

第十二章 对象(下)

↓ 第十二章 对象(下)

- ↓ 12.5.1 通过 @ISA 继承
- ↓ 12.5.2 访问被覆盖的方法
- ↓ 12.5.3 UNIVERSAL: 最终的祖先类
- ↓ 12.5.4 方法自动装载
- ↓ 12.5.5 私有方法
- ↓ 12.6 实例析构器
 - ↓ 12.6.1 用 DESTROY 方法进行垃圾收集
- ↓ 12.7 管理实例数据
 - ↓ 12.7.1 用 use fields 定义的域
 - ↓ 12.7.2 用 Class::Struct 生成类
 - ↓ 12.7.3 使用 Autoloading (自动装载) 生成指示器
 - ↓ 12.7.4 用闭合域生成指示器
 - ↓ 12.7.5 将闭合域用于私有对象
 - ↓ 12.7.6 新技巧
- ↓ 12.8 管理类数据
- ↓ 12.9 总结

12.5.1 通过 @ISA 继承

如果 @ISA 包含多于一个包的名字, 包的搜索都是从左向右的顺序进行的。这些搜索是由浅入深的, 因此, 如果你有一个 Mule 类有象下面这样的继承关系:

```
package Mule;
our @ISA= ("Horse", "Donkey");
```

Perl 将首先在 Horse 里 (和他的任何前辈类里, 比如 Critter) 查找任何在 Mule 里找不到的方法, 找不到以后才继续在 Donkey 和其父类里进行查找。

如果缺失的方法在一个基类里发现, Perl 内部把该位置缓存在当前类里, 依次提高效率, 这样要查找该方法的时候, 它不用再跑老远。对 @ISA 的修改或者定义新的方法就会令该缓存失效, 因此导致 Perl 再次执行查找。

当 Perl 搜索一个方法的时候, 它确信你没有创建一个闭环的继承级别。如果两个类互相继承则可能出现这个问题, 甚至是间接地通过其他类这样继承也如此。试图做你自己的祖父即使对 Perl 而言也是荒谬的, 因此这样的企图导致抛出一个例外。不过, 如果从多于一个类继承下来, 而且这些类共享同样的祖宗, Perl 不认为是错误, 这种情况类似近亲结婚。你的继承级别看起来不再象一棵树, 而是象一个油脂分子图。不过这样不会为难 Perl——只要这个图形是真的油脂分子。

当你设置 @ISA 的时候, 赋值通常发生在运行时, 因此除非你加以预防, 否则在 BEGIN, CHECK, 或者 INIT 块里的代码不能在继承级别里使用。预防之一是 use base 用法, 它令你 require 类并且在编译时把它加入 @ISA 里。下面是你使用它们的方法:

```
packge Mule;
use base ("Horse", "donkey");      # 声明一个超类
```

它是下面东西的缩写:

```
package Mule;
BEGIN {
    our @ISA = ("Horse", "Donkey");
    require Horse;
    require Donkey;
```

```
}
```

只是 **use base** 还算入所有 **use fields** 声明中。

有时候人们很奇怪的是，在 **@ISA** 中包含一个类并没有为你 **require**（请求）合适的模块。那是因为 **Perl** 的类系统很大程度上是与它的模块相冲突的。一个文件可以保存许多类（因为他们只是包），而一个包可以在许多文件中提及。但是在最常见的情况下，一个包、一个类、一个模块、一个文件这样总是相当具有可换性——只要你足够倾斜，**use base** 用法提供了一个声明性的语法，用这个语法可以建立继承，装载模块文件和并且提供任意声明了的基类域。它也是我们不停提到的最便利的对角线。参阅第三十一章的 **use base** 和 **use fields** 获取细节。

12.5.2 访问被覆盖的方法

当一个类定义一个方法，那么该子过程覆盖任意基类中同名的方法。想象一下你有一个 **Mule**（骡）对象（它是从 **Horse**（马）类和 **Donkey**（驴）类衍生而来的），而且你想调用你的对象的 **breed**（种）方法。尽管父类有它们自己的 **breed**（种）方法，**Mule**（骡）类的设计者通过给 **Mule**（骡）类写自己的 **breed** 方法覆盖了那些父类的 **breed** 方法。这这意味着下面的交叉引用可能不能正确工作：

```
$stallion = Horse->new(gender => "male");
$molly = Mule->new(gender => "female");
$colt = $molly->breed($stallion);
```

现在假设基因工程的魔术治好了骡子臭名昭著的不育症，因此你想忽略无法生活的 **Mule::breed** 方法。你可以象平常那样调用你的子过程，但是一定要确保明确传递调用者：

```
$colt = Horse::breed($molly, $stallion);
```

不过，这是回避继承关系，实际上总是错误的处理方法。我们很容易想象实际上没有这么个 **Horse::breed** 子过程，因为 **Horse** 和 **Donkeys** 都是从一个公共的叫 **Equine**（马）的父类继承来的那个秉性。从另一方面来讲，如果你希望表明 **Perl** 应该从一个特定类开始搜索某方法，那么只要使用普通的方法调用方式，只不过用方法名修饰类名字就可以了：

```
$colt = $molly->Horse::breed($stallion);
```

有时候，你希望一个衍生类的方法表现得象基类中的某些方法的封装器。实际上衍生类的方法本身就可以调用基类的方法，在调用前或者调用后增加自己的动作。你可以把这种表示法只用于演示声明从哪个类开始搜索。但是在使用被覆盖方法的大多数情况下，你并不希望自己要知道或者声明该执行哪个父类被覆盖了的方法。

这就是 **SUPER** 伪类提供便利的地方。它令你能够调用一个覆盖了的基类方法，而不用声明是哪个类定义了该方法。（注：不要把这个和第十一章的覆盖 **Perl** 的内建函数的机制混淆了，那个不是对象方法并且不会被继承覆盖。你调用内建函数的覆盖是通过 **CORE** 伪包，而不是 **SUPER** 伪包。）下面的子过程在当前包的 **@ISA** 里查找，而不会要求你声明特定类：

```
package Mule;
our @ISA = qw(Horse Donkey);
sub kick {
    my $self = shift;
    print "The mule kicks!\n";
    $self->SUPER::kick(@_);
}
```

