

鉴于同时掌控所有程序细节的难度，编写大型程序要比编写小程序更需要规范。抽象（找到并利用相似性和近似之处）和封装（组织特定细节并按其所属来访问）是掌控细节之必需。

函数有益于此，但函数本身对庞大的程序来说远远不够。面向对象是一种流行的技术，它将函数组织为可执行相关行为的对象。

Perl 5 的默认对象系统极其简单。它相当灵活——你可以在它之上创建几乎所有其他对象系统——但它几乎不对完成最基本任务提供任何简易的辅助。

Moose

Moose 是一个专为 Perl 5 提供的更为完整的对象系统。它基于现有的 Perl 5 对象系统创建，并提供了更为简单的默认操作、更好的集成、以及来自其它语言——包括 Smalltalk、Common Lisp 和 Perl

6——的更为先进的特性。如果你只想编写一些简单的程序、维护陈年代码或者 Moose 不适用时，Perl 5 的对象系统仍然值得一学，但 Moose 是目前为止在现代 Perl 5 中编写面向对象程序的最佳途径。

面向对象，或面向对象程序设计，是一种程序编排方法，它将组件分块为离散的唯一实体。这些实体称为对象。用 Moose 的术语来说，每一个对象是某一个类的实例，类作为模版描述了对象包含的数据和专属的行为。

类

Perl 5 中的类存储类数据。类可以有一个名称。默认地，Perl 5 类使用包来提供名称空间：

```
{  
    package Cat;  
  
    use Moose;  
}
```

Cat 类看上去没有做任何事，但 Moose 却做这做那以定义此类并向 Perl 注册它。完成之后，你可以创建 Cat（猫）类的对象（或实例）：

```
my $brad = Cat->new();  
my $jack = Cat->new();
```

箭头语法看上去不会那么陌生。就像解引用，此处箭头调用一个类或对象的方法。

方法

method 就是和一个类关联的函数。表面上看起来，它就像一个完全限定的函数调用，但有两点重要的区别。第一，方法调用总是对执行方法 被调用者 进行的。按创建两个 Cat 对象的例子，类的名称（Cat）就是被调用者：

```
my $fuzzy = B<Cat>->new();
```

第二，一处方法调用总是引发 分派 策略。分派策略描述了对象系统如何决定调用 哪些方法。在只有一个 Cat 对象时看上去很显然，但方法分派是对象系统设计的基本要求。

Perl 5 中方法的被调用者是方法的第一个参数。举例来说，Cat 类可以拥有一个名为 meow()

```
(喵) 的方法: {
    package Cat;

    use Moose;

    B<sub meow>
    B<{>
        B<my $self = shift;>
        B<say 'Meow!';>
    B<}>
}
```

现在所有 Cat 实例都会因尚未进食而在清晨苏醒:

```
my $alarm = Cat->new();
$alarm->meow();
$alarm->meow();
$alarm->meow();
```

按照惯例, Perl 中方法的被调用者是一个名为 `$self` 的词法变量, 但这仅仅是一个普遍的惯例。`meow()` 方法的示例实现没有用到被调用者, 因此一旦方法分派完毕就毫不相关了。按此, `meow()` 就像 `new()`; 你可以安全地用类名 (Cat) 作为被调用者。这是一个类方法:

```
Cat->meow() for 1 .. 3;
```

属性

Perl 5 中每一个对象都是唯一的。对象可以包含 属性, 或者说和每一个对象关联的私有数据。你也可能听到它被描述为 实例数据 或 状态。

要定义对象属性, 可以将它们描述为类的一部分:

```
{
    package Cat;

    use Moose;

    B<< has 'name', is => 'ro', isa => 'Str'; >>
}
```

在英语中, 那行代码的意思是 “Cat 对象有一个 `name` 属性。它可读但不可写, 并且它是字符串。” 该行代码创建了一个访问器方法 (`name()`) 且允许你可以向构造函数 传递一个 `name` 参数:

```
use Cat;

for my $name (qw( Tuxie Petunia Daisy ))
{
    my $cat = Cat->new( name => $name );
    say "Created a cat for ", $cat->name();
}
```

属性的类型并非必需，在此情况下，Moose 将为你跳过核查和验证的部分：

```
{
    package Cat;

    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    B<< has 'age', is => 'ro'; >>
}

my $invalid = Cat->new( name => 'bizarre', age => 'purple' );
```

这样更为灵活，但在某人尝试对属性提供无效数据时便会导致奇怪的错误。灵活性和正确性之间的平衡取决于当时的编码标准以及欲捕获的错误类型。

Moose 的文档使用括号来分隔属性名称和它的特征：

```
has 'name' => ( is => 'ro', isa => 'Str' );
```

Perl 对此形式和本书所用的形式做相同的语法分析。你可以编写如下任一代码来实现相同的功能：

```
has( 'name', 'is', 'ro', 'isa', 'Str' );
has( qw( name is ro isa Str ) );
```

.....但在这种情况下，额外的标点使程序更加清晰。Moose 文档中使用的方法在处理多项特征时尤其有用：

```
has 'name' => (
    is      => 'ro',
    isa     => 'Str',

    # 高级 Moose 选项; perldoc Moose
    init_arg => undef,
    lazy_build => 1,
);
```

