Perl语言高级编程专题
Lesson 13

周晓方
courses@xfzhou.homeftp.org

Class & Object in Perl

- To understand *Perl-OOP*, you must understand *perl-ref*.
- These on-line documents are talking about Perl-OOP:
  - *perlboot*: tutorial from a very special point of view.
  - *perltoot*: Tom's object-oriented tutorial for Perl, Must read.
  - *perltootc*: Tom's words on Class Data in Perl
  - *perlobj*: The official document about Perl-Objects
  - *perlbot*: Bag'o Object Tricks
  - *perltie*: Tie variables with objects, very interesting.
- Class is a namespace (*i.e.* a package) of data & functions.
- A function in a Class is usually called a '***METHOD***'.
- Perl searchs for a method hierachically in ***@ISA*** tree, track back to a hidden base class called '***UNIVERSAL***'. Turns to '***AUTOLOAD***' method if none's found.
- Object is an instance of Class. An *obj.* belongs to a *Class*.

Constructor & Destructor

- Constructor can be any name, usually '*new*'.
- Constructor returns a '*bless*'ed hash-ref.
- Destructor is always named 'DESTROY'.
  Perl calls DESTORY automatically when needed.
- Destructor function is usually not necessary at all.

Comp.pm

```
package Comp;
use vars qw($id);
$id = 0;
sub new($$) {
      my($type) = shift; # The hidden argument 'Comp'
      my($x, $y) = @_;   # Two given argument
      $x = 0 if not $x; $y = 0 if not $y; $id++;
      my($this) =        # This is the obj data
            {x=>$x, y=>$y, id=>$id};
      print "Got a new #$id :\t($x, $y).\n";
      bless $this, $type;# Must bless with 'Comp'
}
# sub DESTROY {……} # 十有八九不必写什么DESTROY函数
1;                 # 模块最后必须返回一个非零值
```

Improve our *Constructor*

- How to invoke a constructor?
  $obj = class`->`*new* `(`*arguments…*`)`
- What perl actually do?
  $obj = class`::`*new* `('`*class*`',` *argument…*`)`
- Call constructor like an object method
  We write: *$objB = $objA*`->`*new* `(`*arguments…*`)` ➔
  Perl does: *$objB = class*`::`*new* `(`*$objA*`,` *arguments…*`)`
- ref($obj) returns class name. Let's improve constructor:

Improved Comp.pm

```
…sub new { my $proto = shift;
      my $type = ref($proto) || $proto;
… … … bless $this, $type;      … … …
```

the main-code.pl

```
use Comp;
$a = Comp->new();          # ok
$b = Comp::new();          # bad, missing class name.
$c = Comp::new('Comp'); # Not perl style. Don't do that
$d = $a->new();            # use with improved constructor
1;
```

Object Data

- Object data returned by an constructor must be a blessed reference. It's usually a ***hash-ref***. But also can be a ***array-ref*** (e.g. *TK*::*After*), a ***code-ref*** (very tricky, see examples in *perltoot*), or even an simple ***scalar-ref***. See example code *101~104*.

Array-ref Constructor

```
…my $this=[$x, $y, $id];
 bless $this, $type; … … …
```

Scalar-ref Constructor

```
…my $this= \"$x:$y:$id";
 bless $this, $type; … … …
```

Code-ref Constructor

```
…my $data={x=>$x,y=>$y, id=>$id};
 my $this = sub {
    my $field = shift;
    if (@_){$data->{$field}=shift}
    return $data->{$field};
 }; #对象的数据对外完全不可见,很诡异的
 bless $this, $type;
```

Array-ref data access

```
…my $this=shift;
 my $id=$this->[2];…
```

Scalar-ref data acc

```
…my $this=shift;
 my ($x, $y, $id) =
   split ':', $$this;…
```

Code-ref data access

```
…my $this=shift;
 my $id=$this->('id');
… … …
```

Set/Get method

- Let user modify the object data through a *set*/*get* method. Avoid any directly *access*/*modification* to object data.
- For each $obj→{*key*}, write a method with the same name of *key*, that return the data value, and optionally take an argument as the new value to be set.

```
print $a->x; #get
$a->y(5);    #set
```

- Most set/get method looks almost the same.

