Perl子程序(函数)

定义函数示例：

```
#!/usr/bin/perl

# 定义求平均值函数
sub Average{
   # 获取所有传入的参数
   $n = scalar(@_);
   #n为参数个数
   #_为传入的数组
   $sum = 0;

   foreach $item (@_){
      $sum += $item;
   }
   $average = $sum / $n;
   print '传入的参数为 : ',"@_\n";             # 打印整个数组
   print "第一个参数值为 : $_[0]\n";           # 打印第一个参数
   print "传入参数的平均值为 : $average\n";   # 打印平均值
}

# 调用函数
Average(10, 20, 30);
```

用户可以通过改变 @_ 数组中的值来改变相应实际参数的值。也就是说传入的实际上类似C++引用，但有区别。

函数可以传任何参数，数组、哈希表，元素会议数组的方式传入函数中。

子程序也可以有返回值：

```perl
# 方法定义
sub add_a_b{
   #使用 return
   return $_[0]+$_[1];
}
$a=1,$b=2;
print add_a_b($a, $b);
#会返回a+b;
```

子程序的私有变量：

- 默认情况下，Perl 中所有的变量都是全局变量，这就是说变量在程序的任何地方都可以调用。
- 如果需要设置私有变量，可以使用 my 操作符来设置。
- my 操作符用于创建词法作用域变量，通过 my 创建的变量，存活于声明开始的地方，直到闭合作用域的结尾。
- 闭合作用域指的可以是一对花括号中的区域，可以是一个文件，也可以是一个 if, while, for, foreach, eval字符串。

```perl
#!/usr/bin/perl

# 全局变量
$string = "Hello, World!";

# 函数定义
sub PrintHello{
   # PrintHello 函数的私有变量
   my $string;
   $string = "Hello, Runoob!";
   print "函数内字符串：$string\n";
}
# 调用函数
PrintHello();
print "函数外字符串：$string\n";
```

这样的输出就有了内外之分

```
函数内字符串：Hello, Runoob!
函数外字符串：Hello, World!
```

变量临时赋值

使用 local 为全局变量提供临时的值，在退出作用域后将原来的值还回去。local 定义的变量不存在于主程序中，但存在于该子程序和该子程序调用的子程序中。定义时可以给其赋值。

```
# 全局变量
$string = "Hello, World!";

sub PrintRunoob{
   # PrintHello 函数私有变量
   local $string;
   $string = "Hello, Runoob!";
   # 子程序调用的子程序
   PrintMe();
   print "PrintRunoob 函数内字符串值：$string\n";
}
#PrintRunoob 函数内字符串值：Hello, Runoob!
#PrintRunoob 内的local $string
sub PrintMe{
   print "PrintMe 函数内字符串值：$string\n";
}
#PrintMe 函数内字符串值：Hello, Runoob!
#PrintMe 函数优先找自己存在的区域内的$string
sub PrintHello{
   print "PrintHello 函数内字符串值：$string\n";
}
#PrintHello 函数内字符串值：Hello, World!
#PrintHello 在函数外，看见的是全局变量

# 函数调用
PrintRunoob();
PrintHello();
print "函数外部字符串值：$string\n";
#函数外部字符串值：Hello, World!
```

静态变量

函数内的一个只定义一次，不会主动重置的变量

```
#!/usr/bin/perl

use feature 'state';

```

```perl
sub PrintCount{
   state $count = 0; # 初始化变量

   print "counter 值为：$count\n";
   $count++;
}

for (1..5){
   PrintCount();
}
```

```
counter 值为：0
counter 值为：1
counter 值为：2
counter 值为：3
counter 值为：4
```

- 注1：state仅能创建闭合作用域为子程序内部的变量。
- 注2：state是从Perl 5.9.4开始引入的，所以使用前必须加上 use。
- 注3：state可以声明标量、数组、哈希。但在声明数组和哈希时，不能对其初始化（至少Perl 5.14不支持）。

Perl引用

Perl 引用是一个标量类型可以指向变量、数组、哈希表（也叫关联数组）甚至子程序，可以应用在程序的任何地方。

创建引用

引用的一般创建, "\"

