1 NAME

perlootut - perl 教程之面向对象编程

2 描述

此文档提供了一个使用 perl 进行面向对象编程的介绍。开篇是一个简短的关于面向 对象设计概念的概述，之后介绍了CPAN上基于 perl 的 各种 OO 系统。

默认情况下，perl 内置的 OO 系统非常小巧，这使你必须自己做大部分工作。这种 小巧在1994年那个时代是合理的，但自 perl 5.0发布以来，许多通用的模式在 perl OO 系统里面显现。幸运的是，perl 的灵活性允许 perl OO 的生态系统演化和 繁荣。

如果你想知道 perl OO 底层是如何工作的，perlobj 提供了详细的信息。

此文档假设你已经掌握了基本的 perl 语法，变量类型，操作符和函数调用。如果你 还不理解这些概念，请先阅读 perlintro。同时你还应该阅读 perlsyn， perlop，perlsub。

3 面向对象基础

大部分面向对象编程系统都有一些通用的概念。你可能已经听说过了一些像“类”，“ 对象”，“方法”和“属性”的术语。理解这些概念，使你能够更加容易的阅读和编写面 向对象代码。即使你已经熟悉了这些，你仍然可以阅读一下这一节，因为这里用 perl 的术语解释了这些概念。

perl 的 OO 系统基于类(class-based)。基于类的 OO 是相当通用的，java，C++， C#，Python，Ruby 和很多其他语言都是如此。当然也有其他类型的面向对象。 Javascript 就是使用另一种模式的流行语言，其 OO 系统是基于原型的 (prototype-based)。

对象

对象是聚合了数据和操作数据的函数的数据结构。对象的数据叫做属性，它的 函数叫做方法。一个对象可以想象成为一个名词（一个人，一个网络服务，一台 计算机）。

对象代表了一个单一但不连续的物体。比如，它可能代表一个文件。文件对象的属性可能包括路径，内容，最后修改时间。如果我们创建一个代表 /etc/hostname 文 件的对象，这个文件存放在名为“foo.example.com”的机器上。这个对象的路径就应 该是“/etc/hostname”，内容是“foo\n”，最后修改时间是自 epoch 后1304974868秒 。

和文件相关的方法可以有 `rename()` 和 `write()`。

perl 的大部分对象都是哈希，如果你使用我们推荐的 OO 系统，则你完全不必关心 这一点。实际上，最好将对象看做一个不透明的内部数据结构。

类

类定义了一个类别的对象的行为。类是一个类别的名字(比如 "File")，类还定义 了这个类别中的对象的行为。

所有的对象都属于某个特定类。比如，我们的 /etc/hostname 对象属于 `File` 类。当我们想创建一个特定对象时，我们以它的类开始，创建或者实例化一个 对象。对象经常被称为一个类的实例。

在 perl 中，任何包都可以是一个类。一个包是否为类的区别在于这个包是如何被使 用的。以下是我们的 `File` 类的"类定义"。

```
package File;
```

对于 perl 来说，没有一个特定的关键字来创建对象，但大部分 CPAN 上的 OO 模块 都使用一个名为 new() 的方法。

```
my $hostname = File->new(
    path         => '/etc/hostname',
    content      => "foo\n",
    last_mod_time => 1304974868,
);
```

(别担心那个->操作符，接下来会解释)

Blessing

如前所述，大多 perl 的对象都是哈希，但它也可以是任意 perl 数据类型(标量， 数组等)。把一个普通的 perl 数据结构转换成对象的方式是使用 bless 函数对 它进行 Blessing。

尽管我们强烈建议你不要从最底层开始创建你的对象，但你也应该知道 bless 这 个术语。一个 blessed 的数据结构是一个对象。我们有时候会说一个对象被“ blessed into a class”。

当一个变量被 bless 后，Scalar::Util 模块中的 blessed 函数可以告诉我 们它的类名。当其参数为对象时，这个函数返回对象的类，否则返回false。

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

print blessed($hash);     # undef
print blessed($hostname); # File
```

构造器

构造器用来创建新对象。不像其他语言中提供了构造器语法一样，在 perl 中， 一个类的构造器只是一个普通的方法。大部分 perl 的类使用 new 作为构造器的 名字。

```
my $file = File->new(...);
```

方法

你已经知道了方法就是操作对象的子程序。你也可以将方法看做对象能做的事 情。如果对象是名词，那么方法就是它的动作(保存，打印，打开)。

在 perl 里面，方法是定义类的包里面的子程序。方法的第一个参数总是对象。

