PERL 编程思想

罗刚 hoowa 编著

2003，2004

第1章 PERL 简介

Perl 是一门以处理文本和文件见长的语言。它是一门易上手，开发快速的工具。相对于其它的脚本语言，如 VBScript 等，有不可比拟的优势。而且 Perl 作为一个源码开放的工具，给爱好钻研的程序员提供了更深层次的发挥空间。本节将介绍 Perl 能够做什么（使用范围）以及它是怎么做的（Perl 脚本执行的基本过程）。

1.1 使用范围

Perl 一致公认的强项在于文本处理。internet 和生物信息两个最经常使用 Perl 的领域都因为此。

在 internet 网方面，人们用它来构建网站，以及构建互联网搜索引擎。很多专业的网站采用了 Apache+Perl 组合。

在生物信息处理方面，很多基因序列拼接程序都是用 Perl 写成。

在数据库领域，有很多著名的软件把 perl 打包在他们的软件包中。例如数据库服务器 oracle10g、ETL 软件 informatica、数据仓库软件 Teradata。

除此之外，更多的人使用 Perl 的原因是把它当成一门工具性的语言。它的优点在于不需要编译成可执行文件的灵活性和广泛的可获得性。在 Unix 下，你可以用它替换 shell、awk、sed 等工具。例如，替换文本中的内容或取得文本中的某一列等。在 windows 下，你可以用它执行 VBScript、JavaScript 等脚本语言执行的工作。例如生成 Makefile 文件或以 PerlScript 的形式嵌入在网页中。

1.2 工作原理

通俗的说，Perl 是一种类似 basic 的脚本语言。专业化一点来说，Perl 是一种字节编译语言，并且还是一个字节解释器。它不会象 unix 中的 shell 读程序一样，对程序进行逐行执行。相反，Perl 会先通读一遍文件，将其编译为内部表达式，然后执行指令。

虽然 Perl 是一种脚本语言，但是在所有的脚本语言中，它的执行速度可能是最快的。因为 Perl 本身是采用 C 语言开发，很多模块也是使用 C 语言开发的。换句话说，Perl 执行某项指令可能是直接调用 C 语言开发的函数。

在编译的同时，也进行了一些代码的优化，例如，消除了不可能执行的代码，计算了常量表达式，加载了库定义。

1.3 执行程序

在 windows 下执行 perl 程序。

>perl sample.pl

或利用 windows 的文件扩展名关联：

>sample.pl

或

>perl < sample.pl

在 Unix 下执行。可以使用：

>perl sample.pl

或关联的方式：

>chmod 0777 sample.pl
>sample.pl

因为 sample.pl 代码中的第一行指明了关联方式：

#!/ur12/power/Perl58/bin/perl

perl 通过如下方式查找要执行的程序：

1. 直接包含在命名行中的程序，通过-e 开关指定。例如：

> perl -e "print 'hello world!'"

2. 通过在命名行指定脚本程序文件名。例如：

>perl sample.pl

3. 通过标准输入输入程序。例如：

>perl < sample.pl

第2章 基本概念

本章的目的是：在对 Perl 进行了基本的介绍之后，你可以立即使用它而不需要额外的参考。当然，在阅读本章的同时，查阅 Perl 文档以及使用 google 搜索想要的答案也是不错的主意。

这章首先介绍 Perl 的安装，然后看看安装包的内容。在正式使用 Perl 之前，我们需要知道怎样阅读 Perl 文档，准备一个方便的开发环境，了解 Perl 编程常用的命名规范。然后是几乎任何一门计算机语言都要介绍的常量、变量、数据结构，操作符，控制流。接着是 Perl 中常用的文件与目录操作，定义与执行一个例程，如何执行一个操作系统命令，如何定义正则表达式，如何定义与使用输出格式，Perl 文档介绍。最后是模块的创建与使用。

2.1 windows 下安装

由于 Perl 是开放源码的软件，安装可以采用二进制包的安装方式，也可以采用编译源代码方式。

最简单的方式是从 http://www.activeperl.com 处可取得 windows 版的安装。

如果想要编译源代码，可以从 http://aspn.activestate.com/ASPN/Downloads/ActivePerl/Source 或 http://www.cpan.org/index.html 取得源代码。

如果要在 win2000 平台下编译 perl5.8 源代码，先释放源代码到一个临时目录，然后找到\win32 目录下的 Makefile。可在此文件中，更改 perl 的安装路径等配置信息。最后在\win32 目录下运行 nmake 即可。

2.2 Unix 下安装

很多 Unix 环境都带有 Perl。如果版本太旧，可以自己安装。从 http://www.activeperl.com 处可取得 Solaris 版和 Linux 版的安装。其它的安装版本可以到 http://www.cpan.org/ports/查找。

1．解压

>gzip -dc ActivePerl-5.8.0.806-sun4-solaris.gz | tar -xof –

2．安装

>install.sh

你可以指定安装的路径，例如：/ur12/power/Perl58。安装完成后，把/ur12/power/Perl58/bin

加到环境变量 PATH，/ur12/power/Perl58/man 加到环境变量 MANPATH。如果已有旧版的 perl，确保把新安装 perl 路径加到的 PATH 的最前面就行了。这样做是一种风险最小的升级方式，因为在整个系统中，往往还有其它的程序需要使用旧版的 perl。这样既能使用到最新的版本，又不会影响到其他用户。

>*echo $PATH*

最后确认 perl 的版本：

>*perl –v*

或位置：

>*which perl*

2.3 Active Perl 目录介绍

目录	说明
C:\Perl	根目录。
C:\Perl\bin	可执行文件目录，perl 主程序在此。
C:\Perl\eg	例子。
C:\Perl\html	文档。
C:\Perl\lib	库路径。存放 perl 模块。
C:\Perl\site	库路径。存放 perl 模块。

2.4 使用 POD

学会 perl 编程之前先学会使用 perl 帮助——perldoc。perldoc 相当于 unix 下的 man。执行如下命令可以得到一个 Perl 文档的一个索引。

>*perldoc perl*

输入模块名称可以察看模块相关的 perl 文档：

>*perldoc DBI*

输入函数名称，前面加选项-f 可以察看该函数的帮助：

>*perldoc -f split*

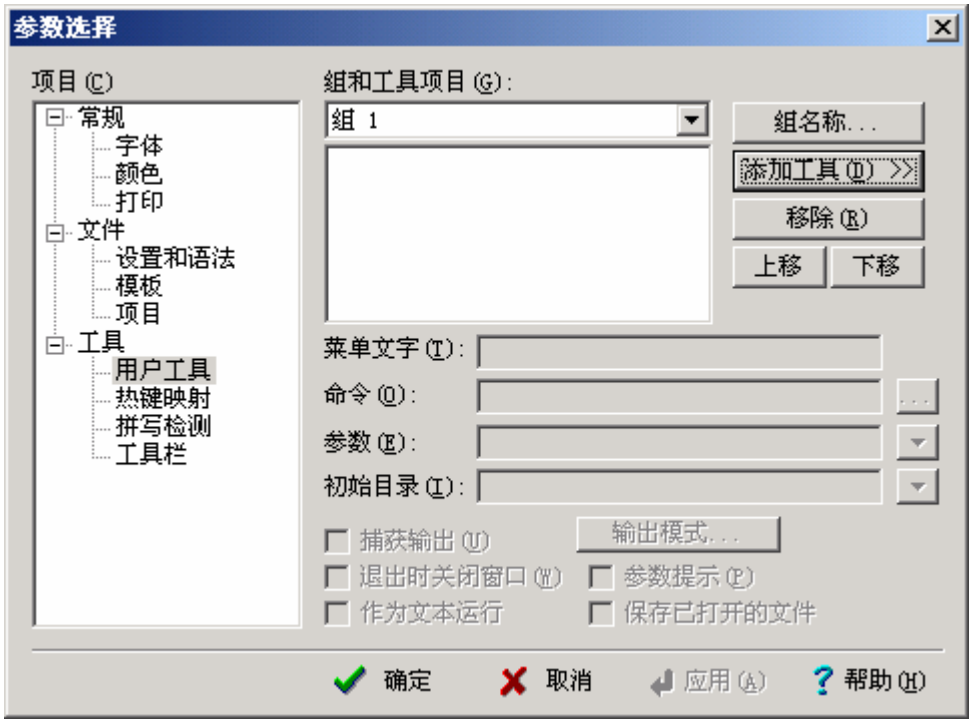
如果觉得阅读不方便，还可以 perl 文档由 POD 格式转换成 html 格式：

