# Perl 学习手札

----------
畅想未来の水星☆
http://hi.baidu.com/hici
----------
by 水星 cici
2007-07-08

## 目录

# 1. 关于Perl

当你翻开这本书的时候，你也就进入了一个奇幻的世界。Perl 确实是一种非常吸引人的程序语言，而之所以这么引人入胜的原因不单单在于他的功能，也在于他写作的方式，或说成为一种程序写作的艺术。即使你只是每天埋首于程序写作的程序设计师，也不再让生活过分单调，至少你可以尝试在程序代码中多一些变化。而且许多 Perl 的程序设计师已经这么作了，这也是 Perl 的理念-「There is more than one way to do it」。

常常遇到有人问我：「Perl 到底可以拿来作什么呢？」，不过后来我慢慢的发现，这个问题的答案却是非常的多样化。因为在不同的领域几乎都有人在使用 Perl，所以他们会给你的答案就会有很大的差异了。有人会觉得 Perl 拿来用在生物信息上真是非常方便，有人也来进行语料的处理，数据库，网页程序设计更是有着广泛的运用。当然，还有许多人把 Perl 拿来当成系统管理的利器，更是处理系统日志的好帮手。

## 1.1 Perl的历史

由 Larry Wall 创造出来的 Perl 在 1987 年时最早出现在 usenet 的新闻群组 comp.source。从当时所释出的 1.0 版本，到 3.0 版为止，几乎维持着一年有一次大版本的更新，也就是说在 1989 年时，Perl 已经有了 3.0 版。而 1991 年，Perl 开发团队发展出相当关键的 4.0 版。因为随着 4.0 版的释出，Perl 发表了新的版权声明，也就是 Perl Artistic Licence(艺术家授权)。Perl4 跟 Perl5 之间相隔了有三年之久，渐渐的，Perl 的架构已经日趋稳定。一直到最近，Perl 释出了新的 Perl 5.8 版，并且同时进行新一代版本的开发。

## 1.2 Perl的概念

Perl是非常容易使用的程序语言，或者我们应该说他是方便的程序语言，你可以随手就写完一支Perl的程序，就像你在命令列中打一个指令一样 (注一)。因为Perl的诞生几乎就在于让使用者能够以更好方便的方式去撰写程序代码，却不必像写C一样的考虑很多细节。
另外，Perl的黏性非常的强 (注二)，你可以用Perl把不同的东西轻易的连接起来。而且你可以用Perl解决你大部份的问题，虽然有些时候你并不想这么作，但这并不表示Perl作不到。

## 1.3 特色

很多人对Perl的印象就是一种写CGI(注三)的程序语言，或者直觉的认为Perl只是拿来处理文字的工具。不过就像我们所说的，Perl几乎可以完成大部份你希望达成的工作。但是不可否认的，正规表示式显然是Perl足以傲人的部份，这也就是Perl大量被拿来使用作为文字处理的原因之一。

而且Perl对于你希望快速的完成某些工作确实可以提供非常大的帮助。甚至在Unix-like的环境下，还可以直接使用Perl为基础的Shell，让你用Perl当指令。而不必像许多程序语言，在还没正式工作之前，你必须先准备一堆事情，包括你的变量定义，你的数据结构等等。也因此，许多Unix-like系统的管理员都喜欢拿Perl来进行系统管理。毕竟没人希望要处理一个邮件纪录文件还要先花一堆时间搞清楚该怎么把纪录文件内的东西转成合适的数据结构。

## 1.4 使用Perl的环境

虽然大多数的Unix-like系统管理员选择Perl来帮助他们管理他们的服务器，可是这绝对不表示Perl只能在这些系统上执行。相反的，Perl可以在绝大多数的操作系统上执行。而目前Windows上的Perl则是以Active Perl(http://www.activeperl.com)这家公司所提供的直译器为主。Perl的使用在不同的操作系统下会略有不同，本书则以unix-like为执行环境。

## 1.5 开始使用 Perl

在开始使用Perl之前，必须先确定你的机器上是否已经安装Perl。在许多unix-like的操作系统中，都预设会安装Perl，你也可以先执行下面的指令来确定目前系统内的Perl版本。

```
[hcchien@Apple]% perl -v

This is perl, v5.8.2 built for darwin-2level

Copyright 1987-2003, Larry Wall

Perl may be copied only under the terms of either
the Artistic License or the
GNU General Public License, which may be found
in the Perl 5 source kit.

Complete documentation for Perl, including FAQ
lists, should be found on
this system using `man perl' or `perldoc perl'.
If you have access to the
Internet, point your browser at
http://www.perl.com/, the Perl Home Page.
```

我们看到在版本的部份，这里使用的是 Perl 5.8.2 的版本，然后有著作者 Larry Wall 的名字，也就是版权拥有着。接下来是 Perl 的版权说明。另外，你应该要注意 `perldoc perl' 这个部份：直接在你的终端机下打这行指令，就可以看到 Perl 内附的文件，而且内容非常详细。

在这里，我们建议使用 Perl 5.8 以上的版本，如果你的版本过于老旧，或是系统中还没有安装 Perl，可以从 http://www.perl.com/下载，并且安装 Perl。
如果你的系统已经有 Perl，并且正常运作，那么你可以开始使用你的 Perl。你可以试着使用所有程序语言都会使用的范例来作为使用 Perl 的开端：

```
[hcchien@Apple]% perl -e 'print "hello
world!\n"'
hello world!
```

不过在 Windows 上，因为命令列不能使用单引号，所以得这样写：

```
[hcchien@Apple]% perl -e "print \"Hello
world\n\""
hello world!
```

## 1.6 你的第一支Perl程序

事实上，你刚刚已经有了你的第一支Perl程序。当然，你可以不承认那是一支Perl程序。不过让我们真正来写一支程序吧。如果你习惯于使用许多整合性程序开发工具，你大概会希望知道要安装什么样的工具来写Perl。不过你可能要失望了，因为我们全部所需要的就只是一个文字编辑器。你在unix上，可以选择vi(vim)，joe或任何你习惯的编辑器，在Windows上可以使用记事本，或下载UltraEdit(http://www.ultraedit.com/)。不过请不要使用类似Word的这样的文书处理工具，因为这样子你只是让事情更复杂了。当然，即使你在Windows上，你还是可以选择Vim或是Emacs这些在Unix世界获得高度评价的文字编辑器，而且他们还是自由软件。
现在，我们可以打下第一支程序了：

```
#!/usr/bin/perl

print "hello world\n";
```

相信大家很快就打完了这支程序。先别管里面到底说了什么（虽然你们应该都看懂了），我们先来执行他吧！

```

```

```
[hcchien@Apple]% perl ch1.pl
hello world
```

好极了，结果就像我们直接用命令列执行的样子。不过至少我们知道了，只要用 Perl 去执行我们写出来的程序就可以了，当然，你还可以有更简单的办法。你可以让你的档案变成可以执行，在 Unix 下，你只需要利用 chmod 来达成这样的目的。当然，我们假设你已经可以操作你的系统，至少能够了解档案权限。修改完权限之后，你只需要在档案的所在目录打：

```
[hcchien@Apple]% ./ch1.pl
hello world
```

那么第一行又是什么意思呢？其实这是 Unix 系统中，表明这支程序该以什么方式执行的表达方式。在这里，我们希望使用"/usr/bin/perl"这个程序来执行。所以请依照你系统内的实际状况适时改写。否则当你在执行的时候，很可能会看到"Command not found"之类的错误讯息。
不过在真正开始写 Perl 之前，我们还要提醒几件事情，这些事情对于你要开始写 Perl 的程序是非常的重要的。
1. Perl 的叙述句是用分号(;)隔开的，因此只要你的叙述句还没出现分号，Perl 就不会把他当成一个完整的结束，除非你的这个叙述句是在一个区块的最后一句。我们可以在 perldoc 里面找到这样的范例；

```
print
"hello world\n"
;
```

而且这样的写法对 Perl 来说并没有什么不同，只是对于需要维护你的程序的人来说显然并不会特别高兴。适时的空白确实可以提高程序的可读性，不过记得不要滥用，造成自己遭受埋怨。
2. Perl 是以井字号(#)作为程序的批注标示，也就是只要以井字号开始，到叙述句结束前的内容都会被当成程序批注，Perl 并不会尝试去执行他，或编译他。对于有些习惯于 C 程序写作的程序员而言，能够使用(/* .... */)来进行程序的批注确实是相当方便的。Perl 并没有正式的定义方式来进行这样整个区块的批注，不过却可以利用其它方式来达到同样的目的。例如使用 pod(plain-old documentation format，简明文件格式)：

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

```
print "hello world\n";

=head1
这里其实是批注，所以也是很方便的
主要是可以一次放很多行批注
=cut
```

如果你还想找出其它可能的替代方案，可以直接看 perlfaq 这份文件，而方法就是直接执行 perldoc perlfaq(注四)就可以了。
接下来，我们便要真正进入 Perl 的环境中了。

习题：
1. 试着找出你计算机上的 Perl 版本为何。
2. 利用 perldoc perl 找出所有的 perl 文件内容
3. 利用 Perl 写出第一个程序，印出你的名字

注一：事实上，Perl 有所谓的单行模式，你就只需要在命令列中执行 Perl 的叙述句。
注二：因此也有人戏称 Perl 是「胶水程序语言」。
注三：就是所谓的「Common Gateway Interface」，动态网站程序的设计界面。
注四：perldoc 里有着许多非常有用的文件，你可以考虑试着看看 perldoc perldoc。

# 2. 纯量变数(Scalar)

在Perl的世界里，变量其实是以非常简单的形式存在。至少比起你必须记忆一大堆int，char等等的数据形态算是方便许多了。对于所有只需要储存单一变量值的数据结构，在Perl里面都是使用纯量数值来进行。所以你在写Perl的时候不需要去考虑你的某个变量是要储存数字或字符串，大多数的时候你也不需要烦恼着要进行数字与字符串间的转换(注一)。

## 2.1 关于纯量

一般的数据型态中，大多就是数值与字符串两种型态。当然，其中的数值也还可以分成整数型态跟浮点数型态，字符串则是由一个或多个字符组合而成。在许多程序语言中，对这方面的定义非常的严格，你不但要考虑程序的流程，还必须随时注意是否该对你的变量进行数据型态的转换。不过在Perl中，对于所有这一类型的变量一律一视同仁，因此不论你所要储存的是数值或字符串，在Perl中都只需要使用纯量变量。

其实就和我们现实生活中相当接近，我们在使用自然语言时，并不会特别去声明接下来要使用什么样的描述或解释，而大多数是取决于当时的语境。而Larry Wall既然是一位自然语言学家，显然对于这一方面特别在行，而这也造就了Perl能够以非常接近口语的方式表达计算机程序语言的重要原因。

### 2.1.1 数值

数值对于Perl的意义在于「整数值」跟「浮点数值」，Perl其实对于数字的看待方式是以倍精度的方式去运算，对于目前大多数的系统而言，这样的精确度显然可以应付大部份的需求。否则许多券商都使用Perl进行系统开发，甚至太空科学或DNA运算也都大量的使用Perl，难道他们会想拿石头砸自己的脚？何况，如果需要的话，Perl也可以支持精确度无限的BigNum（「大数」）运算。

你可以很容易的在Perl中使用数值，不论是整数或浮点数。例如下面都是非常典型的数值表达方式：

```
1
1.2
1.0
1.2e3
-1
-1.2
-1e3
```

其中的 1.2e3 是表示 1.2 乘以 10 的三次方，这也许在科学或数学上的使用比较多，不过你多知道一些也是有帮助的。

另外，也许你希望使用类似 1,300,000 来清楚的显示一个数值的长度，很可惜，

这在 Perl 中会造成误解，当然也不会达到你的要求。如果你真的期待以区隔的方式来表现数字的长度，你可以尝试使用 1_300_300 的形式。不过我个人并不经常使用，也许我处理数字长度都还不足以使用这样的表达方式。你可以看看在 Perl 里面数值的表达方式：

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

$a = 1_300_300;
$b = 1.3e6;

print "$a\n";
print "$b\n";

执行结果会像这样

1300300
1300000
```

你也许会希望使用非十进制的表示方式，例如十六进制的数字就可以在你撰写网络相关程序的时候提供相当的帮助。这时候，Perl 会有一些特殊的方式来帮助你完成这样的工作。例如：

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

$a = 0266;
$b = 0xff;

print "$a, $b\n";
printf "%lo, %lx\n", $a, $b;

我们可以看到结果就会像是：

182, 255
266, ff
```

其中，printf 是以格式化的形式打印，我们稍后会提到。在这里，我们只要知道这样的方式可以印出八进位跟十六进制的数字就可以了。

## 2.1.2 字符串

在纯量的数据形态中，除了数字，另一个重要的部份则是字符串。事实上，Perl的纯量值就不外乎这两种数据型态。在Perl中使用字符串也非常的容易，你只需要在所要表达的字符串前后加上一对的单引号或双引号就可以了。

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

$a = "perl";
$b = 'perl';
$c = "在\tPerl\t 中使用字符串非常容易";
$d = '在\tPerl\t 中使用字符串非常容易';
$e = "23";
$f = "";

print "$a\n";
print "$b\n";
print "$c\n";
print "$d\n";
print "$e\n";
print "$f\n";

于是可以印出结果：

perl
perl
在      Perl     中使用字符串非常容易
在\tPerl\t 中使用字符串非常容易
23
```

在这里，我们看到一些比较有趣的东西。也是在 Perl 里面对字符串处理的一些特性：

1. Perl 虽然可以使用单引号或双引号来表示字符串，但是两者的使用却有些许不同。像我们在$c 跟$d 两个变量中所看到的一样，两个字符串表面上看起来一样，但是却因为使用了单引号跟双引号的差别，使得两个出来的结果就产生了差异。因为在 Perl 的单引号中，是无法使用像\n（换行字符），\t（跳格字符)这些特殊字符的，因此在单引号中的这些字符都会被忠实的呈现。就是我们在变量$d 的结果所看到的。而这些特殊字符还包括了：

```
\a: 会发出哔的警铃声
\d: 代表一个数字的字符
\D: 代表一个非数字的字符
\e: 跳脱符号 (escape)
\f: 换页
\n: 换行
\s: 一个空格符 (包括空行，换页，跳格键也都属于空格符)
\S: 非空格符
\t: 跳格字符 (Tab)
\w: 一个字母，包括了 a-z，A-Z，底线跟数字
\W: 非字母
```

除此之外，我们可以在双引号中内插变量，可是却无法在单引号中使用这样的方式。比如在程序中，我们使用了 print 这个函数，后面接着字符串，字符串里面包含了变量名称跟换行字符。恰巧这两者都是无法以单引号呈现的，因此我们可以看看下面的程序表现出他们的差异：

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

$c = "在 Perl 中使用字符串非常容易";

print '$c\n';
print "$c\n";

我们会得到这样的结果

[hcchien@Apple]% perl ch2.pl
$c\n 在 Perl 中使用字符串非常容易
```

很明显的看到第一行的输出结果就单纯的把单引号的内容表现出来，这样的差异对之后程序的写作其实有着相当的影响。

2. 利用引号将数字括住，虽然可以清楚的表达你希望将这个数值以字符串的方式运算，但是其实这样的方式有点啰唆，而且 Perl 的程序设计师并不会希望自己在写程序时要弄的这么复杂。因此只要在变量的使用时根据语境的不同而有不同的运算方式。也许可以看看以下的例子：

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

$a = "number";
$b = 3674;
```

```
$c = "4358";
$d = $a.$b;
$e = $b+$c;

print "$d\n";
print "$e\n";

可以得到：

number3674
8032
```

所以你可以看到变量$c，我们利用引号把数字括起来，但是当我们把变量$b 跟变量$c 相加时，Perl 就会根据语意，自动把$c 转成数字后再进行加法的运算。也因此，如果你的程序把变量写成像上例的变量$c 一样，虽然语法上不会有错误，不过对于有经验的 Perl 程序设计师而言，反而会显得很不习惯。如果他们看到这样的程序写法，也许还会发出会心的一笑。

3. 我们可以在定义变量时给定初值，就像上述的例子所使用的方式。而在 Perl 中，如果你只是定义了变量，而没有给定初值，那么这个变量会被 Perl 视为 undef，也就是「尚未定义」的意思。

## 2.1.3 数字与字符串转换

毫无疑问，大部份我们在写Perl的时候是不需要特定对数字与字符串进行转换，因为Perl通常会帮我们处理这一类的事情。例如你对两个包含数字的纯量变量进行加法运算以及连接运算则会产生不同的结果，我们可以作个实验：

```
$a = 1357;
$b = 2468;

print $a+$b,"\n";
print $a.$b,"\n";

就可以看到印出：

3825
13572468
```

不过有时候你必须强制要求 Perl 使用某种数据型态，那么就可以使用转换函式，你可以利用 int()函数把变量强制使用整数的型态。不过这样的机会并不多，因此你只需要注意使用这些变量时用的算符，避免让 Perl 产生误会就可以了。

## 2.2 使用你自己的变量

在一般的情况下，Perl并不需要事先定义变量后才能使用，因此就像我们的范例中所看到的，你可以直接指定一个数值到某个变量名称，而Perl大多也会欣然接受这样的方式。可是在大部份的状况，程序设计师所出现错误的机会远大于Perl出现错误的可能。比如你可能会犯下所有程序员都会犯的错误：

```
$foo = 3;
$f00 = 6;
print $foo;
```

程序执行之后会得到'3'这样的结果，这也许是你想要的，也许不是。不过如果你在编译讯息里面要求 Perl 送出编译讯息给你(注二)，你也许会得到这样的讯息：

```
Name "main::f00" used only once: possible typo
at ch2.pl line 4.
```

这并不是错误，你也许因为打错字而造成的结果还不足以影响 Perl 的执行，但是却会影响你希望产生的结果。不过，使用编译的警告参数还有其它用法。你可以在你的程序中加上"use warnings"，所不同的是，你在程序的一开始就使用"-w"这个参数，是要求 Perl 对你程序中每一行都进行检查。可是有时候会因为使用不同的版本，让原来一切正常的程序在换为其它版本时产生出警告讯息。因此有些时候你可以单单对于某个区块进行检查的动作，这样的情况下，你就可以使用"use warnings"这样的方式来要求 Perl 帮忙。相对于此，我们还有其它的方式来跳过某个区块的警告讯息。就像这样：

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

use warnings;

{
no warnings;
$foo = 3;
$f00 = 6;
}

print $foo;
```

我们在程序的一开始就使用了"use warnings"这个选项来对我们的程序进行编译的检查。不过却在某个区块中定义了"no warnings"，让 Perl 跳过这个区块进行

检查。这样一来，我们在执行程序时，Perl 就不会再发出上面的那些警告讯息。当然，除非你真的知道自己在做什么，否则还是尽量不要省略这样的检查才是正途。

另外一个良好的书写习惯，就是在程序的前面加上 use strict 的描述，告诉 Perl 你希望使用比较严谨的方式来对程序进行编译。而一但使用 use strict 来对你自己的 Perl 程序进行严谨的规则时，你就需要用 my 这个关键词来定义你自己所需要的区域变量。因此假设我们刚刚的程序会写成这样：

```
use strict;
my $foo = 3;
$f00 = 6;
print $foo;

那么一但我们想要执行这支程序，Perl 就会发生错误：

Global symbol "$f00" requires explicit package
name at ch2.pl line 5.
Execution of ch2.pl aborted due to compilation
errors.
```

因为我们没有定义$f00 这个变量，Perl 不知道你是忘了，或者只是打错字，这是相当有用的，尤其在你的程序长度已经超过一个屏幕的长度时。因为我们打错字的机会确实还不少，而且这样的程序错误是非常难以除错的。因此能够在程序的一开始就强制使用严谨的定义是比较正确的作法。

## 2.2.1 变数的命名

其实我们已经看了好多例子，里面都包含了Perl的纯量变量，因此相信大家应该都不算陌生了。Perl的变量是以字母，底线，数字为基本元素，你可以用字母或底线作为变量的开始，然后接着其它的字母，底线以及数字。可是在Perl的规定中，是不能以数字开始一个变量名称的。

而在Perl中，纯量变量则是以$符号作为辨识，因此像之前看到的都是属于纯量变量的范围。

当然，怎么帮你的变量名称命名也是必须注意的，因为在Perl中，大小写的字母是会被视为不同的。因此你如果用了$foo跟$fOO，这在Perl中是属于两个不同的变量，只是我们十分不建议这样的命名方式。否则将来可能维护你的程序代码的人也许会默默的咒骂你，那可别怪我们没事先警告了。

另外，你应该让其它看你的程序设计师看到某个变量都大概可以猜出这个变量的作用，你自己想想，如果你看到某支程序里面的变量名称是像这样子：$11，$22，$33，或者像这样：$100000001，$100000002....。你会不会想要杀了这程序的作者呢？

至于大小写也是在定义变量时可以运用的另一项特点，例如有人就习惯利用像这

样的方式来定义变量名称：$ChangeMe。这样可以避免某些因为连字时造成的混淆，不过一般而言，只要能清楚的表现变量的特性，大小写与否则视个人的习惯。不过这一切都是为了将来程序维护上的方便，对Perl来说，这些变量的命名对他并没有特别的意义。不过如果将来维护程序的可能是你自己，还是别找自己的麻烦吧！

## 2.3 赋值

既然变量就像一个容器，是拿来存放变量，能够赋予变量他的内容就是非常重要，而且非常基本的一件事了。在程序中，我们经常会对于变量进行赋值运算，否则我们只需要一个定数就好，何必大费周章的使用变量呢？一般来说，要指定一个值给某一个变量的动作并不复杂，而且大概有这两种主要的方式。

### 2.3.1 直接设定

这是直接用等号(=)来将某个数值或运算结果指定给变量。例如我们也许会这么写：

```
$foo = 3;
$foo = 2 + 1;
$bar = "bar";
$foo = $foo + 2;
$bar = $bar." or foo";

就可以看到印出的结果

8
bar or foo
```

在后面的两个式子中我们看到左，右两边分别出现了两次相同的变量名称。这样的方式是非常常见的，我们在右边先取变量原来的数值，经过运算之后，把得到的结果指定给左边的变量。因为 Perl 的自由度非常高，也许会有人想要把许多赋值的工作一次完成，那么他可能把程序写成这样：

```
use strict;
my $foo = 3;
my $bar;
$foo = 3 + $bar = 2;
print "$foo\n";

那么应该会看到这样的结果：
```

```
Can't modify addition (+) in scalar assignment
at ch2.pl line 6, near "2;"
Execution of ch2.pl aborted due to compilation
errors.
```

没错，我们虽然说过 Perl 的语法其实相当自由，但是这样的写法确实会让 Perl 搞混了。当然，我们可以把程序改写成：

```
use strict;
my $foo = 3;
$foo = 3 + (my $bar = 2);
print "$foo\n";
```

Perl 也会输出结果为 5，可是这样虽然 Perl 可以清楚的了解你要表达的意思，只怕其它看程序的人还是会一头雾水，而且这样并不会让你的程序执行得更快，当然无法靠这种方式让你赢得 Perl Golf(注三)，所以除非你有很好的理由，否则还是少用吧！

## 2.3.2 还可以这样

就像我们刚刚看到的，我们可以在赋值前先在等号右边取出变量的值进行运算，就像这样：$foo=$foo+3。可是其实某些二元算符可以有更方便的赋值方式，我们可以写成这样：

```
use strict;
my $foo = 3;
print "$foo\n";
$foo+=3;  # 其实就是 $foo = $foo + 3
print "$foo\n";
$foo*=3;  # 乘号也可以这样使用
print "$foo\n";
$foo/=3;  # 其实所有的二元运算符都可以这么用
print "$foo\n";

可以发现这样的形式确实非常方便：

3
6
18
6
```

## 2.4 运算

其实变量的运算就跟一般的数值是一样的，我们可以利用大部份我们所熟知的运算符号来对变量进行运算。例如我们当然可以把两个变量相加，然后赋值给另一个变量，就像这样：$third=$first+$second。或者更复杂的运算，这从过去我们学到的数学中都可以看到。当然，在Perl里面也的表达式也符合现实生活中的规则，Perl会先乘号，除号进行运算，然后在把结果作相加或相减（如果你的表达式内有这些算符的话）。所以$foo=3*8+2*4 就应该是 32，而不是 106。
不过Perl里面并没有数学中的中括号或大括号，而所有你希望先行计算的部份，都是由小括号将他括起来，例如你可以改写刚刚的算式成：$foo=3*(8+2)*4，那么结果显然就变成 120 了。而运算符的优先级正是你需要进行运算时非常最要的部份，虽然你已经知道乘号与除号的优先级高于加号跟减号。而在这个时候，你还可能需要知道的某些算符的优先级依照他们的优先性大概有下列几种：

```
++, --
**
*,/,%,x
+,-,.
&
&&
||
+=,-=,*=,/=...
```

当然，Perl 的算符并不只有这些，不过我们后面陆续会提到。如果你现在想要知道更多关于 Perl 算符的说明，可以看一下 perldoc perlop 这份文件。

## 2.5 变量的输出/输入

当我们写了一堆程序之后，我们当然希望程序运算的结果可以被看到，否则即使程序运作的结果让人非常满意，你也无从得知。当然，换个角度想，如果你的程序错的一踏胡涂，也不会有人知道。不过如果如此，那何必还花了大量的时间写这支程序呢？如果你无法从程序得到任何结果。
最简单的输出方式，其实我们已经看了很多了，那就是利用print这个Perl的内建函数。而且用法非常直觉，你只需要把你要的结果透过print送出到标准输出(STDOUT)，当然，通常标准输出指的就是屏幕，除非你自己动了什么手脚。我们可以来看看下面的例子会有什么结果：

```
use strict;
my $foo = 3;
print $foo;
print $foo*3;
```

```
print "打印字符串\n";
print $foo, $foo+3, $foo*3;

很简单的，我们就可以看到：

39 打印字符串
369
```

从范例中我们很清楚的就发现，我们可以单纯的打印一个变量，一个表达式，一个字符串，或者一堆用逗点(,)分隔开来的表达式。因为我们可以在 print 后面连接一个表达式，所以我们当然也可以写成这样：
print "foo = ", $foo;
或者你希望最后的输出结果还可以换行，那么你可以这么写：
print "foo = ", $foo, "\n";
可是如果你有三个变量，那么你写起来也许会像这样：
print "first = ", $first, "second = ", $second, "third = ", $third, "\n";
好吧，虽然吃力，而且可能容易产生错误，不过毕竟你做到了。只是如果现在又多了一倍，那困难度可又增加了不少。

## 2.5.1 变数内插

我想你大概可以慢慢感受到，以Perl的程序设计师的个性，他们绝对不希望这样的事情发生，因为这些程序设计师总是不希望自己的时间浪费在打字这件事情上，因此当然要有方法能够少打一些字，又容易维持程序的正确性。而变量的内插就提供了这样的福音。
我们之前提过对于字符串的表示中，单引号与双引号之间的差异。其实这两者之间还有一个重要的差异，就是双引号中可以进行内插变量，而单引号依然很真实的呈现引号内的字符串内容。我们可以看看其中的差异：

```
my $foo = 3;
print "foo = $foo\n";
print 'foo = $foo\n';

很明显的，输出后就有了极大的不同：

foo = 3
foo = $foo\n
```

在双引号中，不但特殊字符\n 会被转换为换行字符，变量名称 $foo 也会被取代为变数的值后输出。反观利用单引号的时候，不论变量名称或特殊字符都会被完整而原始的表示。不过如果你希望输出这样的字符串呢？
print "$ 表示钱字符号\n"

在双引号中，如果你希望正确的表达某些符号，例如用来提示特殊字符的倒斜线，表示变量的符号时，你必须用以一个倒斜线来让原来符号的特殊意义消失，看看下面的写法：
print "\$ 用来提示纯量变量， \@ 则是数组";
那么你就可以正确的显示你要的结果。除此之外，我们也许还有一些好玩的技巧，记得 Perl 的名言吗？「办法不只一种」。
让我们来看下面的程序：

```
print "\$ 用来提示纯量变量， \@ 则是数组，还有 \"\n";
print qq/\$ 用来提示纯量变量， \@ 则是数组，还有
\"\n/;
print qq|\$ 用来提示纯量变量， \@ 则是数组，还有
\"\n|;

结果看来都是一样的

$ 用来提示纯量变量， @ 则是数组，还有 "
```

这确实非常神奇，首先我们提示一下，为了避免 Perl 误以为你要结束某个字符串，因此如果你要印出双引号时，记得先让 Perl 知道，于是就是利用\"的方式来解除双引号原来的作用。可是一但如此，你的字符串内也许会变得难以判读，尤其常常双引号又是成双的出现时。这时候，你可以利用 qq 来描述字符串，而在范例中，我们用了 qq//，qq||，其实 qq 后面可以接任何成对的符号。如果有兴趣也可以自己动手试试。
接下来，我们可以来谈谈怎么接受使用者的输入，也就是让程序可以根据使用者的需求而有不同的反应。
经常被使用的方式应该是程序在进行时，停下来等使用者输入，当接收到换行字符时，程序就继续往下执行。这时候我们就是大多就是依赖的方式。很显然的就是 STDOUT 的对应，也就是所谓的标准输入，而我们常用的应该大多就是键盘。因此，Perl 在遇到时便会等待输入，我们可以用这个简单的例子来试试：

```
print "please enter your name:";
my $name = ;
print "\n";
print "hello, $name\n";

当我们执行时，就会有这样的结果：

please enter your name:hcchien
```

```
hello, hcchien
```

看起来，王子跟公主似乎过着幸福，快乐的日子。可是唯一小小的缺憾，却是Perl 连我们在结束字符串输入的换行字符也一并当成字符串的一部份了。这样的动作有时候会有很大的影响，因此我们也许要考虑把这样的错误弥补过来。这时候，chomp 函数就派上用场了。这个函数生下来似乎就只为了进行这项工作，至少我们再也不用担心使用者的换行字符该怎么办。他的用法显然也不特别困难，只要把你需要修正的字符串当成传入值就可以了。

chomp($name);

所以我们只要把这一行加到刚刚的程序里面，我想你应该就会发现一些变化。没错，原来我们的输出最后还多了一行空行，那是因为我们输入时打了最后一个换行字符，不过经过 chomp 的修正，那个字符果然就没有了。那么 chomp 有没有什么信息可以让我们参考呢？既然他是一个函数，他就会有一个回传值，而 chomp 的回传值就是被移除的换行字符个数。例如我们如果有一个字符串：

my $name = "hcchien\n";

那么一但我执行了

chomp($name);

理论上就会传回 1 的值。实际上也确实如此，那么我们可以再来试试，如果变量 $name 的值变成

$name = "hcchien\n\n";

然后我们发现回传值还是 1，也就是说，chomp 只对字符串结尾的那个换行字符有效。因此如果我们只执行了一次 chomp，并不是把字符串后面的换行字符全部取消，而只是移除了一个。

## 2.6 Perl预设变数

很多时候，刚学Perl的程序设计师似乎常常会遇到一些问题，也就是不容易看懂其它的Perl程序。这其中的原因当然很多，例如Perl的写作形式非常自由，同样的需求可以利用各种方式达成，有些程序设计师常常会用非常简略的语法而让易读性降低。另外，许多Perl的预设变量对于初学者也是一个问题。你常常会看到一堆符号在程序里飞来飞去，却完全不知道他们在说什么，你当然可以利用Perl的在线文件perlvar去找到你要的答案，不过我们会适时的在不同的章节提到一些Perl常用的预设变量。

## 2.7 defined 与 undef

你也许有过经验，当你在写一支程序的时候，你定义了一个变量，就像我们平常作的：

my $foo;

或者你可能写成这样：

my $foo = "";

于是在你的程序过程中，这样两种方式所定义出来的变量在你的程序并没有产生不同。于是程序平静的结束，你开始想象你多打了好几个字，只为了告诉Perl $foo

这个变量是个空字符串。可是所有的事情一如你所预测的一般，你还是没有想透，到底宣告变量是空字符串到底有没有意义呢？其实大部份的时候，你是不需要在定义变量时宣告为空字符串，因为当这个变量被定义时，Perl会指定他为undef。而当这个变量被作为数值时，他瞬间就被当成零，同样的，在被当成字符串运算时，他则会被作为空字符串。
不过，如果你的程序开启了warnings参数，而打算打印一个undef的变量，可是会遭到警告的，因为Perl显然很难理解你为什么需要打印一个没有被定义的变量。而这很可能是你的程序有某部份发生了问题，所以千万别随便忽略Perl的警告，再仔细检查你的程序吧！
所以你也许要确认你的程序在某些叙述是否正如你所期待的正确的进行了某些运算，这时候有一个函数就可以派上用场了，那就是defined()。你可以用defined来确定某个变量是否是经过定义，很简单的就像这样：
defined($name);
而许多程序的写作中，也经常使用这个函数来进行判断。例如你可以这么写：

```
my $name;
if (defined($name)) {
   print $name;
} else {
   print "it's undefined";
}
```

我们可以很清楚的看到这样的宣告一个变量会被设为 undef。

而且，undef在perl中也是个关键词，你可以直接指定某个变量是undef，就像你在赋值给任何变数一样。所以你可以很简单的写成：
$name=undef;

习题：
1. 使用换行字符，将你的名字以每个字一行的方式印出。
2. 印出'\n', \t'字符串。
3. 让使用者输入姓名，然后印出包含使用者姓名的招呼语(例如：hello xxx)。

注一：有些时候是必须强制进行转换，Perl才知道你真正需要的是什么。
注二：你可以在程序的最前面写成这样「#!/usr/bin/perl -w」，告诉Perl你希望启动编译警告。
注三：一种长期并不定时举行的Perl程序设计游戏，以程序代码最短者获胜，就像高尔夫球，最少杆者获胜，详细可以参考http://perlgolf.sourceforge.net/。

# 3. 串行与数组

在我们已经知道怎么使用纯量变量之后，我们就可以处理非常多的工作。可是有些时候，当我们要使用纯量变量来储存许多性质相近的变量时，却很容易遭遇瓶颈。例如我希望储存某个班级四十位学生的数学期末成绩，这时候如果每个学生的成绩都需要用单独的一个变量来储存的话，那会让数据难以处理，也许你从此再也不想写程序了，而且你的程序大概会长的像这样子：

```
my $first = '40';
my $second = '80';
my $third = '82';
...
...
```

没错，这样的写法虽然可能可以让我们比过去使用纸张的方式正确率高一些，可是却未必会省事。另外，如果我希望从数据库找出今天总共有多少人在我的网络留言板留言，那这时候的留言个数是未知的，要怎么批处理这些数据就很花脑筋了，所以要有适当的数据结构可以作这样的处理。

很显然的，数组的运用非常的广泛，几乎大部分撰写程序的时候都会使用数组来进行数据的存取，在许多程序语言中，数组的结构相当的复杂，这确实是必要的。因为数组的使用要必须足够灵活，才能够发挥它的功能，可是如果太过复杂却也是造成入门者的进入门坎。Perl 对于这方面却有一些不同的做法，它提供的数组结构非常简单，如果你用最入门的方式去看它，很多第一次接触的人甚至也可以轻易上手。可是 Perl 的数组却也可以利用非常强大的方式扩展开来，让许多第一次看到 Perl 数组结构却非常失望的人也能重拾对 Perl 的信心。当然，使用这些技巧来进行 Perl 数组的扩充，不但可以像其它程序语言一般，可以进行多维数组之外，还可以能精准的结合某些数据结构，当然，这部份我们不会在一开始介绍数组时就把大家吓走，不过如果你已经对数组的方式有些熟悉，可以在后面的章节慢慢看出 Perl 在这方面设计的巧妙。

## 3.1 何谓数组

对于我们刚刚提出来的数据结构需求，希望能把相同的东西简单的存取，并且让它们能被归纳在一起。数组正是解决这个问题的方案，也就是把一堆性质接近的变量放在同一个数据结构里，这样可以很方便的处理跟存取。就像一迭盘子一样，他们都是性质相接近的东西，于是我们就把盘子碟子一迭，而属于不同性质的东西就分放在其它地方，比如我们就不太应该把碗跟盘子放在同一迭里面。在Perl里面，你可以定义一个数组，而数组里面存放的就是纯量，当然存放的个数可以由零个到许多个，至于实际可以储存的个数则依据每部计算机不同而有所差别，因为Perl依然依循它自己的个性，并不对程序设计师进行太多的限制，因此它可以允许你使用系统上所有的资源，换句话说，你可能会因为一个数组过大而占用系统的所有资源。

## 3.2 Perl 的数组结构

我们先来看看怎么在Perl里面定义一个数组。在Perl中，数组变量是以@符号开头，例如你可以定义一个变量名称叫做@array。然后利用$array[0]，$array[1]...的方式来存取数组里的元素。也就是说，你在定义了数组@array之后，你可以指定数组里面的值，就像这样的方式：

```
my @array;
$array[0] = 'first';
$array[1] = 'second';
$array[2] = 'third';
....
```

这样比起刚刚我们一个一个变量慢慢的指定虽然方便了不少，至少我们可以很清楚的了解这些数值都是属于同一个群组的，因为它们被放在同一个数组中(注一)。不过这样的写法实在太辛苦了，尤其当你已经知道你数组中的元素个数，以及他们个别的值，你就可以用简单一点的方式来把数组的值指定给你的数组，就像这样：

my ($array[0], $array[1], $array[2]) = qw/first second third..../;

其中，qw/first second third.../这一串东西就被称为串行，例如：
my ($one, $two, $three) = (1, 2, 3);
也就是把一个串行一次指定给三个变数。利用 qw 也是同样的方式，因此刚刚那一行程序其实也可以写成：
my ($first, $second, $third) = qw/first second third/;
这样的方式，就是我们把串行的值指定给变量，所以当然这些变量也可以是数组的元素。不过既然我们确定要把串行的值指定给某个数组，我们显然可以更简单的这么作：
my @array = qw/first second third/;
这样的方式就是直接利用串行赋值给数组的方式，而类似的方式还可以写成这样：

```
my @array = (1...10);
my @array = (0, 1, 2, 4...8, 10);
my @array2 = (3, -1, @array, 13);
my @array2 = qw/3, -1, @array, 13/;   # 这应该不是你想要的东西
```

当然，如果你定义了一个数组，但是却没有赋值给他，那么这个数组就会是一个

空数组。相同的状况，你也可以指定任意的数组大小给 Perl，当然前提是你的计算机有足够的能力承受。这当然也是 Perl 的传统之一。
Perl 从来就不是一个严谨的程序语言，因此对于数组的部份也采取同样的规定。你不需要在程序的一开始就规定你的数组长度，因此你可以在程序里面随时新增元素到你的数组中。例如你的程序也许会写的像这样子：

```
my @array = qw/第零 第一 第二/;
$array[3] = '第三';
$array[4] = '第四';
```

没错，你可以使用串行形式来指定数组的值，也可以直接把值指定给数组的某个索引值，就像我们刚刚所使用的方式。另外，你也会发现，如果你这么写的话，Perl 也不会阻止你：

```
$array[15] = '一下子就到 15 了';
```

那么 Perl 会直接帮你的数组程度扩充到 15，也就是数组的索引值会变成从 0-14，而数组大小变为 16。至于数组中间没有被指定的值，Perl 都会自动帮你设为 undef，所以你的数组中，有许多还没定义的值。好吧，很多人也许对于这样的设计不以为然，不过有时候这样还是很方便的，不是吗？想象你已经预测你的数组会有 20 个元素，可是你现在只知道最后一个元素的值，你总不希望必须先把前面十九个元素值填满之后才能开始使用你期待已久的那个元素值吧？
当然，对于那些认为应该严谨的定义程序语言语法，不能让程序设计师为所欲为的人来说，Perl 显然不是他们会选择的工具。而且这样的战争已经持续了很长的一段时间，也不是我们可以在这里解决的。让我们暂且跳开风格争议，继续回来看 Perl 在数组中的用法吧。
有时候我们需要知道数组中的元素个数，比如我们希望在数组中依序取出数组中的元素并且进行运算，那么我们就可以利用下面的方式来进行：

```
my @array = qw{first second third};
# 记得利用 qq 赋值给字符串的作法吗？用 qw 赋值给数组也是类似
$array[4] = 'fifth';                    # 我们跳过索引值 3
print $#array;                          # 这里取得的是最后一个
索引值
print $array[3];              # 这里应该不会有任何结果
```

既然$#array 是数组中最后一个索引值，所以我们可以利用($#array + 1)得到目

前数组中的元素个数(注二)。不过如果你打算利用这个索引值来确定目前数组的长度，并且加入新的元素，就像这样：

```
my @array = qw/first second third/;
$array[$#array+1] = 'forth';        # 把新的值放到现在最大索引值的下一个
```

当然，如果你这样写也是可以被接受的：

```
my @array = qw/first second third/;    # 一开始，你还是有三个元素值
$array[$#array+1] = 'forth';         # 这时候的 $#array 其实是 2
$array[$#array+1] = 'fifth';         # 可是这时候 $#array 已经变成 3 了

print @array;
```

## 3.3 push/pop

没错，我是说那样的写法可以被接受，可是好像非常辛苦，尤其当你已经被一大堆程序搞到焦头烂耳，却还要随时注意现在的数组到底发展到多大，接下来你应该把最新的值放到那里，这样显然非常辛苦。你一定也猜到了，Perl不会让这种事情发生的。所以Perl提供了push这个指令把你想要新增的值「推」入数组中，同样的，你也可以利用pop从数组中取出最后一个元素。不过为什么要使用push/pop这样的指令，这当然和整个数组的数据结构是具有相关性的，如果你弄清楚了数组的形式也许就很容易理解了。我们可以把数组的储存看成是一迭盘子，因此如果你要放新的盘子，或者是拿盘子，都必须从最上面动作。这也就是为什么我们可以利用push/pop来对数组新增，或是取出元素的最重要原因。我们可以从下面的例子看到 push跟pop的运作：

```
my @array = qw{first second third};
push @array, 'fourth';
print $#array;                 # 这里印出来的是 3，表示 'fortuh'已经被放入数组
pop @array;
```

```
print $#array;                  # 至于 pop，则是把元素从数
组中取出
```

而且利用 pop 取出元素一律是从数组的最后一个元素取出，也就是「后进先出(last in, first out)」的原则。当然，pop 的回传值也就是被取出的数组元素，以上面的例子来看，取出的就是'fourth'这个元素。
另外，在使用 push 时，也不限定只能放入一个元素，你可以放入一整个数组。那么就像这样的写法：

```
my @array = qw{first second third};
my @array2 = qw/fourth fifth/;
push @array, @array2;
print @array;                   # 现在你有五个元素了
```

## 3.4 shift/unshift

没错，push/pop确实非常方便，他让我们完全不需要考虑目前数组的大小，只需要把东西堆到数组的最后面，或者把数组里的最后一个元素拿掉。不过我们也发现了，这样的操作只能针对数组的最后一个元素，实在有点小小的遗憾。其实我们想想，如果我把数组中非结尾的某个元素去掉，那会发生什么事呢？比如我现在有一个数组，他目前总共有三个元素，因此索引值就是 0..2。如果我想要把索引值为 1 的那个元素取消，那么索引值是不是也就需要作大幅更动。尤其当数组的元素相当多的时候，其实也会有一些困扰。
不过Perl还是允许我们从「头」对数组进行运算，也就是利用shift/unshift的指令。如果我们已经知道push/pop的运作，那么我们可以从范例中轻松的了解shift/unshift对数组的影响：

```
my @array = (1...10);
shift @array;                   # 我把 1 拿掉了
unshift @array, 0;     # 现在补上 0
print @array;                   # 现在数组的值变成了(0,
2...10)
```

现在你的数组进行了大幅度的改变，我们应该来检查一下，当我们在进行 shift 运算过程中，数组元素的变化。
我们还是用刚刚的数组来看看完整的数组内容：

```
```

```
my @array = (1...10);                    # 我们还是使用这个
数组
shift @array;                            # 我把 1 拿掉了
print "$_\t$array[$_]" for (0...9);  # 现在数组的值变成
了(0, 2...10)
```

好极了，我们看到了输出的结果：

```
0       2
1       3
2       4
3       5
4       6
5       7
6       8
7       9
8       10
Use of uninitialized value in concatenation (.) or
string at ch3.pl line 7.
9
```

没错，我们看到了错误讯息。因为我们的数组个数少了一个，因此索引值 9 目前并不存在，Perl 也警告了我们。所以我们发现了，Perl 在进行 shift 的时候，会把索引也重新排列过。不过你能不能从中间插入一个值，并且改变数组的索引排列，或是拦腰砍断，取走某些元素，然后希望 Perl 完全不介意这件事呢？目前看来似乎没有办法可以这么作的。不过有些方式可以让你单读取出数组中某些连续性的元素，也就是使用切片的方式。

## 3.5 切片

就如我们之前提到，我们总是把一堆串行放入数组中，虽然放入的方式不尽相同，但是至少我们可以在数组中找出 0 个以上的元素所组成的数组。没错，如果我们知道一个数组中的元素，而且我希望取出这个数组中的某些连续性元素是不是可行呢？例如有一个数组的元素是(2003...2008)，那么如果我希望取得的是这个数组中 2004-2006 这三个元素，并且把这三个元素拿来进行其它运算或运用，我是不是应该这样写：

```
my @year = (2003...2008);
my ($range[0], $range[1], $range[2]) = ($year[1],
$year[2], $year[3]);
```

其实如果你真的这么写了，也不会有人说你的程序有错误，虽然这样的写法总是很容易让人产生错误。即使不是语法上的错误，也容易因为打字的原因而产生可能的逻辑错误。既然如此，我们显然应该找出容易的方法来作这件事。我们用一个很容易看清楚的例子来说明吧：

```
my @array = (0...10);
my @array2 = @array[2...4];
print @array2;                          # 没错，你拿到了
(2, 3, 4) 三个元素
```

这个方法，我们就称为切片，就像我们把生鱼片取出其中的一片。可是如果我要的范围并不属于连续性的话，还能切片吗？其实就像你一个一个取出数组中的元素，只是有些部份是连续的，你不希望把每个元素都打一次。所以如果你希望多切几片，可以考虑这么作：

```
my @array = (0...10);
my @array2 = @array[2...4, 6];
```

这时候，你拿到的不但是(2, 3, 4)三个元素，也包含了 6 这一个元素。这样是不是非常方便呢？

## 3.6 数组还是纯量？

如果你已经开始自己试着写一些Perl程序，不知道你有没有遇到这个问题，你有一个数组@array，你想新增一个数组，元素跟原来的数组@array相同，于是你想写了这样一个式子：
my @array2 = @array;
没想到一时手误，把这个式子打成这样：
my $array2 = @array;
这时候，Perl却没有传回错误给你，可是程序会传回什么结果呢？我们可以来实验看看，只要打这几行：

```
my @array = (0...10);
my $array2 = @array;
print $array2;                          # 程序传回 11
```