.....出于介绍的简易性，本书倾向于使用标点符号较少的形式。Perl 给你以灵活性，让你可以自由选择使代码更为清晰的编码方式。

如果你将某属性标记为可读 且 可写（用 `is => rw`），Moose 将创建一个 突变方法——你可以用它改变属性的值：

```
{
    package Cat;

    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age', is => 'ro', isa => 'Int';
    B<< has 'diet', is => 'rw'; >>
}
```

```

my $fat = Cat->new( name => 'Fatty', age => 8, diet => 'Sea Treats'
);
say $fat->name(), ' eats ', $fat->diet();

B<< $fat->diet( 'Low Sodium Kitty Lo Mein' ); >>
say $fat->name(), ' now eats ', $fat->diet();

```

尝试将 ro 访问器用作突变方法将抛出一个异常：

Cannot assign a value to a read-only accessor at

何时使用 ro 何时使用 rw? 这是关系到设计、便捷和代码纯度的事情。有一派意见 (immutability) 认为所有实例数据都应该采用 ro 并将相关数据都传递给构造函数。在 Cat 一例中, age() 仍可以是一个访问器, 但构造函数可以取得猫出生的 年份 并根据当前日期自行计算出年龄, 而非依赖于手动更新猫的年龄。

此方式有助于巩固验证代码的效果并保证所有创建的对象只包含有效的数据。这些设计目标值得考虑, 尽管在这方面 Moose 并不强制采用特定的哲学。

既然独立的对象可以各自拥有实例数据, 面向对象的价值就更加明显了。一个对象是其包含数据, 同时也是适用于这些数据的行为的书签。一个对象是具名数据和行为的集合。一个类是对该类实例持有的数据和行为的一个描述。

封装

Moose 允许你声明类的实例可以持有 哪些 属性, 以及如何处理这些属性。目前为止的例子尚未描述 如何 存储这些属性。如果真的需要了解, 这些信息是可以获取的, 但是直接说明的方式可以实际地改进你的程序。由此, Moose 鼓励 封装, 或者说对对象的外界使用隐藏内部细节。

考虑对 Cat 做出年龄修改; 并不直接对构造函数请求, 而是通过出生年份来计算一只 Cat (猫) 的年龄:

```

package Cat;

use Moose;

has 'name',          is => 'ro', isa => 'Str';
has 'diet',          is => 'rw';
B<< has 'birth_year', is => 'ro', isa => 'Int'; >>

B<sub age>
B<{>
    B<my $self = shift;>
    B<my $year = (localtime)[5] + 1900;>

    B<< return $year - $self->birth_year(); >>
B<}>

```

虽然 创建 Cat 对象的语法变了, 但 使用 Cat 对象的语法却没有。age() 方法始终完成它应该完成的任务, 至少就 Cat 类之外的代码而言可以理解其行为。它 如何 完成任务的方式已经改变, 但这属于 Cat 类的内部细节——由该类封装于自身内。

之前用于 创建 Cat 对象的语法仍可以原地不动; 定制生成的 Cat 构造函数以允许向其 传递 age 参数, 并由此计算正确的 birth_year。请参看 perldoc Moose::Manual::Attributes。

这个计算 `Cat` 年龄的新方式有着另外的优势；你可以使用 默认属性值 来减少创建 `Cat` 类所需的代码量：

```
package Cat;

use Moose;

has 'name',          is => 'ro', isa => 'Str';
has 'diet',          is => 'rw', isa => 'Str';
B<< has 'birth_year', is => 'ro', isa => 'Int', >>
      B<< default => sub { (localtime)[5] + 1900 }; >>
```

属性上的 `default` 关键字接受一函数引用，该引用在对象构造之时返回此属性的默认值。如果构造函数并未接收此属性某一合适的值，则该对象获得替代的默认值。现在你可以这样创建一只小猫：

```
my $skitten = Cat->new( name => 'Bitey' );
```

.....并且在明年之前这只小猫的年龄都将为 0。

你也可以使用一个简单的值，诸如数字或字符串，来作为默认值。当你需要为每一个对象计算某些唯一值（包括哈希或数组引用）时，使用函数引用。

多态

那行仅处理单一类型数据以及基于此数据之上单一行为的程序从对象的使用中获益不多。一个经过良好设计的面向对象程序应有能力处理各种类型的数据。当设计精良的类在合适处封装特定对象的细节时，会神奇地影响后续程序：一旦有机会，它可以选择变得不那么具体。

换句话说，将程序了解的 `Cat` 对象个体特征（属性）和它能做的事（方法）这类具体细节从程序中移入 `Cat` 类中意味着处理 `Cat` 实例的代码可以快乐地忽略 `Cat` 对象 如何做它该做的事。

用例子说话。考虑一个描述对象的函数：

```
sub show_vital_stats
{
    my $object = shift;

    say 'My name is ', $object->name();
    say 'I am ',      $object->age();
    say 'I eat ',      $object->diet();
}
```

很明显（在上下文中），你可以向此函数传入一个 `Cat` 对象并得到合理的结果。不那么明显的是，你也可以传入其他对象并得到合理的结果。这是一个称为 多态 的重要面向对象属性，你可以用一个类的对象替换另一个类的对象，只要它们以同样的方式提供相同的外部接口。