>pod2html --infile=c:\perl\lib\pod\perltoot.pod --outfile=c:\test1.html

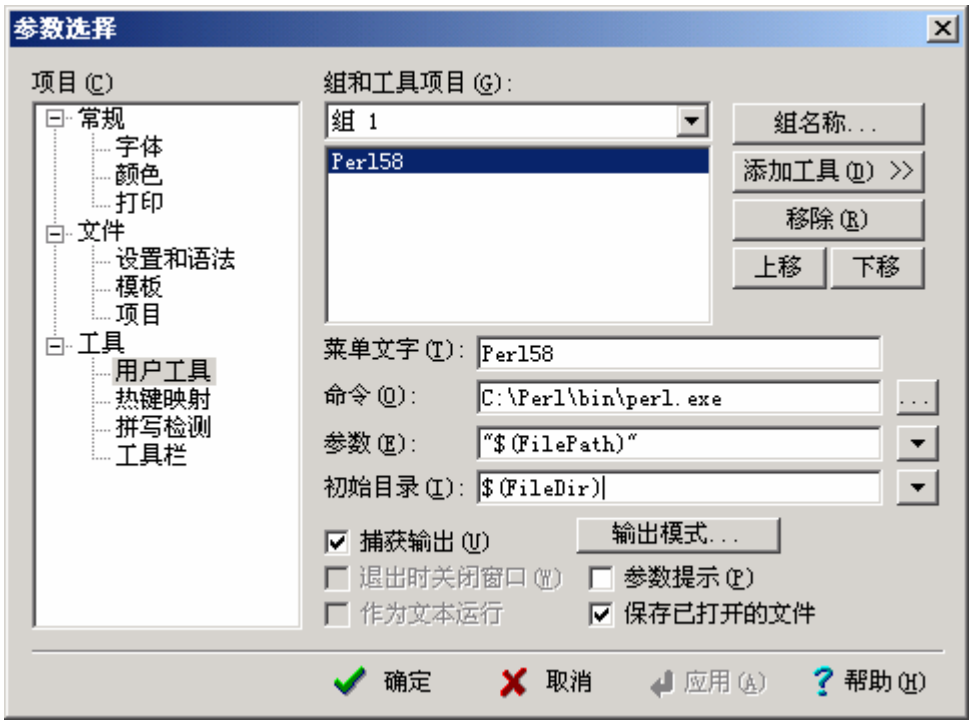
2.5 编辑工具

2.5.1 EditPlus

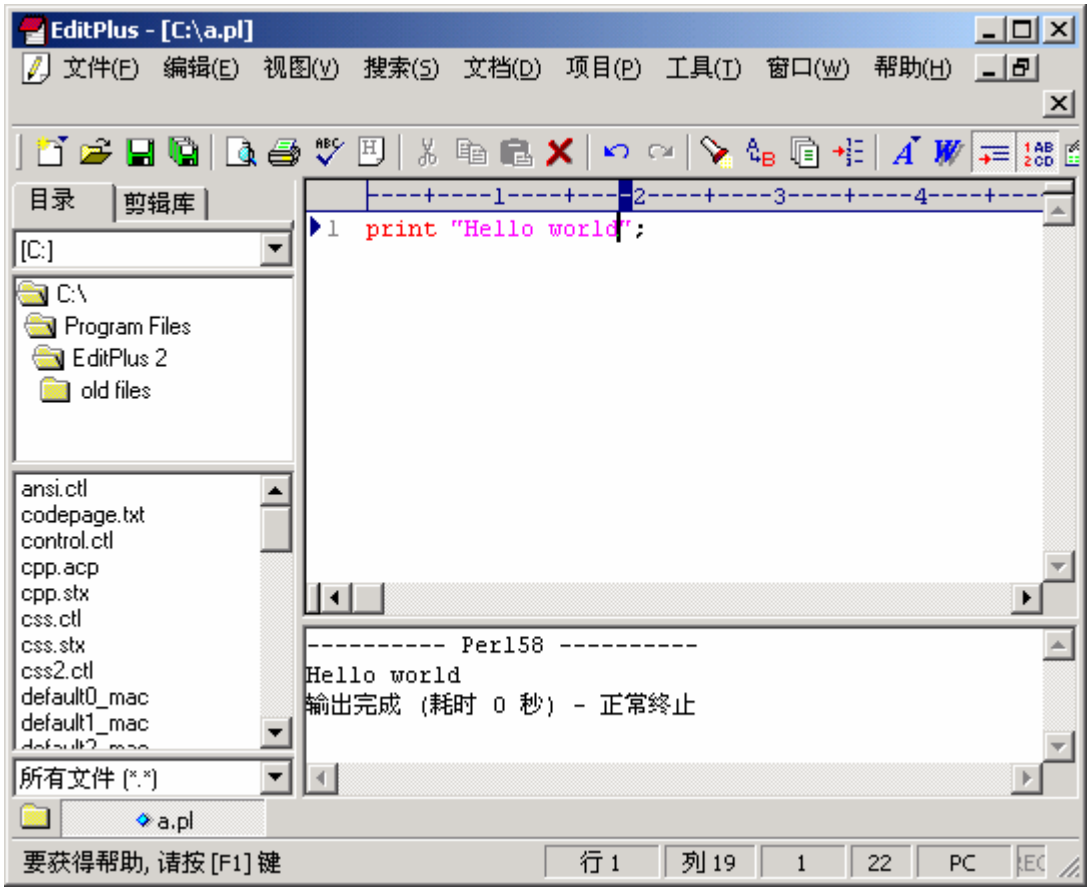
选择“工具”菜单下的“配置用户工具...”选项。选择用户工具然后组名称选择“组 1”(选别的也行,你要把工具栏设置一下就好了)，然后再点“添加工具”下的“应用程序”。



在菜单文字中输入名字，命令中选择可执行文件，参数设置成"$(FilePath)"。注意，这里的引号是必须的，否则就不能运行有空格的目录里的文件。然后把下面的捕捉输出选上。点确定就完成设置了！



打开 editplus，新建文件，输入一个简单的 perl 代码，然后存起来，然后在 editplus 的编辑状态下按组合键 Ctrl+1 就可以运行了。



2.5.2 UltraEdit

UltraEdit 非常有用的特性之一是可以编辑 FTP 服务器上的远程文件。可以通过菜单项的 File 下的 FTP 开始选择，也可以配置工具栏，通过工具栏上的两个按钮选择。配置工具栏的方法是：在工具栏上单击右键，选择 Costomize…。

下面讲最重要的工具配置。通常可以使用三种方式：运行，编译和调试。这三种 Perl 配置的命令行是：

名称	命令行参数
运行	c:\perl\bin\perl.exe -w "%f"
编译	c:\perl\bin\perl.exe -w -c "%f"
调试	c:\perl\bin\perl.exe -w -d "%f"

选择 Advanced> Tool Configuration...

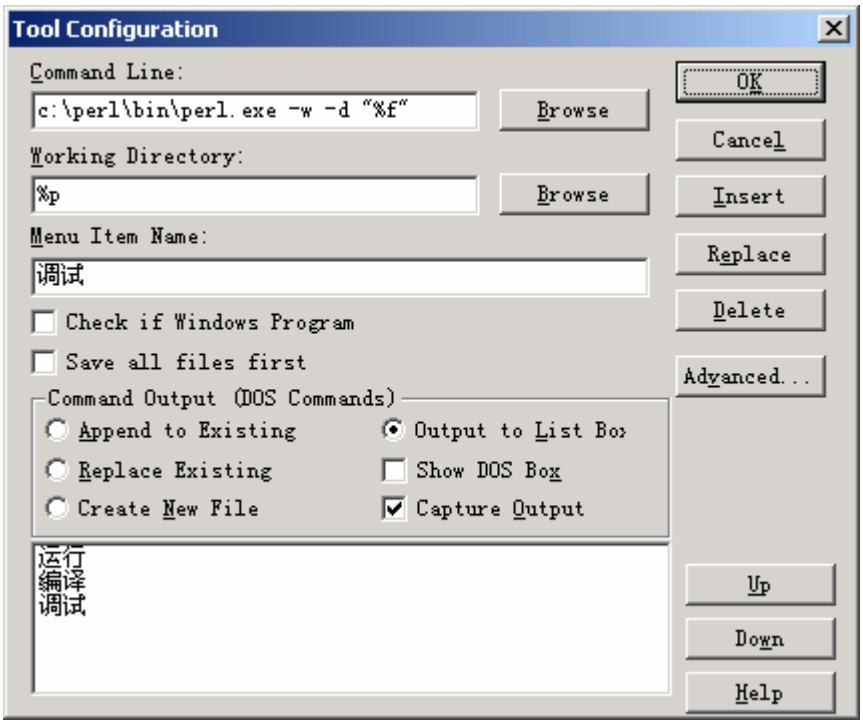
有三个地方要填：

Command Line : c:\perl\bin\perl.exe -w "%f"

