第十二章 Perl5中的引用/指针

by flamephoenix

一、引用简介

引用就是指针，可以指向变量、数组、哈希表（也叫关联数组）甚至子程序。Pascal或C程序员应该对引用（即指针）的概念很熟悉，引用就是某值的地址，对其的使用则取决于程序员和语言的规定。在Perl中，可以把引用称为指针，二者是通用的，无差别的。引用在创建复杂数据方面十分有用。

Perl5中的两种引用类型为硬引用和符号引用。符号引用含有变量的名字，它对运行时创建变量名并定位很有用，基本上，符号引用就象文件名或UNIX系统中的软链接。而硬引用则象文件系统中的硬链接。

Perl4只允许符号引用，给使用造成一些困难。例如，只允许通过名字对包的符号名哈希表（名为_main{}）建立索引。Perl5则允许数据的硬引用，方便多了。

硬引用跟踪引用的计数，当其数为零时，Perl自动将被引用的项目释放，如果该项目是对象，则析构释放到内存池中。Perl本身就是个面向对象的语言，因为Perl中的任何东西都是对象，包和模块使得对象更易于使用。

简单变量的硬引用很简单，对于非简单变量的引用，你必须显式地解除引用并告诉其应如何做，详见《第章Perl中的面向对象编程》。

二、使用引用

本章中，简单变量指像$pointer这样的变量，$pointer仅含一个数据项，其可以为数字、字符串或地址。

任何简单变量均可保存硬引用。因为数组和哈希表含有多个简单变量，所以可以建立多种组合而成的复杂的数据结构，如数组的数组、哈希表的数组、子程序的哈希表等等。只要你理解其实只是在用简单变量在工作，就应该可以正确的在最复杂的结构中正确地解除引用。

首先来看一些基本要点。

如果$pointer的值为一个数组的指针，则通过形式@$pointer来访问数组中的元素。形式@$pointer的意义为“取出$pointer中的地址值当作数组使用”。类似的，%$pointer为指向哈希表中第一个元素的引用。

有多种构建引用的方法，几乎可以对任何数据建立引用，如数组、简单变量、子程序、文件句柄，以及--C程序员会感兴趣的--引用。Perl使你有能力写出把自己都搞糊涂的极其复杂的代码。:)

下面看看Perl中创建和使用引用的方法。

三、使用反斜线(\)操作符

反斜线操作符与C语言中传递地址的操作符&功能类似。一般是用\创建变量又一个新的引用。下面为创建简单变量的引用的例子:

```
$variavle = 22;
$pointer = \$variable;
$ice = "jello";
$iceprt = \$ice;
```

引用$pointer指向存有$variable值的位置，引用$iceptr指向"jello"。即使最初的引用$variable销毁了，仍然可以通过$pointer访问该值，这是一个硬引用，所以必须同时销毁$pointer和$variable以便该空间释放到内存池中。

在上面的例子中，引用变量$pointer存的是$variable的地址，而不是值本身，要获得值，形式为两个$符号，如下:

```
                #!/usr/bin/perl
    $value = 10;
```

```
$pointer = \$value;
printf "\n Pointer Address $pointer of $value \n";
printf "\n What Pointer *($pointer) points to $$pointer\n";
```

结果输出如下:

```
Pointer Address SCALAR(0x806c520) of 10
What Pointer *(SCALAR(0x806c520)) points to 10
```

每次运行，输出结果中的地址会有所改变，但可以看到$pointer给出地址，而$$pointer给出$variable的值。

看一下地址的显示，SCALAR后面一串十六进制，SCALAR说明该地址指向简单变量（即标量），后面的数字是实际存贮值的地址。

注意：指针就是地址，通过指针可以访问该地址处存贮的数据。如果指针指向了无效的地址，就会得到不正确的数据。通常情况下，Perl会返回NULL值，但不该依赖于此，一定要在程序中把所有的指针正确地初始化，指向有效的数据项。

四、引用和数组

关于Perl语言应该记住的最重要的一点可能是：Perl中的数组和哈希表始终是一维的。因此，数组和哈希表只保存标量值，不直接存贮数组或其它的复杂数据结构。数组的成员要么是数（或字符串）要么是引用。

对数组和哈希表可以象对简单变量一样使用反斜线操作符，数组的引用如下:

```
1  #!/usr/bin/perl
2  #
3  # Using Array references
4  #
5  $pointer = \@ARGV;
6  printf "\n Pointer Address of ARGV = $pointer\n";
7  $i = scalar(@$pointer);
8  printf "\n Number of arguments : $i \n";
9  $i = 0;
10 foreach (@$pointer) {
11   printf "$i : $$pointer[$i++]; \n";
12 }
```

