2010/08/1

深未来技术室

神奇的 PERL

Perl 脚本语言最佳入门读物

刘兴（QQ:1477022617）

目录

第一章 拥抱 PERL

1. PERL 是什么?

Perl 最初的设计者为 Larry Wall， Perl 借取了 C、sed、awk、shell scripting 以及很多其他程序语言的特性。Perl 一般被称为“实用报表提取语言”（PracticalExtraction andReportLanguage），有时也被称做“病态折中垃圾列表器”(PathologicallyEclectic Rubbish Lister)。

2. PERL 是神奇的

Perl 的神奇之处在于具有 C 语言一样的强大能力和灵活性，但却比 C 简单很多。

你不用先学习所有 Perl 的东西就可以开始写有用的程序，这对于那些急于完成任务却不得不为此编写一堆代码的人来可谓雪中送炭，从一开始，Perl 就设计成可以把简单工作简单化，同时又不失去处理困难问题能力的语言，Perl 既强大又好用，所以它被广泛地用于日常生活的方方面面，从宇航工程到分子生物学，从数学到语言学，从图形处理到文档处理，从数据库操作到网络管理。

对于不想为程序设计语言买单的人来说，Perl 也不失一种选择，因为 Perl 的解释程序是开放源码的免费软件，使用 Perl 不必担心费用，Perl 也能在绝大多数操作系统运行，可以方便地向不同操作系统迁移。

3. PERL 运行环境

3.1 ACTIVEPERL

ActivePerl 是一个可以让你任意执行 Perl 程序的工具软件，可在 Windows、Mac OS X 、Linux、Solaris、HP-UX、AIX 5 多个平台运行。包括 Perl for Win32、Perl for ISAPI、PerlScript、Perl Package Manager 开发工具程序。

下载地址：http://www.activestate.com/activeperl/

3.2 IDE

1、 Padre

下载地址：http://padre.perlide.org/download.html

运行平台：Windows、Linux、Mac OS X

介绍：使用 Perl 语言开发而成的 Perl IDE（集成开发环境），包括可定制语法高亮显示、语法检查和重构工具（支持 Perl 5 及 Perl 6）、上下文帮助、跨平台支持（Linux、Mac OS X、Windows）等功能。

2、 Open Perl IDE

下载地址：http://sourceforge.net/projects/open-perl-ide/files/

运行平台：windows

介绍：免费且开源，绿色软件，使用时直接运行 PerlIDE.exe。如果在 windows 开台开发，推荐 Open Perl IDE 和 ultraedit 结合使用。

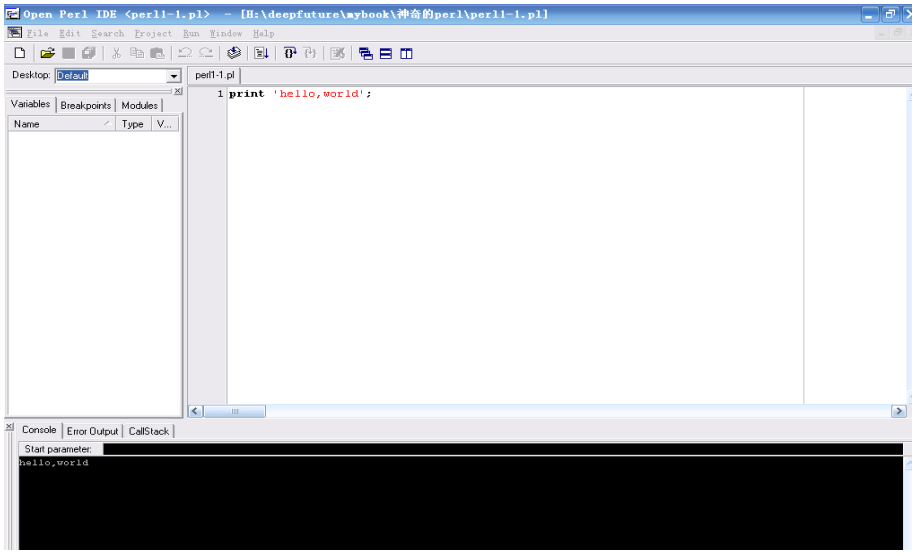
4. HELLO,WORLD

下面我们开始 perl 的 helloworld 之旅。打开 Open Perl IDE，新建一个名为 perl1-1.pl 的文件，内容如下(#表示注释)：

```
#perl1-1.pl

print 'hello,world';#太简单了，就一句
```

选择 RUN 菜单下的 RUN 运行该程序，在监视窗口的 Console 部分，hello,world 已经向我们招手了。



第二章 PERL 语法

1. 语句

1.1 结构

单个语句用;表示结束，结构如下：

语句;注释

例如：

print “hello,word”;#输出 hello,word

1.2 注释

#表示后面的内容为注释。

1.3 语句块

1）语句块用{和}包围。

2）一行可以有多个语句。

2. 执行及扩展名

2.1 程序扩展名

程序通常以 pl 结尾，如果是做为 Apache 容器中的 cgi 程序执行脚本，必须为脚本文件名提供.pl 或.cgi 扩展名。

2.2 程序执行

在命令行中输入:perl 程序名

3. 变量与常量

3.1 变量的表示

变量的表示非常简洁：

1）不要求变量有类型。

2）对于数组，采用以@后接变量名表示,比如@names。

3）对于保存单个值的变量，采用$后接变量名表示，称之为标量，比如$name。

4）对于哈希变量，采用%后接变量名表示。

5)变量在第一次赋值前有一个特殊值 undef，即没有任何值。

3.2 变量的声明

1） 变量可以不声明，直接使用。

2） 声明语法是：

作用域 变量名

3.3 变量的作用域

1)变量前不加作用域限定，表示该变量为全局变量。

2)主要有 my 作用域和 local 作用域。

my 作用域主要用来声明局部变量。my 声明的变量仅作用于声明该的层次及同一层次上定义的其它子程序。

local 作用域只影响声明的层次以及以内的层次，仅能在 local 作用域声明的层次及以内层次修改变量值。

3.4 预定义变量

预定义变量	用途
$_	在执行输入和模式搜索操作时使用的默认空格变量
$.	文件中最后处理的当前行号
$@	由最近一个 eval()运算符提供的 Perl 语法报错信息
$!	获取当前错误信息值，常用于 die 命令

$0	获取当前执行脚本的程序名
$$	正在执行脚本的 Perl 进程号
$PERL_VERSION /z	Perl 解释器的版本、子版本和修订版本信息
@ARGV	获取命令行参数列表
ARGV	一个特殊的文件句柄，用于遍历@ARGV 中出现的所有文件名
@INC	库文件的搜索路径
@_	在子程序中，@_变量含有传给该子程序的变量内容
%ENV	关联数组型变量%ENV 含有当前环境信息
%SIG	关联数组型变量%SIG 含有指向信号内容的句柄

3.5 使用 PRINT

3.5.1 基本用法

Perl 中有一些预定义的文件句柄，如标准输入 STDIN、标准输出 STDOUT、和标准错误 STDERR。print 为输出语句，可以输出到这些文件句柄中，但不输出末尾的回车换行符。

使用语法为：

print 输出句柄 要输出的字符串 1, 要输出的字符串 2,..., 要输出的字符串 n;

如果输出到标准输出句柄中，默认为屏幕，可省略输出设备:

print 要输出的字符串 1, 要输出的字符串 2,..., 要输出的字符串 n;

print STDOUT 要输出的字符串 1, 要输出的字符串 2,..., 要输出的字符串 n;

以上 2 个语句完成同样的功能。比如：

#perl2-6.pl

print STDOUT "abc","def";

print "12345";

输出结果为：

abcdef12345

3.5.2 输出缓冲

Perl 将输出先存在缓冲区，等到缓冲区满后再输出，可使用$|打开和关闭输出缓冲，当$|非 0 值，表示关闭输出缓冲,$|默认为 0。

例如：下列代码向 STDOUT 和 STDERR 输出字符串：

```
print STDOUT "我出错了!\n 故障表现为无法连接网站";

print STDERR "\n 明白，故障正在处理\n";
```

我们希望能输出如下结果：

我出错了!

故障表现为无法连接网站

故障处理中

但是运行这段代码却输出这样的结果：

我出错了!

故障处理中

故障表现为无法连接网站

因为 STDERR 是额外的输出路径，print 函数属于行缓冲，遇到换行符后，就输出“\n”之前的“我出错了!”，而“故障表现为无法连接网站”放入缓冲区中，等待下一次的输出。我们关闭输出缓冲，强制把"我出错了!\n 故障表现为无法连接网站"立即输出。

```
$|=1;

print STDOUT "我出错了!\n 故障表现为无法连接网站";

print STDERR "\n 明白，故障正在处理\n";
```

运行后得到了正确的输出。

3.6 DEFINED、UNDEF 与 DELETE

1、definde 方法检查变量值、子程序、函数是否定义，并返回 Boolean 值。语法如下：

1) 检查变量的值是否定义:

defined EXPR

2) 检查子程序(subroutine)和函数(func)是否定义：

defined(&func)

defined(&subroutine)

3) 检查$_是否定义:

defined

我们以输入密码为例来讲解。

```
my  $passwd;
print  "请输入密码，windows 以 Ctrl  +  Z 结束,linux 以 Ctrl  +D\n";
$passwd=<STDIN>;#用户键盘输入字符并赋值给$passwd 变量。
if  (defined($passwd))  {#如果用户输入了密码，则$passwd 存储了密码
    print  "密码:";
    chomp($passwd);
    print  $passwd;
}
else{#用户没有输入密码，$passwd 值仍未定义
    print  "密码为空";
}
```

在 windows 下，如果用户不输入密码而直接 Ctrl + Z，然后回车，$passwd 无法取得用户键盘输入的值，值仍未定义，所以会输出密码为空。

2、undef 方法可将变量、子程序置为未定义，但并未删除它们。

比如：

undef $student;

undef $city{'湛江'};

undef myarray;

3、delete 删除哈希或数组的切片和某个元素。

比如：

delete $stundent{'张三'};

delete $fruit[1];

3.7 EXISTS

exists 函数检查哈希元素、数组元素、子程序或函数是否存在。

对于哈希元素而言，只要曾被初始化，即使其值未定义，返回真；对于数组元素，仅当其值被定义，才返回真；子程序或函数只要声明过，即使未定义，返回真。

比如:

```
#哈希
if (exists $stundent{'张三'}){
    print “Exists\n”;
}
#数组
if (exists $myarray[1]){
   print "Exists\n"
}
#子程序
if (exists &subroutine){
   print "Exists\n"
}
```

3.8 常量

perl 可以定义普通常量、哈希常量、数组常量.