任何提供 `name()`、`age()` 和 `diet()` 访问器的对象都可以正常使用此函数。这个函数足够通用，每一个遵循接口的对象都是一个合法的参数。

一些语言和环境在程序互换实例之前要求两者间建立正式的关系。Perl 5 提供了多个方法来强制进行这类检查，但并不要求这样做。它默认的特殊系统让你可以足够等同地对待两个拥有同名方法的实例。一些人称其为 `duck typing`，此说法认为任何对象只要能 `quack()`

（“叫”）就足够像鸭子，因而你可以将其作为鸭子对待。在 `show_vital_stats()` 中体现的泛型的好处不是特定的 类型 也非对象的实现起决定作用。任何被调用者只要提供三个方法 `name()`、`age()` 和 `diet()`——它们不带参数，它们的返回值可以在字符串上下文中拼接，那么它就是一个合法的参数。你的代码中也许会包含成上百个类，它们之间也可以没有任何明显的关系，但如果它们符合预期的行为，那么此方法就可以正常工作。

对于为那么多类（哪怕是其中一部分）编写特定的函数提取并显示这类信息来说，这是一种质的提升。这个泛化的方法只需更少的代码，并使用经过良好定义的接口作为访问这些信息的机制，意味着上百个类可以各自采用任何可行的办法计算这些信息。计算方法的细节则位于它们的岗位上：在各个类各自的方法体内。

当然，仅出现名为 `name()` 或是 `age()` 的方法并不暗示对象的行为。一个 `Dog` 对象也可以包含 `age()` 访问器，它可以让你了解 `$rodney` 8 岁了而 `$lucky` 是 3 岁。`Cheese` 对象也可以含有 `age()` 方法，让你控制堆放 `$cheddar` 多久使得 奶酪味道更重；换句话说，`age()` 可以在一个类中是访问器而在另一个类中不是：

```
# 这只猫几岁了？
my $years = $zeppie->age();

# 把这批奶酪在仓库里存六个月
$cheese->age();
```

有时候了解一个对象 做什么 很有用。就是说，你要了解它的类型。

角色

角色 是一个具名行为和状态的集合。类就像是一个角色，它们之间重要的区别就是你可以对一个类进行实例化，但角色就不行。对于对象来说，类是将行为和状态组织为模版主要机制，而角色便是将行为和状态组织为具名集合的主要机制。

简单说来，角色和类是类似的。

某些 `Animal`（动物）——有 `name()`（名称）、有 `age()`（年龄）和偏好的 `diet()`（食物）和 `Cheese`（奶酪）——可以在仓库 `age()`（陈化）——的区别在于 `Animal` 可以饰演 `LivingBeing` 的角色，而 `Cheese` 则饰演 `Storable` 的角色。

虽然你 可以 检查传入 `show_vital_stats()` 的每一个对象是否是 `Animal` 类的实例，这样你便失去了部分泛型的特质。你可以用检查该对象是否 饰演 `LivingBeing` 角色来代替：

```
{
  package LivingBeing;

  use Moose::Role;

  requires qw( name age diet );
}
```

任何饰演此角色的对象必须提供 `name()`、`age()` 和 `diet()` 方法。这并不会自动发生；`Cat` 类必须明确标明它可以饰演此角色：

```
package Cat;

use Moose;

has 'name',      is => 'ro', isa => 'Str';
has 'diet',      is => 'rw', isa => 'Str';
has 'birth_year', is => 'ro', isa => 'Int';
```

```
default => (localtime)[5] + 1900;
```

```
B<with 'LivingBeing';>
```

```
sub age { ... }
```

那行代码有两个左右。第一，它告知 `Moose` 该类饰演具名角色。第二，它将角色合成到类中。该步骤检查类是否以某种方式提供了所需的所有方法和属性并避免了潜在的冲突。

`Cat` 类为具名属性提供了作为访问器的 `name()` 和 `diet()` 方法。它同时声明了自身的 `age()` 方法。

使用 `with` 关键字向类添加角色，其语句必须出现在属性声明之后，使得该合成过程可以识别任何生成的访问器方法。

当被问及它们是否提供 `LivingBeing` 角色时，并非所有的 `Cat` 实例会返回真，并且 `Cheese` 对象不应如此：

```
say 'Alive!' if $fluffy->does('LivingBeing');
say 'Moldy!' if $cheese->does('LivingBeing');
```

这个设计手法看上去可能像是多余的记账，但它从类和对象的实现中分离出了它们的能力。`Cat` 类的特殊行为，即存储动物的出生年份并直接计算年龄，本身就可以是一个角色：

```
{
    package CalculateAgeFromBirthYear;

    use Moose::Role;

    has 'birth_year', is => 'ro', isa => 'Int',
        default => sub { (localtime)[5] + 1900 };

    sub age
    {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year();
    }
}
```

将这段代码从 `Cat` 类中移出到一个分离的角色中使其也可以用于其它类中。现在，`Cat` 对象可以饰演两个角色：

```
package Cat;

use Moose;

has 'name', is => 'ro', isa => 'Str';
has 'diet', is => 'rw';

B<with 'LivingBeing', 'CalculateAgeFromBirthYear';>
```

由 `CalculateAgeFromBirthYear` 提供的 `age()` 方法的实现满足 `LivingBeing` 角色的需要，使之成功合成。无论对象 如何 扮演此角色，对对象是否扮演 `LivingBeing` 角色的检查结果仍然不变。一个类可以选择提供它自己的 `age()` 方法或从其它角色中获得一份，这并不重要。重要的是它包含此方法。这就是 同质异晶性。