```
sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}

$file->print_info;
# The file is at /etc/hostname
```

方法的特殊之处在于它是“怎样被调用的”。箭头操作符->告诉 perl 我们在 调用一个方法。

Invocant 是一个有意思的名字，它代表箭头左边的部分。Invocant 可以是类名或者 一个对象。我们当然也可以给方法传递其他参数。

```
sub print_info {
    my $self  = shift;
    my $prefix = shift // "This file is at ";
```

```
    print $prefix, '', '', $self->path, "\n";
}

$file->print_info("The file is located at ");
# The file is located at /etc/hostname
```

属性

所有的类都可以定义它的属性。当我们初始化一个对象时，我们可以给属性赋值。比如，每个对象有一个路径。属性(attributes)有时也叫做 properties

Perl 没有用来定义属性的语法。在底层，属性就是类所对应哈希的键，不过你不需要关注这点。

我们推荐你只通过存取器来访问属性。存取器是用来获得或设置属性的值的方法。我们在`print_info()` 里面已经看到过了: `$self->path`

你可能也见过 getter 和 setter 这样的术语。这是两种不同的存取器。Getter 用来获得属性的值，而 setter 用来设置值。Setter 的另一个叫法是 mutator

属性可以定义为只读或者可读写的。只读的属性只能在创建对象的时候被赋值，而可读写的属性在任何时候都能够更改。

属性的值也可以是另外一个对象。比如，`File` 类可以返回一个 DateTime 对象代表其最后修改时间，而不是返回数字。

类也可以没有任何公开的可设置的属性。不是每一个类都有属性和方法。

多态

多态是描述来自不同类的对象公用一个接口的说法。比如，`File` 和 `WebPage` 类都可以有 `print_content()` 方法。对于不同的类，调用这个方法的输出可能不同，但它们有通用的接口。

尽管在很多方面，这两个类都不尽相同，但对于 `print_content` 方法来说，他们是一样的。这意味着我们可以在任一类的对象上调用这个 `print_content` 方法，而我们都不需要知道对象属于哪个类!

多态是面向对象设计的关键概念之一。

继承

继承可以让你创建已经存在的类的一个特殊版本。继承创建的新类能够复用原来类的方法和属性。

比如，我们可以创建一个 `File::MP3` 类，这个类从 `File` 继承而来。`File::MP3` 是 `File` 的一个更加具体的版本。所有的 mp3 都是文件，但不是所有的文件都是 mp3。

我们经常把继承关系称为父-子或者 `/` 关系。有时候我们说子类是一个父类（比如 `File::MP3` 类是一个 `File` 类）。

`File` 是 `File::MP3` 的超类，而 `File::MP3` 是 `FILE` 的子类。

```
package File::MP3;

use parent 'File';
```

parent 模块是 perl 让你定义继承关系的诸多方法之一。

Perl 允许多重继承，这意味着一个类可以继承自多个父类。尽管这是可能的，但我们强烈反对这样做。通常，你可以使用角色(roles)来做到用多重继承能做到的所有事情，而且做法还更加清晰。

注意对一个给定类，给它定义多个子类没有什么关系，这是通用且安全的做法。比如，我们可以第一 `File::MP3::FixedBitrate` 和 `File::MP3::VariableBitrate` 类，用来区分不同类别的 mp3 文件。

方法重载和方法解析

继承允许两个类共享代码。默认情况下，父类的所有方法都能在子类中使用。子类 可以提供它自己的实现来重载父类的方法。比如，如果我们有一个 File::MP3 对象，它有从 File 而来的 print_info() 方法。

```
my $cage = File::MP3->new(
    path         => 'mp3s/My-Body-Is-a-Cage.mp3',
    content      => $mp3_data,
    last_mod_time => 1304974868,
    title        => 'My Body Is a Cage',
);

$cage->print_info;
# The file is at mp3s/My-Body-Is-a-Cage.mp3
```

如果我们希望能够打印 mp3 的名字，那么可以重载这个方法。

```
package File::MP3;

use parent 'File';

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
    print "Its title is ", $self->title, "\n";
}

$cage->print_info;
# The file is at mp3s/My-Body-Is-a-Cage.mp3
# Its title is My Body Is a Cage
```

决定使用哪个方法的过程叫做方法解析。 perl 在这里所做的是：先查看对象的 所属类（这里是 File::MP3），如果类定义了这个方法，那么调用这个类的版本 。否者，perl 查看每个父类。对于 File::MP3，它的父类只有 File。如果 File::MP3 没有定义这个方法，而 File 定义了，那么 perl 就会调用 File 里面的定义。

如果 File 继承自 DataSource，而 DataSource 继承自 Thing，那么 有必要的话，perl 会“源着继承链自下而上”的查找方法定义。

可以明确的在子类中调用父类的方法。