1、普通常量如:

use constan A =>”A”;

2、哈希常量

use consant{

ONE=>1,

TWO=>2,

TREE=>3,

FOUR=>4

FIVE=>5,

}

3、数组常量

use constant ARRAY => [1,2,3,4];

4. 子程序

PERL 的子程序可以出现在程序的任何地方。在子程序中，@_变量含有传给该子程序的变量内容，因为通过@_取参数。定义方法为：

sub 子程序名{

my ($参数 1, $参数 2, $参数 3) = @_;

语句 1；

语句 2；

.....

return (返回值);

}

调用子程序的方法如下：

&子程序名;

以下子程序完成求 2 数之和的功能：

```
#perl2-4.pl
sub add{
    my ($x,$y)=@_; #取得子程序的参数
    return($x+$y);
}
print &add(10,5);
```

输出：15

缺省情况下，子程序中最后一个语句的值将用作返回值，这意味着可以不使用 return 返回值。我们把上面的程序改写一下：

```
#perl2-7.pl
sub add{
    my ($x,$y)=@_; #取得子程序的参数
    $x+$y;
}
print &add(10,5);
```

虽然没有使用 return，但子程序仍然完成了它的功能，输出为：15

5. 基本操作符

5.1 算术操作符

+-* / %　分别表示加、减、乘、除和取余。

比如：

```
#perl2-8.pl
$res=10+5;
print $res;#加
print "#";
```

```
$res=10-5;
print $res;#减
print "#";
print 10*5;#乘，没有使用变量，直接使用表达式，效果一样。
print "#";
print 10/5;#除
print "#";
print 10%3;#取 10 除以 3 的余数 ， 结果为 1
print "#";
```

输出结果为：

15#5#50#2#1#

5.2 自增与自减

++为自增，--为自减。

如:

```
#perl2-9.pl
$res=10;
$res++;#res=11
print $res;
print "#";
$res--;#$res=11-1=10
print $res;
```

输出为：

11#10

5.3 比较操作符

1、数字比较

< 小于

= 等于

> 大于

== 等于

<= 小于等于

>= 大于等于

!= 不等于

<=> 比较

2、字符比较

lt 小于

gt 大于

eq 等于

le 小于等于

ge 大于等于

ne 等于

cmp 比较

5.4 字符串操作符

X 使前面的字符串重复

. 连接字符串

比如：

```
#perl2-11.pl
print "xy"."z";#将 xy 和 z 连接成 xyz
print "#";
print "xy"x5;#将 xy 重复 5 次
```

输出：xyz#xyxyxyxyxy

5.5 逻辑运算符

Perl 有以下逻辑运算符

and、&& 逻辑与

or、|| 逻辑或

not 、！逻辑非

xor 异或

5.6 位操作符

& 按位与

| 按位或

~ 按位非

^ 按位异或

<< 左移

>> 右移

5.7 赋值操作符

Perl 支持以下赋值操作符:

=、 **=、 +=、*=、&=、<<=、&&=、-=、/=、|=、>>=、||=、.=、%=、^=、x=

6. 选择控制

在 Perl 中，选择控制包括 if 和 unless，条件表达式值为空串或 0 时，为假，否则为真。

6.1 如果为真 IF

表示如果为真。

语法一:

```
if(条件表达式)
{
    表达式为真时的语句块;
}
```

或者为以下语法(注意不是语句块，是单个语句)：

表达式为真时的语句块 if 条件表达式;

范例 1：

```
#perl0-1.pl
$x=1;
if ($x>0){
  print “ok”;
}
print “ok” if $x>0;#和本程序中的上一个 if 功能完全一样
```

范例 2:

```
#perl0-2.pl
$x=1;
if ($x) {#条件表达式非 0，即真
    print “x:yes\n”;
```

```
}
$y=0;
if ($y) {#条件表达式为 0，即假
    print “y:yes\n”;
}
$z=0.0;
if ($z) {#条件表达式为 0，即假，注意 0.0 也作为数值表达式等于 0
    print “z:yes\n”;
}
$a=’0’;
if ($a) {#条件表达式为 0，即假
    print “a:yes\n”;
}
$b=’00’;
if ($b) {#条件表达式非 0，即真
    print “b:yes\n”;
}
$c=’0.0’;
if ($c) {#条件表达式非 0，即真,注意’0.0’做为字符表达式不等于 0
    print “c:yes\n”;
}
$d=’’;
if ($d) {#条件表达式为空串，即假
```

```
    print “d:yes\n”;
}
$e=’ ‘;
if ($e) {#条件表达式为一个空格，不为空串，即真
    print “e:yes\n”;
}
```

运行结果如下：

```
x:yes
b:yes
c:yes
e:yes
```

语法二:

```
if(条件表达式){
    条件表达式为真时的语句块;
}else{
    条件表达式为假时的语句块;
}
```

范例:

```
#perl0-3.pl
$x=5;
if($x>0)
{
    print”正数!\n”;
}else{
```

```
    print”0 或负数!\n”;
}
```

运行结果如下：

```
正数!
```

语法三:

```
if (条件表达式一)
{
    条件表达式一为真时的语句块;
}elsif(条件表达式二){
    条件表达式二为真时的语句块;
}elsif(条件表达式三){
     条件表达式三为真时的语句块;
}
…
…
…
elsif(条件表达式 n){
     条件表达式 n 为真时的语句块;
}else{
      所有条件表达式为假时的语句块;
}
```

范例:

```
    #perl0-4.pl
```

```
$x=<STDIN>;
chomp($x);
if($x>=10000)
{
  print”大于 1 万的正数!\n”;
}elsif ($x>0){
  print”小于 1 万的正数 0!\n”;
}
elsif ($x==0){
  print”0!\n”;
}
else{
   print “负数!\n”;
}
```

运行几次程序，依次输入测试数字 100、300000、-200、0。

```
Perl perl0-4.pl
100
小于 1 万的正数 0!
Perl perl0-4.pl
300000
大于 1 万的正数!
Perl perl0-4.pl
-200
```

负数!

Perl perl0-4.pl

0

0!

6.2 如果非真 UNLESS

表示如果非真，即：如果条件表达式不为真。

语法一:

```
unless(条件表达式) {
    条件表达式为假执行的语句块;
}
```

也可以写成：

条件表达式为假执行的语句块 unless (条件表达式);

范例:

```
#perl0-5.pl
#%为除余操作符
$x=6;
unless($x%2){
    print “x 是偶数\n”;
}
print “x 是偶数\n” unless($x%2);
```

输出结果为：

x 是偶数

x 是偶数

语法二:

```
unless(条件表达式)
{
    条件表达式为假时执行的语句块;
}else{
    条件表达式为真时执行的语句块;
}
```

范例:

```
#perl0-6.pl
#%为除余操作符
$x=9;
unless($x%2){
   print “x 是偶数\n”;
}
else{
   print “x 是奇数\n”;
}
```

输出结果为:

```
x 是奇数
```

6.3 更简洁的控制方式

6.3.1 替代 IF-ELSE 结构的三目操作符?:

?: 操作符可以替代 if-else 结构，语法格式如下:

条件表示式?表达式 1:表达式 2

表示如果条件表示式为真则求表达式 1 的值，否则求表达式 2 的值。

范例:

```
#perl0-7.pl
#%为除余操作符
$x=9;
$x%2?print "x 是奇数\n":print "x 是偶数\n";
```

输出结果为:

x 是奇数

6.3.2 替代 IF 结构的&&

&&可替代 if 结构，语法格式如下：

条件表达式 && 条件表达式为真时执行的语句块

范例：

```
#perl0-8.pl
#%为除余操作符
$x=8;
$x%2&& print "x 是奇数\n";
```

因为 8 不是奇数，所以 print 语句不会执行，无输出结果。

6.3.3 替代 UNLESS 结构的||

||可替代 unless 结构，语法格式如下：

条件表达式 || 条件表达式为假执行的语句块

```
#perl0-9.pl
#%为除余操作符
$x=8;
```

```
$x%2|| print "x 是偶数\n";
```

因为 8 是偶数，所以输出结果如下：

```
x 是偶数
```

7. 循环控制

Perl 循环控制很丰富，其中某些控制语法在处理和列表数组方面相当实用和高效。

7.1 WHILE

语法一:

```
while(条件表达式) {
    语句块;
}
```

范例:

```
#perl0-10.pl
#计算 1 到 10 的连乘结果
$x=1;
$jg=1;
while($x<=10)
{
$jg=$jg*$x;
$x++;
}
print “$jg\n”;
```

输出结果如下：

3628800

语法二:

语句块 while(判别运算式);

范例:

```
#perl0-11.pl
#计算 1 到 10 的连乘结果
$x=1;
$jg=1;
{
$jg*=$x;
$x++;
} while($x<=10)
print “$jg\n”;
```

请注意$jg*=$x 这种写法，学过 C 语言的都知道：

$jg*=$x 与$jg=$jg*$x 的结果一样，但编译效率更高。

输出结果如下：

3628800

7.2 DO WHILE

与 while 相比，do while 保证语句块至少执行一次。

语法:

```
do
{
    语句块;
```

}while(条件表达式)

范例:

```
#perl0-12.pl

#求和 1 到 100

$jg=0;

$x=1;

do{

    $jg+=$x;
    $x++;

}while($x<=100);
print "$jg\n";
```

输出结果如下：

```
5050
```

7.3 UNTIL

until 直到...才终止循环

语法:

```
until(条件表达式)
{

  语句块;

}
```

也可以写成: 语句块 until(条件表达式);

