

第十二章 对象(上)

- ↓ 第十二章 对象(上)
 - ↓ 12.1 简单复习一下面向对象的语言
 - ↓ 12.2 Perl 的对象系统
 - ↓ 12.3 方法调用
 - ↓ 12.3.1 使用箭头操作符的方法调用
 - ↓ 12.3.2 使用间接对象的方法调用
 - ↓ 12.3.3 间接对象的句法障碍
 - ↓ 12.3.4 引用包的类
 - ↓ 12.4 构造对象
 - ↓ 12.4.1 可继承构造器
 - ↓ 12.4.2 初始器
 - ↓ 12.5 类继承

首先，你需要理解包和模块；请看第十章，包，和第十一章，模块。你还需要知道引用和数据结构；参阅第八章，引用和第九章，数据结构。同样，知道一些面向对象的编程方法（OOP）也是很有用的，所以下一节开始我们给就 OOL（面向对象的语言）上一小节课。

12.1 简单复习一下面向对象的语言

对象是一个数据结构，带有一些行为。我们通常把这些行为称为对象的直接动作，有时候可以把这些对象拟人化。比如，我们可能会说一个长方形“知道”如何在屏幕上显示自己，或者说它“知道”如何计算它自己的区域。

作为一个类的实例，对象从中获益，取得其行为。类定义方法：就是那些应用于类和它的事例的性质。如果需要区分上面两种情况，那么我们就把适用于某一特定对象的方法叫做实例方法，而把那些适用于整个类的方法叫做类方法。不过这样做只是为了方便——对于 Perl 而言，方法就是方法，只是由其第一个参数的类型来区分。

你可以把实例方法看作一个由特定对象执行的某种动作，比如说打印自己啦，拷贝自己啦，或者是更改自己的一个或者多个属性（“把剑命名为 **Anduril**”）。类方法可以在许多共同的对象上执行操作（“显示所有的剑”）或者提供其他不依赖于任何特定对象的操作（“从现在开始，一旦新铸了一把剑，就在数据库里注册它的主人”）。类里面那些生成对象实例的方法叫构造方法（“铸一把镶满宝石并且带有秘密题字的剑”）。这些通常是类方法（“给我造把新剑”），但也有可能是实例方法（“造一把和我的剑一样的剑”）。

一个类可以从父类中继承方法，父类也叫基类或者超级类。如果类是这样生成的，那么它叫派生类或者子类。（让我们把稀泥和得更糊一些：有些文章里把“基类”定义为“最上层”的超级类。我们这里不是这个意思。）继承令新类的行为和现存的类很象，但是又允许它修改或者增加它的父类没有的性质。如果你调用了方法，而在当前的类中没有找到这个方法，Perl 会自动询问父表，以找出定义。比如，剑类可能从一个通用的刀锋类中继承 **attack**（攻击）方法。父类自己也可以有父类，而 Perl 会根据需要也在那些类中进行搜索。刀锋类可能自己又是从更通用的武器类中继承了它的 **attack**（攻击）方法。

当一个对象调用 **attack**（攻击）方法时，产生的效果可能取决于该对象是一把剑还是一支箭。可能这两者之间并没有什么区别——如果剑和箭都是从通用的武器类中继承他们的攻击（**attack**）性质的话。但是如果两者的行为不同，那么方法分选机制总是会选择最适合该对象需要的那个 **attack** 方法。总是为某一类型的特定对象选择最合适的行为的特性叫做多样性。它是“不在意”的一个重要形式。

当你在实现一个类的时候，你必须关注那些对象的“内脏”，但是当你使用一个类的时候，你应该把

这个对象当作一个黑盒子。你不用知道内部有什么，你也不需要知道它是如何工作的，而且你和这个黑盒子的交互只用它的方式进行：通过类提供的方法。即使你知道这些方法对对象做些什么处理，你也应该抑制住自己干的冲动。就好像遥控电视一样，即使即使你知道电视里面是怎样运转的，如果没有特别充分的理由，也不应该在电视里上窜下跳地摆弄它。