Working Directory : %p

Menu Item Name : 运行

另外还要选取 Output to List Box 和 Capture Output 选项，设定好以后先按 Insert，依次插入上面三种方式，最后再按 Ok 按钮。



这些做完后在 Advanced 菜单下会多出三个运行 Ctrl+Shift+0，编译 Ctrl+Shift+1，调试 Ctrl+Shift+2 的菜单项(就是你刚刚自己取的)。

设定好这些以后其实就够了，你只要按下 Ctrl+Shift+0 就会自动让 perl 处理当前脚本，运行后所有的结果会输出在 UltraEdit 的 Output List Window。

2.5.3 SciTE

它是一个小型的文本编辑工具（http://www.scintilla.org/SciTE.html ），它的工作方式和大多数文本编辑器类似。它的特色是支持多种语言的语法着色（当然包括Perl）以及模块语句的折叠、展开等。

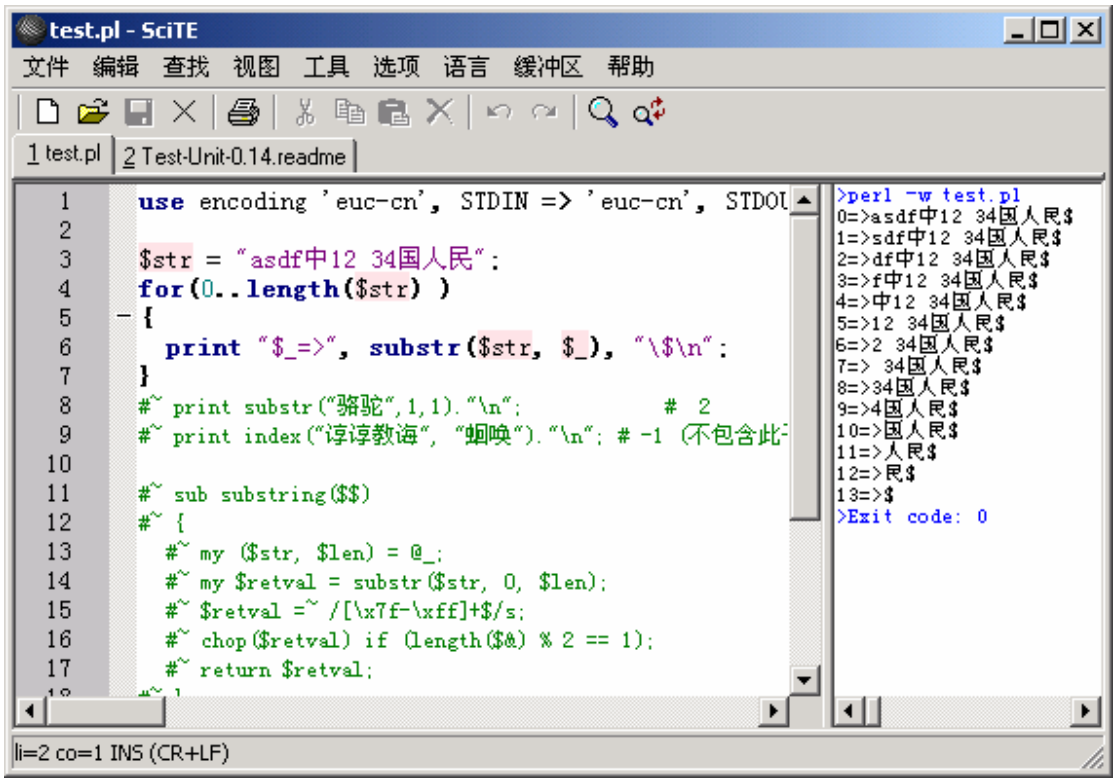
SciTE 初始的配置显示的功能很少，例如缺省情况下没有在文本左边显示行号，没有最近打开的列表选项，没有执行程序输入命令行的地方，也没有汉化。其实这些功能都是通过其配置文件实现的。

功能	配置方法
汉化	修改 locale.properties 文件。

功能	配置方法
编辑中文	修改 SciTEGlobal.properties 文件，设置 code.page=936，character.set=134，这样 SciTE 就可以把汉字看成一个整体而不是两个字节。
同时打开多个文件	SciTE 能够同时打开多个文件，但是只有一个文件是可见的。SciTE 的初始配置只允许编辑一个文件，但是可以通过改变 buffers 属性的值来打开多个文件。 SciTEGlobal.properties 中相关的属性有： buffers=10 设置同时打开的文件个数。 title.show.buffers=1 窗口的标题栏显示 buffer 的个数。 tabbar.visible=1 设置 tabbar.visible 成 1 使页面在 SciTE 启动时可见。 tabbar.hide.one=1 设置 tabbar.hide.one 成 1 时隐藏页面直到打开多个文件时。
在左边显示行号	修改 SciTEGlobal.properties 文件，设置 line.numbers=4。
最近使用文件列表	修改 SciTEGlobal.properties 文件，设置 save.recent=1。
显示工具栏	修改 SciTEGlobal.properties 文件，设置 toolbar.visible=1。
显示状态栏	修改 SciTEGlobal.properties 文件，设置 statusbar.visible=1。状态栏显示当前行号，列号，修改状态，回车定义。
自动完成	修改 SciTEGlobal.properties 文件，设置 autocompleteword.automatic=1。
修改制表符显示长度	修改 SciTEGlobal.properties 文件，设置 tabsize=4，indent.size=4。
缺省语言	当新建一个文件时选择缺省语言模式，例如设置 default.file.ext=.pl，选择 perl 语法风格。

功能	配置方法
传递参数给 perl 脚本	修改 perl.properties 文件，设置 command.go.$(file.patterns.perl)。 例如要传递参数： *>loader_qxxx_sk.pl -f SK010102.TXT -S dwdb -U dwdb -P dwdb* 可以修改： command.go.$(file.patterns.perl)=perl -w $(FileNameExt) -f SK010102.TXT -S dwdb -U dwdb -P dwdb 这样你就只需按 f5 快捷键即可执行脚本，而不用每次在命令行输入 长串的命令了。 在同时编辑多个 perl 脚本文件时，如果每一个脚本文件的运行参数都不同，每次修改 perl.properties 来改变运行参数显得很麻烦。这时可以通过 SciTE 的命令行设置该参数。例如： "C:\Program Files\editor\wscite\SciTE.exe" "-command.go.$(file.patterns.perl)=perl -w $(FileNameExt) c:/test.txt" c:/temp/test.pl

现在介绍 SciTE 中的特殊键。可以按下 Alt 键来选择文本的长方形区域。

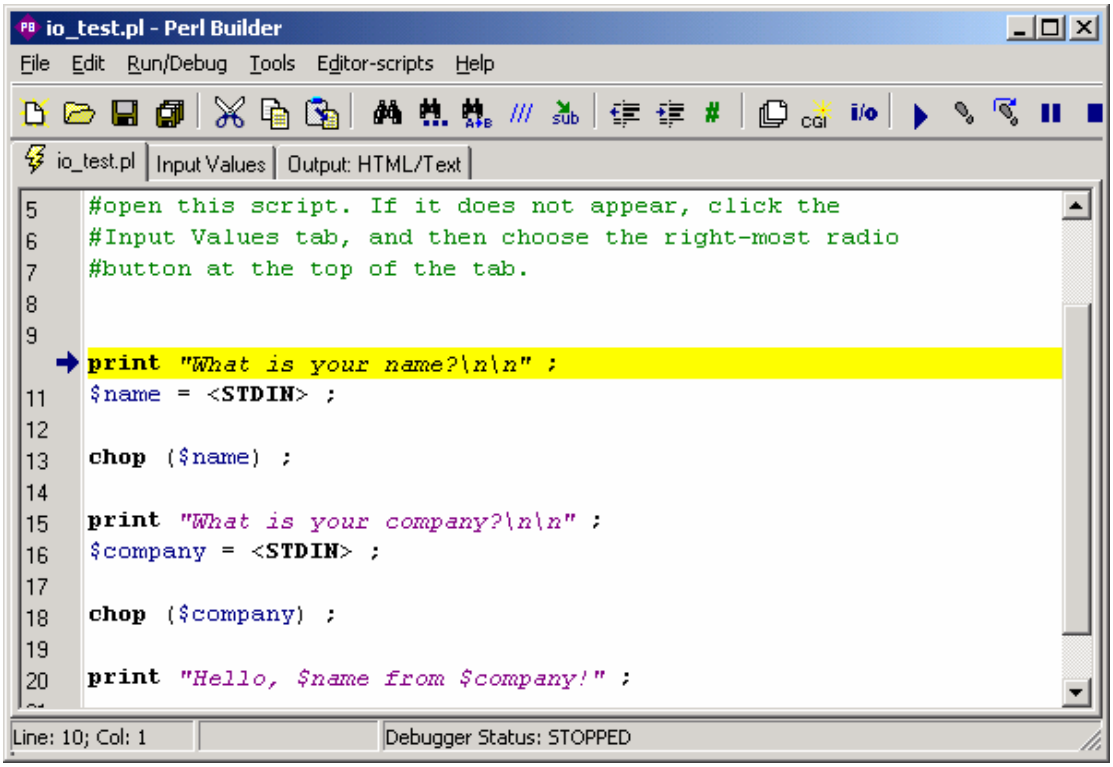


2.5.4 Open Perl IDE

一种集成开发工具（http://open-perl-ide.sourceforge.net/），它支持 Perl 语言的调试等。值得提的是，可以使用 Open Perl IDE 中的帮助功能，利用它可以制作 Active perl 的帮助索引。