这个值恰好就是数组@array 的元素个数，所以我们似乎发现好方法来找到数组的元数个数了。不过也许应该来研究一下，为什么 Perl 对于数据型态能够进行

这样的处理。这其实是非常重要的一个部份，也就是语境的转换。这很像我们在之前曾经遇过的例子，当我有两个变数，分别是：

my $a = 4;
my $b = 6;

可是当我使用 $a.$b 跟 $a+$b 两个不同的运算子时，Perl 也会自动去决定这时候该把两个变量使用字符串，或是变量进行处理。因为语境的不同，让运算的方式也有所不同，这在 Perl 当中是非常重要的观念。不过这个观念绝非由 Perl 所独创，相反的，这样的用法在现实生活中是屡见不鲜。比如有人问你平常用什么写程序，你也会依照当时聊天的情况回答你是用什么编辑器，或者是用什么程序语言。因此在语言的使用中，如何选对适当的语境确实相当重要，而既然 Larry Wall 就是研究语言的专家，把这种方法运用在 Perl 里面也是再自然不过了。
我们再来看看刚刚的例子，我们指定一个数组，并且指定这个数组的元素包括一个从 0 到 10 的串行，而当我们把这个数组赋值给一个纯量变量时，Perl 便会把串行元素个数指定为这个纯量变数的值。这也就表示 Perl 正以纯量变量的语境在处理你的运算，而对一个数组以纯量变量的语境进行运算时，Perl 就如我们所看到的，以数组中串行元素的个数表示。所以你可以写出这样的表达式：

```
my @array = (1...10);                    # 利用串行赋值给数
组
my $scalar = @array + 4;       # 在纯量语境中进行
my @scalar_array = @array + 4;
# 先以纯量语境进行运算，然后以串行方式赋值给数组
```

这样看起来会不会有一点眼花缭乱？程序第一行的中，就像我们所熟知的状况，我们把一个串行赋值给数组。接下来，我们利用纯量语境把数组内串行元素的个数取出，并进行运算，然后把结果放到一个纯量变量里，这里全部都是以纯量变量的方式在进行。第三行就比较复杂一点了，我们先用纯量语境，取出数组的串行元素个数，以纯量方式进行运算，接下来把这个得到的结果以串行的方式指定给数组@scalar_array。所以最后一行其实也像是这样：

```
my @array = (1...10);
my $scale = @array + 4;                  # 这里是纯量语境
my @scalar_array = ($scale);             # 把得到的结果放进
串行中，并且赋值给数组 @scalar_array
```

其实就像这里所看到的，如果你的需求是一个串行，而你却只能得到一个纯量，那么 Perl 就会给你一个只有一个元素的串行。其实要诀就是仔细看看你希望得

到什么样的东西，而 Perl 可以给你什么东西。而有时候，当理想与现实有些落差的时候，也许就会有些 undef 产生。假如我们把刚刚的例子改成这样：

```
my @array = (0...10);
my ($scalar1, $scalar2) = (@array + 4);
```

当我们要求的串行无法获得满足时，Perl 就会帮忙补上 undef。

## 3.7 一些常用的数组运算

既然我们总是喜欢把性质类似的变量放在一起变成数组，那么很多时候我们就会希望对这一整个数组进行某些运算。例如排序，过滤，一起带入某个公式中进行运算等等。这时候我们经常利用循环来帮我们处理这一类的事情，不过有些常用的运算，Perl已经帮我们设想好了，我们只需要轻松的一个式子就可以进行一些繁复的工作。

### 3.7.1 sort

排序总是非常必要的，我们在举数组的时候有提到，如果我们要把某个班级学生的数学成绩放入数组，那么我们也许会希望利用这些成绩来排序。这时候，sort 就非常有用了。我们可以这样作：

```
my @array = qw/45 33 75 21 38 69 46/;
@array = sort { $a <=> $b } @array;
这样 Perl 就会帮我们把数组重新排列成为
21      33      38      45      46      69      75

其实，如果你这样写也是有相同的效果：

@array = sort @array;

当然，如果你需要比较复杂的排序方式，就要把包含排序的区块加入，
所以你也可以写成：

@array = sort { $b <=> $a } @array;
```

其中$a 跟$b 是 Perl 的预设变量，在排序时被拿来作为两两取出的两个数字。而<=>则是表示数字的比较，如果数组中的元素是字符串，则必须以 cmp 来进行排序。
我们可以用接下来的例子来说明怎么样进行更复杂的排序工作。

```
my @array = qw/-4 45 -33 8 75 21 -15 38 -69 46/;
@array = sort { ($a**2) <=> ($b**2) } @array;
 # 这次我们以平方进行排序
```

所以得到的结果会是：

```
-4      8      -15     21      -33     38      45      46
-69      75
```

## 3.7.2 join

有时候，你也许会希望把串行里面的元素值用某种方式连接成一个字符串。比如也许你想要把串行中的元素全部以','来隔开，然后连接成一个字符串，那么join就可以帮上忙了。你可以在串行中这么用：

```
print join ',',  qw/-4 45 -33 8 75 21 -15 38 -69 46/;
```

这一行显然也可以写成：

```
my @array = qw/-4 45 -33 8 75 21 -15 38 -69 46/;
print join ',', @array;
```

和 join 函数相对应的的则是 split，他可以帮忙你把一个字符串进行分隔，并且放进数组中。

## 3.7.3 map

很多人会使用Excel的公式，而公式的作用就是针对某一行/列进行统一的运算。比如小时候在学校考试的时候，老师常常会因为全班成绩普遍太差，而进行所谓「开平方乘以十」的计算。这时候，如果可以用map就显得很方便了。

```
my @array = map { sqrt($_)*10 } qw/45 33 8 75 21 15 38
69 46/;
```

我们可以看到，串行里面是学生的成绩，所谓 map 就是把阵串行里的元素一一提出，并进行运算，然后得到另外一个串行，我们就把所得到的串行放到数组中。于是就可以得到这样的一个数组：

```
67.0820393249937
57.4456264653803
28.2842712474619
86.6025403784439
45.8257569495584
38.7298334620742
61.6441400296898
83.0662386291807
67.8232998312527
```

当然，map 还有许多有趣的使用范例，而且如果能适时运用，确实能大幅降低你写程序的时间，也可以让你的程序更加干净利落。

## 3.7.4 grep

我们既然可以针对串行中的每一个元素进行运算，并且传回另一个串行，那么是否可以针对串行进行筛选呢？例如我希望选出串行中大于零的元素，或者以字母开始的字符串元素，那么我可以怎么作呢？这时候，grep就会是我们的好帮手。如果各位是Unix系统的使用者，应该大多用过系统的grep指令，而Perl的grep函数虽然不尽相同，不过精神却是相近的。我们可以利用grep把串行中符合我们需求的元素保留下来。就像这样：

```
my @array = qw/6 -4 8 12 -22 19 -8 42/;      # 指定一个串行给数组 @array
my @positive = grep {$_ > 0} @array;         # 把 @array 里大于零的数字取出
print "$_\n" for @positive;                  # 印出新的数组 @positive
```

而且答案就正如我们所想象的，Perl 能够正确的找出这个数组中大于零的数字。也许你会有一些不错的想法，如果我们想要把刚刚的数组中所找出大于零的数字取得平方值之后印出，那么我们应该怎么做比较容易呢？当然，一般的情况下，我们就会想到循环，而这也正是我们接下来要说的部份。

习题：
1. 试着把串行 (24, 33, 65, 42, 58, 24, 87) 放入数组中，并让使用者输入索引值 (0…6)，然后印出数组中相对应的值。
2. 把刚刚的数组进行排序，并且印出排序后的结果。
3. 取出数组中大于 40 的所有值。
4. 将所有数组中的值除以 10 后印出。

注一：当然，你也可以把程序中的所有纯量变量全部放在一个数组中，不过很快的，你会发现连你自己都不想再看到这支程序了。
注二：别忘了，Perl 的索引值是由零开始。

# 4. 基本的控制结构

## 4.1 概念

大部份的时候，程序总不会跟着你写程序的顺序，一行一行乖乖的往下走。尤其是当你的程序由平铺直叙渐渐变成有些起伏，这时候，怎么确定你的程序到底应该往那里走，或者他们现在到底到了那里。如果你无法掌握程序的流程，只怕他们很快就会离你而去。你从此再也无法想象你的程序会怎么运作，当然也很有可能你就写出了会产生无穷循环的程序了。

### 4.1.1 关于程序的流程

在程序的进行当中，你经常会因为过程发生了不同的事件，因为结果的不同而必须进行不同的运算，这个时候，你就必须进行程序中的流程控制。或者，你会需要对某些工作进行重复性的运算，这时候，重复性的流程控制就可以大大的帮助你减轻工作负担。因此，你常常会发现，流程的控制在你的程序之中确实是非常重要而且经常被使用的。虽然Perl的流程控制跟其它程序语言并没有太大的差异，不过我们还是假设大家并没有这方面的基础。所以还是从头来看看最基本的流程控制应该怎么作呢！

### 4.1.2 真，伪的判断

流程的基本控制主要在于判断某个叙述句是否成立，并藉以判断在不同情况下该怎么进行程序的流程。比如我们可以用简单的例子来认识一下流程控制的进行：

```
my $num = <STDIN>;
chomp($num);

if ($num<5) {
    print "small";
} else {
    print "big";
```

这样看起来是不是非常简单呢？
不过 Perl 比较特殊的部份在于他并不存在一种独立的布尔数据型态，而是有他独特对于真，伪值的判定方式。所以我们应该先要知道，在 Perl 中，那些值属于真，那些值属于伪，这样一来，我们才能知道判断句是否成立。

```
* 0 属于伪值
* 空字符串属于伪值
* 如果一个字符串的内容是"0"，也会被视为伪值。
* 一个 undef 的值也属于伪值。
```

当然，有些表达式也是透过这些方式来判断，我们可以轻易的找到例子来观察 Perl 的处理方式。例如你可以看看 Perl 对这样的判断式怎么处理：

```
my $true = (1 < 2);
print $true;
```

没错，回传值是 1，表示这是个真值，因此如果你在流程控制中用这样的判断式，很清楚可以知道流程的方向。

### 4.1.3 区块

在开始进入正式的判断式之前，我们应该先来说说Perl程序中的区块。在Perl中，你可以用一对大括号{}来区分出一个Perl区块，这样的方式在程序的流程控制中其实非常常见。
在Perl的语法中，区块中的最后一个叙述不必然要加上分号的，比如你可以这么写：

```
my $num = 3;
{
    my $max = $num;
    print $num
}
```

不过如果你未必觉得自己足够细心的话，也许你该考虑留下这个分号，因为一但你的程序略有修改，你也许会忘了加上该有的分号。那你花在加这些分号的时间可能会让你觉得应该随时记得替你的叙述句加上分号才是。另外，区块本身是不需要以分号作为结束的，不过你可别把区块所使用的大括号跟杂凑所使用的混在一起。

### 4.1.4 变数的生命周期

既然提到区块，我们似乎应该在这里稍微提起Perl里面关于变量的生命周期。一般来说，Perl的生命周期都是以区块来作为区别的。这和有些程序语言的定义方式似乎有些差距，当然，Perl的区分方式应该是属于比较简单的一种，所以一般而言，你只需要找到相对应的位置，就很容易可以知道某个变量现在是否还存在他的生命周期中。我们可以看个范例：

```
my $num = 3;
```

```
{
    my $max = $num;
    print $max;
}
print $max;
```

在程序里，我们在区块中宣告了变量$max 的存在，并且把变量$num 的值给了他。就在这一切的运算结束之后，我们进行了两次的打印动作。而两次打印分别在大括号的结束符号前后，表示一个打印是在区块中进行，另一个则是在区块结束后才打印。不过当我们试着执行这支程序时，发生了一个错误：

```
Global symbol "$max" requires explicit package
name at ch2.pl line 12.
Execution of ch2.pl aborted due to compilation
errors.
```

没错，我们在区块内定义了变量$max，也因此，变量$max 的生命周期也就仅止于区块内，一但区块结束之后，变量$max 也就随之消失了。另外，我们也可以看看这个类似的例子：

```
my $num = 3;
{
    my $max = $num;
    print "$max\n";
}
{
    my $max = $num*3;
    print "$max\n";
}
```

这个例子中，我们看到了变量$max 被定义了两次，可是这两次却因为分属于不同的区块，因此 Perl 会把他们视为是完全独立的个体。也不会警告我们有个叫做$max 的变量被重复定义了。这看起来非常简单吧？！这让你可以在你需要的区块里，定义属于那个区块自己的同名变量，可是有时候其实你会把自己搞的头晕，不信的话，你可以看看接下来的写法：

```
my $a = 3;
my $b = 9;
```

```
{
    print "$a\n";  # 属于外层的区块，所以你会看到 3
    my $b = 6;     # 定义了这区块内自己的变量
    print "$b\n";  # 于是你看到的这个$b 的值其实是 6
}
{
    print "$a\n";  # 这个区块没有自己的$a
    print "$b\n";  # 也没有自己的$b
                   # 所以你在这里看到的值其实是上一层
的变量值
}
print "$a\n";      # 这里似乎毫无疑问
print "$b\n";     # Perl 还是印出期待中的 3 跟 9
```

## 4.2 简单判断

好极了，现在我们已经知道什么是真值，什么是伪值。这样就可以运用在程序的流程判断了。

### 4.2.1 if

if的判断非常的直觉，也就是说，只要判断式传回真值，程序就会执行条件状况下的内容。这是一个非常简单的例子：

```
my $num = 3;
if ($num < 5) {
    print "这是真的";
}
```

没错，这个程序虽然简单，但却很清楚的表达出 if 判断式的精神。在($num < 5)里，Perl 传回一个真值，于是我们就可以执行接下来的区块，也就是打印出字符串"这是真的"。

```
提示:
由于这些判断式会用到大量的二元运算符，为了避免执行上产生难以除
错的问题，我们在这里提醒各位一些容
易忽略的部份。
"<", ">", ">=", "<=", "==", "!=": 这些算符都是在针对数字时
用到的比较算符。
```

"eq", "lt", "gt", "le", "ge", "ne"：如果你是对字符串进行比对，请记得使用这些比较算符。

### 4.2.2 unless

和if相对应的，就是unless了。其实在其它程序语言，很少使用unless的方式来进行判断。因为我们可以使用if的否定来进行同样的工作例如你可以用
if (!($a < 3))
这样的方式来描述一个否定的判断句。可是利用否定的运算符"!"来进行判断显然不够直觉，也因此比较容易出错。这个时候unless就显得方便多了。从口语来看，if叙述就是我们所说的「假如...就...」，而unless就变成了「除非...就...」。这样在表达式看起来，就显得清楚，也清爽多了。所以你可以写成：
unless ($a <3)
这样的写法跟上面的那个例子是一样的效果，不过在易读性上明显好了许多。尤其当你的判断式稍微复杂一些，你就更可以感受到unless的好用之处了。

### 4.2.3 一行的判断

在许多时候，我们会用非常简单的判断来决定程序的走向。这时候，我们便希望能以最简单的方式来处理这个叙述句。尤其当我们进行了判断之后，只需要根据判断的结果来执行一行叙述时，使用区块的方式就显得有点冗长了，比方你有一个像这样的需求：

```
if ($num < 5) {
   $num++;
}
```

这样的写法确实非常工整，可是对于惜字如金的 Perl 程序员来说，这样的写法似乎非常不经济。于是一种简单的模式被大量使用：

```
$num++ if ($num < 5);
```

你没看错，确实就是如此，把判断句跟后续的运算句合并为一个表达式。而且这种用法不仅止于 if/unless 判断式，而是被大量使用在许多 Perl 的表达式中。我们以后还会有很多机会遇到。不过先让我们继续往下看。

### 4.3.4 else/elsif

你总是有很多机会使用到if/unless判断式，而且常常必须搭配着其它的判断才能完整的让你的程序知道他该做什么事。这个时候，比如你也许会想这么写：

```
if ($num == 1) { ... }
if ($num != 1) { ... }
```

这样的程序虽然也对，不过总觉得那里不太对劲，毕竟这两个判断式显然正在对同一件事进行判断，不过却必须分好几个叙述句。当然，如果你的判断还是简单（像我们的例子所写的）也就还能手工进行。可是如果你的判断式长的像这样呢？

```
if ($num == 1) { ... }
if ($num == 2) { ... }
if ($num == 3) { ... }
......
if (($num != 1) && ($num != 2) && ($num != 3) && ...)
{ ... }
```

没错，你现在可以想象人工进行这件事情的复杂度了吧！所以如果有简单的方式来进行，同时还能增加程序的易读性，似乎是非常好的主意，而 else/elsif 就是这个问题的解答。
如果我们在一个，或一大堆 if 判断式的最后希望能有一个总结，表示除了这些条件之外，其它所有状况下，我们都要用某个方式来处理，那么 else 就是非常好的助手。最简单的形式大概就会像这样：

```
if ($num == 1) {
   ...
} else {  # 其实这里就是 if ($num != 1) 的意思了
   ...
}
```

不过如果我们有超过一个判断式的时候，就像之前的例子，我希望$num 在 1..3 的时候，能有不同的处理方式，甚至我如果进行一个礼拜七天的工作，我希望每天都能有不同的状况，那只有 else 显然不够。我总是不希望每次都来个 if，到然后还要判断使用者打错的情况。这时候，elsif 就派上用场了，你每次在其它条件下，如果还要订下其它的条件，那么你就可以写成像这样：

```
if ($date eq '星期一') {
   ....
} elsif ($date eq '星期二') {
   ....
} elsif ($date eq '星期三') {
   ....
   ....
} else {
   print "你怎么会有$date\n";
}
```

其实，利用 if/else/elsif 已经可以处理相当多的问题，可是在许多程序语言中还可以利用 switch/case 来进行类似的工作。在 Perl 中，也有类似的方式，这是由 Damian Conway 写的一个模块，目前已经放进 Perl 的预设套件里了。不过我们并不打算在这里增加所有人的负担。

## 4.3 重复执行

我们刚刚所提到的只是对于某个条件进行判断，并藉由判断的结果来决定程序的流程，因此条件的不同会让程序往不同的地方继续前进。不过很多时候我们需要在某些条件成立的时候进行某些重复的运算，比如我们希望算出 10!，也就是 10 的阶乘。这就表示只要我们指定的数字不超过 10，就让这个运算持续进行，这时候，我们显然需要进行重复的运算。

### 4.3.1 while

while就是一个很好的例子，让我们来看看怎么利用while来完成阶乘的例子：

```
my $num = 1;
my $result = 1;       # 小心，这里一定要指定$result 为 1
while ($num <= 10) {   # 确定你是否超过范围
     $result*=$num;
     $num = $num + 1;
}
```

看起来不难吧！你只要掌握几个原则，理论上就可以很容易让 while 循环轻松上手。

首先，你总得让你的循环有正常运作的机会。当然你如果不希望这个循环有任何机会执行，Perl 也不会在你的耳边大叫，不过维护你的程序的人大概会很难理解这一段不可能执行到的程序代码有什么功用吧。

其次，别忘了让你的程序有机会离开他的循环，除非你知道自己在作什么否则你的程序会不断的持续进行，当然，那就是所谓的无限循环。例如你写了一个非常简单的程序：

```
while (1) {              # 在这里，程序会得到永远的真值
    print "这是无限循环";
}
```

第三，在这里，你还是可以让只有单一叙述的 while 循环利用倒装句达成：
我们假设你完全知道刚刚的程序会发生什么事，而那正是你所希望达成的，那么我们就可以来改写一下，让他变得更简洁：

```
print "这是无限循环" while (1);
```

别怪我们太啰唆，不过这样的写法确实让程序干净许多，而且许多 Perl 程序设计师（也包括我自己在内）非常喜欢这样的用法，如果你有机会读到别人的程序，还是先在这里熟悉一下吧。

## 4.3.2 until

类似if/unless的相对性，你也可以用until来取代while的反面意义，例如你可以用until来作刚刚阶乘的同样程序。语法其实跟while一样：

```
until (判断式) {
    ....
}
```

虽然语法看起来完全一样，不过如果是刚刚的阶乘，判断式就会变成这样：

```
my $num = 1;
my $result = 1;
until ($num > 10) {
     $result*=$num;
     $num = $num + 1;
}
```

## 4.4 for

for循环也是非常有用的循环，尤其在你使用数组时，你可以很方便的取出所有数组中的元素。而且你几乎不需要知道现在数组中有多少元素，听起来非常神奇不是吗？不过先让我们来看看for到底是怎么用的呢？

### 4.4.1 像 C 的写法

如果你写过C语言，你应该对这样的写法非常熟悉，所以你应该可以直接跳过这一小段。当然，我们假设大部份的人都不熟C，那么我以为，如果你觉得太累，也可以晚一点再回来看这一段。因为作者个人的偏见，以为这虽然是Perl的基本语法，可是在实际程序写作时用的机会却比其它方式少了一些。不过我想大家都是好学生，还是让我们来看看基本的for循环应该怎么写呢？还是维持我们的传统，来看看这个例子吧：

```
my $result = 1;
for (my $num = 1; $num <= 10; $num = $num + 1)
{
    $result *= $num;
}
```

跟刚刚的 while/until 终于有些不太一样，而且主要的差别似乎在于这一行：
for (my $num = 1; $num <= 10; $num = $num + 1)
没错，这一行确实就是 for 循环的奥秘所在。首先我们看到小括号里面的三个叙述，这三个分别代表循环的初值，循环的条件，以及每次循环进行后所作的改变。从这个例子来看，我们先定义了一个变量$num，初始值是 1。接下来要求循环执行，只要$num 在不超过 10 的状况下就不断的执行，最后一个则表示，当循环每做一次，$num 的值就被加 1。
当然，这三个部份都是独立的，所以如果你有比较特殊的判断方式，也可以随意修改。例如你可以用总和超过/不超过多少来作为判断的依据。或者每次在执行完一次循环就把$num 的值加 3，这样的写法对于应付比较复杂的状况显然非常好用。
不过如果我们大部份的时候都只是非常有规律的递增，或递减，并且以此为判断循环是否应该结束的依据。既然如此，也许我们还有更清楚，而且简单的方式吗？

### 4.4.2 其实可以用 ...

大家对于Perl程序设计师喜欢简单的体会应该已经非常深刻了，因此我们就来看看怎么样可以让非常具有规则性的for循环可以用更简单的方式来表达吧！例如像我们前面提到的例子：
for (my $num = 1; $num <= 10; $num = $num + 1)
这么简单的for循环还要写这么长一串真是太累人了，记得有一种非常简单的语法吗？

for $num = 1 to 10
没错，如果Perl也可以这样写那就非常口语化了。而且再也不用那么长的叙述句只为了告诉Perl：「给我一个 1 到 10 的循环吧！」不过很显然，Perl的程序设计师找到了更简便的方法，你只要用...就可以表示你需要的循环范围了。

```
for my $num (1...10) {   # 这就是表示$num 从 1 到 10
    print $num;
}

这其实可以写成：

for (1...10) {          # $_ 经常被拿来作为循环的预设变量
    print $_;
}

更简化的写法：

print for (1...10);
```

觉得很神奇吗？其实一点也不会，因为这正是 Perl 程序设计师经常在使用的方式。而你需要更熟悉的也许就是要习惯于这些人对于习惯写作的方式。

### 4.4.3 有趣的递增/递减算符

如果你写过C语言程序，或是你看过其它人写Perl的循环，应该会常常看到这样的叙述句：

```
print for ($i = 1; $i <= 10; $i++);    # 印出
1...10
```

可是对于递增(++)与递减(--)运算子却又是似懂非懂。那么这两个运算子到底在说些什么呢？不过顾名思义，他们主要的工作就是对数字变量进行递增或递减的运算。例如你也许会这么用：

```
my $i = 1;
while ($i <= 10) {
     print $i;
     $i++;                          # 把 $i 加上 1
}
```

没错，于是在 while 循环中，$i 就会由 1 到 10，靠的正是这个递增运算子。当

然，在这里你也可以把这个式子替换为：

$i = $i + 1; # 或是 $i += 1

不过看起来总是没有$i++简洁吧！同样的，递减运算子(--)也是进行类似的工作，也就是每次把你的数值减 1。不过递增或递减运算子总有时候会让人感觉困扰，让我们来看看以下的例子：

```
my $i = 1;
while ($i <= 10) {
    print ++$i."\n";                    # 印出 2...11
}

my $j = 1;
while ($j <= 10) {
    print $j++."\n";                    # 印出 1...10
}
```

从这个例子来看，应该比较清楚，在第一个循环中，Perl 会先帮$i 加 1 之后印出，也就是根据递增（递减）运算子的位置来决定如何运算。当然，我们就可以确定在第二个循环中，递增运算子是怎么运作的。当递增运算子在前时，Perl 会先对操作数进行运算，就像第一个循环的状况，反之，则是在进行完原来的表达式之后，才进行递增（或递减）的运算，也就是像第二个循环中所看到的结果。

### 4.4.4 对于数组内的元素

我们当然可以用 for (;;) 这样的方式来取出数组中的所有元素来运算。不过这时候你只是依照数组的索引顺序，因此你还需要根据索引的值来取得数组中的元素值。就像这样子：

```
my @array = qw/1 2 3 4 5/;
for (my $i = 0; $i < $#array; $i++) {
    print $array[$i];
}
```

显然这样的写法太过繁琐，我们其实可以利用 foreach 来进行更简单的取值动作。那么刚刚的循环部份可以写成：

```
```

```
print foreach (@array);
```

不要怀疑，就是这么简单，不过让我们先来解释一下整个循环的运作。
当你对一个数组使用 foreach 循环时，Perl 就会自动取出这个循环每一个值。接下来，你可以指定 Perl 把取得的值放到某个变量中，例如你可以写成：

```
foreach my $element (@array)
```

不过这时候我们就用到了 Perl 最常用到的预设变量$_，当我们在循环中没有指定任何变量时，Perl 就会把取出来的值放入预设变数$_中。紧接着我们希望把循环取得的值打印出来，也就是执行
print $_;
这样的式子，当然，这样的式子在 Perl 出现的机率真是太少，因为大部份的时候，如果你只要打印单一的$_，Perl 程序员也就会省略$_这个变量。而因为我们在循环中只打算执行 print 这个指令，所以倒装句也就顺势产生。看起来应该显得非常简洁吧！现在你应该还不习惯这样的写法，不过如果你有机会接触其它 Perl 程序员写的程序，那千万要慢慢接受这样的写法。

那么还有一个问题，那就是 foreach 是否只能用在循环的取值，或是 foreach 跟 for 该怎么区别他们的用法呢？
这个问题倒很容易，因为在 Perl 之中，foreach 跟 for 所进行的工作基本上是一模一样，或说他们之中的任何一个都只是另一个的别名。因此只要你可以使用 for 的地方，即使你将他替换为 foreach 也完全可以被 Perl 接受，不过有时候也许你会希望以 foreach 来表达你的算式能够让可能维护你的程序的人比较容易接受，当然，很多时候 Perl 程序员确实不愿意多打四个字母，所以你就需按照当时的语境确定实际执行的是 for 或 foreach 的语意了。

还记得我们在上一章提出的问题吗？如果我们有一个含有整数串行的数组，而我们想取出其中大于零的整数然后取他们的平方值，那么我们应该怎么作呢？现在我们可以尝试来玩玩这个问题。
我们先用循环的方式来解决这个问题：

```
my @array = qw/6 -4 8 12 -22 19 -8 42/;
my @positive;
for (@array) {                                  # 针对数
组的每个元素检察
   push @positive, ($_**2) if ($_ > 0);  # 如果大于零
就取平方值
}
```

```
print for (@positive);
```

那如果使用我们在上一章介绍的函数呢？

```
my @array = qw/6 -4 8 12 -22 19 -8 42/;
my @positive = map { $_**2 }
             grep { $_  > 0 } @array;
# 倒装，把@array 的元素先放进 grep 检查，再把通过检查的结果利用 map 取得平方值放进新的数组

print for (@positive);
```

相当有趣吧，这也就是 Perl 的名言：「解决事情的方法不只一种。」

习题：
1. 算出 1+3+5+...+99 的值
2. 如果我们从 1 加到 n，那么在累加结果不超过 100，n 的最大值应该是多少？
3. 让使用者输入一个数字，如果输入的数字小于 50，则算出他的阶乘，否则就印出数字太大的警告。

# 5. 杂凑(Hash)

杂凑对一般使用者大概都非常不熟悉，尤其是没接触过Perl的人来说，杂凑对他们来说都是全新名词。但是在现实生活中，杂凑却是不断出现在一般人的生活之中。因此只要搞懂杂凑到底在讲什么，你就会觉得这个东西用起来真是自然极了，而且没有了杂凑还可能让很多事情显得不知所措，因为你要花大量的时间跟精力才能利用其它数据结构做出杂凑所达到的结果。

听起来杂凑确实非常吸引人，那我们先来了解一下什么是杂凑。所谓的杂凑，其实用最简单的话来说，也就是一对键值(key-value)，没错，就是这么简单，一个键搭配着一个值的对应方式。当然，你可以搭配Perl所提供复杂的方式来建立多层的杂凑来符合程序的需要，不过那不是基本的杂凑，而且所谓复杂的结构，还是依循着最简单的原理，也就是键跟值的相对应关系。

## 5.1 日常生活的杂凑

没错，如果我们只说杂凑是一对键值的组合，那要让人真正理解显然并不容易。所以如果我们可以使用一般人常用的词汇来解释杂凑这个东西，显然应该会容易许多。既然如此，我们就来看看大家每天接触的数据中，有什么是能够以杂凑精确的表现出来的。

最简单的例子应该算是身份证字号了吧，我们可以很容易的用身份证字号知道一个人的姓名，其中的身份证字号就是杂凑的键(key)，而利用这个键所得到的值(value)就是姓名。而且键是这个杂凑中唯一的值，也就是一个杂凑中，不能有重复的键。这也应该很明显，如果有两个一模一样的身份证字号，那么我们要怎么确认使用者希望找到的是哪一个呢？所以这也是杂凑中的限制，我们必须要求杂凑中的键值必须是不重复的，很显然，这样的限制是非常合理的。另外，我们每个人的行动电话中也藏着使用杂凑的好素材。如果你曾经使用行动电话的电话簿功能，那么你也许每天都会接触这种非常杂凑式的结构，因为电话簿功能也是杂凑足以发挥功用的地方。电话簿就是一整个的杂凑，他里面的键是以姓名为主，值则是这个人的电话。所以你必须为每一个人键入一个独特的键，大多也就是名字，以及这个键所对应的值，当然就是电话了。因为我们只要找到键（姓名）就可以查到依附在这个键的值（电话）。如此一来，我们应该很容易可以理解杂凑的代表意义了。

## 5.2 杂凑的表达

杂凑在Perl中是以百分比符号(%)作为表示，变量的命名方式则维持一贯原则，也就是可以包含字母，数字及底线的字符串，但是不能以数字作为开头。所以你可以像这样的方式定义一个杂凑变量：

```
my %hash;       # 基本的命名方式
my %ID_Hash;    # 包含底线的变量
my %id_hash;    # 大小写还是被认为是不同的字符串
my %_underline;         # 以底线开始的变量名称
```

```
my %2hash;      # 程序会产生错误，因为这不是合法的变量
```

杂凑的存取，我们可以利用大括号来进行，因此我们把所想要取得的杂凑键放入大括号中，就可以藉此找到相对应的值。同样的方式，我们也可以利用这样的形式指定某一对键值，这样的作法非常接近我们存取数组的形式：

```
my %hash;
$hash{key} = 'value';  # 最简单的赋值形式
print $hash{key};
```

就如我们说的，我们是使用大括号({})来标示所要存取的杂凑键，这和使用数组是不同的。不过更重要的是千万别把你的程序写得像这个样子：

```
my $var = 1;
my @var = (1, 2, 3, 4, 5, 6);
my %var;
$var{1} = 2;
$var{3} = 4;
$var{5} = 6;

print $var[2];
```

这样的形式对于 Perl 来说当然是合法的，不过我们显然不希望你用这样的形式来写程序，否则即使 Perl 可以很容易的分辨出来，只怕写程序或维护的人自己还先搞混了。
有时候，我们会忽略一些小地方，那就会让自己找不到杂凑中的值，其中有一个非常重要的部份，也就是杂凑键的数据型态。Perl 会把杂凑键全部转为字符串，这样的转换其实是有些道理的。我们来研究一下这样的程序会发生什么状况呢：

```
my %hash;
$hash{2} = 'two';              # 指定杂凑的一对键值
$hash{'4/2'} = '这是字符串 4/2';       # 注意引号的使用
print $hash{4/2};              # 先运算后转为字符串的键
```

你认为 Perl 会输出什么样的结果呢？答案是'two'。没错，很有趣吧，所以你可以在杂凑键的地方放上一个表达式，那么 Perl 会先进行运算，然后把运算结果转为字符串，所以上面的例子，我们所要求 Perl 输出的其实是$hash{2}，否则你

可以利用引号来指定字符串，就像$hash{'4/2'}这样的方式。我们再看看另一个例子：

```
my %hash;
for (1...5) {
    $hash{$_*2} = $_**2;
}
```

那我们可以得到的杂凑就是像是这个样子：

```
$hash{2} = 1;
$hash{4} = 4;
$hash{6} = 9;
$hash{8} = 16;
$hash{10} = 25;
```

没错，正如我们所预料的，Perl 会把运算出来的结果转为字符串后当成杂凑的键。还记得我们可以利用字符串的内插方式来插入变量到字符串吗？你可以猜测以下的程序会产生出什么不同的结果：

```
my %hash;
for (1...5) {
    $hash{"$_*2"} = $_**2;
}
```

如果你可以想办法看到杂凑的内容，你会发现你得到的杂凑键变成了 "1*2"，"2*2"……。没错，因为他们被视为一个字符串了。所以如果你以为你可以利用$hash{2}或$hash{4}来得到杂凑内的值，恐怕会失望了。所以当你要开始使用杂凑时，可就要小心别搞混了。

## 5.3 杂凑赋值

我们刚刚学到了利用 $hash{2} = 4; 这样的方式来指定一对键值给杂凑，没错，这是赋值给杂凑的最基本方式，不过就跟我们使用数组一样，我们经常需要一次指定大量的杂凑键值，想必Perl的开发者一定也会遇到相同的问题，而且应该有一些合理的解决方案。既然如此，我们应该有其它方式可以一次指定超过一组的键值。利用串行的方式赋值给杂凑就是其中之一，而且当你在定义某个杂凑时就预先知道他的一些键值时特别有用，看看下面的例子：

```
my %hash = qw/1 one 2 two 3 three/;
```

这样的赋值方式看起来跟处理数组时候的方式非常接近，我们利用 qw//来指定一个串行，并且将这个串行赋值给杂凑。这时候，Perl 会按照串行的顺序，分别为【键】，【值】，并且赋予杂凑。所以在这个例子中，所得到的结果就跟我们这么写是一样的：

```
$hash{1} = 'one';
$hash{2} = 'two';
$hash{3} = 'three';
```

或许你会想到某个状况，也就是键值的个数不一的时候。这时候，Perl 会把最后一个键所对应的值设为 undef(注二)，你可以利用这个程序来确认：

```
my %hash = (1, 2, 3, 4, 5);
print 'false' unless defined($hash{5});
```

当然利用串行赋值的方式是方便了一些，可是就像我们刚刚遇到的问题，有时候会发现利用串行赋值的情况似乎比较容易发生错误。尤其当一个串行的元素足够多的时候，你要怎么确认某个串行中的元素应该是键，还是值呢？最简单的方式大概就是进行人工比对，所以你或许可以考虑用另外的方式来赋值给杂凑，就像这样的写法：

```
my %hash = (
          1 => 'one',
          2 => 'two',
          3 => 'three',
          );
```

在这里，我们利用箭号(=>)来表示杂凑中键跟值的相对关系，而且在一对键值的后面加上逗号作为区隔。这样的方式就显得方便、也直觉了许多。不过当你在使用箭号进行指定时，你可能会发现一些不同。因为箭号左边的杂凑键已经完全被视为一个字符串，所以你如果使用这样的方式：

```
```

```
my %hash = (
          4/2 => 3,
          );
print $hash{'4/2'};
print $hash{2};
```

别忘了，跟之前的状况一样，Perl 还是会帮你先把箭号左边的表达式算出结果，然后转成字符串，作为杂凑的键。所以当你在取值时使用了引号确保你要找杂凑键等于'4/2'的值时，你就没办法找到任何结果，因为目前杂凑中只有一个杂凑键为'2'的值。
要从杂凑中取出现有的值以目前的方式应该足够方便，你只需要知道杂凑中的键，就可以取得他的内容值。不过这样显然还不够，因为杂凑跟数组还是有着相当的差异。在数组中，你可以很清楚的知道数组的索引值是从 0 到最后一个数组的大小减 1，可是在杂凑中却并不是这么一回事。如果你没办法知道杂凑的键，又怎么取出他的值呢？那么这个时候，你应该考虑先把整个杂凑读过一次。

## 5.4 each

就像在数组当中，你可以使用foreach这样的循环来找到数组中的每一个值，当然我们也经常需要在杂凑中进行类似的工作，我们希望可以在杂凑中能一次取出所有的键，值。所以你必须仰赖类似foreach的工具来帮助你，那就是each函数。例如你可以利用下面的写法读出刚刚我们所建立起来的杂凑：

```
while (my ($key, $value) = each (%hash)) {
     # 取出杂凑中的每一对键值，并且分别放入$key, $value
     print "$key => $value\n";
}
```

很明显的，每次 each 函数都会送回了一个包含两个值的串行，其中这两个值分别是一个杂凑键跟相对应的值。因此我们把取回的串行指定给$key 和$value 两个变量，接着印出结果，就可以看到一对一对的键值了。而当传回空数组时，while 判断就会变成伪值，while 循环也就结束了。利用这样的函式对我们有很大的帮助，如果我们想要整理一个杂凑的内容，我们可以在完全不知道杂凑中有什么内容的状况下开始进行处理。使用 each 函数在处理杂凑时是让事情显得容易许多，可是有时候还是有点不方便的地方，举例来说：如果我有一个包含着主机 ip 跟主机名称的杂凑，虽然我不知道杂凑里面到底有多少数据，可是我却希望能找出所有的杂凑键值，然后取出以 192 开始的 ip 地址。这时候如果使用 each 来作，那就必须先把所有的键值取出，然后再一一进行比对，所以也许程序就像这样：

```
```

```
my %hash = (
          '168.1.2.1' => 'verdi',
          '192.1.2.2' => 'wagner',
          '168.1.2.3' => 'beethoven',
          );                          # 定义主机跟 ip 的
对应
my @hostname;
while (my ($key, $value) = each (%hash)) {
    if ($key =~ /^192/) {             # 要找出 ip 以 192
开头的部份
       push @hostname, $value;               # 找到之
后放入新的数组中
    }
}

print @hostname;
```

很显然，这样的写法确实可以让程序正确的找出我们要的结果，不过我们总是还会继续思考可以有更干净利落的写法，毕竟使用 Perl 的程序设计师都不太喜欢拉拉杂杂的程序。所以有什么方法可以让过滤出需要的键值可以显得方便些呢？

## 5.5 keys跟values

如果我们可以用简单的方式一次取得杂凑的所有键(keys)，那么要进行过去的过程就非常容易，而我们所需要的就是过滤后留下来的键，跟他们的相对值。当然，有某些时候，你可能只想要拿到杂凑中的所有值，这时候你就不需要担心他们是属于什么键的相关。为了因应这样的需求，有两个函数可以满足我们，他们分别是keys跟values。很显然的，这两个函数所作的工作就是取出杂凑的键跟值。和使用 each相当不同的是：你可以只单读取出所有的键，或所有的值，而不需要一次全部取出。
例如我们可以用这样来把杂凑键放在同一个数组中：

```
my @keys = keys(%hash);
```

如果你希望取出所有的值，那么不妨这样写：

```
my @values = values(%hash);
```

当然，你可以用他来完成 each 的工作，就像这样：

```
my @keys = keys(%hash);
for (@keys) {
   print "$_ => $hash{$_}\n";
}
```

其实跟这么写是一样的效果：

```
while (my ($key, $value) = each(%hash)) {
     print "$key => $value\n";
}
```

不过你显然会发现，有时候用 keys/values 比较简单，有时候用 each 比较方便，当然，至于要使用何者是完全取决于你所想要得出的结果，或者你认为最省力，简洁，或是效率比较好的写法。

在杂凑中使用 keys/values 这两个函数都传回串行，因此我们可以把我们所得到的串行轻易的放入数组，接下来再以数组的方式进行运算。这样的方便之处在于我们可以有很多可供利用的数组函数，所以我们可以把刚刚的那个例子改写成这样：

```
my %hash = (
          '168.1.2.1' => 'verdi',
          '192.1.2.2' => 'wagner',
          '168.1.2.3' => 'beethoven',
          );

my @keys =  map { $hash{$_} }
          grep { (m/^192/) } keys(%hash);

print @keys;
```

这样的写法比起之前的方式看起来是不是干净许多了呢？我们来看看最关键的一行，结果到底怎么产生的：我们先用 keys 函数取出杂凑中的所有键，就如我们所说的，这个函数传回一个串行。然后我们对所得到的串行进行过滤，利用 grep 取出串行中以 192 开头的 ip 子串行，最后利用 map 一一比对得出杂凑中以对应这些 ip 的主机名称。

## 5.6 杂凑的操作

毫无疑问，杂凑这样的数据结构对于程序的写作有着莫大的帮助，但是我们必须

能熟悉对杂凑的操作才能够让我们更容易发挥杂凑的功能。其中最重要的大概就是exists跟delete两个函数了，这两个函式能让我们有效的掌握杂凑的元素，同时它们也是perl内建相关于杂凑函数的最后两个(注一)。

## 5.6.1 exists

我们就继续用ip跟主机的杂凑当例子吧。假如我有一个ip，我不确定我是否有这部主机的数据，如果我们只用刚刚的方法，那我们就必须取得所有的ip，然后把手上的ip跟取得的ip串行一一比对，以便确定自己有没有这个ip的主机数据。所以我们的程序也许长的像这样：

```
my %hash = (
          '168.1.2.1' => 'verdi',
          '192.1.2.2' => 'wagner',
          '168.1.2.3' => 'beethoven',
          );
my $ip = '192.1.2.2';
print "bingo" if ($hash{$ip});
```

在这里，我们有一个杂凑，其中三个键分别是'168.1.2.1'，'192.1.2.2'，'168.1.2.3'，而我们希望判定目前手上的一组 ip'192.1.2.2'是不是我们主机所拥有的 ip。于是我们利用这个 ip 作为杂凑键，并判断如果取得的值为真，那么我们就说这个 ip 属于杂凑的其中一个键，这样的想法似乎暂时解决了我们的需求。不过我们来看看下面的例子：

```
my %hash = (
          'cd' => 2,
          'book' => 10,
          'video' => 0,
          );
my $media = 'video';
print "bingo" if ($hash{$media});
```

我们假设这是某个小区图书馆目前外借的东西数量，其中的键就是代表则可以外借的图书馆资产，其中包含了 CD，书跟录像带。而所对应到的值则是他们目前被借出的数量。我们看到，CD 被借走了两套，书被借走了十本，而录像带则是原封不动，一卷也没被借走。是的，大家都不喜欢录像带了。
这时候，我们希望知道图书馆是否提供录像带外借，也就是要检查 video 这个键是否存在。于是我们利用刚刚的方式，看看$hash{$media}是否传回真值。很遗憾，因为录像带这个键目前的值是 0，因此当我们利用录像带当成键来取的相对

应的值时，Perl 会传回 0 给我们。而我们知道 0 其实是个伪值。于是我们以为'video'这个键并不存在于这个杂凑中，也就是说这个图书馆并没有录像带出借，但是这样的结果跟我们的认知有所不同，因为取得的值为 0 只是代表目前没人借出。所以我们发现这个方法并不正确，至少我们已经知道他会产生错误的结果。所以我们必须尝试其它方法，例如利用 keys 找到包含所有索引键的串行，然后进行一一的比对。就像这样：

```
print "exist" if (grep { $_ eq 'video' } keys
(%hash));
```

这样就可以确定某个键是否存在于这个杂凑，可是程序还是有点长，而且我们也许必须经常去判断某个值是否为杂凑的键。所幸 Perl 提供了简洁的函式可以使用，所以利用 exists 这个函式让我们有了极佳的判断方式。有了 exist 之后，对于刚刚那一行程序，我们只需要这么改写：

```
print "exists" if (exists $hash{video});
```

这样的写法显然轻松了许多。

### 5.6.2 delete

有些时候，我们也会遇到某些键值我们不再需要的状况，这时候如果可以把这些没有必要的键值移除似乎是非常必要的。所以Perl也提供了移除杂凑键值的函式，也就是delete。这个函式的使用其实非常容易，你只需要指定想要删除的某一个杂凑键，就像这样：

```
delete $hash{video};
```

当然，所谓的移除是指这个键将不再存在于这个杂凑，而不是指让这个键对应的杂凑值消失。所以并不是把需要被 delete 的这对键值设为 undef。也就是说，即使有一个键所对应的杂凑值为 undef，那这个键依然被视为存在(exists)的，这在刚刚解释 exists 这个函数的例子中就可以了解了。

## 5.7 怎么让杂凑上手

在Perl中要使用杂凑，有一些重点也许还是应该提醒大家的。首先，Perl对于杂凑的大小限制依然采取了「放任」的态度，也就是以最没有限制的方式。只要计算机可以容量的大小，Perl都可以接受。因此程序设计师可以有很大的挥洒空间，只是也必须注意避免让系统因为被Perl占用太多资源而导致无法正常运作。

另外，使用者可以利用任何的纯量值来表示杂凑中的键与值。可是在杂凑键的部份，Perl会把所有的键转换为字符串。所以如果你在不注意的情况下把表达式当成杂凑的键，Perl会帮你先进行运算，然后利用运算所得的结果作为杂凑键，这样的情况可能会有出乎意料的结果。当然，如果你使用表达式来作为杂凑的键值，那就应该有些准备，因此应该会更小心的注意，而我们也在前面提到了不少例子。另外，你还会希望知道自己什么时候该用杂凑，这就必须依赖你对于杂凑的感觉，最基本的原则还是以杂凑的特性来看，如果你有一个可以辨识的键，而且希望藉由这个键找到相关连的值，这时候你几乎就可以放心的使用杂凑了，只不过这里所谓的值当然不限定单指特定的值，而可能是任何一种纯量值，也就是因为这个特性，可以让我们搭建出复杂的杂凑结构，不过这个部份则是属于进阶的内容，我们就不在这里解释。就像我们所举的例子，你可以利用ip作为每一部主机的辨识，那么你可以藉由ip找到那部机器的相关数据。
还有一个常常被搞混的问题，也就是杂凑的顺序。许多人想当然尔，以为杂凑的顺序是依照新增的顺序来决定的。其实事实并非如此，杂凑的排列方式并非按照使用者加入的顺序，而是Perl会依照内部的算法找出最佳化的排列。