Perl 可以让你在需要的时候从类的外部观察对象的内部。但是这样做就打破了它的封装的原则——所有对对象的访问都应该只通过方法。封装断开了接口的公开（对象应该如何使用）和实现（它实际上是如何运转的）之间的联系。Perl 除了一个未写明的设计者和用户之间的契约之外没有任何其他明确的接口。双方都依靠常识和公德来运做：用户依靠归档的接口，而设计者不会随便打破该接口。

Perl 并不强制你使用某种编程风格，并且它也不会有一些其他的面向对象的语言里的私有性的困惑。不过，Perl 的确会在自由上令人迷惑，并且，作为一个 Perl 程序员，你所拥有的一个自由就是根据自己的喜好选择或多或少的私有性。实际上，Perl 可以在它的类里面有比 C++ 更强的私有性。也就是说，Perl 不限制你做任何事情，而且实际上它也不限制你自己约束自己——假如你必须这么做的话。本章稍后的“私有方法”和“用做对象的闭合”节演示了你如何才能增大自律的剂量。

不过我们承认，对象的内涵比我们这里说的多得多，而且有很多方法找出面向对象的设计的更多内容。但是那些不是我们这里的目标。所以，我们接着走。

12.2 Perl 的对象系统

Perl 没有为定义对象，类或者方法提供任何特殊的语法。相反，它使用现有的构造来实现这三种概念。（注：现在有了一个软件重用的例子了！）

下面是一些简单的定义，可以让你安心些：

一个对象只不过是一个引用...恩，就是引用。

因为引用令独立的标量代表大量的数据，所以我们不应该对把引用用于所有对象 感到奇怪。从技术上来讲，对象不太适合用引用表示，实际上引用指向的是 引用物。不过这个区别很快就被 Perl 的程序员淡化了，而且因为我们觉得这是一个很好的转喻，如果这么用合适的话，我们将永远用下去（注：我们更喜欢语言 的活力，而不是数学的严密。不管你同意与否。）（译注：Larry Wall 是学语言的，不是学数学的，当然这么说。：)）

一个类只是一个包

一个包当作一个类——通过使用包的子过程来执行类的方法，以及通过使用包的 变量来保存类的全局数据。通常，使用一个模块来保存一个或者更多个类。

一个方法只是一个子过程

你只需要在你当作类来使用的包中声明子过程；它们就会被当作该类的方法来 使用。方法调用是调用子过程的一种新的方法，它传递额外的参数：用于调用方法 所使用的对象或包。

12.3 方法调用

如果你把面向对象的编程方法凝缩成一个精华的词，那就是抽象。你会发现这个词是所有那些 OO 的鼓吹者所传播的言辞背后的真正主题，那些言辞包括多型性啦，继承啦还有封装啦。我们相信这些有趣的字眼，但是我们还是会从实际的角度出发来理解它们——它们在方法调用中的作用是什么。方法是对象系统的核心，因为它们为实现所有魔术提供了抽象层。你不是直接访问对象里的一块数据区，而是调用一个实例方法，你不是直接调用某个包里面的子过程，而是调用一个类方法。通过在使用和实现之间放置这个间接层，程序设计人员仍然可以自由修改复杂的类的内部机制，因而要冒一点儿破坏使用它的程序的风险。

Perl 支持调用方法的两种不同的语意。一种是你已经在 Perl 别的地方看惯了的风格，而第二种是你可以在其他编程语言中看到的。不管使用哪种方法调用的形式，Perl 总是会构成这个方法的子过程传递一个额外的初始化参数。如果用一个类调用该方法，那个参数将会是类的名字。如果用一个对象调用方法，那个参数就是对象的引用。不管是什么，我们都叫它方法调用者。对于类方法而言，调用者是包的名字。对于一个实例方法，调用者是调用者是一个声明对象的引用。