角色和 `DOES()`

对一个类应用一个角色意味着你在调用该类和它的实例的 `DOES()` 方法时返回真：

```
say 'This Cat is alive!' if $kitten->DOES( 'LivingBeing' );
```

继承

Perl 5 对象系统的另一个特性就是 继承，即一个类将另一个类专门化。这两个类间建立起一种关系，其中子类从父类继承属性和行为。就两个提供同一角色的类来说，你可以用子类替换父类。在某种意义上说，一个子类通过其父类的存在隐式地提供了某角色。

Perl 5 中最近基于角色的对象系统实验显示，在一个系统中几乎所有用到继承的地方都可以用角色来替代。决定使用何者大体上是熟悉程度的事。角色提供了合成时安全，更好的类型检查，组织良好且更低耦合的代码，可对名称和行为做出细粒度的控制，但继承对其他语言的用户来说更为熟悉。设计上的问题就是一派是否确实扩展了另一派或者说它是否提供了额外（或者，至少是，不同）的行为。

考虑 `LightSource`（光源）类，它提供了两个公共属性（`candle_power` 和 `enabled`）以及两个方法（`light` 和 `extinguish`）：

```
{
    package LightSource;

    use Moose;

    has 'candle_power', is => 'ro', isa => 'Int',
        default => 1;
    has 'enabled',      is => 'ro', isa => 'Bool',
        default => 0,    _writer => '_set_enabled';

    sub light
    {
        my $self = shift;
        $self->_set_enabled(1);
    }

    sub extinguish
    {
        my $self = shift;
        $self->_set_enabled(0);
    }
}
```

`enabled` 属性的 `_writer` 选项创建了一个私有访问器，可在类内部用于设置值。

继承和属性

创建 `LightSource` 的子类使得定义一支行为和 `LightSource` 相似、但提供亮度百倍于常的超级蜡烛成为可能：


```

{
    package LightSource::SuperCandle;

    use Moose;

    B<extends 'LightSource'>;

    has 'B<+>candle_power', default => 100;
}

```

`extends` 语法结构接受一个类名称列表作为当前类的父类。位于 `candle_power` 属性名称前的 `+` 指出当前类扩展了此属性的定义。在这种情况下，超级蜡烛覆盖了光源的默认值，因此任何新建的 `SuperCandle` 的亮度值为 100 支蜡烛。另一个属性以及两个方法对 `SuperCandle` 实例也是可用的；当你在这样一个实例上调用 `light` 或 `extinguish` Perl 会先在 `LightSource::SuperCandle` 内查找这些方法，然后是父类列表。最终它在 `LightSource` 里找到了它们。

属性继承的工作方式与此类似，除了构造此实例的行为使得所有适当的属性按正确的方式提供（参见 `perldoc Class::MOP`）。

单重继承的方法分派顺序理解起来比较简单。当一个类拥有多个父类时（多重继承），分派便不那么显然了。默认的，Perl 5 提供了深度优先的方法解析策略。它先搜索类的首个具名父类，接着在搜索后续具名父类之前递归地搜索此类的所有父类。这个行为通常令人迷惑；在你理解多重继承之前请避免使用它，并尽可能采用其他替代手段。参见 `perldoc mro` 以获取有关方法解析和分派策略的更多细节。

继承和方法

你可以在子类中覆盖父类的方法。设想一个你无法熄灭的光源：

```

{
    package LightSource::Glowstick;

    use Moose;

    extends 'LightSource';

    sub extinguish {};
}

```

所有对此类的 `extinguish` 方法的调用将毫无作用。Perl 的方法分派系统将先找到这个方法并且不会再在父类中查找与此同名的其他方法。

有时候覆盖后方法也需要来自父类同名方法的某些行为。`override` 命令告诉 Moose，该子类故意覆盖此具名方法。`super()` 函数可以用来从覆盖方法分派到被覆盖方法：

```

{
    package LightSource::Cranky;

    use Carp;
    use Moose;
}

```

```

    extends 'LightSource';

    B<override> light => sub
    {
        my $self = shift;

        Carp::carp( "Can't light a lit light source!" )
            if $self->enabled;

        B<super()>;
    };

    B<override> extinguish => sub
    {
        my $self = shift;

        Carp::carp( "Can't extinguish an unlit light source!" )
            unless $self->enabled;

        B<super()>;
    };
}

```

这个子类在点亮和熄灭一个已经处于当前状态的光源时增加了一条警告。`super()` 函数 在遵守正常 Perl 5 方法解析顺序的同时将当前方法分派到最近的父类实现中。

你可以用 Moose 的方法修饰符实现相同的行为。参见 `perldoc Moose::Manual::MethodModifiers`。

继承和 `isa()`

从父类继承意味着子类及其所有实例在调用其上 `isa()` 方法时返回真：

```

say 'Looks like a LightSource' if $sconce->isa( 'LightSource' );
say 'Monkeys do not glow'      unless $chimpy->isa( 'LightSource' );

