不过 Perl 提供了一个特殊的语法用来调用方法，稍后再讲）。方法把它被调用时的对象或者类名称当作它的第一个参数。有两种不同的调用方法的途径，分别成为“调用类方法”和“调用实例方法”。

类方法把类名当作第一个参数。它提供针对类的功能，而不是针对某个具体的对象的功能。构造器通常是一个类方法，参见 perltoot 或者 perltooc。大多数类方法简单地忽略第一个参数，因为方法知道自己处在什么类里面，也不关心它是通过什么类来调用的。（调用类和所处类不一定相同，例如基类的方法被子类调用时，方法的所处类是基类，而调用类是子类，类方法和对象方法都是如此。）举个常见的例子，下面的类方法可以通过名字来查询对象：

```
sub find {
   my ($class, $name) = @_;
   $objtable{$name};
}
```

实例方法把对象作为它的第一个参数。因此典型的做法是把第一个参数 shift 到一个名为“self”或者“this”的变量中。然后再把它当作一个引用来用：

```
sub display {
    my $self = shift;
    my @keys = @_ ? @_ : sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
}
```

调用方法

出于历史遗留的原因，Perl 提供了两种不同的形式去调用一个方法。最简单的形式是采用箭头符号：

```
my $fred = Critter->find("Fred");
$fred->display("Height", "Weight");
```

你可以早就熟悉了引用的 -> 操作符。事实上，因为上面的 $fred 是一个指向了对象的引用，因此你也可以把箭头操作符理解为另外一种形式的 解引用。

出现在箭头左边的引用或者类名，将作为第一个参数传递给箭头右边的方法。 所以上面的代码就分别相当于这样：

```
my $fred = Critter::find("Critter", "Fred");
Critter::display($fred, "Height", "Weight");
```

Perl 怎么知道箭头右边的子程序是哪个包里的呢？答案是通过查看箭头左边的 内容。箭头左边必须是一个对象，或者是一个标识类名的字符串。这两种情况 都行。如果类里没有这个方法，那么 Perl 就从基类中进行检索。

如果必要，你还可以强制 Perl 检索其它类：

```
my $barney = MyCritter->Critter::find("Barney");
$barney->Critter::display("Height", "Weight");
```

这个例子中，MyCritter 类是 Critter 类的子类，并且定义了自己的 find() 和 display() 方法。通过加前缀 Critter:: 可以强制 Perl 执行 Critter 的方法而不是 MyCritter 自己的方法。

上面的例子还有一种特殊情形，那就是你可以用 SUPER 伪类来告诉 Perl 通过当前包的 @ISA 数组来检索究竟应该使用哪个类。

```
package MyCritter;
use base 'Critter';   # sets @MyCritter::ISA = ('Critter');

sub display {
    my ($self, @args) = @_;
    $self->SUPER::display("Name", @args);
}
```

注意：SUPER 表示 当前包 的 超类 而不是 对象 的 超类。 而且，SUPER 符号仅仅只是一个方法名称的 修饰符，因此不能把它当作 类名称使用在其它地方。记住：SUPER 不是类名称，只是修饰符。例如：

```
something->SUPER::method(...);      # OK
SUPER::method(...);                 # WRONG
SUPER->method(...);                 # WRONG
```

最后一点，箭头左边的类名或者对象，也可以用返回类名或者对象的表达式来代 替。所以下面这句是合法的：

```
Critter->find("Fred")->display("Height", "Weight");
```

这句也是合法的：

```
my $fred = (reverse "rettirC")->find("Fred");
```

间接对象语法

另外一种调用方法的方式称为“间接对象”语法。这条语法早在 Perl4 时代就 已经引进，那时还没有对象这个概念。它也可以用在文件句柄上：

```
print STDERR "help!!!\n";
```

同样的语法可以调用对象或者类的方法：

```
my $fred = find Critter "Fred";
display $fred "Height", "Weight";
```

注意在对象/类名称与参数之间不能有逗号，这种语法告诉 Perl 你想要调用一 个对象方法而不是普通的子程序。

但是如果没有参数怎么办？（译者注：这时方法后面只有一个对象/类名称，因 此不能一眼看出到底是想调用一个方法，还是用对象/类名称做参数调用一个普 通的子程序）。这时，Perl 只能猜测你的想法，更糟糕的是，Perl 是在“编 译时”就进行猜测！通常 Perl 可以作出正确的判断，但是有时 Perl 会把一 个函数调用编译成一个类方法，或者把一个类方法编译成一个函数调用。这可 能会导致出现难以察觉的错误。

例如，有一个 new 方法的间接调用－－C++ 程序员通常喜欢这么做－－可 能被编译成一个子程序调用，前提是如果碰巧在当前作用域有一个子程序也叫 new。你的代码最终会调用当前包的 new 子程序，而不是你想要的类方法。

TODO The compiler tries to cheat by remembering bareword "require"s, but the grief when it messes up just isn't worth the years of debugging it will take you to track down such subtle bugs.