2.5.5 Perl Builder

Perl Builder 是一个综合，完善的 Perl 语言编译器。他独有 CGI Wizard 功能可以帮助你简单直接的创建脚本。它拥有完善的编译和调试功能。它尤其适合做网页开发。



2.6 命名规范

常量	大写
类名	开头大写
内部特殊子例程	大写
内部特殊变量	大写

一般变量	小写
一般子例程	小写

2.7 变量

perl 中的变量不用声明，可以直接使用。但是声明变量有助于查错，是一种良好的编程习惯。

有三种变量，用不同的前缀区分：

前缀	变量
$	标量。标量又有数字，字符串等，但这些类型可以自动互相转换。
@	数组。
%	hash，又称关联数组。

2.7.1 数字

int -> string	printf ‘%4d’,2000
float -> int	int($float)
float -> string	printf ‘%e’, $floatnum

比较数字是否相等：

$val == 2

2.7.2 字符串

定义字符串：

q	字符串，一个 q()相当于一个单引号。quote 的缩写。
qq	被插入的字符串，一个 qq ()相当于一个双引号。quote quote 的缩写。
qr	正则表达式串。quote regex 的缩写。
qw	单词表。quote word 的缩写。

	例如： *use vars qw($opt_h $opt_v $opt_f $opt_S $opt_U $opt_P);*
qx	执行外部程序。quote execute 的缩写。

字符串相关函数：

函数	说明
print	输出字符串。 *print @list;*
chop	截断最后一个字符。 *chop $input_string;*
chomp	截断换行符，其返回值是截断的字符：换行符。 *chomp $lines;*
ord	*print ord('A') ; #返回65*
chr	*print chr(65); #返回 'A'*
length	返回字符串长度。 *$len = length($string);*
index	返回子串位置。 *index $string,"look for";*
rindex	逆向查找，返回子串位置。 *rindex $string,"look for";*
Substr(string, offset, length) substr(string, offset)	从一个字符串提取子串。偏移量从 0 开始。 *substr "1234567890",3,4; #返回子串 4567*
uc	大写。 *print uc('upper');*
lc	小写。 *print lc('LOWER');*
join(expr, list)	使用 expr 的值连接分离的字符串列表成为一个单独的字符串，返回新字符串。 *my $text = join('', @lines);*

模拟其它函数

原形	说明
ltrim	来源于 oracle，删除左边的空格。 *$var =~ s/^\s+//;　　#left trim*
rtrim	来源于 oracle，删除右边的空格。 *$var =~ s/\s+$//;　　#right trim*

比较字符串是否相等：

$val eq '2'

2.7.3 here 文档

here 文档定义一个字符串，它的结束符用紧接着<<的符号定义，这个符号可以用双引号或单引号括起来。同时它支持插值。例如下例：

```
my $Price = 'right';
#here documents
print <<EOF;
The price is $Price.
EOF
```

将打印出：

The price is right.

Here 文档仅仅是引号的一种可替代的形式。在你可以使用单引号或者双引号的地方就可以使用 here 文档。下面是一个生成 html 文档的例子。

```
use strict;

my $someURL = 'http://www.perl.com';

my $html =<<ENDHTML;
<HTML>
<BODY>
<P><A HREF="$someURL">Perl Homepage</A></P>
</BODY>
</HTML>
ENDHTML

open(DATAFILE, ">data.file") || die "could not open 'data.file'    $!";
print DATAFILE $html;      # print to file
close(DATAFILE);
```

print $html; *# print to STDOUT*

2.7.4 日期函数

函数	说明
time	返回 1970 年 1 月 1 日起经过的无跳跃秒数。可以用 gmtime 和 localtime 函数做进一步的处理。
times	返回一个四个元素的列表，给出当前进程及其子进程用户和系统时间，精确到秒。 *($user,$system,$cuser,$csystem) = times;* 在标量上下文中，times 返回$user。
localtime EXPR	把一个由 time 函数返回的时间转换成一个 9 个元素的列表，同时把该时间按照本地时区转换。 典型使用如下： *# 0 1 2 3 4 5 6 7 8* *($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);* 例如，要取得机器当前时间： *$time=localtime;* *print $time;*
gmtime EXPR	把一个由 time 函数返回的时间转换成一个 8 个元素的列表，同时把该时间转化成标准的格林威治时区时间。

2.7.5 数组

数组函数列表如下：

函数	说明
pop	从数组的末尾删除元素。 *@values = (1,2,8,14);* *$result = pop(@values); # 结果是 14* *print "@values \n"; # 1 2 8* *print pop(@values),"\n"; # 8* *print "@values\n"; # 1 2*

函数	说明
push	从数组的末尾添加元素。 *push(@values,10); # 1 2 10* *print "@values \n"; # 1 2 10* *push(@values, 11,8); # 1 2 10 11 8* *push(@values, @values); # 1 2 10 11 8 1 2 10 11 8*
shift	从数组的开头删除元素。 *shift(@values); # 2 10 11 8 1 2 10 11 8*
unshift	从数组的开头添加元素。 *unshift(@values, 1, 15) # 1 15 2 10 11 8 1 2 10 11 8*
reverse	把数组倒序。 *@back = reverse(@values);* *print "@back\n"; # 8 11 10 2 1 8 11 10 2 15 1* *print "@values\n"; # 1 15 2 10 11 8 1 2 10 11 8*
sort SUBNAME LIST sort BLOCK LIST sort LIST	*# 按字典方式排序* *@articles = sort @files;* *# 实现同样的目的，但是使用了显式的排序函数* *@articles = sort {$a cmp $b} @files;* *# 现在是大小写不敏感* *@articles = sort {uc($a) cmp uc($b)} @files;* *# 现在是倒排序* *@articles = sort {$b cmp $a} @files;* *# 按数字递增方式排序* *@articles = sort {$a <=> $b} @files;* *# 按数字递减方式排序* *@articles = sort {$b <=> $a} @files;* *# 现在使用内联函数按照值而不是关键字的方式排序哈西表 %age* *@eldest = sort { $age{$b} <=> $age{$a} } keys %age;*
map	*@numbers = (80,101,114,108);* *@characters = map (chr $_, @numbers); # 数值转换成字符*

函数	说明
split	语法： split /PATTERN/,EXPR,LIMIT split /PATTERN/,EXPR split /PATTERN/ split 分割一个字符串成为一个字符串的列表并返回该列表。分割符是匹配 PATTERN 的字符串，因此分割符长度可能大于一。 一个简单的例子如下： *$info = "Caine:Michael:Actor:14, Leafy Drive";* *@personal = split(/:/, $info);* *#@personal = ("Caine", "Michael", "Actor", "14, Leafy Drive");* 如果想把列表元素中的前后的空格去掉，可以把中间一行改成： *@personal = split(/\s*:\s*/, $info);* 缺省情况下，split 函数保留开头的空字符串，而删除结尾的空字符串。
grep	语法： grep BLOCK LIST grep EXPR,LIST 通常的调用方式是使用一个正则表达式，加上一个数组，但并不局限于此。 对每个传入数组的元素执行 BLOCK 或 EXPR ，然后返回由表达式为真的元素组成的数组。在标量上下文中，返回表达式为真的次数。 例如，要排除注释行： *@foo = grep(!/^#/, @bar); # weed out commentsor* 或等价的： *@foo = grep {!/^#/} @bar; # weed out comments* 注意：可以通过$_访问数组元素。当然也可以通过它修改数组元素，但是，修改同时也反映到原数组中去了。
scalar	返回数组大小。 *@names = (Jo, Pete, Bill, Bob, Zeke, Al);* *print scalar(@names); #6*