换句话说，调用者就是调用方法的那个东西。有些 OO 文章把它叫做代理或演员。从文法上看，调用者既不是动作的对象也不是动作的承担者。它更象一个间接的对象，是代表动作执行后受益人的东西，就向在命令“给我铸把剑！”里的“我”一样。从语意上来看，你既可以把调用者看作动作的施动者，也可以把它看作动作的受动者——更象哪个取决于你的智力取向。我们可不打算告诉你怎样看待它们。

大多数方法是明确调用的，但是也可以隐含调用——由对象析构器，重载的操作符或者捆绑的变量触发的时候。准确地说，这些都不是正常的子过程调用，而是代表对象的 Perl 自动触发的方法调用。析构器在本章后面描述，重载在第十三章，重载，描述；而捆绑在第十四章，捆绑变量。

方法和普通子过程之间的一个区别是，它们的包在什么时候被解析——也就是说，Perl 什么时候决定应该执行该方法或者子过程的哪些代码。子过程的包是在你的程序开始运行之前，在编译的时候解析的。（注：更准确地说，子过程调用解析成一个特定的类型团，它是一个填充到编译好的操作码树上的引用。这个类型团的含义甚至在运行时也是可以协商的——这就是为什么 AUTOLOAD 可以为你自动装载一个子过程。不过，类型团的含义通常在编译的时候也被解析——由一个命名恰当的子过程定义解析。）相比之下，一个方法包直到实际调用的时候才解析。（编译的时候检查原型，这也是为什么编译时可以使用普通子过程而却不能使用方法的原因。）

方法包不能早些解析的原因是相当简单的：包是由调用的类决定的，而在方法实际被调用之前，调用者是谁并不清楚。OO 的核心是下面这样简单的逻辑：一旦得知调用者，则可以知道调用者的类，而一旦知道了类，就知道了类的继承关系，一旦知道了类的继承关系，那么就知道了实际调用的子过程了。

抽象的逻辑是要花代价的。因为方法比较迟才解析，所以 Perl 里面向对象的解决方法要比相应的非 OO 解决方法慢。而对我们稍后要介绍的几种更加神奇的技巧而言，它可能慢很多。不过，解决许多常见的问题的原因并不是做得快，而是做得聪明。那就是 OO 的闪光点。

12.3.1 使用箭头操作符的方法调用

我们说过有两种风格的方法调用。第一种调用方法的风格看起来象下面这样：

```
INVOCANT->METHOD (LIST)
INVOCANT->METHOD
```

这种方法通常被称做箭头调用（原因显而易见）。（请不要把->和=>混淆，“双管”箭头起神奇逗号的作用。）如果有任何参数，那么就需要使用圆括弧，而当 INVOCANT 是一个包的名称的时候，我们把那个被调用的 METHOD 看作类方法。实际上两者之间并没有区别，只不过和类的对象相比，包名字与类本身有着更明显的关联。还有一条你必须记住：就是对象同样也知道它们的类。我们会告诉你一些如何把对象和类名字关联起来的信息，但是你可以在不知道这些信息的情况下使用对象。

比如，使用类方法 `summon` 的构造一个类，然后在生成的对象上调用实例方法 `speak`，你可以这么说：

```
$mage = Wizard->summon("Gandalf");    # 类方法
$mage->speak("friend");                # 实例方法
```

`summon` 和 `speak` 方法都是由 `Wizard` 类定义的——或者是从一个它继承来的类定义的。不过你

用不着担心这个。用不着管 **Wizard** 的闲事。

因为箭头操作符是左关联的（参阅第三章，单目和双目操作符），你甚至可以把这两个语句合并成一条：

```
Wizard->summon("Gandalf")->speak("friend");
```

有时候你想调用一个方法而不想先知道它的名字。你可以使用方法调用的箭头形式，并且把方法名用一个简单的标量变量代替：

```
$method = "summon";
$mage = Wizard->$method("Gandalf");    # 调用Wizard->summon

$stravel = $companion eq "Shadowfax" ? "ride" : "walk";
$mage->$stravel("seven leagues");    # 调用 $mage->ride 或者 $mage->walk
```

虽然你间接地使用方法名调用了方法，这个用法并不会被 **use strict 'refs'** 禁止，因为所有方法调用实际上都是在它们被解析的时候以符号查找的形式进行的。

