Perl 面向对象编程的两种实现和比较

Perl 面向对象编程的散列表实现和数组实现

童 鑿 (yitong@cn.ibm.com), 软件工程师, IBM

简介： 本文比较了在 Perl 中两种主流的面向对象编程的实现方式，基于匿名哈希表的实现和基于数组的实现。深刻地剖析了两种实现的技术内幕，并且提供了可供读者直接使用的代码和模块 示例。在文章的最后作者比较了两种实现方式的优劣，并对读者给出了在实际工作中选择何种方式实现面向对象编程的建议。

发布日期： 2009 年 9 月 17 日
级别： 初级
访问情况： 2724 次浏览
评论： 0 (查看 | 添加评论 - 登录)

背景

我们常常可以从软件工程的书和文章中，或者项目经理的口中，听到面向对象编程这样的字眼。与大多数时髦的技术用词不同，面向对象编程的确可以 为我们的软件设计和开放工作带来本质性的变化。 Perl 作为一种成熟的"面向过程"的语言，同样也提供了对于面向对象编程的支持。

一个好的"面向对象"的设计不仅是以数据为中心，它还尽力地封装并且隐藏了实际的数据结构，而且只对外界开放有限的，具备良好文档的接口。在下文中，我们将看到如何使用 Perl 语言的特性来实现这些面向对象设计的优点的。

Perl 中有两种不同地面向对象编程的实现，一是基于匿名哈希表的方式，每个对象实例的实质就是一个指向匿名哈希表的引用。在这个匿名哈希表中，存储来所有的实例 属性。二是基于数组的方式，在定义一个类的时候，我们将为每一个实例属性创建一个数组，而每一个对象实例的实质就是一个指向这些数组中某一行索引的引用。 在这些数组中，存储着所有的实例属性。

回页首

面向对象的概念

首先，我们定义几个预备性的术语。

实例 (instance)：一个对象的实例化实现。

标识 (identity)：每个对象的实例都需要一个可以唯一标识这个实例的标记。

实例属性 （instance attribute）：一个对象就是一组属性的集合。

实例方法 (instance method)：所有存取或者更新对象某个实例一条或者多条属性的函数的集合。

类属性（class attribute）：属于一个类中所有对象的属性，不会只在某个实例上发生变化。

类方法（class method）：那些无须特定的对性实例就能够工作的从属于类的函数。

回页首

基于匿名散列表的方法

首先我们来谈谈基于匿名散列表的面向对象实现。首先，我们需要定一个匿名散列表，并用一个引用指向这个匿名散列表。如清单 1 所示，我们定义了一个初始化函数来封装这个匿名散列表的初始化过程。这个函数接受参数作为初始值，并且用这些值初始化其内部包含的匿名散列表，并且返回一 个指向这个匿名散列表的引用。在这个例子当中，我们创建了一个 Person 模块，并且定义了一个可以实例化模块 Person 的 new 函数。

清单 **1.** 基于匿名哈希表的面向对象编程

```
package Person;
 sub new {
 my ($name, $age) = @_;
 my $r_object = {
“ name ” => $name,
“ age ” => $age
 }
```

```
 return $r_object;
 }

 my $personA = Person->new ( “ Tommy ” , 22 );
 my $personB = Person->new ( “ Jerry ” , 30 );
 print “ Person A ’ s name: ” . $personA->{name} . “ age: ” . $personA->{age} . ” .\n ” ;
 print “ Person B ’ s name: ” . $personB->{name} . “ age: ” . $personB->{age} . ” .\n ” ;
```

但是，现在的这个方案有一个致命的缺点，Perl 的编译器并不知道如何 new 函数所返回的指向匿名哈希表的引用属于哪个类（模块）。这样的话，如果要使用类中的实例方法，只能直接标出方法所属于的类（模块）的名字，并将引用作为方 法的第一个参数传递给它，如清单 2 所示。