习题：
1. 将下列数据建立一个杂凑：
John => 1982.1.5
Paul => 1978.11.3
Lee => 1976.3.2
Mary => 1980.6.23
2. 印出 1980 年以后出生的人跟他们的生日。
3. 新增两笔资料到杂凑中：
Kayle => 1984.6.12
Ray => 1978.5.29
4. 检查在不修改程序代码的情况下，能否达成第二题的题目需求

注一：可以利用perldoc perlfunc来查看perl所提供的函数。
注二：其实，如果你在程序里打开了警告讯息的选项，这样的指定会让Perl产生警告讯息："Odd number of elements in hash assignment"。

# 6. 副例程

当你的程序在很多时候总是不断在进行类似的工作时，而且你总是为了这些工作在写相同的重复程序代码时，你应该考虑替你自己把这些段落用更简洁的方式来萃取出一个可以重复使用的区块。这时候，你就可以考虑使用副例程，不过如果要更清楚的了解副例程的意义，也许我们应该先来看看这样的例子：

```
my @array = qw/6 8 -4 18 -9 -22 36 48/;
my @new_array = map { ($_**2) }
             grep { $_ > 0 } @array;
print @new_array;
print "\n";
my @array2 = qw/16 8 -24 8 -12 20 16 28/;
my @new_array2 = map { ($_**2) }
             grep { $_ > 0 } @array2;
print @new_array2;
```

从这几行程序，你应该很容易看出一些端倪，因为我们发现在这个小小的程序中，几乎除了数组中的元素不同之外，其它的几行程序显然都在进行相同的工作。也许你觉得只有两个数组需要运算还算容易，不过当你需要进行相同运算的数组达到十个，或二十个的时候，你总不希望又把相同的程序代码进行大量的复制吧？也许你会认为复制，贴上这样的动作总比弄懂副例程来得简单许多，不过你也许需要考虑，当你的程序需要进行修改怎么办？让我们来举一个例子吧，如果你的程序在取平方值的地方现在希望可以取立方值，或甚至进行更复杂的运算，而你现在总共复制了一百份的程序代码在做同样的运算。没错，恭喜你，你现在就因为这样小小的改变而必须修改这一百个地方，而且这对程序的除错会变成非常大的负担。除非你认为你的工作时间就应该花在这样的工人智慧上，否则你确实要考虑来使用副例程了。当然，如果你还是坚持不用副例程，千万别找别人去维护你的程序。

使用副例程不但可以增加程序的可重用性，降低维护成本，当然还可以提升程序的可读性。就像刚刚的例子，你对于每次需要进行某个运算时，只需要使用不同的参数，而如果运算部份的程序需要修改时，也只需要修改一次。对于想要看程序代码的人来说，他也可以清楚的看懂这一部份到底进行什么样的工作。因此适当的在自己的程序中使用副例程确实是非常必要而且提高效率的方式。

## 6.1 关于Perl的副例程

在Perl中，使用副例程并不困难，尤其当你曾经使用其它程序语言写过副例程，那么Perl的副例程对你来说更是容易上手。不过我们假设你从来没写过任何程序语言，那么我们准备从Perl来学习副例程了。

在Perl中，我们可以用&来表明副例程，而标识符的命名方式也是和其它变量的命名方式相同。也就是可以用数字，底线和字母组成，但是不能以数字开始。关

于副例程，大概有两个部份你必须特别注意的，也就是副例程的定义跟调用。也就是副例程本身的区块以及使用副例程。就像我们刚刚说到的，副例程的本身是以&符号作为辨识，所以如果你有一个叫做DoSub的副例程，那么你就可以利用&DoSub的方式来调用。当然，利用&符号调用副例程本身，并不是绝对必要的，除非你的副例程名称和Perl内建的函式名称有所重迭，否则你其实可以省略&，也就是说，你可以直接使用DoSub来调用你自己写的DoSub副例程。当然，有时候我们会建议你在调用副例程时尽量加上&符号，除非你能够非常确定你使用的副例程名称和Perl并不重复。但是当你看到许多程序设计师都省略&符号时，可别以为他们写错了，他们也许都是经验老练的高手，已经能轻易确定自己所使用的函式名称不会发生冲突。如果你还是不太了解我们解释的意思，那还是让我们来看看这样的写法吧：

```
my $num = 12;
print hex($num),"\n";          # 这是 Perl 提供的 hex 函式
print &hex($num),"\n";         # 我们自己写的 hex 副例程


sub hex {
    my $param = shift;
    $num*2;
}
```

这个例子应该就可以非常简单的看出&带来的不同，我们第一次使用了 hex 来呼叫函式，因为 Perl 内建了 hex 这个函式，所以 Perl 会直接使用内建的 hex 函式，而第二次我们使用&hex 呼叫时，才真正叫用了我们自己定义的副例程 hex。
不过说了这么多，我们还是必须在开始叫用副例程之前，先尝试写出你自己的副例程。也就是主要运作的那一个部份，你可以使用 sub 这个 Perl 的关键词来定义一个副例程，而且既然我们已经使用了 sub 这个关键词，你总不会还以为我们会喜欢多打一个&字符吧，所以最典型的副例程大多会长的像这样子：

```
sub subroutine {
    ...
    ...
}
```

当然，副例程内的缩排并非绝对必要的，不过为了保持程序的可读性跟维持好的程序写作习惯，我们还是极力建议各位在进行程序写作时，能够养成区块内的缩排。在Perl中，一般的使用情况下(注一)，你可以把副例程放在程序中的任何位置，只要你调用的时候，能够让程序本身不至于找不到副例程而发生错误就可以。虽然笔者自己习惯把副例程放在最后，不过对于已经有其它程序语言写作习惯的

人来说，也许有规定副例程的位置，而养成在程序的一开始就定义出程序中用了那些副例程的习惯。不过不管如何，Perl对于这些情况都是允许的，所以你可以试着找到自己习惯的方式。
另外，在Perl中使用副例程还有一个特点，也就是对于程序中全域变量的存取。由于副例程也是属于程序的一部份(对Perl来说，那就是另一个程序中的区块)，因此在Perl的设计中，你可以任意的存取程序中的全域变量，就像我们之前使用的那些变量。对很多使用其它程序语言的人来说，这实在非常不可想象，当然，也因此持反对意见的人应该也不在少数。不过保留这样的功能却未必就是鼓励使用者以这样的方式来写程序，而只是保留某种弹性的空间。笔者还是建议各位能尽量使用参数的方式，并且使用副例程中的私有变量，这样的建议当然是有一些理由的，因为你很可能在程序的发展过程中写了一些副例程，并且把他们放在程序之中，等到程序慢慢成熟之后，你也许就可以把这些副例程放进模块里，以方便建立程序的可重用性(注二)，以及属于自己的函式库。这时候，如果你的副例程足够独立的话，那么搬移的工作就可以轻松许多，也不容易产生一些难以除错的状况。相反的，如果你在开始使用副例程的时候就大量使用全域变量时，你可能会发现要把这些副例程放入模块中就显得特别困难。不过有时候能够在副例程中使用全域变量也是非常方便的，例如有些模块中，你可能会有一些不提供给外部使用的副例程，这时候你也许会直接叫用程序的全域变量。
现在，我们来写我们的第一个副例程：

```
sub hello {
    print "hello\n";
}
```

没错，这个副例程虽然简单，但是却能够让我们一窥副例程的奥秘，所以当你叫用这个副例程时，他只会印出"hello"这个字符串。那我们就来试试：

```
&hello;       # 印出 hello
&hello;       # 再印一次

sub hello {
    print "hello\n";
}
```

你应该发现了，副例程就是这么简单。

## 6.2 参数

没错，副例程的用法其实并不太困难，不过要能发挥副例程更重要的功能，可就还要下些功夫了，也就是让副例程能根据我们的需求进行不同的响应。所以我们

应该想办法让副例程能根据我们的需求来进行一些调整，进行不同的运算。首先，我们需要的是参数，所谓的参数也就是需求不同的那一个部份，利用参数来告诉副例程我们所需要的调整。使用参数，当然会有传入跟接收的部份。发送端也就是调用的部份，也就是我们要告诉副例程，我们需要进行调整的内容，我们只要直接把所要传送的值放进小括号内，就像这样：

&hello('world');

不过只有传送当然是不够的，我们的副例程也需要知道外面的世界发生了什么事，它需要接收一些信息。那我们来看看传送的信息去哪里了，我们先来做个实验：

```
&hello("world");    # 我们传了参数 "world"

sub hello {
   print @_;              # 原来参数传到这里了
}
```

我们可以看到，当我们呼叫副例程，并且把参数传给副例程时，参数会被放到预设的数组变量@_里。这样我们就可以调用参数来进行操作了。既然如此，我们来改写 hello 这个副例程吧！

```
&hello("world");    # 传参数"world"

sub hello {
   my $name = shift @_;  # 把参数从预设数组拿出来
   print "hello $name\n";  # 根据参数不同印出不同的招呼
}
```

## 6.3 返回值

大多数的时候，我们除了参数，还会希望副例程可以有回传值，也就是让副例程利用我们的参数运算之后，也能够传回运算结果给我们，比如我们想要写一个找阶乘(注四)的副例程，因此我们告诉副例程，我们希望找到某个数的阶乘，而当然也期望从副例程得到运算的结果，也就是我们需要副例程的回传值。最简单的方式，就是在副例程中使用return这个指令来要求副例程回传某个值。我们可以试着把阶乘的副例程写出来：

```
my $return = &times(4);                # 把回传值放到变数
$return
print $return;

sub times {
    my $max = shift;           # 把参数指定为变数$max
    my $total = 1;             # 如果不指定，预设会是 0，那
乘法会产生错误
    for (1...$max) {           # 从 1 到 $max
  $total *= $_;                        # 进行阶乘的动作
    }
    return $total              # 传回总数
}
```

在这里，又有一些简便的使用方式来处理 Perl 的传回值，因为 Perl 会把副例程中最后一个运算的值当成预设的回传值，所以你可以省略在进行运算后还必须再进行一次 return 的动作。就像这样的写法：

```
my $return = &square(4);
print $return;

sub square {
    my $base = shift;
    $base**2;
}
```

这时候，我们看到副例程的最后一次运算是把参数进行了一次取平方的动作，而这个运算结果就会直接被 Perl 当为回传值，所以你就不需要再另外进行回传的动作。这样确实可以简化写副例程时的手续，继续维持了 Perl 的简朴风格。当然，如果你还是不太熟悉这种回传的方式，你还是可以加上 return 的叙述，不过当你在看其它 Perl 程序设计师的程序时，可别被这样的写法搞混了。

## 6.4 再谈参数

我们已经知道了在副例程中怎么使用参数及回传值，而且我们还看到了Perl在处理参数时所使用的预设数组。聪明的读者应该早就猜到，当我们使用超过一个的参数时，应该就是依照数组的规则一个一个被填入预设的数组中，因此我们也可以按照这样的原则来取出使用。我们可以用刚刚的概念，很容易的理解多个参数时的运用：

```
my $return = &div(4, 2);          # 这时候有两个
参数
print $return;

sub div {
    $_[0]/$_[1];            # 只是进行除法
}
```

这样的写法其实非常粗糙，不过我们只是举例来说明副例程的参数运用。这次我们直接取出预设数组中的元素来进行预算，因为只有一个表达式，所以运算结果也自然的被当成回传值了。这样的运用方式非常的简明，所以当你在写副例程的时候，你便可以使用许多组的参数。不过如果我们在叫用副例程的时候传了三个参数，就像：

&div(4, 2, 6);

那会产生什么结果呢？其实回传值就跟原来的一样，因为 Perl 并不会去在意参数的个数问题。不过如果你的程序有需要，应该去确认参数的个数，避免参数个数无法应付需要，以确保程序能正常而顺利的进行。
既然 Perl 的参数是以数组的方式储存，而我们也知道，Perl 的数组并没有大小的限制，也就是以系统的限制为准。那么我们很容易的可以传入多个参数，而且还可以正确的运算并且回传运算的结果。就像这样：

```
my $return = &adv(4, 2, 6, 4, 9);  # 我们一次传入五个参
数
print $return;

sub adv {
    my $total;
    for (@_) {                          # 针对预设数组进行
运算
  $total += $_;                         # 加总
    }
    $total/($#_+1);             # 除以总数 (取平均)
}
```

这时候，不论你的参数个数多少，Perl 都可以轻松的应付，然后算出所有参数的平均值而且这时候。而且我们所需要的就是不管使用者有多少参数，都可以正确的算出他们的平均值。不过使用不定个数的时机或许不像固定参数个数来得频繁，很多时候，我们都会使用固定的参数个数，然后确定每个参数的用途。当然

这样的用法有时候会让人产生一些困扰，尤其在你的程序会被大量重用时(注五)，不过要考虑这个问题还需要对 Perl 有更深入的了解，所以暂时我们就先不讨论这种深入的用法。

## 6.5 副例程中的变量使用

就像大部分人所想的，副例程也是一个区块，所以有属于这个区块自己的变量，也就是副例程的私有变量。不过就如我们所说的，副例程是可以使用程序中的全域变量，就像程序中的其它区块一般。因此我们只需要在副例程中宣告my变量，也就是定义了副例程的私有变量。那么就像我们知道的，变量将会维持到这个区块的结束，也就是你无法在程序的其它地方存取这个变量。

另外，在副例程中，还有一种相当特殊的变量，也就是利用local来定义变量。不过这个部份目前用的人已经非常的少，所以你可以记着副例程里面有这样的用法，然后跳过一个部份。而我们打算在这里提出来的原因是因为各位也许会有机会在某些程序里面看到这样的用法，为了避免大家看到这种用法却又不知道它的作用，我们就在这里简单的介绍local的用法，让大家未来有机会看到时可以能有一些参考的数据。

其实local的用途在于确认某些变量是在副例程中私用的，可是因为副例程会有机会被其它程序引用，所以你无法预期在某个引用的程序之中是否也有名称相同的变量。因此使用local来确立这是副例程中的私有变量，而如果原来的程序中有相同的变量名称时，就把主程序的变量放入堆栈，也就是先暂时储存了主程序的这个变量，然后把相同的变量名称清空以提供副例程使用。一但离开了副例程之后，Perl就会复原原来被储存，并且清空的变量了。这样子看起来，local和my的用法看起来似乎非常接近。

当然，你会发现这跟my之间会有什么差异呢？我们先来看看这个程序：

```
$var1 = "global";
&sub1;                          # 印出 sub1
print "$var1\n";          # 印出 global
&sub2;                          # 现在变成 sub2
print "$var1\n";          # 又回到 global

sub sub1 {
    my $var1 = "sub1";
    print "$var1\n";
}

sub sub2 {
    local $var1 = "sub2";
    print "$var1\n";
}
```

看起来没什么不同，好像两者之间没有太大的差别，可是如果我们改写一下程序：

```
$var1 = "global";
$var2 = "for local";
&sub1;                              # 印出 local,
for local
&sub2;                              # 印出 global,
for local

sub sub1 {
    local $var1 = "local";
    my $var2 = "my";
    &sub2;
}

sub sub2 {
    print "var1=$var1\tvar2=$var2\n";
}
```

从这里，我们好像可以发现一些不同。差别就在于当我们先呼叫 sub1 的时候，sub1 会把原来的变量$var1 放进堆栈，清空后把新的值"local"放入。而在呼叫 sub2 的时候，因为还在 sub1 的区块内，因此 local 还占用着$var1 这个变量。所以印出"local"的值，可是使用 my 就有所不同的。虽然我们在 sub1 使用了 my 来定义区域变量$var2，可是 my 却不会把占用原来$var2 变量的空间。所以当我们呼叫 sub2 时，会使用 sub2 里的$var2 变量。而在 sub2 里面因为没有定义$var2，所以 Perl 直接叫用全域变量，也就印出了"for local"的字符串。

习题：
1. 下面有一段程序，包含了一个数组，以及一个副例程 diff。其中 diff 这个副例程的功能在于算出数组中最大与最小数值之间的差距。请试着将这个副例程补上。

```
#!/usr/bin/perl -w

use strict;

my @array = (23, 54, 12, 64, 23);
my $ret = diff(@array);
print "$ret\n";     # 印出 52 (64 - 12)
```

```
my @array2 = (42, 33, 71, 19, 52, 3);
my $ret2 = diff(@array2);
print "$ret2\n";  # 印出 68 (71 - 3)
```

2. 把第四章计算阶乘的程序改写为副例程型态，利用参数传入所要求得的阶乘数。

注一：如果你想要了也更复杂的副例程使用方式，可以参考 perldoc perlsub。
注二：就像你弄出了小螺丝钉，你总不希望每次遇到一样的需要就重作一次螺丝钉。
注三：笔着第一次学 Perl 的时候就是被预设变量$_打败的。(XXX 正文中没有出现)
注四：阶乘就是从一乘到某个数，比如 4 的阶乘就是 1x2x3x4。
注五：也就是你的副例程被放入模块中，而会不断被重用时。那么你固定的参数的个数及顺序，一但将来副例程要改写时，很容易影响过去使用的程序代码，而产生无法正确执行的问题。不过这属于进阶的问题，我们并不在这里讨论。

# 7. 正规表示式

正规表示式其实并不是Perl的专利，相反的，在很多Unix系统中都一直有不少人使用正规表示式在处理他们日常生活的工作。尤其在许多Unix系统中的log更是发挥正规表示式的最好历练，系统把所有发生过的状况都存在log档之中，可是你应该怎么找出你要的信息，并且统计成有用的资料。当然，大部份的Unix管理员可以求助许多工具，不过很大多数的状况下，这些工具也是利用正规表示式在进行，所以如果说一个足够深入管理Unix的系统管理员都曾经直接，或是间接的使用过正规表示式，我想应该很少人会反对吧。不过这显然也充分的表达出正规表示式的重要性。

## 7.1 Perl 的第二把利剑

没错，正规表示式并不是Perl所独有，或由Perl首创。可是在Perl之中却被充分发挥，还有人说如果Perl少掉了杂凑跟正规表示式，那可就什么都不是了。情况也许没有这么夸张，可是却可以从这里明显感觉出来正规表示式在Perl世界中所占有的地位。对于许多人而言，听到Perl的时候总不免听其它人介绍Perl的文字处理能力，而这当然也大多是拜正规表示式所赐。

## 7.2 什么是正规表示式

讲了那么多，那到底什么是正规表示式呢？简单的说，就是样式比对。大部份的人用过各种文字处理器，文书编辑器，应该或多或少都用过编辑器里面的搜寻功能，或是比对的功能吧！我彷佛听到有人回答：那是基本功能啊。是啊，而且那也是最基本的样式表示。就像我要在一大堆的文字中找到某个字符串，这确实是非常需要的功能。不过如果你写过其它程序语言，那么你不妨回想一下，这样的需求你应该怎么表达呢？或者假设你现在是公司的网络管理员，如果你拿到一个邮件服务器的log档案，你希望找到所有寄给某个同事的所有邮件寄送数据，而你现在手上也许正在使用C或Java，或其它程序语言，你要怎么完成你的工作呢？这样说好像太过抽象，也许我们应该来举个文件中搜寻关键词的例子。
例如我希望在perlfunc这份Perl文件中找sort这个字符串，这样的需求很简单，大部份的时候你也都可以完成这样的程序。可是我如果希望找到sort或者delete呢？好吧，虽然麻烦，不过多花点时间还是没问题的。不过实际去找了之后，我发现找出来的结果真是非常的多。于是我看到某些找到的结果是这样的：

```
sort SUBNAME LIST
     sort BLOCK LIST
     sort LIST
```

没错，这些正是我想要找的结果，可是如果一个一个找也实在太辛苦了。所以如果我可以把这些东西写成一个样式，让程序去辨别这样样式，符合样式条件的才传回来，这样一来，应该比较符合我们的期待了。而所谓符合的条件，也就是我们所希望的「样式」，于是我们开始想象这个样式会是什么样子，在这个例子中，

我们开始设计我们需要的样式：以 sort 开始，中间可能有一些其它的字，可能没有，最后接着一个 LIST，于是符合这样的样式都是我们所要搜寻的结果。相反的，如果在文章中其它地方出现的 sort，可是并没有符合我们的样式，那么也不能算是成功的比对。
就这样，当我们再度拿起其它程序语言时，好像忽然觉得很难下手，因为要完成这样的工作，显然是非常的艰辛。不过在 Perl 的正规表示式中，这才是刚开始。因为你也许会希望在浩瀚的网络中找到你想要的某些数据，你也许知道某个网站有你所需要的信息，比如每天的股票收盘价格，而你希望程序每天自动收集这些信息之后自动去分析股票的走势。当然，也许你已经可以每天派出机器人去各大新闻网站收集最新的消息，可是你也许需要利用正规表示式去萃取对你有帮助的新闻内容。或者你根本就想模仿 google，去进行新闻的比对，然后过滤掉相同的新闻，利用机器人完成一份足够动人的报纸。当然，并不是用了正规表示式就可以轻易完成这些工作，不过相较于其它开发工具，Perl 在这方面显然占有相当大的优势。

## 7.3 样式比对

在Perl中，你要进行比对前，应该先产生出一个你所需要的「样式(pattern)」，也就是说，你必须告诉Perl：在寻找的目标里，如果发现存在着我所指定的样式，就回传给我。也就是说，你必须告诉Perl，我需要的东西大概长的像这个样子，如果你有任何发现，就回传给我。
所以样式的写法与精准与否就会影响比对的结果，通常而言，如果你发现比对出来的结果跟你的想象有所差距，那么你显然应该从比对的样式着手，看看样式上到底出了什么差错。因为当你把结果反过来跟原来所写的样式比对，就会发现这些回传结果确实还是符合比对的样式。当然，要写出正确的样式是必须很花精神的，或者应该说要非常小心的。
如果我们要以简单的方式来描述样式的模型，那么我们可以说样式其实是由一个个单一位所组成出来的一个比对字符串。例如最简单的一个单字是一个样式，就像你写了一个"Perl"，他就是一个样式。可是在样式中也可能有一些特殊符号，他们虽然没办法用一般的字符来表示，可是使用了特殊符号之后，在Perl的比对中，他们还是逐字符的进行比对。很常见的就是我们在打印程序结果也会用到的"\t"或是"\n"等等。所以如果你写了这样的一个字符串，他也算是一个比对的样式："Perl\tPython\tPHP"
另外，你还可能会用到一些量词，也就是用来表达数量。量词的使用对于Perl的正规表示式中是占有重要地位的，因为使用了比对量词，你就可以让你的比对样式开始具有弹性。例如你想在你的比对字符串内找到一个字，这个字可能是：

```
wow
woow
wooow
```

不过你又不想要把每一个字都放到你的比对样式中，所谓的每一个字就也许包含

了'wow'，'woow'，'wooow'…，而且也许他们有可能会变成"woooooooow"，甚至中间夹杂了更多的"o"，甚至在你写程序的时候也都还无法预测中间会出现多少次的'o'，这时候就是你需要使用量词的时候了。另外还有许多技巧跟参数，例如你希望进行忽略大小写的比对，或是你希望这个样式只出现在句首或句尾等等，而这种种的东西都是拿来描述比对的样式，让 Perl 能更精准的比对出你所需要的字符串。而在 Perl 之中使用正规表示式其实有许多的技巧，我们接下来就是要来讨论该怎么学习这些技巧。

## 7.4 Perl 怎么比对

我们之前提过，Perl所使用的是逐字符比对，也就是说，Perl根据你的样式去目标内容一个字符一个字符进行比对。例如你的目标内容是字符串 "I am a perl monger"，而你的样式是字符串"monger"。那么Perl会根据样式中的第一个字符"m"去字符串中比对，当他浏览过"I"，空格键，"a"之后，他遇到了句子中的第一个"m"字符。于是Perl拿出样式字符串中的第二个字符"o"，可是目标字符串的下一个字符却是另一个空格键，于是Perl退回到比对字符串的第一个字符"m"继续比对。
就这样继续前进，一直到Perl找到下一个"m"。于是又拿出比对样式的第二个字符"o"，发现也符合目标字符串的下一个字符。然后继续往前进，等到Perl把整个比对字符串都完成，并且在目标字符串对应到相同的字符串，整个比对的结果就传回 1，也就是进行了成功的比对。
也许我们可以用图示的方式来表达Perl在正规表示式中的比对方式。

[图]

## 7.5 怎么开始使用正规表示式

如果你对Perl进行比对的方式有点理解，那么要怎么开始写自己的正规表示式呢？
首先，我们要先知道，Perl使用了一个比对的运算子(=~)，也就是利用这个运算子来让Perl知道接下来是要进行比对。接下来，就要告诉Perl你所要使用的样式，在Perl中，你可以用m//来括住你的样式。而就像其它的括号表达，//也可以替换为其它成对出现的符号，例如你可以用m{}，m||，或是m!!来表达你的样式。不过对于习惯使用传统的m//作为样式表达的程序设计师来说，Perl倒是允许他们可以省略"m"这个代表比对(match)的字符。所以下面的方式都可以用来进行正规表示式：

```
$string =~ m/$patten/
$string =~ m{$patten}
$string =~ m|$patten|
$string =~ m!$patten!
$string =~ /$patten/
```

Perl 在完成比对之后，会传回成功与否的数值，所以你可以将正规表示式放到判断式中，作为程序流程控制的决定因素。不过也仅止于此，也就是说，当比对成功时，正规表示式就会结束，而且传回比对成功的结果。当然，如果 Perl 比对到字符串结束还是没有找到符合比对样式的字符串，那么比对依然会结束，然后 Perl 会传回比对失败的结果。例如下面的例子就是一个利用正规表示式来控制程序的例子：

```
my $answer = "monger";
until ((my $patten = <STDIN>) =~ /$answer/) {
    # 持续进行，直到使用者输入含有 monger 的字符串
    print "wrong\n";      # 在这里，表示比对失败
};
```

我们首先定义了一个字符串"monger"，并且把这个字符串作为我们的比对样式，其实我们也可以直接把这个样式放到正规表示式装，不过我们在这里只是让大家可以比叫清楚的分辨出样式的内容。。接下来，我们从标准输入装置（一般就是键盘）读取使用者输入的字符串，并且把读进来的字符串放到变量$patten 中，接下来再去判断使用者是否输入含有"monger"的字符串，如果没有，就一直持续等候输入，然后继续进行比对，一直到比对成功才结束这个程序。
当然，如果正规表示式只能作这么简单的比对，那就真的太无趣了。而且如果他的功能这么阳春，也实在称不上是 Perl 的强力工具。还记得我们提过的量词吗？他可以让我们的比对样式变得更有弹性，现在我们可以用最简单的量词来重新描述我们的样式。我们继续使用刚刚的例子来看看：

```
my $answer = "mo*r";                    # 使用量词
while (1) {                             # 所以其实是无限循
环
    if ((my $patten = <STDIN>) =~ /$answer/) {  # 判断
是否比对成功
        print "*match*\n";
    } else {
        print "*not match*\n";
    }
};

我们试着来执行看看

[hcchien@Apple]% perl ch3.pl
mor
```

```
*match*
mooor
*match*
moor
*match*
mar
*not match*
mur
*not match*
muur
*not match*
```

在这里，我们用了这一次的量词来进行比对。也就是"*"这个比对的量词，它代表零次以上的任何次数，在这里因为他接在字母"o"的后面，也就表示了"o"这个字符出现零次以上次数都符合我们所想要的样式。所以我们看到前面几次的比对都是比对成功即使我们只有输入"mr"这个字符串，但是因为这个字符串中，"m"跟"r"之间，"o"总共出现了零次，因此对 Perl 而言，这也算是比对成功的。不过至少我们可以开始更有弹性的使用比对的样式了，可是该怎么要求 Perl 能够最少比对一个"o"呢？在正规表示式中，'+'就表示至少出现一次，所以这时候我们就可以把"*"换成"+"符号。也就是说，我们如果以刚刚的例子来看，当我们把比对样式改成"mo+r"，原来可以成功比对的"mr"就不再成立了。
既然可以要求某个字符出现 0 次或 1 次，那么如果我希望"o"至少出现二次，或其它更多的次数，有没有办法可以做到呢？答案也是肯定的，我们可以使用另一种方式来表示所需要的量词数目，也就是说可以让你限定次数的量词，而它的表示方式会像这个样子：

```
{min, max}
```

让我们还是继续以刚刚的例子来看，如果你希望掌握"o"出现的次数在某个区间内，那你就可以用这样的方式。让我们来改写一下刚刚的程序变成这样：

```
my $answer = "mo{2,4}r";  # 新的比对样式
while (1) {
    if ((my $patten = <STDIN>) =~ /$answer/) {
        print "*match*\n";
    } else {
        print "*not match*\n";
    }
};
```

我们试着执行看看：

```
[hcchien@Apple]% perl ch3.pl
mor
*not match*
mooor
*match*
mr
*not match*
moor
*match*
```

很显然的，比对样式和刚刚有了明显的变化。我们利用 o{2,4}来限制了"o"只能出现两次至四次，所以只要"o"出现的次数少于两次或大于四次，我们都无法接受。而从执行的结果来看，Perl 也符合我们的期待，因为当我们输入"mor"或"mooooor"时，Perl 都传回比对失败的讯息。不过如果"m"跟"r"中间能够比对到二到四次的"o"，那也就成功的比对了我们的样式。
我们当然可能只需要设定某一边的限制，例如我也许只要求某个字符出现三次以上，至于最多可能出现多少次我并不在意。这时候我们可以用这样的样式：mo{3,}r。很显然，我们也可以这么写：mo{,8}r，这也就是表示我们并不限制"o"出现的最少次数，即使没出现也可以，可是最多却不能能出现超过八次。
另外，我们刚刚都一直在讨论某个位使用量词的比对，可是我们还希望能同时对某个字符串使用量词进行比对。就像这样的字符串"wowwow"，他也可能是"wow"或是"wowwowwow"。那么我们应该怎么来使用量词呢？这时候，我们就需要定义某个群组了，而在正规表示式中，我们可以利用小括号()来把我们想要进行一次比对的字符串全部拉进来，成为一个群组。所以如果我们希望比对出现一次以上的"wow"字符串，那么我们应该这么写：

```
my $answer = "(wow)+";   # 新的比对样式
while (1) {
    if ((my $patten = <STDIN>) =~ /$answer/) {
        print "*match*\n";
    } else {
        print "*not match*\n";
    }
};
```

没错，当我们定义了群组(wow)之后，接下来 Perl 的比对每次都会以(wow)这个字符串为主，也就是必须这个字符串同时出现才算是比对成功。当然，你还是可

以利用群组比对作限定量词的方式，只要把刚刚的比对样式改成(wow){2,4}，那么跟比对单一字符是一样的方式。Perl 还是会比对"wow"这个字符串是不是出现二到四次之间，就像我们比对单一字符的状况一样。
我们好像讲了不少关于 Perl 正规表示式的技巧，不过这只是一小部份，其实关于正规表示式中还有许多技巧可以善加利用的。不过我们把这些留在下一章再来讨论，这时候也许是该喝杯茶休息一下了。

习题：
1. 让使用者输入字符串，并且比对是否有 Perl 字样，然后印出比对结果。
2. 比对当使用者输入的字符串包含 foo 两次以上时(foofoo 或是 foofoofoo 或是 ...)，印出比对成功字样。

# 8. 更多关于正规表示式

正规表示式确实能够完成很多字符串比对的工作，可是当然也需要花更多的时间去熟悉这个高深的学问。如果你从来没有用过正规表示式，你可以在学Perl时学会用Perl，然后在很多其它Unix环境下的应用程序里面使用。当然，如果你曾经用过正规表示式，那么可以在这里看到一些更有趣的用法。我们在上一章已经介绍了正规表示式的一些基本概念，千万别忘记，那些只是正规表示式最基本的部份，因为Perl能够妥善的处理字符串几乎就是仰赖正规表示式的强大功能。所以我们要来介绍更多关于正规表示式的用法。

## 8.1 只取一瓢饮

当你真正使用了正规表示式去进行字符串比对的时候，你会发现，有时候会有可选择性的比对。比如我希望找「计算机」或「信息」这两个词是否在一篇文章里，也就是只要「计算机」或「信息」中任何一个词出现在文章里都算是比对成功，那么我们就应该使用管线符号/|/来表式。所以我们的样式应该试写成这样：/计算机|信息/。
还有可能，你会想要找某个字符串中部份相等的比对，就像这样：

```
/f(oo|ee)t/          # 找 foot 或 feet
/it (is|was) a good choice/   # 在句子中用不同的字
/on (March|April|May)/     # 显然也可以多个选择
```

## 8.2 比对的字符集合

在Perl中的所有的命名规则都必须以字母或底线作为第一个字符，那么我们如果要以正规表示式来描述这样的规则应该怎么作呢？你总不希望你的样式表达写成这个样子吧？

(/a|b|c|d|.....|z|A|B|C|D|.....|_|)

这样的写法也确实是太过壮观了一些。那么我们应该怎么减少自己跟其它可能看到这支程序的程序设计师在维护时的负担呢？Perl提供了一种不错的方式，也就是以「集合」的方式来表达上面的那个概念。因此刚刚的写法以集合的方式来表达就可以写成这样：
[a-zA-Z_]
很显然的，有些时候我们希望比对的字符是属于数字，那么就可以用[0-9]的方式。如果有需要，你也可以这么写[13579]来表示希望比对的是小于 10 的奇数。有时候你会遇到一个问题，你希望比对的字符也许是各种标点，也就是你在键盘上看到，躲在数字上缘的那一堆字符，所以你想要写成这样的集合：
[!@#$%^&*()_+-=]
可是这时候问题就出现了，我们刚刚使用了连字号(-)来取得a-z，A-Z的各个字符，可是这里有一个[+-=]会变成什么样子呢？这恐怕会产生出让人意想不到的结

果。所以我们为了避免这种状况，必须跳脱这个特殊字符，所以如果你真的希望把连字号放进你的字符集合的话，就必须使用(\-)的方式，所以刚刚的字符集合应该要写成：
[!@#$%^&*()_+\-=]
另外，在字符集合还有一个特殊字符^，这被称为排除字符。不过他的效用只在集合的开始，例如像是这样：
[^24680]
这就表示比对 24680 以外的字符才算符合。

## 8.3 正规表示式的特别字符

就像我们在介绍正规表示式的概念的时候所说的，Perl是逐字符在处理样式比对的。可是对于某一些字符，我们却很难使用一般键盘上的按键去表达这些字符。所以我们就需要一些特殊字符的符号。这些就是Perl在处理正规表示式时常用的一些特殊字符：

\s：很多时候，我们回看到要比对的字符串中有一些空白，可是很难分辨他们到底是空格，跳格符号或甚至是换行符号 (注一)，这时候我们可以用\s 来对这些字符进行比对。而且\s 对于空白符号的比对掌握非常的高，以处理(\n\t\f\r)这五种字符。除了原来的空格键，以及我们所提过的跳格字符(\t)，换行字符(\n)外，\s 还会表示回行首的\r 跟换页字符\f。
\S：在大部份的时候，正规表示式特殊字符的大小写总是表示相反的意思，例如我们使用\s 来表示上面所说的五种空格符，那么\S 也就是排除以上五种字符。
\w：这个特殊字符就等同于[a-zA-Z]的字符集合，例如你可以比对长度为 3 到 10 的英文单字，那就要写成：\w{3,10}，同样的，你就可以比对英文字母或英文单字了。
\W：同样的，如果你不希望看到任何在英文字母范围里的字符，不妨就用这个方式避开。
\d：这个特殊的字符就是字符集合[0-9]的缩写。
\D：其实你也可以写成[^0-9]，如果你不觉得麻烦的话。

这些缩写符号也可以放在中括号括住的集合内，例如你可以写成这样：[\d\w_]，这就表示字母，数字或底线都可以被接受。而且看起来显然比起[a-zA-Z0-9_]舒服多了。
另外，你也可以这么写[\d\D]，这表示数字或不是数字，所以就是所有字符，不过既然要全部字符，那就不如用"."来表示了。

## 8.4 一些修饰字符

现在是不是越来越进入状况了呢？我们已经可以使用一般的比对样式来对需要的字符串进行比较了。于是我们拿到了一篇文字，就像这样：

```
I use perl and I like perl. I am a Perl Monger.
```

我们现在希望找出里面关于 Perl 的字符串，这应该相当简单，所以我们把这串文字定义为字符串$content。然后只要用这样的样式来比对：

$content =~ /perl/;

不过好像不太对劲，或许我们应该改写成这样：

$content =~ /Perl/;

可是万一我们打算从档案里面取出一篇文字，然后去比对某个字符串，这时候我们不知道自己会遇到的是 Perl 或 perl。既然如此，我们可以用字符集合来表示，就像我们之前说过的样子：

$content =~ /[pP]erl/;

可是我要怎么确定不会写成 PERL 呢？其实你可以考虑忽略大小写的比对方式，所以你只要这样表示：

$content =~ /perl/i;

其中的修饰字符 i 就是告诉 Perl，你希望这次的比对可以忽略大小写，也就是不管大小写都算是比对成功。所以你有可能比对到 Perl，perl，PERL。当然也可能有 pErL 这种奇怪的字符串，不过有时候你会相信没人会写出这样的东西在自己的文章里。
Perl 在进行比对的修饰字符，除了/i 之外，我们还有/s 可用。我们刚刚稍微提到了可以使用万用字符点号(.)来进行比对，可是使用万用字符却有一个问题，也就是如果我们拿到的字符串不在同一行内，万用字符串是没办法自动帮我们跨行比对，就像这样：

```
my $content = "I like perl. \n I am a perl monger.
\n";
if ($content =~ /like.*monger/) {
    print "*$1*\n";
}
```

我们想要找到 like 到 monger 中间的所有字符，可是因为中间多了换行符号(\n)，所以 Perl 并不会找到我们真正需要的东西。这时候我们就可以动用/s 来要求 Perl 进行跨行的比对。因此我们只要改写原来的样式为：

$content =~ /like.*monger/s

那么就可以成功的进行比对了。可是如果有人还是喜欢用 Perl Monger 或是 PERL MONGER 来表达呢？我们当然还是可以同时利用忽略大小写的修饰字符，因此我们再度重写整个比对样式：

$content =~ /like.*monger/is

这两个修饰字符对于比对确实非常有用。

## 8.5 取得比对的结果

虽然样式比对的成功与否对我们非常有用，可是很多时候我们并无法满足于这样的用法。尤其当我们使用了一些量词，或修饰字符之后，我们还会希望知道自己到底得到了什么样的字符串。就以刚刚的例子来看，我的比对样式是表示从like开始，到monger结束，中间可以有随便任何字符。可是我要怎么知道我到底拿到了什么呢？这时候我就需要取得比对的结果了。
Perl有预设变量来让你取得比对的结果，就是以钱号跟数字的结合来表示，就像这样：($1，$2，＄3....)。
而用法也相当简单，你只要把需要放入预设变量的比对结果以小括号刮起括就可以了，就以我们刚刚的例子来看，你只要改写比对样式，就像这样：

```
my $content = "I like perl. \n I am a perl monger.
\n";
if ($content =~ /(like.*monger)/s) {
   print "$1\n";
}
```

这里的$1 就是表示第一个括号括住的的比对结果。所以 Perl 会送出这样的结果：

```
[hcchien@Apple]% perl ch3.pl
like perl.
 I am a perl monger
```

当然，预设的比对变量也是可以一次撷取多个比对结果，就像下面的例子：

```
my $content = "I like perl. \n I am a perl monger. \n";
```

```
if ($content =~ /(perl)\s(monger)/s) {      # $1 =
"perl", $2 = "monger"
   print "$1\n";                        # 印出 perl
}
```

不过我们如果再把这个小程序改写成这样呢？

```
my $content = "I like perl. \n I am a perl monger. \n";
if ($content =~ /((perl)\s(monger))/s) {
   print "$1\n$2\n$3\n";
}
```

结果非常有趣：

```
[hcchien@Apple]% perl ch3.pl
perl monger
perl
monger
```

看出来了吗？我们用括号拿到三个比对变量，而 Perl 分配变量的方式则是根据左括号的位置来进行。因此最左边的括号是整个比对结果，也就是"perl monger"，接下来是"perl"，最后才是"monger"。相当有趣，也相当实用。
不过在使用这些暂存变量有一些必须注意的部份，那就是这些变量的生命周期。因为这些变量会被放在内存中，直到下次比对成功，要注意，是比对成功。所以如果你的程序是这么写的话：

```
my $content = "Taipei Perl Monger";
$content =~ /(Monger$)/;    # $1 现在是 Monger
print $1;
$content = /(perl)/;        # 比对失败
print $1;                  # 所以还是印出 Monger
```

当你第一次成功比对之后，Perl 会把你所需要的结果放如暂存变量$1 中，所以你第一次打印$1 时就会看到 Perl 印出 Monger，于是我们继续进行下一次的比对，这次我们希望比对 perl 这个字符串，并且把比对要的字符串同样的放入$1 之中。可惜我们的字符串中，并没有 perl 这个字符串，而且我们也没有加上修饰符号去进行忽略大小写的比对，因此这次的比对是失败的，可是 Perl 并不会先清空

暂存变量$1，因此变量的内容还是我们之前所比对成功的结果，也就是 Monger，这从最后印出来的时候就可以看出来了。
比较容易的解决方式就是利用判断式去根据比对的成功与否决定是否打印，就像这样：

```
my $content = "Taipei Perl Monger";
print $1 if ($content =~ /(Monger$)/);    # 因为比对成功，所以会印出 Monger
print $1 if ($content = /(perl)/);        # 这里就不会印出任何结果了
```

## 8.6 定位点

要能够精确的描述正规表示式，还有一项非常重要的工具，就是定位点。其中你可以指定某个样式必须要被放在句首或是句尾，比如你希望比对某个字符串一开始就是"Perl"这个字符串。那么你可以把你的样式这样表示：

/^Perl/

其中的^就是表示字符串开始的位置，也就是只有在开始的位置比对到这个字符串才算成功。 当然，你可以可以使用$来表示字符串结束的位置。以这个例子来看：

```
my $content = "Taipei Perl Monger";
if ($content =~ /Monger$/s) {          # 以定位字符进行比对
   print "*Match*";                    # 在这里可以成功比对
}
```

## 8.7 比对与替换

就像很多编辑器的功能，我们不只希望可以找到某个字符串，还希望可以进行替换的功能。当然正规表示式也有提供类似的功能，甚至更为强大。不过其实整个基础还是基于比对的原则。也就是必须先比对成功之后才能开始进行替换，所以只要你能了解整个Perl正规表示式的比对原理，接下来要置换就显得容易多了。现在我们先来看一下在Perl的正规表示式中该怎么描述正规表示式中的替换。

我们可以使用s///来表示替换，其中第一个部份表示比对的字符串，第二个部份则是要进行替换的部份。还是举个例子来看会清楚一些：

```
my $content = "I love Java";
print $content if ($content =~ s/Java/Perl/);  # 假如置换成功，则印出替换过的字符串
```

当然，就像我们所说的，置换工作的先决条件是必须完成比对的动作之后才能进行，因此如果我们把刚刚的程序改写成

```
my $content = "I love Java";
print $content if ($content =~ s/java/perl/);
```

那就什么事情也不会发生了。当你重新检查字符串$content 时，就会发现正如我们所预料的，Perl 并没有对字符串进行任何更动。
不过有时候我们会有一些问题，就像这个例子：

```
my $content = "水果对我们很有帮助，所以应该多吃水果";
print $content if ($content =~ s/水果/零食/);  # 把水果用零食置换
```

看起来好像很容易，我们把零食取代水果，可是当结果出来时，我们发现了一个问题。Perl 的输出是：「零食对我们很有帮助，所以应该多吃水果」。当然，这跟我们的期待是不同的，因为我们实在想吃零食啊。可是 Perl 只说了零食对我们有帮助，我们还是得吃水果。
没错，我们注意到了，Perl 只替换了一次，因为当第一次比对成功之后，Perl 就接收到比对成功的讯息，于是就把字符串依照我们的想法置换过，接着....收工。好吧，那我们要怎么让 Perl 把整个字符串的所有的「水果」都换成「零食」呢？我们可以加上/g 这个修饰字符，这是表示全部置换的意思。所以现在应该会是这个样子：

```
my $content = "水果对我们很有帮助，所以应该多吃水果";
print $content if ($content =~ s/水果/零食/g);  # 把水果全部换成零食吧
```

就像我们在比对时用的修饰字符，我们在这里也可以把那些修饰字符再拿出来使用。就像这样：

```
my $content = "I love Perl. I am a perl monger";
print $content if ($content =~ s/perl/Perl/gi);
```

我们希望不管大小写，所有字符串中的 Perl 一律改为 Perl，所以就可以在样式的最后面加上/gi 两个修饰字符。而且使用的方式和在进行比对时是相同的方式。

## 8.8 有趣的字符串内交换

这是个有趣的运用，而且使用的机会也相当的多，那就是字符串内的交换。这样听起来非常难以理解，举个例子来看看。
我们有一个字符串，就像这样：

$string = "门是开着的，灯是关着的"

看起来真是平淡无奇的一个句子。可是如果我们希望让门关起来，并且打开灯，我们应该怎么作呢？
根据我们刚刚学到的替换，这件事情好像很简单，我们只要把门跟灯互相对调就好，可是应该怎么作呢？如果我们这么写：

$string =~ s/门/灯/;

那整个字符串就变成了「灯是开着的，灯是关着的」，那接下来我们要怎么让原来「灯」的位置变成「门」呢？所以这种作法似乎行不通，不过既然要交换这两个字，我们是不是有容易的方法呢？利用暂存变量似乎是个可行的方法，就像这样：

```
my $string = "门是开着的，灯是关着的";

print $string if ($string =~
s/(门)(.*)(灯)(.*)/$3$2$1$4/);
```

看起来好像有点复杂，不过却非常单纯，我们只要注意正规表示式里面的内容就可以了。在样式表示里面，非常简单，我们要找门，然后接着是「门」和「灯」中间的那一串文字，紧接在后面的就是「灯」，最后的就全部归在一起。按照这样分好之后，我们希望如果 Perl 比对成功，就把每一个部份放在一个暂存变数