SUPER 伪包只有在一个方法里使用才有意义。尽管一个类的实现器可以在它们自己的代码里面使用 **SUPER**，但那些使用一个类的对象的却不能。

当出现多重继承的时候，**SUPER** 不总是按照你想象的那样运行。你大概也可以猜到，它也象 **@ISA** 那样遵循普通的继承机制的规则：从左向右，递归，由浅入深。如果 **Horse** 和 **Donkey** 都有一个 **speak** 方法，而你需要使用 **Donkey** 的方法，你将不得不明确命名该父类：

```
sub speak {
    my $self = shift;
    print "The mule speaks!\n";
    $self->Donkey::speak(@_);
}
```

用于多重继承的情况的更灵活的方法可以用 **UNIVERSAL::can** 方法进行雕琢，该方法下节介绍。或者你可以从 CPAN 抓 **Class::Multimethods** 方法下来，它提供了更灵活的解决方法，包括搜索最接近的，而不是最左端的。

Perl 里的每一段代码都知道自己现在在哪个包里，就象最后的 **package** 语句说的那样。只有在调用 **SUPER** 的包编译过以后，**SUPER** 才询问 **@ISA**。它不关心调用者的类，也不关心调用的子过程所属的包。不过，如果你想在另外一个类中定义方法，而且只是修改方法名，那么就有可能出问题：

```
package Bird;
use Dragonfly;
sub Dragonfly::divebomb { shift->SUPER::divebomb(@_) }
```

不幸的是，这样会调用 **Bird** 的超类，而不是 **Dragonfly** 的。要想按照你的意愿做事，你还得为 **SUPER** 的编译明确地切换到合适的包：

```
package Bird;
use Dragonfly;
{
    package Dragonfly;
    sub divebomb { shift->SUPER::divebomb(@_) }
}
```

如上例所示，你用不着只是为了给某个现有类增加一个方法去编辑一个模块。因为类就是一个包，而方法就是一个子过程，你所要做的就是在那个包里定义一个函数，就象我们上面做的那样，然后该类就一下子有了一个新方法。没有要求继承。只需要考虑包，而因为包是全局的，程序的任意位置都可以访问任意包。（小意思！湿湿碎！）

12.5.3 UNIVERSAL：最终的祖先类

如果对调用者的类和所有他的祖先类递归搜索后，还没有发现有正确名字的方法定义，那么会在一个预定义的叫 **UNIVERSAL** 的类中最后再搜索该方法一次。这个包从来不会在 **@ISA** 中出现，但如果查找 **@ISA** 失败总是要查找它。你可以把 **UNIVERSAL** 看作最终的祖先，所有类都隐含地从它衍生而来。

在 **UNIVERSAL** 类里面有下面的预定义的方法可以使用，因此所有类中都可以用它们。而且不管它们是当被当作类方法还是对象方法调用的都能运行。

```
INVOCANT->isa(CLASS)
```

如果 **INVOCANT** 的类是 **CLASS** 或者任何从 **CLASS** 继承来的，**isa** 方法返回真。除了包名字以外，**CLASS** 还可以是一个内建的类型，比如 **"HASH"** 或者 **"ARRAY"**。（准确地检查某种类型在封装和多态性机制中并不能很好地工作。你应该依赖重载分检给你正确的方法。）

```
use FileHandle;
if (FileHandle->isa("Exporter")) {
```

```

        print "FileHandle is an Exporter.\n";
    }

    $fh = FileHandle->new();
    if ($fh->isa("IO::Handle")) {
        print "\$fh is some sort of IOish object.\n"
    }
    if ($fh->isa("GLOB")) {
        print "\$fh is really a GLOB reference.\n";
    }

    INVOCANT->can(METHOD)

```

如果 **INVOCANT** 中有 **METHOD**，那么 **can** 方法就返回一个可以调用的该子过程的 引用。如果没有定义这样的子过程，**can** 返回 **undef**。

```

    if ($invocant->can("copy")) {
        print "Our invocant can copy.\n";
    }

```

我们可以用这个方法实现条件调用——只有方法存在才调用：

```

    $obj->snarl if $obj->can("snarl");

```

在多重继承情况下，这个方法允许调用所有覆盖掉的基类的方法，而不仅仅是最 左边的那个：

```

sub snarl {
    my $self = shift;
    print "Snarling: @_ \n";
    my %seen;
    for my $parent (@ISA) {
        if (my $code = $parent->can("snarl")) {
            $self->$code(@_) unless $seen{$code}++;
        }
    }
}

```

我们用 **%seen** 散列跟踪那些我们已经调用的子过程，这样我们才能避免多次调用 同一个子过程。这种情况在多个父类共享一个共同的祖先的时候可能发生。

会触发一个 **AUTOLOAD**（在下一节描述）的方法将不会被准确地汇报，除非该包已 经声明（但没有定义）它需要自动装载的子过程了。

INVOCANT-VERSION(NEED)

VERSION 方法返回 **INVOCANT** 的类的版本号，就是存贮在包的 **\$VERSION** 变量里的 那只。如果提供了 **NEED** 参数，它表示当前版本至少不能小于 **NEED**，而如果真的 小于就会抛出一个例外。这是 **use** 用以检查一个模块是否足够新所调用的方法。

```

use Thread 1.0;      # 调用 Thread->VERSION(1.0)
print "Running versino ", Thread->VERSION, " of Thread.\n";

```

你可以提供自己的 **VERSION** 方法覆盖掉 **UNIVERSAL** 的。不过那样会令任何从 你的类衍生的类也使用哪个覆盖类。如果你不想发生这样的事情，你应该把你的 方法设计成把其他类的版本请求返回给**UNIVERSAL**。