定义变量的时候，在变量名前面加个\，就得到了这个变量的一个引用

```
$scalarref = \$foo;      # 标量变量引用
$arrayref  = \@ARGV;     # 列表的引用
$hashref   = \%ENV;      # 哈希的引用
$coderef   = \&handler; # 子过程引用
$globref   = \*foo;      # GLOB句柄引用
```

不定义数组或者哈希，直接定义引用

```
$aref= [ 1,"foo",undef,[13,1,2]];
#用匿名数组引用，使用 [] 定义
$href= { APR =>4, AUG =>8 };
#用匿名哈希引用，使用 {} 定义
$coderef = sub { print "Runoob!\n" };
#创建一个没有子程序名的匿名子程序引用
```

引用的取消

实例如下：

```
#!/usr/bin/perl

$var = 10;

# $r 引用 $var 标量
$r = \$var;

# 输出本地存储的 $r 的变量值
print "$var 为 : ", $$r, "\n";

@var = (1, 2, 3);
# $r 引用  @var 数组
$r = \@var;
# 输出本地存储的 $r 的变量值
print "@var 为: ",  @$r, "\n";

%var = ('key1' => 10, 'key2' => 20);
# $r 引用  %var 哈希
$r = \%var;
# 输出本地存储的 $r 的变量值
print "\%var 为 : ", %$r, "\n";

ref($r);
```

```
#这个语句可以判断引用的变量的类型
=pod
SCALAR
ARRAY
HASH
CODE
GLOB
REF
=cut
```

上例中，$r是一个引用，即地址，在知道地址之后，为了明确print的是什么，还需要使用$$r来确定输出变量的类型，引用的格式并不能混用。注意，上例中运用了perl同名变量如果定义的时候的形式不同，则会成为独立的新变量的特性。即定义了%var之后$var并没有因此消失或者改变，仍然可以通过$var调用。

循环引用

```
#!/usr/bin/perl

my $foo = 100;
$pos = \$foo;
#定义变量的时候本身就是在定义一个引用
#用一个同名引用去指示一个引用意义不明
print "Value of pos is : ", $$pos, "\n";
$foo = \$foo;
print "Value of foo is : ", $$foo, "\n";
```

```
Value of pos is : 100
Value of foo is : REF(0x2711bd0)
```

如果是引用，最好不要有同名引用。是数组或者哈希表的时候会因为perl的特性并不会有冲突。

引用函数

函数引用格式: &

调用引用函数格式: & + 创建的引用名。

```

```

```perl
#!/usr/bin/perl

# 函数定义
sub PrintHash{
   my (%hash) = @_;

   foreach $item (%hash){
      print "元素 : $item\n";
   }
}
%hash = ('name' => 'runoob', 'age' => 3);

# 创建函数的引用
$cref = \&PrintHash;

# 使用引用调用函数
&$cref(%hash);
```

```
元素 : age
元素 : 3
元素 : name
元素 : runoob
```

perl格式化输出

Perl format模板

Perl 中可以使用 format 来定义一个模板，然后使用 write 按指定模板输出数据。

```perl
format FormatName =
fieldline
value_one, value_two, value_three
fieldline
value_one, value_two
.
```

参数解析：

- FormatName：格式化名称。
- fieldline：一个格式行，用来定义一个输出行的格式,类似 @,^,<,>,| 这样的字符。
- value_one,value_two..... ：数据行，用来向前面的格式行中插入值,都是perl的变量。
- .：结束符号。

```
#!/usr/bin/perl

$text = "google runoob taobao";
format STDOUT =
first: ^<<<<<  # 左边对齐，字符长度为6
    $text
second: ^<<<<< # 左边对齐，字符长度为6
    $text
third: ^<<<< # 左边对齐，字符长度为5，taobao 最后一个 o 被截断
    $text
.
write
```

```
first: google
second: runoob
third: taoba
```