在我们的例子里，我们把一个子过程的名字存储在 **\$stravel** 里，不过你也可以存储一个子过程引用。但这样就忽略了方法查找算法，不过有时候你就是想这样处理。参阅“私有方法”节和在

“**UNIVERSAL**：最终的祖先类”节里面的 **can** 方法的讨论。要创建一个指向某方法在特定实例上的调用的引用，参阅第八章的“闭合”节。

12.3.2 使用间接对象的方法调用

第二种风格的方法调用看起来象这样：

```
METHOD INVOCANT (LIST)
METHOD INVOCANT LIST
METHOD INVOCANT
```

LIST 周围的圆括弧是可选的；如果忽略了圆括弧，就把方法当作一个列表操作符。因此你可以有下面这样的语句，它们用的都是这种风格的方法调用：

```
$mage = summon Wizard "gandalf";
$nemesis = summon Balrog home => "Moria", weapon => "whip";
move $nemesisis "bridge";
speak $mage "You cannot pass";
break $staff;          # 更安全的用法： break $staff();
```

你应该很熟悉列表操作符的语法；它是用于给 **print** 或者 **printf** 传递文件句柄的相同的风格：

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

它还和 **"Give Gollum the Precioussss"** 这样的英语句子类似，所以我们称他为间接对象形式。Perl 认为调用者位于间接对象槽位中。当你看到传递一个内建的函数，象 **system system** 或 **exec** 什么的到它的“间接对象槽位中”时，你的实际意思是在同一个位置提供这个额外的，没有逗号的参数（列表），这个位置和你用间接对象语法调用方法时的位置一样。

间接对象形式甚至允许你把 **INVOCANT** 声明为一个 **BLOCK**，该块计算出一个对象（引用）或者类（包）。这样你就可以用下面的方法把那两种调用组合成一条语句：

```
speak {summon Wizard "Gandalf" } "friend";
```

12.3.3 间接对象的句法障碍

一种语法总是比另外一种更易读。间接对象语法比较少混乱，但是容易导致几种语法含糊的情况。首先就是间接对象调用的 **LIST** 部分和其他列表操作符一样分析。因此，下面的圆括弧：

```
enchant $sword ($pips + 2) * $cost;
```

是假设括在所有参数的周围的，而不管先看到的是什。那么，它就等效于下面这样的：

```
($sword->enchant($pips + 2)) * $cost;
```

这样可不象你想要的：调用 **enchant** 时只给了 **\$pips + 2**，然后方法返回的值被 **\$cost** 乘。和其他列表操作符一样，你还必须仔细对待 **&&** 和 **||** 与 **and** 和 **or** 之间的优先级。

比如：

```
name $sword $oldname || "Glamdring";    # 在这不能用"or"
```

变成：

```
$sword->name($oldname || "Glamdring");
```

而：

```
speak $mage "friend" && enter();    # 这儿应该用"and"
```

变成奇怪的：

```
$mage->speak("friend" && enter());
```

这些可以通过把它们写成下面的等效形式消除错误：

```
enter() if $mage->speak("friend");
$mage->speak("friend") && enter();
speak $mage "friend" and enter();
```

第二种语法不适用于间接对象形式，因为它的 **INVOCANT** 局限于一个名字，一个未代换的标量值或者一个块。（注：仔细的读者应该还记得，这些语法项是和允许出现在趣味字符后面的列表是一样的，那些语法项标识一个变量的解引用——比如 **@ary**，**@\$aryref**，或者 **{ \$aryref }**。当分析器看到这些内容之一时，她就有自己的 **INVOCANT** 了，因此她开始查找她的 **LIST**。所以下面这些调用：

```
move $party->{LEADER};    # 可能错了！
move $riders[$i];        # 可能错了！
```

实际分析成这样：

```
$party->move->{LEADER};
$riders->move([i]);
```

但是你想要的可能是：

```
$party->{LEADER}->move;
$riders[$i]->move;
```

分析器只是为一个间接对象查找一个调用时稍稍向前看了一点点，甚至看的深度都不如为单目操作符那样深远。如果你使用第一种表示法是就不会发生这件怪事，因此你可能会选择箭头作为你的“武器”。