中。接下来就是进行替换的动作，我们把代表「门」跟「灯」的暂存变量$1 及$3 进行交换，其余的部份则维持不变。我们可以看到执行之后的结果就像我们所期待的一样。
当然，这样只是最简单的交换，如果没有正规表示式，那真的会非常的复杂，不过现在我们还可以作更复杂的交换动作。

## 8.9 不贪多比对

其实在很多状况下，我们常常不能预期会比对什么样的内容，就像我们常常会从网络上抓一些数据回来进行比对，这时候我们也许有一些关键的比对样式，但是大多数的内容却是未知的。因此比对的万用字符(.)会经常被使用，可是一但使用了万用字符，就要小心Perl会一路比对下去，一直到不合乎要求为止，就像这样：

```
<table>
    <tr><td>first</td></tr>
    <tr><td>second</td></tr>
    <tr><td>third</td></tr>
</table>
```

这是非常常见的 HTML 语法，假设我们希望找到其中的三个元素，所以就必须过滤掉那些 HTML 标签。如果你没注意，也许会写成：

```
my $string =
"<table><tr><td>first</td></tr><tr><td>second
</td></tr><tr><td>
third </td></tr></table>";

if ($string =~ m|<tr><td>(.+)<\/td><\/tr>|) {
    print "$1";
}
```

可是当你看到执行结果时可能会发现那并不是你要的结果，因为程序印出的$1 居然是：

first</td></tr><tr><td>second</td></tr><tr><td>third

让我们来检查一下程序出了什么问题。我们的比对样式中告诉 Perl，从<tr><td>开始比对，然后比对所有字符，一直到遇到</td></tr>时结束。 而且 Perl 也很符合我们的期望，他找到了符合我们需求的最大集合。这就是重点了，Perl 预设会去找到符合需求的最大集合。因此在这里他就取得了比对结果

"first</td></tr><tr><td>second</td></tr><tr><td>third"。可是我们要的却是「从<tr><td>开始，遇到</td></tr>就结束」，而不是「找出字符串中<tr><td>到</td></tr>的最大字符串」。
可是我们刚刚的比对样式中并没有告诉 Perl：「遇到</td></tr>就停下来」，所以他会一直比对到字符串结束，然后找出符合样式的最大字符串，这就是所谓的贪多比对。相对于此，我们就应该告诉 Perl，请他以不贪多的方式进行比对。所以我们就在比对的量词后面加上问号(?)来表示不贪多，并且改写刚刚的比对样式：

```
$string =~ m|<table><tr><td>(.+?)<\/td><\/tr>|
```

如此一来，就符合我们的要求了。

## 8.10 如果你有迭字

在正规表示式中，有一种比对的技巧称为回溯参照 (backreference)。我们如果可以用个好玩的例子来玩玩回溯参照也是不错的，比如我们有个常见的句子：「庭院深深深几许」。如果我希望比对中间三个深，我可以怎么作呢？当然，直接把「深深深」当作比对的样式是个方法，不过显而易见的，这绝对不是个好方法。至少你总不希望看到有人把程序写成这样吧：

```
my $string = "庭院深深深几许";
print $string if ($string =~ /深深深/);
 # 这样写程序好像真的很糟
```

这时候回溯参照就是一个很好玩的东西，我们先把刚刚的程序改成这样试试：

```
my $string = "庭院深深深几许";
print $string if ($string =~ /(深)\1\1/);
```

你应该发现了，我们把「深」这个字先放到暂存变量中，然后告诉 Perl：如果有东西长的跟我们要比对的那个变量里的东西一样的话，那么就用来继续比对吧。可是这时候你却不能使用暂存变量$1，因为暂存变量是在比对完成之后才会被指定的，而回溯参照则是在比对的期间发生的状况。刚刚那个例子虽然可以看出回溯参照的用途，可是要了解他的有用之处，我们似乎该来看看其它的例子：

```
```

```
my $string = "/Chinese/中文/";
if ($string =~ m|([\/\|'"])(.*?)\1(.*?)\1|)
{       # 这时候我们有一堆字符集合
    print "我们希望用 $3 来替换 $2 \n";
}
```

看出有趣的地方了吗？我们在字符集合里面用了一堆符号，因为不论在字符集合里的那一个符号都可以算是正确比对。但是我们在后面却不能照旧的使用[\/\|'"]来进行比对，为什么呢？你不妨实验一下这个例子：

```
my $string = "/Chinese|中文'";
if ($string =~
m|([\/\|'"])(.*?)[\/\|'"](.*?)[\/\|'"]|) {
    print "我们希望用 $3 来替换 $2 \n";
}
```

很幸运的，我们在这里还是比对成功。为什么呢？我们来检查一下这次的比对过程：首先我们有一个字符集合，其中包括了/|'"四种字符，而这次我们的字符串中一开始就出现了/字符，正好符合我们的需求。接下来我们要拿下其它所有的字符，一直到另一个相同的字符集合，不过我们这次拿到了却是|字符，最后我们拿到了一个单引号(')。显然不单是方便，因为没有使用回溯参照的状况下，我们拿到了错误的结果。
那我们回过头来检查上一个例子就会清楚许多了，我们一开始还是一个字符集合，而且我们也比对到了/字符。接下来我们要找到跟刚刚比对到相同的内容 (也就是要找到下一个/)，然后还要再找最后一次完全相同的比对内容。我们经常会遇到单引号(')或双引号(")必须成对出现，而利用回溯参照就可以很容易的达成这样的要求。

## 8.11 比对样式群组

我们刚刚说了关于回溯参照的用法，不过如果我们的比对并没有那么复杂，是不是也有简单的方式来进行呢？我们都知道很多人喜欢用blahblah来进行没有什么意义的留言，于是我们想把这些东西删除，可是他们可能是写"blahblah"或是"blahblahblah"等等。这时候使用回溯参照可能会写成这样：

```
my $string = "blahblahblah means nothing";

if ($string =~ s/(blah)\1*//) {
    print "$string";
}
```

当然这样的写法并没有错，只是好像看起来比较碍眼罢了，因为我们其实可以用更简单的方法来表达我们想要的东西，那就是比对样式群组。这是小括号(())的另外一个用途，所以我们只要把刚刚的比对样式改成这样：/(blah)+/就可以了。这样一来，Perl 就会每次比对(blah)这个群组，然后找寻合乎要求的群组，而不是单一字符(除非你想把某一个字符当群组，只是我们并不觉得这样的方式会有特殊的需求)。而当我们设定好某个群组之后，他的操作方式就跟平常在写比对样式没什么两样了，我们就可以利用/(blah)+/找出(blah)这个群组出现超过一次的字符串。如果你觉得不过瘾，/(blah){4,6}/来确定只有 blah 出现四到六次才算比对成功也是可以的。

## 8.12 比对样式的控制

一开始使用正规表示式的人总有一个疑问，为什么要写出正确比对的样式这么不容易。而比对错误的主要原因通常在于得到不必要的数据，也就是比对样式符合了过多的文字，当然，还有可能是比对了不被我们期待的文字。就像我们有这样的一堆字符串：

```
I am a perl monger
I am a perl killer
it is so popular.
```

如果你的比对样式是/p.*r/，那么你会比对成功：

```
perl monger
perl killer
popular
```

可是这跟我们的需求好像差距太大，于是你希望用这样的样式来进行比对：/p\w+\s\w+r/，那你也还会得到

```
perl monger
perl killer
```

这两种结果。所以怎么在所取得的信息中写出最能够精确比对的样式确实是非常重要，也需要一些经验的。

习题：

1. 延续第七章的第一题，比对出 perl 在字符串结尾的成功结果。
2. 继续比对使用者输入的字符串，并且确定是否有输入数字。
3. 利用回溯参照，找出使用者输入中，引号内(双引号或单引号)的字符串。
4. 找出使用者输入的第一个由 p 开头，l 结尾的英文字。

注一：有时候因为操作系统的不同，换行符号并不会被忠实的呈现。

# 9. 再谈控制结构

程序中的控制结构是用来控制程序进行方向的重要依据，所以有足够灵活的程控结构能节省程序设计师的大量时间。可是也必须谨慎的使用，否则如果一个程序里面到处充满了循环控制，然后中断，转向，反而让程序变得非常没有结构，在程序的结构遭到破坏之后，一旦程序出了问题而需要要开始追踪整个程序的行进就变成要花大量的时间。很显然的，如果没有保持程序良好的结构性，对于日后的维护将会是一大负担。

不过如果能够小心使用，这些控制程序流程的工具会是程序设计师重要的工具。所以我们就来看看除了之前提过的for，while，until，if等等各式各样的流程控制之外，我们还能有什么其它的方式可以方便的操控Perl吧。

## 9.1 循环操作

既然是流程控制，我们有时候会因为程序的状况而希望离开循环或某些条件叙述的判断区块，也可能需要省略循环中的某次运算，或进行其它的跳跃。听起来这些方式好像很容易让人搞的眼花缭乱，现在我们就要来看看这些功能到底有什么帮助。

### 9.1.1 last

顾名思义，这也就是最后的意思，因此Perl看到这个关键词就会相当高兴，表示他距离休息又靠近了一步。不过这样可以提前结束循环的函数到底扮演着什么样的角色呢？让我们来试试：

```
for (1...10) {
    last if ($_ == 8);
    print;                  # 这样会印出  1...7
}
```

好玩吧，当你在循环内加上了另外一个判断式，并且允许循环在某个条件下跳出循环的执行，这时候，Perl 就提早下班，回家休息了。当然，last 的使用是有限制的，也就是他只允许在可以执行结束的区块内，其中最常被应用的是在循环内，而这里的循环指的是 for，foreach，while，until，另外也可以在单独的区块中使用，就像这样：

```
print "start\n";
{
    print "last 前执行\n";          # 会顺利印出
    last;
```

```
    print "这里就不执行了\n";        # 所以这一行永
远不会被执行
}
print "然后就结束了\n";
```

没错，这个例子的程序确实非常无趣，因为我们写了一行永远不会执行的程序，不过却让我们透过这个小程序清楚的看到 last 的执行过程。所以我们可以轻易的让 Perl 跳出某个区块，而既然可以提前结束区块的执行范围，我们有时候也需要 Perl 可以重复执行循环内的某些条件这时候 redo 这个函式就派上用场了。

## 9.1.2 redo

虽然我们可以用last让Perl提早下班，同样的，我们也可以要求Perl加班，也就是利用redo这个函式让Perl重新执行循环中的某些条件。例如我们在循环中可以利用另一个判断叙述来决定目前的状况，并且利用这个额外的叙述判断来决定是否要让循环中的某个条件重复执行。例如我们有一个循环，我们可以很容易的要求perl在循环中的某个阶段重复执行一次，就像下面的例子：

```
for (1...10) {
    $_++;                    # redo 其实会来这里
    redo if ($_ == 8);             # 我们希望 redo
的条件
    print;                         # 会印出 2, 3,
4, 5, 6, 7, 9, 9, 10, 11
}
```

我们可以研究一下 redo 的过程，在上面那个程序的第三行，我们要求 Perl 在循环的变量等于 8 的时候就执行 redo。所以当我们在循环内的条件符合 redo 的要求时，Perl 就会跳到循环的第一行，也就是说当循环的值进行到 7 的时候，经过$_++的运算就会让$_变成 8，这时候就符合了 redo 的条件，因此还来不及印出来变量$_，Perl 就被要求回到循环中的第一行，于是$_变成了 9，这就是第一次 9 的出现。接下来循环恢复正常，就接连印出 9…11。也就是我们看到的结果了。
不过这里要注意的是我们循环的使用方式，我们使用了 for(1…10)，而不是使用 for ($_ = 1; $_ <= 10; $_++)这样的叙述，而这两者有着相当的差异。如果各位使用了后者的循环表示，结果就会有所不同。
我们还是实际上来看看使用 for (;;)来检查 redo 的效果时，也可以藉此看看两者的差异了：

```
for ($_ = 1; $_ <= 10; $_++) {
```

```
    $_++;                    # 我们还是先把得到的元
素进行累加
    redo if ($_ == 8);       # 遇到 8 的时候就
重复一次
    print $_;                # 印出目前的
$_，我们得到 2, 4, 6, 9, 11
}
```

很有趣吧，我们来看看这两者有什么不同，首先我们看到第一个例子中，Perl 是拿出串行 1...10 的元素，并且把得到的元素放进变量$_中。接下来就像我们在循环中所看到的样子了，所以循环并不是以$_作为计数的依据。这样的方式就像这样的写法：

```
for (my @array = (1...10))
```

可是当我们看到第二个例子的时候，我们却是指定了$_作为循环的计数标准。所以我们在循环中对$_进行累加，就完全影响到循环的执行。因此我们一开始拿到$_等于 1，可是一进循环就马上又被累加了一次，我们就印出了 2，接着 Perl 又执行循环的递增，让我们取得 3，我们自己又累加了一次，也就印出 4，等我们累加到 8 的时候，循环被要求执行 redo，因此我们又累加一次，$_变成 9，紧接着最后一次的循环，经过累加之后，我们印出了 11。看起来好像非常复杂，不过你只要实际跟着循环跑一次应该就可以看出其中的变化了。
不过在使用 redo 的时候必须非常小心，因为你很可能因为设定了 redo 的条件而产生无穷循环。就像刚刚的例子，如果我们改写成：

```
for (1...10) {
    redo if ($_ == 8);
    print $_;
}
```

在这个循环里，我们希望循环的控制变量在 8 的时候可以进行 redo，于是它就一直卡在 8 而跳不出来了。就像我们说的，这里变成了无穷循环，你的程序也就有加不完的班了。

## 9.1.3 next

我们刚刚使用了last来结束某个区块，也透过redo来重复执行循环中的某个条件叙述，那么既然可以在循环内重复执行某个条件的叙述，那么略过某个条件下的叙述也应该不是太难的问题。是的，其实只要利用next，那么我们可以在某些情

况下直接结束这次的执行，也就是省略循环中某一些状况的执行。当然，描述还是不如直接看看实例，我们还是利用简单的例子来了解next的作用：

```
for (1...10) {
    next if ($_%2);                # 以串行值除以 2 的余数判断
    print $_;
}
```

这个例子里面，我们会印出 1...10 之间的所有偶数。首先，这是一个从 1 到 10 的循环，主要的工作在于印出目前循环进行到的值，不过就在打印之前，我们使用了一个 next 函式，而决定是否执行 next 的判断是以串行值除以 2 的余数来作为条件
。如果余数为真(在这里的解释就是：如果余数为 1)，就直接结束这次的执行，当然，如果余数为 0(在这个程序中，我们可以解释为「遇到偶数时」)，就会印出串行的值，所以程序会印从 1 到 10 的所有偶数值。
虽然有些时候，你会发现 next 的好用之处，可是如果你会因为 next 而造成追踪程序的困扰时，那就可能要修改一下你的使用方式了。例如改变循环的判断条件或是索引的递增方式等等，就像上面的例子，我们也许可以改用 while 来判断，或者使用 for(;;)，而不是使用 foreach 加上 next 来增加程序的复杂性，不过这些都必须依赖经验来达成。

### 9.1.4 标签

标签的作用主要就是让Perl知道他该跳到哪里去，这样的写法并不太常被使用，主要是因为对于程序的结构会有一定程度的破坏，因为你可以任意的设置一个卷标位置，然后要求Perl跳到卷标的位置，当然，他确实有一些使用上的要求，而不是完全漫不限制的随便下一个标签就让Perl转换执行的位置，至少这并不是goto在做的工作。不过撇开这个暂且不谈，我们先来看看怎么使用标签。下面的例子应该可以让大家能够看出轮廓：

```
LABEL: for my $outter (1...5) {
    for (1...10) {
  if ($_ > 2) { next LABEL; } else { print "inner $_ \n"; }
    }
    next LABEL if ($outter%2);
    print $_;
}
```

当我们有时候单单利用 next 或 last 无法逃离循环到正确的地方时时，使用标签就能够帮助我们找到出路。就像我们的例子中，我们一共有两个 for 循环，两个 if 判断，我们要怎么让 Perl 不会在里面迷路呢？这时候标签的使用就很方便了，就像我们在内部的 for 循环中根据得到的值来决定是否要跳出上一层的 for 循环。可是使用标签时有一个特别需要注意的部份，就是标签的使用并非针对程序中的某一个点，而必须是一个循环或是区块。否则整个标签的使用就会太过混乱，你会发现要检查程序的错误变成了「不可能的任务」。当然，如果你在你的循环中插了大量的标签也会让其它人非常困扰，因为就算是 Perl 可以处理这样的标签，只怕你自己也会搞的头晕。这又是写程序时的风格问题了。
标签可以配合我们之前所提的几种控制指令来运用，因此你可以要求使用 next，redo，last 加上标签来标明循环的方向。就像上面的例子，我们先在第一行的地方加上标签'LABEL'，表明接下来如果需要，要求 Perl 直接来这里。接下来我们用了一个 foreach 循环，其中的值是从 1 到 10。可是在这个循环中，我们又使用了 next，要求如果变量$_大于 2 就执行 next，而且是跳到卷标 LABEL 的位置。也就是说，他除了跳过里面的循环之外，也会跳出外层循环的其它叙述。所以当内层循环的$_变量大于 2 的时候，程序中最后面的两行叙述都不会被执行。当然，大家应该还是想要知道这样的程序会产生出什么样的结果：

```
inner 1
inner 2
inner 1
inner 2
inner 1
inner 2
inner 1
inner 2
inner 1
inner 2
```

你应该发现了，程序一直都只执行了一部份，因为当内圈的变量$_大于 2 的时候，Perl 就急着要回去 LABEL 的地方，所以就连里面的循环都没办法完整执行，外面的循环更是被直接略过，这样应该就很容易理解了。

## 9.2 switch

如果你用过其它程序语言，例如C或Java，你现在也许会很好奇，为什么我们到目前为止还没有提到Switch这个重要的流程控制函式，主要是因为Perl在最初的设计是没有放入Switch的。其实很多人对于Perl没有提供switch都觉得非常不可思议，不过Larry Wall显然有他的理由，至于这些历史原因，我们也没必要在这里讨论。
好吧，我听到一阵哗然，为什么Perl没有这个可以为程序画上彩妆的工具呢？其

实我个人也觉得Switch用来进行各种条件判断的流程控制确实是非常方便，而且会让程序看起来相当整齐，不过大部份的时候，你有什么流程控制非得需要Switch才能完成呢？因为我们在进行Switch的时候，其实也就是希望表达出许多层的if {} elsif {} elsif {} ..... 。也就是说，if叙述其实已经可以满足我们的需求了，那么Switch就真的是帮助我们取得比较整齐，易读的程序代码。不过在大部份的情况下，你想要用漂亮的程序代码来吸引Perl的黑客(注一)们协助完成一项工作，倒不如告诉他们怎么样可以少打一些字。

### 9.2.1 如果你有复杂的 if 叙述

Switch之所以受到欢迎，当然有过人之处，虽然我们也可以用其它方式达到同样的目的，可是至少对我来说，程序的易读性似乎还是以Switch来得好些，不过这部份可就是见仁见智了。就像我们说的，如果你有一大堆if {} elsif {} elsif {} .... 的叙述时，你的程序看起来也许看起来会像这样：

```
my $day = <STDIN>;
chomp($day);
if ($day eq 'mon') {
   ...
} elsif ($day eq 'tue') {
  ...
} elsif ($day eq 'wed') {
  ...
} elsif ($day eq 'thu') {
  ...
} elsif ($day eq 'fri') {
  ...
}
```

其实这样的程序代码也没什么不妥，可是你也许会觉得这样的写法有点麻烦。当然，对这些人来说，如果可以把上面这段程序代码利用 Switch 写成这样，那好像看起来更让人感觉神清气爽：

```
my $day = <STDIN>;
chomp($day);
swich ($day) {
     case ('mon') { ... }
     case ('tue') { ... }
     case ('wed') { ... }
     case ('thu') { ... }
     case ('fri') { ... }
```

```
}
```

以可读性来讲，使用 Switch 确实比用了一大堆的 if {...} elsif {...} elsif {...} 要好的多，那么我们要怎么样可以使用 Switch 来写我们的程序呢？

## 9.2.2 利用模块来进行

很显然的，还是有许多Perl的程序设计师对于switch的干净利落难以忘怀。因此有人写了perl模块，我们就可以利用这个模块来让我们的程序认识switch。
利用Switch模块，我们就可以写出像上面一样的语法，让你的程序看起来更简洁有力。而且switch的使用上，不单可以比对某个数字或字符串，你还可以使用正规表示式进行复杂的比对来决定程序的进行方向。我们在这里只是告诉大家一些目前已经存在的解决方案，而不应该在这里讲太多关于模块的使用，以免造成大家的负担。
另外，还有部份程序设计师不太喜欢目前Switch的运作，认为破坏了原来Perl在流程控制的结构而也会因此而破坏原来Perl程序的稳定性。因为不管如何，这些意见都是仅供参考。不过既然「办法不只一种」，那么就看个人的接受度如何了。

# 9.3 三元运算符

另外也有一种非常类似 if {...} else {...} 的运算符，我们称为三元运算符。他的写法也就是像这样：

```
my ($a, $b) = (42, 22);
my $max = ($a > $b) ? $a : $b;
print "$max\n";
```

首先我们把串行 (42, 22) 指定给变量 $a 跟 $b，接着我们要找到两个值中较大的一个，于是利用判断式 ($a > $b) 来检查两个数字之间的关系。如果 $a > $b 成立，那么 $max 就是 $a，否则就是 $b。所以很明显的，上面的三元运算符也可以改写成这样：

```
my ($a, $b) = (44. 22);
if ($a > $b) { $max = $a } else { $max = $b }
print "$max\n";
```

以上面两个例子来看，相较之下，三元运算符的方式应该简单许多，只是这样的方式并不够直觉，对于刚开始写 Perl 的人而言可能会有点障碍。不过我们还是必须提醒，这样的写法很可能常常出现在其它的程序里，所以即使你只想依赖 if {...} else {...} 来完成同样的工作，至少你也要知道别人的程序代码中表达的是

什么。
而且，其实利用三元运算符也可以完成不少复杂的工作。例如你可以在判断式的地方用一个副例程，并且根据回传的结果来决定你要的值等等。因此一但有机会，也许你也可以试试。在这里，我们可以再举一个怎么增加便利性的写法的例子：

```
my $return = cal(5);
print "$return\n";

sub cal {
    my $param = shift;
    ($param > 4) ? $param*2 : $param**2;  # 利
用参数来判断回传值的运算方式
}
```

## 9.4 另一个小诀窍

接下来我们来点饭后甜点，也就是 || 算符。其实不只 || 算符，其它的逻辑算符也可以拿来作流程控制的小小螺丝钉。不过首先我自己偏爱使用 || (也是使用机会比较高的)，而且我们只打算来个甜点，这时候显然不适合大餐了。
我们有时候会希望某些变量可以有默认值，例如副例程的参数，或是希望使用者输入的变量等等。所以你当然可以这样写：

```
sub input {
    my $key = shift;
    $key = "默认值" unless ($key);
    print "$key\n";
}
```

这个副例程什么也没作，就只拿了使用者传来的参数，然后印出来。可是我们还可以让他更简单一些，我们把他改成这样：

```
sub input {
    my $key = shift || "默认值";
    print "$key\n";
}
```

这时候，|| 算符被我们拿来当一个判断的工具。我们先确定使用者有没有传入

参数，也就是平常我们所使用的 shift，如果@_中是空数组，那么 $key = shift 就会得到伪值，这时候 || 就会启动，让我们的默认值产生效果。因此我们就得到 $key = "默认值"。
另外，|| 还常常被用来进行意外处理。因为我们必须知道，如果某个表达式失败，那么我们就可以让程序传回错误讯息。就像这样：

```
output() || die "没有回传值";

sub output {
    return 0;
}
```

我们在程序里面呼叫 output 这个副例程，不过因为回传值是 0，于是 || 也发生效用，就让程序中断在这里，并且印出错误讯息。

习题：
1. 陆续算出 (1...1) 的总和，(1...2) 的总和，...到 (1...10) 的总和。但是当得到总和大于 50 时就结束。
2. 把下面的程序转为三元运算符形式：

```
#!/usr/bin/perl -w

use strict;

chomp(my $input = <STDIN>);
if ($input < 60) {
    print "不及格";
} else {
    print "及格";
}
```

注一：其实我们指的就是 hacker，不过现今大多数人都误用 cracker(指潜入或破坏其它人系统者)为 hacker(指对某些领域有特别研究的人)

# 10. Perl的档案存取

档案系统在写程序时是非常重要的一个部份，尤其对于Perl的使用者来说，因为Perl能够处理大量而且复杂的资料，所以经常被拿来作为Unix操作系统的管理工具，尤其对于Unix-like系统管理员而言，在进行系统日志的管理时，存取档案，读取档案内容并加以分析就是最基本的部份。当然，你还可能进行目录的修改，档案权限的维护等等跟系统有密切关系的操作。

## 10.1 档案代号 (FileHandle)

当你的Perl想要透过操作系统进行档案存取时，可以利用档案代号取得和档案间的系结，接下来的操作就是透过这个档案代号和实体的档案间进行沟通。也就是说，我们要进行档案操作时，可以先定义相对应实体档案的代号，以便我们用更简便的方式对档案进行存取。

而所谓的档案代号其实就是由使用者自行命名，并且用来跟实体档案进行连结的名称，他的命名规则还是依循Perl的命名规则，大家对于这个规则应该相当熟悉了，不过我们还是再次提醒一下：可以数字，字母及底线组成，但是不能以数字作为开始。而且一般来说，我们几乎都习惯以全部大写来作为档案代号，因为档案代号并不像其它变量，会使用某些符号作为识别，所以几乎约定成俗的全部大写习惯也是有存在的道理。

当然，你也可以依照自己的习惯来为档案代号命名(注一)，这表示所谓的全部大写绝对不是一种铁律，就像Perl程序语言本身，希望以最少的限制来进行程序设计的工作。

## 10.2 预设的档案代号

对于档案的输出，输入而言，其实就跟平常时候，你利用Perl在进行其它的操作非常接近，有时候只是输出到不同的媒介上。所以Perl其实已经预定了几种档案代号，让你不需要每次写Perl的程序就必须去重新定义这些代号，很显然的，几乎大部份的程序都会需要这些档案代号。

这六个预设的档案代号分别是：STDIN，STDOUT，STDERR，DATA，ARGV，ARGOUT，看起来相当熟悉吧？没错，因为很多时候，我们其实就是靠这些预设的档案代号在进行程序的输出，输入。只是我们还没有了解这些其实就是档案代号。换个角度来看，其实即使我们都不知道他们是预设的档案代号，我们就能运作自如，那么对于档案代号的使用显然就不是太难。不过，我们还是要再来看看这六个Perl预设的档案代号。其中有些我们已经使用过了，我们就先对其中几个预设的档案代号来进行介绍：

STDIN：这也就是我们常看到的「标准输入装置」，当Perl开始执行时，它预设接受的外部信息就是从这里而来。就像我们之前曾经看过的写法：

```
my $input = <STDIN>;                    # 从标准输入装置取得数据
```

```
print $input;
```

这时候，当我们从键盘输入时，Perl 就可以正确的取得信息，并且透过 STDIN 取得使用者用键盘打入的一行字符串。因此他的运作方式就是以一个档案代号来进行。当然，你可以透过系统函式库的配合，让你的标准输入转为其它设备之后你就进行其它运用，不过这显然不是这里的主题，还是让我们言归正传。对于 Perl 来说，他从档案系统读入数据是以行为单位，因此即使是利用 STDIN，Perl 还是会等到使用者键入换行键时才会有所动作。

STDOUT：相对于标准输入，这就是所谓的标准输出，也就是在正常状况下，你希望 Perl 输出的结果就是透过 STDOUT 来进行输出的。而一般来说，我们所使用的就是屏幕输出。你可以看看这个程序里的写法：

```
my $output = "标准输出";

print "$output\n";
print STDOUT "$output\n";
```

没错，就像我们所预期的，Perl 透过屏幕印出了两行一模一样的结果，也就是印了两行「标准输出」。原因非常简单，因为当我们使用 print 的指令时，Perl 会使用 STDOUT 当作预设的档案代号，所以一般状况下，如果我们没有指定档案代号时，Perl 就会自动输出到 STDOUT。所以事实上，我们早就开始使用档案代号了，只是我们自己并没有发觉。或说，Perl 原来的期望就是希望使用者都可以在最没有负担的状况下任意输出到屏幕，或从键盘输入，毕竟 Perl 程序设计师那么的怕麻烦，一般的键盘输入，屏幕输出又是使用的那么频繁，当然要让程序设计师以最简单的方式达成。而且非常显然，这个目的也算达到了。

STDERR：标准的错误串流，也就是程序错误的标准输出。正常而言，当程序发生错误时，程序可以发出错误讯息来通知使用者，这时候这些错误讯息也能透过档案代号处理，把这些讯息丢进错误讯息串流。不过这样说实在不太容易理解，那我们来玩个游戏吧：

```
my $output = "标准输出";

print "$output\n";
print STDERR "$output\n";
```

我们一开始定义了一个字符串$output，一开始我们先直接从标准输出印出这个字符串，接下来我们便要求 Perl 把这个字符串送出到错误串流中。这样会发生什么有趣的事呢？让我们来看看：

```
[hcchien@Apple]% perl stderr.pl
标准输出
标准输出
[hcchien@Apple]% perl stderr.pl > error.txt
标准输出
```

第一次，我们直接执行了 stderr.pl 这支程序，而结果显然有点平淡无奇。于是我们第二次执行时，就在后面加上了">error.txt"，对于熟悉 Unix 操作的人大概知道，这样的方式其实是把程序执行时的错误讯息导向档案"error.txt"了。所以 STDOUT 只输出了第一行的 print 结果，而系统也产生了另外的 error.txt 的档案，因为我们把标准错误串流送到了这个档案里，所以我们可以发现档案里正好有我们输出到标准错误串流的字符串。这样的作法对于可能把 Perl 拿来进行系统管理的脚本程序时，就可以发挥很大的功能。因为我们也许希望某个程序可以帮我们进行一些日常的琐事，而在处理这些琐事的同时，如果发生什么异常状况，可以把错误讯息存在某个档案中，这样一来我们就可以只检查这个日志档案。

ARGV：我们可以直接利用参数来读取某些档案的内容，使用者只需要在执行程序时，在程序后加上文件名称作为参数，然后在程序中我们就可以直接读到档案的内容了。还是用个例子比较容易理解：

```
my $input = <ARGV>;
print "$input\n";

于是我们试着执行它，并且加上参数"error.txt"

[hcchien@Apple]% perl argv.pl > error.txt
标准输出
```

没错，当我们用了刚刚得到的 error.txt 当参数时，程序里面直接使用预设档案代号 ARGV 来读取档案内容，所以当我们印出来时，就可以看到刚刚写入档案的内容了。不过由于 Perl 读档案的性质，其实我们只印出了档案内的第一行，不过这部份我们稍后会再提到，这里暂且略过不谈。

不过 Perl 的 ARGV 其实非常好用，让我们来看看使用数组形式的@ARGV。也就是程序的参数，跟我们曾经提过的副例程参数有几分相似。它也是把取得的参数放入数组中，然后在程序里，就可以直接叫用数组，取出参数，就像这样：

```
my $input = shift @ARGV;
```

```
print "$input\n";

我们用同样的方式执行，可以看到这样的结果：

[hcchien@Apple]% perl argv.pl error.txt
error.txt
```

另外，我们也可以对 ARGV 进行一般档案代号的操作方式，不过这些将在稍后提到档案操作时再来讨论。

## 10.3 档案的基本操作

我们刚刚提到了一些Perl预设的档案代号，这些档案代号都是由Perl自动产生的。因此当我们开始执行Perl的程序时，就可以直接使用这些档案代号。可是除此之外，当我们希望自己来对某些档案进行存取时，就必须手动控制某些程序。所以现在应该来关心一下，当我们要手动进行这些档案的控管时，应该怎么做呢？

### 10.3.1 开档/关档

最基本的，我们要先开启一个档案，也就是我们必须将档案代号和我们想要存取的档案接上线。首先，我们可以使用open这个指令来开启档案代号，并且指定这个档案代号所对应的文件名称，所以我们使用的指令应该会会这样：

```
open FILE, "file.txt";
open OUTPUT, "<output.txt";  # 从档案输出
open INPUT, ">input.txt";  # 输入到档案
open append, ">>append.txt";  # 附加在现有档案结
尾
```

其实要开起一个档案代号非常的容易，至少从上面的例子来看，应该还算是非常的平易近人。那么我们只需要稍微的解释一些特殊的部份，大部份的人应该就可以轻松的开始使用档案代号了。

首先，最基本的语法也就是利用 open 这个指令来结合档案代号跟系统上实际的档案。所以我们看到了所有的叙述都是以 open 接下档案代号，接着是档案的名称。这样一来，我们就把档案代号跟文件名称连接起来，当然，前提是没有错误发生。不过不管如何，这看起来应该非常容易了。接下来，看看在文件名称前面有一些大，小于符号，这些又是什么意思呢？这些符号主要在于对于档案操作需求不同而产生不同的形式。首先我们看到的是一个小于(<)符号，这个符号代表我们会从这个档案输出数据，其实如果你对 Unix 系统有一点熟悉，你会发现这些表示方式跟在一般使用转向的方式接近。所以当你使用小于符号时，就像把档案的数据转向到档案代号中。如果你可以想象小于符号的方向性，那么大于符号也就是同样道理了。大于符号也就是把数据从档案代号中转入实际的档案系统

里，也就是写入到某个档案中。而如果系统中没有这个档案，Perl 会细心的帮你建立这个档案，然后你透过档案代号送出的数据就会由 Perl 帮你写入档案中。不过有一个部份必须要特别注意的地方，也就是如果你透过大于符号建立的档案系结，Perl 会把你指定的档案视为全新的档案，就如我们所说的，如果你的系统中没有这个档案，Perl 会先帮你建立一个新的档案。不过如果你的系统本来就已经存在同样的档名，那么 Perl 会把原来的档名清空，然后再把数据写入。 当然，这样就遇到问题了，因为如果你的程序正在监视网站服务器，而你希望只要服务器有状况发生就把发生的状况写入日志文件。这时候你大多会希望保留旧的日志，那么如果 Perl 每次都清空旧的日志内容就会让我们造成困扰。这时候我们总会希望 Perl 能把新的状况附加在原来的档案最后面的位置，那么我们就应该使用两个大于(>>)的符号，这也就是">>"跟">"的不同之处。

既然你开启了一个档案代号，最好的方式就是在你使用完后要归回原处(从小妈妈就这么告诫我们)。因此如果你不再使用某个档案代号时，你最好养成关闭这些档案代号的习惯，对了，应该还要提醒的是「适时」关闭不需要的档案代号。虽然 Perl 会在程序结束时自动帮你关闭所有还开着的档案代号，不过有些时候，你如果没有在档案处理完之后就尽快处理的话，恐怕会有让系统资源的负担增加。

至于关闭档案代号的方式也是非常简单，你只要使用 close 这个关键词，然后告诉 Perl 你所要关闭的档案代号，这样就没问题了。因此你如果需要关闭档案代号，你只需要这么做：

```
close FILE;
```

没错，就是这么容易。不过却也相当重要，至少你应该考虑好你自己的系统资源管理。否则等到等到持续拖累系统资源时才要怪罪 Perl 时可就有失公允了。另外，Perl 也会在你关闭档案代号时检查缓冲区是否还存有数据，如果有的话，Perl 也会先把数据写入档案，然后关闭档案。另外，档案也可能因为你的开启而导致其它人无法对它正常的操作，因此尽可能在完成档案操作后马上关闭档案代号是重要的习惯。

## 10.3.2 意外处理

有些时候，当我们想要开启档案时却会发现一些状况。例如我们想要从某个已经存在的档案中读入某些数据，可是却发生档案不存在，或是权限不足，而无法读入的状况。我们先看看以下的例子：

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

use strict;

open FILE, "<foo.txt";
```

```
while (<FILE>) {
   print $_;
}
```

在这里，我们希望开启一个档案"foo.txt"，并且从档案中读取数据，接着再把档案内容逐行印出。不过非常可惜，我们的系统中并没有这个档案。不过 Perl 预设并不会提醒你这样的状况，而且如果你没有使用任何的警告或中断，Perl 也能安稳的执行完这个程序，当然结果是「没有结果」。可是当我们在写程序，或是使用者在跟程序进行互动时，实在难保这些时候都不会什么错误会发生，也许只要把文件名称打错，可是 Perl 却不会自动的警告你。于是我们应该考虑发出一些警告，让发生错误的人可以实时修正错误。当然，你可以使用 warnings 来让 Perl 对于人为的错误发生一些警告，不过我们还有另外一种方法可以让你更轻易的掌握错误发生的状况，也就是让程序「死去(die)」。
die 函式就像他的字面意思，他可以让程序停止执行，也就是让程序「死去」。因此当我们希望程序在某些状况下应该停止执行时，我们就可以使用 die 函式来达成。而档案发生问题的状况则是 die 函式经常被使用的地方。因为很多时候我们一但开启了某个档案，大多就会把操作内容围绕着这个被开启的档案，可是如果档案其实没有被正确的开启，就很容易产生一些难以预料的问题，因此我们可以在档案开启失败时就让程序停止执行。以刚刚的程序作为例子，我们就可以把开启档案的部份写成：

```
open File, "foo.txt" or die "开启档案失败: $!";
```

在这里，有几个地方需要解释的，首先自然就是 die 的用法。我们先尝试开启 foo.txt 这个档案，接着用了一个逻辑操作数'or'，后面接着使用 die 这个叙述。根据我们对 or 运算符的了解，程序会先尝试开启档案"foo.txt"，如果成功开启，就会传回 1，因此 or 后面的叙述就会被省略。相反的，如果开启档案失败，open 叙述会传回 0。如此一来，Perl 就会去执行 or 后面的叙述，因此他就会 die 了，也就是只执行到这里为止。
利用 die 结束程序的执行时，我们会希望知道程序为什么进入 die 的状况，因此我们便利用 die 印出目前的情况。这听起来就像程序说完遗言之后就不动了。而 die 的打印就跟我们一般使用 print 没什么不同，因此我们可以加上可以提醒程序写作者或使用者的字符串。不过在刚刚的例子，我们看到了一个不寻常的变量："$!"。这是 Perl 预设的一个变量，他会储存系统产生出来的错误讯息。因为当我们透过 Perl 要进行档案的存取时，其实只是透过 Perl 和操作系统进行沟通，因此一但 Perl 对操作系统的要求产生失败的状况，他便会从操作系统得到相关的错误讯息，而这个讯息也会被存入$!这个变数中。
所以如果我们执行刚刚改过的那个程序，就可以得到像这样的结果：

```
[hcchien@Apple]% perl ch3.pl
```

```
开启档案失败: No such file or directory at ch3.pl
line 5.
```

因为档案不存在的原因，导致这一支 Perl 程序无法继续执行而在执行完 die 之后就停止了。而且 die 这个指令也在我们的要求下，传达了系统的错误讯息给我们，问题发生在你要开启档案时却没有发现这个档案或资料夹。所以利用 die 这个指令，你就可以在程序无法正确开启档案时，就马上中断程序，以避免不可预知的问题产生。
既然提到 die，我们就顺便来谈一下 die 的亲戚，"warn"吧！当你发生一些状况，可能导致程序发生无法正常运作时，你会希望使用 die 来强制中断程序的执行。可是有些时候，错误也许并没有这么严重，那么你就只需要发出一些警告，让执行者知道程序出了一点问题，让他们决定是否应该中断程序吧！我门把刚刚的程序改成这样：

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

use strict;

open FILE, "<foo.txt" or warn "open failed: $!";
while (<FILE>) {
   print $_;
}

print "程序在这里结束了\n";
```

你应该发现了，我们把 die 改成了 warn，然后最后加了一行打印的指令，告诉我们程序的结尾在那里。接下来我们来试着执行这支修改过的程序，你会看到这样的结果：

```
[hcchien@Apple]% perl ch3.pl
open failed: No such file or directory at ch3.pl
line 5.
the end of the script
```

## 10.3.3 读出与写入

在我们可以正确的开启档案代号之后，接下来我们就可以开始存取档案中的资料，当然最主要的就是读取，以及写入档案。

透过档案代号来读取档案内容倒是不太有什么困难。我们大多使用钻石符号(<>)来进行档案内容的读取。所以我们可以像这样进行档案操作：

```
#!/usr/local/bin/perl -w

use strict;

open LOG, "/var/log/messages";   # 打开这个日志文件
while (<LOG>) {                  # 利用钻石符号读入数据
   print if (/sudo/);            # 符合比对的数据就打印出来
}
```

看起来非常容易，不是吗？
我们先用刚刚了解的方式开启了一个档案代号，并且利用这个档案代号联系到档案"/var/log/messages"。在一些 Unix 系统中也许会看到这个档案，它会纪录一些使用者登入或是使用 root 权限的消息。而在这个档案中，如果有使用者利用 sudo 这个指令进行某些操作时也会被记录下来。因此我们就可以透过这个档案知道服务器上有些什么状况正在发生。
接下来我们透过钻石符号开始逐行读取日志档案中的数据，透过循环 while 读取档案中的数据时，while 会把所读到的数据内容放进 Perl 的预设变量$_中，一直到档案结束，传回 EOF 时，循环便会结束。因此我们就将所读取的数据进行比对，以 sudo 这个关键词作为比对样式，把符合的结果印出来。
这样一来，只要系统中有人使用 sudo 进行系统操作时，我们就可以检查出来，而且印出来的结果会像是这样：

如果你是负责管理一些 Unix 的服务器，利用这样简单的方式，确实可以帮忙你完成不少工作。很显然，利用档案的操作，你还可以进行更多对日志档案的分析。例如你可以分析网站服务器的各项数据，虽然其实已经有很多人用 Perl 帮你完成这样的工作了。(注二)
基本上，从档案内读取内容的方式就是这么容易，因此你可以简单的运用档案的内容进行所需要的工作。还记得我们在介绍 open 时的说明吗？我们有几个开启档案的方式包括了几种描述子，例如大于(>)，小于(<)，以及两个大于(>>)。而且我们也都简单的描述过他们的差异，现在也许就是测试这些描述子的好时机，我们先来看看小于符号用于开文件的时候，会有什么影响。
我们之前也提过小于符号用在开文件作为描述的话，是用来表示从档案内读取数据。那我们是不是就只能允许使用者读取数据呢？先来看看这个小小的程序吧：

```
open LOG, "<log.txt" or die $!;
while (<LOG>) {
    print $_;
}
print LOG "write to log" or die $!;
```

假设我们已经有了"log.txt"这个档案，否则程序就会挂在中间，没办法继续执行。那么我们来看看执行结果吧：

```
file for log
Bad file descriptor at ch3.pl line 9, <LOG> line
1.
```

第一行就是原来 log.txt 里面的内容，我们可以很轻松的读出其中的资料，并且印出来，可是当我们要将数据写入时，却出现了错误讯息。没错，当初我们在开启这个档案时，只要求 Perl 给我们一个可以读出资料的档案，如今要求写入，果然就遭到拒绝。

看来一但我们使用了小于符号作为开启档案代号的描述子，那么我们就不能轻易的把数据写入所开启的档案中。想当然尔，Perl 应该也不会让我们在开启一个利用大于符号指定为写入的档案中把数据读出吧？要想测试这样的结论，我们只需要把刚刚的程序修改一个字符，也就是把小于符号改成大于，那么就让我们来看看执行后的结果吧：

我们尝试着执行被我们修改了一个字符的程序，结果发生了什么事呢？档案没有输出任何结果。好像很出乎意料？其实一点也不，而且正如 Perl 所要求我们的，我们使用了大于符号表明我们想要把数据写入档案 log.txt，因此当我们想要从档案读取数据并且逐行印出结果时就无法成真。不过我们接下来去看看 log.txt 的内容。正如我们所预料的，程序已经正确的把字符串"write to log"写到档案 log.txt 里面了。

既然使用大于符号跟小于符号都符合我们的期待，那么如果我们什么描述子都没有使用，会是什么样的情况呢？我们只需要使用刚刚的测试程序，并且把描述子全部取消，再来试试结果如何吧！

结果我们发现，Perl 还是可以读出档案的内容，可是却无法写入。也就是跟我们使用小于符号时是一样的状况，这点其实对于经常必须使用档案的人来说其实是非常重要的。所以如果你有机会使用档案的存取时，可别忘了这一点。

另外，大于符号与两个大于的差别我们也曾经提过，这部份对于可能使用 Perl 来进行日常管理工作的人更是必须牢记。我们之前提过，一样是开启一个可以写入的档案，使用一个大于符号(>)的时候，Perl 会判断你是否已经有存在这个档名的档案，如果档案已经存在，那么 Perl 将会清空档案内容，把他视作一个新的档案来进行操作。如果在系统中档案并不存在，那么 Perl 就会跟系统要求开启一个新的档案。当然，在你使用两个大于符号的时候，Perl 会把你要写入档案

的内容以附加的方式存入。当然，如果你的系统中并没有这个档案，那么 Perl 也会先开启一个新档，并且把你所要求的内容写入档案中。这对于想要建立类似日志文件的需求有着绝对的帮助，例如你可能会需要 Perl 来作为监控网络的状况，这时候你会需要每次有新状况时就把它记录下来，而且需要保留原来的纪录。那么如果你还是使用大于符号的话，你可就要小心原来的数据内容遗失了。
当然，我们知道开启档案时可以利用三种描述子去指定所要开启档案代号的状态，不过如果你什么都没加的状况下，Perl 又会作怎么样的处理呢？我们继续用刚刚的例子来进行实验吧。我们把开启档案的描述子拿掉，其它的部份一切照旧。所以你的程序就像这样：

```
open LOG, "log.txt" or die $!;
print LOG "write to log\n" or die $!;
```

接着我们发现，这样的结果就跟我们使用小于符号的效果是相同的，也就是 Perl 只会从档案中读出数据，却无法写入。

有了基本读写档案的能力之后，我们还必须了解该怎么样透过 Perl 去控制系统的档案以及数据夹。这样才能确实掌握系统的档案管理，尤其当你希望使用 Perl 来进行系统管理时，也就会更需要这样的能力，所以我们接下来就要讨论利用 Perl 对档案系统的操作。

习题：
1. 试着将下面的数据利用 perl 写入档案中：

```
Paul, 26933211
Mary, 21334566
John, 23456789
```