格式行(图形行)语法

- 格式行以 @ 或者 ^ 开头，这些行不作任何形式的变量代换。
- @ 字段(不要同数组符号 @ 相混淆)是普通的字段。
- @,^ 后的 <, >,| 长度决定了字段的长度，如果变量超出定义的长度,那么它将被截断。
- <, >,| 还分别表示,左对齐,右对齐,居中对齐。
- ^ 字段用于多行文本块填充。

值域格式:

格式	值域含义
@<<<	左对齐输出
@>>>	右对齐输出
@|||	中对齐输出
@##.##	固定精度数字

格式	值域含义
@*	多行文本

每个值域的第一个字符是行填充符，当使用@字符时，不做文本格式化。

在上表中，除了多行值域@*，域宽都等于其指定的包含字符@在内的字符个数，例如：

```
@###.##
#表示七个字符宽，小数点前四个，小数点后两个。
```

```
#!/usr/bin/perl

format EMPLOYEE =
===================================
@<<<<<<<<<<<<<<<<<<<<<<< @<<
$name, $age
@#####.##
$salary
===================================
.
#这里定义了一个输出的模板，包括画的线、输出$name占据的长度等等。
select(STDOUT);
$~ = EMPLOYEE;

@n = ("Ali", "Runoob", "Jaffer");
@a  = (20,30, 40);
@s = (2000.00, 2500.00, 4000.000);

$i = 0;
foreach (@n){
    $name = $_;
    $age = $a[$i];
    $salary = $s[$i++];
    write;
}
```

格式变量

- $~ ($FORMAT_NAME)：格式名字 $^ ($FORMAT_TOP_NAME)：当前的表头格式名字存储在
- $\% ($FORMAT_PAGE_NUMBER)：当前输出的页号
- $= ($FORMAT_LINES_PER_PAGE)：每页中的行数

- $| ($FORMAT_AUTOFLUSH)：是否自动刷新输出缓冲区存储
- $^L ($FORMAT_FORMFEED)：在每一页(除了第一页)表头之前需要输出的字符串存储在

write可以在上下都找到匹配的format

```
#!/usr/bin/perl

$~ = "MYFORMAT"; # 指定默认文件变量下所使用的格式
write;           # 输出 $~ 所指定的格式

format MYFORMAT = # 定义格式 MYFORMAT
==================================
      Text # 菜鸟教程
==================================
.
write;
```

```
==================================
      Text # 菜鸟教程
==================================
==================================
      Text # 菜鸟教程
==================================
```

不指定$~的情况下，会输出名为STDOUT的格式：

```
#!/usr/bin/perl

write;          # 不指定$~的情况下会寻找名为STDOUT的格式

format STDOUT =
~用~号指定的文字不会被输出
---------------
  STDOUT格式
---------------
.
```

```
---------------
  STDOUT格式
---------------
```

通过添加报表头部信息来演示 $^ 或 $FORMAT_TOP_NAME 变量的使用：

```
#!/usr/bin/perl

format EMPLOYEE =
===================================
@<<<<<<<<<<<<<<<<<<<<<<< @<<
$name, $age
@#####.##
$salary
===================================
.

format EMPLOYEE_TOP =
===================================
Name                    Age
===================================
.

select(STDOUT);
$~ = EMPLOYEE;
$^ = EMPLOYEE_TOP;

@n = ("Ali", "Runoob", "Jaffer");
@a  = (20,30, 40);
@s = (2000.00, 2500.00, 4000.000);

$i = 0;
foreach (@n){
   $name = $_;
   $age = $a[$i];
   $salary = $s[$i++];
   write;
}
```

```
===================================
Name                    Age
```

```
==================================
==================================
Ali                     20
  2000.00
==================================
==================================
Runoob                  30
  2500.00
==================================
==================================
Jaffer                  40
  4000.00
==================================
```