函数	说明
delete	清空该位置的元素，但不改变各元素的位置。 *my @array = (0,1,2,3,4,5,6);* *delete $array[3]; print join(':', @array),"\n"; #0:1:2::4:5:6*
exists	判断该元素是否已被删除。 *my @array = (0,1,2,3,4,5,6);* *delete $array[3];* *print join(':', @array),"\n" unless exists $array[3]; #0:1:2::4:5:6* 注意它不同于判断该元素是否 undef。 *my @array = (0,1,2,3,4,5,6);* *$array[3]= undef;* *print "exists\n" if exists $array[3]; #exists*
splice	清除该元素的位置。 *my @array = (0,1,2,3,4,5,6);* *splice(@array, 3, 1);* *print join(':', @array),"\n"; #0:1:2:4:5:6*
undef	让数组变成空白。
chop	每一个元素去掉最后一个字符。
chomp	去掉每一个元素尾部的换行符。

比较数组是否相等：

```
use Array::Compare;

my @arr1 = 0 .. 10;
my @arr2 = 0 .. 10;

my $comp1 = Array::Compare->new;

if ($comp1->compare(\@arr1, \@arr2)) {
    print "数组是相等的\n";
} else {
    print "数组是不同的\n";
}
```

2.7.6 哈希表

Perl 中的哈希表(Hash)用来存储关键字——值对。有的也把它叫做关联数组。哈希表相关的函数列表如下：

函数	说明
keys	返回一个键值的数组。 *$ranks{”UCLA”} = 1;* *$ranks{”OSU”} = 2;* *@teams = keys (%ranks); # @teams 是 (”UCLA”,”OSU”)* *# 也有可能是 (”OSU”,”UCLA”)*
values	返回一个值的数组。
each	返回一个“关键字——值”对。随后的调用返回剩下的“关键字——值”对，可用这个函数来遍历 hash。 *$ranks{”UCLA”} = 1;* *$ranks{”OSU”} = 2;* *while (($team, $rank) = each (%ranks)) {* *print (“Ranking for $team is $rank\n”);* *}*
delete	从 hash 删除一个“关键字——值”对，返回被删除的元素的值。 *$ranks{"UCLA"} = 1;* *$ranks{"OSU"} = 2;* *$x = delete $ranks{”UCLA”}; #现在%ranks 仅剩一个“关键字——值”对了。* *print ("$x \n"); # $x 是 1*
exists	判断该元素是否已被删除。 *$ranks{"UCLA"} = 1;* *$ranks{"OSU"} = 2;* *print ("存在\n") if exists $ranks{"UCLA"}; #打印出“存在”。*

2.8 引用

Perl 中有两种引用：硬引用和符号引用。因为符号引用被 use strict 禁止了，所以一般的引用都是指硬引用。

创建	使用反斜杠操作符可以创建引用。\相当于 C 语言中的&。 *$numberref = \42;* *$messageref = \”hello ref”;* [..]和{…}创建一个指向数组或 hash 的引用。它们创建一个自己内容的副本并返回指向它的一个引用，所以与 \ 操作符不一样。 *@array = [1,2,3,4];* *@copyhasref = {%hash};*

访问	$相当于 C 中的*，用于访问引用指向的值。

各种类型的引用：

引用	例子
标量引用	*$ra = \$a; # 指向标量的引用* *$$ra = 2; # 标量引用解引用* *$ra = \1.6; # 指向常量标量的引用*
数组引用	*$rl = \@l; # 指向已存在数组的引用* *$rl = [1,2,3]; # 指向匿名数组的引用* *push (@$rl, "a"); # 解引用* *print $rl->[3] # $rl 指向的数组的第 4 个元素*
哈希引用	*$rh = \%h; # 指向 hash 的引用* *$rh = {"laurel" => "hardy", "romeo" => "juliet"}; # 指向匿名 hash 的引用* *print keys (%$rh); # 解引用* *$x = $rh->{"laurel"}; # 取得单个元素的箭头符号* *@slice = @$rh{"laurel","romeo"}; # Hash 片断*
代码引用	$rs = \&foo; # reference to existing subroutine foo $rs = sub {print "foo"}; # reference to anonymous subroutine # (remember the semicolon at the end) &$rs(); # dereference: call the subroutine

通过 ref 函数返回引用的类型。例如：

$ref = \[1,2,3,4];
print “ref type “.ref($ref);

该函数返回的所有值如下表：

类型	含义
SCALAR	标量引用
ARRAY	数组引用
HASH	Hash 引用
CODE	例程引用
GLOB	Typeglob 引用

类型	含义
IO	文件句柄引用
REF	指向另一个引用
LVALUE	除了 SCALAR、ARRAY、HASH 之外的可分配的值。

2.9 多维数组

perl5 通过其强大的引用功能解决多维数组的问题。

简单的引用：

$abc=\$dbc

实际上$$abc,$dbc 等价。这很像在 c 中通过引用传值。

但事实上 perl5 的引用并不是一个简单的指针，它有些像 unix 下面的连接文件，只有所有的引用都被去掉，perl5 才释放这个变量占用的空间。

引用的最大特色是多重引用。数组和哈希表的元素也可以是引用。当然对于一般人来说，数组的数组，也就是多维数组,更有实际价值。

功能	举例
创建数组	*$abc=[["00","01"],["10","11"]];*
使用数组	*$var=$abc->[1][1];*

2.10 常量

使用 constant 编译指示允许定义常量。

常量类型	说明
标量常量	*use constant PI => 4 * atan2(1, 1);*

常量类型	说明
列表常量	声明： *use constant WEEKDAYS => qw(* *Sunday Monday Tuesday Wednesday Thursday Friday Saturday* *);* 使用必须加括号： *print "Today is ", (WEEKDAYS)[1], ".\n";*
哈希常量	*use constant WEEKABBR =>{* *Monday => 'Mon',* *Tuesday => 'Tue',* *Wednesday => 'Wed',* *Thu => 'Thursday',* *Fri => 'Friday');* 使用举例如下： *%abbr = WEEKABBR;* *$day = 'Wednesday';* *print "The abbrevaiation for $day is ", $abbr{$day};*

2.11 操作符

Perl 中的操作符细分起来有 14 种，分别是：赋值操作符、算术操作符、字符操作符、比较操作符、位操作符、逻辑操作符、位操作符、组合赋值操作符、递增和递减操作符、正则表达式操作符、逗号操作符和关系操作符、引用操作符和访问引用操作符、箭头操作符、范围操作符、三元操作符、文件操作符、命令操作符。其中正则表达式操作符、文件操作符、命令操作符将在后面专门介绍。

在后面的介绍中我们会看到，同样的操作符在不同的上下文中会有不同的运算方法。例如，逗号操作符在列表上下文和标量上下文中的表现就不一样。

2.11.1 赋值操作符

它把右边的表达式的值赋给它左边的变量。例如：

$x = 'value';

在 Perl 中，lvalue 表示赋值运算符左边的实体。lvalue 必须是变量，可以给它分配值。例如不能向字符串赋值，如"constant"=132 这个语句就是错误的！因为"constant"常量字符串不能作为一个 lvalue。

赋值操作符还可以用其他的操作符修饰，如算术操作符等，这就是组合操作符，我们将在后面专门介绍。

2.11.2 算术操作符

操作符	说明
+	加。
-	减。
*	乘。
/	除。
**	乘幂。 例如：3 ** 3 得 27。2 ** 0.5 得 1.414。但是当底数是负数时，指数不可以是小数。
%	取余。例如：3 % 2 得 1。
-	单目负。

需要指出的是 Perl 自身提供了常用的数学运算符，但是没有提供四舍五入的函数，Math-Round 模块提供了一个四舍五入的实现。

2.11.3 字符操作符

Perl 的字符串操作符包括连接操作符” .”和复制操作符”x”：

连接操作符相当于字符串的加法：

$example = ‘Hello ’.’World’;

但是如果实际使用

$example = ‘Hello ’+’World’;

返回的结果将是 0，因为执行的是整型运算，相加的字符串被转换成了整数 0 然后相加。

复制操作符”x”左边是一个字符串或一个列表，右边代表左边的元素复制的次数。例如：

$example = ”\t” x 8;
@array = (1,2,3,4,5) x 2; # @array =(1,2,3,4,5,1,2,3,4,5);

2.11.4 比较操作符

需要注意的是，数字和字符串有不同的比较操作符。

数字	字符串	说明
<	lt	小于
>	gt	大于
==	eq	等于
<=	le	小于等于
>=	ge	大于等于
!=	ne	不等于
<=>	cmp	比较两个值。当两个值相等时返回 0，当第一个值大时返回 1，第二个值大时返回 –1。