Still in CompE.pm

```
…sub x {
   my $this = shift;
   $this->{x} = shift if (@_);
   $this->{x};
}

sub y {
   my $this = shift;
   $this->{y} = shift if (@_);
   $this->{y};
}

sub id {
   my $this = shift;
   $this->{id} = shift
        if (@_ and $_[0] > $id);
   $this->{id};
} … … …
```

Proxy Method: *AUTOLOAD*

- Most Get/Set are similar, lets use ***AUTOLOAD***!
- Full qualified method name in an *'our'* var *$AUTOLOAD*
- Whenever user calls a method that Perl can't find it anywhere, it will put the method name in *$AUTOLOAD* and call the ***AUTOLOAD*** method. If the ***AUTOLOAD*** method is not defined, perl will complain and fail.

in CompF.pm, replace method x, y, id with AUTOLOAD

```
…
our $AUTOLOAD;
sub AUTOLOAD {
      my $this = shift;
      my $name = $AUTOLOAD;
      $name =~ s!.*::!!; # remove package_name::
      return undef unless exists $this->{$name};
      $this->{$name} = shift if (@_);
      print "AUTOLOAD $name method successful!\n";
      $this->{$name};
}
```

Class Data

- Class data is the common data to all objects of a class, e.g., the *'$id'* counter in ***Comp.pm***.
- The *'$id'* in ***Comp.pm*** is global, can be accessed from outside world like this: *$Comp::id*. We must avoid this.
- To avoid this, declare *$id* as **my (*$id*)**. But direct access to a class data is still buggy, esp, when doing inheritance.

CompE.pm

```
package CompE;
my $id = 0;
my $count = 0;
sub new {
       my $proto = shift;
       ……my($this) = {x=>$x, y=>$y, id=>$id, _COUNT=>\$count};
       ${$this->{_COUNT}}++;       bless $this, $type;
}
sub DESTROY {my($this) = shift;  my($id) = $this->{id};
       my($count) = --${$this->{_COUNT}};
       print  "Let's destroy #$id :\t$this. " .
              "We's $count number of " . ref($this) .
              " left.\n"; 1;
}
1;
```

Solution other than *AUTOLOAD*?

- Yes, Get/Set are similar. Let's use a '**foreach**' loop '**generate**' these methods when module was loaded, instead of a run-time '***AUTOLOAD'*** search!
- Be aware of the **no strict "refs"** program, since we are using a symbolic-ref.

in CompG.pm, replace AUTOLOAD method with a loop

```
…
for my $field qw(x y id) {
      no strict "refs"; # turn of symbolic-ref check
      # let's generate method on-the-fly
      *$field = sub {
            my $this = shift;
            $this->{$field} = shift if @_;
            return $this->{$field};
      }
}
…
```

- After that, you can call get/set method '*x*', '*y*', '*id*' as well.

@ISA and inheritance in Perl

- 'Inheritance': child class gets all the method from parents classes for free. Child class usually has something new.
- 'Overload': child class redefine and override a parents method. But still got a chance to call the original one.
- 'Multiple Inheritance': one child chass has more than one parents, and even many grand parents… When using *Multiple Inheritance*, use 'SUPER::" to refer to the correct base class.
- '*UNIVERSAL*' is the root of all classes in perl.
  - ***isa*** method: `$CompH->isa('Vector')`
  - ***can*** method: `$CompH->can('add')`

Example: Inheritance & Overload

Vector.pm 基类

```
package Vector;
……基类,定义new, plus, minus, neg, x, y等方法
sub string { my $this = shift;
                    '('. $this->x . ', ' . $this->y .
')';} ……
sub compare { # which one far from zero dot
       my $this = shift;
       my $z = shift;
      ?$this->abs <=> $z->abs;
}
sub abs { warn "Abstract method\n"; 0; }
……
```

CompH.pm 继承类

```
package CompH;
use strict;
use Vector;
our @ISA = qw(Vector);
sub abs {
  my $this = shift;
  return
   $this->x * $this->x
 + $this->y * $this->y;
}
1;
```

CompH.pl 主程序

```
use strict;
use CompH;
my @l = map CompH->new(int rand 10,
       int rand 10), 1..10;
my @s = sort {$a->compare($b)} @l;

print "\@l is :", join(",",
       map $_->string, @l), "\n";
print "\@s is :", join(",",
       map $_->string, @s), "\n";
1;
```

'SUPER' and Overload

- Overload 'new' and make a grid-comples-number class.