甚至英语在这方面也有类似的问题。看看下面的句子：“**Throw your cat out the window a toy mouse to play with.**”如果你分析这句话速度太快，你最后就会把猫仍出去，而不是耗子（除非你

意识到猫已经在窗户外边了)。类似 Perl, 英语也有两种不同的方法来表达这个代理: “**Throw your cat the mouse**” 和 “**Throw the mouse to your cat.**” 有时候长一点的形式比较清晰并且更自然, 但是有时候短的好。至少在 Perl 里, 我们要求你在任何编译为间接对象的周围放上花括号。

12.3.4 引用包的类

间接对象风格的方法调用最后还有一种可能的混淆, 那就是它可能完全不会被当作一个方法调用而分析, 因为当前包可能有一个和方法同名的子过程。当用一个类方法和一个文本包名字一起做调用者用的时候, 有一个方法可以解析这样的混淆, 而同时仍然保持间接对象的语法: 通过在包后面附加两个冒号引用类名。

```
$obj = method CLASS::;      # 强制为 "CLASS"->method
```

这个方法很重要, 因为经常看到下面的表示法:

```
$obj = new CLASS;          # 不会分析为方法
```

如果当前包有一个子过程叫 **new** 或者 **CLASS** 时, 将不能保证总是表现得正确。即使你很仔细地使用箭头形式而不是间接对象形式调用方法, 也有极小可能会有问题。虽然引入了额外标点的杂音, 但 **CLASS::** 表示法却能保证 Perl 正确分析你的方法调用。下面例子中前面两个不总是分析成一样的东西, 但后面两个可以:

```
$obj = new ElvenRing;      # 可以是 new("ElvenRing")
                           # 甚至是 new(ElvenRing())
$obj = ElvenRing->new;      # 可以是 ElvenRing()->new()

$obj = new ElvenRing::;    # 总是 "ElvenRing"->new()
$obj = ElvenRing::->new;   # 总是 "ElvenRing"->new()
```

包引用表示法可以用一些富有创造性的对齐写得更好看:

```
$obj = new ElvenRing::
      name => "Narya",
      owner => "Gandalf",
      domain => "fire",
      stone => "ruby";
```

当然, 当你看到双冒号的时候可能还是会说, “真难看!”, 所以我们还要告诉你, 你几乎总是可以只使用光光的类名字, 只要两件事为真。首先, 没有和类同名的子过程名。(如果你遵循命名传统: 过程名, 比如 **new** 以小写开头, 而类名字, 比如 **ElvenRing**² 以大写开头, 那么就永远不会有这个问题。)第二, 类是用下面的语句之一装载的:

```
use ElvenRing;
require ElvenRing;
```

这两种方法都令 Perl 意识到 **ElvenRing**² 是一个模块名字, 它强制任何在类名 **ElvenRing**² 前面的光板名字, 比如 **new**, 解释为一个方法调用, 即使你碰巧在你的当前包里定义了一个自己的 **new** 子过程, 也不会错误解释成子过程。我们通常不会在使用间接对象中碰到问题, 除非你在一个文件里填满多个类, 这个时候, Perl 就可能不知道一个特定的包名字就是一个类名字。而且那些把子过程的名字命名为类似 **ModuleNames**² 这样的人最终也会陷入痛苦。

12.4 构造对象

所有对象都是引用, 但不是所有引用都是对象。一个引用不会作为对象运转, 除非引用它的东西有特

殊标记告诉 **Perl** 它属于哪个包。把一个引用和一个包名字标记起来（因此也和包中的类标记起来了，因为一个类就是一个包）的动作被称作赐福（**blessing**），你可以把赐福（**bless**）看作把一个引用转换成一个对象，尽管更准确地说是它把该引用转换成一个对象引用。

bless 函数接收一个或者两个参数。第一个参数是一个引用，而第二个是要把引用赐福（**bless**）成的包。如果忽略第二个参数，则使用当前包。

```
$obj = { };          # 把引用放到一个匿名散列
bless($obj);         # Bless 散列到当前包
bless($obj, "Criticter");    # Bless 散列到类 Critter。
```