```

Moose 和 Perl 5 OO

Moose 提供了许多原本需要你自行在 Perl 5 的默认对象模型中实现的特性。虽然你 可以自行构建来自 Moose 的全部特性（参见 `blessed_references`），或者用一系列 CPAN 发行包修修补补地实现，但 Moose 是一个合适且连贯的包，包括了优秀的文档，且是许多成功项目的一部分，还有，它正由一个善解人意且有才的社区积极地开发着。

默认地，Moose 对象不需要你担心构造器、析构器、访问器和封装。Moose 对象可以扩展并和来自平淡无奇的 Perl 5 对象系统的对象协同工作。你同时也得到了 元编程——一种通过系统自身访问系统实现的方式——以及随附的扩展性。如果你曾考虑过某类或对象上可以调用什么方法或者对象支持什么属性，这类元编程信息可以通过 Moose 得到：

```

my $metaclass = Monkey::Pants->meta();

say 'Monkey::Pants instances have the attributes: ';

```

```

say $_->name for $metaclass->get_all_attributes;

say 'Monkey::Pants instances support the methods: ';

say $_->fully_qualified_name for $metaclass->get_all_methods;

```

你甚至可以知道那些类扩展了一个给定的类:

```

my $metaclass = Monkey->meta();

say 'Monkey is the superclass of: ';

say $_ for $metaclass->subclasses;

```

请分别参阅 `perldoc Class::MOP::Class`、`perldoc Class::MOP` 以获取有关元类操作符、`Moose` 元编程的更多信息。

`Moose` 和它的 元对象协议 (或称 `MOP`) 为一个更好的、用于在 `Perl 5` 中操作类和对象的语法提供了可能。如下是合法的 `Perl 5` 代码:

```

use MooseX::Declare;

B<role> LivingBeing { requires qw( name age diet ) }

B<role> CalculateAgeFromBirthYear
{
    has 'birth_year', is => 'ro', isa => 'Int',
        default => sub { (localtime)[5] + 1900 };

    B<method> age
    {
        return (localtime)[5] + 1900 - $self->birth_year();
    }
}

B<class Cat with LivingBeing with CalculateAgeFromBirthYear>
{
    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';
}

```

来自 CPAN 的 `MooseX::Declare` 扩展使用了一个称为 `Devel::Declare` 的聪明的模块 向 `Perl 5` 添加新语法, 特别是为了 `Moose`。 `class`、`role` 和 `method` 关键字减少 了在 `Perl 5` 中编写良好的面向对象代码所需的样版数量。 特别注意本例称述性的本质, 还有 现在 `age` 方法开头的 `my $self = shift;` 代码行不是必须的。

自 `Perl 5.12` 起, `Perl 5` 核心提供了对 `Devel::Declare` 的支持, 但这个模块不是核心模块。

采用这整个方案的一个缺点是你必须能够安装 CPAN 模块 (或使用诸如 `Strawberry Perl` 或 `Strawberry Perl Professional` 之类定制的 `Perl 5` 发行版, 它们会替你包括这些), 但和 `Perl 5` 的内置面向对象比较, `Moose` 在清晰和简洁方面的优势应是明显的。

参见 `perldoc Moose::Manual` 以获取有关使用 `Moose` 的更多信息。

经 `bless` 后的引用

Perl 5

的默认对象系统故意最小化。以下三条简单的规则相组合构成了简单——但有效的——基本对象系统：

- * 一个类就一个包；
- * 一个方法就是一个函数；
- * 一个 (`bless` 后的) 引用就是一个对象。

你已经在 `Moose` 里见过了前两条规则。第三条规则是新出现的。`bless` 关键字将一个类的名称和一个引用关联起来，使得任何在该引用上进行的方法调用由与之相关联的类来解析。它听上去比实际的要复杂一些。

虽然这些规则很好地解释了 Perl 5 的底层对象系统，它们在实际应用时显得捉襟见肘，特别是对于较大型的项目来说。特别地，它们几乎没有提供元编程（使用 API 操作程序本身）的组件。

`Moose` (`moose`) 对于正式而现代化的、且大于几百行 Perl 程序来说是更好的选择，但你很可能在现存代码中遇见赤裸裸的 Perl 5 OO。

默认的 Perl 5 对象构造器是一个创建引用并对其进行 `bless` 的方法。出于惯例，构造器通常命名为 `new()`，但并非一定如此。构造器几乎总是 类方法：

```
sub new
{
    my $class = shift;
    bless {}, $class;
}
```

`bless` 接受两个参数，与类相关联的引用以及类的名称。虽然程序抽象建议使用一个单独的方法来处理，但你仍可以在构造器或类之外使用 `bless`。类名称不需要事先存在。

设计上，构造器以被调用者的形式接收类名。直接硬编码类名称也是可能的，但不推荐。参数式的构造器使得此方法可以通过继承、委托或导出来重用。

引用的类型对对象上的方法调用没有影响。它只掌控对象如何存储 实例数据——对象自身的信息。哈希引用最为常见，但你可以 `bless` 任何类型的引用：

```
my $array_obj = bless [], $class;
my $scalar_obj = bless \ $scalar, $class;
my $sub_obj = bless \&some_sub, $class;
```

`Moose` 中创建的类由它们自己声明式地定义各自的对象属性，Perl 5 默认的 OO 则非常宽松。一个存储球衣号和位置、代表篮球运动员的类可能会使用如下构造器：

```
package Player;

sub new
{
    my ($class, %attrs) = @_;
```

```

        bless \%attrs, $class;
    }