范例:

```
#perl0-13.pl

#从 1 开始求和，直到和大于 1000 终止
```

```
$jg=0;
$x=1;
until ($jg >1000)
{
$jg+=$x;
$x++;
}
print "1->$x:";
print "$jg\n";
```

输出结果如下：

```
1->46:1035
```

7.4 DO …UNTIL

do …until 直到...才终止循环

语法:

```
do{
   语句块;
}until (条件表达式);
```

范例:

```
#perl0-14.pl
#从 1 开始求和，直到和大于 1000 终止
$jg=0;
$x=1;
do
```

```
{

$jg+=$x;

$x++;

} until ($jg >1000);

print "1->$x:";

print "$jg\n";输出结果如下：

1->46:1035
```

7.5 FOR

循环

语法一:

```
for (初始表达式;条件表达式;循环过程运算式)
{

    语句块;

}
```

范例一:

```
#perl0-15.pl

#从 1 开始求和，直到和大于 1000 终止

$jg=0;

$x=1;

for ($jg=0,$x=1; $jg <=1000;$x++)

{

    $jg+=$x;

}
```

```
print "1->$x:";

print "$jg\n";
```

输出结果如下：

```
1->46:1035
```

语法二:

每次循环依次将数组变量的元素指定给标量

```
for  标量(数组变量)
{

    语句块;

}
```

范例：

```
#perl0-16.pl

#从数组中挑出奇数

@myarr=(2,5,7,10,23,33,18);

print"奇数如下:\n";

for $num(@myarr)

{

  $num%2&&print "$num  ";

}

print"\n";
```

输出结果如下：

```
奇数如下:

5  7  23  33
```

语法三:

每次循环依次将数组变量的元素指定给预定义变量$_

```
for  (数组变量)
{
    读取$_，处理数组的相关语句块;
}
```

范例：

```
#perl0-17.pl
#从数组中挑出奇数
@myarr=(2,5,7,10,23,33,18);
print"奇数如下:\n";
for (@myarr)
{
  $_%2&&print "$_  ";
}
print"\n";
```

输出结果如下：

```
奇数如下:
5  7  23  33
```

7.6 FOREACH

每次循环依次将数组变量的元素指定给标量

语法一:

```
foreach  标量(数组变量)
{
```

```
    语句块;
}
```

如果把$variable 变量省略的话,就会将数组@array 的元素一一指定给$_这个内定的输出变量.

范例:

```
#perl0-18.pl
#从数组中挑出奇数
@myarr=(2,5,7,10,23,33,18);
print"奇数如下:\n";
foreach $num(@myarr)
{
  $num%2&&print "$num  ";
}
print"\n";
```

输出结果如下:

```
奇数如下:
5  7  23  33
```

语法二:

每次循环依次将数组变量的元素指定给预定义变量$_

```
foreach  (数组变量)
{
    语句块;
}
```

范例:

```
#perl0-19.pl
#从数组中挑出奇数
@myarr=(2,5,7,10,23,33,18);
print"奇数如下:\n";
foreach (@myarr)
{
  $_%2&&print; #使用无参数的 print 表示输出$_
  print" ";
}
print"\n";
```

输出结果如下：

```
奇数如下:
5  7  23  33
```

7.7 LAST 退出循环

退出循环

语法:

```
last
```

范例:

```
#perl0-20.pl
#从 1 开始累加，直到和大于 500 为止
$x=1;
$jg=0;
while (1)
```

```
{
        $jg+=$x;
        $jg>500&&last;#求和>500 退出循环
        $x++;
 }
print"sum(1:$x)=$jg\n";
```

输出结果如下：

```
sum(1:32)=528
```

7.8 NEXT 终止本次循环

终止本次循环，进入下一循环。

语法:

```
next
```

范例:

```
#perl0-21.pl
#求出 1-100 内的奇数之和
foreach ($x=1,$jg=0;$x<=100;$x++){
        $x%2||next;#如果是偶数则结束本次循环，不累加，继续下次循环
        $jg+=$x;
        $x++;
 }
print "$jg\n";
```

第三章 PERL 处理输入输出

1. 第一个 PERL 任务

我们先放松想像一下：你是公司一个普通的开发人员，一大早，阳光明媚，你急不可待打开 QQ,跟最近泡上的 MM 聊天。此时项目组长急匆匆过来了，交给你一个紧急任务，将 perl1-2.txt 中的人员按城市分别存为几个文本文件，文件名是城市名，文本内容如下：

刘欢欢,20,长沙#冯军,25,上海#李兵,21,北京#李军,23,北京#李志,27,北京#黄王兵,29,长沙#赵兵,22,上海#李强兵,25,上海。。。。。。。

你不禁倒吸一口凉气，天呀，又要编写一大堆代码来完成这个任务，而且还要读写文件，好久没操作文件 IO 了，有些命令都忘了。你也许足够聪明，想到干脆把 perl1-2.txt 导入到数据库中，然后运行 SQL 查询，按城市生成不同的表再导出，很快你打消这个念头，工作量大，如果城市众多，手工将生成的表导出几乎不可能。用 C、C++、Pascal、java 等高级语言确实完成这个任务，但代码量较大，使用 Perl，10 多行就能搞定。请允许我在此卖个关子，我们先了解一下 Perl 的输入输出基础，回头再来解决这个任务。

2. PERL 的文件处理原则

Perl 的文件处理很简单，只需掌握以下 2 个原则：

1、打开文件使用 open，关闭文件使用 close,使用 print 向文件输出内容。

2、>表示写，<表示读，>>表示在原有内容上增加，Perl 使用这几个符号来表示对文件的处理方式。

比如：

1)读取文件（FH 表示文件句柄，以下几个例子均是）：

open (FH,“<文件名”);

2)写入文件,写入之前，如果文件已经存在，则删除它，重新建立一个新的。

open (FH,“>文件名”);

3)打开一个文件并增加内容。如果文件不存在，则建立一个文件再打开。如果已经存在则直接打开。

Open (FH,”>>文件名”);

3、关闭一个文件

Close (FH);

3. 分割字符串 SPLIT

split 函数功能：把字符串进行分割并把分割后的结果放入数组中。

语法格式如下：

split(分隔符，被分割的字符串)

范例：

```
#perl1-15.pl
#以：为分隔符，将$mytext 按姓名分割成 3 个元素放入数组中，并显示出来 。
$mytext=”张三:李四:王五”;
@people=split(“:”, $mytext);
foreach $name(@people){
   print “$name \n”;
}
```

输出结果如下：

张三

李四

王五

4. 读取文本文件

我们首先试试看能不能按每人每行来读取并显示文件内容

```
#perl1-2.pl

use 5.010;

open FH,'<.\perl1-2.txt';

$mytext=<FH>;

@peoples=split '#',$mytext;

foreach  $people(@peoples){
   say $people;
}
close FH;
```

运行一下，效果不错

```
刘欢欢,20,长沙
冯军,25,上海
李兵,21,北京
李军,23,北京
李志,27,北京
黄王兵,29,长沙
赵兵,22,上海
李强兵,25,上海
```

我们来解释一下这段 perl 程序:

第 1 行：use 后接 perl 的版本号，表示使用 perl5.1。

第 2 行：打开一个文件，并给予一个文件句柄，Perl 对文件的处理方式是操作文件句柄，而不直接操作文件名，你可以把它理解为一个文件的别名。根据 Perl 的文件处理的第 2 个原则，perl1-2.txt 表示处理的文件名，<是处理方式为读取。open 的调用方式为 open 句柄,处理方式后接文件名

第 3 行：这一行表示打开 FH 文件句柄所代表的文件，然后，把文件内容赋值给$mytext 变量。

第 4 行：从变量名的前缀@可以看出@peoples 是一个数组，那么 split('#', $mytext)是什么？split 是一个函数，这个函数的功能是从$mytext 变量的文本中取得被#分割的部分,即:取得每个人的具体情况。

第 5 行：foreach 表示从一个循环，在循环中每次从@peoples 数组中取得一个元素值给$people，

第 6 行：循环体，唯一的任务就是将每个取得的元素打印出来，因为这些元素已经放在$people 中了。say 和 print 的不同之处在于：say 在输出时会在结尾处加上一个换行，print 不会。如果使用 print，可以将 say $people;改成 print “$people\n”；

细心的你一定发现了 “$people\n”这种写法，$people 不是变量名吗，怎么能直接放在双引号中呢，Perl 管这个叫变量内插，在双引号的变量最后会以变量值表示。

第 7 行：循环结束

第 8 行：表示关闭文件句柄 FH。

5. 单引号与双引号

等等，还有问题，单引号呢?除了双引号，单引号也可以表示字符串呀，答案是单引号内的变量不会内插，会直接以变量名表示。最后一个小提示，在 Perl 字符串的连接可以使用小数点符号.解决。如:“hello,”.’”’ world”表示’”’ helloworld”。

这节结束之前我们把上面代码改一下，看看单引号和双引号的区别：

```
#perl1-3.pl
use 5.010;
$|=1;
open FH,'<.\perl1-2.txt';
$mytext=<FH>;
@peoples=split '#',$mytext;
foreach  $people(@peoples){
```

```
    print '$people:'."$people\n";
}
close FH;
```

从以下运行结果可以看出，单引号直接将变量名显示出来了。

$people:刘欢欢,20,长沙

$people:冯军,25,上海

$people:李兵,21,北京

$people:李军,23,北京

$people:李志,27,北京

$people:黄王兵,29,长沙

$people:赵兵,22,上海

$people:李强兵,25,上海

从上面程序中，你也许看出来了：

1、“=”这个符号太神奇了，后面接文件句柄，就能读取文件

```
$mytext=<FH>;
```

但是，上面这个语句只适于文件只有一行的情况（所谓一行是指从行首一直到换行符为止），因为$mytext 是标量，只能保存一个元素。 文件超过一行，应使用数组方式：

```
@myext=<FH>;#Perl 返回所有内容给@mytext，数组每个元素是文件的一行。
```