使用 $\% 或 $FORMAT_PAGE_NUMBER 为报表设置分页：

```perl
#!/usr/bin/perl

format EMPLOYEE =
==================================
@<<<<<<<<<<<<<<<<<<<< @<<
$name, $age
@#####.##
$salary
==================================
.

# 添加分页 $%
format EMPLOYEE_TOP =
==================================
Name                    Age Page @<
                                 $%
==================================
.

select(STDOUT);
$~ = EMPLOYEE;
$^ = EMPLOYEE_TOP;

@n = ("Ali", "Runoob", "Jaffer");
@a  = (20,30, 40);
@s = (2000.00, 2500.00, 4000.000);
```

```
27
28 $i = 0;
29 foreach (@n){
30    $name = $_;
31    $age = $a[$i];
32    $salary = $s[$i++];
33    write;
34 }
```

```
1  ====================================
2  Name                    Age Page 1
3  ====================================
4  ====================================
5  Ali                     20
6    2000.00
7  ====================================
8  ====================================
9  Runoob                  30
10   2500.00
11 ====================================
12 ====================================
13 Jaffer                  40
14   4000.00
15 ====================================
```

输出到其它文件

默认情况下函数write将结果输出到标准输出文件STDOUT，我们也可以使它将结果输出到任意其它的文件中。最简单的方法就是把文件变量作为参数传递给write，如：

```
1 write(MYFILE);
```

代码write就用默认名为MYFILE的打印格式输出到文件MYFILE中。

但是这样就不能用$~变量来改变所使用的打印格式。系统变量$~只对默认文件变量起作用，我们可以改变默认文件变量，改变$~，再调用write。

```perl
#!/usr/bin/perl

if (open(MYFILE, ">tmp")) {
$~ = "MYFORMAT";
write MYFILE; # 含文件变量的输出，此时会打印与变量同名的格式，即MYFILE。$~里指定的值被忽略。

format MYFILE = # 与文件变量同名
==================================
       输入到文件中
==================================
.
close MYFILE;
}
```

此时的tmp文件中：

```
$ cat tmp
==================================
       输入到文件中
==================================
```

可以使用select改变默认文件变量时，它返回当前默认文件变量的内部表示，这样我们就可以创建子程序，按自己的想法输出，又不影响程序的其它部分。

```perl
#!/usr/bin/perl

if (open(MYFILE, ">>tmp")) {
select (MYFILE); # 使得默认文件变量的打印输出到MYFILE中
$~ = "OTHER";
write;           # 默认文件变量，打印到select指定的文件中，必使用$~指定的格式 OTHER

format OTHER =
==================================
  使用定义的格式输入到文件中
==================================
.
close MYFILE;
}
```

此时的tmp中：

```
$ cat tmp
==================================
        输入到文件中
==================================
==================================
   使用定义的格式输入到文件中
==================================
```

文件操作

文件操作

目录操作

目录操作

错误处理

错误处理

特殊变量

Perl中的特殊变量

Perl 语言中定义了一些特殊的变量，通常以 $, \@, 或 \% 作为前缀，例如：$_。

很多特殊的变量有一个很长的英文名，操作系统变量 $! 可以写为 $OS_ERROR。

如果你想使用英文名的特殊变量需要在程序头部添加 use English;。这样就可以使用具有描述性的英文特殊变量。

最常用的特殊变量为 $_，该变量包含了默认输入和模式匹配内容。实例如下：

```
#!/usr/bin/perl

foreach ('Google','Runoob','Taobao') {
    print $_;
    print "\n";
}
```

```
#!/usr/bin/perl

foreach ('Google','Runoob','Taobao') {
    print;
    print "\n";
}
```

```
Google
Runoob
Taobao
```

两程序输出相同

特殊变量类型

根据特殊的变量的使用性质，可以分为以下几类：

- 全局标量特殊变量。
- 全局数组特殊变量。
- 全局哈希特殊变量。
- 全局特殊文件句柄。
- 全局特殊常量。
- 正则表达式特殊变量。
- 文件句柄特殊变量。
  特殊变量

Perl正则表达式

正则表达式

正则表达式(regular expression)描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。

Perl语言的正则表达式功能非常强大，基本上是常用语言中最强大的，很多语言设计正则式支持的时候都参考Perl的正则表达式。

- 匹配：m//（还可以简写为//，略去m）
- 替换：s///
- 转化：tr///
  这三种形式一般都和 =~ 或 !~ 搭配使用， =~ 表示相匹配，!~ 表示不匹配。