```

.....运动员可以这样创建:

```

my $joel = Player->new(
    number    => 10,
    position => 'center',
);

my $jerryd = Player->new(
    number    => 4,
    position => 'guard',
);

```

在类的内部, 方法可以直接访问哈希元素:

```

sub format
{
    my $self = shift;
    return '#' . $self->{number} . ' plays ' . $self->{position};
}

```

类之外的方法也可以这样做。这样便违反了封装——特别是, 它意味着你绝不能在不破坏外部代码的情况下改变对象的内部表示, 除非投机取巧——因此, 保险起见还应提供访问器方法:

```

sub number { return shift->{number} }
sub position { return shift->{position} }

```

即便只有两个属性, Moose 在那些非必须代码方面显得更具吸引力。

Moose 的创建访问器的默认行为鼓励你在注重封装和泛型的情况下编写正确的代码。

方法查找和继承

除实例数据外, 对象的另一部分就是方法分派。给定一个对象 (一个 `bless` 后的引用), 如下形式的方法调用:

```

my $number = $joel->number();

```

.....将查找与经 `bless` 后的引用 `$joel` 相关联类的名称。在此例中, 该类就是 `Player`。接下来, Perl 在 `Player` 包内查找一个名为 `number` 的函数。如果 `Player` 类从其它类继承而来, Perl 也会在父类查找 (如此继续) 直到它找到 `number` 方法为止。如果存在的话, Perl 以 `$joel` 作为调用物调用它。

Moose 类在元模型中存放各自的继承信息。每一个经 `bless` 后的引用的类将父类信息存放在一个名为 `@ISA` 的包全局变量中。方法分派器会在一个类的 `@ISA` 中查找它的父类, 以在其中搜索合适的方法。因此, 一个 `InjuredPlayer` 类会在其 `@ISA` 中包含 `Player`。你可以这样编写这重关系:

```

package InjuredPlayer;

@InjuredPlayer::ISA = 'Player';

```

许多现存的 Perl 5 项目都这样做，但用 `parent` 编译命令来替代会更容易些：

```
package InjuredPlayer;

use parent 'Player';
```

Perl 5.10 为替换增加于 Perl 5.004_4 的 `base` 编译命令而添加了 `parent`。如果你无法使用 `Moose`，请使用 `parent`。

可以从多个父类继承：

```
package InjuredPlayer;

use parent qw( Player Hospital::Patient );
```

在解析方法分派时，Perl 5 在传统上偏向于对父类使用深度优先搜索。这就是说，如果 `InjuredPlayer` 从 `Player` 和 `Hospital::Patient` 两者继承，一个在 `InjuredPlayer` 实例上调用的方法将先分派到 `InjuredPlayer`，然后是 `Player`，接着经过所有 `Player` 的父类来到 `Hospital::Patient`。

Perl 5.10 增加了一个名为 `mro` 的编译命令，它允许你另行使用称作 `C3` 的方法解析策略。虽然特定的细节可能在处理复杂的多重继承布局时变得复杂，关键的区别是，方法解析过程将在访问父类之前访问所有的子类。

虽然其他技巧（诸如 角色 `roles` 和 `Moose` 方法修饰符）允许你避开多重继承，但 `mro` 编译命令可以帮助你避免方法分派时令人惊讶的行为。可以这样在类中启用：

```
package InjuredPlayer;

use mro 'c3';
```

除非你在编写具有互操作插件的复杂框架，你几乎不会用到它。

AUTOLOAD

如果在调用者及其超类的类定义内没有可用的方法，Perl 5 接下来将按照所选方法解析顺序在每个类中查找 `AUTOLOAD` 函数。Perl 会调用它找到的任何 `AUTOLOAD`，由此提供或谢绝所需方法。参加 `autoload` 以获取更多细节。

如你所想的那样，在面对多重继承和多个候选 `AUTOLOAD` 目标时会变得很复杂。

方法覆盖和 SUPER

与 `Moose` 中一样，你可以在默认的 Perl 面向对象系统中覆盖方法。不幸的是，Perl 5 核心并没有提供指出你覆盖父类方法 意图 的机制。更糟糕的是，任何预声明、声明或导入子类的函数都可能因重名而覆盖父类方法。你可以忘记使用 `Moose` 的 `override` 系统，但在 Perl 5 默认的面向对象系统中你根本没有这样（甚至是可选的）一重保护。

要在子类中覆盖一个方法，只需声明一个和父类方法同名的方法。在覆盖的方法内，你可以通过 `SUPER::` 分派指示来调用父类方法：

```
sub overridden
{
    my $self = shift;
    warn "Called overridden() in child!";
```

```

        return $self->SUPER::overridden( @_ );
    }

```

方法名的 `SUPER::` 前缀告诉方法分派器将此方法分派到 父类 的具名实现。你可以向其传递任何参数，但最好还是重用 `@_`。

注意当重分派到父类方法时，这个分派器依赖于覆盖方法最初被编译的包。这长期以来是一个错误的特性，只是为了向后兼容而保留着。如果你向其它类或角色导出方法或手动合成类和角色，你会和此项特性正面冲突。CPAN 上的 `SUPER` 模块可以为你绕过它。`Moose` 同样能够出色地处理该问题。