2、foreach 自动从数组中抽取元素。

```
foreach  $people(@peoples){
    say $people;
}
```

学以致用，我们尝试一下：读取 ActivePerl 目录下的 Copyright.html。

```
#perl1-4.pl
#笔者的 ActivePerl 安装在 d 盘下
use 5.010;
open FH,'<D:/Perl/html/Copyright.html';
@mytexts=<FH>;

foreach  $mytext(@mytexts){
  print $mytext;
}
```

网页文件成功读出，结果如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0  Transitional//EN">

<html>

<head>
<!-- saved from url=(0017)http://localhost/ -->
<script language="JavaScript" src="./displayToc.js"></script>
<script language="JavaScript" src="./tocParas.js"></script>
<script language="JavaScript" src="./tocTab.js"></script>
```

```
<title>Welcome to ActivePerl</title>

<link rel="stylesheet" href="Active.css" type="text/css">

</head>
```

7. SAY、PRINT、<STDIN>与 CHOMP

2 个新疑问产生了，这里为什么不用 say，好吧，我们通过下面的小例子解释这一切

新建一个 perl1-4.txt 的文本文件，内容如下：

第 1 行

第 2 行

第 3 行

第 4 行

我们分别用 say 和 print 完成对这个文件的读取

say 版本：

```
#perl1-4.pl

#笔者的 ActivePerl 安装在 d 盘下

use 5.010;

open FH,'<./perl1-4.txt';

@mytexts=<FH>;

foreach  $mytext(@mytexts){

  say $mytext;

}
```

程序运行结果如下：

第 1 行

第 2 行

第 3 行

第 4 行

print 版本：

```
#perl1-5.pl
#笔者的 ActivePerl 安装在 d 盘下
use 5.010;
open FH,'<./perl1-4.txt';
@mytexts=<FH>;
foreach  $mytext(@mytexts){
  print "$mytext";
}
```

程序运行结果如下：

第 1 行

第 2 行

第 3 行

第 4 行

say 版本在每行后输出一个回车换行，因为 perl1-4.txt 文件本身每行有一个换行符，因此，每读取文件一行就会多输出 1 行，这也是 say 和 print 的主要区别。某些情况下，我们并不需要换行符，即使有换行符也要去掉，有一个函数能帮我们这个忙：chomp。我们打开 perl1-5.txt，可以看到内容如下：

西红柿

黄瓜

丝瓜

冬瓜

白菜

鸡蛋

我们完成一个小任务，将这些人名在一行内输出

```
#perl1-6.pl

use 5.010;

open FH,'<./perl1-4.txt';

@mytexts=<FH>;

foreach  $mytext(@mytexts){

  chomp($mytext);

 print “$mytext,”;

}
```

输出如下：

西红柿 黄瓜 丝瓜 冬瓜 白菜 鸡蛋

因为 foreach 循环一次，从@mytexts 中取得下一个元素放在$mytext 中，这个过程并不包括删除每个元素的换行符，这个例子中每个元素就是一行，自然包括换行符，所以必须要使用 chomp 函数去掉末尾的换行符：chomp($mytext)。

Chomp 还有一个地方用得最多，就是接受用户输入时。Perl 处理用户键盘输入的语句形式如下：

变量=<STDIN>

我们先看一段程序，这段程序的本意是接受用户输入，并将用户和用户的表妹的年纪在一行内显示出来。

```
#perl1-7.pl

print "你表妹多大了？";
```

```
$myinput1=<STDIN>;

print "你今年多大了？";

$myinput2=<STDIN>;

print "你的年纪是$myinput1,";

print "你表妹的年纪是$myinput2。";

$myinput=<STDIN>;
```

上面程序中的$myinput1=<STDIN>表示接受输入，并将输入的内容赋值给$myinput1。输出结果如下：

```
你表妹多大了？ 15

你今年多大了？ 22

你的年纪是 15

,你表妹的年纪是 22
```

程序并没有按想像中的运行，用户和其表妹的年纪在 2 行内显示出来，因为 Perl 接受输入，但并不包括清除用户完成输入最后敲入的换行符。因为应去掉$myinput1 所含的换行符。程序修改如下：

```
#perl1-8.pl

print "你表妹多大了？";

$myinput1=<STDIN>;

print "你今年多大了？";

$myinput2=<STDIN>;

chomp($myinput1);

print "你的年纪是$myinput1,";

print "你表妹的年纪是$myinput2";
```

输出结果如下，任务完成。

```
你表妹多大了？ 15

你今年多大了？ 22
```

你的年纪是 15,你表妹的年纪是 22。

8. 读取每行的多列数据

上面的例子涉及的都是单列数据，而多数情况下，我们需要读取文件的多列数据。

我从朋友那得到一份北京现代汽车的报价清单，做为消费者，想从中挑选出 10-15 万的车型，并显示出来。

清单文件为 perl1-9.csv，内容如下：

伊兰特 悦动 1.6GL 手动舒适型 2010 款,9.98

伊兰特 悦动 1.6GL 自动舒适型 2010 款,10.88

伊兰特 悦动 1.6GLS 手动豪华型 2010 款,10.98

伊兰特 悦动 1.6GLS 自动豪华型 2010 款,11.98

伊兰特 悦动 1.8GLS 手动豪华型 2010 款,12.18

伊兰特 悦动 1.8GLS 自动豪华型 2010 款,12.98

……

…...

仔细观察可以发现，型号和价格是用“，”分隔，我们再次使用 split 函数。

```
#perl1-9.pl

use 5.010;

open CAR,'<.\perl1-9.csv';

foreach $carmes(<CAR>){

    ($name,$price)=split(',',$carmes);

        chomp($price);

        say "$name:$price 万" if $price<15 and $price>10;

}

close CAR;
```

程序运行结果如下(……表示以下有若干行省略没列出)：

伊兰特 悦动 1.6GL 自动舒适型 2010 款:10.88 万

伊兰特 悦动 1.6GLS 自动舒适型 2010 款:10.98 万

伊兰特 悦动 1.6GLS 自动豪华型 2010 款:11.98 万

伊兰特 悦动 1.8GLS 手动豪华型 2010 款:12.18 万

伊兰特 悦动 1.8GLS 自动豪华型 2010 款:12.98 万

……

……

这段程序有几处耐人寻味：

1、($name,$price)=split(',',$carmes);

($name,$price)表示一个列表，列表可以理解为标量的有序集合，列表存储着数据的集合，而数组可理解为存储着列表的变量。列表可用用(元素 1，元素 2，元素 3,…..,元素 n)的方式表示。这句代码完成一个功能把等号右边分隔的每个元素赋值给左边的列表，其中第 1 个元素给$name，第二个元素给$price。

2、chomp($price);

$price 难道会包括换行符？这是初学者最容易忽视的问题：文件中每行只有 2 个元素，最后一个元素（也就是第二个元素）的末尾包括换行符。

3、foreach $carmes(<CAR>)

从 CAR 文件句柄中读取数据给$carmes 变量，每循环一次读取一行，也可以如下书写：

```
@mytext=<CAR>;
foreach $carmes(@mytext){
  ……
}
```

4、say "$name:$price 万" if $price<15 and $price>10;

想必大家对 if 语句并不陌生，这是 if 的一种特殊用法，此语句的含义表示：如果$price<15 并且$price>10，则执行 say "$name:$price 万"语句。关于 if，我们将在下一章详述。

9. 写文件

Perl 写文件的方式非常简洁，使用以下 2 种形式：

say 文件句柄 内容

print 文件句柄 内容

把 Perl1-9 的代码稍加修改，将 10 万到 15 万之间车写入另一个 CSV 格式文件中。

```
#perl1-10.pl

use 5.010;

open CAR,'<.\perl1-9.csv';

open CAROUT,'>.\perl1-10.txt';#以写入方式打开文件 perl1-10.txt

foreach $carmes(<CAR>){

   ($name,$price)=split(',',$carmes);

        chomp($price);

        say CAROUT "$name:$price 万" if $price<15 and $price>10;#写文件

}

close CAR;
```

10.带格式输出 SPRINTF 和 PRINTF

Sprintf 返回格式化后的字符串，语法如下：

sprintf(“格式”,要格式的字符串列表)

格式主要有以下几种：

%%　　百分号

%c　　把给定的数字转化为字符

%s 字符串

%d 带符号整数，十进制

%u 无符号整数，十进制

%o 无符号整数，八进制

%x 无符号整数，十六进制

%e 浮点数，科学计算法

%f 浮点数，用于固定十进制计数

%g 浮点数，包括%e 和%f

sprinf 等同于以下语句：

print sprintf(“格式”,要格式的字符串列表);

我们使用 sprintf，以元为单位输出 10 万到 15 万的车的价格列表。

```
#perl1-11.pl
use 5.010;
open CAR,'<.\perl1-9.csv';
foreach $carmes(<CAR>){
   ($name,$price)=split(',',$carmes);
       chomp($price);
       say $name.':'.sprintf(“%d”,$price*10000)  if $price<15 and $price>10;
# $price*10000 完成将万元转换成元
 }
close CAR;
```

11. JOIN

split 和 join 函数在处理文本时经常使用，join 的功能与 split 相反，它将列表中用逗号分隔的字符串元素连接成单个字符串，并返回这个字符串，调用格式如下：

变量=join(分隔符,字符串元素列表);

我们将 perl1-11.pl 修改一下，从 perl1-9.csv 中提取价格在 10 万到 12 万之间的汽车，输出为另一个 CSV 格式文件。

```
#perl1-12.pl

use 5.010;

open CAR,'<.\perl1-9.csv';

open CAROUT,'>.\perl1-11.csv' ;以写入方式打开文件 perl1-11.csv

foreach $carmes(<CAR>){

   ($name,$price)=split(',',$carmes);

      chomp($price);

      say CAROUT   join( ',' ,$name,sprintf("%d",$price*10000))

      if $price<12 and $price>10;