清单 **2.** 基于匿名哈希表的面向对象编程中实例方法

```
package Person;
…
 sub change_name {
 my ($self, $new_name) = @_;
 $self->{name} = $new_name;
 }

 my $object_person = Person->new ( “ Tommy ” , 22);
 print “ Person ’ s name: ” . $object_person->{name} . “ .\n ” ;
 Person::change_name ($object_person, “ Tonny ” );
 print “ Person ’ s new name: ” . $object_person->{name} . “ .\n ” ;
```

对于这个问题，Perl 中的 bless 函数提供了一个解决问题的桥梁。 bless 以一个普通的指向数据结构的引用为参数，它将会把那个数据结构（注意：此处不是引用本身）标记为属于某个特定的包，这样就赋予了这个匿名哈希表的引用以多 态的能力。同时，我们使用箭头记号来直接调用那些实例方法。见清单 3 。

清单 3 中的“ bless ($self) ”，将指向一个匿名哈希表的引用标记为属于当前包，也就是 package Person 。所以，当 Perl 看到“ $object_person->change_name ($name) ”时，它会决定 $object_person 属于 package Person 。 Perl 就会如下所示地调用这个函数，“ Person::change_name ($object_person, $name) ”，和清单 2 中的第一种实现一样。换而言之，如果使用箭头的方式调用一个函数，箭头左边的那个对象将作为相应子例程的第一个参数。 Perl 的实例方法的本质其实就是一个第一个参数碰巧为对象引用的普通子例程。

清单 **3.** 基于匿名哈希表的面向对象编程中改进的实例方法

```
package Person
 sub new {
 my $self = {};
 shift;
 my ($name, $age) = @_;
 $self->{name} = $name;
 $self->{age} = $age;
 bless ($self);
 return $self;
 }

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

 my $object_person = Person->new ( “ David ” , 27);
 print “ Name: “ . $object_person->{name} . “ \n ” ;
 $object_person->change_name ( “ Tony ” );
 print “ Name: “ . $object_person->{name} . “ \n ” ;
```

Perl 的这种调用相应模块函数的能力被称做为运行时联编。调用 new 方法之后，返回一个匿名哈希表的引用，并且包含相应类的名字。

与其他流行的面向对象编程语言不同，Perl 中并没有针对类属性和类方法的特定语法。类属性只是包中的全局变量，而类方法则是不依赖于任何特定实例的普通子例程。清单 4 是一个关于类属性和类方法的例子。与实例方法不同，我们使用 Person::calculate_person_number () 的形势来调用类方法。这样的话，指向匿名哈希表的引用将不会作为第一个调用参数传入，我们与不需要在包的子例程附加处理传入引用的代码。

清单 **4.** 基于匿名哈希表的面向对象编程中的类属性和类方法

```
package Person;
…
 my $person_number = 0;
```

```
…
 sub new {
…
 $person_number++;
 }
…
 sub calculate_person_number {
 return $person_number;
 }

 my $object_personA = Person->new ( “ David ” , 27);
 my $object_personB = Person::new ( “ Tonny ” , 27);
 my $person_number = Person::calculate_person_number ();
 print “ We have ” . $person_number . “ persons in all. \n ” ;
```

回页首

基于匿名散列表的方法中的继承:

Perl 允许一个模块在一个特殊的名为 @ISA 的数组中制定一组其他模块的名称。当在模块中找不到某个实例方法时，它就为检查那个模块的 @ISA 是否被初始化。如果已经初始化了，它就为检查其中的某个模块是否支持这个“缺少”的函数。如果它按照深度优先的层次结构搜索 @ISA 数组并且发现同名的方法，它会调用第一个被发现的同名方法并将控制权交给它。我们利用 Perl 语言的这个特性实现了继承。

考虑这样一个类的层次，我们定义一个 Employee 类，继承于基类 Person，如清单 5 所示。

我们将类名 Person 放入包 Employee 的 ISA 数组中，这样当调用一个在包 Employee 中没有定义的函数时，Perl 编译器会自动在 Person 类寻找这个函数。当用户调用 new 函数初始化一个 Employee 对象实例的时候，Employee 的 new 函数会在内部调用它的基类的 new 函数，并且返回一个包含部分以初始化的基类实例属性的匿名哈希表。接着 Employee 的 new 函数将继续执行 new 函数的剩余代码，完成属于 Employee 自身的初始化工作，为 Employee 中剩余的实例属性赋值。