在 **UNIVERSAL** 里的方法是内建的 **Perl** 子过程，如果你使用全称并且传递两个参数，你就可以调用

它们，比如 `UNIVERSAL::isa($formobj, "HASH")`。（但是我们不推荐这么用，因为通常而言 `can` 包含你真正在找的答案。）

你可以自由地给 `UNIVERSAL` 增加你自己的方法。（当然，你必须小心；否则你可能真的把事情搞砸，比如有些东西是假设找不到你正在定义的方法名的，这样它们就可以从其他地方自动装载进来。）下面我们创建了一个 `copy` 方法，所有类的对象都可以使用——只要这些对象没有定义自己的。（我们忘了给调用一个对象做解析。）

```
use Data::Dumper;
use Carp;
sub UNIVERSAL::copy {
    my $self = shift;
    if (ref $self) {
        return eval Dumper($self);    # 没有 CORE 引用
    } else {
        confess "UNIVERSAL::copy can't copy class $self";
    }
}
```

如果该对象包含任意到子过程的引用，这个 `Data::Dumper` 的策略就无法运转，因为它们不能正确地复现。即使能够拿到源程序，词法绑定仍然会丢失。

12.5.4 方法自动装载

通常，当你调用某个包里面未定义子过程，而该包定义了一个 `AUTOLOAD` 子过程，则调用该 `AUTOLOAD` 子过程并且抛出一个例外（参阅第十章，“自动装载 Autoloading”）。方法的运做略有不同。如果普通的方法查找（通过类，它的祖先以及最终的 `UNIVERSAL`）没能找到匹配，则再按同样的顺序运行一遍，这次是查找一个 `AUTOLOAD` 子过程。如果找到，则把这个子过程当作一个方法来调用，同时把包的 `$AUTOLOAD` 变量设置为该子过程的全名（就是代表 `AUTOLOAD` 调用的那个子过程。）

当自动装载方法的时候，你得小心一些。首先，如果 `AUTOLOAD` 的子过程代表一个叫 `DESTROY` 的方法调用，那么它应该立即返回，除非你的目的是仿真 `DESTROY`，那样的话对 Perl 有特殊含义，我们将在本章后面的“实例析构器”里描述。

```
sub AUTOLOAD {
    return if our $AUTOLOAD =~ /:DESTROY$/;
    ...
}
```

第二，如果该类提供一个 `AUTOLOAD` 安全网，那么你就不能对一个方法名使用 `UNIVERSAL::can` 来检查调用该方法是否安全。你必须独立地检查 `AUTOLOAD`：

```
if ($obj->can("methname") || $obj->can("AUTOLAOD")) {
    $obj->methname();
}
```

最后，在多重继承的情况下，如果一个类从两个或者更多类继承过来，而每个类都有一个 `AUTOLOAD`，那么只有最左边的会被触发，因为 Perl 在找到第一个 `AUTOLOAD` 以后就停下来了。

后两个要求可以很容易地通过声明包里的子过程来绕开，该包的 `AUTOLOAD` 就是准备管理这些方法的。你可以用独立的声明实现这些：

```
package Goblin;
```

```
sub kick;
sub bite;
sub scratch;
```

或者用 **use subs** 用法，如果你有许多方法要声明，这样会更方便：

```
package Goblin;
use subs qw(kick bite scratch);
```

甚至你只是声明了这些子过程而并没有定义它们，系统也会认为它是真实的。它们在一个 **UNIVERSAL::can** 检查里出现，而且更重要的是，它们在搜索方法的第二步出现，这样就永远不会进行第三步，更不用说第四步了。

“不过，”你可能会说，“它们调用了 **AUTOLOAD**，不是吗？”的确，它们最终调用了 **AUTOLOAD**，但是机制是不一样的。一旦通过第二步找到了方法存根(stub)，Perl 就会试图调用它。当最后发现该方法不是想要的方法时，则再次进行 **AUTOLOAD** 搜索，不过这回它从包含存根该类开始搜索，这样就把方法的搜索限制在该类和该类的祖先（以及 **UNIVERSAL**）中。这就是 Perl 如何查找正确的 **AUTOLOAD** 来运行和如何忽略来自最初的继承树中错误的 **AUTOLOAD** 部分的方法。

12.5.5 私有方法

有一个调用方法的手段可以完全令 Perl 忽略继承。如果用的不是一个文本方法名，而是一个简单的标量变量，该变量包含一个指向一个子过程的引用，则立即调用该子过程。在前一节的 **UNIVERSAL->can** 的描述中，最后一个例子使用子过程的引用而不是其名字调用所有被覆盖了的方法。

这个特性的一个非常诱人的方面是他可以用于实现私有方法调用。如果你的类放在一个模块里，你可以利用文件的词法范围为私有性服务。首先，把一个匿名子过程存放在一个文件范围的词法里：

```
# 声明私有方法
my $secret_door = sub {
    my $self = shift;
    ...
};
```

然后在这个文件里，你可以把那个变量当作保存有一个方法名这样来使用。这个闭合将会被直接调用，而不用考虑继承。和任何其他方法一样，调用者作为一个额外的参数传递进去。

```
sub knock {
    my $self = shift;
    if ($self->{knocked}++ > 5) {
        $self->$secret_door();
    }
}
```

这样就可以让该文件自己的子过程（类方法）调用一个代码超出该词法范围（因而无法访问）的方法。

12.6 实例析构器

和 Perl 里任何其他引用一样，当一个对象的最后一个引用消失以后，该对象的存储器隐含地循环使用。对于一个对象而言，你还有机会在这些事情发生的时候（对象内存循环使用）捕获控制，方法是在类的包里定义 **DESTROY** 子过程。这个方法在合适的时候自动被触发，而将要循环使用的对象是它的唯一的参数。

在 Perl 里很少需要析构器，因为存储器管理是自动进行的。不过有些对象可能有一个位于存储器系统之外的状态（比如文件句柄或数据库联接），而且你还想控制它们，所以析构器还是有用的。

```
package MailNotify;
sub DESTROY {
    my $self = shift;
    my $fh = $self->{mailhandle};
    my $id = $self->{name};
    print $fh "\n$id is signing off at " . localtime( ) . "\n";
    close $fh;    # 关闭mailer的管道
}
```