应付经 `bless` 后引用的策略

可能的话避免使用 `AUTOLOAD`。如果 必须 用到，你应该用它来转发函数（`functions`）定义以帮助 `Perl` 知道哪个 `AUTOLOAD` 会提供方法的实现。

使用访问器方法而非直接通过引用访问实例数据。甚至在类内部的方法体内也应该这样做。由你自己来生成这些方法相当乏味；如果你无法使用 `Moose`，考虑使用诸如 `Class::Accessor` 这类模块来避免重复编写样板。

准备好某人某时最终将继承你的类（或委托或重新实现接口）。通过不对代码内部细节做出假定、使用两参数形式的 `bless` 和把类分割为最小职责单元，可以使得他人的工作更加轻松。

不要在同一个类里混用函数和方法。

为每一个类使用单独的 `.pm` 文件，除非该类是一个仅用于某处的小型自包含辅助类。

考虑使用 `Moose` 和 `Any::Moose` 来替代赤裸的 `Perl 5 OO`；它们可以轻松和 `Perl 5` 对象系统的类和对象互动，还减轻了几几乎所有因类声明而带来的无趣，同时提供了更多及更好的特性。

反射

反射（或称 自省）是运行期间向一个程序询问其自身情况的过程。即便你可以在不使用反射的情况下编写不少有用的程序，一些诸如元编程（`code_generation`）等技术从深刻了解系统中的实体获益良多。

`Class::MOP`（`class_mop`）简化了许多对象系统中的反射任务，但是很多有用的程序并非彻底面向对象，很多程序也不会用到 `Class::MOP`。因为没有有一个正规的系统，`Perl` 中存在着一些有效进行反射的惯用语（`idioms`）。

检查一个包是否存在

为检查一个包是否存在于系统之中——即，如果一段代码在某一点执行了一条带包名称的 `package` 指令——就是检查它是否从 `UNIVERSAL` 继承下来，这可以通过检查该包是否能够执行 `can()` 方法来实现：

```
say "$pkg exists" if eval { $pkg->can( 'can' ) };
```

虽然你 可以 使用名为 `0` 或 `''`仅仅在你符号化地定义它们时，因为这些 并非 `Perl 5` 语法分析器禁止的标识符。的包，`can()` 方法会在你将其作为调用物时抛出一个方法调用异常。`eval` 代码块可以捕获这类异常。

你也 可以 对符号表的条目进行一一礼拜，但上述方法更加快速同时也更易理解。

检查一个类是否存在

由于 `Perl 5` 对包和类不加以严格区分，检查包存在性的技巧同时可用于检查一个类是否存在。尚没有方法可以判断一个包是否是一个类。你 可以 用 `can()` 来检查一个包 是否可以执行 `new()`，但这样无法保证任何找到的 `new()` 是方法，更不要说是构造 器了。

检查一个模块是否被加载

如果知道模块的名称，你可以通过查看 `%INC` 哈希来确定 Perl 是否相信它已经从磁盘加载了这个模块。这个哈希和 `@INC` 是对应的；当 Perl 5 用 `use` 和 `require` 加载代码时，会在 `%INC` 中存储一个条目，其中键是欲加载模块的路径化名称，值是模块完整的磁盘路径。就是说，加载 `Modern::Perl` 等效于：

```
$INC{'Modern/Perl.pm'} =>
    '/path/to/perl/lib/site_perl/5.12.1/Modern/Perl.pm';
```

路径的细节大部分取决于安装，但出于测试 Perl 是否成功加载一个模块目的，你可以把模块名转换为正规文件形式并测试在 `%INC` 内是否存在：

```
sub module_loaded
{
    (my $modname = shift) =~ s!::!/!g;
    return exists $INC{ $modname . '.pm' };
}
```

没有什么阻止其它代码修改 `%INC`。按你偏执的程度，你可以自行检查路径和预期的包内容，但是拥有充足利用修改此变量的模块（诸如 `Test::MockObject` 和 `Test::MockModule`）可以这样做。修改 `%INC` 但理由不足的代码应该被替换。

检查模块的版本

无法保证某给定的模块是否提供版本号。即便如此，所有模块都继承自 `UNIVERSAL (universal)`，因此它们全含有 `VERSION()` 方法以供调用：

```
my $mod_ver = $module->VERSION();
```

如果给定的模块不覆盖 `VERSION()` 或不包含包变量 `$VERSION`，这个方法将返回一个未定义值。类似地，如果该模块不存在，则对方法的调用将会失败。

检查一个函数是否存在

确定函数是否存在最简单的机制就是对包名使用 `can()` 方法：

```
say "$func() exists" if $pkg->can( $func );
```

除非 `$pkg` 是合法的调用物，否则 Perl 将会抛出异常；如果对其合法性有任何怀疑，可以将此方法调用包装在一个 `eval` 块内。注意，若函数所在的包没有正确地覆盖 `can()`，使用 `AUTOLOAD()` (`autoload`) 实现的函数可能会导致错误结果。这在其他包里便会 导致代码缺陷。

你可以用这个技巧决定一个模块的 `import()` 是否向当前名称空间导入某函数：

```
say "$func() imported!" if __PACKAGE__->can( $func );
```