这个语法还有一个问题：间接对象仅限于一个名称、或者一个标量、或者一个 块，之所以这么做是因为优先级的问题。如果不加这个限制的话，将导致 Perl 在分析你的程序时需要多做很多向前扫描工作，比如解引用之类的。 这个诡异的规则同样适用于 print 和 printf。

请看下面的两行

```
move $obj->{FIELD};             # 很可能是错的！
move $ary[$i];                  # 很可能是错的！
```

上面两行从 Perl 理解的角度来看，相当于：

```
$obj->move->{FIELD};            # 看这儿！
$ary->move([$i]);               # 你真的想这样吗？
```

而你真正希望的也许是：

```
$obj->{FIELD}->move();          # 这么做多好
$ary[$i]->move;                 # 这可能才是你的意思。
```

要想用间接对象语法正确的表达你的意图，你得加上花括号：

```
move {$obj->{FIELD}};
move {$ary[$i]};
```

即使是这样，还是会存在隐患（考虑如果当前包就有一个名为 move 的函数，那么很显然 Perl 会把它理解成函数调用而不是间接对象调用）。因此，我们大力推荐你只使用 ->！。不管怎样，你仍然会看到很多以前遗留下来的间接对象语法，因此熟悉它们还是很有必要的。

默认的 UNIVERSAL 方法

UNIVERSAL 包为它的子类提供如下几个方法：

isa(CLASS)

如果调用 isa 的对象是隶属于 CLASS 或者它的子类，那么 isa 返回真值。

你也可以用传递两个参数的办法直接调用 UNIVERSAL::isa：第一个参数是一个对象（甚至是普通的引用），这个办法可以用来检查一个对象是不是属于指定的类型。例如：

```
if(UNIVERSAL::isa($ref, 'ARRAY')) {
  #...
}
```

要想确定一个引用是不是一个 bless 过的对象，你可以这么写：

```
print "It's an object\n" if UNIVERSAL::isa($val, 'UNIVERSAL');
```

can(METHOD)

can 检查一个对象是不是拥有一个叫做 METHOD 的方法。如果有，那么将返回那个方法（实际上就是子程序）的引用。如果没有，那么返回 undef

也可以用两个参数来直接调用 UNIVERSAL::can。当第一个参数不是一个对象或者是类名的时候，它返回 undef，所以我们也可以用这个办法知道一个引用是不是一个对象。

```
print "It's still an object\n" if UNIVERSAL::can($val, 'can');
```

你也可以用 Scalar::Util 模块的 blessed 函数来达到同样的目的：

```
use Scalar::Util 'blessed';

my $blessing = blessed $suspected_object;
```

如果 $suspected_object 是一个对象，那么 blessed 返回对象所属的类名称，不然返回 undef。

VERSION([NEED])

VERSION 返回一个类的版本号。如果提供了 NEED 参数，那么它还会检查 当前版本号（就是类里面的那个 $VERSION 变量）是不是小于 NEED，如果小 于，它会导致 Perl 程序 die。此方法通常作为一个类方法来调用。

use 语句中会自动调用此方法，请看下面：

```
use A 1.2 qw(some imported subs);
```

上面的语句相当于隐含地调用了：

```
A->VERSION(1.2);
```

注意： can 直接在 Perl 内部实现，isa 也是，并且还采用了缓冲 技术。因此当你的程序动态地修改 @ISA 数组时，可能会出现稀奇古怪的问题。

你也可以通过 Perl 程序或者 XS 程序自己给 UNIVERSAL 类添加方法，并且 不需要 use UNIVERSAL 就可以在你的程序中使用新加的方法。

析构器

当对象的最后一个引用释放时，对象会自动析构。（如果你把对象储存在全局 变量中，那么一直到你的程序退出时才会析构）。如果你想在析构的时候做些 什么，那么你可以在类中定义一个名为“DESTROY”的方法。它将在适合的时机 自动调用，并且按照你的意思执行额外的清理动作。Perl 会把对象的引用作为 唯一的参数传递给 DESTROY。注意这个引用是只读的，也就是说你不能通过访问 $_[0] 来修改它。（译者注：参见 perlsub）但是对象自身（比如 ${$_[0] 或者 @{$_[0]} 还有 %{$_[0]} 等等）还是可写的，

如果你在析构器返回之前重新 bless 了对象引用，那么 Perl 会在析构器返回 之后接着调用你重新 bless 的那个对象的 DESTROY 方法。这可以让你有机会 调用基类或者你指定的其它类的析构器。需要说明的是，DESTROY 也可以手工 调用，但是通常没有必要这么做。