因为 Perl 只使用一个方法来构造一个对象，即使该构造器的类是从一个或者多个其他类继承过来的也这样，Perl 也只是每个对象使用一个 DESTROY 方法来删除对象，也不管继承关系。换言之，Perl 并不为你做分级析构。如果你的类覆盖了一个父类的析构器，那么你的 DESTROY 方法可能需要调用任意适用的基类的 DESTROY 方法：

```
sub DESTROY {
    my $self = shift;
    # 检查看看有没有覆盖了的析构器
    $self->SUPER::DESTROY if $self->can("SUPER::DESTROY");
    # 现在干你自己的事情
}
```

这个方法只适用于继承的类；一个对象只是简单地包含在当前对象里——比如，一个大的散列表里的一个数值——会被自动释放和删除。这也是为什么一个简单地通过聚集（有时候叫“有xx”关系）实现的包含器要更干净，并且比继承（一个“是xx”关系）更干净。换句话说，通常你实际上只需要把一个对象直接保存在另外一个对象里面而不用通过继承，因为继承会增加不必要的复杂性。有时候当你诉诸多重继承的时候，实际上单继承就足够用了。

你有可能明确地调用 DESTROY，但实际上很少需要这么做。这么做甚至是有害的，因为对同一个对象多次运行析构器可能会有让你不快的后果。

12.6.1 用 DESTROY 方法进行垃圾收集

正如第八章的“垃圾收集，循环引用和弱引用”节里介绍的那样，一个引用自身的变量（或者多个变量间接的相互引用）会一直到程序（或者嵌入的解释器）快要退出的时候才释放。如果你想早一些重新利用这些存储器，你通常是不得不使用 CPAN 上的 [WeakRef²](#) 方法来明确地打破或者弱化该引用。

对于对象，一个候补的解决方法是创建一个容器类，该容器类保存一个指向这个自引用数据结构的指针。为该被包含对象的类定义一个 DESTROY 方法，该方法手工打破自引用结构的循环性。你可以在 Perl Cookbook 这本书的第十三章里找到关于这些的例子，该章的名称是，“Coping with Circular Data Structures”（对付循环数据结构）。

当一个解释器退出的时候，它的所有对象都删除掉，这一点对多线程或者嵌入式的 Perl 应用非常重要。对象总是在普通引用被删除之前在一个独立的回合里被删除。这样就避免了 DESTROY 方法处理那些本身已经被删除的引用。（也是因为简单引用只有在嵌入的解释器中才会被当作垃圾收集，因为退出一个进程是回收引用的非常迅速的方法。但是退出并不运行对象的析构器，因此 Perl 先做那件事。）

12.7 管理实例数据

大多数类创建的对象实际上都是有几个内部数据域（实例数据）和几个操作数据域的方法的数据结构。

Perl 类继承方法，而不是数据，不过由于所有对对象的访问都是通过方法调用进行的，所以这样运行得很好。如果你想继承数据，那么你必须通过方法继承来实现。不过，Perl 在多数情况下是不需要这么做的，因为大多数类都把它们的对象的属性保存在一个匿名散列表里。对象的实例数据保存在这个散列表里，这个散列表也是该对象自己的小名字空间，用以划分哪个类对该对象进行了哪些操作。比如，如果你希望一个叫 `$city` 的对象有一个数据域名字叫 `elevation`，你可以简单地 `$city->{elevation}` 这样访问它。可以不用声明。方法的封装会为你做这些。

假设你想实现一个 `Person` 对象。你决定它有一个叫“`name`”的数据域，因为某种奇怪的一致性原因，你将把它按照键字 `name` 保存在该匿名散列表里，该散列表就是为这个对象服务的。不过你不希望用户直接修改数据。要想获得封装的优点，用户需要一些方法来访问该实例变量，而又不需揭开抽象的面纱。

比如，你可能写这样的一对访问方法：

```
sub get_name {
    my $self = shift;
    return $self->{name};
}

sub set_name {
    my $self = shift;
    $self->{name} = shift;
}
```

它们会导致下面的代码的形成：

```
$him = Person->new();
$him->set_name("Laser");
$him->set_name( ucfirst($him->get_name) );
```

你甚至可以把两个方法组合成一个：

```
sub name {
    my $self = shift;
    if (@_) { $self->{name} = shift }
    return $self->{name};
}
```

这样会形成下面的代码：

```
$him = Person->new();
$him->name("BitBIRD");
$him->name( ucfirst($him->name) );
```

给每个实例变量（对于我们的 `Person` 类而言可能是 `name`, `age`, `height` 等等）写一个独立的函数的优点是直接，明显和灵活。缺点是每当你需要一个新的类，你最终都要对每个实例变量定义一个或两个几乎相同的方法。对于开头的少数几个类而言，这么做不算太坏，而且如果你喜欢这么干的话我们也欢迎你这么干。但是如果便利比灵活更重要，那么你可能就会采用后面描述的那种技巧。

请注意我们会变化实现，而不是接口。如果你的类的用户尊重封装，那么你就可以透明地从一种实现切换到另外一种实现，而不会让你的用户发现。（如果你的继承树里的家庭成员把你的类用于子类或者父类，那可能不能这么宽容了，毕竟它们对年你的认识比陌生人要深刻得多。）如果你的用户曾经

深入地刺探过你的类中的私有部分，那么所导致的不可避免的损害就是他们自己的问题而不是你的。你能所做的一切就是通过维护好你的接口来快乐地过日子。试图避免这个世界里的每一个人做出一些有些恶意的事情会消耗掉你的所有时间和精力，并且最终你会发现还是徒劳的。

对付家成员更富挑战性。如果一个子类覆盖了一个父类的属性的指示器，那么它是应该访问散列表中的同一个域呢还是不应该？根据该属性的性质，不管那种做法都会产生一些争论。从通常的安全性角度出发，每个指示器都可以用它自己的类名字作为散列域名字的前缀，这样子类和父类就都可以有自己的版本。下面有几个使用这种子类安全策略的例子，其中包括标准的 **Struct::Class** 模块。你会看到指示器是这样组成的：

```
sub name {
    my $self = shift;
    my $field = __PACKAGE__ . "::$name";
    if (@_) { $self->{$field} = shift }
    return $self->{$field};
}
```