运行结果如下:

```
$ test 1 2 3 4
Pointer Address of ARGV = ARRAY(0x806c378)
Number of arguments : 4
0 : 1;
1 : 2;
2 : 3;
3 : 4;
```

第5行将引用$pointer指向数组@ARGV，第6行输出ARGV的地址。$pointer返回数组第一个元素的地址，这与C语言中的数组指针是类似的。第7行调用函数scalar()获得数组的元素个数，该参数亦可为@ARGV，但用指针则必须用@$pointer的形式指定其类型为数组，$pointer给出地址，@符号说明传递的地址为数组的第一个元素的地址。第10行与第7行类似，第11行用形式$$pointer[$i]列出所有元素。

对关联数组使用反斜线操作符的方法是一样的--把所有关联数组名换成引用$poniter。注意数组和简单变量（标量）的引用显示时均带有类型--ARRAY和SCALAR，哈希表（关联数组）和函数也一样，分别为HASH和CODE。下面是哈希表的引用的例子。

```
#!/usr/bin/perl
1  #
2  # Using Associative Array references
3  #
4  %month = (
5   '01', 'Jan',
6   '02', 'Feb',
```

```
7  '03', 'Mar',
8  '04', 'Apr',
9  '05', 'May',
10 '06', 'Jun',
11 '07', 'Jul',
12 '08', 'Aug',
13 '09', 'Sep',
14 '10', 'Oct',
15 '11', 'Nov',
16 '12', 'Dec',
17 );
18
19 $pointer = \%month;
20
21 printf "\n Address of hash = $pointer\n ";
22
23 #
24 # The following lines would be used to print out the
25 # contents of the associative array if %month was used.
26 #
27 # foreach $i (sort keys %month) {
28 # printf "\n $i $$pointer{$i} ";
29 # }
30
31 #
32 # The reference to the associative array via $pointer
33 #
34 foreach $i (sort keys %$pointer) {
35   printf "$i is $$pointer{$i} \n";
36 }
```

结果输出如下:

```
              $ mth
Address of hash = HASH(0x806c52c)
01 is Jan
02 is Feb
03 is Mar
04 is Apr
05 is May
06 is Jun
07 is Jul
08 is Aug
09 is Sep
10 is Oct
11 is Nov
12 is Dec
```

与数组类似，通过引用访问哈希表的元素形式为$$pointer{$index}，当然，$index是哈希表的键值，而不仅是数字。还有几种访问形式，此外，构建哈希表还可以用=>操作符，可读性更好些。下面再看一个例子:

```
              1  #!/usr/bin/perl
2  #
3  # Using Array references
```

```
4 #
5 %weekday = (
6   '01' => 'Mon',
7   '02' => 'Tue',
8   '03' => 'Wed',
9   '04' => 'Thu',
10  '05' => 'Fri',
11  '06' => 'Sat',
12  '07' => 'Sun',
13  );
14 $pointer = \%weekday;
15 $i = '05';
16 printf "\n ================== start test ================= \n";
17 #
18 # These next two lines should show an output
19 #
20  printf '$$pointer{$i} is ';
21  printf "$$pointer{$i} \n";
22  printf '${$pointer}{$i} is ';
23  printf "${$pointer}{$i} \n";
24  printf '$pointer->{$i} is ';
25
26  printf "$pointer->{$i}\n";
27 #
28 # These next two lines should not show anything 29 #
30  printf '${$pointer{$i}} is ';
31  printf "${$pointer{$i}} \n";
32  printf '${$pointer->{$i}} is ';
33  printf "${$pointer->{$i}}";
34 printf "\n ================== end of test ================= \n";
35
```

结果输出如下:

```
                ================== start test =================
$$pointer{$i} is Fri
${$pointer}{$i} is Fri
$pointer->{$i} is Fri
${$pointer{$i}} is
${$pointer->{$i}} is
================== end of test =================
```

可以看到，前三种形式的输出显示了预期的结果，而后两种则没有。当你不清楚是否正确时，就输出结果看看。在Perl中，有不明确的代码就用print语句输出来实验一下，这能使你清楚Perl是怎样解释你的代码的。

五、多维数组

语句@array = list;可以创建数组的引用，中括号可以创建匿名数组的引用。下面语句为用于画图的三维数组的例子:

```
$line = ['solid' , 'black' , ['1','2','3'] , ['4','5','6']];
```