```
package File::MP3;

use parent 'File';

sub print_info {
    my $self = shift;

    $self->SUPER::print_info();
    print "Its title is ", $self->title, "\n";
}
```

SUPER:: 部分告诉 perl ，在 File::MP3 继承链中查找 print_info() 方 法。当在父类中找到了这个方法，则调用之。

之前我们提到了多重继承。它的主要问题就是使方法解析变得非常复杂。更多细节 可以查阅 perl obj。

封装

封装的理念是每个对象都是不透明的。当其他开发人员使用你的类时，他们 不需要知道这个类是如何实现的，而只需要知道它做什么就可以了。

封装在许多方面都是非常重要的。首先，它允许你将公用接口和内部实现分开。这 意味着你能在修改实现的同时不破坏接口。

其次，如果很好的封装了类，继承也将变得更加简单。理想情况是，子类使用和父 类相同的接口来访问对象数据。实际上，有时候继承会破坏封装，但好的接口会减 少这种破坏。

之前我们提到大部分 perl 的对象都是用哈希实现的。封装的原则告诉我们不应依赖 于此，而应该使用存取器来获得哈希中的数据。我们接下来推荐的对象系统都能自 动生成存取器。如果你使用它们之中的任意一个，你永远不需要使用哈希来访问对 象。

组合

在面向对象的代码中，我们经常能看到一个对象指向另外一个对象，这叫做对象 组合，或者有一个关系。

之前我们提到 `File` 类的 `last_mod_time` 方法能够返回一个 `DateTime` 对 象。这就是一个对象组合的例子。我们也可以更进一步，让 `path` 和 `content` 存取器也返回对象。那么 `File` 就和许多其他的对象进行了组合。

角色 Roles

角色是一个类“做的事情”，而不是类是什么。对 perl 来说，角色是一个相对较 新，同时也很流行的概念。角色应用于类。有时我们说类消耗角色。

角色是继承之外，提供多态性的另一种选择。假设我们有两个类，`Radio` 和 `Computer`。这两种物体都有开关键，我们想在类定义中为之建立模型。

我们可以让两个类都继承自同一父类，比如 `Machine`，但不是所有的机器都有开 关键。我们也可以建立一个叫 `HasOnOffSwitch` 的父类，但这也太文绉绉了。 Radio 和 Computers 不是这种父类的特殊体。这个父类定义得有点滑稽。

这就是角色所适合的地方。此时建立一个 `HasOnOffSwitch` 角色并将之应用于类 显得更加合理。这个角色可以定义如 `turn_on()` 和 `turn_off()` 之类的接口 。

Perl 并没有关于角色的内置接口。过去，人们都只能别无选择的使用多重继承。而 现在， CPAN 上有许多模块可以让你使用角色。

什么时候使用 OO

面向对象并不是所有问题的最好解决方案。在Perl 最佳实践(copyright 2004, Published by O'Reilly Media, Inc.)中，Damian Conway 提供了一个决定OO 是否 是解决你的问题的最好方案的列表。

- 在设计的系统非常庞大，或者可能变得庞大。
- 数据可以聚合成一个明显的结构，特别是每个聚合中有大量数据的时候。
- 各种类型的数据会形成自然的层次，让继承和多态的使用更为容易。
- 你有一些数据，许多不同的运算都会应用在那些数据上面。
- 你必须对一些相关类型的数据做一些相同的通用运算，但是会根据运算所应用于特 定的数据类型而有些细微的差异。

- 你可能日后要增加新的数据类型。
- 数据之间的交互最好以运算符表示
- 系统中个别组件的时间可能随时间而改变
- 系统设计已经是面向对象的
- 有很多其他程序员会使用你的代码模块

4 perl OO 系统

就像前面所说的一样，perl 内置的 OO 系统非常小巧，也相当灵活。这些年来，在 perl 的内置系统上面，人们开发了许多高级系统，用以提供更多的特性和便利。

我们强烈推荐你使用这些系统中的一个。它们之中即使是最轻巧的实现都能简化许 多重复工作。没有任何理由从零开始用 perl 构建你的类。

如果你对这些系统的内部实现感兴趣，请参阅 perlobj

Moose

Moose自称为“perl 5的后现代对象系统”。不要被吓到，“后现代”一词出现 在这里，只是对 Larry 将 perl 称作“第一种后现代计算机语言”的呼应。

Moose提供了一个完全而现代化的 OO 系统。对它影响最大的是 Common Lisp 的 面向对象系统，同时它也借鉴了 Smalltalk 和许多其他编程语言的理念。Moose 由 Stevan Little 创建，并从他对 perl 6 OO 的设计工作中获益良多。

这是我们使用 Moose 的 File 类