在随后的每个例子里，我们都创建一个简单的 **Person** 类，它有 **name**，**race**，和 **aliases** 域，每种类都有一个相同的接口，但是有完全不同的实现。我们不准告诉你我们最喜欢哪种实现，因为根据实际使用的环境，我们几乎都喜欢。有些人喜欢弯曲的实现，有些人喜欢直接的实现。

12.7.1 用 **use fields** 定义的域

对象不一定要用匿名散列来实现。任何引用都可以。比如，如果你使用一个匿名数组，你可以这样设置一个构造器：

```
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    return bless [], $class;
}
```

以及象下面这样的指示器：

```
sub name {
    my $self = shift;
    if (@_) { $self->[0] = shift }
    return $self->[0];
}

sub race {
    my $self = shift;
    if (@_) { $self->[1] = shift }
    return $self->[1];
}

sub aliases {
    my $self = shift;
    if (@_) { $self->[2] = shift }
    return $self->[2];
}
```

数组访问比散列快一些，而且占的内存少一些，不过用起来不象散列那样方便。你不得不跟踪所有下标数字（不仅仅在你自己的类里面，而且还得在你的父类等等里面），这些下标用于指示你的类正在使用的数组的部分。这样你才能重复使用这些空间。

`use fields` 用法可以对付所有这些问题：

```
package Person;
    use fields qw(name race aliases);
```

这个用法不会为你创建指示器方法，但是它的确是基于一些内建的方法之上（我们叫它伪散列）做一些类似的事情的。（不过你可能会希望对这些域用指示器进行封装，就象我们处理下面的例子一样。）伪散列是数组引用，你可以把它们当作散列那样来用，因为他们有一个相关联的键字映射表。

`use fields` 用法为你设置这些键字映射，等效于声明了哪些域对 `Person` 对象是有效的；以此令 Perl 的编译器意识到它们的存在。如果你声明了你的对象变量的类型（就象在下个例子里的 `my Person $self` 一样），编译器也会聪明得把对该域的访问优化成直接的数组访问。这样做更重要的原因可能是它令域名在编译时是类型安全的（实际上是敲键安全）。（参阅第八章里的“伪散列”。）

一个构造器和例子指示器看起来可能是这个样子的：

```
package Person;
use fields qw(name race aliases);
sub new {
    my $type = shift;
    my Person $self = fields::new(ref $type || $type);
    $self->{name} = "unnamed";
    $self->{race} = "unknown";
    $self->{aliases} = [];
    return $self;
}
sub name {
    my Person $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
sub race {
    my Person $self = shift;
    $self->{race} = shift if @_;
    return $self->{race};
}
sub aliases {
    my Person $self = shift;
    $self->{aliases} = shift if @_;
    return $self->{aliases};
}
1;
```

如果你不小心拼错了一个用于访问伪散列的文本键字，你用不着等到运行时才发现这些问题。编译器知道对象 `$self` 想要引用的数据类型（因为你告诉它了），因此它就可以对那些只访问 `Person` 对象实际拥有的数据域的代码。如果你走神了，并且想访问一个不存在的数据域（比如 `$self->{mane}`），那么编译器可以马上标出这个错误并且绝对不会让有错误的程序跑到解释器那里运行。

这种方法在声明获取实例变量的方法的时候仍然有些重复，所以你可能仍然喜欢使用下面介绍的技巧之一，这些技巧实现了简单指示器方法的自动创建。不过，因为所有的这些技巧都使用某种类型的间接引用，所以如果你使用了这些技巧，那么你就会失去上面的编译时词法类型散列访问的拼写检查好处。当然，你还是能保留一点点的时间和空间的优势。

如果你决定使用一个伪散列来实现你的类，那么任何从这个类继承的类都必须知晓下面的类的伪散列

实现。如果一个对象是用伪散列实现的，那么所有继承分级中的成员都必须使用 `use base` 和 `use fields` 声明。比如：

```
package Wizard;
use base "Person";
user fields qw(staff color sphere);
```

这么干就把 `Wizard` 模块标为 `Person` 的子类，并且装载 `Person.pm` 文件。而且除了来自 `Person` 的数据域外，还在这个类中注册了三个新的数据域。这样，当你写：

```
my Wizard $mage = fields::new("Wizard");
```

的时候，你就能得到一个可以访问两个类的数据域的伪散列：

```
$mage->name("Gandalf");
$mage->color("Grey");
```

因为所有子类都必须知道他们用的是一种伪散列的实现，所以，从效率和拼写安全角度出发，它们应该使用直接伪散列句法：

```
$mage->{name} = "Gandalf";
$mage->{color} = "Grey";
```

不过，如果你希望保持你的实现的可互换性，那么你的类以外的用户必须使用指示器方法。

尽管 `use base` 只支持单继承，但也不算上非常严重的限制。参阅第三十一章的 `use base` 和 `use fields` 的描述。

12.7.2 用 `Class::Struct` 生成类

标准的 `Class::Struct` 模块输出一个叫 `struct` 的函数。它创建了你开始构造一个完整的类所需要的所有机关。它生成一个叫 `new` 的构造器，为每个该结构里命名的数据域增加一个指示器方法（实例变量）。

比如，如果你把下面结构放在一个 `Person.pm` 文件里：

```
package Person;
use Class::Struct;
struct Person => {    # 创建一个"Person"的定义
    name => '$',      # name域是一个标量
    race => '$',      # race域也是一个标量
    aliases => '@',   # 但 aliases 域是一个数组引用
};
1;
```

然后你就可以用下面的方法使用这个模块：

```
use Person;
my $mage = Person->new();
$mage->name("Gandalf");
$mage->race("Istar");
$mage->aliases( ["Mithrandir","Olorin", "Incanus"] );
```