2.11.5 逻辑操作符

操作符	命名	说明
!	not	逻辑非
||	or	逻辑或
&&	and	逻辑与
	xor	逻辑异或

操作符和其命名在运算时是完全等价的，但是有不同的优先级。操作符有更高的优先级。例如 && 比 and 有更高的优先级。

2.11.6 位操作符

操作符	说明
&	位与。例如把一个字符转换成大写：print 'a'&'_';　#得到 A

|	位或。例如设置打开文件的模式：O_CREAT|O_TRUNC|O_RDWR。把一个字符转换成小写：print 'A'| '; #得到 a
~	位非。
^	位异或。
<<	左移。
>>	右移。

2.11.7 组合赋值操作符

操作符	操作符类型	说明
+=	算术操作符	相加并赋值。 $x += $y; 等价于 $x = $x + $y;
-=	算术操作符	相减并赋值。$x -= $y; 等价于 $x = $x - $y;
*=	算术操作符	相乘并赋值。$x *= $y; 等价于 $x = $x * $y;
/=	算术操作符	相除并赋值。$x /= $y; 等价于 $x = $x / $y;
%=	算术操作符	取余并赋值。$x %= $y; 等价于 $x = $x % $y;
**=	算术操作符	乘幂并赋值。$x **= $y; 等价于 $x = $x ** $y;
x=	字符串操作符	重复并赋值。$x x= 3; 等价于 $x = $x x 3;
.=	字符串操作符	连接并赋值。$x .= $y; 等价于 $x = $x . $y;
<<=	移位操作符	左移并赋值。$x <<= $y; 等价于 $x = $x << $y;
>>=	移位操作符	右移并赋值。$x <<= $y; 等价于 $x = $x << $y;
&&	逻辑操作符	逻辑与并赋值。$x &&= $y; 等价于 $x = $x && $y;
||=	逻辑操作符	逻辑或并赋值。$x ||= $y; 等价于 $x = $x || $y;
|=	位操作符	位或并赋值。$x |= $y; 等价于 $x = $x | $y;
&=	位操作符	位与并赋值。$x &= $y; 等价于 $x = $x & $y;
^=	位操作符	位异或并赋值。$x ^= $y; 等价于 $x = $x ^ $y;

2.11.8 递增和递减操作符

递增操作符 ++ 和递减操作符 -- 都是一元操作符，分别对一个标量变量（操作数）执行加一或减一操作。

这两个操作符既可以放在操作数的前面，也可以放在操作数的后面。放在前面时，执行的是预加一或预减一的操作，放在后面时，执行的是后加一或后减一的操作。

```
$var = 1;
print ++$var;       #打印出 2
print $var++;       #打印出 2
print $var;         #打印出 3
```

执行递增或递减操作的标量变量可以是一个整形变量,浮点型变量或者是一个字符串变量。

字符串变量增加时，变化的是结尾字符，而且能发生进位。结尾字符是按照字符排序的顺序进行的。分别是 a-z，A-Z，0-9。'z'的下一位是'a'、'Z'的下一位是'A'、'9'的下一位是'0'。例如：

```
$stringvar = "abc";
$stringvar++;
print $stringvar;    #打印出"abd"

$stringvar = "abz";
$stringvar++;
print $stringvar;    #打印出"aca"

$stringvar = "AGZZZ";
$stringvar++;
print $stringvar;    #打印出"AHAAA"
```

注意，只有完全由字母和数字组成的字符串才可以递增。

当使用--时，PERL 会先将字符串转换为数字再进行自减。

```
$stringvar = "abc";
$stringvar--;
print $stringvar;    #打印出-1
```

2.11.9 逗号和关系操作符

从左到右的计算逗号操作符，在列表上下文中，Perl 返回从左至右的列表。例如：

```
@color=($r,$g,$b);
```

在标量上下文中，返回最右边的操作数：

```
$x=(20,40,10);
print $x;     #打印 10
```

关系操作符 => 是一个定义哈西变量的关键字-值对的逗号，它还允许把裸词用于关键字：

```
%x = ('Red'=>20,'Green'=>40,'Blue'=>10);
%x = (Red=>20,Green=> 40,Blue=>10);
```

如下的例子可以看出，关系操作符和逗号操作符是等价的：

```
%ages = ('Tom',20,'Jerry',40);
foreach $key (keys %ages){print "$key is $ages{$key} years old\n";}
%ages = ('Tom'=>'20','Jerry'=>'40');
foreach $key (keys %ages){print "$key is $ages{$key} years old\n";}
```

2.11.10 引用操作符

引用操作符即前面在引用一节中介绍的反斜杠操作符。它是一元操作符，它创建并返回一个指向它随后的各种类型的值、变量或子例程的引用。由于它并不会拷贝原值，因此改变引用指向的值会改变原始值。

```
$message = "hello ref";
$messageref = \$message;
$$messageref = "hello world";
print $message;          #打印 hello world
```

2.11.11 箭头操作符

箭头操作符有两种用法：第一种用法是访问引用中的数据元素。例如：

```
$array_element = $arrayref -> [1];
$hash_element = $hashref -> {$key};
```

这种方法的一个特例是使用多维数组或者 hash 的 hash，其中隐含使用了箭头操作符，以下两种方法是等价的：

```
$element = $pixel[$x][$y][$z];
```

和：

```
$element = $pixel[$x] -> [$y] -> [$z];
```

第二种用法是在对象中使用，通过它来调用一个对象（或实例）的方法：

```
$object ->method(@arguments);
```

2.11.12 范围操作符

范围操作符可以在列表上下文或在标量上下文中使用。它在这两种方式下，工作模式不一样。

在列表上下文中，范围操作符返回一个列表，该列表的第一个元素是左边的操作数，以后的元素是依次递增这个操作数，直到右边的元素为止。例如： 1..5 返回列表： (1,2,3,4,5)。递增的方法是以递增操作符的方式实现的，因此：'a'..'e'返回列表：('a','b','c','d','e')。

如果左边等于右边，则返回一个单元素的列表；如果左边的操作数更大，则返回空列表。

在标量上下文中使用难以理解，这里不做介绍。

2.11.13 三元操作符

三元操作符"? :"可以看成是一个 if 语句。例如下面的操作：

<condition>?<true-result>:<false-result>;

其等价的表达形式是：

if <condition> then return <true-result> else return <false-result>;

例如，下面的语句通过 ?: 操作符，取得最大值：

$max_value=($a > $b ? $a : $b);

2.11.14 操作符的连接性

连接性	操作符（按优先级排列）
左	项、列表操作符
左	->
无连接性	++
右	** (幂)
右	! ~ \ 和一元+ 、一元- (逻辑非，位非，引用，一元加，一元减)

连接性	操作符（按优先级排列）
左	=~ !~ (匹配，不匹配)
左	* / % x　(乘，除以，模，字符串复制)
左	+ - . (加，减，字符串连接)
左	<< >> (左位移，右位移)
无连接性	命名一元操作符，文件测试操作
无连接性	< > <= >= lt gt le ge (小于，大于，不大于，不小于,以及它们的字符串等价形式)
无连接性	== != <=> eq ne cmp (等于，不等于，符号比较，以及它们的字符串等价形式)
左	&（位与）
左	| ^ (位或，位异或)
左	&& (逻辑与)
左	|| (逻辑或)
无连接性	.. ... (范围)
右	?: (三元条件操作)
右	= += -= *=　等 (赋值操作)
左	, => (逗号，箭头)
无连接性	List operators (向右)
右	not (逻辑反)
左	and (逻辑与)
左	or xor (逻辑或,异或)

2.12 控制流

Perl 中有四种最常用的控制流：wile\for\foreach\if…else。此外还有和 wile 等价的控制流 unless\until 和局部跳转语句 next\last\redo。最后再加上多分支条件语句 switch。另外，wile\unless\until\if 控制流还有单行简写形式。

2.12.1 if, else, elsif

它的基本语法结构如下：

if (boolean_expression1)

{sequence_of_statements1;}

[elsif(boolean_expression2)

{ sequence_of_statements2;}]

…

[else

{ sequence_of_statementsn;}]

这里 boolean_expressionx 是任意布尔表达式。除了 if 外，elsif 和 else 都是可选结构。而且 elsif 可以任意次重复。例如下面一个判断成绩的例子：

```
print "Enter your grade ";
$grade = <STDIN>;
if ( $grade < 70 ) # < is the less than operator
{
print "You get a D\n";
}
elsif ( $grade < 80 )
{
print "You get a C\n";
}
elsif ( $grade < 90)
{
print "You get a B\n";
}
else
{
```

```
print "You get an A\n";
}
print "end of construct\n";
```