匹配操作符

基本使用

匹配操作符 m// 用于匹配一个字符串语句或者一个正则表达式，例如，要匹配 标量 $bar 中的 "run"

```
#!/usr/bin/perl

$bar = "I am runoob site. welcome to runoob site.";
if ($bar =~ /run/){
   print "第一次匹配\n";
}else{
   print "第一次不匹配\n";
}

$bar = "run";
if ($bar =~ /run/){
   print "第二次匹配\n";
}else{
   print "第二次不匹配\n";
}
```

```
第一次匹配
第二次匹配
```

模式匹配修饰符

修饰符	描述
i	忽略模式中的大小写
m	多行模式
o	仅赋值一次
s	单行模式，"."匹配"\n"（默认不匹配）
x	忽略模式中的空白
g	全局匹配
cg	全局匹配失败后，允许再次查找匹配串

正则表达式变量

三个正则表达式变量

perl处理完后会给匹配到的值存在三个特殊变量名:

$`: 匹配部分的前一部分字符串
$&: 匹配的字符串
$': 还没有匹配的剩余字符串
如果将这三个变量放在一起,你将得到原始字符串。

```
#!/usr/bin/perl

$string = "welcome to runoob site.";
$string =~ m/run/;
print "匹配前的字符串: $`\n";
print "匹配的字符串: $&\n";
print "匹配后的字符串: $'\n";
```

```
匹配前的字符串: welcome to
匹配的字符串: run
匹配后的字符串: oob site.
```

替换操作符

基本使用

替换操作符 s/// 是匹配操作符的扩展，使用新的字符串替换指定的字符串。基本格式如下：

```
s/PATTERN/REPLACEMENT/;
```

PATTERN 为匹配模式，REPLACEMENT 为替换的字符串。

例如我们将以下字符串的 "google" 替换为 "runoob":

```
#!/usr/bin/perl

$string = "welcome to google site.";
$string =~ s/google/runoob/;

print "$string\n";
```

替换操作修饰符

修饰符	描述
i	如果在修饰符中加上"i"，则正则将会取消大小写敏感性，即"a"和"A" 是一样的。
m	默认的正则开始"^"和结束"$"只是对于正则字符串如果在修饰符中加上"m"，那么开始和结束将会指字符串的每一行：每一行的开头就是"^"，结尾就是"$"。
o	表达式只执行一次。
s	如果在修饰符中加入"s"，那么默认的"."代表除了换行符以外的任何字符将会变成任意字符，也就是包括换行符！
x	如果加上该修饰符，表达式中的空白字符将会被忽略，除非它已经被转义。
g	替换所有匹配的字符串。
e	替换字符串作为表达式

转换操作符

基本操作

修饰符	描述
c	转化所有未指定字符
d	删除所有指定字符
s	把多个相同的输出字符缩成一个

```
#!/usr/bin/perl

$string = 'welcome to runoob site.';
$string =~ tr/a-z/A-z/;
#将a-z转化为A-Z
```

```
$string =~ tr/\d/ /c;
# 把所有非数字字符替换为空格
$string =~ tr/\t //d;
# 删除tab和空格
$string =~ tr/0-9/ /cs
# 把数字间的其它字符替换为一个空格。
```

更多正则表达式规则

表达式	描述
.	匹配除换行符以外的所有字符
x?	匹配 0 次或一次 x 字符串
x*	匹配 0 次或多次 x 字符串,但匹配可能的最少次数
x+	匹配 1 次或多次 x 字符串,但匹配可能的最少次数
.*	匹配 0 次或多次的任何字符
.+	匹配 1 次或多次的任何字符
{m}	匹配刚好是 m 个 的指定字符串
{m,n}	匹配在 m个 以上 n个 以下 的指定字符串
{m,}	匹配 m个 以上 的指定字符串
[]	匹配符合 [] 内的字符
[^]	匹配不符合 [] 内的字符