`Class::Struct` 模块为你创建上面的所有四种方法。因为它遵守子类安全原则，总是在域名字前面前缀类名字，所以它还允许一个继承类可以拥有它自己独立的与基类同名的域，而又不担心会发生冲突。这就意味着你在用于这个实例变量的时候，必须用 `"Person::name"` 而不能用 `"name"` 当作散列键字来访问散列表。

在结构声明里的数据域可以不是 Perl 的基本类型。它们也可以声明其他的类，但是和 **struct** 一起创建的类并非运行得最好，因为那些对类的特性做出假设的函数并不是对所有的类都能够明察秋毫。比如，对于合适的类而言，会调用 **new** 方法来初始化它们，但是很多类有其他名字的构造器。

参阅第三十二章，标准模块，以及它的联机文档里关于 **Class::Struct** 的描述。许多标准模块使用 **Class::Struct** 来实现它们的类，包括 **User::pwent** 和 **Net::hostent**。阅读它们的代码会很有收获。

12.7.3 使用 Autoloading（自动装载）生成指示器

正如我们早先提到过的，当你调用一个不存在的方法的时候，Perl 有两种不同的手段搜索一个 **AUTOLOAD** 方法，使用哪种方法取决于你是否声明了一个存根方法。你可以利用这个特性提供访问对象的实例数据的方法，而又不需为每个实例书写独立的函数。在 **AUTOLOAD** 过程内部，实际被调用的方法的名字可以从 **\$AUTOLOAD** 变量中检索出来。让我们看看下面下面的代码：

```
user Person;
$him = Person->new;
$him->name("Weiping");
$him->race("Man");
$him->aliases( ["Laser", "BitBIRD", "chemi"] );
printf "%s is of the race of %s. \n", $him->name, $him->race;
printf "His aliases are: ", join(", ", @{$him->aliases}), ".\n";
```

和以前一样，这个版本的 **Person** 类实现了一个有三个域的数据结构：**name**，**race**，和 **aliases**：

```
package Person;
use Carp;

my %Fields = (
    "Person::name" => "unnamed",
    "Person::race" => "unknown",
    "Person::aliases" => [],
);

# 下一个声明保证我们拿到自己的autoloader（自动装载器）。

use subs qw(name race aliases);

sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = { %Fields, @_ }; # 类似Class::Struct的克隆
    bless $self, $class;
    return $self;
}

sub AUTOLOAD {
    my $self = shift;
    # 只处理实例方法，而不处理类方法
    croak "$self not an object" unless ref($invocant);
    my $name = our $AUTOLOAD;
    return if $name =~ /::DESTROY$/;
    unless (exist $self->{$name}) {
        croak "Can't access '$name' field in $self";
    }
}
```

```

        if (@_) {return $self->{$name} = shift }
        else { return $self->{$name} }
    }
}

```

如你所见，这里你可找不到叫 `name`，`race`，或者 `aliases` 的方法。`AUTOLOAD` 过程为你照看那些事情。当某人使用 `$him->name("Aragorn")` 的时候，那么 `Perl` 就调用 `AUTOLOAD`，同时把 `$AUTOLOAD` 设置为 `"Person::name"`。出于方便考虑，我们用了全名，这是访问保存在对象散列里的数据域的最正确的方法。这样你可以把这个类用做一个更大的继承级中的一部分，而又不用担心会和其他类中使用的同名数据域冲突。

12.7.4 用闭合域生成指示器

大多数指示器方法实际上干的是一样的事情：它们只是简简单单地从实例变量中把数值抓过来并保存起来。在 `Perl` 里，创建一个近乎相同的函数族的最自然的方法就是在一个闭合区域里循环。但是闭合域是匿名函数，它们缺少名字，而方法必须是类所在包的符号表的命名子过程，这样它们才能通过名字来调用。不过这不算什么问题——只要把那个闭合域赋值给一个名字合适的类型团就可以了。

```

package Person;

sub new {
    my $invocant = shift;
    my $self = bless( {}, ref $invocant || $invocant );
    $self->init();
    return $self;
}

sub init {
    my $self = shift;
    $self->name("unnamed");
    $self->race("unknown");
    $self->aliases([]);
}

for my $field (qw(name race aliases)) {
    my $slot = __PACKAGE__ . "::$field";
    no strict "refs";    # 这样指向类型团的符号引用就可以用了
    *$field = sub {
        my $self = shift;
        $self->{$slot} = shift if @_;
        return $self->{$slot};
    };
}

```

闭合域是为你的实例数据创建一个多用途指示器的最干净的操作方法。而且不论对计算机还是你而言，它都很有效。不仅所有指示器都共享同一段代码（它们只需要它们自己的词法本），而且以后你要增加其他属性也方便，你要做的修改是最少的：只需要给 `for` 循环增加一条或更多条单词，以及在 `init` 方法里加上几句就可以了。

12.7.5 将闭合域用于私有对象

到目前为止，这些管理实例数据的技巧还没有提供"避免"外部对数据的访问的机制。任何类以外的对象都可以打开对象的黑盒子然后查看内部--只要它们不怕质保书失效。增强私有性又好象阻碍了人们完成任务。`Perl` 的哲学是最好把一个人的数据用下面的标记封装起来：

```
IN CASE OF FIRE
```

BREAK GLASS

如果可能，你应该尊重这样的封装，但你在紧急情况（比如调试）下仍然可以很容易地访问其内容。

但是如果你确实想强调私有性，Perl 不会给你设置障碍。Perl 提供低层次的制作块，你可以用这些制作块围绕在你的类和其对象周围形成无法进入的私有保护网--实际上，它甚至比许多流行的面向对象的语言的保护还要强。它们内部的词法范围和词法变量是这个东西的关键组件，而闭合域起到了关键的作用。