你也可以检查符号表和类型团来决定函数是否存在，但这种机制更简单也更好解释。

检查一个方法是否存在

没有检查某给定函数究竟是函数还是方法的通用办法。一些函数身兼双职，既是函数也是方法，尽管听上去过于复杂并且同时是失误，但这确实是一个允许的特性。

搜查符号表

Perl 5 符号表是一个种类特别的哈希，其中的键是包全局符号的名称，值则是类型团。类型团是一种核心数据结构，它可以包含一个标量、一个数组、一个哈希、一个文件句柄和一个函数。Perl 5 内部在查找这些变量时使用类型团。

通过在包名末尾添加双冒号，你可以将符号表当作哈希访问。例如，`MonkeyGrinder` 包的符号表可以通过 `%MonkeyGrinder::` 访问。

你可以用 `exists` 操作符检查特定的符号名是否存在于符号表中（或随你喜好 修改符号表来添加 或 `or` 删除 符号）。还应注意到一些对 Perl 5 核心的变更已经修改了默认出现的类型团条目。特别是，Perl 5 早期版本总是为每个类型团默认提供一个标量变量，现代化的 Perl 5 已经取消。

参见 `perldoc perlmod` 中的“Symbol Tables”小节以获取更多细节，最好采用不同于本章介绍的反射技巧。

高级 Perl 面向对象

利用 `Moose (moose)` 在 Perl 5 中创建并使用对象是容易的。设计 一个好的对象系统则不那么简单。额外的抽象能力同时也为混乱提供了可能。只有实践经验才能帮助你理解最重要的设计技能.....还有一些原理可以作为指导。

多用合成而非继承

初级 OO

设计通常过度地使用继承。常见的类结构尝试将所有实体的行为建模于单类系统之中。由于你必须理解整个层次结构，这样就给理解该系统增加了概念上的开销；同时这也向每一个类增加了技术上的份量，因为冲突的职责和方法可能成为所需行为和进一步修改的绊脚石。

由类提供的封装为代码组织提供了更好的方法。你并不需要从超类继承以向对象的用户提供所需行为。`Car`（车）对象无需从 `Vehicle::Wheeled` 对象继承，它可以实例属性的形式包含若干 `Wheel` 对象。

将复杂的类分解为较小的、专一的实体（无论类或是角色）可提升封装性并降低类或角色身兼数职的可能。更小更简单、封装更好的实体无论是从理解、测试、维护上说都更为简易。

单一职责原则（SRP）

当你设计你的对象系统时，应按职责给问题建模，或按每一个特定实体改变的理由。举例说来，一个 `Employee` 对象也许会代表有关人名、联系方式以及其他个人数据的特定信息，而 `Job` 对象可能代表的是业务职责。一个简单的设计也许会将这两者合并为单一的实体，但职责分离允许 `Employee` 类仅考虑管理有关此是何人的问题，且 `Job` 类只需代表其人的工作。（例如，两个 `Employees` 也许实行工作（`Job`）分担制。）

当每个类都有单一的职责时，你可以更好地封装特定于类的数据和行为，并降低类间的耦合。

不要重复你自己（DRY）

复杂性和重复使开发和维护活动更加纷繁。DRY 原则（“Don't Repeat Yourself”）提醒你挑出并减少系统内的重复。重复的形式是多种多样的，既出现在数据也出现在代码中。如果你发现自己正重复配置信息、用户数据以及系统内其它人为数据；作为替代，找一个正式的单一的信息表示形式，接着从此表示生成其它人为数据。

这个原则有助于减少系统中重要部分不同步的可能，并帮助你找到系统及其数据的优化表达。

Liskov 替换原则（LSP）

Liskov

替换原则提出给出某类型（类或角色或子类的特化形式），其子类型应能在不窄化其接收数据的类型、不扩张其产出数据类型的情况下替换其父类型。换句话说，它们应该尽可能将接收数据一般化并将产出数据专一化。

理解该原则最简单的方式是想象两个类，`Dessert` 和 `PecanPie`，后者是前者的子类。如果这两个类遵循 Liskov 替换原则，则在测试套件中每一个使用 `Dessert` 对象的地方用 `PecanPie` 对象替换后，测试仍应通过。更多细节参见 Reg Braithwaite 的

"IS-STRICTLY-EQUIVALENT-TO-A",
<http://weblog.raganwald.com/2008/04/is-strictly-equivalent-to.html>。。

子类型和强制转换

Moose

允许你声明及使用类型，并通过子类型对其扩展，以进一步对数据代表的事物及其行为作出更为专一的描述。你可以使用这些类型注解验证特定函数或方法处理的是否为合适的数据，甚至凭此指定从一个类型强制转换到另一个类型的机制。

更多信息请参见 `Moose::Util::TypeConstraints` 和 `MooseX::Types`。

不可变性

一个常见于面向对象程序设计新手的模式便是将对象视为一捆记录，并使用方法获得和设置内部的值。虽然容易实现和理解，但是它可能引发不幸的诱惑，使行为上的职责散播到整个系统各个独立的类中。

高效处理对象最为有用的技巧便是告诉它们做什么而非怎么做。如果你发现自己正访问对象的实例数据（即使通过访问器方法）都有过度访问类职责之嫌。

避免这种行为的一种方法是对象视为不可变。向对象的构造器传入所有相关的配置数据，接着禁止所有来自该类外部对此类信息的修改。不要暴露任何改变实例数据的方法。

一些设计仅制止类 内部 对实例数据的修改，虽然这更加难以做到。