```
package File;
use Moose;

has path         => ( is => 'ro' );
has content      => ( is => 'ro' );
has last_mod_time => ( is => 'ro' );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

Moose 提供了许多特性：

- Declarative sugar
- 声明式的语法糖

  Moose提供了一层声明式的语法糖，用来定义类。这些语法糖只是一系列导出的函数，它们可以使定义你的类的工作变得更加简单和更具可移植性。你可以描述你的类是什么，而不用去告诉 perl 怎样实现它。

  has() 子函数定义了一个属性，Moose 会自动为其创建存取器。它还帮你创建了 new() 方法。这个构造器知道你所定义的属性，所以你能够在创建一个 File 对象时给它们赋值。

- 内置角色

  Moose 让你像定义类一样定义角色

```
package HasOnOfSwitch;
use Moose::Role;

has is_on => (
    is  => 'rw',
    isa => 'Bool',
);

sub turn_on {
    my $self = shift;
    $self->is_on(1);
}

sub turn_off {
    my $self = shift;
    $self->is_on(0);
}
```

- 一个小型的类型系统

  在上面的例子中，创建 is_on 属性的时候，我们给 has() 传递了参数 isa => 'Bool'。这是在告诉 Moose，这个属性必须是一个布尔值。如果我们 给它设置非法值，代码将会抛出错误。

- 完全的内省和控制机制

  Perl 的内置内省特性及其微小。Moose 在它之上，为你的类建立了一个完全的内 省层，使得你可以问诸如“File 类实现了哪些方法？”之类的问题。它还可以让你 随心所欲的修改你的类。

- 自宿主(Self-hosted)和可扩展性

  Moose 使用自己的内省 API 描述自己。除了这是一个很酷的技巧外，它还意味着 你可以用 Moose 来扩展 Moose。

- 丰富的生态系统

  在 CPAN 上的MooseX 命名空间下，有一个关于 Moose 扩展的丰富的生态系统。此外，CPAN 上很多模块已 经在使用 Moose，提供了许多可供学习的例子。

- 更多特性

  Moose 是一个非常强大的工具，我们不能在这里介绍它全部的特性。我们鼓励你从Moose::Manual|http://search.cpan.org/ perl doc?Moose::Manual开始，通 过阅读 Moose 的文档来学习它，

当然，Moose 也并不完美。

Moose 会使你的代码载入变慢。Moose 不是一个小型系统，当你定义你的类时，它做了大量代码生成工作。这意味着你 的代码在运行期间能尽可能的快，但在你的模块第一次被载入时，必须付出一些时间 代价。

这个载入时间在启动速度重要的时候会成为一个问题。比如命令行脚本，或者必须 在每次执行都被载入的纯 CGI 脚本。

不过先别慌，许多人的确在写命令行工具和其他对启动时间敏感的代码中使用了Moose。我们鼓励你先试用一下它，然后再考虑启动速度的问题。

同时 Moose 还有很多其他模块的依赖关系，其中大部分都是小型的独立模块，还 有一部分是为 Moose 而写的。Moose 本身，以及它的一些依赖模块需要用到编译器。如果你想将你的软件安装在一个没有 编译器的系统上面，或者有任何依赖关系都是问题，那么 Moose 可能不适合 你。

Mouse

如果你使用了 Moose 之后，发现这些问题之一阻止你继续使用它，我们建议你接 下来考虑一下 Mouse。Mouse 用一个更简单的包，实现了 Moose 功能的一 个子集。对于它所实现的所有特性，其接口和 Moose 是完全相同的，这意味 着你可以非常容易的从 Moose 迁移到Mouse。

Mouse 没有实现大部分 Moose 的内省接口，所以在载入你的模块时，它的速 度更快。同时，它的所有依赖模块都由 perl 核心提供，且不需要编译器就能运行。 如果你有编译器，那么Mouse 会用它来编译一些代码，从而加快运行速度。