2. 在档案中新增下列数据：

```
Peter, 27216543
Ruby, 27820022
```

3. 从刚刚已经存入数据的档案读出档案内容，并且印出结果。

注一：不过当你打算这么作的时候，也许要考虑这支程序未来只有你在维护，否则你这样的动作很可能会因为接下来维护的人需要花更多的时间来看懂程序而提高不少维护成本。
注二：其实跟这章主题不太有关，不过例如 awstats 就是这类型的工具。

# 11. 档案系统

上一章我们提到了一些关于在Perl当中使用档案代号来进行档案存取的工作，不过要能灵活运用这些操作，你应该要有对于系统本身的档案架构有一些认识。因为运用档案代号，实际上你也是在操控整个系统的目录跟档案。所以我们接下来就要简单提醒大家一些基本的事项，并且告诉大家应该怎么利用Perl去进行档案的操作。

## 11.1 档案测试

我们在上一章曾经尝试打开一个档案，并且从档案内读出其中的内容。不过我们也遇到了一些问题，也就是档案可能会因为不存在而使数据读取发生问题。因此我们利用die的方式来判断，假如程序无法打开这个档案代号，那么就中止程序继续进行。当然，找不到档案是我们设法开启档案代号时可能发生的错误之一。我们也许还可能发生其它问题，比如没有权限打开指定的档案等等。不过对于这当中的某些状况，我们其实在准备开启档案时可以先进行测试，也就是所谓的档案测试，在说明可以进行测试的项目之前，我们可以先来看看这个例子：

```
#!/usr/local/bin/perl -w

use strict;

my $logfile = "/var/log/messages";  # 先指定档案
到变量 $logfile
if (-e $logfile) {                  # 判断档案是否
存在
    open LOG, $logfile or die "$!";  # 开启档案
代号
    my $line_num = 1;
    while (<LOG>) {
        print "$line_num\t$_\n";
        $line_num++;
    }
} else {
    print "档案不存在\n";
}
```

这个程序的主要工作在于读出系统日志文件的内容，并且帮忙加上行号印出。当然，我们先指定要开启的文件名称是"/var/log/messages"这个档案，接下来便利用档案判断的参数"-e"来确定档案是否存在。如果这个档案确实存在，我们就打开档案代号"LOG"，用来联系"$logfile"这个档案，也就是"/var/log/messages"。当然，这时候我们虽然确定档案存在，可是因为还是可能存在其它导致无法正常

开启档案的状况，因此我们还是决定一但开启失败就利用 die 印出错误讯息，然后中断程序。如果档案开启没有问题，我们就可以开始一行一行把资料读进来，然后加上行号后输出了。
这样看来，"-e"的判断似乎功用不大，因为我们判断如果档案不存在，好像也没有特殊的动作。所以我们来让"-e"看起来能有些帮助：

```
#!/usr/local/bin/perl -w

use strict;

while (-e (my $logfile = shift)) {  # 判断档案是否存在
    open LOG, $logfile or die "$!";  # 开启档案代号
    my $line_num = 1;
    while (<LOG>) {
        print "$line_num\t$_";
        $line_num++;
    }
}
```

这样看起来好像有趣了一些，我们来看看到底改了什么。首先，我们把原来指定给变量$logfile 的档案取消，让$logfile 变成是使用者由执行时输入的参数。接着我们依然检查了这个档案是否存在，如果存在则打开并加上行数印出。
其实并不困难，我们只需要以指定的参数就可以用来检查档案的属性。所以我们来看看到底有那些参数可以使用：

```
-A  档案上次存取至今的时间
-B  档案被判断为二进制文件
-C  档案的 inode 被更改至今的时间
-M  档案上次修改至今的时间
-O  目前实际使用者是否为该档案或目录的拥有者
-R  目前实际的使用者具有读的权限
-S  档案代号是否为 socket
-T  档案判断为文字文件
-W  目前实际的使用者具有写的权限
-X  目前实际的使用者具有执行的权限
-c  字符型档案
-e  检查档案或目录是否存在
-f  判断档案是否为文字文件
```

```
-g  档案或目录具有 setgid 属性
-k  档案或目录设定了 sticky 位
-l  档案代号是一个符号连结
-o  目前的使用者是否为该档案或目录的拥有者
-r  目前的使用者具有读的权限
-s  档案或目录存在而且有内容
-t  档案代号是 TTY 装置
-u  档案或目录具有 setuid 属性
-w  目前的使用者具有写的权限
-x  目前的使用者具有执行的权限
-z  档案或目录存在而且没有内容
```

其中有许多是关于系统本身的相关知识，例如使用者 id，群组 id 等等。这部份建议各位应该能够针对自己所使用的操作系统，去找到相关的参考书籍。其它例如在 Unix 系统上使用，大多则采用相类似的权限判断方式。当然，其中有些部份是仅供参考，例如档案是否为文字文件，或是二进制文件。对于 big5 档案来说，Perl 就可能会误判成二进制文件。
当然，很多时候我们还是需要在对档案进行存取之前，先确定他们相关的状况。例如是否能够有足够的权限，或是我们可以得到档案最后被修改的时间等等。大部份的时候，这些判断可以给我们当作很好的参考。例如我们可以设定时间清除过久没有更新的档案等等。这些工具对于使用 Perl 来管理日常工作的管理者来说更是能够提供非常好的帮助。

## 11.2 重要的档案相关内建函式

对于系统中的档案系统，Perl大多数的时候总是透过底层的操作系统去进行操作，因此你会发现很多的函式和操作系统提供的函式大多非常接近(注一)。这样其实也非常能够帮助使用者用简单的方式记忆，而不需要多背另一套指令函式。例如我们刚刚提到的档案测试，也就是Perl所提供第一个属于档案操作的函式。因此如果你想要获得更精准的说明，你可以考虑使用"perldoc -f -X"来查看所有的测试符号。
接下来，我们来看看还有那些函式是我们可以善加利用的部份。Perl在处理档案代号或其它档案相关的函式多达十几个，其实已经足以应付大多数的使用。接下来我们将挑出几个经常被使用的内建函式，让读者可以开始熟悉该怎么在Perl中控制档案系统。

chdir：就像你在大多数操作系统下所使用的指令一样，你可以利用chdir来切换目前工作的目录。因此我们可以使用下面的方式来指定我们想要操作的目录：

```
chdir "/tmp" or die $!;
open LOG, ">log.txt" or die $!;
print LOG "write to log\n" or die $!;
```

没错，我们只是小小的修改了刚刚的程序，把原来没有指定目录的状况，改成在目录"/tmp"下了一个档案 log.txt，并且写入字符串。就像你在大多数 Unix 操作系统中的状况，你也可以单独使用 chdir 而没有附带任何的参数，这时候系统会根据你的环境变量$ENV{HOME}来决定应该切换到哪一个目录。

chmod：对于熟悉 Unix 的系统对于此应该也是非常的熟悉，这个函式就是呼叫系统中 chmod 的操作，来修改档案或是目录的权限。如果你对于系统的权限结构还不太熟悉，建议你先看一些相关的文件，可以了解 Unix 系统下对于权限的限制跟实作的方式。当然，Perl 并不太愿意改变大家的使用习惯，所以如果你经常使用 Unix 下的 chmod 指令，那么你可以继续你的使用习惯，就像这样：

```
chmod 0444, 'log.txt';
```

不过也有比较具有弹性的用法，例如你可以这样使用：

```
$mode = 0644;
chmod $mode, 'log.txt';
```

有些部份通常会让你搞错，因此你必须特别注意。如果你刚刚把$mode 这个变量写成下面的形式，那么可能执行之后，可能会发生一些让你意想不到的状况。

```
$mode = "0644";
chmod $mode, 'log.txt';
```

我们直接来看看实际的状况吧！它的权限目前是 0444。如果我们想要把它利用刚刚的权限来修改它，那么会发生什么事呢？

```
Inappropriate file type or format at ch3.pl line
6.
```

Perl 毫不留情的给了我们一个错误讯息，告诉我们这样指定档案权限是不被允许的。很多人可能已经一头雾水了，我们加了引号之后到底有什么差别呢？你还记得我们刚刚指定权限的作法吗？

```
$mode = 0644;
```

其实当你在使用这样的纯量赋值时，Perl 会把你所指定的数字设定为八进位。可是当你帮它加上引号之后，也就是使用了$mode = "0644"后，它就变为一个字符串了。可是 chmod 所需要的可不是字符串，而是一个八进位的数字，所以如果你使用了引号来定义权限的值，别忘了把他转为八进位制，所以我们可以改写成这样：

```
$mode = '0644'; chmod oct($mode), 'log.txt';
```

当然，最省力还是前一种的方式，不过既然方法不只一种，使用者可以选择自己最容易接受的方式。至于直接使用八进位的变量定义，应该是最被推荐的使用方式。不需要繁杂的转换手续，也减少打字跟错误的机会。

chown：修改档案或是数据夹的拥有者也是你在管理 Unix 系统会遇到的状况。其中这包括了使用者 id(uid)跟群组 id(gid)，使用的方式则是将使用者 id 跟群组 id 利用串行的方式来描述，配合上想要修改的档案，所以指令的形式应该是：

```
chown LIST;
```

用实际的例子来看，我们则可以写成这样：

```
chown $uid, $gid, 'log.txt';
```

至于后面的档案，则可以利用串行的方式表示，或直接以数组方式。也就是说，你当然可以用这样的方式来表达 chown 的形式：

```
chown $uid, $gid, @array;
```

直接使用数组确实是有相当的好处，我们可以利用样式比对找出我们要的所有档案，然后一次进行相关的修改。例如在系统的使用上，我们常常使用星号(*)作为万用符号，比如你可以利用这样的方式找出所有 Perl 的档案：

```
ls *.pl
```

而在 Perl 中，也有相关的用法，也就是 glob。因为这个功能非常重要，所以我们接下来就来看看 glob 的用法。

glob：他的语法其实相当简单，也就是利用 glob 接上一个样式，作为比对的标准。所以你可能会这么使用：

```
@filelist = glob "*.pl";
```

这样的方式就跟你在系统下寻找符合某些条件的档案用法一样，所以你可以把利用 glob 所传回来的档案串行放入一个数组之中。然后再针对这个数组进行 chown 或是 chmod 相关的操作。也许你会考虑，这样的作法跟你在 shell 底下的运作有什么差别吗？其实很多时候，Perl 可以利用这些方式把你日常必须重复进行的工作处理掉。
不过其实有时候你也许会看到某种写法，就像这样：

```
@filelist = <*.pl>;
```

这样得出的结果其实跟你使用 glob 有着异曲同工之妙，也就是取得目前目录下的档案，并且依据你所描述的样式传回你需要的档案。因此你可以轻易的取得你想要的档案，例如你想要印出目录下的所有附属文件名是 txt 的档案，那么你只需要这么作：

```
for $file (<*.txt>) {
    open FH, $file;
    print $_ while (<FH>);
}
```

看出这其中有一些奥妙了吗？我们利用角符号代替了 glob 的工作，可是同时角符号也被我们拿来作为读取档案内容的运算。确实是如此，那么 Perl 会如何分辨其中的差别呢？其实由于档案代号必须符合 Perl 的命名原则，因此 Perl 可以藉此判断你目前的语境下是里是用角括号来处理档案代号或是进行 glob 的处理。当然，其中会有一些例外，比如你用这样的方式来表达档案代号：

```
open FH, $file;
$filehandle = "FH";
print $_ while (<$filehandle>);
```

这时候角括号里面放的其实是一个 Perl 的纯量变数，不过这个纯量变量却是被指定到另外一个档案代号，所以 Perl 还是会以对待档案代号的方式来对待它。这应该一点都不让人意外，不过你现在应该可以应付大多数的状况了。

link：你有时候会需要把档案建立起链接，在系统底下，你可以直接使用"ln"这个指令来达成你需要的目的。而透过 Perl，则可以利用 link 来达到一样的工作。他的语法就像这样：

```
my $res = link "/home/foo", "/home/bar";
```

这样的意思就是把"/home/foo"这个档案连结到"/home/bar"，或者你可以说"/home/bar"是"/home/foo"的一个连结。至于 link 这个指令则会有回传值，如果连结成功，则回传值为真值，相反的，如果连结失败，则会传回伪值。我们来试试这个例子：

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

use strict;

my $ret = link "log.txt", "log.bak";
open FH, "log.bak" or die $! if ($ret);
print $_ while (<FH>);
```

执行之后，我们就可以看到数据夹中多了一个叫做"log.bak"的档案，不过如果你需要真正了解他的运作，我们还是建议你去看看关于 Unix 下关于档案及数据夹的解释，其中 inode 这个观念可以帮助你确实了解这样的连结所产生的意义。不过在这里，我们就暂且先不深入的探讨 Unix 下的相关部份。

mkdir：接下来，我们应该来告诉大家，该怎么开启自己的一个数据夹。这个指令跟你在 Unix 底下的使用非常接近，你只需要使用这样的方式就可以了：

```
mkdir PATH;
```

这看起来跟你在命令列下的用法一模一样，而且就是这么简单。所以你几乎不需要学习新的东西就可以很轻松的在 Perl 底下新增一个数据夹。另外，你还可以透过 umask 来指定这个新数据夹的权限。而用法也是跟刚刚类似，唯一的差别只是把你希望指定的 umask 放在叙述的最后。所以看起来应该就像这样：

```
mkdir PATH, umask;
```

所以你可以把新增加的这个资料夹指定某个特殊的权限，例如你希望开一个所有人都可以任意存取的数据夹，那么就可以这样写：

```
mkdir foo, 0777;
```

rename：接下来我们来看看如何使用 Perl 来帮你的档案改名字，其实当你开始利用 Perl 来对档案进行操作时，修改档名是非常有用的一项工具。我们可以先来看看一个实际的范例：

```
my $file = "messages.log";
if (-e $file) {
   rename $file, "$file.old";
}
open FH, ">$file";
print FH, "接下来就可以写入数据";
```

在实际运用时，如果我们可以适时的搭配档案的测试运算，那就可以产生出很不错的效果。就这个例子，我们先利用"-e"来判断档案是否存在。如果档案存在，我们就把档案更名，也就是再档案结尾加上".old"，在这里，我们就看到了 rename 的用法，也就是：

```
rename $oldfile, $newfile;
```

当我们正确的把档案改了名字之后，就可以安心的把新的数据写入档案了，你应该注意到了，我闷在这里因为是使用了大于(>)符号来进行开启档案代号的动作，所以如果之前没有先把档案更名，那么旧有的数据就会被取代了。

rmdir：就像你在操作系统下的作法一样，你可以利用 rmdir 来删除一个数据夹。

不过也跟你在终端机前使用 rmdir 一样，如果数据夹里面还有存在其它档案，rmdir 就会产生失败，而且会传回伪值，很显然，这是相对于删除成功所传回的真值。所以如果你是 Unix 系统的惯用者，也许你应该非常熟悉这个函式，你只需要这么指定：

```
rmdir FILENAME;
```

stat：其实如果你想要更灵活的使用我们介绍的这些函式来对档案系统进行控制时，你应该要先了解 stat 这个重要的函式。为什么 stat 这个函式这么重要，也许我们来看看下面的范例就能够很快的了解了：

```
my @ret = stat "log.txt";
print "$_\n" for (@ret);

于是我们试着执行这个程序，会看到这样的结果：

234881034     # 装置编号
1183005             # inode 编号
33060               # 档案模式(类型及权限)
1             # 档案的连结数目
501           # uid
501           # gid
0             # 装置辨识
17            # 档案大小
1078894964    # 最后存取时间
1078894638    # 最后修改时间
1078939576    # inode 修改的时间
4096          # 档案存取时的区块大小
8             # 区块的数目
```

毫无疑问，我们确实可以利用 stat 个函式得到相当多的档案相关信息，因此如果你想要对档案进行操作之前，也许可以先利用 stat 来得到相关的讯息。
我们刚刚利用一个数组来储存 stat 的回传值，这样也许不容易分辨各个值所代表的意义，所以你当然可以改用这样的方式来取得相关的资料：

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size
,$atime,$mtime,$ctime,$blksize,
$blocks) = stat("log.txt");
```

另外，有些时候你也许会看到有人使用 lstat 来取得档案的相关信息，不过基本上这两个函式所进行的工作应该是一样的，所以除非你想要多打一些字，否则还是可以直接使用 stat 就好了。

unlink：就像你使用的 rm 一样，unlink 也可以让你删除系统中的某些档案。而且 unlink 的用法十分简单，基本上就是传进你想要删除的档案串行。意思就是说，如果你搭配着 glob 或是角括号(<>)使用，那么你就可以过滤出某些特殊的档案，并且加以删除。相信大家经过上面几个函式的训练，应该可以很轻易的使用这个函式，就像这样：

```
my @files = <*.txt>;
unlink @files or die $!;
```

当然，别忘了要删除档案千万要非常的小心，可别因为一时大意就把资料全部的毁了(注二)。当然，我们刚刚说了，在 unlink 后面所连接的参数是一个串行，所以你可以使用任何表达串行的方式，其中当然包括一一列出你所要删除的档案。所以如果有一个程序写的像这样：

```
#!/usr/bin/perl
use strict;
unlink @_ or die $!;
```

那么他看起来像不像阳春的 rm 指令呢？其实有时候玩玩也是还满有趣的。

utime：Perl 另外也提供了一个让你修改档案时间的函式，也就是 utime。utime 的用法也是传入一个串行，所以基本上会是：

```
utime LIST;
```

不过不太一样的地方在于你必须指定你所要修改的时间参数，所以其实比较常看到的用法也许会比较接近这样的形式：

```
utime $atime, $mtime, @files;
```

其中第一个参数就是档案存取时间，第二个参数就是档案最后一次修改的时间。

## 11.3 localtime

这个函式看起来跟档案操作并没有什么直接的关系，不过我们刚刚看到了一些不太友善的数字，也就是对于档案相关时间的描述。例如我们利用stat传回来的日期都是这样子的表示方式：

```
1078894964    # 最后存取时间
1078894638    # 最后修改时间
1078939576    # inode 修改的时间
```

这时候，我们就可以使用 localtime 来转换成一般人可以接受而且使用的信息。localtime 会传回一个串行，分别代表用来表示时间的各个字段，所以你可以利用这样的方式取得你需要的字段：

```
@realtime = localtime($timestamp);

只是如果你使用这样的方式，恐怕自己也很难很快的使用，所
以也许可以换一个方式：

($sec, $min, $hour, $day, $mon, $year, $wday,
$yday, $isdat) =
localtime ($timestamp);
```

所以如果你想要取得档案最后修改的正确时间，你可以利用下面的方式达成：

```
my
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size
,$atime,$mtime,$ctime,$blksize,
$blocks) = stat("log.txt");
($sec, $min, $hour, $day, $mon, $year, $wday,
$yday, $isdat) =
localtime ($mtime);
```

呼，确实有一点冗长。不过确实可以让你正确的取得大部份的信息。

习题：
1. 列出目前所在位置的所有档案/数据夹名称。
2. 承一，只列出数据夹名称。
3. 利用 Perl，把目录下所有附文件名为.pl 的档案修改权限为可执行。

注一：当然所谓的接近，指的是和 Unix 系统的接近。
注二：上面的程序就让本书内容差点付之一炬，幸好作者使用了版本控制系统来进行备份。

# 12. 字符串处理

我们前面两章提到了许多关于档案的操作，现在我们应该可以很轻松的从档案中取得我们需要的信息了。不过如果只是空有一大堆的信息，却没有办法处理的话，只怕也没有什么帮助。不过既然对于Perl来说，大多数的东西都是由数字跟字符串组合而成，那么一但我们可以用简单的方式来整理字符串的话，那么应该就可以让这些信息变得相当有用。

## 12.1 简单的字符串形式

我们在讲解变量的时候已经提过关于Perl是如何对待字符串的，虽然对于不少其它程序语言来说，字符串其实只是字符的串行。可是Perl却简化了这样的观念，因此反而让字符变成长度为一的字符串。就像对待数字一样，Perl并不会要求程序设计师去强制规定某些变量只能放整数，某些变数只能放浮点数。
这样宽松的规定确实让程序设计师省了许多麻烦，不过当你的分类越粗略时，要怎么有效的对这些数据进行处理就显得更加重要。而就像许多人对于Perl的印象，它在处理字符串时非常的具有威力。这当中的原因除了正规表示式之外，Perl对于字符串的控制显然也有一些有趣的部份。
对于字符串最基本的操作应该就是长度了，我们经常会要求知道字符串的长度，这时候，只需要使用length这个函式就可以取得你所指定字符串的长度了。

```
my $string = "string";
print length($string);    # 长度是 6
```

当然，有时候你会被某些控制字符所欺骗，因为他们也是占有长度的，就像这样：

```
my $string = "string\n";
print length($string);    # 这时候长度变成 7 了
```

取得字符串长度只是第一步，接下来我们可能需要找出字符串的相关性。例如我们会想要取得某个字符串的其中一段。不过我们可能会先需要知道这个子字符串在原来字符串的位置这时候就可以使用 index 来取得相关的讯息了。用实际的例子我们可以很容易看到 index 的用法：

```
my $mainstring = "Perl Mongers";
my $substring = "Mongers";
print index($mainstring, $substring);  # 印出 5
```

看来相当容易对吧，Perl 会告诉你子字符串第一个字母所在的位置，只是字符串是由 0 开始算起。也就是说，如果你在字符串的一开始就找到符合的子字符串，那么 Perl 就会传回 0。不过如果 Perl 发现你所指定的子字符串不在原来的字符串中，那么就会传回-1。
当然，有些人会关心中文字符串的处理，我们先来试试下面的例子：

```
my $mainstring = "台北 Perl 推广组";  # Big5 码的
中文
my $substring = "推广组";
print index($mainstring, $substring);  # 印出 8
```

其实 index 传回的是位，所以如果你要利用 index 来找到某些中文字在在字符串中是位于第几个位那么就没有问题。当然，如果你要的是以中文的角度来看，那么「字」的观念在这里显然并不存在。

另外，就像正规表示式一样，index 在进行比对时，也会确定找到你所需要的字符串就停止了。所以 index 传回的就永远会是第一次找到子字符串的位置。实际的结果会像是这样：

```
my $mainstring = "perl training by Taipei perl
mongers";
my $substring = "perl";
print index($mainstring, $substring);
 # 结果是 0
```

因为 Perl 在一开始就找到了比对成功的字符串"perl"，因此它马上传回 0，然后就停止比对了。可是这样有时候是不是会有些不方便呢？所以我们来看看 Perl 对于 index 这个函式的描述。

```
index STR,SUBSTR,POSITION
index STR,SUBSTR
```

我们好像发现一些署光，没错，根据上面的语法，其实我们还可以使用 index 的第三个参数，也就是位置。所以你可以要求 index 从第几个位开始找起，例如：

```
```

```
my $mainstring = "perl training by Taipei perl
mongers";
my $substring = "perl";
my $first = index($mainstring, $substring);          # 先找到第一次出
print index($mainstring, $substring,
$first+1);    # 接下去找
```

这样的用法就可以让你找到下一个出现子字符串的地方。当然，如果你没有加第三个参数的话，那么 index 会把它预设为 0。也就是我们一开始一直使用的方式。不过如果你不知道子字符串会出现多少次，可是你又想找到最后一次出现的位置，那么你会想要怎么作呢？用个循环好像是我们目前可以想到的作法，所以我们就来试试吧：

```
my $mainstring = "perl training by Taipei perl
mongers";
my $substring = "perl";
my ($pos, $current);
my $pos;
my $current = -1;
until ($pos == -1) {
# 到找不到正确字符串为止
    $pos = index($mainstring, $substring,
$current + 1);  # 从上次找到的位置往下找
    $current = $pos unless ($pos == -1);
}

print $current;
# 印出 24
```

看起来好点小小的复杂，因为我们必须用一个循环去搜寻所有的子字符串，一直到它找到最后一个。不过有没有可能从字符串的尾端去找，那么我们就只需要找到第一个符合的字符串，因为对于从字符串开头而言，那就会是最后一次比对成功的子字符串了。
看来这样的需求不少，因此 Perl 的开发者也就提供了另外一个函式，也就是 rindex，基本上 rindex 的使用方式跟 index 几乎一模一样，只不过它是从字符串尾端开始找起。既然如此，我们就改用 rindex 来完成刚刚的工作：

```
my $mainstring = "perl training by Taipei perl
mongers";
my $substring = "perl";
```

```
print rindex($mainstring, $substring);
 # 同样印出 24
```

这样显然方便了许多，不过对于 rindex 来说，如果我们指定了第三个参数，那其实是用来表示搜寻的上限。也就是我们要求 rindex 在某个位置之前的就不找了。这样描述似乎太过笼统，我们不如来看看实际的运作情形吧：

```
my $mainstring = "Taipei perl mongers";
my $substring = "perl";
print rindex($mainstring, $substring, 4);  # 结果传回 -1
```

其实参数的意义也就是「以这里为开始搜寻的起点」，所以如果我们把参数设定为 4 的话，Perl 就只会从第四个位往回进行比对，所以当然不会比对成功。

利用 index 找出子字符串的位置之后，我们还可以利用 substr 来取出某个字符串内的子字符串。我们先看看 substr 的标准语法：

```
substr EXPR,OFFSET,LENGTH
substr EXPR,OFFSET
```

最简单的方式就是只有指定要处理的字符串跟另一个我们想取得子字符串的起始点，所以你可以让它看起来像这样：

```
my $string = "substring";
print substr($string, 3);  # 果然印出 string 了
```

如果你没有传入长度这个参数，那么 Perl 会预设帮你取到字符串结束。所以我们刚刚取得的字符串就是"string"，如果你想要的只是"str"三个字母，你就可以指定长度，也就是像这样：

```
my $string = "substring";
print substr($string, 3, 3);  # 这样就只会印出 str
```

有时候如果字符串太长，也许从字符串结尾开始算起会比较容易，就像 index 搜

寻子字符串的位置，可以利用 rindex 来要求 Perl 从字符串尾端找起，那么 substr 要如何使用类似的方式呢？答案就是利用负数的起始点，这样说好像不如直接看个范例：

```
my $string = "Taipei Perl Mongers";
print substr($string, -12, 4);  #印出 Perl
```

另外，我们之前使用过正规表示式来进行取代的工作，例如下面的字符串，我们想把"London"以"Taipei"取代，所以可以利用正规表示式，作这样的处理：

```
my $string = "London Perl Mongers";
$string =~ s/London/Taipei/;
```

当然，有些时候使用正规表示式未必比较方便。或是我们可以取得的资料有限，这样的情况下，也许可以利用 substr 来进行字符串替换。substr 也可以进行替换，别担心，你没看错，我们就来实验看看，利用 substr 来把"London"换成"Taipei"。

```
my $string = "London Perl Mongers";
substr($string, 0, 6) = "Taipei";
print $string;                  # 就会印出
"Taipei Perl Monger"
```

这样看起来好像没什么，显然不够绚丽，我们来把它改写一下吧！

```
my $string = "London Perl Mongers";
print substr($string, 0, 6) = "New York";
print $string;                  # 你完全不需要
考虑字符串长度
```

字符串长度对 Perl 来说并不是个问题，所以我们可以很安心的使用长度不相等的字符串来进行替换，Perl 可以自动的帮你处理长度的问题。其实这种需求显然相当的高，所以这也是 substr 的另一种标准语法，也就是说，我们可以把刚刚的语法用这种方式来取代：

```
my $string = "London Perl Mongers";
substr($string, 0, 6, "New York");  # 使用第四个
参数
print $string;                      # 也是会替换为
New York Perl Mongers
```

## 12.2 uc 与 lc

字符串中，偶而会有一些恼人的状况，也就是字符串的大小写问题。例如你弄了一个会员账号系统，因此这个系统必须让管理者可以开账号，使用者可以登入等等。有许多牵涉到账号的输入，比对问题，这时候如果还有字母大小写的问题，也许会更让人气馁，尤其目前的大多数使用者几乎都习惯了大小写不分的使用状况。所以有时候也许需要藉由系统自动转换的方式来避开这一类琐碎的事。

uc也就是upper case的意思，所以很清楚的，它会帮你把字符串中的英文字母传换成大写，然后回传，就像这样：

```
my $string = "I want to get the uppercased
string";
print uc $string;    # 结果就变成了 "I WANT TO GET
THE UPPERCASED STRING"
```

怎么样，一点都不意外吧！而且依此类推，lc 则是转成小写之后回传，这应该不需要重新举例了。
这样一来，我们虽然可以取得全部大小或全部小写的字符串，可是在更多时候，我们其实只要前缀的大小就好了，那么可以怎么作呢？也许可以考虑使用 ucfirst，看这个函式名称就觉得它是我们想要的东西。，既然如此，那我们就直接来试一下吧：

```
my $string = "upper case";
print ucfirst $string;    # 印出 Upper case
```

就像我们所预期的一样，我们让 Perl 把第一个字母印出了大写，不过这完全是意料之中？相对应于 ucfirst，Perl 也提供了 lcfirst 这个函式，而且正如大家所猜想的一样，它会把字符串的第一个字母转为小写。

## 12.3 sprintf

我们已经非常习惯使用print来印出我们执行程序所得到的结果了，可是很多时候print印出的结果却未必让人满意，不满意的原因有很多时候是因为它的输出格式无法依照我们的要求，或者说我们需要花更多的力气才能达到我们所期待的样子。所以这时候，sprintf就可以派上用场了。sprintf主要是可以帮助我们作格式化的打印指令。例如你总是希望印出两位数的小数点，那么这时候，你应该就会非常需要sprintf来帮助你。我们来看看我们可以怎么作呢？

```
my $num = 21.3;
my $formatted = sprintf "%.2f", $num;
 # 先设定好格式
print $formatted;
```

当然，sprintf 的功能相当的丰富，如果你打算使用的话，应该先来看看 sprintf 提供什么样的强大功能：

```
%%   百分比符号
%c   字符
%s   字符串
%d   包含正负号的十进制整数
%u   不包含正负号的十进制整数
%o   不包含正负号的八进位整数
%x   不包含正负号的十六进制整数
%e   以科学符号表示的符点数
%f   固定长度的十进制浮点数
%X   使用大写表示的%x
%E   使用大写表示的%E
```

其它还有一些不同的格式指定方式，当你开始使用的时候，你可以参考 printf 的说明文件。

## 12.4 排序

对于字符串的另一个重头戏，也就是排序了。因为当我们有了数据之后，要怎么让数据可以更容易的让人可以进行检索，或如何进行有效的整理就是非常重要的议题了，而排序正是这些议题的第一门课程。所谓的排序其实主要在进行的也就是「比较」，「交换」的工作，因此我们可以先从Perl如何交换两个变量的值来

看起。
在传统的方式，或其它程序语言目前的实作方式还是如此，也就是使用另一个变量来作为暂存的变量。例如我们如果想要把$a跟$b两个变量里面的值进行交换，那么可能的作法也许会是这样：

```
$tmp = $a;    # 先把$a 的值放进暂存变数
$a = $b;      # 把$b 的值指定给$a
$b = $tmp;    # 从$tmp 中取得$a 原来的值，并指定给 $b
```

可是在 Perl 当中，我们就可以轻松一些了。我们如果要交换两个变量的值，只需要使用这样的方式就可以了：

```
($a, $b) = ($b, $a);
```

这样看起来好像有点差距，可是又相差不大，部过一但变量够多，你利用其它方式可能只会让自己变得头昏脑胀，不然你试着自己弄一个四个变量的状况，然后用原来的方式写写看，我想总还是很难比这样看起来更方便了吧：

```
($a, $b, $c, $d) = ($b, $c, $d, $a);
```

能够轻松的交换变量内的值之后，我们如果利用排序的结果来决定是否要把两个正在进行比较的变量值交换，那么最后就可以完成整个串行的排序。如果你学过某些相关的内容，应该会觉得非常熟悉，这似乎是某种被称为「泡沫排序法」的方式。当然，你可以使用其它在数据结构那堂课中所学的其它排序，好吧，不过暂时先忘了这些课本上的东西。我们先来看看最基本的排序方式：

```
sub my_sort {
  my ($a, $b) = @_;
  ($a, $b) = ($b, $a) if ($a > $b);
  .......                  # 继续其它运算
}
```

利用比较，交换的方式，我们似乎完成了一个简单，可以用来排序的副例程。不过既然每次排序我们都需要这样的东西，那么 Perl 很显然的，应该会有更简易的方式。于是我们发现了一个新的运算符：<=>。
有人称这个符号为宇宙飞船符号，确实是有几分像，那么它有什么便利性呢？我

们实际利用这个符号来进行排序吧。这里还有一个很大的特点，当我们在进行比较时，通常会定义两个变量来表示正在进行比较的值，很多时候我们都用$a 跟$b 来代表这两个值。只不过如果每次我们都需要这两个变量，那不是很累人吗？Perl 也非常体谅我们打字的辛苦，所以$a 跟$b 已经被设为 Perl 排序时的内建变量。意思也就是说，以后如果你在 Perl 中要进行排序，你不需要自己另外定义这两个变量。

```
my @array = (6, 8, 24, 7, 12, 14);
my @ordered = sort { $a <=> $b } @array;
print @ordered;
# 结果变成 6, 7, 8, 12, 14, 24
```

你可能会很好奇，这样的方式难道不能直接用 sort 来作吗？我们之前学过，直接使用 sort 这个函式来对数组进行排序。所以现在的状况应该可以使用同样的方式来进行排序。那么何不来试试呢？

好啊，这已经让我快要一头雾水了。因为上面的例子实在让人很想改写成这样：

```
my @array = (6, 8, 24, 7, 12, 14);
my @ordered = sort @array;
print @ordered;
# 这次输出 12, 14, 24, 6, 7, 8
```

聪明的你可能已经看出排列出来的结果了，没错，sort 预设会使用字符串排列的方式，这时候，我们应该先提示一下 sort 的语法：

```
sort SUBNAME LIST   # 你可以使用副例程
sort BLOCK LIST             # 或使用一个区块
sort LIST                   # 这是我们一开始说的方
式
```

因此，如果你没有指定区块或是副例程，Perl 预设会使用字符串的方式去进行排序，也就是我们第二次看到的结果了。那么如果我要强制 Perl 使用字符串比对，或是针对字符串进行比对，那应该怎么写呢？你可以参考另一个和<=>相对应的运算符，也就是'cmp'，这也就是比较的意思。让我们直接来试试这样的比较方式吧：

```
my @array = (6, 8, 24, 7, 12, 14);
my @ordered = sort { $a cmp $b } @array;
print @ordered;
# 这次还是输出 12, 14, 24, 6, 7, 8
```

没错吧，果然和我们第二次只使用 sort 的结果是一样的。特别要注意的就是'cmp'这个东西，如果你要进行字符串的排序，可不能使用宇宙飞船符号。另外，我们还可以直接进行递减的排序，而且非常简单，我们直接利用第一个例子来试试吧：

```
my @array = (6, 8, 24, 7, 12, 14);
my @ordered = sort { $b <=> $a } @array;
print @ordered;
# 递减排序： 24, 14, 12, 8, 7, 6
```

其实一但可以利用区块或副例程来进行独特的排序方式，我们可以玩出不少其它的花样。例如你可以对杂凑进行排序，或是比对多个值来进行排序。其中杂凑的排序是非常常用的。尤其我们知道，杂凑的安排是依据系统计算出存取的最佳化方式，因此大多数的时候，我们拿到一个杂凑通常是没有什么顺序性。要能够对于其中的键或值排序都是非常重要的，而透过 sort 的方式，我们就很容易做到了。

```
my %hash = (john, 24, mary, 28, david, 22);
my @order = sort { $hash{$a} <=> $hash{$b} } keys
%hash;
print @order;                              # 依序是
david john mary
```

虽然只有三行程序，不过我们还是应该来解释一下其中到底发生了什么事，否则看起来实在让人有点头晕。第一行的问题应该不大，或者说如果你第一行看起来有点吃力，那你可能要先翻回去看看杂凑那一章，至少你应该要懂得怎么定义一个杂凑，然后指定杂凑的键跟值。这里所用的方式一点也不特别，我们只是用串行来赋值给一个杂凑。最复杂的应该是第二行 (除非你觉得最后一行要印出一个数组对你而言太过困难)，我们先看等号左边，那里定义了一个数组，因为我们希望可以得到一个依照杂凑值排序过的杂凑键数组。这听来好像不难，让我们先想象一下，我们该怎么取得这样的数组呢？
首先我们应该先拿到包含所有杂凑键的数组，也就是利用 keys 这个函式取得的一个数组。拿到这个数组之后，我们就可以来进行排序了。排序的重点在于区块内的那一小段程序。我们还是使用了 Perl 预设的两个变量，也就是$a 跟$b，分

别代表从数组(keys %hash)拿出来准备比较的两个数值。部过我们并不是直接对变量$a，$b 进行比较，而是以他们为键，而取的杂凑值来进行排序。

## 12.5 多子键排序

很多时候，我们会希望排序的根据不单单只是一个单纯的键值，例如在刚刚的例子中，如果我们希望当排序时，在遇到年龄相同的时候，还能以名字排序，那么我们就会需要多子键排序。另外还有非常常见的就是网络上经常看到的IP，我们如果要按照顺序将IP排序，那么这是没有办法依照正常的方式来进行排序的。例如我们看到这些IP：

```
140.21.135.218
140.112.22.49
140.213.21.4
140.211.42.8

依照正常字符串排序之后会变成：

140.112.22.49
140.21.135.218
140.211.42.8
140.213.21.4
```

这看起来实在不太对劲，因为是藉由字符串的关系，所以 21 被排在 112 的后面。所以我们想要作的其实是把每一个部份都拆开来，然后进行数字的比对。所以我们可以这么作：

```
#!/usr/bin/perl -w

use strict;

my @ip = ("140.21.135.218", "140.112.22.49",
"140.213.21.4", "140.211.42.8");
my @order = sort ipsort @ip;     # 直接叫用副例
程
print "$_\n" for @order;


sub ipsort {
```

```
    my ($a1, $a2, $a3, $a4) = $a =~
/(\d+).(\d+).(\d+).(\d+)/;       # 分为四个数字
    my ($b1, $b2, $b3, $b4) = $b =~
/(\d+).(\d+).(\d+).(\d+)/;
    $a1 <=> $b1 or $a2 <=> $b2 or $a3 <=> $b3 or
$a4 <=> $b4;         # 进行多子键排序
}
```

这个程序的重点在于两个部份，第一个部份是直接叫用副例程进行排序。所以我们看到在这里，我们呼叫了副例程 ipsort 来帮我们进行多子键的排序部份。而且我们一样可以直接在副例程之中使用预设变量$a，$b。在我们把排序的程序放进副例程之后，我们就开始进行 ip 的拆解工作，利用正规表示式把每一个 ip 都拆解成四个部份。所以我们就分别有了$a1…$a4 以及$b1…$b4 这样的子键。然后利用子键来进行排序，并且利用 or 来作为是否进行下一个子键排序的关键。因为宇宙飞船符号的比较会传回-1, 0 或是 1，因此如果是 0 就表示两者相等，于是继续比对下一个子键。利用这样排序之后，我们就可以得到这样的结果：

```
140.21.135.218
140.112.22.49
140.211.42.8
140.213.21.4
```

习题：

1. 让使用者输入字符串，取得字符串后算出该字符串的长度，然后印出。
2. 利用 sprintf 做出货币输出的表示法，例如：136700 以$136,700，26400 以$26,400 表示。
3. 利用杂凑%hash = (john, 24, mary, 28, david, 22, paul, 28)进行排序，先依照杂凑的值排序，如果两个元素的值相等，则依照键值进行字符串排序。

# 13. 模块与套件

Perl之所以可以这么受到欢迎，除了本身有许多专为懒人设计的语法以及相对于其它程序语言，更接近自然语言的用法之外，丰富的模块资源更是让Perl能持续维持高人气的主要因素。而数以千计的模块不但能吸引住众多的Perl开发者，更能让这些开发者贡献出其它的模块，如此一来，便会造成「网络效应」，而持续让更多人愿意投入Perl的怀抱。

对于Perl的使用者来说，如果你不会使用各式各样的模块，那么你对Perl的使用率可能不到十分之一。因此能够写Perl的人，可能也因此对于程序代码重用的部份乡对于其它程序语言的程序设计师有更深的感受。当然，大多数的Perl程序设计师总是需要学会如何开始使用模块，紧接着便会了解如果善用模块，找到自己所需要的资源。再下一阶段就是如何写出自己的模块。

可惜很多程序设计师，或是项目管理员对于这方面并不重视，他们总是只看着手边的东西。而不肯多花时间把手边的程序代码整理成模块，很多人不相信自己还会重新用到这些程序代码，或者不认为同样的这些程序代码如果整理成模块，可以让许多人节省更多的时间。当然，对于这些程序设计师或管理项目的人而言，更不用提要怎么进行好一个项目，版本控制，分支，合并了。 (注一)

不论如何，一但你开始使用Perl，你应该就必须有足够的能力去使用各式各样的模块。而且还必须了解模块与套件的结构，因为你可能会需要对于你使用的Perl模块进行除错的工作，虽然这些事情未必经常发生。不过你从这里开始，就会开始慢慢的学会如何写好自己的模块。所以现在就开始来进入的世界吧！

## 13.1 关于程序的重用

我们之前提到过可以节省程序代码写作的时间，大幅提升程序可重用性的方式就是副例程。可是如果你没有好好管理你的程序代码，等到下一次你需要同样的函式时，你还是必须重写一次。当然，很多人这时候就会利用复制，贴上的方法。把原来的副例程复制到新的程序之中，这样一来，就可以再度使用同样的副例程了。

可是利用这样的方式还是会有一些问提存在，就像我们在描述副例程时所说的，当你一再使用复制，贴上这样的方式时，很容易就会造成管理上的问题。因为你还是没有办法统一的管理一个套件，让你以后只要修改模块，接着就可以一次修正使用相同模块的所有程序。

而不会像你使用拷贝，贴上的方式，你一但找到副例程的一个错误，就必须同时修正所有使用这个副例程的程序，当然还可以能因为你忘了某个程序中忘了修改而让自己踩到地雷。所以既然你都已经使用了副例程，除非你确定某些副例程只会在目前的程序使用，否则他们都有机会成为模块。

另外，当你开始使用独立的模块之后，你更需要做好档案的管理，因为你可能会公开给所有的人使用，就像CPAN (Comprehensive Perl Archives Network，Perl综合典藏网) 上面所有的模块一样，或是开放给公司内部使用。不管如何，你的程序一但公开释出之后，就应该考虑使用者的使用性以及如何更新版本，修正错误的问题。这其实是非常严肃的问题，因为一但没有办法做好程序代码的管理，很容易会加重负担，反而增加管理成本。这个部份对于部份许多公司或个人来说，

都还需要更深的着墨，我们应该在附录用一些篇幅，介绍这个部份，虽然它们并不属于Perl的范围之内。

## 13.2 你该知道的 CPAN

也许你不太同意我们刚刚所说的部份，不过如果你确定你要开始使用Perl来解决生活，或工作上的问题时，你大概很难不先知道，而且学会如何使用CPAN。

刚刚说了，CPAN就是Perl典藏网，可是葫芦里到底卖的是什么药呢？其实CPAN上主要的就是上面的许多模块，目前已经有好几千个模块在CPAN上。所以你可以在上面取得这些模块的原始码，文件，有些模块可能还提供其它的二进制档案，让你在没有办法编译时也可以使用。

目前的CPAN，上面已经充满了各式各样的模块，大部份的需求几乎都可以有现成的模块来解决你的问题，或七成以上的问题。当然，如果你想要在CPAN上找到符合需求的模块确实需要花点功夫，因为浩瀚CPAN大海，既然有各种各样的模块，虽然你可以使用搜寻的功能，可是如果你完全没有头绪，恐怕是需要一点时间来适应CPAN这个图书馆了。不过接下来的章节，我们会在各个部份介绍相关使用的模块，这些模块很大部份的时候几乎都是你在写相关程序时一定会用到的模块。另外，我们也会在附录整理五十个在CPAN上非常有用的模块，这个清单将会包括很多领域的部份模块，相信可以作为一个参考清单及介绍。

接下来，也许你终于花了一些时间找到了一个符合你需求的模块，那么你在开始使用之前，必须先安装这个模块。最传统的方式，你可以下载这个模块的原始码，然后试着自己编译。这时候，你可以先到CPAN网站上搜寻你需要的模块。(图一)，然后下载原始档，接着就开始编译，像这样。(图二)

不过这样确实有点辛苦，尤其Perl的使用上，我们会经常大量的安装模块，如果每次都要这样子来一步一步来就会显得相当吃力。因此我们必然需要更方便的工作来帮助我们完成安装模块的工具，而在你安装完Perl之后，其实Perl就会给你一个叫做cpan的工具程序，而他正是帮助你完成大量安装Perl模块的好帮手。很多时候，你可以直接进入cpan的命令列中。(图三)

不过如果你在Win32 的作业平台上，CPAN对你的帮助显然就小的多了，尤其当大多数的Windows使用者并没有安装相关可以提供编译这些模块的编译器，那么他们所需要的，应该是能够提供Windows平台上的二进制文件安装了。当然，这时候你应该使用由ActivePerl所提供的ppm程序，以便让你可以容易的安装Perl模块。

cpan是目前的Perl版本内附的CPAN模块安装程序工具，不过下一个阶段的取代性程序也正在发展当中，而且目前也已经相当稳定，不久之后将会取代cpan，成为Perl内附的工具程序，这个新的工具就是cpanplus。就像这个模块的名称，他正是cpan的加强版，例如他可以帮你确认目前机器上安装的模块版本，以及该模块的最新释出版本，提醒你该升级。你可以轻松的解除某个模块的安装或是下载某个模块，解除安装等等。(图四)

很可惜，截至目前为止，cpanplus还是只能在Unix的环境下执行，主要的问题还是因为编译器的问题，就如我们刚刚说的，绝大多数的Windows系统中并没有编译器，所以如果你的Windows环境下能够装起合适的编译器，当然还是可以使用这些方便的工具。另外，Mac OS X的系统预设也并没有安装编译器，所以如果你希望使用cpan/cpanplus的话，就必须安装相关的套件。

## 13.3 使用CPAN与CPANPLUS

如果你只是想要学习Perl的语法，那么也许你不需要使用CPAN，不过当你要开始利用Perl来完成某些工作，或作业。而且完全不想自己重新开始，那么学会使用CPAN/CPANPLUS就是一件非常重要的工作了。正如我们所说的，如果你在Linux/*BSD的操作系统中，一般而言，你一但装好了perl，核心安装也会自动把CPAN这么模块安装进去。所以也就有命令列的执行程序"cpan"，如果你是第一次执行cpan，会需要作一些设定，以确定你计算机内的环境以及各种程序的位置。完成设定之后，你会看到提示符号，就像这样：

```
cpan>

这就表示你可以开始使用 cpan 了。
```