在"私有方法"节里，我们看到了一个类如何才能使用闭合域来实现那种模块文件外部不可见的方法。稍后我们将看看指示器方法，它们把类数据归拢得连类的其他部分都无法进行不受限制的访问。那些仍然是闭合域相当传统的用法。真正让我们感兴趣的东西是把闭合域用做一个对象。该对象的实例变量被锁在该对象内部--也就是说，闭合域，也只有闭合域才能自由访问。这是非常强的封装形式；这种方法不仅可以防止外部对对象内部的操作，甚至连同一个类里面的其他方法也必须使用恰当的访问方法来获取对象的实例数据。

下面是一个解释如何实现这些的例子。我们将把闭合域同时用于生成对象本身和生成指示器：

```
package Person;
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $data = {
        NAME => "unnamed",
        RACE => "unknown",
        ALIASES => [],
    };

    my $self = sub {
        my $field = shift;
        #####
        ### 在这里进行访问检查      ###
        #####
        if (@_) { $data->{$field} = shift }
        return $data->{$field};
    };
    bless ($self, $class);
    return $self;
};
```

生成方法名

```
for my $field (qw(name race aliases)) {
    no strict "refs";    # 为了访问符号表
    *$field = sub {
        my $self = shift;
        return $self->(uc $field, @_);
    };
}
```

`new` 方法创建和返回的对象不再是一个散列，因为它在我们刚才看到的其他的构造器里。实际上是一个闭合域访问存储在散列里的属性数据，该闭合域是唯一可以访问该属性数据的类成员，而存储数据的散列是用 `$data` 引用的。一旦构造器调用完成，访问 `$data`（里面的属性）的唯一方法就是通过闭合域。

在一个类似 `$him->name("Bombadil")` 这样的调用中，在 `$self` 里存储的调用对象是那个闭合域，这个闭合域已经由构造器赐福（`bless`）并返回了。对于一个闭合而言，我们除了能调用它以外，干不了太多什么事，因此我们只是做 `$self->(uc $field, @_)`。请不要被箭头糊弄了，这条语句只是一个正规的间接函数调用，而不是一个方法调用。初始参数是字串 `"name"`，而其他的参数就是那些传进来的。（注：当然，双函数调用比较慢，但是如果你想快些，你还会首先选用对象吗？）一旦我们在闭合域内部执行，那么在 `$data` 里的散列就又可以访问得到了。这样闭合域就可以自由地给任何它愿意的对象访问权限，而封杀任何它讨厌的对象的访问。

在闭合域外面没有任何对象可以不经中介地访问这些非常私有的实例数据，甚至该类里面的其他方法都不能。它们可以按照 `for` 循环生成的方法来调用闭合域，可能是设置一个该类从来没有听说过的实例变量。但是我们很容易通过在构造器里放上几段代码来阻止这样的方法使用，放那些代码的地方就是你看到的上面的访问检查注释的地方。首先，我们需要一个通用的导言：

```
use Carp;
local $Carp::CarpLevel = 1;    # 保持牢骚消息短小
my ($cpack, $cfile) = caller();
```

然后我们进行各个检查。第一个要确保声明的属性名存在：

```
croak "No valid field '$field' in object"
    unless exists $data->{$field};
```

下面这条语句只允许来自同一个文件的调用：

```
carp "Unmediated access denied to foreign file"
    unless $cfile eq __FILE__;
```

下面这条语句只允许来自同一个包的调用：

```
carp "Unmediated access denied to foreign package ${cpack}::"
    unless $cpack eq __PACKAGE__;
```

所有这些代码都只检查未经中介的访问。那些有礼貌地使用该类指定的方法访问的用户不会受到这些约束。**Perl** 会给你一些工具，让你想多挑剔就有多挑剔。幸运的是，不是所有人都这样。

不过有些人应该挑剔。当你写飞行控制软件的时候，严格些就是正确的了。如果你想成为或者要成为这些人员，而且你喜欢使用能干活的代码而不是自己重新发明所有东西，那么请看看 **CPAN** 上 **Damian Conway** 的 **Tie::SecureHash** 模块。它实现了严格的散列，支持公有，保护和私有约束。它还对付我们前面的例子中忽略掉的继承性问题。**Damian** 甚至还写了一个更雄心勃勃的模块，**Class::Contract**，在 **Perl** 灵活的对象系统上强加了一层正式的软件工程层。这个模块的特性列表看起来就象一本计算机科学教授的软件工程课本的目录，（注：你知道 **Damian** 是干什么的吗？顺便说一句，我们非常建议你看看他的书，**Object Oriented Perl**（面向对象的 **Perl**）（**Manning Publications, 1999**））。包括强制封装，静态继承和用于面向对象的 **Perl** 的按需设计条件检查，以及一些用于对象和类层次的属性，方法，构造器和析构器定义的可声明的语法，以及前提，后记和类固定。天！

12.7.6 新技巧

到了 **Perl 5.6**，你还可以声明一个方法并指出它是返回左值的。这些是通过做值子过程属性实现的（不要和对象方法混淆了）。这个实验性的特性允许你把该方法当作某些可以在一个等号左边出现的东西：

```
package Critter;

sub new {
```

```

        my $class = shift;
        my $self = { pups => 0, @_ };      # 覆盖缺省。
        bless $self, $class;
    }

    sub pups : lvalue {                  # 我们稍后给pups()赋值
        my $self = shift;
        $self->{pups};
    }

package main;
$varmint = Critter->new(pups => 4);
$varmint->pups *= 2;                      # 赋给 $varmint->pups!
$varmint->pups =~ s/(.)/$1$1/;           # 现场修改 $varmint->pups!
print $varmint->pups;                     # 现在我们有88个pups。

```

这么做让你以为 `$varmint->pups` 仍然是一个遵守封装的变量。参阅第六章，子过程，的“左值属性”。

如果你运行的是一个线程化的 Perl，并且你想确保只有一个线程可以调用一个对象的某个方法，你可以使用 `locked` 和 `method` 属性实现这些功能：

```

    sub pups : locked method {
        ...
    }

```

当任意线程调用一个对象上的 `pups` 方法的时候，Perl 在执行前锁住对象，阻止其他线程做同样的事情。参阅第六章里的“`locked` 和 `method` 属性”。