在当前对象释放后，包含在当前对象中的其它对象会自动释放（假如别的地方没 有什么引用指向它们的话）。

摘要

以上就是所有的内容了 ^_^。 你现在需要做的就是马上出去买本书，关于“面向对象设计模式”的那种（相信 很好买到，因为这种书现在满大街都是）然后一头扎进去，啃上至少 6 个月再 出来 ^_^

Two-phased 垃圾回收

为了更多的目的，Perl 采用了一套快速、简便，基于引用计数的垃圾回收机制。 这意味着有一些额外的解引用操作发生在某个层次，因此如果你不用 C 编译器 的 -O 开关编译你的 Perl 的话，性能会有些损失，如果你的 Perl 已经 是用 cc -O 编译过的了，那就没什么问题。

还有个很严重的问题：有时候引用计数根本就不可能为 0，也就是说内存永远 不会释放。例如下面的代码就有这个问题：

```
{
  my $a;
  $a = \$a;
}
```

虽然 $a 已经完全超出作用域了，但是它还是不能释放。当创建递归数据结构时，你必须明确打破这种自引用，否则内存就会泄漏。例如，下面就是一个引用了自身的节点（类似的代码可能会出现在树型结构中）：

```
sub new_node {
    my $class = shift;
    my $node  = {};
    $node->{LEFT} = $node->{RIGHT} = $node;
    $node->{DATA} = [ @_ ];
    return bless $node => $class;
}
```

如果你创建这样的节点，那么内存就会无法自动释放，除非你自己打断自引用的结构。换句话说，这不是一个特性，所以你别指望它。

差不多就这些。

当一个解释器线程最后快要退出的时候（通常发生在你的程序结束时），就会进行垃圾回收，然后所有这个线程拥有的对象都会释放。这一点对于嵌入式的 Perl 应用或者多线程程序非常重要。下面这个程序演示了 Perl 的 two-phased 垃圾回收：

```
#!/usr/bin/perl
package Subtle;

sub new {
    my $test;
    $test = \$test;
    warn "创建 " . \$test;
    return bless \$test;
}

sub DESTROY {
    my $self = shift;
    warn "销毁 $self";
}

package main;

warn "开始运行";
{
    my $a = Subtle->new;
    my $b = Subtle->new;
    $$a = 0;  # break selfref
    warn "块结束之前";
}

warn "块结束之后";
warn "程序结束...";
exit;
```

好比我们把它保存成 /foo/test，那么输出结果就应该是下面这样的：

```
开始运行 at /foo/test line 18.
创建 SCALAR(0x8e5b8) at /foo/test line 7.
创建 SCALAR(0x8e57c) at /foo/test line 7.
块结束之前 at /foo/test line 23.
销毁 Subtle=SCALAR(0x8e5b8) at /foo/test line 13.
块结束之后 at /foo/test line 26.
程序结束... at /foo/test line 27.
销毁 Subtle=SCALAR(0x8e57c) during global destruction.
```

注意看“global destruction”这个地方！这里就是线程的垃圾回收阶段。甚至可以回收自引用的数据。

对象总是要析构，并且是在引用还没有析构的时候就进行析构，这样就可以阻止把一个已经析构的引用进行对象析构。普通引用仅仅只有当析构层次大于 0 时才开始析构。

//TODO: You can test the higher levels of global destruction by setting the PERL_DESTRUCT_LEVEL environment variable, presuming `-DDEBUGGING` was enabled during perl build time. See PERL_DESTRUCT_LEVEL in perlhack for more information.

一个更加完整的垃圾回收机制将在不久的将来完成。

在此期间，最好的解决方案就是创建一个非递归结构的包容器类，该包容器类包含有一个指针指向自引用的数据结构。然后为该被包容的对象类定义一个 DESTROY 方法，用 DESTROY 手工打破自引用的结构的循环。

3 参见

Perl 有几个入门级的面向对象编程教程，它们是 perltoot, perlboot 和 perltooc。你还可以在 perlbot 里看到关于对象的技巧、陷阱、和提示。另外，perlmodlib 中还有一些制作自己的类和模块的指南。

4 翻译者及翻译声明

本文由 flw (`flw@cpan.org`) 翻译，翻译成果首次出现在 中国 Perl 协会 http://www.perlchina.org) 的协作开发平台上。

PerlChina.org 本着“在国内推广 Perl” 的目的，组织人员翻译本文。读者可以在遵守原作者许可协议、尊重原作者及译作者劳动成果的前提下，任意发布或修改本文。

如果你对本文有任何意见，欢迎来信指教。本人非常欢迎与各位交流。