当然，你可以利用 help 取得完整的 cpan 使用说明，不过我们还是就一些常用的功能进行介绍。最常用的大概就是 install 了，几乎还无疑问，你可以利用 install 来安装需要的模块。所以如果你想安装 CPANPLUS 这个模块，就只需要这么作：

```
cpan>install CPANPLUS
```

利用 cpan 安装模块，它会帮你进行完整的步骤，也就是一般我们手动从原始码安装时会进行的步骤：

```
perl Makefile.PL
make
make test
make install
```

所以有时候你会在使用 cpan 安装的过程中遇到测试不过或其它状况，这时候你可以使用强迫安装的方式来要求 cpan 进行强制安装。使用的方式也非常简单，就只要在 install 时加上 force 的选项：

```
cpan>force install CPANPLUS
```

另外，如果你只想下载某个模块，而不想进行编译或安装，那么就使用 get 指令。

```
cpan>get CPANPLUS
```

类似的指令则有 make，test，install，clean 等等，这些都是针对某个模块进行安装相关的操作。而如果你想查询某个模块的相关数据，你可以使用 i 这个指令，就像这样：

```
cpan> i CPANPLUS
Strange distribution name [CPANPLUS]
Module id = CPANPLUS
    CPAN_USERID  AUTRIJUS (Autrijus Tang
<autrijus@autrijus.org>)
    CPAN_VERSION 0.049
    CPAN_FILE
A/AU/AUTRIJUS/CPANPLUS-0.049.tar.gz
    MANPAGE     CPANPLUS - Command-line access
to the CPAN interface
    INST_FILE
/usr/local/lib/perl5/site_perl/5.8.4/CPANPLUS
.pm
    INST_VERSION 0.049
```

我们可以知道模块的名称，版本，作者等等各种信息。另外，i 也可以使用正规表示式，所以如果你使用

```
cpan>i /CPANPLUS/
```

就会传回一串内容有关 CPANPLUS 的模块。不过因为使用 i 这个指令会传回所有关于模块，作者，散布或集结而成的模块(例如 Bundle::CPAN)等等信息。而如果你想单纯的搜寻其中一个部份，就可以使用作者(a)，模块(m)，散布(m)跟集结(b)。我们继续用 CPANPLUS 作例子：

```
cpan> d /CPANPLUS/
Distribution
A/AU/AUTRIJUS/CPANPLUS-0.049.tar.gz
Distribution
B/BD/BDULFER/CPANPLUS-Shell-Tk-0.02.tar.gz
```

```
Distribution
K/KA/KANE/CPANPLUS-0.042.tar.gz
Distribution
M/MA/MARCUS/CPANPLUS-Shell-Curses-0.06.tar.gz
4 items found
```

这样应该清楚多了，这表示我们只要搜寻内容有 CPANPLUS 的相关散布档案，而不想要包山包海的把所有相关信息都收集起来，当然你也可以只使用 m 或 a 来取得相关的信息。
接下来，还有一个你也许会常用到的功能，也就是"reload index"。CPAN 上的模块几乎无时无刻不在更新，所以你的计算机里面的各种数据其实会需要经常更新，这时候你只需要利用这样的方式，CPAN 会自动取网络上找到最新的内容索引：

```
cpan> reload index
Fetching with LWP:

ftp://ftp.perl.org/pub/CPAN/authors/01mailrc.
txt.gz
Going to read
/Users/hcchien/.cpan/sources/authors/01mailrc
.txt.gz
Fetching with LWP:

ftp://ftp.perl.org/pub/CPAN/modules/02package
s.details.txt.gz
Going to read
/Users/hcchien/.cpan/sources/modules/02packag
es.details.txt.gz
  Database was generated on Tue, 11 May 2004
09:34:08 GMT
Fetching with LWP:

ftp://ftp.perl.org/pub/CPAN/modules/03modlist
.data.gz
Going to read
/Users/hcchien/.cpan/sources/modules/03modlis
t.data.gz
Going to write /Users/hcchien/.cpan/Metadata
```

最后，如果你要离开，就请使用 quit，CPAN 会移除某些暂存盘，让你平安的回到地面。

CPAN 在 perl 的使用上确实是非常方便的，不过有些地方还是让人有点感觉不够，例如你如果顺手想要安装 CPANPLUS，那么你可能需要进入 cpan 的命令列下，或是利用 Perl 的单行模式执行这样的指令：

```
>perl -MCPAN -e"install CPANPLUS"
```

这倒还好，虽然指令长了一点，不过总是一次可以解决。只是我们都知道，因为在 Perl 中使用其它各式各样的模块是缩短开发时程跟降低成本的好方法，所以大部份模块其实都还是用了其它模块，我们就说这个要被安装的模块必须依赖其它某些模块，可是在 CPAN 里却没办法帮我们完成这些相关性的安装工作。所以如果你安装某一个模块却发现它必须依赖其它模块时，CPAN 会发出错误讯息给你，然后就停摆了。当然在我们的期望中，如果它可以「顺便」帮我们把其它需要的模块也安装进去，那显然会减少许多手动的工作。因此在这样的需求下，CPANPLUS 也就因应而生了。
CPANPLUS 在使用上有一些不同于 CPAN 的地方，例如你可以直接在 shell 下面执行 CPANPLUS 的安装手续，就只要这么打：

```
>cpanp -i SVK
```

接着，神奇的事情就要发生了，当我们安装一个模块，而它所依赖的其它模块并不存在时，系统就会自动询问使用者是不是要同时安装相关的模块，就像这样：

```
[root@Apple]# cpanp -i IO::All
CPANPLUS::Shell::Default -- CPAN exploration
and modules installation (v0.03)
*** Please report bugs to
<cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS::Backend v0.049.  ReadLine
support suppressed in batch mode.

Installing: IO::All
Warning: prerequisite Spiffy 0.16 not found. We
have 0.15.
Checking if your kit is complete...
Looks good
Writing Makefile for IO::All

Spiffy is a required module for this install.
```

```
Would you like me to install it? [Y/n]:
```

接下来，CPANPLUS 也有自己的终端机，你只需要用"cpanp"就可以进入：

```
[root@Apple]# cpanp
CPANPLUS::Shell::Default -- CPAN exploration
and modules installation (v0.03)
*** Please report bugs to
<cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS::Backend v0.049.
*** ReadLine support available (try 'i
Term::ReadLine::Perl').

CPAN Terminal>
```

CPANPLUS 还有一个非常好用的功能，就是列出目前系统中还没更新的模块清单，所以你只要进入 cpanp，然后使用"o"就可以得到系统中需要更新的模块，不过这通常需要花费一段时间：

```
CPAN Terminal> o

   1    3.04      3.05   CGI
LDS
   2    1.06      1.08   Digest
GAAS
   3              1.05   Encode::CN::HZ
DANKOGAI
   4    0.56      0.59
ExtUtils::AutoInstall
AUTRIJUS
   ......
   ......
```

另外，还有跟 CPAN 比较不同的部份在于你使用 CPANPLUS 可以解安装某个模块，也就是使用"u"这个选项，也就是代表 uninstall 的意思。你还可以利用 cpanp 进行本地端的 perl 模块管理，例如使用 e 来新增某些目录倒你自己的@INC 中。至于在 cpan 中使用 reload index 的工作，在 cpanp 中只需要按下 x 就可以了。其实在不久之后，CPANPLUS 将取代 CPAN 在 Perl 核心中的地位，因此现在开始熟悉 CPANPLUS 似乎也不是什么坏事。

## 13.4 使用模块

看了一堆长篇大论，我们终于要开始写程序了，或是你根本直接跳过前面的叙述来到这里。不管如何，我们假设你已经学会怎么装模块了，而现在该来学习怎么使用这些已经存在你硬盘中的模块了吧！
先来用一个简单的模块吧，这个模块对于将来你在写程序的除错时会有相当的帮助。就像下面这一段程序代码所写的样子：

```
use strict;
use Data::Dumper;   # 说明我们要使用的模块名称

my %hash = ("john", 24, "mary", 28, "paul", 22,
"alice", 19);
print Dumper(%hash);       # 这就是模块里面提供的
函式
```

当我们决定要使用某个模块时，我们就用关键词"use"，也就是「使用某某模块」的意思，还真是口语化。接下来，你就可以使用模块内提供的函式了。所以我们在接下来的地方，定义了一个杂凑，是包含了名字，以及他们的年纪。程序最后，我们用了 Dumper 这个函式来印出杂凑 hash 里的所有内容，而 Dumper 其实就是 Data::Dumper 这个模块所提供的函式。
最基本，对于模块的使用大概就是这个样子。当然，这也是最简单的方式。在你使用 use 来指定你所要使用的模块时，Perl 会加载模块，并且把模块汇入。而当你在使用 use 这个指令时，其实还可以指定汇入模块中的某些函式。例如我们找到一个模块，叫做 Cwd，它主要的功能是可以帮助我们找到目前的路径，如果你是一个 Unix 的使用者，那么他就非常接近 ped 这个 Unix 指令。这个模块提供了不少函式，不过我们有时候并不想全部用到，所以你虽然可以像原来的方式，这么使用它：

```
use Cwd;

my $dir = cwd();

print $dir;  # 印出目前的目录
```

另外一种方式则是在 use 后多加一个参数，用来表示要汇入的函式。就像这样：

```
```

```
use Cwd qw(abs_path);              # 我只想用
abs_path

my $dir = Cwd::abs_path;           # 这时候，要加上
完整的模块名称

print $dir;
```

在你使用模块的时候，你也许还要注意另外一件事，也就是 Perl 能不能正确的加载你的模块。大多数的时候，你会从 CPAN 安装模块，这些状况下其实并不会有太大的问题，因为系统都会帮你安排好你的模块所应该摆放的位置。可是如果你利用其它方式取模块，或准备安插自己的模块时，有时候却会因为 Perl 找不到你指定的模块而无法进行加载。因为对于 Perl 来说，它有一个模块加载的路径，而这些路径其实是被纪录在@INC 底下的，所以如果你没有安装某个模块，或是你的模块并不在 Perl 的加载路径内的话，那么它就会告诉你无法找到这个模块。
因为@INC 也是 Perl 内建的数组变量，所以如果你想要知道系统中的@INC，你也可以直接用这样的方式印出来：

```
print "@INC\n";      # 别怀疑，只需要这样
```

或者你直接在命令列底下使用 perl -V 也可以看到相关的内容。不过这样会输出非常多的内容，只怕在这里列出来会占去一大页，所以还是由各位自己试试看吧。当然，如果你的模块地处边缘，没有办法被@INC 含盖进去的话，也不用担心，你可以自己指定程序内使用的模块路径。其中一种方式是直接修改@INC 变量，只是这个部份牵扯到关于模块加载的时机，也就是如果 Perl 在编译的时候无法找到指定的模块，它就会开始不高兴，然后大声哭闹。所以你在程序代码执行的部份修改了@INC 这个变量对于停止 Perl 的问题并没有太大的帮助 (注二)。既然解决方法不只一种 (There is more than one way to do it)，我们显然可以试试其它办法。有另外一个方式，也就是直接使用 use 指令，就像这样：

```
use lib "/home/hcchien/pm/";

use Personal;
```

其实你还是有其它各式各样的方式，不过在这里我想还是这样就足够了。至少这也是目前我遇过最常用的方式，如果某一天你觉得这样的方式已经不敷使用，相信你已经有能力找到更多方式来帮助你解决问题了。

## 13.5 开始写出你的套件

经过一番努力，你应该要慢慢熟悉怎么使用cpan/cpanplus从CPAN安装各式各样的模块了 (注三)。接下来，你应该盖上这本书，然后开始写Perl程序了。或是你已经可以自己写一些程序，然后拿来解决一些日常生活上的问题，或工作上的需要。接下来，你已经准备把手上已经写好的程序代码集结起来，先把它们集结出一个套件吧。

我们刚刚一直在讨论怎么使用CPAN上丰富的模块资源，不过现在我们应该回过头来看看模块的组成元素。其中非常重要的一个部份，也就是套件。套件其实就是你在写Perl程序时的零件箱，也就是你可以放进一堆可以重新使用的小零件，那么在程序里面，你就可以直接拿出来，兜起来，很快就可以写好自己的程序。

我们先来用个简单的例子吧，虽然大多数的工作已经有了模块可以让我们使用，可是要让大家简单明瞭的看懂套件的写法，我们还是来看看下个这个程序吧：

```
#!/usr/bin/perl -w

use strict;

my @grades = (67, 73, 57, 44, 82, 79, 67, 88, 95,
70);;

my $adv = adv(@grades);                 # 叫用
adv 这个副例程
print $adv;

sub adv {
    my @input = @_;
    my $total;
    $total+=$_ for (@input);      # 算总和
    $adv = $total/scalar(@input);  # 求平均
}
```

这程序相当简单，我们在主要的程序部份看到两个重点，第一个就是定义一个数组，在这里我们定义了一个关于成绩的数组，里面放满了一堆学生的成绩。第二个重点则是在程序里面叫用了 adv 这个副例程。至于在副例程 adv 里面，我们取得了主程序传来的成绩数组，紧接着计算总和，然后算出平均。接下来，我们希望开放这个方便的副例程给其它程序使用，所以我们必须把它放进套件中，就像这样：

```

```

```
package Personal;  # 套件的开始

sub adv {
    my @input = @_;
    my $total;
    $total+=$_ for (@input);
    $adv = $total/scalar(@input);
}

1;            # 回传一个真值
```

于是我把它储存为另一个档案，叫做 Personal.pm，接下来我们就可以开始使用这个套件了。也就像我们之前所说的方式，我们还是使用 use 这个指令。所以原来的程序就会变成这样：

```
use strict;
use Personal;                    # 现在我们直接
使用这个套件了

my @grades = (67, 73, 57, 44, 82, 79, 67, 88, 95,
70);;

my $adv = Personal::adv(@grades);      # 然后呼
叫套件的 adv
print $adv;
```

乍看之下，好像不过就是把原来的程序切成两半，实在一点也没什么特殊的。不过事实当然绝非如此，现在我们假设又要写另一个程序了，这次要算的是一大群人的平均年龄。没错，我们又需要用到算平均的函式了。所以我们写了这样的程序：

```
#!/usr/bin/perl -w

use strict;
use Personal;

my %member = (  'john' => 22,
              'mary' => 42,
              'paul' => 27,
              'alice' => 19,
```

```
                'josha' => 37  );

my @age = values %member;  # 取出所有人的年龄，放入数组

my $adv = Personal::adv(@age);  # 还是使用了 adv 这个函式
print $adv;
```

这样的用法应该不难理解，我们现在有了自己的套件档案，也就是 Personal.pm。接下来，我们只要再度需要使用 adv 的部份，就只需要加载 Personal.pm 就可以。当然，使用者也可以自己不断加入新的函式，来让自己的函式库越来越丰富。一般来说，我们都以.pm 来作为分辨一般 Perl 程序与套件的方式。而我们在套件的一开始，则是以关键词 package 来表明这个套件的名称，就像我们刚刚写的方式：

```
package Personal;
```

当我们开始使用套件的时候，其实套件内部就像是另一个独立的程序一样，你在使用一个套件时，并没有办法提供一个全域变量来供所有人使用。而且这当然是完全合理的逻辑，否则不是会让人一团混乱吗？不过在套件内部确实也有一些机制来保持一个专属的空间，避免套件与套件之间，套件与程序之间，所有的变量名称，副例程变成像一盘意大利面一样，全部打结混在一起。因此，我们在将自己的套件完成到某个程度之后，使用套件名称作为档名储存起来，就可以开始使用了。以刚刚的套件为例，我们就把他储存为 Personal.pm。
Perl 使用的方式就是所谓的「符号表 (symbol table)」。一般没有被定义套件名称的的部份，其实都是被委以 main 这个套件，当然你也可以随时使用 package 来定义套件名称。套件的有效范围是从你使用 package 宣告开始，一直到这个区块的结束，或是另一个 package 的宣告。所以这也就是某个套件的有效命名空间，也就是说在这有效范围内的变量命名都会是属于这个套件的命名空间下。用简单的表示方式就会是：

```
$PACKAGE::$VARIABLE
```

这样的方式其实就让人比较可以理解为什么我们会使用类似 Cwd::abs_path 这样的方式来呼叫某个副例程。而且你也可能偶而会发现这样的程序错误：

```
```

```
Undefined subroutine &main::adv called at ch3.pl
line 14.
```

也就是 Perl 在 main 这个命名空间下并没有找到 adv 这个副例程。

而就像很多情况看到的，我们可以使用包含的关系来使用套件的命名空间，所以你当然也可能看到类似这样的方式：

```
$PACKAGE1::$PACKAGE2::$VARIABLE
```

套件与模块的使用对于大多数的程序写作都是非常重要的议题，而对于 Perl 来说更是如此，因为他可以让你大量减少程序写作的时间。至少在看完这一章之后，你应该可以开始学习使用 CPAN 上广大的资源。

习题：
1. 试着在你的 Unix-like 上的机器装起 CPANPLUS 这个模块。
2. 还记得我们写过阶乘的副例程吗？试着把它放入套件 My.pm 中，并且写出一个程序呼叫，然后使用这个副例程。

注一：许多项目管理的方式就是采用拷贝，复制的方式，和我们所提的方式显然大异其趣。
注二：我们确实可以在程序代码中要求 Perl 在编译期间就修改变量@INC，不过我们不打算在这里把这件事情搞的这么复杂。
注三：千万别小看这部份，如果没有 CPAN，你写 Perl 会感觉太辛苦了。

# 14. 参照 (Reference)

对于一个刚开始使用Perl的使用者来说，要深切的了解参照确实是非常困难的一件事。可是如果不使用参照，你就会发现有很多时候，事情会变得相当复杂。因此大略的了解Perl参照的使用实在是相当重要。不过对于许多从事Perl教学的人来说，Perl的参照是既复杂又困难的事，因此在大约 20 小时的教学中，并不容易包含参照的使用。所以有些给Perl入门使用者的书籍也会避开这个题目，而在更进阶的书籍再专文介绍。可是我以为如果失去了参照的使用，很多相当方便的使用方式都无法被实作，让人无法领略Perl的方便性。可是要如何以简单的方式概略的介绍Perl的参照也是另外一项复杂的议题。不过虽然如此，我们还是来尝试以比较浅显易懂的方式来介绍参照的常用方式。

## 14.1 何谓参照

就像C程序语言造成许多程序设计师的困扰一般，参照之于Perl也有类似的效果。当然造成这个状况的原因大概也是因为参照的抽象观念也足够跟C的指标媲美。其实这也许只是危言耸听，我们应该想办法让指标变得更容易使用，而且根据Perl的 80/20 定律，我们只需要学会其中的百分之二十，就可以应付百分之八十的状况。

「参照」，其实跟所谓的指标在意义上是非常接近的，也就是某个变量指向另外一个变量。比如我们有一个数组变量像是这样：

```
my @array = (1...10);
```

那么我希望使用一个纯量变量$ref 来表示@array 这个数组时跟怎么办呢？这时候，我们只要表明它是一个数组，而且它被储存的位置在那里。我们就可以顺着这个线索找到@array 这个数组，而且得到他的内容。所以我们可以用这样子来表示$ref 变量：

```
ARRAY(0x80a448)
```

这样看起来就非常清楚，我们有一个数组，地址在 0x80a448。所以我们可以藉由这样的数据取得@array 这个数组的详细数据。这就跟我们在 Unix 系统下使用档案连结的方式有点接近，我们透过某个连结信息来找到被连结的档案。

而在 Perl 里面，所谓的参照，其实确实是建立了另外一个符号，就像储存一般的变量一样，不过这次我们得到的是一个纯量变量。Perl 确实可以使用纯量变量来指向任何其它的数据结构，也就是另一个纯量变量，数组或杂凑，就像我们刚刚看到的样子。

我想我们可以用一个比较简单的方式来解释参照，例如你们家的成员可以组成一

个数据结构的型态(你可以使用数组或杂凑，看你想要储存的数据内容而定)，于是如果想要用一个纯量的数据型态来取得家里成员的内容，那么我们也许可以用门牌号码来代表(当然，如果是有户政单位的作业疏失造成门牌号码重复等各种错误可不在我们讨论的范围内)。

## 14.2 取得参照

当然，你其实大可不必担心怎么找出参照的地址，因为这个部份可以由Perl代劳。我们来看看怎么取得参照：

```
my @array = (1...10);
my $ref = \@array;  # 取得数组@array 的参照

print $ref;
```

当然，这个程序的执行结果就会看起来像是我们刚刚写的样子，只是地址会有所差异。接下来，我们也可以来看看怎么利用一个参照来取得被参照的数据结构内容。就以刚刚的例子来看，我们定义了一个数组@array，然后利用反斜线(\)来取得@array 的参照。那么我们要怎么取得$ref 所参照的@array 内容呢？其实我们只需要这么作就可以了：

```
print @$ref;        # 利用参照找回数组
```

很显然的，我们可以利用纯量变量来取得另一个纯量变量，或是数组，或是杂凑的参照。当然，取得的方式都是类似的，所以我们只要这样：

```
$scalar = "1...10";
@array = (1...10);
%hash = (1...10);
$scalar_ref = \$scalar;
$array_ref = \@array;
$hash_ref = \%hash;
```

这样看起来应该就清楚多了，不过很多人可能还是没办法想象这样的参照能有什么很大的用途。其实参照很重要的用途之一，就是在增加数据结构的弹性，或者你也可以说它是增加数据结构的复杂性。我们先来看看一个实例：

```
@john = (86, 77, 82, 90);
@paul = (88, 70, 92, 65);
@may = (71, 64, 68, 78);
```

我们现在假设有一个考试，总共考了四个科目，上面的数组是这三个学生每一科的成绩。可是我们如果要针对某个科目，一次取得每个学生的成绩就会显得很麻烦，因为我们可能需要$john[0], $paul[0], $mat[0]这样的方式。因此在其它程序语言的实作方式则是使用所谓的「多维数组」，例如你可以定义一个数组，那么它的结构会看起来样这样子：

```
grades[0][0] = 86;
grades[0][1] = 77;
grades[0][2] = 82;
.....
.....
grades[2][2] = 68;
grades[2][3] = 78;
```

很可惜，Perl 的数组并不接受这种方式的定义，也就是并没有提供所谓的多维数组的概念。可是却不能依此推论 Perl 的数据结构太过简陋，因为透过参照的方式可以让 Perl 的三种数据结构都获得最充分的运用。所以我们来看看该怎么使用 Perl 的参照方式来实作多维数组。
首先我们还是有三个数组，分别代表三个学生的各科成绩，就像刚刚的例子一般。接下来的重点就是把这三个数组的参照放进另外一个数组中，就像这样：

```
@grades = (\@john, \@paul, \@may);
```

所以现在看来，@grades 这个数组中其实已经包含了@john，@paul，@may 三个数组了。这样就可以实作出一个多维的数组。各位应该可以想象整个数组中夹杂着数组参照的状况，其实我们可以用图来观察，也许会比较容易理解。

<<图一>>

这个图里面，我们其实并不是忠实的表达出数据在计算机内存中储存的形式，不过在概念上却可以清楚的看出利用参照来表达复杂数据结构的方法。也就是达到过去我们曾经在其它个种程序语言中所使用的多维数组的方式。当然，在实际的使用上，如果我们想要达到过去利用其它语言做出来的多维数组的，还是需要一点点小小的转换，因为我们必须不只一次的在数组中使用数组参照，如此一来，

如果要取出最后一层的数组值就会让人有点头痛。而不像过去我们使用多维数组的方式，如果有多维数组，我们几乎就只要这么写：

```
array[3][2][4];
```

既然如此，那么为什么 Perl 不直接给我们多维数组就好了呢？首先，在实际运用上，我们最常用的还是以一维跟二维数组为主，所以利用参照的方式就可以容易的达成这个需求。其次，一但利用参照的方式，我们就可以使用更有弹性的数据结构，而不单只是一个多维数组。听起来好像非常神秘，不过其实仔细想想，确实很有道理。还记得吗？我们说过数组的元素其实就是一堆的纯量所组成的，而且参照本身就是一个纯量值，只是利用这个纯量值，我们可以取得被参照的数据储存在内存的位置。然后还有一个提示，也就是参照可以用来取得各种在 Perl 中原来就有的数据结构型式(注一)。说到这里，也许你已经看出一些端倪，也就是利用参照的方式，你可以把大多数的数据都以纯量的方式来表示，因此就有各式各样的运用方式。下面就是一些我们可能会运用到的方式：

```
my @array = ('john', 'paul', 'ken');
my %info = ( 'date' => '3/27',
         'people' => \@array,
      'place' => '台北车站' );
```

这也许是某一次活动的资料，我们先取得了一个数组，其中是参加活动的人员。接下来，我们会有或动的其它数据，例如活动的日期，地点。然后我们还希望把参加的人员也一起放入活动数据中，所以我们就使用了杂凑来储存这些数据，可是杂凑中关于人员这个部份，我们是以数组参照来表示。这就是另外一个非常典型使用参照来活化数据结构的例子。当然，如果你还有力气，可以看看更复杂的例子，就像这样：

```
my @john_grades = (65, 87, 92, 77, 53);
my %john = ( id => '7821434',
         birth => '1983/11/12',
      grades => \@john_grades );
......
......
my %students = ( john => \%john,
               ......
        ...... );
```

这个例子显然可以好好来解释一下，就像魔术一般，我们用了杂凑跟数组的多次排列，把学生的资料全部堆在一起了。首先我们先拿到学生的成绩，这是一个数组，而且是最简单的数组，所有的成绩依序排列在数组中。接下来，我们要取得某个学生的资料，其中包括了他的个人成绩。因此我们使用了一个杂凑来储存学生的个人数据，而在成绩的部份，则是使用了数组参照。这样子，我们可以完整的描述一个学生的资料，接下来我们只需要另外一个杂凑来收集所有学生的资料就可以，而在这里，我们这个杂凑的每个键是学生的名字(在假设学生不会同名的状况下)，然后把刚刚取得的单一学生数据取参照，作为这个整合参照的值。
这样的写法看起来确实复杂了许多，不过这是因为我们使用了逐步讲解的方式，所以把整个过程的详细的列出来。不过很多时候我们其实会使用匿名数组或匿名杂凑的方式来表现。那么就可以让整个架构看起来容易，也清楚一些。我们把这种匿名数组/杂凑写法放在这里，也许可以让大家参考一下：

```
#!/usr/bin/perl

use strict;

my %students = ( john => { id => 'foo',                    # 这是杂凑的第一
                 tel => '11223344',
                 grades => [34, 56, 78]},
        paul => { id => 'bar',
 # 第二对键值从这里开始
                 tel => '223344',
                 grades => [44, 55, 66]},
       );
```

这样的写法显然干净许多，虽然你可能还有点不习惯，不过可以确定的是，至少你不会再看到一大堆的变量名称，而且每一个部份的相互关系也清楚了不少。当然，这时候你完全可以先忽略这个部份，不过为了让你之后回过头来看这一段程序代码时可以了解其中的奥妙之处，我们还是先说明一下这种方式的解读方式。
首先，我们定义了一个学生的杂凑，这也就是我们最终想要的数据处理方式。接下来，我们看到杂凑%students 中包含了两对的键值。其中第一对的键就是 john，而其相对应的值则是一个杂凑参照。这时候出现了第一个匿名的杂凑，他包含了三对的键值，其中的键分别是 id，tel，grades。而 grades 对应的值则是一个匿名数组的参照，这个数组一共有三个值，分别代表三个科目的成绩。因此在杂凑 %students 的第一对键值中，我们包含了两个参照，分别为一个杂凑参照与另一个数组参照。接下来的第二对键值则是以 paul 为对应的键，并且包含着一个结构相同的值。
好吧，你可以暂时先忘了这么复杂的部份，至少你暂时应该可以使用最简单的参照结构来实作一个二维数组。

## 14.3 参照的内容

我们刚刚以经学到利用各种方式得到某个数据型态的参照，并且可以把取得的参照值放入其它的数据型态内，组成其它比较复杂的数据储存形式。可是接下来我们总会在程序当中取出这些值，因此该怎么解开参照，让他们指向原来所代表的那一个资料内容呢？
我们先看看这样一个简单的参照：

```
my @aray = qw/John Paul May/;           # 一个数组
my $array_ref = \@array;                # 取得这个数组的参照
```

接下来，我们用大括号将参照包起来，并且恢复他应该有的数据型态代表符号，在这个例子中就是@号。所以看起来应该像是这样子：

```
print @{$array_ref};                    # 印出 JohnPaulMary
```

当然，我们也可以利用数组的方式来取得某个索引的值，也就是这样：

```
print ${$array_ref}[1];                 # 这样就跟 $array[1] 一样
```

看起来好像不太困难，那我们来依样画葫芦，试试看杂凑参照的解法。当然，还是先建立一个杂凑吧，并且取得他的参照吧：

```
my %hash = qw/John 24 Paul 30 May 26/;
my $hash_ref = \%hash;
```

接下来，好像并不困难，我们只要把{$hash_ref}视为一个杂凑变量的名称，所以要取得杂凑中，杂凑键为"John"的值就只需要这么作：

```
```

```
print ${$hash_ref}{John};                # 果然印
出 24
print ${$hash_ref}{Paul};                # 结果是
30
print ${$hash_ref}{may};                 # 正如我
们的期待，就是 26
```

当然，你也可以把%{$hash_ref}当成一个一般的杂凑来运作，所以你几乎可以毫无疑问的这么使用：

```
for (keys %{$hash_ref}) {
   print ${$hash_ref}{$_}."\n";          # 印出
24, 26, 30
}
```

你是一个很简单的例子，我们可以直接把%{$hash_ref}当成一般的杂凑来操作。所以一般使用于杂凑的函数也可以直接用于%{$hash_ref}上，相同的状况，我们也可以在解开数组参照之后，用相同的方式来操作。所以如果用刚刚的例子，我们也可以这么写：

```
my @aray = qw/John Paul May/;
my $array_ref = \@array;
for (@{$array_ref}) {
   print "姓名: $_\n";
}
```

## 14.4 利用参照进行二维数组

我们在前面已经提过了利用参照来实作二维数组的方式，可是为什么这一小节还要再重新解释一次呢？主要是因为我们刚刚可以利用参照建立一个简单的二维数组，可是我们却还不知道怎么能灵活的操作这个数组。而且利用参照来营造一个二维数组是非常常见的参照使用方式，所以我们必须再详细的逐步解释二维数组的建构，以及解构。最后并且引申出利用杂凑的值包含数组参照的运作与利用。如果你还不熟悉，我们先来建立一个二维数组。我们假设这是一个日期与气温的对照，每天定时量测当地气温三次，分别纪录于数组中。所以我们以比较繁杂的手续建立起这样的一个二维数组：

```
my @d1 = (24.2, 26.3, 23.4);            # 每天的
温度
my @d2 = (23.5, 27.5, 22.6);
my @d3 = (25.2, 28.7, 24.8);
......
......
my @d30 = (19.8, 22.1, 19.2);

my @daily = (\@d1, \@d2, \@d3, ..... , \@d30);
# 当月每天的温度
```

虽然复杂，不过终于把整个数组建立起来了。目前我们已经有了 30 个数组，各代表了第一天到第三十天中每天的温度纪录，接下来就是定义一个数组包含了这三十个数组的参照值，而这个数组也就包含了这个月每天三次温度的纪录。于是我们可以利用参照的方式取得某一天的温度，例如${$daily[4]}[0]就代表了第五天的第一次测量。这次你一定受不了了，这么复杂的结构并没有为程序设计师带来比较舒适的环境，反正让人徒增困扰。因为我们必须为每一天先建立一个数组，然后再将数组参照放入另一个数组中，接着解参照，取出第二层数组中的值。很显然，如果我们只有这种方式可以使用，那么负责 Perl 设计与维护的那些黑客们一定自己先受不了。所以我们的另一个方式就是「匿名数组」，「匿名杂凑」，而且这个作法我们刚刚已经稍微看过了。现在我们再来了解一下它们的用法。首先在赋值上，数组所使用的是中括号[]，也就是当你在对数组取值时的符号。而对于匿名杂凑，则是使用大括号{}，同样的，也是利用你对杂凑取值时所用的方式类似。所以刚刚的例子如果重新定义数组@daily 就应该要写成：

```
@daily = ([24.2, 26.3, 23.4], [23.5, 27.5, 22.6],
[25.2, 28.7, 24.8]...);
```

看起来好像跟其它程序语言的方式比较接近了，可以取值应该怎么作呢？还是要先解参照，然后取出数组的某个值，然后再来解参照吗？很庆幸的，这种复杂的工作实在不适合用来放在这种可能在日常生活中会大量使用的二维数组中，因此我们也可以用很方便的方式来取得其中的值。所以要取值的方法就像这样：

```
print $daily[2][1];
```

这样真的清爽多了，如果你用过其它程序语言的二维数组，其实大概也都是这样的写法。当然，你可以作的绝对不只二维数组，你可以用同样的方式来实作多维数组，就像你可以很容易的造出一个三维数组。

```
@demo = ([[2, 4, 5], [3, 2], [2, 6, 7]], [4, 7,
2], [[1, 3, 5], [2, 4, 6]]);
```

现在回想起来，如果这个数组要一个一个把名字定义出来，然后取它们的参照，放入其它数组中……，这实在太辛苦了。于是匿名数组节省了我们不少的时间，当然，想必也降低了很多错误的机会。

## 11.5 数组中的参照，参照中的数组，数组中的数组

这个标题实在太绕口令了，虽然我们应该直接取标题为：「匿名杂凑与匿名数组」，不过这样的标题好像非常不容易平易近人，所以还是维持这个冗长的标题吧。
在上一节其实已经利用匿名数组了，也就是我们用来实作二维数组的轻松愉快版本。另外，我们也尝试过在杂凑里面放入数组，可是既然我们可以方便的利用匿名数组来进行多维数组的实作，那么利用类似的方式，把匿名杂凑，匿名数组的交互使用，显然可以让整个数据结构更具有弹性。
还记得我们怎么整理学生的资料吗？那时候我们已经用了这样结构的处理方式。学生的个人数据项是一个匿名杂凑，而每个学生的成绩则是由匿名数组来组成的。因此我们就可以用简单的方式来取出需要的值，所以我们就可以这么用：

```
print $students{john}{grades}[2];
```

这样应该非常方便，你并不需要手动去解参照，或者进行什么繁杂的手续。而就像一般的数组或参照的用法一样，用中括号来取得数组的值，或是用大括号才使用杂凑。而匿名杂凑也是常用的方式，它们可能被隐藏在数组或杂凑中，就像我们刚刚看到学生资料的例子，就是一个「杂凑中的杂凑」实作的例子。
另外很常用的的一种匿名杂凑方式则是数组中的杂凑，很好的一个例子就是从数据库撷取出来的数据，这时候我们常常会把每一笔数据依据字段的名称，跟所得到值存放在杂凑中，然后将每笔这样的杂凑存入数组中所以一个数组看起来会像是这个样子：

```
@data = ( { 'column1' => 'data1',
            'column2' => 'data2' },
    { 'column1' => 'data3',
      'column2' => 'data4' } );
```

如果你的数据储存形式像是这个样子，在数组中放入匿名杂凑，那么你如果要取出某个值，就只需要这么写：

```
print $data[1]{column2};  # 这样你就可以得到
data4
```

其实你也许不太习惯，为什么在使用匿名数组，或匿名杂凑时，总会有不同于正常指定数组或杂凑的方式呢？不过我们可以来看看这样的状况：

```
my @array = ((3, 5, 7, 9), (1,4, 8, 6), (2, 5,
4, 2));
```

这时候，我们知道最外面一层是一个数组，我们利用串行的方式指定了三个元素给这个数组，而这三个元素却都是串行，也就是说，我们希望把这三个串行放入数组中。可是这时候问题就出现了，因为很明显的，我们必须在最外层的数组里面定义三个变量，才能利用参照的方式把串行放入数组里，可是在一般使用的时候，不管数组或参照，我们都可以使用串行的方式来赋值。像这样的两种形式其实都是可能的：

```
@temp = (3, 5, 7, 9);
%temp = (3, 5, 7, 9);
```

所以如果我们利用刚刚的方式，希望把三个串行利用匿名数组或匿名杂凑放进数组@array 的话，就会造成 Perl 的错乱，因为它无法清楚的明白你所需要的是匿名的数组或是杂凑。这也就是你必须清楚的表示你的需求，因此你如果希望使用匿名数组或杂凑，就必须适当的分别清楚，所以依据你自己的需求，你就必须作不同的定义，就像这样：

```
my @array = ([3, 5, 7, 9], [1,4, 8, 6], [2, 5,
4, 2]);
my @array = ({3, 5, 7, 9}, {1,4, 8, 6}, {2, 5,
4, 2});
```

因为在 Perl 当中，你都是利用最简单的数组，杂凑的数据结构，配合上参照(当然还包括匿名数组与杂凑)的方式，来组成更复杂的数据结构，例如多维数组，或是数组中的杂凑，杂凑中的数组等等。也就因此，你可以有更大的弹性来玩弄各种结构的组成。比如你可以在数组中的各个不同的元素里，摆放不同数据结构的参照，所以你当然可以这么作：

```
my @array = ({3, 5, 7, 9}, [1, 4, 8, 6], {2, 5,
4, 2});
```

所以，你对于这个变量的取值就有可能是：

```
print $array[0][2];  # 得到的结果是 7
print $array[1]{8};  # 这里会印出 6
```

其实参照的用法并不仅只于这些数据结构上的变化，你还可以取得副例程的参照，当然也可以使用匿名副例程的方式。就像你在使用数组或杂凑的参照一般。参照的用法非常的灵活，而且运用非常的广泛，Perl 的对象导向写法也是参照的运用。不过我们不希望刚入门的使用者被大量的参照困扰，所以等各位写过一阵子的 Perl 之后可以再去参考其它的 Perl 文件，了解更多关于 Perl 参照的用法。

习题：

1. 下面程序中，%hash 是一个杂凑变量，$hash_ref 则是这个杂凑变数的参照。试着利用$hash_ref 找出参照的所有键值。

```
%hash = ( name => 'John',
          age => 24,
    cellphone => '0911111111' );
$hash_ref = \%hash;
```

2. 以下有一个杂凑，试着将第一题中的杂凑跟这个杂凑放入同一数组 @array_hash 中。

```
%hash1 = ( name => 'Paul',
           age => 21,
     cellphone => '0922222222',
     birthday => '1982/3/21' );
```

3. 承上一题，印出数组$hash_array 中每个杂凑键为'birthday'的值，如果杂凑键不存在，就印出「不存在」来提醒使用者。

注一：其实不单只是这些数据可以取得参照，还有其它部份也可以使用参照来操作，不过我们并不在此讨论。

# 15. 关于数据库的基本操作

如果读者练习足够认真的话，也许你已经对Perl可以慢慢的上手了。当然，你可能也准备用Perl来写公司的一些小系统，或是个人日常生活使用的一些小程序。而大部份的时候，你都会希望能把存下一些数据，至少总不会每次要执行程序时，又要重新输入这些相关的信息。当然，很多数据也总是随着程序不断进行时被记录下来的。这时候你可以利用不少不同的方式来储存资料，例如你可以写入档案中。最简单的方式应该也是如此，我们总是可以举出这样的例子。

```
sub savePhone {
    my ($name, $phone) = shift;
    open PHONE, ">>phone";
    print PHONE "$name\t$phone";
    close PHONE;
}
```

这是一个很小，也很简单，作为通讯簿的副例程。或者它根本算不上是一个通讯簿，只是纪录姓名跟电话的相对应数据。而我们的作法也不过就是把得到的数据写入档案中，而这个档案的名称叫做"phone"。于是我们把姓名跟电话新增到档案的最后，而且就只作这个动作，然后就把档案关闭。如果我们持续新增朋友的电话，那么档案也就会不断增长，也就长的像一个通讯簿了。

这样的通讯簿有什么特色呢？特色就是它会长长的一串，如果你交友广阔，也许这样的档案可以让你印出好几页。当然，如果你要搜寻也不是太容易，要排序也是要自己处理。所以归纳出一个特色，这样的通讯簿纯粹只是作为「纪录」，实在无法拿来运用。不相信的话，你可以想象一下，如果你现在想纪录的不只姓名跟电话，你还想把众多朋友的地址，Email 全部记下来，那么你会发现用档案真是非常辛苦。何况如果你想在使用档案储存这些数据的同时还可以方便的搜寻，删除，排序等等，那么你大概很快就会放弃这样的想法。

当然，如果你只是想要简单的记下一些数据，那么单纯的使用档案也会让你自己轻松一点，不过如果你打算开始利用 Perl 帮你处理一些稍微复杂一点的数据时，那么你应该好好考虑使用数据库的形式。

## 15.1 DBM

一种非常简单的数据库形式，而且在你安装Perl之后，你也就同时拥有这样的数据库系统，也就是「DBM档案」。不过虽然同样都是「DBM档案」，运作的方式却根据所在的环境而不尽然完全相同。所以如果对于DBM的运作方式有兴趣的人也许就得自己去翻翻其它数据了。

不过就像本章大多数的部份，我们会尽量让大家在比较没有负担的状况下使用数据库。毕竟在这个时候，很多技巧与程序的语法似乎都是使用比自己动手开始写要重要许多。

### 15.1.1 与DBM连系

「DBM档案」的数据库非常有趣，它是利用特殊的杂凑来存取数据库，或说达成数据库的形式。所以它会利用杂凑跟所谓的DBM进行紧密的结合。这从我们开始操作DBM就可以看出来，所以我们先来开启一个DBM档案，试试怎么使用DBM。

```
dbmopen (%HASH, "dbmfile", 0666) or
    die "档案打不开!";
```

看起来是不是有点面熟？其实跟开档案的方式确实有点接近。不过我们所指定的并不是档案代号，而是一个杂凑，用来联系 DBM 档案，这样的方式可以让你在操作杂凑时就等于在操作 DBM 档案一般，也就是让你用简单的方式对杂凑进行改变时，Perl 会直接帮你反应到档案中。
其实当我们使用了 dbmopen 时，Perl 也会自动帮我们建立起相对的数据库档案，可是我们在写程序所对应的还是只有杂凑，这样想象起来好像非常的轻松。即使 Perl 已经帮我们在系统中建立了两个实体档案，我们却并不需要理会，继续使用方便的杂凑吧。
当然，这个杂凑就像我们平常所看到的一样，所以你也要遵守杂凑的命名规则。不过很多时候，程序设计师总会有一些惯用法，就像档案代号大多使用全部大写的方式，我们也习惯使用全部大写的杂凑代号来代表系结 DBM 档案的杂凑。而且请各位务必记住，所谓的系结是指在目前的程序当中，一但我们利用 dbmopen 这个指令时，Perl 会让我们以杂凑的方式来对 DBM 档案运作，可是一但我们关闭这个系结或是离开程序后，这样的关系就不存在了。所以档案内当然也不会有相关的杂凑名称被记录下来。
而关闭的方式则是只要使用像这样的指令：

```
dbmclose(%HASH);
```

没错，还是跟你在关闭一个档案一样的方式。而且如果你没有手动关闭这个 DBM 档案，Perl 也会在程序结束时自动将它关闭，不过你应该知道，这不会是一个好习惯的。

### 15.1.2 DBM档案的操作

在开启一个DBM档案数据库之后，我们就可以对它进行存取，也就是进行一般的操作。例如你可以修改数据库的内容，新增数据，搜寻你要的数据等等。至少值得庆幸的是，所有的事情都可以利用杂凑来完成，那是大家都还算熟悉的东西。其实我们如果看个例子，应该很容易就可以上手了。

```
dbmopen (%HASH, "dbmfile", 0666) or
    die "档案打不开!";
print $HASH{'John'} if (exists $HASH{'Hohn'});
$HASH{'Mary'} = '0227331122';

@sort_keys = sort {$a cmp $b} (keys %HASH);
for (@sort_keys) {
    print "$_: $HASH{$_}\n";
}

delete $HASH{'Paul'};
dbmclose(%HASH);
```

没错吧，除了第一行需要让你的杂凑跟 DBM 档案建立系结以及最后一行关闭 DBM 档案之外，你几乎看不出来你正在对一个 DBM 数据库进行操作。因此如果你打算要用 DBM 数据库，那么几乎是可以非常容易的上手。不过其实使用 DBM 数据库还是有一些比较深入的技巧，例如你可能要锁定数据库，以避免因为超过一个以上的行程在存取数据库而产生问题。不过一开始使用，你暂时还不需要让自己烦恼这么多 (否则只怕光担心这些问题就让人不敢使用了)。而这相关的问题，你可以在某些进阶的 Perl 相关书籍中找到合适的作法与解答。

### 15.1.3 多重数据

如果你还记得杂凑的特性(希望你会记得，不然该怎么使用DBM数据库呢)，那么你应该记得杂凑其实是由一对的键值所构成的。也就是说，杂凑中每个独特的键都会被对应到一个值上。而我们刚刚也利用了这个特性，纪录了姓名与电话的对应关系。但是这时候却有问题了，如果我希望记下的不只电话，或说除了家里电话之外，我还希望记下好友的行动电话号码，那么在杂凑中好像就不适用了。
可是如果一次只能纪录一个键跟一个值，那么DBM数据库还真是不实用，毕竟大多数的时候，我们总会需要使用一大堆的数据域位。因此最好可以让DBM数据库可以让一个键对应到多个字段，这样子我们就可以纪录行动电话，或是生日，地址等等其它数据了。
可是既然杂凑的特性没办法改变，那么要怎么样可以把多个字段挤在一个杂凑值中呢？很直觉的，我们当然可以直接把所有的资料「连」成一个超长的字符串。所以你可能得到这样的数据：

```
$personal_data =
"02-27631122\t0931213987\t1974.12.3"
```

接着你利用 split 来把字符串切成一个小数组，让你可以独立使用每一个部份，就像这样的方式：

```
@data = split /\t/, @personal_data;
```

这样好像很容易，你可以利用简单的方式来储存多个字段的数据。可是如果有其它更有效率的方式也许会更好，毕竟把所有东西全部胡乱的挤成一团似乎不是什么太好的方式。所以 pack 跟 unpack 这样的函式似乎可以帮我们解决这类的问题，而且它们也正是为了这个理由而存在的。当然，有一部份的理由是在于把数据打包成为系统或网络的传输时使用。
pack 的运作方式主要在于格式化我们的数据，也就是当我们在进行打包的时候，我们必须先定义出打包的方式。例如你要打包成字节，整数等等不同的数据格式。所以 Perl 会根据我们所定义出来的格式，把资料打包成整齐的字符串，以方便我们进行储存或传输的工作。至于我们常用的格式字符包括 c(字符)，s(整数)，l(长整数)，a(字符串)，h(以低字节开始的十六进制字符串)，H(以高字节开始的十六进制字符串)，f(浮点数)，x(一个 null 的字节)等等。当然，各式各样的格式字符种类繁杂，各位如果有兴趣可以利用 perldoc 来检查手上版本所有可用的格式字符集。我们先用简单的例子来看看 pack 的用法。

```
my $data1 = length(pack("s l", 23, 6874192));
my $data2 = length(pack("s l", 463, 66250921));
print "$data1\t$data2";
```