此语句建立了一个含四个元素的三维数组，变量$line指向该数组。前两个元素是标量，存贮线条的类型和颜色，后两个元素是匿名数组的引用，存贮线条的起点和终点。访问其元素语法如下:

```
                $arrayReference->[$index]     single-dimensional array
$arrayReference->[$index1][$index2]   two-dimensional array
$arrayReference->[$index1][$index2][$index3] three-dimensional array
```

可以创建在你的智力、设计经验和计算机的内存允许的情况下极尽复杂的结构，但最好对可能读到或管理你的代码的人友好一些--尽量使代码简单些。另一方面，如果你想向别人炫耀你的编程能力，Perl给你足够的机会和能力编写连自己都难免糊涂的代码。:)

建议：当你想使用多于三维的数组时，最好考虑使用其它数据结构来简化代码。

下面为创建和使用二维数组的例子：

```
1  #!/usr/bin/perl
2  #
3  # Using Multi-dimensional Array references
4  #
5  $line = ['solid', 'black', ['1','2','3'] , ['4', '5', '6']];
6  print "\$line->[0] = $line->[0] \n";
7  print "\$line->[1] = $line->[1] \n";
8  print "\$line->[2][0] = $line->[2][0] \n";
9  print "\$line->[2][1] = $line->[2][1] \n";
10 print "\$line->[2][2] = $line->[2][2] \n";
11 print "\$line->[3][0] = $line->[3][0] \n";
12 print "\$line->[3][1] = $line->[3][1] \n";
13 print "\$line->[3][2] = $line->[3][2] \n";
14 print "\n"; # The obligatory output beautifier.
```

结果输出如下：

```
$line->[0] = solid
$line->[1] = black
$line->[2][0] = 1
$line->[2][1] = 2
$line->[2][2] = 3
$line->[3][0] = 4
$line->[3][1] = 5
$line->[3][2] = 6
```

那么三维数组又如何呢？下面是上例略为改动的版本。

```
1  #!/usr/bin/perl
2  #
3  # Using Multi-dimensional Array references again
4  #
5  $line = ['solid', 'black', ['1','2','3', ['4', '5', '6']]];
6  print "\$line->[0] = $line->[0] \n";
7  print "\$line->[1] = $line->[1] \n";
8  print "\$line->[2][0] = $line->[2][0] \n";
9  print "\$line->[2][1] = $line->[2][1] \n";
10 print "\$line->[2][2] = $line->[2][2] \n";
11 print "\$line->[2][3][0] = $line->[2][3][0] \n";
12 print "\$line->[2][3][1] = $line->[2][3][1] \n";
13 print "\$line->[2][3][2] = $line->[2][3][2] \n";
14 print "\n";
```

结果输出如下：

```
$line->[0] = solid
$line->[1] = black
$line->[2][0] = 1
$line->[2][1] = 2
$line->[2][2] = 3
```

```
$line->[2][3][0] = 4
$line->[2][3][1] = 5
$line->[2][3][2] = 6
```

访问第三层元素的方式形如$line->[2][3][0]，类似于C语言中的Array_pointer[2][3][0]。本例中，下标均为数字，当然亦可用变量代替。用这种方法可以把数组和哈希表结合起来构成复杂的结构，如下：

```
                1 #!/usr/bin/perl
2 #
3 # Using Multi-dimensional Array and Hash references
4 #
5 %cube = (
6 '0', ['0', '0', '0'],
7 '1', ['0', '0', '1'],
8 '2', ['0', '1', '0'],
9 '3', ['0', '1', '1'],
10 '4', ['1', '0', '0'],
11 '5', ['1', '0', '1'],
12 '6', ['1', '1', '0'],
13 '7', ['1', '1', '1']
14 );
15 $pointer = \%cube;
16 print "\n Da Cube \n";
17 foreach $i (sort keys %$pointer) {
18 $list = $$pointer{$i};
19 $x = $list->[0];
20 $y = $list->[1];
21 $z = $list->[2];
22 printf " Point $i = $x,$y,$z \n";
23 }
```

结果输出如下：

```
              Da Cube
Point 0 = 0,0,0
Point 1 = 0,0,1
Point 2 = 0,1,0
Point 3 = 0,1,1
Point 4 = 1,0,0
Point 5 = 1,0,1
Point 6 = 1,1,0
Point 7 = 1,1,1
```

这是一个定义立方体的例子。%cube中保存的是点号和坐标，坐标是个含三个数字的数组。变量$list获取坐标数组的引用：$list = $$ pointer{$i}; 然后访问各坐标值：$x = $list->[0]; ... 也可用如下方法给$x、$y和$z赋值：($x,$y,$z) = @$list;