需要指出的是 elsif 不是 elseif，其中少了一个 e，这个关键字来源于 ada 语言。

if 控制流的单行简写形式是：

```
statement if (boolean_expression1);
```

例如：

```
print 'OK' if $valid;
```

2.12.2 switch

条件多分支判断语句 switch 是用 Switch 模块实现的。Switch 模块在 Perl 5.8.0 版本才引入。

它引入两个新的控制语句：switch 和 case。switch 语句接受一个任何类型的标量参数，作为条件。一个 case 语句接受一个标量参数，作为判断。 如果条件和判断匹配上，则执行 case 随后的语句。如果一个 case 匹配上以后，整个 switch 判断将会结束。也就是说，不会再执行随后的 case 判断。如下是一个例子：

```
use Switch;

switch ($val) {
        case 1          { print "number 1" }
        case "a"  { print "string a" }
        case [1..10,42]{ print "number in list" }
        case (@array) { print "number in list" }
        case /\w+/      { print "pattern" }
        case qr/\w+/   { print "pattern" }
        case (%hash)   { print "entry in hash" }
        case (\%hash) { print "entry in hash" }
        case (\&sub)   { print "arg to subroutine" }
        else        { print "previous case not true" }
    }
```

2.12.3 unless

它的基本语法结构很简单：

```
unless (boolean_expression1)
```

{sequence_of_statements1;}

例子如下：

```
unless ( $a == $b ) # ==  是等于操作符
{
print "this stuff executes when\n";
print '$a is not equal to $b\n';
}
```

unless 控制流的单行简写形式是：

statement unless (boolean_expression1);

例如：

```
print ‘ERROR’ unless $valid;
```

2.12.4 while

while 循环的基本语法结构如下：

while (boolean_expression)

{sequence_of_statements;}

当 Perl 遇到 while 语句时，它就计算该条件。如果条件计算的结果是真，代码块就运行。当运行到代码块的结尾时，表达式被重新计算，如果结果仍然是真，代码块重复执行，如下例所示：

```
$i = 0;
while ( $i <= 10) #  当 $i <= 10
{
print "$i\n";
$i = $i + 1;
}
print "code beneath the loop\n";
#  另外一个例子
while (($key,$value) = each(%hash))
{
    print “$key => $value \n”;
}
```

2.12.5 until

until 循环的基本语法结构如下：

until (boolean_expression)

{sequence_of_statements;}

until 与 while 不同的是：当条件为真时退出。如下例打印 1 到 10 所示：

```
$i = 1;
until ( $i > 10)
{
print "$i\n"; #打印 1 到 10
$i = $i + 1; #
}
```

2.12.6 for

for 语句是能够实现自动增加的循环结构中。它的语法结构如下：

for (initialization, boolean_expression, increment)

{sequence_of_statements;}

当 Perl 遇到一个 for 循环时，执行顺序如下：

- 初始化表达式被计算。
- 测试表达式被计算。如果它的计算结果的真，代码块就运行。
- 当该代码块执行结束后，便执行递增操作，并再次计算测试表达式。如果该测试表达式的计算结果仍然是真，那么代码块再次运行。这个进程将继续下去，直到测试表达式的计算结果变为假为止。

一个例子如下：

```
for ( $i = 1; $i <= 10; $i = $i + 1)
{
$s = $s + $i;
} # 计算前 10 个整数的和

//另外一个例子
for my $i ( 0 .. $#values) {
```

```
    next if $noquote->{$types->{$fields->[$i]}};
}
```

2.12.7 foreach

foreach 语句可以看成 for 循环结构的一个特例版本。它的语法结构如下：

foreach (list)

{sequence_of_statements;}

foreach 语句中，括号中的表达式用于产生一个列表。然后列表中的每个元素依次赋给循环变量，并对每个元素执行一次循环语句。注意循环变量是元素本身的一个引用，而不是元素的一个拷贝。因此，修改循环变量将修改原来的数组。

示例如下：

```
@list = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
foreach $item (@list)
{
$sum = $sum + $item;
}
```

在使用中往往用它遍历哈西变量：

```
//另外一个例子
$ages{'Tom'}=23;
foreach $key (keys %ages)
{
print "$key is $ages{$key} years old\n";
}
```

2.12.8 last

last 控制流允许你跳到当前循环的末尾，也就是跳出循环。它类似 c 语言中的 break。

例如下例把 1 到 4 之间的数值显示出来：

```
for($i=1;$i<=10;$i++)
{
last if ($i==5); #如果$i 等于5 的话就退出for 循环
print"$i\n";
}
```

2.12.9 next

next 控制流允许你跳到当前循环的末尾，开始下一次循环。它类似 c 语言中的 continue。

例如下面的例子把 1 以 10 之间的奇数显示出来。

```
for($i=0;$i<=10;$i++)
{
    #如果是 2 的倍数的话,就进入下一次循环
    next unless ($i%2);
    print"$i 是一个奇数!\n";
}
```

Perl 的所有的循环都可以附加一个 continue 块。这个 continue 块在 next 控制流跳到当前循环的末尾之前执行，例如：

```
$i = 0;
while ($i <=10)
{
    next unless ($i %2);
    print $i,"是一个奇数!\n";
}
continue{
    $i ++;
}
```

刚才这个 while 循环和 for 循环的例子是等价的。

2.12.10 redo

它类似上面的 next 语句，但是不会判断循环的退出条件。

```
while (boolean_expression)

{

    # redo  跳到这里

    sequence_of_statements1;

    redo;

    sequence_of_statements2;
```

```
}
```

例如：

```
$i = -1;
while ($i <=10)
{
    $i++;
    redo unless ($i %2);
    print $i,"是一个奇数!\n";
}
```

会打印出：

1 是一个奇数!

3 是一个奇数!

5 是一个奇数!

7 是一个奇数!

9 是一个奇数!

11 是一个奇数!

2.13 文件与目录

Perl 程序是通过文件句柄来进行 I/O 操作的。特别地，Perl 提供了缺省的文件句柄 STDIN（代表标准输入）、STDOUT（代表标准输出）和 STDERR（代表标准错误输出）。下面分别介绍读写文件，以及删除和重命名文件，创建和删除目录。

文件和目录的基本操作如下：

操作	例子
写文件	*$append = 0;* *if ($append)* *{* *open(MYOUTFILE, ">filename.out"); #覆盖写* *}* *else* *{* *open(MYOUTFILE, ">>filename.out"); #追加写* *}* *print MYOUTFILE "Timestamp: "; #写文本，没有换行。* *print MYOUTFILE timestamp(); #把函数返回值写到文本*

	print MYOUTFILE "\n"; #写换行符 *#*** 打印自由文本，需要后面的分号**** *print MYOUTFILE <<"MyLabel";* *Steve was here* *and now is gone* *but left his name* *to carry on.* *MyLabel* *#关闭文件* *close(MYOUTFILE);*
读文件（单行）	*open(MYINPUTFILE, "<filename.out");* *while(<MYINPUTFILE>)* *{* *# 最好把$_存下来，因为以后的操作可能会改变该值。* *my($line) = $_;* *# 最好把结尾的换行符去掉。* *chomp($line);* *# 把该行的小写转换成大写。* *$line =~ tr/[a-z]/[A-Z]/;* *# 打印该行，并增加一个换行符。* *print "$line\n";* *}*
读文件（多行）	*open(MYINPUTFILE, "<filename.out"); # 以只读的方式打开文件。* *my(@lines) = <MYINPUTFILE>; #把文件的内容一次性读到数组。* *@lines = sort(@lines); # 把数组排序。* *foreach my $line (@lines)* *{* *print "$line"; # 按顺序打印。* *}* *close(MYINPUTFILE);*
删除文件	*unlink ('c:/test.txt'); # 删除文件 test.txt* *unlink ("cowbird","starling"); # 一石二鸟* *unlink <*.o>; #如同 shell 命令 "rm *.o"*
重命名	*rename("fred","barney") || die "Can't rename fred to barney: $!";*
创建目录	*mkdir("gravelpit",0777) || die "cannot mkdir gravelpit: $!";*
删除目录	*rmdir("gravelpit") || die "cannot rmdir gravelpit: $!";*

2.14 例程

例程（Subroutine）又称函数，是结构化程序设计的基础。它接受多个输入参数，返回一个输出参数。

定义语法：

```
sub Subroutine_name[()]

{

    sequence_of_statements;

}
```

举例如下：

```
sub GetCurrentPath()
{
    $fs = Win32::OLE->new("Scripting.FileSystemObject");
    $folder = $fs->GetFolder(".");
    $ls_current_path=$folder->path;
    return    $ls_current_path;
} # GetCurrentPath
```

调用语法有如下三种：

subname;	对于 perl 5.6 以上，推荐用此种调用方式。
do subname;	
&subname;	perl 5.6 以上，不推荐用此种调用方式。