结果我们发现两个长度都是一样的，没错，因为 Perl 依照我们的格式把数据打包起来了。也就是整数以两个字节，长整数以四个字节来储存。当我们取得被打包过的资料后，我们可以用 unpack 把已经打包起来的资料解开，以便取得原来的数据格式，简单的写法就像这样：

```
my $pack = pack("s l", 23, 6874192);
my ($short, $long) = unpack("s l", $pack);
print "$short\t$long";
```

另外，你可以在格式字符后使用重复次数的方式，例如使用"cccc"来表示重复使用 c 四次，不过一般来说，我们都不会这么用，因为使用 c4 可以达到同样的效果，而且当然更简洁易读。只是当你使用数字来代表重复的次数时，有一个要注意的部份，因为有些格式字符的数字并不是用来表示重复的用法，例如你使用 a6 其实是表示长度为 6 的字符串。当然，如果你的字符串长度不足，Perl 是会

帮你补进 NULL 的。
其实使用 pack/unpack 这一组打包资料的函式的另一项好处就是你可以利用固定的长度来储存某些特定的数据烂位。例如你订好某些数据的字段，接下来当你把打包过的数据存到档案里面，那么你就可以保证每一比数据的长度都是相等的。以后当你要查某一笔资料的时候，你可以直接用 seek 的方式，迅速的找到数据所在的位置。
回头来看我们的通讯簿吧！我们现在希望储存不只一种字段，所以使用 pack 来把各种需要的字段打包在一起，当成杂凑的值存入 DBM 数据库中。所以我们先使用 pack 来储存电话跟行动电话两个字段吧，首先要先订出需要的长度。假设他们各是长度为 12 的字符串，所以我们应该这么作：

```
my $packed = pack("a12 a12", $tel, $cellphone);
$HASH{'John'} = $packed;

my $data = $HASH{'Mary'};
my ($mary_tel, $mary_cell) = unpack("a12 a12",
$data);
```

接下来，你可以试着用 DBM 作简单的数据管理，而且可以使用多个字段。

## 15.2 DB_File

另外，使用Perl，你也可以简单的使用杂凑的方式来存取Berkeley DB，而所需的模块DB_File目前已经内附在Perl的核心当中，因此你装完Perl之后，如果你的系统中已经有Berkeley DB的话，你就可以在程序中直接使用DB_File来存取Berkeley DB。
使用DB_File其实非常容易，和使用dbmopen非常类似，你只需要把数据库档案和杂凑建立起系结，那么你就可以直接使用杂凑来控制数据库的档案，不过实际的使用上，还是有着很大的差别。最简单的使用方式，其实只需要这么作：

```
$filename = "test";
tie %hash, "DB_File", $filename;
```

接下来，你就可以把建立起系结(tie)的杂凑直接使用，就像一般杂凑一般。于是我们就直接使用%hash 来存取，就像这样：

```
$hash{'John'} = '27365124';
```

```
$hash{'Mary'} = '26421382';
```

没错，完全就只是杂凑的操作，感觉被骗了吗？其实在操作上，最简单的方式也就是这样，跟你直接使用 dbmopen 没有相差太远。不过就使用上而言，DB_File 还是有些比较方便的地方，例如你可以在建立系结的时候就指定档案的权限。另外，你也可以控制相同杂凑键的处理方式。就像这样的写法：

```
$DB_BTREE->{'flags'} = R_DUP;    # 允许键值重复
$tie = tie %h, "DB_File", "test", O_RDWR, 0644,
$DB_BTREE or die $!;
```

接下来，你就可以利用 $tie 来进行一些操作。你可以使用

```
$tie->del($key);
$tie->put($key, $value);
$tie->get($key, $value);
```

等等方式来对数据库进行方便的存取。

可是你慢慢会发现，这样的数据库虽然方便，可是有时候却未必能够满足我们的需求。于是我们也许可以转向求助于关系型数据库，虽然它的数据库形式复杂不少，但是以功能来说，却能够让你在处理大量数据时还能够随心所欲。

## 15.3 DBI

接下来，我们假设你已经知道的需求，你对于简单的DB_File，也就是Berkeley DB所提供的简易数据库再也无法满足。你需要更强大的数据库功能，最好是能应付各种复杂状况的「关系型数据库」。你的数据需要大量的表格来储存，表格和表格间也许还存在着交缠的连结与相关性。总之，至少在这时候，你应该要有「关系型数据库」的概念，当然也要可以自己独立掌握数据库的操作，因为我们并没有打算在这里教导大家怎摩使用数据库。当然，目前在开放源码社群经常使用的MySQL也是属于「关系型数据库」，你也可以在网络上找到非常完善的相关文件与使用手册等等。所以如果你对于这部份还不太熟悉，那么可以先阅读相关的资料后再回头来看这一节，利用Perl连结到各种的关系型数据库。
不过在正式上路之前，我们还是必须来看看在Perl的使用上，整个DBI的概念。所谓的DBI，其实是Database Interface，所以其实对于DBI来说，它只是一个让使用者可以降低成本的方式去控制各种数据库，也就因此，它的使用必须搭配所谓的DBD，也就是Database driver。我们可以利用简单的图来表达DBI，DBD跟实际数据库之间的关系。

<<图一>>

从图上可以看得出来，其实使用者所接触到的部份几乎只有DBI的部份，只不过在使用前必须根据使用者实际搭配的数据库安装DBD。当然，我们在这里也就只针对DBI的部份介绍。可是因为DBI的使用其实是利用Perl对象导向的程序写作方式来进行，而我们并没有介绍对象导向的程序写法，所以对于大多数的人来说也许有些困扰，不过一开始就让初学者学习怎么使用Perl的对象导向写法似乎太过躁进。不过即使还没开始自己动手写对象导向程序前，当然也可以安心的使用已经现成的各式各样对象导向模块。而且目前在CPAN上有不少模块都已经是以对象导向的方式开发，因此使用者也可以利用学习DBI的机会，顺便也学学对象导向模块的使用。

一般来说，我们在开始使用一个对象时，都会先利用建构元来产生一个对象。有些程序语言也许是使用 new
这个「方法(method)」来达到进行这样的工作，在 Perl 的对象导向语法中，虽然也有很多模块使用 new
来达成建构一个对象的目的，不过 Perl 本身的对象导向语法并不强制规定建构方式的关键词。所以你可以使用

```
$dbh = DBI->connect(...);
```

来建构出一个 DBI 对象，然后使用$dbh 来进行各式各样该对象所提供的方法。

一开始，你显然要先在你的程序使用 DBI 这么模块，接着就是连接上你的数据库，就像这样：

```
use DBI;

my $dbh = DBI->connect("dbi:Pg:dbname=foo",
"user", "passwd");
```

其中的 Pg 其实代表的是数据库的驱动程序部份(Pg 指的是 PostgreSQL 的驱动程序)，其实比较精准的用法应该是这样：

```
$dbh = DBI->connect($data_source, $username,
$auth, \%attr);
```

其中的$data_source 也就是描写相关的数据库驱动以及所要连结的数据库等信息，当然，如果需要，你还必须把数据库所在的主机位置也写在这里，就像这样：

```
$dbh =
DBI->connect("dbi:Pg:dbname=foo;host=db.host"
, "user", "password");
```

另外，由于 DBI 可以处理相关失败时的错误状况，如果连接数据库失败，它会传回错误讯息：$DBI::errstr。所以我们在进行数据库连结时，可以使用这样的方式：

```
$dbh = DBI->connect("dbi:Pg:dbname=foo", $user,
$passwd) or die $DBI::errstr;
```

有些时候，你可能不知道你的数据库是不是已经安装了专门给 DBI 使用的驱动程序，这时候你可以利用这样的方式来取得目前机器上可以使用的所有驱动程序的数组：

```
@available = DBI->available_drivers;
```

连接上数据库之后，最简单的方式就是直接执行你的 SQL 命令，所以我们要用最简单的方式应该是这么写的：

```
my $sql = "DELETE FROM foo";
my $return = $dbh->do($sql);     # 传回受到改变
的数据笔数
```

这样的方式是让你执行 SQL 语法的最简单方式，不过它的主要限制在于并没有办法传回你从数据库中选取的数据。因此大多数的时候，我们都会使用其它的方式来进行数据库的 select 动作。一般的方式大概都会是这个样子：

```
```

```
my $sql = "SELECT * FROM foo";
my $sth = $dbh->prepare($sql);
$sth->execute;
while (my @result = $sth->fetchrow_array) {
    print $result[0];
    # @result 回依序取得每笔数据的各字段值
}
```

这样是非常常见的写法，主要就是用来从数据库取出某些特定的数据。这是因为我们并不能直接使用 do 这个方法来执行 select 的语法，因为那并不会得到我们真正想要的内容。所以我们还是先写好我们要取值的 SQL 语法，也就是"select * from foo"这个叙述。接下来，我们要告诉$dbh，处理我们所要的这个 SQL 叙述，并且建构出一个叙述的控制器(statement handler)，因为我们在进行 select 的时候，大多是透过执行某个 select 语法，接着一笔一笔取回执行的结果。而叙述控制器刚好就是为了这样的需求而存在的。因此在利用 prepare 产生出控制器之后，我们就可以要求控制器执行我们的 SQL 叙述。紧接着利用 while 这个循环逐笔的把取得的数据当到其它的变量供我们使用。
而取回的方式最简单的就是利用数组的方式，也就是利用 fetchrow_array 这个方法来一笔一笔，所以每当 fetchrow_array 执行后有取得结果，就会让 while 叙述成真，因此我们就可以从数组@result 中取得该笔的值。而数组的元素则是依照数据库 select 出来的结果来排序。
所以很多人并不喜欢这样的用法，因为当你的数据表格字段足够多时，你会发现这种用法真的让人很头痛。你看看下面的例子吧：

```
my $sql = "SELECT a, b, c, d, e, f, g, h, i, j
FROM data_table";
my $sth = $dbh->prepare($sql);
$sth->execute;
while (my @result = $sth->fetchrow_array) {
    ....
}
```

然后你希望取出字段 e 跟 i 来作运算，于是你就要从 0 开始算，算出 e 是索引为 4，i 则是索引为 8 的字段。这种事情真的是太辛苦了，当然，你的字段数目也许还会更多，那时候你可能会想要找一片坚固一点的墙了。所以我们还是宁愿使用杂凑的方式来取得数据库送出来的值，也就是 DBI 提供的 fetchrow_hashref。至少使用杂凑参照的方式能够让使用者在取值时更为直觉。如果我们改写刚刚的取值方式，应该可以这么使用：

```
while (my $hash_ref = $sth->fetchrow_hashref) {
```

```
    print $hash_ref->{'e'};         # 你可以直接叫
用字段名称
    print $hash_ref->{'i'};         # 这样显然愉快
了
}
```

另外，有时候你还会需要更方便的形式来取出数据，其中常用的包括了 fetchall_arrayref 跟 fetchall_hashref。其中的用法会是类似这样的形式：

于是，你在也不需要手动计算某个字段应该位于数组的第几个元素了，希望这样能够让你心情愉快的写程序。可是你又发现，很多时候，你只是打算先把全部的数据从数据库撷取出来之后，再一次进行运算，还是干脆包成一个数组，丢给副例程去处理，那么你可能会写成这样的方式：

```
$arrayref = $sth->fetchall_arrayref;
或是
$hashref = $sth->fetchall_hashref($key);
```

这两种方法都会一次传回使用者所选定的条件，只是传回值储存的方式有所差异。其中特别有趣的是 fetchall_hashref 这个方法，使用者可以选定数据库键值字段，并依此来得到相关的其它字段值。假如我们刚刚的众多字段中，字段'a'是数据表的 Primary Key，那么我们就可以使用这种方式，取出特殊键值的那一笔数据：

```
$hashref = $sth->fetchall_hashref('a');
print $hashref->{foo}->{c};
```

这一行很显然的，我们使用了 fetchall_hashref 这个对象的方法，重要的是我们指定了主要键的字段'a'。于是叙述控制器到数据库选出了我们要的所有数据，并且以主要键作为杂凑的键，而它的值则是其它个字段键，值所形成的杂凑参照。因此我们要取得某个主要键值为'foo'这笔数据中，字段'c'的值，就可是使用第二行的方式取得。

以上我们提到的大多是一般在进行一次的 DBI 操作时会使用到的方法，可是整个 DBI 的操作却是相当复杂，提供的功能也非常强大，要详细讨论的话，都可以写出一本书来。不过即使你没打算买回 DBI 的书仔细钻研，在你可以使用简单的 DBI 操作之后，我们还是建议你看看 DBI 的官方文件，你可以直接使用 perldoc DBI 来阅读。

你以为我忘了另外一个重要的部份了吗？当然没有。接下来我们要来看看在你对

于数据库的操作结束时，你应该要作的工作就是把它释放，也就是利用 disconnect 的方法来释出它所占用的资源。虽然 Perl 会在程序结束时自动释出还没占用的相关资源，可是我们还是强烈建议你，你应该在使用完 DBI 的资源后，手动将它关闭。因为你的程序也许会在系统运作一段时间，而且你却只有某一段程序使用了这些资源，更有甚者，你也许一次开启了多个数据库连结，那么适时的释放这些资源显然是程序写作的好习惯。

```
$dbh->disconnect;
```

## 15.4 DBIx::Password

相对于DBI的复杂程度，DBIx::Paswsword只能算是协助使用DBI的一个小小工具。它也没有提供类似DBI那样魔术般的强大功能，简单的说，它只是让你用比较偷懒的方式来建构出一个数据库控制对象，也就是所谓的database handler。别忘了，数据库是相当重要的，我们经常要帮数据库设定出繁杂的密码，以避免遭人破解。可是这些密码到底有多复杂呢？很多时候，连管理人员也总是搞不清楚这些密码，再加上如果你的服务器上还有不只一个数据库，或是你的程序要连接多部数据库，而且它们还分属于不同的种类，那么要写个程序倒也相当辛苦。
针对这个需要考验程序设计师记忆力的问题，DBIx::Password提出了解决的方式。也就是只要设定一次，那么设定会被写入Password.pm这个档案之中。当你第一次安装DBIx::Psassword这个模块时，它会问你一堆问题，其实也就是问你目前使用中的数据库设定。而所有的数据将会像这样的存在数据库中：

```
my $virtual1 = {
    'dbix' => {
        'database' => 'db_name',   # 数据库名称
        'password' => 'passwd',    # 使用者密码
        'attributes' => {},        # 连接数据库的
其它参数
        'port' => '',
        'username' => 'user',      # 使用者名称
        'host' => 'localhost',     # 主机
        'driver' => 'mysql',       # 数据库类型
        'connect' =>
'DBI:mysql:database=db_name;host=localhost'
    },
};
```

那么以后当你要连接数据库时，就只需要使用虚拟的名称了，原来建构数据库控制器的方式也由 DBI 转移倒 DBIx。所以使用者再也不需要记一长串的数据库连接需要的参数，因为所有的东西现在都由 DBIx::Password 负责了。让生活快乐一点的写法就变成了：

```
my $dbh = DBIx::Password->connect($user);
```

刚刚我们举的例子中，就可以把$user 设为"dbix"，那么 DBIx::Password 就会帮忙我们使用 DBI 进行数据库的连结。而且在 DBIx::Password 中，你当然可以设定多组的参数，因为它只是利用杂凑存起你所有数据库的相关数据。

虽然 DBIx::Password 还提供其它的一些操作方式，不过其实我们几乎有超过百分之九十的机会都是使用 connect 这个对象方法，所以我们应该暂时可以先下课了。

习题：

1. 利用自己熟悉的数据库系统(例如 MySQL 或 Postgres)，建立一个数据库，并且利用 DBI 连上数据库，取得 Database Handler。
2. 试着建立以下的一个数据表格，并且利用 Perl 输入数据如下：

```
数据表格：
name: varchar(24)
cellphone: varchar(12)
company: vrchar(24)
title: varchar(12)
数据内容
[ name: 王小明
  cellphone: 0911111111
  company: 甲上信息
  title: 项目经理 ]
[ name: 李小华
  cellphone: 0922222222
  company: 乙下软件
  title: 业务经理 ]
```

3. 从数据库中取出所有数据，并且利用 fetchrow_array 的方式逐笔印出数据。
4. 呈上题，改利用 fetchrow_hashref 进行同样的工作。

# 16. 用Perl实作网站程序

很多人开始接触Perl都是因为把他拿来作为写CGI(Common Gateway Interface)程序的工具，当然，也因此不少人都把Perl定位在「写网站程序」。虽然事实绝非如此，不过用Perl来写CGI也确实是非常方便的。尤其在不少前辈的努力之下，让我们现在得以更方便的建立网络相关的程序。虽然目前已经有其它非常方便的脚本语言(scripting language)也可以让人非常轻松的写出CGI程序，例如像PHP或ASP一开始就是以最方便的嵌入HTML来进行交互式网页作为主要目的(目前PHP已经把触角延伸到其它方面，例如PHP-GTK)，可是Perl所依赖的不单单只是方便的CGI写成模式，更重要的是程序语言本身所能达到的效果。

很多人在准备开始写CGI的时候都会遇到类似的问题：「我应该学PHP或是Perl？」其实如果你大多数的时候只希望把Perl拿来写网站，而且你手上的东西又非常的急迫，那么PHP也许可以很快的让你达到目的。虽然很多人可能不以为然，不过我倒是以为，可以把学Perl当成纯粹的学习一种程序语言，而CGI只是一种实作Perl GUI(图形使用界面，Graphical User Interface)的方式。何况越来越多人把浏览器当成是GUI以及Client/Server架构最容易的达成方式，其中当然也因为使用浏览器作为客户端程序可以降低使用平台的困扰，而Perl正是验证了这种诠释。

利用Perl来写CGI程序的另外一个最大的疑虑大概是Perl的效率问题，这也许要从网站结构跟原理说起。一般来说，整个网站的原理并不算太困难，也就是客户端发出一个需求，服务器端收到之后，根据用户的需求发出响应，然后关闭两者的联机。而如果想要达到动态网站的目的，也就是根据使用者的需求，由服务器端在收到需求之后，根据服务器的设定把数据传给后端的程序，接着程序依照需求产生出结果之后再传回给网站服务器。接着，网站服务器就根据正常的流程把数据传回给客户端。我们可以从图一看到比较清楚的流程。

在正常的状况下，Perl每次收到由网站服务器传来的需求时，就会重新开启一个程序(process)，然后开始根据需求来产生结果。可是问题在于Perl在初始化的过程必须耗费相当的时间跟内存，因此当Perl完成初始化，然后产生出适当的结果并且回传给网站服务器。这样的过程其实是相当漫长的，尤其当你的服务器负载过大，或是硬件本身的效率不佳的时候，就会壤人感到非常的不耐。而这也经常是Perl作为CGI程序最让人疑虑的部份。

当然，这部份也已经有解决的方案了。现在使用者可以利用FastCGI，或是在Apache中搭配mod_perl使用，这样一来就可以让原来的程序被保留在内存中，而不必因为每次有使用者发出浏览的需求就必须重新启动Perl，造成因为Perl初始化的延迟问题。不过如果想要有更好的效能来使用mod_perl，那么足够的内存就变成非常重要的。所幸硬件的价格不断下降，让这样的资源使用不至于发生太大的问题。

如果要使用Perl来增进网站的速度，是需要进行一些设定上的改变，当然有时候能够跟程序配合是更理想的。不过这对于刚开始准备使用Perl来作为网站程序的工具而言显然是有些困难的，所以我们并没有要在这一章中介绍更多相关于mod_perl的使用。待各位对于使用Perl来建构出网站这样的工作熟悉之后，可以继续选择相关书籍或文件进行研究。

## 16.1 CGI

现在使用Perl作为网络应用程序的主要程序语言时，几乎所有人第一个会遇到的就是CGI这个模块。CGI模块提供的功能非常的强大，不管是输出至网页上或是透过CGI取得使用者输入的参数，另外也可以利用CGI动态产生HTML的各种窗体选项。对于动态网页的支持，CGI目前已经可以算是非常完善。甚至有时候会让人感到相当意外，因为有些时候你会忽然发现，原来这些东西CGI.pm也可以达到。现在我们就来看一下，如果要使用Perl开始写CGI程序，那么应该怎么下手呢？毫无疑问，你总得先加载CGI这么模块，所以就像我们所熟知的方式，先在你的程序加入这一行吧：

```
use CGI;
```

接下来，我们应该建立一个新的 CGI 对象，方法也不太难，也就是利用 new 这个对象的操作方法。所以只要这样子就可以建立起 CGI 对象：

```
my $q = CGI->new;
```

如果你想要把任何内容输出到网页上，你大概都要先送出 HTML 的标头档到网页上，也就是所谓的 header。最简单的方式当然就只要这么写：

```
print $q->header;
```

不过其实 header 有很多的参数，例如你应该要可以设定内容的编码或输出的内容型态等等。所以 header 这个函式也允许你使用相关的参数来改变 header 的属性，例如你可以设定网页输出的内容编码为 UTF-8：

```
print $q->header(-charset => 'utf-8');
```

而且你可以一次指定多种属性，就像这样的方式：

```
print $q->header(-charset => 'utf-8',
                 -type => 'text/html');
```

当然，HTML 的标头档还有各式各样的属性，使用者也都可以利用 header 来设定，而且这样子的方式似乎也比起手动输入各种 header 的设定值要简洁不少，因此这个函式对于经常使用 CGI 的人来说还是非常方便的。而另一个重要的功能则是 CGI 的 param，也就是利用 CGI 传回来的参数。很多时候，我们都会利用这样的方式来取得由 HTML 窗体中传回来的字段值：

如果你的 HTML 里面写的是这个样子：

```
<input type="text" name="user">
```

那么你的 Perl 就可以这么使用：

```
$q->param('name');
```

也就是藉由 HTML 里面的窗体字段名称作为 param 的参数来取得使用者输入的值，这样就可以让程序设计师很方便的取得使用输入的结果。举个简单的例子吧！如果我们想制作一个使用者登入的程序，那么画面上大多不外乎就是使用者名称、密码两个字段。所以我们通常的作法就是利用取得的使用者名称去数据库搜寻相对应的密码数据，如果没有相关的数据就表示没有这个使用者，或是使用者输入的使用者的名称有误。如果可以找到相对应的使用者，那么我们就比对数据库中撷取出来的密码与使用者输入的是否一样。而程序的写法大概就像是这样：

```
#!/usr/bin/perl -w

use strict;
use CGI;
use DBI;

my $q = CGI->new;
print $q->header;
my $user = $q->param('user');
my $dbh = ('DBD:mysql:database=foo', 'user',
'passwd');
my $sql = "SELECT password FROM table WHERE user
= '$user'";
my $sth = $dbh->prepare($sql);
$sth->execute;
unless (my @pwd_check = $sth->fetchrow_array) {
   print "没有这位使用者";
```

```
else {
   unless ($pwd_check[0] eq
$q->param('passwd')) {
       print "密码错误";
   else {
       ......
   }
}
```

有些时候，你会希望可以一次取得所有由 HTML 窗体传过来的字段名称，这时候 CGI 的另一个函式就可以派上用场了。也就是可以使用 params 这个函式，它会传回一个数组，这个数组就包含了所有由 HTML 传来的窗体字段。因此你的程序中可以这样写：

```
my @params = $q->params();
```

我们一开始就提到了关于利用 CGI 这么模块送出动态 header 的方式，其实它不只能够送出档头，你还可以动态的产生其它的网页元素，例如你可以使用这样的方式来印出字级为 h1 的文字内容：

```
print h1('这是 h1 级的字');

或是使用

print start_h1,"这是 h1 级的字,end_h1;

结果都可以产生

<h1>这是 h1 级的字</h1>

这样的 HTML 内容
```

类似的用法还有包括下面几种：

```
start_table()  # 送出<table>这个标签
end_table()  # 送出</table>这个标签
start_ul()  # 送出<ul>
end_ul()  # 送出</ul>
```

当然，你还可能不直接送出 header，因为你想要在处理完某些状况之后，把网页的地址送到其它地方。这时候你就应该使用 redirect 这个函式。它的用法也是非常的简单，你只需要告诉它你想转往的其它网址就可以了。

```
$q->redirect('http://url.you.want/');
```

至于 HTML 的元素，也有很多时候你可以利用 CGI 这么模块动态产生，例如你可以使用 img 来产生出<img src="">这样的 HTML 标签。其它诸如 ul，comment 等等也都能够轻易的被产生。不过另外一组非常完整的则是 HTML 窗体的产生方式，也就是你可以利用 CGI 来产生大多数的窗体字段。
一个非常基本的例子，就是利用 CGI 模块来产生一个 text 的窗体字段。

```
print $q->textfield( -name=>'field',
                  -default=>'default value');
```

另外的一些字段也都可以用类似的方式产生，就像接下来的例子：

```
print $query->textarea( -name => 'textarea',
                    -default => '会放在字段的默认
值',
                    -rows => 5,
                    -columns => 10);
print $query->password_field( -name =>
'password',
                         -value => '这里也是默认
值',
print
$query->checkbox_group( -name=>'checkbox',

-values=>['value1','value2','value3','value4'
],

-default=>['value1','value3'],
                         -linebreak=>'true',
                  -labels=>\%labels);
```

当然，还有各式各样的窗体字段，使用的方式也都大同小异，事实上，CGI 模块

的文件中有详细的描述。不过有一个非常重要的却是不可忽略的，也就是错误讯息。当使用者使用 CGI 进行一些运作时，有时候会有一些错误，这时候 CGI 模块会透过 cgi_error 来传回错误的讯息，所以你可以在程序中加上这个函式，让 CGI 发生错误时能送出错误讯息。就像这样的方式：

```
if ($q->cgi_error) {
   print "无法处理 CGI 程序";
   print $q->cgi_error;
}
```

接下来，我们再来讲一个 CGI 模块的重要功能，也就是 cookie 的存取。许多时候，网站设计者为了减少使用者重复输入数据的困扰，或是取得使用者浏览的纪录，常常会藉由客户端浏览器的 cookie 功能来纪录一些相关性的数据。而 CGI 也可以针对这些 cookie 进行存取。如果我们想要取得已经存在客户端浏览器上的 cookie 数据，我们只需要这么作：

```
use CGI;
my $q = CGI->new;
my $cookie = $q->cookie('cookie_name');
```

接下来，我们当然也可能需要写入 cookie 到使用者端，那么我们可以利用 CGI 的档头来完成这项工作，也就是 HTML 的 header。而完整的用法就是先把你所要写入的 cookie 值，属性都先设定好，然后直接用 header 来把这些内容送给使用者的浏览器中。

```
$cookie = $q->cookie( -name=>'cookie_name',
                      -value=>'value',
                      -expires=>'+1h',
                      -domain=>'.my.domain',
                      -secure=>1);
    print $q->header( -cookie=>$cookie);
```

很显然的，CGI 的功能还不止于此，虽然我们已经介绍了大部份使用 CGI 时常用的功能。不过如果你还有进一步的需求，应该务必使用 perldoc CGI 来详细阅读 CGI 的相关文件。

## 16.2 Template

Template虽然不单单用于网络应用程序中，可是却有许多人在写网站相关的程序时总会大量的使用，因为对于能够让程序设计师单独的处理程序而不需要担心网页的设计对于许多视设计为畏途的程序设计师而言，实在是非常的重要。
Template其实完整的名称是Template Toolkit，因为目前的Template Toolkit的版本是 2.13，所以一般人又习惯称呼目前的为TT2。至于正接受Perl基金会发展的则是Template::Toolkit的第三版，也就是俗称的TT3。当然，既然可以接受Perl基金会的赞助开发新版Template::Toolkit，可见这个模块对于Perl社群的重要性了。
首先我们先来谈谈template系统的概念，让大家能更深刻的感受使用template系统对程序设计师在网站程序的重要性。我们在刚刚可以看到CGI这个模块的运作，所以我们可以透过CHI的使用，输出绝大多数的HTML标签，也就是完成一个HTML页面的输出。当然，还有一种可能就是直接使用print指令，一个一个的把HTML标签手动印出。可是不管是上面两种方式选择哪一种都会遇到相同的问题，也就是要怎么跟网页设计者一起合作来完成一个美观，功能又强的网站呢？有一种可能的方式也许是由设计者做好一般的HTML页面，然后你将这个页面当作一般的文本文件将它逐行读入。然后自行置换掉要动态产生的部份。如此一来，只要网页设计者能在网页中加上让你的程序可以认得的关键词，那么你就可以不理会画面的改变，而且还能够让程序正确的执行。这样的概念正是孕育出模板系统的主要想法。
Perl的模板系统其实不只一种，可是Template Toolkit却是深受许多人喜爱的，因为它不但可以让使用者把模板与程序代码分隔，也可以让程序设计师在模板中加上各式各样的简单控制。虽然说简单，可是功能却也一点也不含糊。因为使用者可以在里面使用循环，可以取得由程序传来的各种变量，当然也可以在模板中自己设定变量。当然，最后如果所有的方法都用尽，还不能达成你的要求，你也可以在模板里面加上Perl的程序代码。
基本的使用TT2，你应该要有一个Perl的程序代码，跟相关的模板。而如果你只需要单纯的变量替换，那么使用的方式非常的简单。你可以这么写：

```
use Template;

my $config = {
    INCLUDE_PATH => '/template/path',
    EVAL_PERL    => 1,
};

my $template = Template->new($config);

my $replace = "要放入模板的变数";
my $vars = {
    var  => $replace,
};
```

```
my $temp_file = 'template.html';
my $output;
$template->process($temp_file, $vars, $output)
    || die $template->error();

print $output;
```

这时候，你就必须有一个符合 Perl 程序代码中指定的模板档案，已就是 template.html。而在这个模板中，大多数都是一般的 HTML 标签。而需要被置换的变量则被定义在$vars 中。我们可以先来看看这个 templte 可能的形式。

```
<html>
    <head>
  <title>这是 TT2 示范</title>
    </head>
    <body>
  这里可以替换变量 [% var %]
    </body>
</html>
```

好了，现在你可以把这个 HTML 拿去给网页设计的人，让他负责美化页面，只要他在设计完页面后，能把关键的标签[% var %]留在适当的位置就可以了。不过你可不能轻松，让我们来研究一下 Template 是怎么运作的。首先，在我们新建立一个 Template 的对象时，我们必须设定好相关的内容，在这里的例子中，我们只有设定了两个参数，一个是 INCLUDE_PATH，也就是你的 template 档案所放置的位置，这里的内容可以是一个串行。也就是说，你可以指定不只一个路径。另一个我们设定的是 EVAL_PERL 这个选项是设定是否让你的 Template 执行 Perl 的区块。当然，选项还有好几项，例如你可以设定 POST_CHOMP，这个选项跟 chomp 函式有一些类似，它可以帮你去除使用者参数的空格符。另外，还有 PRE_PROCESS 的选项则是设定所有的模板在被加载之后，都必须预先执行某个程序，例如先把文件头输出到模板中等等。另外，你还可以修改 Template 预设的标签设定，例如我们刚刚看到的[% var %]，就是利用 Template 的预设标签[% %]把它表现出来。而 Template 允许你在建立 Template 对象时使用 START_TAG 跟 END_TAG 来改变这样的预设标签。如果你用了这样的方式：

```
my $template = Template->new({
                START_TAG => quotemeta('<?'),
                END_TAG   => quotemeta('?>'),
            });
```

那么刚刚在模板里的变量就应该改写成

```
<? var ?>
```

这样似乎跟 PHP 有一点像了。

不过 Template 的作者也知道很多人大概很习惯 PHP 或是 ASP 的标签，所以在 Template 也提供了另一个设定选项，也就是 TAG_STYLE。你可以设定成 php(<? ... ?>)或是 asp(<% ... %>)，不过其实我个人以为[% ... %]的原创形式还算顺手，所以倒是没换过任何其它标签风格。Template 的设定选项非常多样化，不过只要了解以上的这些项目就大概都能应付百分之八十的情况了。如果还需要其它的信息，则可以参阅 Template::Manual::Config。
接下来，我们来看看使用上有什么需要注意的。在程序中，基本上你如果有什么特别需要注意的部份，那大概就是变量的传递了。如果你要传一个纯量变量，跟我们刚刚的范例一样，那么就只是把变量指定为杂凑变量的一个值。就像我们刚刚的用法一样，你就只要指定：

```
my $vars = {
   var => $replace
};
```

可是很多时候，我们也许会传送一整个数组或是杂凑，那么这时候你最方便的方式就是传送这些变量的参照，写法也许就像是这样：

```
my @grades = (86, 54, 78, 66, 53, 92, 81);
my $vars = {
    $var => $replace,
    $var2 => \@grades,
};
```

这时候，你的模板内容显然也需要改写，把印出数组的这部份加入你的模板中，一般来说，我们可以使用 Template 提供的 FOREACH 循环。所以你可以在你的 template 加上类似的一块：

```
[% FOREACH grade = grades %]
成绩: [% grade %]
[% END %]
```

当然，除了数组，我们还可以使用杂凑。使用的方式却也不太一样，虽然你还是传递杂凑参照，可是在模板中的使用却是直接利用杂凑键。所以你的 Perl 程序代码跟模板中分别是这么写的：

```
my %hash = ( height => 178,
         weight => 67,
       age => 28 );
my $vars = { var => \%hash };
```

于是你必须在模板中做出相对应的修改：

```
[% var.height %]
[% var.weight %]
[% var.age %]
```

我们刚刚看到在模板中放了 FOREACH 这个 Template 提供的循环，其实模板中还有许多可供利用的特殊功能，例如一个非常方便的就是[% INCLUDE %]。很多时候，人们都喜欢在网页的某个部份加上一些制式的内容，最常见的当然就是版权说明了。所以我们可以把这些版权说明的内容放到某一个档案中，例如就叫做 copyright.tt2 吧！所以我们有了一个 template 档案，内容其实就是一些文字叙述：

```
copyright.tt2:
Copyright Hsin-Chang Chien 2004 - 2005
```

好了，现在我们有不少其它的模板档案，希望每一页都能加上这一段内容。那么我们只需要在这些档案的适当位置加上这样的一行()所谓的适当位置就是你希望看到这些内容的位置：

```
[% INCLUDE copyright.tt2 %]
```

这样的写法可以让你一次省去相当多的麻烦，尤其当你有可能更动这些档案的内容时。例如我现在想把版权的内容进行调整，那么你只需要修改 copyright.tt2 一个档案，而不需要把所有的档案一个一个叫出来修改。不过其实 INCLUDE 可以应付比较复杂的模板系统，而像版权声明这种纯文字的档案，还有更简洁的加载方法，也就是 INSERT，如果你要被加载的档案是一个纯文字文件，不需要 Template 帮你进行任何的处理，那么就考虑使用[% INSERT %]吧！

另外，Template 还支持另一种常用的重复叙述，也就是 WHILE。它的语法相当简单，也就是使用这样的区块把要执行的内容标示出来：

```
[% WHILE condition %]
   ....
[% END %]
```

而 IF 叙述的基本用法也是和 Perl 语法几乎一样只是他是使用大写字母，而且用 Template 的标签区隔出来，所以你可以轻松的使用：

```
[% IF condition %]
   ....
[% END %]

或是

[% IF condition %]
   ....
[% ELSE %]
   ....
[% END %]

而 ELSIF 也是被允许的，用法也是类似：

[% IF condition %]
   ....
[% ELSIF condition2 %]
   ....
[% ELSE %]
   ....
[% END %]

当然，你还有 UNLESS 可以使用，用法也是相同：

[% UNLESS condition %]
   ....
[% END %]
```

至于如果你真的想在模板里面写 Perl 程序，也只需要这么作：

```
[% PERL %]
   # perl 程序代码
   ....
[% END %]
```

只是我个人并不建议你常常需要这么使用，否则你有可能其实是挑错工具了。因为 Perl 有其它模板系统也许比较符合你的习惯跟需求。而我们接下来就要介绍另一种在 Perl 社群中最近非常风行的另一套网络应用程序的搭配系统，也就是 Mason。至于 Template，它还有很多奥妙，你可以试着参考相关的官方文件。

## 16.3 Mason

网页设计跟程序代码怎么切割，这个想法会根据写程序的人的习惯而有很大的差距。很多人喜欢利用像Template::Toolkit这样的工具来让网页的版面跟程序代码分离的越干净越好。当然也有人认为像php形式的内嵌式作法可以让网站的雏型在很短的时间就产生出来，因此Mason也就以类似的作法诞生了。因此在这一，两年来，Mason已经成为非常重要的模块，尤其在作为网站的工具时。根据job.perl.org(一个专门张贴Perl相关工作机会的网站)上的信息，Mason已经是许多国外企业在征求网站相关程序开发人员时需求度很高的技术了。而且许多大型网站现在也都使用了Mason来产生他们的网页内容，例如亚马逊书店(http://www.amazon.com)就是一例。
Mason基本的操作原理是在你的Apache中加上一个控制器(Handler)，让使用者的要求全部送给Mason处理，这样一来，Mason就可以把各式各样的使用者需求都预先处理好，然后送出合适的内容。使用Mason，你除了装上HTML::Mason这个模块之外，你的Apache还必须支持mod_perl，在一切准备就绪之后，你可以在你的Apache中像这样的进行设定：

```
PerlModule HTML::Mason::ApacheHandler

<Location />
    SetHandler perl-script
    PerlHandler HTML::Mason::ApacheHandler
</Location>
```

在 Mason 中，你可以使用百分比(%)符号作为 Perl 程序代码的引导符号，或是<% ... $>。如果是一行的开始是由%引导，那么表示这一行是 Perl 程序代码。或是利用<% ... %>来设定一个区块的 perl 程序代码。所以你的网页可以像这样子：

```
<%perl>
my $num = 1;
my $sum = 0;
while ($num <= 10) {
    $sum+=$num;
}
</%perl>
总和是： <% $sum %>
```

那么结果就会在网页上呈现出计算后的总和，因此他已经把网页的 HTML 跟 perl 程序代码作了紧密的结合。尤其如果我们使用%作为程序行的起始，就更能表达出其中不可切割的关系了：

```
% $foo = 70;
% if ($foo >= 60) {
你的成绩及格了
% } else {
你被当了
% }
```

我们常使用这样的方式来把条件判断穿插在 HTML 里面，所以你的内容已经是夹杂各种语法的一个综合体了。而且不像 Template 使用自己定义的特殊语法，你在 Mason 中使用的大多是标准的 HTML 跟 Perl 语法(虽然他们总是夹杂在一起)。所以你当然可以这样写：

```
% my @array = (67, 43, 98, 72, 87);
% for my $grade (@array) {
你的成绩是： <% $grade %>
% }
```

接下来比较特殊的是一些 Maon 专用的组件，利用这些组件，你可以很容易的处理一些数据。例如使用者传来的需求就是其中一个很好的例子。在 Mason 中最基本的两个全域变量(每次使用者发出各种要求时，就会产生的两个变量)分别是 $r 跟$m。其中$r 是 Apache 传来的需求内容，至于$m 则是负责 Mason 自己的 API。所以你可以藉由$r 来取得由 Apache 传来的资料，例如：

```
$r->uri
$r->content_type
```

不过我们暂时先不管$m 这个负责处理 Mason API 的变量，因为我们还有更有趣的东西要玩。也就是在 Mason 页面中常常会被使用的<%args>...</%args>区块。这个区块可以用来取得由使用者藉由 POST/GET 传来的参数，一个很简单的例子当然就是像这样：

```
http://my.site/mason.html?arg1=value1&arg2=va
lue2

于是我们就在 mason.html 里面加上 args
区块，利用 ARGS 取得这些变量之后，我们就可以在页面中自
由使用了。

<%args>
$arg1
$arg2
</%args>
```

所以刚刚累加的程序，我们可以由使用者输入想要累加的数字，这时候，我们只要把 while 的结束条件利用使用者输入的变量来替换就可以了。

```
<%perl>
my $sum = 0;
while ($end > 0) {
    $sum+=$end;
}
</%perl>
总和:  <% $sum %>

<%args>
$end
</%args>
```

看来应该不是太困难，只是有点杂乱。如果我们每次都想要在页面开始之前，就先用 perl 进行一堆运算，判断的时候，可以把<%perl> ... </%perl>这一大段的区块搬离开应该属于 HTML 的位置吗？其实在 Mason 也替你想到了这个问题，所以在 Mason 中你可以使用<%init> ... </%init>这个区块，也就是进行初始化的工作。我们把刚刚的那一段页面的程序代码重新排列组合一下。

```
总和:  <% $sum %>

<%init>
my $sum = 0;
while ($end > 0) {
    $sum+=$end;
}
</%init>

<%args>
$end
</%args>
```

这样看起来显然干净多了，不过记得我们在使用 Template::Toolkit 的时候有一个很不错的概念，也就是[% INSERT %]/[% INCLUDE %]的方式，在 Mason 中也有类似，也就是利用<& ... &>的方式来加载你的自订组件。用个简单的例子来看：

```
<HTML>
<HEAD><TITLE>标题</TITLE></HEAD>
<BODY>
<TABLE>
% for my $grade (@grades) {
<TR><TD><% $grade %></TD></TR>
% }
</TABLE>
<HR>
copyright 2004-2005 Hsin-chan Chien
</BODY>
</HTML>

<%args>
@grades
</%args>
```

这时候，我们可以把前、后的 HTML 分别放到 header 跟 footer 两个地方，然后利用<& ... &>来加载，所以这个内容就会被改写为：

```
<& header &>
```

```
<TABLE>
% for my $grade (@grades) {
<TR><TD><% $grade %></TD></TR>
% }
</TABLE>
<& footer &>

<%args>
@grades
</%args>
```

这对于一整个网站维持所有网页中部份元素的统一是一种非常方便而且有用的方式。而且任何在独立的元素中被修改的部份也会在所有的页面一次更新，这绝对比起你一个一个档案修改要来得经济实惠许多。尤其当你所进行的是一个非常庞大的网站时，更能了解这种用法的重要性。

不可否认，我们花了这么多的页面来讲这三个目前在 Perl 社群中最被常用来进行网站程序的工具模块，却只能对每一个部份做非常入门的介绍。毕竟这三个模块都是非常复杂而且功能强大的。另外的特点就是他们都可以详细到各自出版一本完整的使用手册。不过对于一开始想要尝试使用这几个模块的人来说，事实上也能用阳春的功能帮你进行许多繁复的工作了。别忘了，大多数的 Perl 模块或语法，你只要了解他们的百分之二十，就可以处理百分之八十的日常工作。

习题：

1. 以下是一个 HTML 页面的原始码，试着写出 action 中指定的 print.pl，并且印出所有字段中，使用者填入的值。

```
<HTML>
    <HEAD>
        <TITLE>习题</TITLE>
    </HEAD>
    <BODY>
        <FORM ACTION="print.pl" METHOD="POST">
      姓名: <INPUT TYPE="text" NAME="name"><BR/>
      地址: <INPUT TYPE="text"
NAME="address"><BR/>
      电话: <INPUT TYPE="text" NAME="tel"><BR/>
      <INPUT TYPE="submit">
  </FORM>
    </BODY>
</HTML>
```

2. 承上题，试着修改刚刚的 print.pl，并且利用 Template 模块搭配以下的模板来进行输出。

```
<TABLE>
    <TR><TD>姓名：</TD><TD>[% name %]</TD></TR>
    <TR><TD>地址：</TD><TD>[% address
%]</TD></TR>
    <TR><TD>电话：</TD><TD>[% tel %]</TD></TR>
</TABLE>
```

3. 承上题，将利用 Template 输出的部份改为 HTML::Mason。

# 17. Perl与系统管理

Perl之所以能够一直在脚本程序中占有一席之地，有一些部份其实也是因为在系统管理中，Perl还能发挥着不错的效用，而且在使用上也是非常方便。它能够像shell script一样，拿了就直接用，而不需要定义一堆变量，对象，对象的方法之后才开始写程序代码。你可以找到相关的模块，然后非常迅速的完成你想达到的目的。所以它保留了shell script的方便性，却又比shell script拥有更多的资源。当然优点也在某些程度上被当为缺点，例如有些人认为Perl程序代码非常的不够严谨，因为相对于Java或Python，它显然太过自由了。
其实Perl对于Linux/*BSD的方便性几乎是不言可喻，很多时候在这些系统中都会预设安装了Perl，因为Perl能够相当程度的处理系统中的杂事。但是不只如此，如果你是一个系统管理员，Perl能帮你作的事情也许比你想象中的还要多。尤其当你需要对很多系统的数据进行搜寻，比对的时候，Perl就更能显示它的重要性。何况很多时候，我们可以透过日志文件(log)的分析来进行系统的监控跟效率的评比，而这也正是Perl的其中一项专长。这也就对于很多系统管理的统计部份(例如MRTG跟awstats)是藉由perl来达成能做出很好的解释了。
这一章的大纲是根据Autrijus Tang在对一群准Linux系统管理员上课前，我们一起讨论出来的结果，而内容则有许多是直接使用他上课中使用的范例。这些范例在原作中以开放文本的方式释出，各位可以直接使用在测试或其它正式的产品中。

## 17.1 Perl在系统管理上的优势

其实就像我们前面所提的，Perl在系统管理上有着非常重要的应用跟地位，对于许多Unix-like之类的操作系统管理来说，Perl经常是他们的好帮手。可是Perl到底有什么特别的优势呢？除了我们刚刚提到的文字比对，处理的优势外，Perl其实还有不少能够吸引人的地方。其中最特殊几点大概包括：

1. Perl的黏性特强：Perl被称为是一种黏胶程序语言，他能够用最省力的方式把各种东西绑在一起。作为一个系统管理员总是会大量接触各式各样的工具，因此需要整合这些工具的机会就非常的大，所以perl的优势在这个时候就能够容易的显现出来。
2. Perl资源丰富：我们之前多次提到的CPAN就是最丰富的数据库，不单单是网络程序，数据库处理，其实就像系统管理相关的部份，也有为数不少的模块。这些模块不但是许多系统管理员实际用来管理的工具，当然也是他们的经验。所以你看这些模块，不单单是找寻可供应用的工具，也可以藉此挖掘这些人管理系统的方式。
3. Perl的作业环境：除了Unix-like的各种操作系统能够轻易的安装，执行Perl之外，微软的Windows或是OS/2，以及苹果公司的Mac OS也都可以让Perl正常的运作。因此很多时候，当系统管理者同时必须管理一种以上的作业平台时，也能够轻易的使用相同的工具。
4. Perl几乎是一种标准：这里所谓的标准，其实是因为目前已经很多的系统管理