表达式	描述
[0-9]	匹配所有数字字符
[a-z]	匹配所有小写字母字符
[^0-9]	匹配所有非数字字符
[^a-z]	匹配所有非小写字母字符
^	匹配字符开头的字符
$	匹配字符结尾的字符
\d	匹配一个数字的字符,和 [0-9] 语法一样
\d+	匹配多个数字字符串,和 [0-9]+ 语法一样
\D	非数字,其他同 \d
\D+	非数字,其他同 \d+
\w	英文字母或数字的字符串,和 [a-zA-Z0-9_] 语法一样
\w+	和 [a-zA-Z0-9_]+ 语法一样
\W	非英文字母或数字的字符串,和 [^a-zA-Z0-9_] 语法一样
\W+	和 [^a-zA-Z0-9_]+ 语法一样
\s	空格,和 [\n\t\r\f] 语法一样
\s+	和 [\n\t\r\f]+ 一样
\S	非空格,和 [^\n\t\r\f] 语法一样
\S+	和 [^\n\t\r\f]+ 语法一样
\b	匹配以英文字母,数字为边界的字符串
\B	匹配不以英文字母,数值为边界的字符串
a	b
abc	匹配含有 abc 的字符串 (pattern) () 这个符号会记住所找寻到的字符串,是一个很实用的语法.第一个 () 内所找到的字符串变成 $1 这个变量或是 \1 变量,第二个 () 内所找到的字符串变成 $2 这个变量或是 \2 变量,以此类推下去.
/pattern/i	i 这个参数表示忽略英文大小写,也就是在匹配字符串的时候,不考虑英文的大小写问题. \ 如果要在 pattern 模式中找寻一个特殊字符,如 "*",则要在这个字符前加上 \ 符号,这样才会让特殊字符失效

Perl面向对象

Perl的特殊点

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，存储了所有的实例属性。

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。

Perl中的面向对象

面向对象有很多基础概念，这里我们接收三个：对象、类和方法。

- 对象：对象是对类中数据项的引用。.
- 类：类是个Perl包，其中含提供对象方法的类。
- 方法：方法是个Perl子程序，类名是其第一个参数。

Perl 提供了 bless() 函数，bless 是用来构造对象的， 通过 bless 把一个引用和这个类名相关联，返回这个引用就构造出一个对象。

定义类

一个类只是一个简单的包。

可以把一个包当作一个类用，并且把包里的函数当作类的方法来用。

Perl 的包提供了独立的命名空间，所以不同包的方法与变量名不会冲突。

Perl 类的文件后缀为 .pm。

接下来我们创建一个 Person 类：

```
1  package Person;
```

类的代码范围到脚本文件的最后一行，或者到下一个 package 关键字前。

对象的创建与使用

介绍

创建一个类的实例 (对象) 我们需要定义一个构造函数，大多数程序使用类名作为构造函数，Perl 中可以使用任何名字。

可以使用多种 Perl 的变量作为 Perl 的对象。大多数情况下我们会使用引用数组或哈希。

接下来我们为 Person 类创建一个构造函数，使用了 Perl 的哈希引用。

在创建对象时，你需要提供一个构造函数，它是一个子程序，返回对象的引用。

示例：

```
package Person;
sub new
{
    my $class = shift;#下例中的Person
    #class是
    my $self = {
        _firstName => shift,
        #下例中的小明
        _lastName  => shift,
        #下例中的王
        _ssn       => shift,
        #下例中的数字
    };
    #$self是一个哈希表的引用
    # 输出用户信息
    print "名字：$self->{_firstName}\n";
    print "姓氏：$self->{_lastName}\n";
    print "编号：$self->{_ssn}\n";
    bless $self, $class;
    #将类名与类的内容关联
    return $self;
    #返回类的引用
}
#要注意在这个文件里都算是这个类的东西，
#需要在结尾再加一个package关键字表明类的定义结束
```