      # $price*10000 完成将万元转换成元

 }

close CAR;

close CAROUT;
```

注意下面这个语句：

```
say CAROUT   join( ',' ,$name,sprintf("%d",$price*10000))

      if $price<12 and $price>10;
```

因为 csv 格式每行元素之间以逗号分隔，所以使用逗号做为分隔符，将$name 和格式转换后的$price 连接为单个字符串输出到 CAROUT 文件句柄中。

12. 转义字符表示

Perl 的主要转义字符如下：

结 构	含 义
\n	换行
\r	回车
\t	水平置表符
\f	换页符
\b	退格
\v	垂直置表符
\a	响铃
\e	Esc
\007	任一八进制 ASCII 值(这里 007 表示 bell)
\x7f	任一十六进制 ASCII 值
\cC	任一“控制”字符
\\	反斜杠
\″	双引号
\l	下一字母小写
\L	以后所有字母小写直到\E
\u	下一字母大写
\U	以后所有字母大写直到\E
\E	结束\L 和\U

13. 引用、符号引用、指针

Perl 没有像 C 语言一样的可直接操纵内存的指针，但有类似于指针的的引用(也称为硬引用)。引用是一种标量，含有其他类型数据的地址，可把它理解为一个内容为地址值的变量。

在 perl 里面使用“\”来创建引用，使用$后接硬引用方式直接使用引用指向的变量，并且这种使用方式是具有修改权力的。如下面的例子：

```
#perl1-13.pl

$hello="hello";

$myhello=\$hello;

print "$myhello\n";#输出$hello 地址

print "$$myhello\n";#输出$hello 变量内容

$$myhello="hello!";#通过$myhello 引用修改$hello 变量的值

print "$hello\n";#输出$hello 变量内容
```

输出结果如下：

```
SCALAR(0x182a6f4)

hello

hello!
```

这个程序中，$myhello 存有$hello 的地址，存有$hello 变量的地址，通过$myhello 能直接操作$hello，而不是操作$hello 的复制版本。

符号引用（也称为软引用）操作符为$，相当于宏替换，如以下代码中，$$var 相当于取$var 的内容为变量名，完成宏替换，最终表示$name。

```
$var="name";

$$var="张三";

print "$name\n";
```

输出结果为

```
张三
```

14.在子程序中使用引用传参

到目前为止，我们只介绍了如何给传给子程序值，子程序对传进来的参数进行有效修改时，需要使用引用传参。

我们以完成加法子程序为例，传 3 个参数给该子程序，前 2 个参数是要计算的数，第 3 个参数是计算结果。

```
#perl1-16.pl
sub add{
    ($num1,$num2,$jg)=@_;
     $$jg=$num1+$num2;
   #注意$jg 是个引用，通过$后接引用($$jg)使用引用($jg)指向的变量，即解引用
}
$jg=0;
&add(5,10,\$jg);
print $jg;
```

输出为 15

15. 完成第一个任务

这一章要结束了，输入输出基础介绍完了，能完成本章开始时提出的任务了，将 Perl1-2.txt 中的人员按城市分别存为几个文本文件，文件名是城市名。

```
#perl1-14.pl
use 5.010;
open PEOPLE,'<.\perl1-2.txt';
$mytext=<PEOPLE>;
@peoples=split('#', $mytext);
foreach $peoplemes(@peoples){
    ($name,$age,$city)= split(',',$peoplemes);
    chomp($city);
```

```
    open CITY,">>$city.txt";
    say CITY join(',',$name,$age,$city);
    close CITY;
}
close PEOPLE;
```

请大家注意这个语句：open CITY,">>$city.txt";

打开文件的方式使用的是增加方式。根据 Perl 的文件处理原则，>>表示在原有内容上增加。

第四章 哈希与数组

1. USE STRICT 和 USE WARNINGS

1.1 要求

use strict 强制所有变量必须用 my 来声明，当试图使用不是用 my 声明的裸单词(变量或函数名)时，编译无法通过。如果使用这个选项时不能使用符号引用，因此 Perl 5 推荐使用引用代替软引用。

use warnings 在 Perl 程序运行时打开某些警告。

通常来说，这 2 个命令总是同时使用，并且书写在程序的最前面。

1.2 作用

程序员有时会输错变量名、函数名等， use strict 和 use warnings 对程序员进行了约束，避免了这类简单错误发生，也许算是懒惰的程序员的恶梦吧，毕竟会导致多敲 N 次键盘。

笔者认为，与其将大量时间花在程序调试排错上，不如约束程序员，使他们少犯简单错误，在工程规模较大的情况下，这种结束效果非常明显。比如以下 test.pl 的程序(因为程序员疏忽，将$sum+=$l 中的$i 输成了$l)

```
#test.pl
for ($i=1;$i<100;$i++){
    $sum+=$l;
}
print $sum;
```

程序输出 0，但这并不是程序需要的求和结果。

使用 use strict 和 use warnings 后,程序改成如下：

```
#test.pl
use strict;
use warnings;
my $i;
my $sum;
for ($i=1;$i<100;$i++){
    $sum+=$l;
}
print $sum;
```

程序提示错误：

```
Global symbol "$l" requires explicit package name at test.pl line 7.
Execution of test.pl aborted due to compilation errors.
```

得益于 use strict 和 use warnings，不习惯使用变量前事先声明的懒惰程序员犯的错误被轻易发现了。

2. 哈希

2.1 什么是哈希

哈希是一种数据结构，以数字、字符串等为索引将值存放到其中，或者从中取回值。哈希把任意长度的输入(键)，通过散列算法，变换成固定长度的散列值(值)输出，将输入视为索引，将输出视为给索引分配的唯一内存地址，地址里存放的是这个索引代表的内容。

2.2 访问哈希

2.2.1 访问语法

2.2.1.1 读取哈希(HASH) 元素

语法如下：

$hash 变量名{哈希键}

如：

```
$studentage=$age{张三};
```

2.2.1.2 修改或增加哈希(HASH)元素

语法如下：

$hash 变量名{哈希键}=键值。

如：

$age{张三}=28;

2.2.1.3 访问不存在的 HASH 键值

不存在的 hash 键值的访问结果是 undef，undef 即未定义。

2.2.2 哈希变量声明

对于使用了 use strict 语句的程序，必须事先声明变量，语法如下：
变量范围 %哈希变量名

如：

my %studentage;

2.2.3 哈希拷贝与反转

哈希之间的拷贝语法如下：
%新变量名 = %被拷贝的变量名;

如：

%newstudentage=%studentage;

哈希反转语法如下：

%newstudentage=reverse %studentage;

哈希的反转指哈希键和哈希值调换，例如：

```
my %student;
my %newstudent;
%student=("张三"=>"上海","李四"=>"广州","王五"=>"北京");
print   "$student{'张三'}\n";
%newstudent=reverse   %student;
print   "$newstudent{'上海'}\n";
```

输出如下：

```
上海
张三
```

上述代码中，未反转之前，张三为键，上海为值，反转后，上海为键，张三为值。

2.2.4 哈希赋值

使用大箭头符号(=>)对 HASH 赋值，箭头左边是键，右边是值。

语法如下：

my %变量名=(

键名 1=>键值 1,

键名 2=>键值 2,

……

……

键名 n=>键值 n

)

如：

```
#perl3-1.pl
use strict;
use warnings;
#哈希赋值
my %studentage;
%studentage=(
"张三"=>19,
"李四"=>22
);
#访问哈希
print "张三:$studentage{'张三'}";
print "\n";
```

2.3 哈希内嵌哈希

很多情况下，需要几层哈希来存储一个数据表，比如以下学生成绩表：

姓名	物理	化学	数学
张三	85	79	90

李四	77	96	79
王五	86	81	93

这个成绩表需要 2 层哈希，所谓哈希内插哈希。第 1 层哈希存储行，即学生姓名。第 2 层哈希，存储每行的信息，即学生的每门成绩。2 层哈希如何结合起来存储学生成绩表，并提供相关访问呢？

首先，我们来解决存储问题。

```
#第 1 层哈希

my %student;

#第 2 层哈希

%student=(

   “张三”=>{“物理”=>85,“化学”=>79,“数学”=>90},

   “李四”=>{“物理”=>77,“化学”=>96,“数学”=>79},

   “王五”=>{“物理”=>86,“化学”=>81,“数学”=>93},

);
```

在上述代码中，%student 由 3 个键组成：张三、李四、王五，与这些键相关的值则位于花括号中，并含有嵌套的哈希（即：键值对）。张三键的相应值中含有 3 个嵌套的键：物理、化学、数学，其值分别是 85、79、90。

李四键的相应值中含有 3 个嵌套的键：物理、化学、数学，其值分别是 77、96、79，王五也同理。

上述代码还可写成：

```
#第 1 层哈希

my %student;

#第 2 层哈希

$student{“张三”}={“物理”=>85,“化学”=>79,“数学”=>90};

$student{“李四”}={“物理”=>77,“化学”=>96,“数学”=>79};

$student{“王五”}={“物理”=>86,“化学”=>81,“数学”=>93};
```

其次，访问这 2 层哈希，比如访问王五同学的数学成绩。

第一种方式：

```
print $student{“王五”}{“数学”};
```

第二种方式:

my $studentreport=$student{“王五”};#取得第一层 HASH

print ${$studentreport }{“数学”};#访问第二层 HASH

总结以上方式形成以下 2 个程序:

perl3-2.pl:

```
#perl3-2.pl
#第 1 层哈希
my %student;
#第 2 层哈希
%student=(
  "张三"=>{"物理"=>85,"化学"=>79,"数学"=>90},
  "李四"=>{"物理"=>77,"化学"=>96,"数学"=>79},
  "王五"=>{"物理"=>86,"化学"=>81,"数学"=>93},
);
print $student{"李四"}{"化学"};
```

perl3-3.pl:

```
#perl3-3.pl
#第 1 层哈希
my %student;
#第 2 层哈希
$student{"张三"}={"物理"=>85,"化学"=>79,"数学"=>90};
$student{"李四"}={"物理"=>77,"化学"=>96,"数学"=>79};
$student{"王五"}={"物理"=>86,"化学"=>81,"数学"=>93};
#访问
my $studentreport=$student{"李四"};
print ${$studentreport}{"物理"};
```

其实还有第 3 种方法，通过 PERL 的硬引用操作符\完成内嵌哈希的赋值。改写以上代码如下：