工具都是使用Perl所开发出来的，所以如果系统管理人员如果可以拥有Perl的能力，那么自己维护或修改这些系统的可能性就可以大大增加。

## 17.2 Perl的单行执行模式

很多系统管理员使用Perl当然是因为Perl的顺手跟方便，就像我们说的，你总不会希望找一个档案，或置换档案的某些字符串前还要先定义一大堆变量，或是先弄一个对象，然后拿来继承，再利用被继承的对象写出置换字符串的对象方法。然后又没有好的正规表示式，然后你可能得用substr一个一个去找出来。最惨的可能是你写完这样的程序已经是下班时间，所以你还得加班把需要的结果搞定。而且你知道，程序写出来不一定可以那么顺利，尤其你写了那么长的程序，有bugs也在所难免，然后……。于是，下场应该是可以预料的。
虽然很多人批评Perl非常不严谨，可是换个角度看，应该是说Perl允许你「随手」写出可以解决手边工作的工具。而且Perl的「随手」还不是普通的随手。因为Perl提供单行执行模式，所以你可能可以看到这样的一行程序：

```
perl -MEncode -pi -e '$_ =
encode_utf8(decode(big5 => $_))' *
```

执行完这一行之后，你该目录下的 Big5 档案就会全部都变成 UTF8 了。这实在非常神奇，不是吗？其实不只这样，你还可以利用一行的程序把档案中的某些字符串一次换掉。就像这样：

```
perl -pi.bak -e "s,foo,bar," *
```

这样一来，所有档案中的 foo 就全部被换成 bar 了。希望你没有继续想象如果你手动替换或是正在用其它结构严谨的程序语言在进行中。当然，如果你现在还在这里，我们就利用那些坚持一定要有足够完整的程序架构才能执行的家伙还在奋斗的时间，来看看怎么执行单行的 Perl 吧。
首先，最基本的就是"-e"这个参数了，你可以使用 perl -e 来执行一行程序。不过请注意，我说的是一行程序，而不是一个叙述句，所以你当然可以这么写：

```
perl -e "$foo = 2; print $foo"
```

接下来，虽然只是一行的 Perl 程序，可是你还是可以使用 Perl 的其它各式各样模块，如果没有办法使用模块，那单行的执行模式大概也没有人愿意使用了吧！所以我们使用了"-M"的选项来指定所要使用的模块，就像前面的第一个例子中，我们用了"-MEncode"来指定使用 Encode 这个模块。

接下来，我们要让这一行程序能对所有我们指定的档案工作，所以我们使用了"-p"这个选项这个选项允许使用者以循环的方式来执行这一行程序，在这里也就让我们可以使用*这个万用字符来指定所有的档案。接下来，我们看到另一个选项是"-i"，这是让 perl 会自动帮你进行备份，免得你对档案进行操作之后，结果非常不满意，却还得手动改回来。所以当我们指定了"-i.bak"就是让所有被修改的档案在修改前就先备份一个扩展名为.bak 的档案。不过这是小写 i，如果你用了大写的 I，那意义可就大不相同了，"-I"让你可以指定@INC 的内容。

## 17.3 管理档案

系统管理一开始大概都要面对一大堆的档案吧，而基本的档案管理其实还没有太过复杂，只要你的shell够熟，几乎也可以应付很多状况。例如你可以轻松的利用find找到你要的档案，甚至襄他们排序，就像这样：

```
find /home/hcchien/svn/ *.txt -print | sort
```

如果你用 Find::File::Rule 来写，可能会像这样：

```
#!/usr/bin/perl -w

use File::Find::Rule;

my $rule =  File::Find::Rule->new;
my @files =
$rule->file->name( '*.txt' )->in('/home/hcchi
en/');

print "$_$/" for @files;
```

Perl 的写法不但比较长，比较复杂，而且速度比起你在 shell 底下真是慢了好几倍。暂时看起来，Perl 在这样简单的状况下实在不特别好用。可是别忘了，像这样简单的状况，每个人都习惯随手就用 shell 解决，可是如果情况稍微复杂一点呢？比如我想找出档案状态是可执行的.txt 档案，那么你应该怎么做呢？接下来，我想把这些档案的可执行模式取消，然后也许再修改某些内容……。如果只是单一内容，shell 确实非常轻巧，快速，可是一但我们要把一堆操作集合在一起时，你就会发现 perl 有什么过人之处了。

```
#!/usr/bin/perl -w

use File::Find::Rule;
```

```
my $rule =  File::Find::Rule->new;
$rule->file;
$rule->executable;
my @files =
$rule->name( '*.txt' )->in('/home/hcchien/');

for my $file (@files) {
    open READ, $file;
    s/foo/bar/g while (<READ>);
}
```

当然，你还可以对这些档案做其它的操作，例如每个档案都插入一列新的数据等等。这时候，有另外一个模块就显得非常有用了，这么模块就是 IO::All。还记得我们之前怎么读入一个档案的内容吧？IO::All 现在可以让你非常简单的控制档案，我们来比较看看：

```
用传统的作法，我们可以这么写：

open FILE, "<foo.txt";
$buf.=$_ while (<FILE>);

倒也相当简洁，不过现在使用 IO::All，就只要这么做：

my $buf < io('foo.txt');
```

不过我们并不打算在这里深切的介绍 IO::All 这个模块，因为我们还有其它更重要的事情要做。

## 17.4 邮件管理

接下来让我们来看看作为服务器的一个重要工作，也就是关于Mail的管理。也因为对于邮件的管理需求其实非常的大，所以其实相关的管理工具也不在少数。例如可以过滤广告信件，发送大量信件或是寄发一般的通知信件等等。不过由于某些工作的特殊性质，使得Perl成为这些工作中非常能够胜任重要工具，这些工作中当然有不少是字符串内容的分析，而最具代表性的也就是广告信的过滤了。

### 17.4.1 Mail::Audit + Mail::SpamAssassin

这是一个非常具有杀手级实力的一个Perl模块，如果你已经是一个现任的网络管

理人员，整天听到你所管理的邮件服务器中不断传来广告邮件，而且你还不知道这个模块，那么应该先去CPAN上搜寻Mail::SpamAssassin这个模块。然后你还可以搭配Mail::Audit这个模块。这里的例子是许多人使用Mail::Audit跟Mail::SpamAssassin时经常会作为过滤信件第一关的检查工具：

```
use Mail::Audit;
use Mail::SpamAssassin;
my $m = Mail::Audit->new(
    emergency   => "~/emergency_mbox",
    nomime      => 1,
);
my $sa = Mail::SpamAssassin->new;
$m->pipe("listgate cle") if $m->from =~
/svk-devel/;    # 送到 pipe
$m->accept("~/perl")     if $m->from =~ /perl/;
# 接进特定信箱
$m->reject("no rbl")     if $m->rblcheck;
# 拒绝黑名单
$m->ignore              if $m->subject =~
/sex/i;      # 忽略信件
my $status = $sa->check($m);
# 检查垃圾信
if ($status->is_spam) {
    $status->rewrite_mail;
# 加上档头
    $m->accept("~/Mail/spam");
# 收进垃圾桶
}
$m->noexit(1); $m->accept("~/Mail/%Y%m");
$m->noexit(0);# 按月汇整
$m->accept;
# 其余接收
```

其实如果利用 Mail::Audit，还可以直接套用 Mail::Audit::MAPS。因为这样就可以直接把一些已经被列为黑名单的寄件者先排除在外了，而且使用上也没有什么太大的差别。其实你只需要多做一个判断:

```
if ($mail->rblcheck) {
    ......
}
```

另外，Mail::Audit 还有一个常用的外挂程序则是 Mail::Audit::KillDups。他可以帮你删除一些 ID 重复的信件，其实这也是可能的广告信来源之一，所以你可以又事先过滤掉一些垃圾邮件。其实不只这些，Mail::Audit 的外挂程序种类之多，实在让人觉得有趣，我自己另一个常用的是 Mail::Audi::PGP，不过这就未必适合所有人了。
至于 Mail::SpamAssassin，除了搭配 Mail::Audit 之外，其实也有命令列程序。透过命令列程序，你可以设定自己的黑名单(blacklist)跟白名单(whitelist)，也可以透过手动训练的方式，来增进 SpamAssassin 的准确率。

## 17.4.2 Mail::Sendmail 与 Mail::Bulkmail

另外，perl也可以用非常方便的方式来传送mail。虽然你在系统管理时，如果需要寄发mail，可以方便的使用sendmail来传送。可是其实有些时候你会在系统中定时执行一些程序，或许是进行系统的意外检查，或许是做定时的工作。那么你可能会希望这些工作如果发生意外，你可以很快的收到电子邮件通知。这时候Mail::Sendmail就变得很有用了。

```
use Mail::Sendmail;

%mail = ( To      => 'yourmail@hostname.com',
          From    => 'mail@server.com',
          Message => "救命啊！Apache 不动了！！"
        );

sendmail(%mail) or die $Mail::Sendmail::error;
```

各位大概都听过「自相矛盾」的成语故事吧？而且千万别以为那是书上拿来骗小孩的故事，因为真实的就发生在 perl 的模块中。我们之前介绍了目前几乎被公认最好阻挡广告信的模块：SpamAssassin，可是现在我们也要介绍另一个被认为发广告信的强力模块，也就是 Mail::Bulkmail。

```
use Mail::Bulkmail;
my $bulk = Mail::Bulkmail->new(
    LIST         => "~/listfile",      # 地址清单
    From         => 'admin@bulkmail.com',  # 寄
件人
    Subject      => "System Information",  # 标
题
```

```
    Message      => '~/announcement.txt'   # 内
文档名
    message_from_file => 1                 # 从档
案读取内文
);
$bulk->bulkmail() or die Mail::Bulkmail->error;
```

在这里，你只需要把要一次传送的一大堆邮件地址逐行放进一个纯文字文件。Mail::Bulkmail 就会帮你读出这些地址，而且发送邮件。至于内文，你虽然可以直接写在程序中，可是更有弹性的方式也是可以利用文字文件来读入邮件内容。这样一来，如果你的邮件是每周定时发送，那么你只需要修改传送名单跟邮件内容的档案就可以轻松的让程序替你完成其它工作了。

## 17.4.3 POP3Client 及 IMAPClient

有些时候，你可能有一些账号是专门用来处理一些特定的工作，那么其实这些送到该账号的信件未必需要由一个特定的人去收，而可以利用程序去进行处理。这个时候，你可以使用Mail::POP3Client这个模块来读取这些邮件。我们先用一个简单的例子来看看这个模块的用法：

```
use Mail::POP3Client;
my $pop = Mail::POP3Client->new(
    USER     => "me",
    PASSWORD => "mypassword",
    HOST     => "pop3.example.com",
);
foreach ( 1 .. $pop->Count-1 ) {
    $pop->Head( $_ ) =~
/^From:\s+somebody\@example.com/ or next;
    open my $fh, '>', "mail-$_.txt" or die $!;
    $pop->RetrieveToFile($fh, $_);
}
```

这个程序可以让某个人寄来的信都被备份到一个特定的档案中，其实主要的就是透过邮件档头中的寄件者进行比对。所以如果你可以针对主题进行比对，就可以对邮件进行分类。当然，如果你想出其它更好用的邮件过滤算法，也许可以从这里开始进行实做(虽然我们并不建议你这么做)。其实一但可以直接取出邮件中的每一封邮件，我们可以做的应用就非常的广泛了。而有了 POP3Client，好像也少不了 IMAPClient。我们还是先来看另一个例子吧：

```
use Mail::IMAPClient;

my $imap = Mail::IMAPClient->new;
$imap = Mail::IMAPClient->new(
             Server => $host,
                    User    => $id,
                    Password=> $pass,
       )      or die "无法连上主机：$host as $id:
$@";
```

使用 IMAPClient，也因为邮件服务器种类的特性差异，让它提供更多的操作方式给使用者。最基本的比如 IMAPClient 的 Connected 跟 Unconnected，另外可以透过 Range 来选定某一个特定范围的邮件。例如：

```
$imap->Range($imap->messages);
```

其中$imap->messages 会取得目前所在数据夹的所有邮件，然后透过 $imap->Range()来把这些邮件设定成要操作的邮件范围。不过你也可以在 $imap->Range()中使用逗点分隔来指定某几封特定的信件。另外，如果你考虑搬移信件，你可以使用 move 来进行，其实非常方便，就像这样：

```
my $newUid = $imap->move($newFolder, $oldUid)
         or die "Could not move: $@\n";
$imap->expunge;
```

另外，还有一些常用的方法例如 delete_messages 和 restore_messages 就分别是用来删除邮件跟回复被删除的邮件。或是你可以用 search 来搜寻邮件。其实 IMAPClient 非常的强大，确实需要根据自己的需求去研究相关的文件才能确实掌握。

## 17.5 日志文件

作为一个系统管理，能确实掌握每日的纪录文件确实是非常重要的工作。不过如果你进入/etc/log数据夹，就会发现里面的档案其实是相当多的。也就是说，如果你身为一个系统管理员，每天要注意这些档案中有没有异常，如果你还在使用传统的工人智慧，那么每天进行这样的工作就要浪费许多时间。因此要希望能够在这部份节省更多的时间来从事其它的管理工作，我们可以使用一些工作来简化

这些日常的程序，Parse::Syslog就是其中之一。
我们先来看看一个简单的例子，来了解Parse::Syslog怎么帮我们处理这些日志档案。

```
use Parse::Syslog;
my $syslog =
Parse::Syslog->new('/var/log/syslog');

while (my $entry = $syslog->next) {
    $entry->{program} =~ /sudo$/ or next;
    print localtime($entry->{timestamp})."\n",
        "$entry->{text}\n\n";
}
```

其实程序并不困难，首先我们设定要处理的档案，在这里我们针对"/var/log/syslog"这个档案来检查。接着 Parse::Syslog 传回一个对象，也就是$syslog。接下来我们使用 next 这个对象操作方法逐行处理这个档案，在档案结束之前，$syslog->next 都会传回真值，因此让我们可以一行一行来进行比对的工作。我们试图找出日志文件中关于有使用者使用 sudo 这个指令，所以我们使用了正规表示式。
接下来，Parse::Syslog 可以帮你取得事件发生的时间，所以当我们比对成功之后，就可以印出使用者使用 sudo 这个指令的时间。
所以如果我们一次把所有要监视的档案内容利用 Parse::Syslog 设定好，那么其实以后的日子就会轻松愉快许多了。当然，如果你撷取出这魔一堆档案，那么要把这些让人需要特别注意的纪录进行处理，才能方便管理者阅读或检查，这时候可以使用另一个模块来进行。
Log::Dispatch 是不错的选择。你只要新建立一个相关的对象，并且指定要写入的档名，那么可以依照纪录的等级来将相关的讯息写入档案中。

```
use Log::Dispatch;
my $log = Log::Dispatch->new;              # 建
立纪录对象
# 新增纪录文件
$log->add( Log::Dispatch::File->new(
    name => 'file',                       # 对象名
称
    min_level => 'debug',                  # 纪录
门坎
    filename => '/var/log/test.log',       # 纪
录文件名
```

```
) );

接下来，我们把想要写入的讯息像这样的加入档案中：

$log->log( level => 'alert', message => 'Strange
data in incoming request' );
```

这里有一个有趣的部份，也就是纪录门坎。他可以让使用者依照不同的等级来分门别类，而门坎的类别分别为：

```
除错 (debug)
消息 (info)
提示 (notice)
警告 (warning)
错误 (error)
关键 (critical)
警铃 (alert)
紧急事件 (emergency)
```

既然有了这些分别，你就可以进行更多的动作来确保系统正常的运作。例如你可以在系统遇到异常现象时发出信件给自己，就像这样：

```
$log->add( Log::Dispatch::Email::MailSend->ne
w(
    name => 'email',                        # 对象名
称
    min_level => 'emergency',                # 纪录
门坎
    to => [ 'admin@example.com' ],           # 收
件地址
    subject => 'HELP!!!',                   # 邮件
标题
) );
```

于是如果你的日志得到了一些紧急事件的讯息时，就会自动发出电子邮件给你。如此一来你可以在第一时间就知道系统异常，并且进行检修。当然，如果你是超级负责的系统管理员，你也许希望在你没有网络的时候也能得到相关的紧急警告，这时候手机简讯也许是另一种通知的好方法。

```
use Net::SMS;
my $df = `df -h`;          # 取得硬盘配置信息
$df =~ /-\d/ or exit;      # 若没有数字是负的就直
接离开
my $sms = Net::SMS->new;                # 简讯物件
$sms->accountId("123-456-789-12345");   # 账号
$sms->accountPassword("mypassword");    # 密码
$sms->sourceAddr("0928-000-000");       # 来源
$sms->destAddr("0928-999-999");         # 目的
$sms->msgData("HARD DISK FULL:\n$df");  # 讯息
$sms->submit;                           # 送出简讯
$sms->success or die $sms->errorDesc;   # 侦测错
误
```

我们使用 Net::SMS 可以直接发送简讯，在这个例子中，我们使用 shell 指令去检查磁盘空间。并且当发现磁盘空间不足时就发出手机简讯来警告系统管理人员。虽然你写了这些程序之后，未必就可以获得老板的赏识而加薪，不过我想让老板少点机会找你麻烦应该还算是一项大利多吧！

## 17.6 报表

身为系统管理人员，尤其当你是在企业内部进行系统管理时，最容易遭到忽略。因为当大家系统非常顺畅时，几乎没人会归功于系统管理员的认真。于是要怎么拿出实际的内容来说服其它的人，这也算是系统管理人员的重大业务之一了吧！很多人使用MRTG(注一)来监测网络流量，使用awstats(注二)来看网站的各式数据。

可惜的是大多数的人对这些报表并不太有兴趣，除非你管理的是一个(或一堆)网站，而且公司的业务也就是经营这些网站。既然如此，那么一个系统管理员有些时候其实还是要呈现出自己优良的管理绩效，在没有现成的套装工具下，要怎么样快速的建立出漂亮的报表其实也是不小的学问。

如果你需要的是文字的报表，那么我们前一章使用的Template就非常适合，你可以利用Template::Toolkit画一个HTML的报表。

```
#!/usr/bin/perl -w

use strict;
use Template;
use IO::All;
use Mail::MboxParser;
```

```
my $dir = io('/var/mail');                  # 准备
逐个检查信箱
my %all;
while (my $io = $dir->next) {               # 一个
一个看信箱的信件数目
   if ($io->is_file) {
# 只检查档案
       eval {                               # 避免
因为某些原因中断
          my $mb = Mail::MboxParser->new("$io",
newline => '#DELIMITER');
          %all{$io} = $mb->nmsgs;           # 把结
果放入杂凑
       }
   }
}

my $config = {
            INCLUDE_PATH => '/search/path',
            POST_CHOMP   => 1,
        };

my $template = Template->new($config);
# 建立新的模板对象
my $vars = { messages  => \%all };
my $input = 'report.html';
my $output;

$template->process($input, $vars, $output)  #
处理模板内容
     || die $template->error();
print $output;
```

这时候，你可以取得一个含有所有信箱邮件个数的一个杂凑变量(我们使用了 Mail::MboxParser)。如此一来，你只要弄一个漂亮的模板，就可以让系统动态把数据填入，随时可以监控目前大家信箱内的邮件数目了。不过这样其实还没结束，因为很多老板或主管对于文字的接受度总是比较低，所以如果你有漂亮的报表，那么给他们的印象应该也会随之提高。这时候，你也许可以认真考虑使用 GD::Graph 这个模块。这个模块可以让你轻松的画出漂亮的统计图表，你可以根据数据的属性以及需求的差异，画出例如圆饼图，曲线图，柱状图等等，如果你还想更绚丽，GD::Graph 还可以画出立体的 3D 图形。
我们用另一个例子来看看怎么使用 GD::Graph 吧：

```
use GD::Graph::bars3d;
my $graph = GD::Graph::bars3d->new(800, 600);
# 新增柱状图
my @files = </var/log/maillog.*.bz2>;

my $image = $graph->plot([                # 订
出横坐标，纵坐标内容
    [map /(\d+)\./g, @files],
    [map -s, @files],
]) or die $graph->error;

open my $fh, '>', '3.png' or die $!;
print $fh $image->png;     # 储存影像
```

我们取得了每次的电子邮件日志文件的，然后根据这些档案的大小进行统计。这时候，我们只需要订出横坐标跟纵坐标的内容，交给 GD 去画就好了，你当然也可以定时的要求程序帮你画出某些统计图表。很有用吧！你可以定时交出漂亮的工作报表，而且还是由计算机自动产生。

利用 Perl 来协助系统管理其实还算是非常方便的，何况已经有许多的系统管理员早就在做这些工作，也因此我们有很多方便的工具可以使用。这完全让我们省下许多时间，尤其当更多的系统管理员每天都花许多时间在这些繁琐而且又没有变化的工作上。

习题：
1. 找出 maillog 中被 reject(退信)的数据，也就是找到日志文件中以 reject 标明的内容。例如：

```
   Jun  3 00:00:46 dns2 postfix/smtpd[71431]:
D988D6A: reject: RCPT
   from smtp2.wanadoo.fr[193.252.22.29]: 450 <
   fnatterdobkl@hcchien.org>: User unknown in
local recipient
   table; from=<>
to=<fnatterdobkl@hcchien.org>
   proto=ESMTP helo=<mwinf0203.wanadoo.fr>
```

2. 承上题，统计当月每天的退信数字，并且画成长条图。

注一：Multi Router Traffic Grapher
(http://people.ee.ethz.ch/~oetiker/webtools/mrtg/)
注二：http://awstats.sourceforge.net/

# 附录A. 习题解答

第一章：
1. 试着找出你计算机上的Perl版本为何。

解答：当然，你得先确定你的计算机上确实装了Perl。如果你在Unix/Linux/*BSD或是Mac OS X上，打开你的终端机(terminal)，进入shell，接着打入perl -v，其中第一行中就可以看到你计算机上的Perl版本了。详细的内容可以参阅第一章内容。

2. 利用perldoc perl找出所有的perl文件内容

解答：当你能看到perl -v的内容之后，你的计算机应该已经安装Perl。接下来，你可以在shell中打入perldoc perl。于是你可以看到所有的文件，像这样：

```
      Overview

          perl              Perl overview (this
section)
          perlintro          Perl introduction
for beginners
          perltoc            Perl documentation
table of contents

     ...........

          perluts            Perl notes for UTS
          perlvmesa          Perl notes for
VM/ESA
          perlvms            Perl notes for VMS
          perlvos            Perl notes for
Stratus VOS
          perlwin32          Perl notes for
Windows
```

至于如果你想看其中的任何一份文件，只要使用 perldoc 这个指令即可，例如可以使用 perldoc perlsyn 来看关于 Perl 语法的相关文件。

3. 利用 Perl 写出第一个程序，印出你的名字

解答：你只需要使用 print 就可以解决这个问题，所以像是这样：

```
print "简信昌";
当然，你还可以用单引号，至少在这里的用法是一样的：
print '简信昌';
```

第二章：
1. 使用换行字符，将你的名字以每个字一行的方式印出。

解答：最简单的方式，你可以这么写

```
print "简\n 信\n 昌\n";
另外，你当然可以逐行印出：
print "简\n";
print "信\n";
print "昌\n";
```

2. 印出'\n', \t'字符串。

解答：你可以单纯的使用单引号

```
print '\n \t";
或是使用双引号，然后加上跳脱字符：
print "\\n \\t";
```

3. 让使用者输入姓名，然后印出包含使用者姓名的招呼语(例如：hello xxx)。

解答：这里主要是要能够让使用者输入，所以我们应该使用<STDIN>

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
print "Hello $input \n";
```

第三章：
1. 试着把串行 (24, 33, 65, 42, 58, 24, 87) 放入数组中，并让使用者输入索引值 (0…6)，然后印出数组中相对应的值。

解答：在这里，我们并不先对输入值做判断，也就是假设使用者都会乖乖的输入 0…6 的数字。

```
#!/usr/bin/perl

use strict;

my @array = (24, 33, 65, 42, 58, 24, 87);
chomp(my $input = <STDIN>);     # 使用者输入
print $array[$input];
```

2. 把刚刚的数组进行排序，并且印出排序后的结果。

解答：这部份其实只需要使用一个排序的函式 sort。

```
print sort @array;
```

3. 取出数组中大于 40 的所有值。

解答：至于这一个部份，我们则是可以使用 gerp 这个函式直接完成：

```
print grep {$_ > 40} @array;
你当然还可以把过滤出来的值再进行排序，就像这样：
print sort grep {$_>40} @array;
```

4. 将所有数组中的值除以 10 后印出。

解答：至于要把一个数组中的所有值同时进行某种转换，对应，就可以使用 map

```
print map {$_/10} @array;
同样的，你还是可以试着将结果排序
```

第四章：
1. 算出 1+3+5+…+99 的值。

解答：我们可以使用 for 循环或是 while 循环来进行。

```
$!/usr/bin/perl
```

```
use strict;
my $sum = 0;
for (my $i = 0; $i < 100; $i+2) {
    $sum+=$_;
}

print $sum;
```

如果使用 while，那么程序代码应该像是这样

```
#!/usr/bin/perl

use strict;

my ($sum,$i);
while ($i < 100) {
    $sum+=$i;
    $i++;
}
print $sum;
```

2. 如果我们从 1 加到 n，那么在累加结果不超过 100，n 的最大值应该是多少？

解答：这时候，我们用 while 循环似乎就比较方便了

```
#!/usr/bin/perl

use strict;

my ($sum, $i);
while ($sum <= 100) {
    $sum+=$i;
    $i++;
}

print $i;
```

3. 让使用者输入一个数字，如果输入的数字小于 50，则算出他的阶乘，否则就印出数字太大的警告。

解答：这里有两个重点，一个是 if 判断式，另一个则是计算阶乘的循环。

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
if ($input < 50) {
   my $total = 1;     # 这跟算总和不同
   for (my $i = 1; $i <= $input; $i++) {
     $total*=$i;    # 进行阶乘
   }
   print $total;
} else {
   print "数字太大了";
}
```

第五章：

1. 将下列数据建立一个杂凑：

John => 1982.1.5
Paul => 1978.11.3
Lee => 1976.3.2
Mary => 1980.6.23

解答：我们可以很简单的使用串行，或是=>来建立杂凑：

```
my %hash = (   John => "1982.1.5",
           Paul => "1978.11.3",
           Lee  => "1976.3.2",
           Mary => "1980.6.23" );
```

至于如果使用串行，则非常单纯的只要：

```
my %hash = qw/John 1982.1.5 Paul 1978.11.3 Lee
1976.3.2 Mary 1980.6.23/;
```

2. 印出 1980 年以后出生的人跟他们的生日。

解答：我们逐个取出杂凑的键值，然后比较数据。

```
my %hash = qw/John 1982.1.5 Paul 1978.11.3 Lee
1976.3.2 Mary 1980.6.23/;
while ( ($key, $value) = each %hash) {
```

```
    print "$key, $value" if ($value gt "1980");
}
```

3. 新增两笔资料到杂凑中：
Kayle => 1984.6.12
Ray => 1978.5.29

解答：要新增杂凑中的内容很简单，只需要单纯的指定键跟对应的值就可以了。

```
$hash{Kayle} = '1984.6.12';
$hash{Ray} = '1978.5.29';
```

4. 检查在不修改程序代码的情况下，能否达成第二题的题目需求

解答：由于我们使用 while 循环，它会自动检查杂凑中所有的内容，因此即使我们新增了两笔数据，对于循环的运作并不会有所影响。

第六章：
1. 下面有一段程序，包含了一个数组，以及一个副例程 diff。其中 diff 这个副例程的功能在于算出数组中最大与最小数值之间的差距。请试着将这个副例程补上。

```
#!/usr/bin/perl -w

use strict;

my @array = (23, 54, 12, 64, 23);
my $ret = diff(@array);
print "$ret\n";     # 印出 52 (64 - 12)
my @array2 = (42, 33, 71, 19, 52, 3);
my $ret2 = diff(@array2);
print "$ret2\n";  # 印出 68 (71 - 3)
```

解答：我们需要进行的工作包括读取透过副例程传来的数组内容，并且取得数组中的最大值与最小值，再进行两个值的计算。

```
sub diff {
    my @param = @_;
    my ($max, $min) = ($param[0], $param[0]);
    for (@param) {
```

```
      $max = $_ if $_ > $max;           # 求最大
值
  $min = $_ if $_ < $min;         # 求最小值
    }
    $max - $min;
}
```

2. 把第四章计算阶乘的程序改写为副例程型态，利用参数传入所要求得的阶乘数。

解答：我们先来看看第四章中关于阶乘的这段程序代码。

```
my $total = 1;
for (my $i = 1; $i <= $input; $i++) {
    $total*=$i;
}
print $total;
```

接下来我们将它改为副例程型态：

```
sub times {
    my $input = shift;                  # 取得使
用者传入的参数
    my $total = 1;
    for (my $i = 1; $i <= $input; $i++) {  # 计
算阶乘
      $total*=$i;
    }
    return $total;
}
```

第七章：

1. 让使用者输入字符串，并且比对是否有 Perl 字样，然后印出比对结果。

解答：这里只需要使用最单纯的样式比对来判断比对的结果。

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
if ($input =~ /Perl/) {
```

```
    print "比对成功\n";
} else {
    print "比对失败\n";
}
```

2. 比对当使用者输入的字符串包含 foo 两次以上时(foofoo 或是 foofoofoo 或是 foofoofoofoo...)，印出比对成功字样。

解答：使用群组比对的方式似乎可以简单的达到这个要求，所以设定样式为 foo。请注意，你不能将样式设定为 foofoo，否则如果 foo 的出现次数是单次(例如三次)的话，那就无法正确比对了。所以我们的写法可以像这样：

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
if ($input =~ /(foo){2,}/) {     # 必须出现两次
以上
    print "比对成功\n";
} else {
    print "比对失败\n";
}
```

第八章：
1. 延续第七章的第一题，比对出 perl 在字符串结尾的成功结果。

解答：和第七章的第一个问题不同的是在于我们必须使用定位点的概念。所以我们只写出需要进行字符串结尾的样式。

```
$input =~ /perl$/;
```

2. 继续比对使用者输入的字符串，并且确定是否有输入数字。

解答：这个问题主要的部份在于可以使用字符集或是字符集的简写。最简单的当然是直接使用\d 的简写形式。

```
$input =~ /\d/;
而其实也就是可以使用
$input =~/[0-9]/;
```

3. 利用回溯参照，找出使用者输入中，引号内(包括双引号或单引号)的字符串。

解答：在这里，我们特别要求不管使用者使用单引号或双引号时都可以可以找出引号中的字符串。因此在比对时，就必须使用字符集，也就是['"]必须同时被纳入。但是一但使用字符集来进行比对，为了避免产生错误的对称，例如"单引号'"，我们就必须使用回溯参照，以确定我们比对的是对称的引号。另外，则是要注意使用记忆变量来取得我们比对出来的内容。所以比对的样式应该可以这么写：

```
$input =~ /['"](.+?)\1/;
print $1;            # 比对出来的内容
```

我们还要注意括号里的内容，首先是一个万用符号.，接下来是重复符号+，这是指至少出现一次的重复符号。接下来是为了避免比对超过第一次对称的引号范围，所以我们用了不贪多的修饰符号'?'。而这里面正是我们所要取得的全部内容，所以就用了记忆变量的符号。

4. 找出使用者输入的第一个由 p 开头，l 结尾的英文字。

解答：这里我们要确定的有几个部份，也就是我们要比对的是一个「字(word)」，因此 p 跟 l 分别是这个字的两个端点，我们也就可以利用\b 来画出这个字的界线。当然，记忆变量还是需要的，因为我们不但要确定是否比对成功，因为我们还想取得比对成功的字符串内容。所以我们就把比对样式写成这样：

```
$input =~ /\b(p\w*l)\b/;
print $1;
```

第九章：
1. 陆续算出 (1...1) 的总和，(1...2) 的总和，...到 (1...10) 的总和。但是当得到总和大于 50 时就结束。

解答：这个题目主要有两个部份，第一个是关于计算加总的部份，一般我们也许常用 for 循环来进行加总的部份，当然你也可以使用 while 循环或其它方式。接下来，你要考虑计算出来的总和，让他不超过 50。这个情况下，可以使用 last 来做循环的饿外控制。

```
#!/usr/bin/perl -w

use strict;

my ($base, $sum) = (0, 0);
```

```
for $base (1...10) {
    $sum = sum($base);
    last if ($sum > 50);
    print "$base => $sum\n";
}

sub sum {
    my $index = shift;
    my $summary;
    for (1...$index) {
        $summary += $_;
    }
    return $summary;
}
```

2. 把下面的程序转为三元运算符形式：

```
#!/usr/bin/perl -w

use strict;

chomp(my $input = <STDIN>);
if ($input < 60) {
    print "不及格";
} else {
    print "及格";
}
```

解答：这部份其实只要考虑 if 叙述句内的部份。所以我们先找出关键的部份，也就是 if 条件跟 else 的内容。接下来就只需要一一对照转换就可以了。

```
原来写法：      if ($input < 60) { print "及格" } else
{ print "不及格" }
三元算符写法：   ($input < 60) ? print "及格" : print
"不及格";
```

第十章：

1. 试着将下面的数据利用 perl 写入档案中：

```
Paul, 26933211
```

```
Mary, 21334566
John, 23456789
```

解答：这里主要的重点就是开启档案，写入内容，至于资料，我们可以使用杂凑来处里。

```
#!/usr/bin/perl -w

use strict;

my %tel = ("Paul", 26933211, "Mary", 21334566,
"John", 23456789);
open FILE, ">telephone";
for (keys %tel) {
   print FILE "$_ => $tel{$_}\n";
}
close FILE;

然后你可以去看档案"telephone"的内容，也就是：

John => 23456789
Mary => 21334566
Paul => 26933211
```

2. 在档案中新增下列数据：

```
Peter, 27216543
Ruby, 27820022
```

解答：刚刚我们在开启档案时使用了">"来表示写入一个新档。接下来，我们只是要在现有的档案中加入新的内容，因此我们应该改用">>"的方式，以避免原来的档案被清空。

```
#!/usr/bin/perl -w

use strict;

my %tel = ("Peter", 27216543, "Ruby", 27820022);
open FILE, ">>telephone";
for (keys %tel) {
```

```
    print FILE "$_ => $tel{$_}\n";
}
close FILE;
```

这样我们就可以很容易的看出其中的不同了。

3. 从刚刚已经存入数据的档案读出档案内容，并且印出结果。

解答：不同于刚刚写入档案，我们现在需要的是把档案内容读出。

```
#!/usr/bin/perl -w

use strict;

open FILE, "telephone";
while (<FILE>) {
    print $_;
}
close FILE;
```

第十一章：

1. 列出目前所在位置的所有档案/数据夹名称。

解答：我们可以用简单的角括号方式来取得目前目录下的所有内容。

```
#!/usr/bin/perl -w

use strict;

my @files = <*>;
print "$_\n" for @files;
```

2. 承一，只列出数据夹名称。

解答：在这里，我们只需要修改刚刚的程序，在打印前判断我们取得的是档案或数据夹。

```
my @files = <*>;
for (@files) {
    print "$_\n" if (-d $_);
}
```

3. 利用 perl，把目录下所有附文件名为.pl 的档案修改权限为可执行。

解答：首先我们还是使用角括号，但是我们这次要取出的只有所有附档名为.pl 的档案。接下来，再以 chmod 来修改权限。

```
my @files = <*.pl>;
chmod 0755, @files;
```

第十二章：
1. 让使用者输入字符串，取得字符串后算出该字符串的长度，然后印出。

解答：这里主要还是要使用 length 这个函式，来取得字符串长度。

```
#!/usr/bin/perl -w

use strict;

chomp(my $str = <STDIN>);
print length($str);
```

2. 利用 sprintf 做出货币输出的表示法，例如：136700 以$136,700，26400 以 $26,400 表示。
3. 利用杂凑%hash = (john, 24, mary, 28, david, 22, paul, 28)进行排序，先依照杂凑的值排序，如果两个元素的值相等，则依照键值进行字符串排序。

第十三章：
1. 试着在你的 Unix-like 上的机器装起 CPANPLUS 这个模块。

解答：你可以直接透过 CPAN 来安装 CPANPLUS，或是到 http://search.cpan.org/ 下载 CPANPLUS 的原始码，解开之后直接安装。成功安装之后，你可以在 shell 底下使用 CPANPLUS，就像这样：

```
[hcchien@Apple]% cpanp
CPANPLUS::Shell::Default -- CPAN exploration
and modules installation (v0.03)
*** Please report bugs to
<cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS::Backend v0.049.
```

```
*** ReadLine support available (try 'i
Term::ReadLine::Perl').

CPAN Terminal>
```

2. 还记得我们写过阶乘的副例程吗？试着把它放入套件 My.pm 中，并且写出一个程序呼叫，然后使用这个副例程。

解答：其实如果知道 Package 的包装方式跟使用方式，这个问题可以很容易的解决。

```
sub times {
   my $input = shift;                    # 取得使
用者传入的参数
   my $total = 1;
   for (my $i = 1; $i <= $input; $i++) {  # 计
算阶乘
     $total*=$i;
   }
   return $total;
}
```

第十四章：
1. 下面程序中，%hash 是一个杂凑变量，$hash_ref 则是这个杂凑变数的参照。试着利用$hash_ref 找出参照的所有键值。

```
%hash = ( name => 'John',
         age => 24,
    cellphone => '0911111111' );
$hash_ref = \%hash;
```

解答：其实你只要解开杂凑参照，就可以简单的使用 keys 函式来取得参照的所有键。

```
#!/usr/bin/perl -w

use strict;

my %hash = ( name => 'John',
           age => 24,
```

```
           cellphone => '0911111111' );
my $hash_ref = \%hash;

my @keys = keys %{$hash_ref};
print $_ for @keys;
```

2. 以下有一个杂凑，试着将第一题中的杂凑跟这个杂凑(@hash_array)放入同一数组中。

```
%hash1 = ( name => 'Paul',
           age => 21,
     cellphone => '0922222222',
     birthday => '1982/3/21' );
```

解答：由于数组中的元素都是纯量，所以我们需要的是把两个杂凑的参照放进数组@hash_array 中。

```
my @hash_array = ( { name => 'John',
                     age => 24,
                   cellphone => '0911111111' },
                 { name => 'Paul',
                   age => 21,
             cellphone => '0922222222',
             birthday => '1982/3/21' } );
```

3. 承上一题，印出数组$hash_array 中每个杂凑键为'birthday'的值，如果杂凑键不存在，就印出「不存在」来提醒使用者。

解答：在这里，我们应该先从数组中依序取出杂凑的参照，然后解开参照，判断参照键'birthday'是否存在。如果存在就可以取出其中的杂凑值。

```
#!/usr/bin/perl -w

use strict;

my @array = ( { name => 'John',
                age => 24,
              cellphone => '0911111111' },
        { name => 'Paul',
              age => 21,
```

```
       cellphone => '0922222222',
       birthday => '1982/3/21' } );

for (@array) {
   if (exists ${$_}{birthday}) {  # 解开参照，并且判断杂凑键是否存在
  print ${$_}{birthday};
    } else {
  print "the key doesn't exist";
    }
}

其实你可以用更简洁的方式来解开参照，也就是
$_->{birthday}
```

第十五章：

1. 利用自己熟悉的数据库系统(例如 MySQL 或 Postgres)，建立一个数据库，并且利用 DBI 连上数据库，取得 Database Handler。

解答：假设我们在 MySQL 建了一个数据库叫做'perlbook'所以我们要连接上数据库，就只要使用 DBI。

```
my $dbh =
DBI->connect('dbi:mysql:dababase=perlbook',
'user', 'password');
```

2. 试着建立以下的一个数据表格，并且利用 Perl 输入数据如下：

```
数据表格：
name: varchar(24)
cellphone: varchar(12)
company: vrchar(24)
title: varchar(12)
数据内容
[ name: 王小明
  cellphone: 0911111111
  company: 甲上信息
  title: 项目经理 ]
[ name: 李小华
  cellphone: 0922222222
  company: 乙下软件
```

```
  title: 业务经理 ]
```

解答：建立数据表格，我们可以透过各种方式，例如 MySQL 的客户端程序，或现成的管理程序。当然也可以利用 DBI 的方式来建立新的数据表格。

```
#!/usr/bin/perl -w

use strict;
use DBI;

my $dbh =
DBI->connect('dbi:mysql:database=perlbook',
'user', 'password');

my $create = <<"END";
CREATE TABLE address (
    name varchar(24),
    cellphone varchar(12),
    company vrchar(24),
    title varchar(12)
);
END

$dbh->do($create) or die "can't create";    # 先
把数据表格建起来

my $sql;
$sql = "INSERT INTO address VALUES ('王小明',
'0911111111', '甲上信息', '项目经理')";
$dbh->do($sql);
$sql = "INSERT INTO address VALUES ('李小华',
'0922222222', '乙下软件', '业务经理')";
$dbh->do($sql);
```

3. 从数据库中取出所有数据，并且利用 fetchrow_array 的方式逐笔印出数据。

解答：和新增数据不同，一般我们要从数据库抓数据出来，都会先使用 prepare，然后 execute 之后才取得数据内容。所以写法和刚刚会有不少的差别。

```
#!/usr/bin/perl -w
```

```
use strict;
use DBI;

my $dbh =
DBI->connect('dbi:mysql:database=perlbook',
'user', 'password');

my $sql = "select * from address";
my $sth = $dbh->prepare($sql);
$sth->execute;                          # 先取得
所有的内容
while (my @result = $sth->fetchrow_array) {  #
逐笔取出
    print "姓名: $result[0]\n";
    print "电话: $result[1]\n";
    print "公司: $result[2]\n";
    print "职称: $result[3]\n";
}

$dbh->disconnect;
```

4. 呈上题，改利用 fetchrow_hashref 进行同样的工作。

解答：在这里，我们只需要修改 while 循环内的程序代码。将原来使用 fetchrow_array 的部份改成使用 fetchrow_hashref 就可以了。当然，因为 fetchrow_hashref 拿到的是一个杂凑参照，所以我们得先解开参照，然后取得其中的值。

```
while (my $result = $sth->fetchrow_hashref) {  #
逐笔取出
    print "姓名: $result->{name}\n";
    print "电话: $result->{cellphone}\n";
    print "公司: $result->{company}\n";
    print "职称: $result->{title}\n";
}
```

第十六章：
1. 以下是一个 HTML 页面的原始码，试着写出 action 中指定的 print.pl，并且印出所有字段中，使用者填入的值。

```
<HTML>
```

```
    <HEAD>
        <TITLE>习题</TITLE>
    </HEAD>
    <BODY>
        <FORM ACTION="print.pl" METHOD="POST">
      姓名: <INPUT TYPE="text" NAME="name"><BR/>
      地址: <INPUT TYPE="text"
NAME="address"><BR/>
      电话: <INPUT TYPE="text" NAME="tel"><BR/>
      <INPUT TYPE="submit">
  </FORM>
    </BODY>
</HTML>
```

解答：基本上，这个题目我们想要的就是取得使用者输入的内容，所以利用 CGI 模块就可以简单的做到这件事。

```
#!/usr/bin/perl -w

use strict;
use CGI;

my $q = CGI->new;
print "姓名: ".$q->param('name')."\n";
print "地址: ".$q->param('adress')."\n";
print "电话: ".$q->param('tel')."\n";
```

2. 承上题，试着修改刚刚的 print.pl，并且利用 Template 模块搭配以下的模板来进行输出。

```
<TABLE>
    <TR><TD>姓名: </TD><TD>[% name %]</TD></TR>
    <TR><TD>地址: </TD><TD>[% address
%]</TD></TR>
    <TR><TD>电话: </TD><TD>[% tel %]</TD></TR>
</TABLE>
```

解答：这里的主要工作就是把 Template 的对象建起来，这样一来，我们就可以使用 Template::Toolkit 来建立漂亮的模板。我们假设把上面的模板存成 template.html。

```
#!/usr/bin/perl -w

use strict;
use Template;
use CGI;

my $q = CGI->new;
my $config = {
    INCLUDE_PATH => './',
    EVAL_PERL    => 1,
};

my $template = Template->new($config);
my $vars = {
    name  => $q->param('name'),
    address => $q->param('address'),
    tel => $q->param('tel')
};

my $temp_file = 'template.html';
my $output;
$template->process($temp_file, $vars, $output)
    || die $template->error();

print $output;
```

3. 承上题，将利用 Template 输出的部份改为 HTML::Mason。

解答：我们假定各位的 HTML::Mason 都设定完成，也就是其实目前都可以执行 HTML::Mason 的相关程序。因此我们接下来需要的只是处理这一页的 Mason 程序。

```
<TABLE>
    <TR><TD>姓名: </TD><TD><% $name %></TD></TR>
    <TR><TD>地址: </TD><TD><% $address
%></TD></TR>
    <TR><TD>电话: </TD><TD><% $tel %></TD></TR>
</TABLE>

<%args>
$name
```

```
$address
$tel
</%args>
```

第十七章：

1. 找出 maillog 中被 reject(退信)的数据，也就是找到日志文件中以 reject 标明的内容。例如：

```
Jun  3 00:00:46 dns2 postfix/smtpd[71431]:
D988D6A: reject: RCPT from
smtp2.wanadoo.fr[193.252.22.29]: 450
<fnatterdobkl@hcchien.org>:
User unknown in local recipient table; from=<>
to=<fnatterdobkl@hcchien.org> proto=ESMTP
helo=
<mwinf0203.wanadoo.fr>
```

解答：对于系统的日志文件而言，其实最有利的大多还是格式的固定(规则)化。所以我们可以比较容易的处理这些日志文件，进而用比较轻松的方式取得我们需要的数据。在这里，我们发现邮件服务器的日志文件格式是以': '来作为区隔。所以如果我们把每一笔数据(一列)视为一个字符串，利用 split 来将字符串切开为包含各字段的数组的话，我们就发现数组的第三个元素就可以用来判断是否为退信的数据，因此这样就显得容易多了。让我们来试试看：

```
#!/usr/bin/perl -w

use strict;

my $file = "/var/log/mail.log";
open LOG, $file;
while (<LOG>) {
    my @columns = split /: /, $_;
    print $_ if ($columns[2] eq 'reject');
}
close LOG;
```

2. 承上题，统计当月每天的退信数字，并且画成长条图。