清单 **5.** 基于匿名哈希表的面向对象编程中的继承

```
use Person;

 package Employee;
 @ISA = qw (Person);

 sub new {
 shift;
 my ($name, $age, $salary) = @_;
 my $self = Person->new ($name, $age);
 $self->{salary} = $salary;
 bless ($self);
 return $self;
 }

 sub change_salary {
 my $self = shift;
 my $new_salary = shift;
 $self->{salary} = $new_salary;
 }

 my $object_employee = Employee->new ( "Tonny", 28, 10000 );
 print "Name : " . $object_employee->{name} . ", Age : " . $object_employee->{age} .
 ", Salary : " . $object_employee->{salary} . ". \n";
 $object_employee->change_name ("Tommy");
 $object_employee->change_salary (13000);

 print "Name : " . $object_employee->{name} . ", Age : " . $object_employee->{age} .
 ", Salary : " . $object_employee->{salary} . ". \n";
```

当用户调用 Employee 的 change_name 方法和 change_salary 方法时，Perl 解析器会在 Employee 包和 Person 包中搜索，寻找符合的函数供期调用。

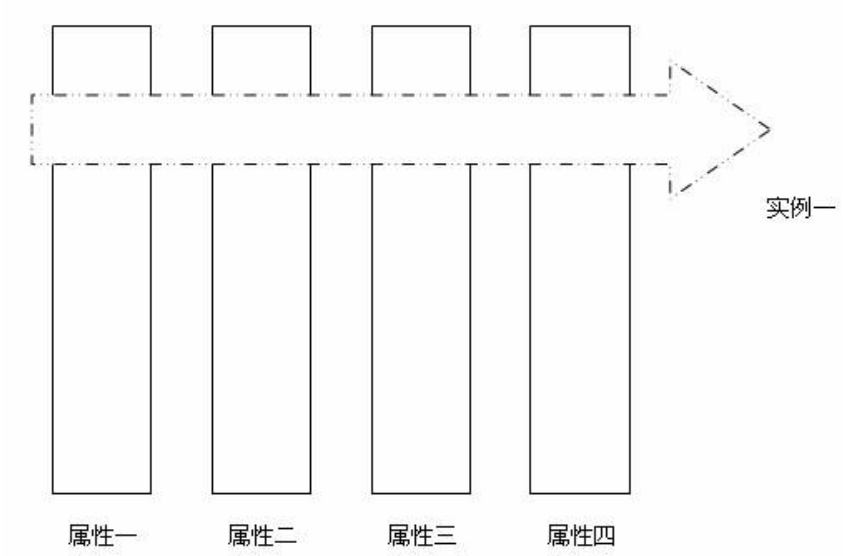
回页首

基于数组的方法

基于匿名哈希表的面向对象编程方法中有两个明显的不足：一是无法为属性提供一种访问限制，限制外部对内部属性的访问和改变。二是在处理大规模 的实例的情况下，系统的内存开销颇大。 100 个实例意味着将创建 100 个散列表，这 100 个散列表都要为插入新纪录的操作而分配额外的存储空间。除了基于匿名散列表的实现，我们也可以利用数组来存储属性，实现面向对象的编程。

整个实现的数据结构非常简单，我们将为每一个类的实例属性分配一个数组（见图一，图中的每一列对应于类的一个实例属性），而每一个新的实例将 是跨越所有数组列的一个切片（图中的每一个被使用的行对应于类的一个实例）。每次需要实例化一个新的对象，new 函数将被调用。一个新的逻辑行将被分配，新的实例的实例属性将以新的行偏移量插入到相应的属性列当中去。