12.8 管理类数据

我们已经看到了按对象访问对象数据的几种不同方法。不过，有时候你希望有些通用的状态在一个类里的所有对象之间共享。不管你使用哪个类实例（对象）来访问他们，这些变量是整个类的全局量，而不只是该类的一个实例，（C++ 程序员会认为这些是静态成员数据。）下面是一些类变量能帮助你情况：

- 保存一个曾经创建的所有对象的计数，或者是仍然存活的数量。
- 保存一个你可以叙述的所有对象的列表
- 保存一个全类范围调试方法使用的日志文件的文件名或者文件描述符。
- 保存收集性数据，比如想一个网段里的所有ATM某一支取的现金的总额。
- 跟踪类创建的最后一个或者最近访问过的对象。
- 保存一个内存里的对象的缓存，这些对象是从永久内存中重新构建的。
- 提供一个反转的查找表，这样你就可以找到一个基于其属性之一的数值的对象。

然后问题就到了哪里去存储这些共享属性上面。Perl 没有特殊的语法机制用于声明类属性，用于实例属性的也多不了什么。Perl 给开发者提供了一套广泛强大而且灵活的特性，这些特性可以根据不同情况分别雕琢成适合特定的需要。然后你可以根据某种情况选择最有效的机制，而不是被迫屈服于别人的设计决定。另外，你也可以选择别人的设计决定——那些已经打包并且放到 CPAN 去的东西。同样，“回字有四种写法”。

和任何与类相关的东西一样，类数据不能被直接访问，尤其是从类实现的外部。封装的理论没有太多关于为实例变量设置严格受控的指示器方法的内容，但是却发明了 `public` 来直接欺骗你的类变量，就好象设置 `$SomeClass::Debug = 1`。要建立接口和实现之间干净的防火墙，你可以创建类似你用于实例数据的指示器方法来操作类数据。

假设我们想跟踪 **Critter** 对象的全部数量。我们将把这个数量存储在一个包变量里，但是提供一个方法调用 **population**，这样此类的用户就不用知道这个实现：

```
Critter->population()    # 通过类名字访问
$gollum->population()    # 通过实例访问
```

因为在 **Perl** 里，类只是一个包，存储一个类的最自然的位置是在一个包变量里。下面就是这样一个类的简单实现。**population** 方法忽略它的调用者并且只返回该包变量的当前值 **\$Population**。（有些程序喜欢给它们的全局量大写开头。）

```
package Critter;
our $population = 0;
sub pupulation { return $Population; }
sub DESTROY { $Population -- }
sub spawn {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    $Population++;
    return bless { name => shift || "anon" }, $class;
}
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

如果你想把类数据方法做得想实例数据的指示器那样，这么做：

```
our $Debugging = 0;    # 类数据
sub debug {
    shift;              # 有意忽略调用者
    $Debugging = shift if @_;
    return $Debugging;
}
```

现在你可以为给该类或者它的任何实例设置全局调试级别。

因为它是一个包变量，所以 **\$Debugging** 是可以全局访问的。但是如果你把 **our** 变量改成 **my**，那么就只有该文件里后面的代码可以看到它。你还可以走得再远一些——你可以把对类属性的访问限制在该类本身其余部分里。把该变量声明裹在一个块范围里：

```
{
    my $Debugging = 0;    # 词法范围的类数据
    sub debug {
        shift;           # 有意忽略调用者
        $Debugging = shift if @_;
        return $Debugging;
    }
}
```

现在没有人可以不通过使用指示器方法来读写该类属性，因为只有那个子过程和变量在同一个范围因而可以访问它。

如果一个生成的类继承了这些类指示器，那么它们仍然访问最初的数据，不管变量是用 **our** 还是用 **my** 定义的。数据是包无关的。当方法在它们最初定义的地方执行的时候，你可以看到它们，但是在

调用它的类里面可不一定看得到。

对于某些类数据，这个方法运行得很好，但对于其他的而言，就不一定了。假设我们创建了一个 `Critter` 的 `Warg` 子类。如果我们想分离我们的两个数量，`Warg` 就不能继承 `Critter` 的 `population` 方法，因为那个方法总是返回 `$Critter::Population` 的值。

你可能会不得不根据实际情况决定类属性与包相关是否有用。如果你想要包相关的属性，可以使用调用者的类来定位保存着类数据的包：

```
sub debug {  
    my $invocant = shift;  
    my $class = ref($invocant) || $invocant;  
    my $varname = $class . "::__Debugging";  
    no strict "refs";          # 符号访问包数据  
    $$varname = shift if @_;  
    return $$varname;  
}
```

我们暂时废除严格的引用，因为不这样的话我们就不能把符号名全名用于包的全局量。这是绝对有道理的：因为所有定义的包变量都存活在一个包里，通过该包的符号表访问它们是没什么错的。

另外一个方法是令对象需要的所有东西——甚至它的全局类数据——都可以由该对象访问（或者可以当作参数传递）。要实现这些功能，你通常不得不为每个类都做一个精制的构造器，或者至少要做一个构造器可以调用的精制的初始化过程。在构造器或者初始化器里，你把对任何类数据的引用直接保存在该对象本身里面，这样就没有什么东西需要查看它们了。访问器方法使用该对象来查找到数据的引用。

不要把定位类数据的复杂性放到每个方法里，只要让对象告诉方法数据在哪里就可以了。这个办法只有在类数据指示器方法被当作实例方法调用的时候才好用，因为类数据可能在一个你用包名字无法访问到的词法范围里。

不管你是怎么看待它，与包相关的类数据总是有点难用。继承一个类数据的指示器方法的确更清晰一些，你同样有效地继承了它能够访问的状态数据。参阅 `perltootc` 手册页获取管理类数据的更多更灵活的方法。

12.9 总结

除了其他东西以外，大概就这么多东西了。现在你只需要走出去买本关于面向对象的设计方法学的书，然后再花 `N` 个月的时间来学习它就行了。

Revision: r1.2 - 29 Aug 2005 - 14:01 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [Objects2](#)

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](#)