这里我们使用了一个指向匿名散列的引用，也是人们通常拿来做他们的对象的数据结构的东西。毕竟，散列极为灵活。不过请允许我们提醒你的是，你可以赐福（**bless**）一个引用为任何你在 **Perl** 里可以用作引用的东西，包括标量，数组，子过程和类型团。你甚至可以把一个引用赐福（**bless**）成一个包的符号表散列——只要你有充分的理由。（甚至没理由都行。）**Perl** 里的面向对象的特性与数据结构完全不同。

一旦赐福（**bless**）了指示物，对它的引用调用内建的 **ref** 函数会返回赐福了的类名字，而不是内建的类型，比如 **HASH**。如果你需要内建的类型，使用来自 **attributes** 模块的 **reftype**。参阅第三十一章，实用模块，里的 **use attributes**。

这就是如何制作对象。只需要使用某事的引用，通过把他赐福（**bless**）到一个包里给他赋一个类，仅此而已。如果你在设计一个最小的类，所有要做的事情就是这个。如果你在使用一个类，你要做的甚至更少，因为类的作者会把 **bless** 隐藏在一个叫构造器的方法里，它创建和返回类的实例。因为 **bless** 返回其第一个参数，一个典型的构造器可以是：

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

或者略微更明确地拼写：

```
package Critter;
sub spawn {
    my $self = {};          # 指向一个空的匿名散列
    bless $self, "Criticter";    # 把那个散列作成 Critter 对象
    return $self;           # 返回新生成的 Critter
}
```

有了那个定义，下面就是我们如何创建一个 **Criticter** 对象了：

```
$pet = Critter->spawn;
```

12.4.1 可继承构造器

和所有方法一样，构造器只是一个子过程，但是我们不把它看作一个子过程。在这个例子里，我们总是把它当作一个方法来调用——一个类方法，因为调用者是一个包名字。方法调用和普通的子过程调用有两个区别。首先，它们获取我们前面讨论过的额外的参数。其次，他们遵守继承的规则，允许一个类使用另外一个类的方法。

我们将在下一章更严格地描述继承下层的机制，而现在，通过几个简单的例子，你就应该可以理解他们的效果，因此可以帮助你设计构造器。比如，假设我们有一个 **Sppider** 类从 **Spider** 类继承了方法。特别是，假设 **Spider** 类没有自己的 **spawn** 方法。则有下面对应的现象：

方法调用	结果子过程调用
<code>Criticter->spawn()</code>	<code>Citter::spawn("Criticter")</code>

```
Spider->spawn()Critter::spawn("Spider")
```

两种情况里调用的子过程都是一样的，但是参数不一样。请注意我们上面的 `spawn` 构造器完全忽略了它的参数，这就意味着我们的 `Spider` 对象被错误地赐福（`bless`）成了 `Critter` 类。一个更好的构造器将提供包名字（以第一个参数传递进来）给 `bless`：

```
sub spawn {
    my $class = shift;      # 存储包名字
    my $self = { };
    bless( $self, $class);  # 把赐福该包为引用
    return $self;
}
```

现在你可以为两种情况都使用同一个子过程：

```
$vermin = Critter->spawn;
$shelob = Spider->spawn;
```

并且每个对象都将是正确的类。甚至是间接运转的，就象：

```
$type = "Spider";
$shelob = $type->spawn;      # 和 "Spider"->spawn 一样
```

这些仍然是类方法，不是实例方法，因为它的调用者持有的是字符串而不是一个引用。

如果 `$type` 是一个对象而不是一个类名字，前一个构造器的定义将不会运行，因为 `bless` 需要一个类名字。但是对许多类而言，只有拿一个现有的对象当模板去创建另外一个对象的时候它才有意义。在这些情况下，你可以设计你的构造器，这样他们就可以与对象或者类名字一起运转了：

```
sub spawn {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;  # 对象或者类名字
    my $self = { };
    bless ($self, $class);
    return $self;
}
```