图 **1.** 基于数组方法的面向对象编程实现



虽然在 CPAN 上有许多基于这一方法的实现，为了更加清楚地说明如何实现基于数组存储属性的面向对象编程，我们自己动手实现了一个简单的实例。我们定义了一个 InsideOut 类（模块），所有的需要使用基于数组存储属性的面向对象编程的类必须继承这个类。InsideOut 通过为每个包维护一个称做为 @_free 的"空余行列表"来重用那些被定义之后又被释放的行（空余行）。通过精心设计的数据结构，这个列表成为了一个包含所有空余行信息 的链表，并且通过一个名为 $_free 的变量变量指向链表的头部。表中的每个元素包含了下一个空余行的索引。当一个对象的实例被删除时，$_free 将指向这个被释放的行，而空余列表中相应的这个行中的元素将含有指向原有 $_free 所指向的前一个条目。因为被释放的"所谓"空余行和被使用的行不会重叠，所以我们可以自己的使用其中的一个属性列来保存 @_free 。这是通过 typelogb 别名机制来实现的。

我们设计的 InsideOut 模块为一个继承它的类提供如下的功能：

一个名为 new 的构造函数，负责将为 bless 到继承类中的对象分配空间。 new 函数将会自动地调用 initialize，而 initialize 可以在继承它的类中被重载，进行用户自己定义的初始化工作。

我们将定义一组访问函数，用于存取属性。这是一组已 get_attribute 和 set_attribute 为名称的方法，将在继承类被自动创建，包括对象自己的方法，任何人只能通过这些方法来存取对象属性。由于 InsideOut 模块是唯一知道如何存取属性的模块，所以用户无法通过除此之外的任何方法来存取对象的实例属性。

一个名为 DESTROY 的析构函数。

InsideOut 模块的具体实现如下，见清单 7 到清单 11 。例七部分包含了 InsideOut 模块的对外接口函数。继承 InsideOut 模块的类通过调用它提供的 define_attributes 函数，自动生成自己类的构造函数和实例属性访问函数。

清单 **7. InsideOut** 模块的对外接口函数 **define_attributes**

```
package InsideOut;
 require Exporter;

 @InsideOut::ISA = qw (Exporter);
 @InsideOut::EXPORT = qw (define_attributes);

 sub define_attributes {
 my $package = caller;
 @{"${package}::_ATTRIBUTES_"} = @_;

 my $code = "";
 foreach my $attribute ( get_attribute_names($package) ) {
 @{"${package}::_$attribute"} = ();
 unless ( $package->can("get_${attribute}") ) {
 $code = $code . _define_get_accessor ($package, $attribute);
 }
 unless ( $package->can("set_${attribute}") ) {
 $code = $code . _define_set_accessor ($package, $attribute);
 }
 }
 $code .= _define_constructor ($package);
 eval $code;

 if ($@) {
 print $code . "\n";
 die "ERROR: Unable to define constructor and accessor for $package \n" ;
```

```
 }
 }
```

清单 8 定义了内部函数 _define_get_accessor 和 _define_set_accessor，分别负责自动生成实例属性的存取方法。清单 9 定义了内部函数 _define_constructor，这个函数负责生成继承与 InsideOut 模块的类的构造函数 new () 。例十是一个由 InsideOut 模块自动生成的代码的清单。

清单 **8.** 负责自动生成存取实例属性方法的代码片断

```
sub _define_get_accessor {
 my ($package, $attribute) = @_;
 my $code = qq {
 package $package;
 sub get_${attribute} {
 return \$_${attribute}\[\${\$_[0]}]
 }
 if ( !defined ( \$_free ) ) {
 \*_free = \*_$attribute;
 \$_free = 0;
 }
 };
 return $code;
 }

 sub _define_set_accessor {
 my ($package, $attribute) = @_;
 my $code = qq {
 package $package;
 sub set_${attribute} {
 if ( scalar (\@_) > 1 ) {
 \$_${attribute}\[\${\$_[0]}] = \$_[1];
 }
 }
 };
 return $code;
 }
```

清单 **9.** 自动生成构造函数的代码片断

```
sub _define_constructor {
 my $package = shift;
 my $code = qq {
 package $package;
 sub new {
 my \$class = shift;
 my \$id;
 if ( defined (\$_free[\$_free]) ) {
 \$id = \$_free;
 \$_free = \$_free[\$_free];
 undef \$_free[\$_id];
 } else {
 \$id = \$_free++;
 }
 my \$object = bless \\\$id, \$class;
 if ( \@_ ) {
 \$object->set_attributes (\@_)
 }
 \$object->initialize();
 return \$object;
 }
 };
 return $code;
 }
```