CompGrid.pm 继承类

```
package CompGrid;
use strict;
use Vector;
our @ISA = qw(Vector);

sub new {
        my $proto = shift;
        my $class = ref $proto || $proto;
        my $this = $class->SUPER::new(@_);
        #don't say $class->Vector::new(@_);
        #Hard-code is always a bad habit
        $this->x(int $this->x);
        $this->y(int $this->y);
        bless ($this, $class);      # reconsecrate
}

sub abs {
        my $this = shift;
        return abs $this->x + abs $this->y;
}

1;
```

Operator overload —— *use overload*

- use ***overload*** module to overload perl operators!!!

CompWork.pm 继承类

```
package CompWork;
use strict;
use Vector;
our @ISA = qw(Vector);
use overload
       '+=' => 'plus',
       '"""' => 'string';
sub abs {
       ……
}
1;
```

CompWork.pl 主程序

```
#!/usr/bin/perl -w
use strict;
use CompWork;

my @l = map CompWork->new(
  int rand 10, int rand 10), 1..10;
my $s = CompWork->new();
$s += $_ foreach @l;
print "$s";

1;
```

- 'op' => 'obj_method', 'op' => &package_function
- All perl-ops can be overloaded. (see *overload*)

```
'+', '-', '*', …   '+=', '-=', '*=', … '<', 'eq', '<=>', …
'bool', '"""', '0+', 'abs', 'sin', … '${}', '%{}', '<>', …
'nomethod', 'fallback', '=',…
```

- Study '**overload**' manual page carefully before start

Tied variables

- Perl打破了计算机语言“love me, love my built-in semantics!”的惯例，最基本类型(标量、向量、散列)的行为都可以重新定义
- Tie采用OO方法，使被Tie变量的外表(运算符)不变，但行为(运算所实现的具体操作)得到重新定义，相当于给变量换脑，例如：
  - Tie::Watch　　可以监控变量的赋值等
  - 将散列Tie到数据库　　速度换内存的方法
  - 可以创建创建健值大小写无关的散列等
  - Tie一个文件句柄　　创建虚拟文件
- Tie的两个缺点：
  - 性能下降
  - Perl5.004之前不支持Tie Handle

Tieing a scalar实例，随机变量

- 产生一个随机变量，每次读出时给出一个新的随机数，[$a, $b)均匀分布；写入时给srand一个种子

TieRand.pm类

```
package TieRand;

sub TIESCALAR {
    my ($class, $a, $b) = @_;
    ($a, $b) = (-1, 1)
        unless defined $a and
            defined $b and
            $a < $b;
    bless [$b - $a, $a], $class;
}

sub FETCH {
    my ($width, $offset) = @{$_[0]};
    return $offset + rand $width;
}

sub STORE {
    my ($obj, $seed) = @_;
    srand $seed;
}
1; #这个例子无须写DESTROY析构函数
```

tb_Rand.pl测试代码

```
#!/usr/bin/perl -w
use strict;
use TieRand;
my $r;
tie $r, "TieRand", -100, 100;
$r = 6;
print "$r\n" for 0..10;
1;
```

测试结果

```
C:\>tb_Rand.pl
-59.271240234375
-3.900146484375
-84.197998046875
-10.14404296875
61.6943359375
73.126220703125
-92.340087890625
32.110595703125
-64.8193359375
51.0986328125
61.57836914062 5
```

Tieing a hash table

- TIEHASH(类,其他参数)　　创建时，返回对象
- FETCH(对象,key)　　读取时，返回value
- STORE(对象,key,val) 写入时，无返回
- EXITS(对象,key)　　exists函数，返回真值
- DELETE(对象,key)　delete函数，可无返回值
- CLEAR(对象)，`%hash=()`，无返回值
- FIRSTKEY(对象), each/keys/values, 返回1st key
- NESTKEY(对象，lastkey)，返回nextkey/undef
- DESTORY(对象), 析构函数，无返回值

Tieing an Array

- TIEARRAY/DESTORY/FETCH/STORE/ CLEAR 和前面的类似
- FETCHSIZE(对象), 返回整数值
- STORESIZE (对象,大小), $#arr=5,无返回
- EXTEND((对象,大小), 内部调用, 无返回
- PUSH/UNSHIFT(对象,list),不返回
- POP/SHIFT(对象),返回一个标量
- SPLICE(对象,offset,number,@insertlist)，返回被删除的那部分list