这样就能定义一个对象：

```
$object = new Person( "小明", "王", 23234345);
```

定义方法

相关概念

Perl类的方法只是个Perl子程序，即通常所说的成员函数。

Perl面向对象中Perl的方法定义不提供任何特别语法，但规定方法的第一个参数为对象或其被引用的包。

Perl 没有提供私有变量，但我们可以通过辅助的方式来管理对象数据。

定义方法示例：

方法

```
sub getFirstName {
    return $self->{_firstName};
}

sub setFirstName {
    my ( $self, $firstName ) = @_;
    #从参数中读到$self部分和$firstName
    $self->{_firstName} = $firstName if defined($firstName);
    #改变$self中的参数为$firstName
    return $self->{_firstName};
    #应该并不需要return
}
```

完整的package内容

这样我们就得到了一个完整的可以设置不同参数的package Person

```
#!/usr/bin/perl

package Person;
```

```perl

sub new
{
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName  => shift,
        _ssn       => shift,
    };
    # 输出用户信息
    print "名字: $self->{_firstName}\n";
    print "姓氏: $self->{_lastName}\n";
    print "编号: $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}

sub getFirstName {
    my( $self ) = @_;
    return $self->{_firstName};
}
1;
#这里是文件结尾，当然package结尾也可以
```

package的调用

在pl文件中使用的时候就可以用use来导入类

```perl
#!/usr/bin/perl

use Person;

$object = new Person( "小明", "王", 23234345);
# 获取姓名
$firstName = $object->getFirstName();

print "设置前姓名为 : $firstName\n";
```

```
10
11 # 使用辅助函数设置姓名
12 $object->setFirstName( "小强" );
13
14 # 通过辅助函数获取姓名
15 $firstName = $object->getFirstName();
16 print "设置后姓名为 : $firstName\n";
```

perl中引用使用->操作来调用方法

继承

相关说明

Perl 里 类方法通过@ISA数组继承，这个数组里面包含其他包（类）的名字，变量的继承必须明确设定。

多继承就是这个@ISA数组包含多个类（包）名字。

通过@ISA只能继承方法，不能继承数据。

Employee 类继承 Person

Employee.pm 文件代码如下所示：

```
1 #!/usr/bin/perl
2
3 package Employee;
4 use Person;
5 use strict;
6 our @ISA = qw(Person);    # 从 Person 继承
```

现在 Employee 类包含了 Person 类的所有方法和属性，我们在 main.pl 文件中输入以下代码，并执行:

```
1 #!/usr/bin/perl
2
3 use Employee;
4
```

```perl
$object = new Employee( "小明", "王", 23234345);
# 获取姓名
$firstName = $object->getFirstName();

print "设置前姓名为 : $firstName\n";

# 使用辅助函数设置姓名
$object->setFirstName( "小强" );

# 通过辅助函数获取姓名
$firstName = $object->getFirstName();
print "设置后姓名为 : $firstName\n";
```

如何理解？$self是和Person绑定的属性变量。所以在调用new的方法的时候能直接用Person的相同定义方法。

继承方法重写

在 Employee 类中添加一些新方法，并重写了 Person 类的方法：

```perl
#!/usr/bin/perl

package Employee;
use Person;
use strict;
our @ISA = qw(Person);
# 从 Person 继承

# 重写构造函数
sub new {
    my ($class) = @_;

    # 调用父类的构造函数
    my $self = $class->SUPER::new( $_[1], $_[2], $_[3] );
    # 添加更多属性
    $self->{_id}    = undef;
    $self->{_title} = undef;
    bless $self, $class;
    return $self;
}

# 重写方法
```

```perl
sub getFirstName {
    my( $self ) = @_;
    #这句话是把对象中的东西取出来
    #这些东西的引用是用一个$self装的
    #$self是一个哈希表
    #这是子类函数
    print "这是子类函数\n";
    return $self->{_firstName};
}

# 添加方法
sub setLastName{
   #方法的定义，需要用到包本身的东西，
   #所以先用$self来接对象的东西，
   #这样就用$self来装对象本身的数据，
   #然后用lastName来装要传的东西
    my ( $self, $lastName ) = @_;
    $self->{_lastName} = $lastName if defined($lastName);
    return $self->{_lastName};
}

sub getLastName {
    my( $self ) = @_;
    return $self->{_lastName};
}

1;
```