使用哈希表和数组时，用$和用->是类似的，对数组而言下面两个语句等效：

```
$$names[0] = "kamran";
$names->[0] = "kamran";
```

对哈希表而言下面两个语句等效：

```
$$lastnames{"kamran"} = "Husain";
$lastnames->{"kamran"} = "Husain";
```

Perl中的数组可以在运行中创建和扩展。当数组的引用第一次在等式左边出现时，该数组自动被创建，简单变量和多维数组也是一样。如下句，如果数组contours不存在，则被创建：

```
$contours[$x][$y][$z] = &xlate($mouseX, $mouseY);
```

六、子程序的引用

perl中子程序的引用与C中函数的指针类似，构造方法如下：

```
$pointer_to_sub = sub {... declaration of sub ...};
```

通过所构造的引用调用子程序的方法为：

```
&$pointer_to_sub(parameters);
```

- 子程序模板

子程序的返回值不仅限于数据，还可以返回子程序的引用。返回的子程序在调用处执行，但却是在最初被创建的调用处被设置，这是由Perl对Closure处理的方式决定的。Closure意即如果你定义了一个函数，它就以最初定义的内容运行。(Closure详见OOP的参考书)下面的例子中，设置了多个错误信息显示子程序，这样的子程序定义方法可用于创建模板。

```
                #!/usr/bin/perl
    sub errorMsg {
      my $lvl = shift;
      #
      # define the subroutine to run when called.
      #
      return sub {
        my $msg = shift; # Define the error type now.
        print "Err Level $lvl:$msg\n"; }; # print later.
      }
    $severe = errorMsg("Severe");
    $fatal = errorMsg("Fatal");
    $annoy = errorMsg("Annoying");
    &$severe("Divide by zero");
    &$fatal("Did you forget to use a semi-colon?");
    &$annoy("Uninitialized variable in use");
```

结果输出如下：

```
                Err Level Severe:Divide by zero
    Err Level Fatal:Did you forget to use a semi-colon?
    Err Level Annoying:Uninitialized variable in use
```

上例中，子程序errorMsg使用了局域变量$lvl，用于返回给调用者。当errorMsg被调用时，$lvl的值设置到返回的子程序内容中，虽然是用的my函数。三次调用设置了三个不同的$lvl变量值。当errorMsg返回时，$lvl的值保存到每次被声明时所产生的子程序代码中。最后三句对产生的子程序引用进行调用时$msg的值被替换，但$lvl的值仍是相应子程序代码创建时的值。

很混淆是吗？是的，所以这样的代码在Perl程序中很少见。

七、数组与子程序

数组利于管理相关数据，本节讨论如何向子程序传递多个数组。前面我们讲过用@_传递子程序的参数，但是@_是一个单维数组，不管你传递的参数是多少个数组，都按序存贮在@_中，故用形如my(@a,@b)=@_;的语句来获取参数值时，全部值都赋给了@a，而@b为空。那么怎么把一个以上的数组传递给子程序呢？方法是用引用。见下例：

```
                #!/usr/bin/perl
    @names = (mickey, goofy, daffy );
    @phones = (5551234, 5554321, 666 );
    $i = 0;
    sub listem {
      my ($a,$b) = @_;
      foreach (@$a) {
```

```
      print "a[$i] = " . @$a[$i] . " " . "\tb[$i] = " . @$b[$i] ."\n";
      $i++;
    }
  }
  &listem(\@names, \@phones);
```

结果输出如下：

```
                a[0] = mickey      b[0] = 5551234
  a[1] = goofy        b[1] = 5554321
  a[2] = daffy        b[2] = 666
```

注意：

1、当想传递给子程序的参数是多于一个的数组时一定要使用引用。

2、一定不要在子程序中使用形如 (@variable)=@_; 的语句处理参数，除非你想把所有参数集中到一个长的数组中。

八、文件句柄的引用

有时，必须将同一信息输出到不同的文件，例如，某程序可能在一个实例中输出到屏幕，另一个输出到打印机，再一个输出到记录文件，甚至同时输出到这三个文件。相比较于每种处理写一个单独的语句，可以有更好的实现方式如下：

```
spitOut(\*STDIN);
spitOut(\*LPHANDLE);
spitOut(\*LOGHANDLE);
```

其中子程序spitOut的代码如下：

```
                sub spitOut {
    my $fh = shift;
    print $fh "Gee Wilbur, I like this lettuce\n";
  }
```

注意其中文件句柄引用的语法为*FILEHANDLE。

上一章 下一章 目录