Tieing a file handle

- TIEHANDLE/DESTROY are similar
- WRITE/READ, when call to syswrite/sysread
- PRINT/PRINTF, print/printf, 返回真值
- GETC, getc, return next char or undef
- READLIN, <F>, return next line or undef
- CLOSE, close, 返回真值
- 视Perl版本而定，Tie文件句柄还定义了OPEN, EOF, FILENO, SEEK, TELL等对应方法。

TieTee: 多通句柄

- 同时输出到多个句柄

TieTee.pm类

```
package TieTee;

sub TIEHANDLE {
    my ($class, @handles) = @_;
    bless [@handles], $class;
}

sub PRINT {
    my (@handles) = @{shift @_};
    print $_ @_ foreach @handles;
}

1;
```

tb_Tee.pl测试代码

```
#!/usr/bin/perl -w
use strict;
use TieTee;

open LOG, ">log.txt";
tie *F, 'TieTee', \*STDOUT, \*STDERR, \*LOG;
select F;
print "Hello!\n";
print "I'm writing to 3 files at once!\n";
untie *F;
close LOG;
1;
```

运行结果

```
C:\>tb_Tee.pl
Hello!
Hello!
I'm writing to 3 files at once!
I'm writing to 3 files at once!

C:\>type log.txt
Hello!
I'm writing to 3 files at once!

C:\>
```

更多的PerlObj

- Damian Conway, "*Object Oriented Perl*", Manning Publications Co. 2000
- Simon Cozens, "*Advanced Perl Programming, 2nd Edition*", O'Reilly 2005, Chap 4
- Damian Conway, "*Perl Best Practices*", O'Reilly 2005, Chap 15-16
- E.S. Peschko, Michele DeWolfe, "*Perl 5 Complete*", McGraw-Hill 1998, Chap 13-20
- P. Fenwich, J. Richardson, "Object Orientied Perl", Perl Training Australia 2007
- Study those Tie::forbar modules

overload运算符重载的进一步考虑

- 自返类运算改变对象自身：**$oa *= 100**
- 单、双目运算不能改变自身，应返回新对象
- '-'=>\&minus后,减法怎么对应**minus(•)**？
  - **$oa-$ob → minus($oa, $ob, '')**
  - **$oa-2 → minus($oa, 2, '')**
  - **2-$oa → minus($oa, 2, 1)**
- $oc = $oa到底干了什么：简单复制了引用
  - perl的赋值运算符是无法重载的，'='重载的是复制构造函数
  - **$oc = $oa; $oc++;**
  - 上例的++自增之前，'='被自动调用，使得**$oc**和**$oa**分离
- 细看perldoc overload及BigInt.pm的写法

Complex_1234.pm 类

```
……
use overload '-'=>\&minus, '-='=>'selfminus', '='=>'clone',……;
sub clone {
   my $self = shift;
   bless {r => $self->{r}, i => $self->{i}}, ref $self;
}

sub selfminus {
   my ($self, $obj) = @_;
   if (ref $obj) {
        $self->{r} -= $obj->{r};
        $self->{i} -= $obj->{i};
   } else {     # $obj is a number
        $self->{r} -= $obj;
   }
   $self;
}
sub minus {
   my ($self, $obj, $mutate) = @_;
   my $c;
   unless ($mutate) {# $self-$obj
        $c = $self->clone;
        $c->selfminus($obj);
   } else {
        $c = ref $obj ? # $obj - $self
                $c = $obj->clone :     # $obj is an object
                $self->new($obj, 0);   # $obj is a number
        $c->selfminus($self);
   }
   return $c;
}
……
```

1234.pl 主程序

```
#!/usr/bin/perl -w
use strict;
use lib '.';
use aliased
  'Complex_1234' => 'Complex';
my $a = Complex->new(15, -5);
my $b = Complex->new( 5, 3);
print "$a - 2 = ", $a - 2, "\n";
print "2 - $a = ", 2 - $a, "\n";
my $c = $a;
$c -= $b - (-5);
print "$a, $b, $c.";
1;
```

回家作业12-perlobj作业

- Complex_学号.pm，写一个支持复数运算的Perl对象，重载+ - * / "" = ==。共扼conj()方法的算符是~。abs是复数的模
- 实部虚部的set/get函数分别是r()和i()
- ""写成(实部, j*虚部)、(实部, -j*虚部)
- 主程序 学号-12.pl
  - use lib '.';
  - 测试模块Complex_学号.pm的各种功能。
  - 复数没有序，无法定义比较运算，但可以用sort {abs($a)<=>abs($b)} @comp_arr对一个复数数组按照幅度来排序。