默认载入

概念

如果在当前类、当前类所有的基类、还有 UNIVERSAL 类中都找不到请求的方法， 这时会再次查找名为 AUTOLOAD() 的一个方法。如果找到了 AUTOLOAD，那么就会 调用，同时设定全局变量 $AUTOLOAD 的值为缺失的方法的全限定名称。

如果还不行，那么 Perl 就宣告失败并出错。

示例

如果你不想继承基类的 AUTOLOAD，很简单，只需要一句：

```
sub AUTOLOAD;
```

析构函数及垃圾回收

DESTORY

当对象的最后一个引用释放时，对象会自动析构。

如果你想在析构的时候做些什么，那么你可以在类中定义一个名为"DESTROY"的方法。它将在适合的时机自动调用，并且按照你的意思执行额外的清理动作。

```
package MyClass;
...
#...表示这部分代码未完成
sub DESTROY
{
    print "MyClass::DESTROY called\n";
}
```

Perl 会把对象的引用作为 唯一的参数传递给 DESTROY。注意这个引用是只读的，也就是说你不能通过访问$_[0] 来修改它。（译者注：参见 perlsub）但是对象自身（比如 "${$_[0]" 或者 "@{$_[0]}" 还有 "%{$_[0]}" 等等）还是可写的。

如果你在析构器返回之前重新 bless 了对象引用，那么 Perl 会在析构器返回之后接着调用你重新 bless 的那个对象的 DESTROY 方法。这可以让你有机会调用基类或者你指定的其它类的析构器。需要说明的是，DESTROY 也可以手工调用，但是通常没有必要这么做。

在当前对象释放后，包含在当前对象中的其它对象会自动释放。

Perl面向对象完整示例：

```
#!/usr/bin/perl

# 下面是简单的类实现
package MyClass;

```

```perl
sub new
{
   print "MyClass::new called\n";
   my $type = shift;            # 包名
   my $self = {};               # 引用空哈希
   return bless $self, $type;
}

sub DESTROY
{
   print "MyClass::DESTROY called\n";
}

sub MyMethod
{
   print "MyClass::MyMethod called!\n";
}


# 继承实现
package MySubClass;

@ISA = qw( MyClass );

sub new
{
   print "MySubClass::new called\n";
   my $type = shift;            # 包名
   my $self = MyClass->new;     # 引用空哈希
   return bless $self, $type;
}

sub DESTROY
{
   print "MySubClass::DESTROY called\n";
}

sub MyMethod
{
   my $self = shift;
   $self->SUPER::MyMethod();
   print "   MySubClass::MyMethod called!\n";
}
```

```
49
50 # 调用以上类的主程序
51 package main;
52
53 print "调用 MyClass 方法\n";
54
55 $myObject = MyClass->new();
56 $myObject->MyMethod();
57
58 print "调用 MySubClass 方法\n";
59
60 $myObject2 = MySubClass->new();
61 $myObject2->MyMethod();
62
63 print "创建一个作用域对象\n";
64 {
65   my $myObject2 = MyClass->new();
66 }
67 # 自动调用析构函数
68
69 print "创建对象\n";
70 $myObject3 = MyClass->new();
71 undef $myObject3;
72
73 print "脚本执行结束...\n";
74 # 自动执行析构函数
```

```
1  调用 MyClass 方法
2  MyClass::new called
3  MyClass::MyMethod called!
4  调用 MySubClass 方法
5  MySubClass::new called
6  MyClass::new called
7  MyClass::MyMethod called!
8  MySubClass::MyMethod called!
9  创建一个作用域对象
10 MyClass::new called
11 MyClass::DESTROY called
12 创建对象
13 MyClass::new called
14 MyClass::DESTROY called
15 脚本执行结束...
```

```
16 MyClass::DESTROY called
17 MySubClass::DESTROY called
```