12.4.2 初始器

大多数对象维护的信息是由对象的方法间接操作的。到目前为止我们的所有构造器都创建了空散列，但是我们没有理由让它们这么空着。比如，我们可以让构造器接受额外的参数，并且把它们当作键字/数值对。有关 OO 的文章常把这样的数据称为“所有”，“属性”，“访问者”，“成员数据”，“实例数据”或者“实例变量”等。本章稍后的“实例变量”节详细地讨论这些属性。

假设一个 `Horse` 类有一些实例属性，比如 `"name"` 和 `"color"`：

```
$steed = Horse->new(name => "shadowfax", color => "white");
```

如果该对象是用散列引用实现的，那么一旦调用者被从参数列表里删除，那么键字/数值对就可以直接代换进散列：

```
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = { @_ };      # 剩下的参数变成属性
    bless($self, $class);   # 给予对象性质
}
```



```

        return $self;
    }

```

这回我们用一个名字叫 **new** 的方法做该类的构造器，这样就可以把那些 C++ 程序员哄得相信这些都是正常的。不过 Perl 可不认为“new”有任何特殊的地方；你可以把你的构造器命名为任意的东西。任何碰巧创建和返回一个对象的方法都是实际上的构造器。通常，我们建议你把你的构造器命名为任何在你解决的问题的环境中有意义的东西。比如，在 Tk 模块中的构造器命名为它们创建的窗口构件。在 DBI 模块里，一个叫 **connect** 的构造器返回一个数据库句柄对象，而另外一个叫 **prepare** 的构造器是当作一个实例方法调用的，并且返回一个语句句柄对象。不过如果没有很好的适合环境的构造器名字，那么 **new** 也不算是一个太坏的选择。而且，随便挑一个名字，这样强制人们在使用构造器之前去读接口文档（也就是类的文档）也不是太坏的事情。

更灵活一些，你可以用缺省键字/数值对设置你的构造器，这些参数可以由用户在使用的时候通过提供参数而覆盖掉：

```

sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = {
        color => "bay",
        legs => 4,
        owner => undef,
        @_,          # 覆盖以前的属性
    };

    return bless $self, $class;
}

$ed      = Horse->new;                # 四腿湾马
$stallion = Horse->new(color => "black"); # 四腿黑马

```

当把这个 **Horse** 构造器当作实例方法使用的时候，它忽略它的调用者现有的属性。你可以设计第二个构造器，把它当作实例方法来调用，如果你设计得合理，那你就可以使用来自调用对象的数值作为新生成的对象的缺省值：

```

$steed = Horse->new(color => "dun");
$foal = $steed->clone(owner => "EquuGen Guild, Ltd.");

sub clone {
    my $model = shift;
    my $self = $model->new(%$model, @_);
    return $self;          # 前面被 ->new 赐福过了
}

```

（你可以把这个功能直接放进 **new** 里，但是这样的话名字就不是那么适合这个函数了。）

请注意我们即使是在 **clone** 构造器里，我们也没有硬编码 **Horse** 类的名字。我们让最初的那个对象调用它自己的 **new** 方法，不管是什么。如果我们把它写成 **Horse->new** 而不是 **\$model->new**，那么该类不能帮助实现 **Zebra**（斑马）或 **Unicorn**（独角兽）类。你应该不会想克隆一匹飞马但是却突然发现你得到是一匹颜色不同的马。

不过，有时候你碰到的是相反的问题：你不是想在不同的类里共享一个构造器，而是想多个构造器共享一个类对象。当一个构造器想调用一个基类的构造器作为构造工作的一部分的时候就会出现这种问题。Perl 不会帮你做继承构造。也就是说，Perl 不会为任何基类或者任何其他所需要的类自动调用构造器（或者析构器），所以你的构造器将不得不自己做这些事情然后增加衍生的类所需要的附加的任