我们继承 InsideOut 模块并且定义一个名为 People 的对象，如清单 10 所示。看看 InsideOut 模块如何为我们自动生成实例属性访问函数和 People 对象的构造函数 new () 。

清单 **10.** 使用 **InsideOut** 模块创建自己的对象

```
package People;
 use InsideOut;
 @ISA = qw (InsideOut);
 define_attributes qw (name age);

 $object_people = People->new ( “ name ” => “ Tonny ” ,
```

```
                              “ age ” => 28 );
print “ Name : ” . $object_ people->get_name () . “ , Age : ” .
$object_people->get_age () . “ . \n ” ;
```

清单 **11.** 自动生成的代码片断

```
package People;
 sub get_name {
 return $_name[${$_[0]}]
 }
 if ( !defined ( $_free ) ) {
 *_free = *_name;
 $_free = 0;
 }

 package People;
 sub set_name {
 if ( scalar (@_) > 1 ) {
 $_name[${$_[0]}] = $_[1];
 }
 }

 package People;
 sub new {
 my $class = shift;
 my $id;
 if ( defined ($_free[$_free]) ) {
 $id = $_free;
 $_free = $_free[$_free];
   undef $_free[$_id];
 } else {
 $id = $_free++;
 }
 my $object = bless \$id, $class;
 if ( @_ ) {
 $object->set_attributes (@_)
 }
 $object->initialize();
 return $object;
 }
```

在清单 10 中，我们定义了两个实例属性，name 和 age 。在 People 类的定义中，函数 define_attributes（）被调用，自动生成了例十一中所显示的构造函数 new（）和实例属性访问函数 set_name（），get_name（）和没有被放在例十一中的 set_age（），get_age（）。define_attributes（）函数首先调用内部函数 get_attribute_names（），这个函数将递归操作包的 @ISA 数组中包含的模块和其本身的 _ATTRIBUTES_ 数组，来获取这个类在整个继承链中的所有实例属性的名称并且以一个数组的形式返回。 define_attributes（） 函数将会为每一个实例属性初始化一个数组。在 Perl 中所有模块都隐含地继承了一个被称做为 UNIVERSAL 的内建模块，这个模块将自动为 InsideOut 模块提供 can（函数名）的方法。如果一个类或者它的任何基类包含有 can 中设定的函数名的函数，那么 can 方法将返回一个 true 的值。 define_attributes（）函数将检查继承 InsideOut 模块的类和它的基类中是否已定义了 get_$attribute（）和 set_$attribute（），没有就自动为这个 $attribute 的实例属性生成一个存取方法。这样的设计提供了让用户在自己的类定义模块简单地重载这些存取方法的接口。在此之 后，define_attributes（）函数调用了内部函数 _define_constructor（），为用户定义的类生成构造函数 new（）。

在内部函数 _define_constructor（）中，变量 $code 纪录了自动生成的构造函数的代码。在 qq 函数包含的结构内，构造函数 new（）最后的返回实质上就是一个指向属性数组行的索引的引用而已。每次 new（）韩树被调用，我们将在 @_free 数组中找到一个空余行的索引，然后将要返回的那个引用指向的标量置为这个空余行的索引。如果没有空余行的存在，则在属性数组的后面加上一行，用于存储新建 实例的实例属性。然后调用内部函数 set_attributes（），为已经分配了存储空间的实例属性按用户输入的数据赋值。最后调用函数 initialize（），这个函数可以在用户类中被改写，用于完成用户自己订制的初始化工作。

其余在 InsideOut 模块中被定义的函数见清单 12 到清单 14，清单 12 中的 get_attribute_names（）函数在上文中已经讨论过了，主要返回一个对象所有的实例属性。