最后，它还包含了一个 Mouse::Tiny 模块，这个模块将大部分 Mouse 特性打 包到了一个文件里面。把这个文件拷贝到你应用的库目录，你可以轻易打包你的应用 。

Moose 的作者希望通过充分改进 Moose，使 Mouse 终有一天能够过时，但 现在 Mouse 仍然提供了除 Moose 之外的一个有价值的选择。

Class::Accessor

Class::Accessor 完全和 Moose 相反。它提供了很少的特性，也不是自宿主 的。

但它非常简单，完全由 perl 实现，没有非核心依赖，同时还提供了“类 Moose”的 借口。

尽管做的不多，它仍然比你从头开始写你的类好。

这是用 Class::Accessor 实现的 File 类：

```
package File;
use Class::Accessor 'antlers';

has path          => ( is => 'ro' );
has content       => ( is => 'ro' );
has last_mod_time => ( is => 'ro' );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

antlers 导入符告诉 Class::Accessor 我们想用类 Moose 的语法来定义 属性。你唯一可以传给 has 的参数是 is。当你使用 Class::Accessor 时 ，我推荐你使用类 Moose 的语法，这意味着当你将来决定使用 Moose 时，你可 以更平滑的升级。

像 Moose 一样，Class::Accessor 为类自动创建构造器和存取器。

Object::Tiny

最后介绍 Object::Tiny。这个模块就如它的名字，完全没有依赖关系且借口及其 简单。但我们仍然认为比起自己写你的 OO 代码，使用这个模块更加容易。

再一次，这是我们的 File 类：

```
package File;
use Object::Tiny qw( path content last_mod_time );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

这就足够了!

使用 Object::Tiny，所有的存取器都是只读的。它为你类生成构造器，也为你定 义的属性生成存取器。

Role::Tiny

我们前面提过，角色提供了继承之外的其他选择，但 perl 没有内置的角色支持。如 果你选择使用 Moose，它提供了一个成熟的角色实现。如果你使用其他我们推荐的OO 模块，你仍然可以通过 Role::Tiny 使用角色。

Role::Tiny 提供了一些Moose 角色系统的特性，但它更加小巧。值得注意的是，它不支持任何属性定义，所以你必须手动定义它们。但它任然十分有用，并且和 Class::Accessor 或 Object::Tiny 在一起工作得很好。

OO 系统总结

这是关于我们提到的模块的一个简单回顾

- Moose

  Moose 是最好的选择。它提供了许多特性，强大的生态系统，大量的用户基础。 我们也简单的提到了 Mouse，它是 Moose 的简化版，也是当 Moose 不适合于你 的应用的时候的一个合理选择。

- Class::Accessor

  Class::Accessor 做得比 Moose 少许多，如果你发现 Moose 太重量级了，它 是一个很好的选择。这个模块已经存在了相当长时间，而且也通过了很好的测试。它 还提供了一个轻量的 Moose 兼容接口，这使得从Class::Accessor 迁移到 Moose 非常简单。

- Object::Tiny

  Object::Tiny 是最后的选择。它没有任何依赖，不需要学习任何语法。当你需 要一个超级简单的环境，并且不需要考虑细节的将一些东西捏合起来时，它是一个 不错的选择。

- Role::Tiny

  如果你发现自己在考虑多重继承的时候，可以组合使用 Role::Tiny 和 Class::Accessor 或 Object::Tiny。如果你使用 Moose，它提供了自己的 角色实现。

其他 OO 系统

除了这里提到的 OO 系统外，CPAN 上还有成堆的相关模块。当你在使用别人的代码时 ，很有可能就会碰到它们。

其次，还有许多代码使用 perl 的内置 OO 特性“手动”的实现它自己的 OO。如果你 需要维护这种代码，你必须阅读 perlobj，理解 perl 的内置 OO 是如何工作的 。

5 总结

就像我们前面所说，perl 轻量的 OO 系统使得 CPAN 上面存在大量的 OO 模块。尽管你 仍然可以选择自己手动编写你的类，但在现在，2011，你没有任何理由要去那样做。

对于小项目，Object::Tiny 和 Class::Accessor 都提供了一个轻量的对象系 统，能够解决你的基本问题。

对于更大的项目，Moose 提供了丰富的特性，使你能够关注在业务逻辑层面上。

我们鼓励你使用并测试这些模块，Moose，Class::Accessor， Object::Tiny，然后再决定哪个适合于你。

6 TRANSLATORS

Woosley Xu woosley.xu@gmail.com