何属性。因此情况不象 `clone` 过程那样，你不能把一个现有的对象拷贝到新对象里，而是先调用你的基类的构造器，然后把新的基类对象变形为新的衍生对象。

12.5 类继承

对 Perl 的对象系统剩下的内容而言，从一个类继承另外一个类并不需要给这门语言增加特殊的语法。当你调用一个方法的时候，如果 Perl 在调用者的包里找不到这个子过程，那么它就检查 `@ISA` 数组（注：发音为 "is a"，象 "A horse is a critter." 里哪样）。Perl 是这样实现继承的：一个包的 `@ISA` 数组里的每个元素都保存另外一个包的名字，当缺失方法的时候就搜索这些包。比如，下面的代码把 `Horse` 类变成 `Critter` 类的子类。（我们用 `our` 声明 `@ISA`，因为它必须是一个打包的变量，而不是用 `my` 声明的词。）

```
package Horse;
our @ISA = "Critter";
```

你现在应该可以在原先 `Critter` 使用的任何地方使用 `Horse` 类或者对象了。如果你的新类通过了这样的空子类测试，那么你就可以认为 `Critter` 是一个正确的基类，可以用于继承。

假设你在 `$steed` 里有一个 `Horse` 对象，并且在他上面调用了 `move`：

```
$steed->move(10);
```

因为 `$steed` 是一个 `Horse`，Perl 对该方法的第一个选择是 `Horse::move` 子过程。如果没有，Perl 先询问 `@Horse::ISA` 的第一个元素，而不是生成一个运行时错误，这样将导致查询到 `Critter` 包里，并找到 `Critter::move`。如果也没有找到这个子过程，而且 `Critter` 有自己的 `@Critter::ISA` 数组，那么继续查询那里面的父类，看看有没有一个 `move` 方法，如此类推直到上升到继承级别里面一个没有 `@ISA` 的包。

我们刚刚描述的情况是单继承的情况，这时每个类只有一个父类。这样的继承类似一个相关包的链表。Perl 还支持多继承；只不过是向该类的 `@ISA` 里增加更多的包。这种继承的运做更象一个树状结构，因为每个包可以有多个的直接父类。很多人认为这样更带劲。

当你调用了调用者的一个类型为 `classname` 的方法 `methname`，Perl 将尝试六种不同的方法来找出所用的子过程（译注：又是孔乙己？）：

1. 首先，Perl 在调用者自己的包里查找一个叫 `classname::methname` 的子过程。如果失败，则进入继承，并且进入步骤 2。
2. 第二步，Perl 通过检查 `@classname::ISA` 里列出的所有父包，检查从基类继承过来的方法，看看有没有 `parent::methname` 子过程。这种搜索是从左向右，递归的，由浅入深进行的。递归保证祖父类，曾祖父类，太祖父类等等类都进入搜索。
3. 如果仍然失败，Perl 就搜索一个叫 `UNIVERSAL::methname` 的子过程。
4. 这时，Perl 放弃 `methname` 然后开始查找 `AUTOLOAD`。首先，它检查叫做 `classmane::AUTOLOAD` 的子过程。
5. 如果上面的失败，Perl 则搜索所有在 `@classname::ISA` 里列出的 `parent` 包，寻找任何 `parent::AUTOLOAD` 子过程。这样的搜索仍然是从左向右，递归的，由浅入深进行的。
6. 最后，Perl 寻找一个叫 `UNIVERSAL::AUTOLOAD` 的子过程。

Perl 会在找到的第一个子过程处停止并调用该子过程。如果没有找到子过程，则产生一个例外，也是你经常看到的：

```
Can't locate object method "methname" via package "classnaem"
```

如果你给你的 C 编译器提供了 `-DDEBUGGING` 选项，做了一个调试版本的 Perl，那么如果你给

Perl 一个 `-Do` 开关，你就能看到它一边解析方法调用一边走过这些步骤。

我们将随着我们的继续介绍更详细地讨论继承机制。

Revision: r1.2 - 27 Aug 2005 - 14:43 - [TingYu](#)

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

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](#)