清单 **12. get_attribute_names** 函数

```
sub get_attribute_names {
 my $package = shift;
 if ( ref ($package) ) {
 $package = ref ($package);
 }
 my @result = @{"${package}::_ATTRIBUTES_"};
 if ( defined ( @{"${package}::ISA"} ) ) {
 foreach my $base_package (@{"${package}::ISA"}) {
 push ( @result, get_attribute_names ($base_package) );
 }
 }
 return @result;
```

```
}
```

清单 **13. set_attributes** 和 **get_attribute** 函数

```
sub set_attributes {
 my $object = shift;
 my $attribute_name;
 if ( ref ($_[0] ) ) {
 my ($attribute_name_list, $attribute_value_list) = @_;
 my $i = 0;
 foreach $attribute_name (@{$attribute_name_list}) {
 my $set_method_name = "set_" . $attribute_name;
 $object->$set_method_name ($attribute_value_list->[$i++]);
 }
 } else {
 my ($attribute_name, $attribute_value);
 while (@_) {
 $attribute_name = shift;
 $attribute_value = shift;
 my $set_method_name = "set_" . $attribute_name;
 $object->$set_method_name ($attribute_value);
 }
 }
 }

 sub get_attributes {
 my $object = shift;
 my (@retval);
 foreach $attribute_name (@_) {
 my $get_method_name = "get_" . $attribute_name;
 push ( @retval, $object->$get_method_name() );
 }
 return @retval;
 }
```

清单 14 中定义了析构函数 DESTROY（）和初始化函数 initialize（）。初始化函数 initialize（）不做任何事情，只是对继承 InsideOut 模块的类提供了一个可以重载的方法用于定制用户需要的初始化工作。析构函数 DESTROY（）释放与对象相关的所有属性值，并将在实例属性数组中与该对象相关的行中的所有属性元素标记为 undef 。最后将实例所占用的 id 号释放回空余列表中去。

清单 **14.** 初始化函数和析构函数

```
sub initialize {
 }

 sub DESTROY {
 my $object = shift;
 my $package = ref ($object);
 local *_free = *{"{$package}::_free"};
 my $id = $$object;
 local (@attributes) = get_attribute_names ($package);
 foreach my $attribute (@attributes) {
 undef ${"${package}::_$attribute"}[$id];
 }
 $_free[$id] = $_free;
 $_free = $id;
 }
```

回页首

基于数组的方法中的继承

基于数组的方法中的继承与基于匿名哈希表的方法中的继承完全一样。我们设计的 InsideOut 类中利用 @ISA 数组提供了对继承的支持。

回页首

总结

相比于基于匿名哈希表的方法，基于数组的方法对存取属性的访问提供了更好的控制和保护并且实现了对于对象的封装，同时也提高了存储空间的利用 效率。但是基于匿名哈希表的方法也有着简单易学，逻辑上较为直观而且无需要第三方模块支持的优点。具体使用哪种方式实现面向对象的设计，还要在工作中根据 实际情况进行考虑才对。

参考资料

- O'Reilly perl.com Web site 一些 O'Reilly 关于 Perl 的网上教程，书籍和社区。
- use Perl 一个发布关于 Perl 的最新消息的站点。
- CPAN 一个发布第三方的 Perl 模块的站点。。
- Yet Another Perl Object Model (Inside Out Objects) 一篇讨论 Perl 中基于数组的面向对象编程实现的好文章。
- Threads and fork and CLONE 一篇讨论 Perl 中对于多线程支持的好文章。
- Introduction to Inside-Out Objects 另外一篇讨论 Perl 基于数组的面向对象编程实现的好文章。
- 在 developerWorks Linux 专区 寻找为 Linux 开发人员（包括 Linux 新手入门）准备的更多参考资料，查阅我们 最受欢迎的文章和教程。
- 在 developerWorks 上查阅所有 Linux 技巧 和 Linux 教程。

关于作者

本人就职于 IBM 中国，负责 IBM 存储设备与第三方存储管理软件和数据库软件的兼容性测试工作。同时还负责自动化脚本的开发工作，擅长的编程语言包括 Perl 和 Python 。

为本文评分

评论