```
#第 1 层哈希
my %student;
#第 2 层哈希
    my %cj=("物理"=>85,"化学"=>79,"数学"=>90);
    $student{"张三"}=\%cj;
    my %cj=("物理"=>77,"化学"=>96,"数学"=>79);
    $student{"李四"}= \%cj;
    my %cj=("物理"=>86,"化学"=>81,"数学"=>93);
    $student{"王五"}=\%cj;
#访问
my $studentreport=$student{"李四"};
print ${$studentreport}{"物理"};
```

3. 数组

3.1 列表

列表是标量的有序集合，但不能把列表看成一个变量，数组才是变量，数组中存储列表。

列表的构造方法有以下 2 种：

1) (元素 1,元素 2，元素 3)

   构造由 3 个元素组成的列表。元素类型可以不一致，比如：（‘abc’,123,’xyz’）

2) qw(字符串 1　字符串 2　字符串 3)

   构造由 3 个字符串组成的列表，做为列表元素的字符串，不必加上引号，且以空格隔开。qw 构造的列表，Perl 会将其中元素视为加上单引号的字符串，比如：

   qw(abc　xyz)

   qw 构造中的()实际为定界符，如果你愿意，也可以写成：

   qw/abc xyz/　　以“/”为定界符

qw! abc xyz!　　以“!”为定界符

3.2 数组声明与赋值

数组声明格式如下：

my　@数组变量名;

数组赋值方式如下：

1) @数组变量名=(元素 1，元素 2，元素 3，...,元素 n);

2) @数组变量名=qw (元素 1，元素 2，元素 3，...,元素 n);

比如，声明学生姓名数组：

my @student;

@student=(“李明”,”张亮”,”王波”,”李安”);

那么对于二维数组，如何处理呢？ 比如，一段曲线坐标点集合，可如下声明赋值：

my　@coordinate;

@coordinate=(

[1.2,3.4],

[1.3,3.2],

[1.6,3.9]

);

数组不要求其元素的数据类型一致，以下赋值方式是合法的：

my @myinfo=(“李四”,“湛江”，”广东”,26,198.22);

对于范围数组的赋值，使用“起点..终点”的格式

比如，赋值元素是从 0 到 10。

my @mynum=(0 .. 10);

my @myalpha=(‘a’ .. ‘z’);

my @myalpha=(‘A’ .. ‘Z’);

3.3 元素访问与修改

语法格式如下：

$数组变量名[下标]

其中下标从 0 开始计数。

例如：

```
my @mynum=(0 .. 10);
print $mynum[3];#访问数组元素
$mynum[3]=100;#修改数组元素
print $mynum[3];#输出修改后的数组元素
```

输出结果如下：

```
3
100
```

4. 哈希内嵌数组

语法格式如下：

声明及定义

```
my %哈希变量=(
哈希键 1=>[数组元素 1，数组元素 2，...，数组元素 n],
哈希键 2=> [数组元素 1，数组元素 2，...，数组元素 n],
.....
......
哈希键 n=> [数组元素 1，数组元素 2，...，数组元素 n]
);
```

访问内嵌数组的单个元素：

$哈希变量名{哈希键}->[数组下标]

访问内嵌数组：

@{$哈希变量名{哈希键}}

例如，游戏地图上有一组建筑物的坐标，分别如下：

武器店：123，35

修练场：85，196

防具店：67,96

魔法店：128,45

用哈希内嵌数组的方式来存储和访问这些数据。

1）存储

```
my %game=(
        “武器店”=>[123,35],
        “修练场”=>[85,196],
        “防具店”=>[67,96],
        “魔法店”=>[128,45]
);
```

2）访问

访问防具店的坐标值:

```
printf ("%d,%d",@{$game{"防具店"}});
```

访问魔法店的坐标 Y 值：

```
print $game{“防具店”}->[1];
```

可使用硬引用符来完成内嵌数组的赋值。

```
my    %game;
my    @array=(123,35);
$game{"武器店"}=\@array;
my    @array=(85,196);
$game{"修练场"}=\@array;
my    @array=(67,96);
$game{"防具店"}=\@array;
my    @array=(67,96);
$game{"魔法店"}=\@array;

printf ("%d,%d",@{$game{"防具店"}});
```

5. 数组内嵌哈希

1、声明及定义

my @数组变量名=(

{

数组第 1 个元素的哈希键 1=>值 1，

数组第 1 个元素的哈希键 2=>值 2,

.....

数组第 1 个元素的 哈希键 n=>值 n

},

{

数组第 2 个元素的哈希键 1=>值 1，

数组第 2 个元素的哈希键 2=>值 2,

.....

数组第 2 个元素的哈希键 n=>值 n

}

....

{

数组第 n 个元素的哈希键 1=>值 1，

数组第 n 个元素的哈希键 2=>值 2,

.....

数组第 n 个元素的哈希键 n=>值 n

}

);

2、访问内嵌哈希的键值：

$数组变量名[数组下标]{哈希键}

例如，我们使用数组内嵌哈希来存储和访问某文具店的库存:

品名	价格	数量
钢笔	8.06	50

2B 铅笔	2.10	300
胶水	5.80	60
剪刀	12.20	28

存储：

```
my  @wenju=(
{name=>" 钢笔",price=>8.06, amount =>50},
{name=>"2B 铅笔",price=>2.10,amont=>300},
{name=>" 胶水",price=>5.80,amont=>60},
{name=>" 剪刀",price=>12.20,amont=>28}
);
```

访问品名和数量：

```
print $wenju[2]{ name };#品名
print $wenju[2]{ amont };#数量
```

6. 数组内嵌数组

1、声明及定义

首先，定义内嵌的数组

```
my  @内嵌数组 1=(元素 1,元素 2,元素 3,...,元素 n);
my  @内嵌数组 2=(元素 1,元素 2,元素 3,...,元素 n);
……
```

接着，使用“\”来创建内嵌的数组的引用，以引用为元素定义外层数组的元素

```
my  @数组=(\@内嵌数组 1,\ @内嵌数组 2,...);
```

2、访问内嵌数组的值：

$数组变量名[1 维数组下标][2 维数组下标]

我们仍以某文具店的库存为例：

```
#perl3-9.pl
#数组内嵌数组
```

```
my @arr1=("  钢笔",8.06,50);

my @arr2=("2B 铅笔",2.10,300);

my @arr3=("胶水",5.80,60);

my   @wenju=(

\@arr1,\@arr2,\@arr3

);

#访问 2B 铅笔数量、价格：

print $wenju[1][1];#  数量

print $wenju[1][2];#  价格
```

输出结果如下:

2.1,300

7. 删除、清空哈希和数组

清空和删除的区别在于，清空某哈希和数组只是置为未定义，而删除哈希和数组后，它们不再存在。

1、清空使用以下语法：

清空整个数组：undef @数组变量名;

清空整个哈希：undef %哈希变量名；

清空哈希的某个键值：undef $哈希变量名{哈希键};

a) 删除使用以下语法：

删除哈希，假设 HASH 变量名为%hash

```
foreach $key (keys %hash) {
        delete %hash {$key};
}
```

删除数组，假设数据变量名为@myarray

```
foreach $index (0 .. $# myarray) {
```

```
        delete $ myarray [$index];
    }
```

8. 哈希的遍历

8.1 第二个任务

同事小王是某 BBS 驾游板块的版主，最近想在成员最多的城市组织聚会。小王将该版块发言成员信息导出为 qqyou.txt 文本文件，其中每行存放一位成员的信息，第一列是姓名，第二列是城市，每列以空格分隔，内容如下：

王兴 湛江
张华 深圳
李光 广州
王安 湛江
张美 深圳
张江美 深圳
胡海明 深圳
黄国刚 深圳
刘灿 湛江
李发 广州

哈希的遍历可通过 while 循环完成，每次循环从哈希变量中取出一对键值。语法如下：

```
while (( $键名,$键值)=each %哈希变量名){
        #对取出的键名值进行操作
}
```

我们来完成第二个任务，代码如下:

```
#perl3-4.pl
use strict;
use warnings;
my %zjycity;
my $mycity;
my $cityinfo;
my $name;
my $city;
my $zjycity;
my $citycount;
my $pcount=0;
my @citys,;

open CITY,"<perl3-4.txt";
#从 perl3-4.txt 文件中读成员的姓名及城市信息
```

```
foreach $cityinfo(<CITY>){
    ($name,$city)=split(' ',$cityinfo);
    $zjycity{$city}++;
}
#循环遍历哈希，并找到成员数最多的城市
while (($zjycity,$citycount)=each %zjycity){ #$zjycity 为城市名，$citycount 为成员数
    if ($citycount>$pcount) {
        $pcount=$citycount;
        $mycity=$zjycity;
     }
}
print "最佳聚会地点：$mycity\n" ;
```

以上代码中，%zjycity 为哈希变量，首先从文本文件 perl3-4.txt 读取数据，并生成城市统计哈希%zjycity，键为城市名，键值为城市的成员数目，然后通过循环遍历哈希变量%zjycity，找到成员数最多的城市，即找到最大键值对应的键名。

8.2 第三个任务

BOSS 交给小张一个文本文件，为去年销售人员季度完成的销售额，要求小张统计出前三名做为金牌销售员人选。

```
张三  25   9    10   60
李四  17   16   79   25
黄三  5    10   5    2
黄光  10   3    6    9
刘发  6    5    9    12
```

以上文件文件第一列是姓名，第二列是第一季度销售额，第三列是第二季度销售额，第四列是第三季度销售额，第五列是第四季度销售额，每列用 tab 分隔。

8.2.1 遍历内嵌哈希

完成这个任务需要用到内嵌哈希的遍历，语法如下：

```
while (( $键名,$键值)=each %哈希变量名){
    #对取出的键名值进行操作，其中键值是内嵌哈希的引用。
    #访问内嵌哈希键值的语法为：${$键值}{$内嵌哈希的键名};
}
```

我们先计算每个人的全年的总销售额并输出：

```
#perl3-5.pl

use strict;

use warnings;

use 5.010;

my   %p;
```

```
open PINFO,"<perl3-5.txt";
#从 perl3-5.txt 文件中读成员的姓名及销售额信息
foreach my $pinfo(<PINFO>){
        chomp($pinfo);
         my    %info;
        (my $name, my $jd1,my $jd2,my $jd3,my $jd4)=split('\t',$pinfo);
        $info{jd1}= $jd1;
        $info{jd2}= $jd2;
        $info{jd3}= $jd3;
        $info{jd4}= $jd4;
        $info{all}= $jd1+$jd2+$jd3+$jd4;
        $p{$name}=\%info;
}
close PINFO;
while    ((my $key,my $value)=each    %p){
          say "$key=>${$value}{all}";
}
```

结果如下：

刘发=>32

张三=>104

李四=>137

黄光=>28

黄三=>22

8.2.2 EACH、VALUES、KEYS、SORT

1、each

each 一般作用于哈希和数组，以 2 元素的列表形式返回哈希的键值对和数组的索引、值对。语法如下：

each %hash

each @array

比如：

```
while (($key, $value) = each %hash) {#$key 取得键，$value 取得值
    print   “$key:$value\n";
}
while (($index, $value) = each @array) {#$index 取得索引，$value 取得值
    print   "$index:$value\n";
}
```

2、values

values 函数以列表形式返回哈希和数组的所有值。语法如下：

values %hash

values @array

比如：

```
for   $value(values %hash)
{ #输出 hash 的所有值(注意不是健值对)。
    print “$value\n”;
}
```

3、keys

keys 函数以列表形式返回哈希和数组的所有健。语法如下：

keys %hash

keys @array

比如：

```
for    $key(keys %hash)
{ #输出 hash 的所有键(注意不是健值对)。
    print "$key\n";
}
```

以一段程序为例说明 keys 和 values 的用法：

```
$hash{A}=0;
$hash{B}=1;
$array[0]="a";
$array[1]="b";
for    $value(values %hash)
{ #输出 hash 的所有值(注意不是健值对)。
    print "$value\n";
}
for    $key(keys %hash)
{ #输出 hash 的所有键(注意不是健值对)。
    print "$key\n";
}
for    $value(values @array)
{ #输出数组的所有值，注意不是索引、值对)。
    print "$value\n";
}
for    $index(keys @array)
{ #输出数组的所有索引，注意不是索引、值对)。
```

```
    print "$index\n";
}
```

输出结果如下：

0

1

A

B

a

b

0

1

4、 sort

sort 实现对列表排序后返回列表，主要有以下几种用法：

1）sort 列表

按普通字符串比较排序.比如：

```
my @myarr;
@myarr= sort @myarr;
```

2) sort 子程序名 列表

如果使用了子程序名，可以实现自定义排序，主要分为数字排序、字符串排序和综合排序。$a 和$b 代表需要比较列表的 2 个元素。

首先编写排序用函数。

A) 数字排序

```
sub by_number{$a<=>$b};#<=>是飞碟操作符，比较 2 个数字返回-1,0,1
```

B) 字符串排序

```
sub by_str{$a cmp $b};
```

C) 综合排序

定义一个子程序，返回 1、0、-1。经过子程序判断，第一个参数($a)位于第二个参数($b)之前，返回-1,否则返回 1,如果顺序不分先后，返回 0。

```
sub by_num{#定义一个数字升序排序
  if ($a<$b) {-1} elsif ($a>$b) {1} else {0};
}
```

然后实施排序，以数字排序为例。

```
 my @myarr;
 @myarr= sort by_number @myarr;
```

8.2.3 完成第三个任务

我们利用 sort 排序完成内嵌哈希值的排序

哈希排序利用

```
#perl3-6.pl
use strict;
use warnings;
use 5.010;
my    %po;
open PINFO,"<perl3-5.txt";
#从 perl3-5.txt 文件中读成员的姓名及城市信息
foreach my $pinfo(<PINFO>){
        chomp($pinfo);
        my    %info;
       (my $name, my $jd1,my $jd2,my $jd3,my $jd4)=split('\t',$pinfo);
       $info{jd1}= $jd1;
```

```
        $info{jd2}= $jd2;
        $info{jd3}= $jd3;
        $info{jd4}= $jd4;
        $info{all}= $jd1+$jd2+$jd3+$jd4;
        $po{$name}=\%info;
}

foreach my $key (reverse sort {${$po{$a}}{all}<=>${$po{$b}}{all}} keys %po){
    print "$key=>${$po{$key}}{all}\n";
}
```

1.1 PUSH、POP、SHIFT、UNSHIFT

1、push 在数组末尾处增加元素，pop 就是从数组的末尾取出元素。具体语法如下：

push ARRAY, (以逗号分隔的要增加的值列表)
push ARRAY,要增加的值

```
pop ARRAY
```

比如：

```
my @myarr;
push @myarr,-1;
push @myarr,0;
push @myarr,1;
print pop @myarr;
print "\n";
print pop @myarr;
print "\n";
```

```
print pop @myarr;

print "\n";
```

输出如下 ：

```
1

0

-1
```

2、unshift 在数据的起始增加元素，shift 在数组的起始取出一个元素。shift 弹出数组的值，并返回它，unshift 的队列中增加值。具体语法如下：

```
unshift (ARRAY,(以逗号分隔的要增加的值列表))

unshift ARRAY,要增加的值


shift ARRAY
```

比如：

```
my @myarr;


unshift @myarr,-1;

unshift @myarr,0;

unshift @myarr,1;

print shift @myarr;

print "\n";

print shift @myarr;

print "\n";

print shift @myarr;
```

```
print "\n";
```

输出如下：

```
1

0

-1
```

3、通过对 push、pop、shift、unshift 使用组合，可以实现堆栈和队列的操作。

1）堆栈，后进先出，通过 push 和 shift 组合实现

```
my @myarr;

push @myarr,-1;

push @myarr,0;

push @myarr,1;

print pop @myarr;

print "\n";

print pop @myarr;

print "\n";

print pop @myarr;

print "\n";
```

输出如下：

```
1

0

-1
```

2)队列，先进先出，通过 push 和 shift 组合实现

```
my @myarr;


push @myarr,-1;

push @myarr,0;

push @myarr,1;

print shift @myarr;

print "\n";

print shift @myarr;

print "\n";

print shift @myarr;

print "\n";
```

输出如下：

```
-1

0

1
```

1.2 第四个任务

BOSS 要求编写一个脚本，搜索某目录下所有以 PHP 扩展名结尾的源代码文件，并生成一个 HTML 文件，其内容是按目录归类的文件列表。

我们先简单介绍一下如何操作目录及文件：

通常使用 opendir、closedir 、readdir 函数操作目录，格式如下：

```
opendir(DH, $some_dir) || die "can't opendir $some_dir: $!";
```

#opendir 打开目录，将目录句柄赋值给 DH。$some_dir 为要打开的目录名。

#开始循环读取目录下的文件

```
    foreach    $file(readdir DH){#读取目录下的文件,readdir 返回文件列表
        if ( -d $file){
            # -d  文件名称  检测$file 储存的文件名称是否为目录。
           #$file 是子目录，对子目录进行处理
       }
       else {
          #$file 是文件，对文件进行处理。
       }
  }
  closedir DH;#关闭目录句柄
```

我们的思路是：

1) 打开要读取的目录。
2) 循环读取目录中所有文件，将新发现的子目录存储在数组中。
3) 继续取出数组中的目录，如果数组中没有元素，转到第 4 步，否则转到第 1 步。
4) 退出。

存放目录的数组采用堆栈的方式操作，可模拟深度优先搜索，较好地展示目录树结构。代码如下：

```
#perl3-8.pl

use 5.010;

use warnings;

use strict;

my $dirs='E:\TDDOWNLOAD\latest';#  初始路径

my $lsfile;

my @filedir;

my %filediryj;#已经访问过的目录 ，哈希键值对为：目录名=>文件数目
```

```
my $flcount;

my $ppfile;#文件匹配符  以正则方式表达的

open MYTXT,">phpfiles.htm";

#输出 HTML 文件头

say MYTXT '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">';

say MYTXT '<html xmlns="http://www.w3.org/1999/xhtml">      ';

say MYTXT '<head> ';

say MYTXT '<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
   ';

say MYTXT '<title>查找某目录下的某类文件</title>    ';

say MYTXT '</head> ';

say MYTXT '<body> ';



$ppfile='php$'; #要寻找文件扩展名为 php，php 后紧跟符号$表示文件名以 php 结尾。

push @filedir,$dirs;

$|=1;

my $filedir;

while ($filedir=shift(@filedir)){

    opendir DH,$filedir or die "不能打开$filedir"; #打开目录

    say MYTXT "<br>$filedir<br>";

    $flcount=0;

    foreach $lsfile(readdir DH){#读取目录

        #变量名=~m/正则表达式/  表示对变量进行正则表达式匹配，

        #如果匹配成功，则返回真，否则返回假

        next if $lsfile=~m/^\./;#~m/^\./对$lsfile 进行正则表达式匹配，^\.表示以点号开头
```

```
        # .和..均以点号开头，分别表示当前路径和上一路径，它们不属于子目录和文件。
        $lsfile="$filedir\\$lsfile";#”\”有特殊意义，因此必须以\\表示符号\
        if ( -d $lsfile){
            push @filedir,$lsfile;
        }
        elsif ($lsfile=~m/$ppfile/i) {
            $flcount++;
            print '.'; #在屏幕上输出进度
            say MYTXT
    "<br>           &nb
    sp;|$lsfile<br>";#匹配文件名写入 HTML 文件的 body 部分
        }
    }
    close DH;
    $filediryj{$filedir}=$flcount;#目录下匹配文件数目
    say MYTXT
    "<br>           &nb
    sp;|共$flcount 个匹配文件<br>";
}
#输出 HTML 文件尾
say MYTXT '</body>    ';
say MYTXT '</html>    ';
```

用浏览器打开输出的 HTML，效果如下：

E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Documentation\images

|共0个匹配文件

E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Upload\admin

|E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Upload\admin\index.php

|共1个匹配文件

E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Upload\archive

|E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Upload\archive\global.php

|E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Upload\archive\index.php

|共2个匹配文件

E:\TDDOWNLOAD\latest\MyBB1413chsFPbuild100419utf8\Upload\cache

|共0个匹配文件

第五章 正则表达式

1. PERL 正则基础

正则表达式是一种序列或字符模式，负责在搜索和替换文本时对文本内容进行字符串匹配。Perl 中的正则表达式由待匹配字符串或模式串或 2 者混合而成，一般以斜杠（/）作为定界符。

Perl 正则模式串主要有以下几种：

. 匹配单个除换行符以外的字符

a? 匹配 0 次或一次 a 字符(a 为任意字符，如:a*、b*、2*等)

a* 匹配 0 次或多次 a 字符(a 为任意字符，如:a*、b*、2*等)

a+ 匹配 1 次或多次 a 字符(a 为任意字符，如:a+、b+、2+等)

.* 匹配 0 次或一次的任何字符

.+ 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

{m,n} 匹配在 m 个 以上 n 个 以下 的指定字符

{m,} 匹配 m 个 以上 的指定字符

[] 匹配符合 [] 内的字符

[^] 匹配不符合 [] 内的字符

(x|y) 匹配 x 模式串或 y 模式串

(x) 匹配 x 模式串

[0-9] 匹配单个数字

[a-z] 匹配单个小写字母

[^0-9] 匹配单个非数字字符

[^a-z] 匹配单个非小写字母字符

^ 匹配字符开头的字符

$ 匹配字符结尾的字符

\d 匹配单个数字的字符，和 [0-9] 语法一样

\d+ 匹配多个数字字符，和 [0-9]+ 语法一样

\D 单个非数字字符

\D+ 匹配多个非数字字符

\w 单个英文字母或数字的字符，

\W 单个非英文字母或数字的字符

\W+ 匹配多个非英文字母或数字字符

\s 空格 等同于 [\n\t\r\f]

\s+ 等同于 [\n\t\r\f]+

\S 单个非空格字符

\S+ 多个非空格字符

\b 匹配任何单词的首尾。

\B 匹配所有\b 不能匹配的位置。

我们来看几个例子:

/abc/ 匹配所有包括 abc 的字符串,比如，匹配“123abcxxx”、“abde88”。

/ab(c|d)/ 匹配所有包括 abc 或 abd 的字符串,比如，匹配“123abcxxx”、“abcde”。

/abc\d+/ 匹配所有包括 abc 后接至少一个数字的字符串,比如，匹配“xxabc456”、“abc678de”。

/^abc\d+/ 匹配所有以 abc 后接至少一个数字开头的字符串,比如，匹配“abc456”、“abc678de”，不匹配“xxabc456”、 “xxbc456”。

/^abc\d+$/ 匹配所有以 abc 开头，至少一个数字结尾，且 abc 后接至少一个数字的字符串,换句话说，仅匹配符合 abc 后接至少一个数字，不包括任何其它字符。比如，匹配“abc456”、 “abc1”，不匹配“abc456x”、 “bc456”、“abc”。

/[^0-9]abc\d+/ 匹配所有包括非数字字符后接 abc 及至少一个数字的字符串，比如，匹配“xyabc456”、 “aabc1”，不匹配“456abc98”、 “bdc456”、“0abc1”。

/^[^0-9]abc\d+/ 匹配所有以非数字字符后接 abc 及至少一个数字的形式开头的字符串，比如，匹配“aabc456”、“xabc1you”，不匹配“8abc98ww”、“bd456”、“0abc1”。

2. 匹配

2.1 基础语法

1、使用正则模式串实现字符串匹配。
2、语法格式(regexp 为模式串)：
$字符串变量名=~m/<regexp>/
也可以简写为$字符串变量名=~ /<regexp>/
这 2 种语法格式，如果字符串变量含有模式则为真，返回 1，否则返回空值。
通常如下使用:

```
If ($字符串变量名=~m/<regexp>/){
    #匹配成功，执行成功后的语句
}
```

3、示例

```
#perl4-0.pl
my $name=”zhangsang”;
if   ($name=~m/san/){#匹配成功
    print “ok”;
}
```

很显然, zhangsang 是能够匹配的，因此输出为：
ok

2.2 匹配修饰

匹配修饰符在模式定界符之后使用

i ：忽略大小写，如:/abc/i

g ：查找所有匹配值，返回一个匹配到的列表。如：/abc/g

例如：

```
#perl4-1.pl

my $name="hello!Beijing";

#我们首先使用 i，用大写“LLO”串进行匹配，由于忽略大小写，匹配是成功的。

print "使用 i\n";

if  ($name=~m/LLO/i){

    #使用 i，匹配成功

    print "使用 i-ok\n";

}

#使用 g，用“e“进行全局匹配，e 出现了 2 次，因此返回 2 个 e 组成的列表

print "使用 g\n";

my @result=($name=~m/e/g);

print @result;#输出匹配列表数据

print "\n";

print scalar(@result);#返回@result 数组大小(全局匹配成功数目)为 2。scalar(@数组变量名)返回数组大小

print "\n";

if ($name=~m/e/g){

#使用 g，匹配成功 2 处

    print "使用 g-ok\n";

}
```

```
#同时使用 g 和 i 进行匹配
print "使用 g 和 i\n";
if ($name=~m/L/gi){
    #使用了 g 和 i 匹配成功
    print "使用 g 和 i-ok\n";
}
```

输出结果如下：

```
使用 i
使用 i-ok
使用 g
ee
2
使用 g-ok
使用 g 和 i
使用 g 和 i-ok
```

2.3 特殊字符匹配

Perl 正则中，某些字符具有特殊含义，譬如“?”、“*”、“.”、“+”等。如果模式串需要把这些符号当作纯文本来处理的话，就必须在它前面加上反斜杠（\）。

比如:

```
#perl4-2.pl
my $name="你叫什么名字?";
my $filename="plane.jpg";
if   ($name=~m/\?/){
      print "$name 是疑问句\n";#匹配疑问句
}
if   ($filename=~m/\.jpg$/){
```

```
        print "$filename 为 JPG 类型图像文件\n";#文件扩展名为 jpg
    }
```

输出结果如下：

你叫什么名字?是疑问句

plane.jpg 为 JPG 类型图像文件

2.4 第五个任务

公司新开发的小型 ERP 系统试运行时，经常出故障，小黄奉命查出故障来源，系统记录用户操作保存在 run.log 的日志文件中。

run.log 的结构大致如下，每列信息以#相隔。

2010-07-01#09:12:10#zhangsang login success

2010-07-01#09:15:30#zhangsang save success

2010-07-01#09:16:50#lisi login error#系统错误#用户不存在

2010-07-01#09:18:05#zhangsang exit success

2010-07-01#13:19:10#wangwu login error#系统错误#密码错误

2010-07-02#09:20:10#lisi login success

2010-07-02#09:31:32#lisi save error#SQL 执行失败#insert error

2010-07-02#09:28:32#lisi exit success

由于日志文件过于巨大，小黄无法人工查找错误，只能用 perl 编写以下脚本筛选错误，并将错误输出到单独的文本文件 runerr.txt 中。

```
#perl4-3.pl
use strict;
use warnings;

open LOG,"<run.log";
open ERRLOG,">runerr.txt";
foreach my $line(<LOG>){
    if ($line=~m/error/i) {#只匹配日志中的错误事件，错误事件都包括“error”字符
        chomp($line);#去除换行符
        (my $mydate,my $mytime,my $mysj,my $myerr,my $errinfo)=split("#",$line);#获取事件信息
        print ERRLOG "--------\n";
        print ERRLOG "日期:$mydate\n 时间:$mytime\n 事件:$mysj\n 错误类型:$myerr\n 错误信息:$errinfo\n";#输出错误信息到 runerr.txt 中
    }
}
close LOG;
close ERRLOG;
```

脚本执行完毕后，打开 runerr.txt，错误信息如下，一目了解：

日期:2010-07-01
时间:09:16:50
事件:lisi login error
错误类型:系统错误
错误信息:用户不存在

日期:2010-07-01
时间:13:19:10
事件:wangwu login error
错误类型:系统错误
错误信息:密码错误

日期:2010-07-02
时间:09:31:32
事件:lisi save error
错误类型:SQL 执行失败
错误信息:insert error

3. 替换

1、语法格式:

$字符串变量名=~s/<pattern>/<replacement>/

<pattern>为需要替换的匹配模式串，<replacement>为替换的字符串。 把符合 <pattern>模式的字符串替换为 <replacement>。

比如：将$myinfo 字符串中的广东替换成北京。

$myinfo=~s/广东/北京/;

2、示例：

1)　不使用匹配修饰符

```
#perl4-4.pl
my $mycity="广东湛江";
$mycity =~s/湛江/深圳/;#将湛江替换成深圳
print $mycity;
```

输出结果如下：

广东深圳

2)　使用匹配修饰符

```
#perl4-5.pl
#将字符串中深圳或湛江替换成广东
my $userlist="深圳刘发#上海李慧#北京张兵#湛江张济#湛江黄明";
$userlist =~s/湛江|深圳/广东/g;#将所有的湛江或深圳替换成广东。
print $userlist;
```

运行结果如下：

广东刘发#上海李慧#北京张兵#广东张济#广东黄明

4. 转化

1、语法格式:

$字符串变量名!~tr/<pattern>/<replacemnt>/

<pattern>为需要替换的匹配模式串，<replacement>为替换的字符串。 把符合 <pattern>模式的字符串转换为 <replacement>。

2、示例：

1) 将小写字母全部转成大写

```
#perl4-6.pl

my $zmb="abcde";

$zmb!~tr/[a-z]/[A-Z]/;#将小写转换成大写

print $zmb;
```

输出结果如下:

ABCDE

2) 简单加密数字。

```
#perl4-7.pl
#简单加密数字，0 加密成 e，1 加密成 f，2 加密成 g...以此类推
my $passnum=<stdin>;
$passnum!~tr/[0-9]/[e-n]/;#将数字转换成字母，实现简单加密
print "\n";
print $passnum;
```

运行后，我们随意输入一串数字:834235902

输出简单加密的信息：mhighjneg