举例如下：

```
$current_path = GetCurrentPath;
print "$current_path\n";
```

例程可以预定义：

```
sub GetCurrentPath();
```

传递参数给例程是通过特殊变量@_完成的。例如：

```
sub Logger($)
{
  ($line) = @_;
  print "$line\n";
}
```

如下是一个通过 shift 函数访问输入参数的例子：

```
sub Logger($)
```

```
{
   $line = shift;
   print "$line\n";
}
```

2.15 执行命令

执行操作系统命令的方法有如下几种：

执行命令方法	例子
system	*my $ret = system($cmd);* *if ($ret!=0)* *{* *die "error during execute $cmd\n extend_error=$^E \nerrorno=$!\n ";* *}*
反小点引号	返回值是该命令的输出。也可以用反小点引号操作符 qx。 *my $this_output = `( $cmd ) 1>$log_file`;* *$rc = ($? >> 8);* *print ($this_output);* *unless ($rc == 0)* *{* *die ("Problem running command (Returned $rc): \n$cmd\n$!\n");* *}*
WScript.Shell	*$WshShell = Win32::OLE->new("WScript.Shell");* *$WshShell->Run("$ls_cmd");*
Win32::Spawn	*my $command = 'c:\windows\notepad.exe';* *my $args = 'notepad.exe c:\my documents\test.txt';* *my $pid = 0;* *Win32::Spawn($cmd, $args, $pid) or die Win32::FormatMessage(Win32::GetLastError());*
Win32::Process	*Win32::Process::Create($process, $program, $comd, $handle, $option, $dir);*

其中通过调用 system 函数和反小点引号方法比较常用，WScript.Shell 和 Win32::Spawn 以及 Win32::Process 的方法是 windows 操作系统特有的。

另外还可以通过 Net::Telnet 模块，远程执行命令：

```
use Net::Telnet ();
$t = new Net::Telnet (Timeout => 10, Prompt => '/bash\$ $/');
$t->open("sparky");
$t->login($username, $passwd);
@lines = $t->cmd("who");
print @lines;
```

2.16 正则表达式

正则表达式是一个描述模式（pattern）的字符串。在 Perl 中，正则表达式用来查找/替换字符串，提取字符串中想要的部分等。

2.16.1 基本类型

最简单的正则表达式是匹配一个单词，例如：

```
"Hello World" =~ /World/;  # 匹配上了
```

这里"Hello World"是一个字符串。World 是正则表达式而 // 包围的 /World/ 告诉 perl 为了匹配搜索字符串。操作符 =~ 联接字符串和正则表达式匹配。如果正则表达式匹配成功，整个表达式返回 true ，否则返回 false。在这个例子中， World 匹配上了字符串 "Hello World"，因此表达式是真。可以像下例把这样的表达式用在条件判断中：

```
if ("Hello World" =~ /World/) {
    print "It matches\n";
}
else
{
    print "It doesn't match\n";
}
```

正则表达式操作符有两种：=~和!~。!~相当于=~取反。

正则表达式一般由前缀，分隔符，模式，和修饰符等组成。按前缀分，正则表达式有如下几种类型：

匹配方式	说明
m/pattern/gimosxe	匹配字符串。
?pattern?	仅匹配一次，其他与 m 相同。
s/pattern/replacement/egimosx	置换。
tr/pattern1/pattern2/cds	单个字符匹配。
y/pattern1/pattern2/cds	单个字符匹配。

如下分别描述：

m/pattern/gimosxe

修饰符	说明
g	全局匹配，即发现所有的出现位置。
i	不区分大小写的匹配。
m	把匹配字符串看成多行。让^匹配任何逻辑行的开始（包括字符串的开始和任何新行符后的开始），让$匹配任何逻辑行的结束(任何新行符后的开始和字符串的结束)。 使用/m 对“圆点”操作符没有影响。因此，当单独使用/m 时，“圆点”匹配除换行符之外的任何字符。 例如，从一个配置文件读取配置选项到一个哈西表： *my $text = read_file($file) ;* *my %config = $test =~ /^(\w+)=(.+)$/mg ;*
o	仅编译模式一次。这样能节省 Perl 编译长的正则表达式的时间。
s	把匹配字符串看成单行。强迫^和$不特殊看待新行符号，同时让“圆点”匹配任何字符。没有该修饰符时，“圆点”匹配除换行符之外的任何字符。 当修饰符 s 和 m 一起使用时，效果和单独使用 m 一样，除了此时“圆点”匹配任何字符。
x	使用扩展的正则表达式。

qr/pattern/imosx

创建预编译的正则表达式。

s/pattern/replacement/egimosx

例如：

$string = "abc123def";
$string =~ s/123/456/; # 现在 $string = "abc456def";

修饰符	说明

修饰符	说明
e	把右边看成一个表达式。 例如，要匹配 123 和 4 中间的“a b f 7”： *$a="zzzzz123a4xxx123b4www123f4ssss12374";* *$a=~s/123(.*?)4/&getdata($1)/eg;* *sub getdata(){* *my ($a)=@_;* *$data.=$a." ";* *return $data;* *}* *print $data;*
g	全局替换，也就是替换所有的。 例如： *$text =~ s/：/:/g;*
i	执行大小写无关的匹配。
m	把字符串看成多行。
o	仅编译模式一次。
s	把字符串看成单行。
x	使用扩展的正则表达式。

tr/pattern1/pattern2/cds

y/pattern1/pattern2/cds

修饰符	说明
c	补足模式 1。
d	删除发现的但不替换字符。
s	压缩重复替换的字符。

2.16.2 正则表达式模式

元字符表

元字符	说明
\	转义符，转义随后的一个字符。 比如\$不再代表结束符，而代表'$'。 例如模式： /a\.b/ 匹配： "proga.bat" 不匹配： "a..b"
.	匹配任何单个字符除了一个新行 (除非使用了/s)。 例如模式： /f./ 匹配： "this is fun" 不匹配： "nothing beyond the f"
^	匹配字符串的开始(或者是行的开始，如果使用/m)。 例如： *$var =~ s/^\s+//;　　#左边的 trim*
$	匹配字符串的结束 (或者是行的结束，如果使用/m)。 例如： *$var =~ s/\s+$//;　　#右边的 trim*

元字符	说明

*	匹配前面的元素 0 或多次。
	
	缺省情况*是贪婪的，要让它不贪婪，需要在后面加上?。
	
	$a="(12)34(56)78(90)abc";
	$a=~s/\(.\)//g;*
	print"$a\n"; #abc
	$a="(12)34(56)78(90)abc";
	$a=~s/\(.?\)//g;*
	print"$a\n"; #3478abc

+	匹配前面的元素 1 或多次。 例如： /\d+/ 匹配一个无符号整数。 和*一样，缺省情况+是贪婪的，要让它不贪婪，需要在后面加上?。
?	匹配前面的元素 0 或 1 次。特殊的，?可以用来修饰*。
{...}	声明前面元素出现的次数范围。 例如： {num} 匹配前面的字符 num 次。 {min, max} 匹配前面的字符至少 min 次，但不超过 max 次。 {12,} 匹配前面的字符 12 次或更多次。
[...]	匹配括号中的任何一个字符。 例如： [0-9]匹配所有数字。 [^0-9] 匹配所有非数字内容。 下面这段代码去掉尾部的中文空格： *$var ='test　';* *$var =~ s/[\xA1][\xA1]$//;* *print "$var";*
(...)	正则表达式组。

元字符	说明
|	匹配前面或者后面的表达式。

转义符

转义符	说明
\t	tab
\n	新行
\r	回车
\f	进纸
\a	警报
\e	ESC
\033	八进制数字符
\x1B	十六进制数字符
\c[	控制字符
\l	小写下一个字符
\u	大写下一个字符
\L	小写一直到\E
\U	大写一直到\E
\E	结束大小写修改

字符类元字符

元字符	说明
\d	找寻符合数值的字符串
\D	找寻符合不是数值的字符串

\s	找寻符合空白的字符串
\S	找寻符合不是空白的字符串
\w	找寻符合英文字母,数值的字符串
\W	找寻符合非英文字母,数值字符的字符串
[:class]	POSIX 字符类
\p	匹配某属性
\P	匹配非某属性
\X	匹配多字节的 Unicode 字符。
\C	匹配单个 8 位组。

属性元字符

例如我们可以把

```
$var =~ s/\s+$//;       #右边的 trim
```

写成：

```
$var =~ s/\p{Space}+$//;
```

属性元字符相关的定义在 C:\Perl\lib\unicore\lib 路径下的 Space.pl 文件中：

属性元字符	说明
Alpha	字母字符。
Alnum	字母数字字符。 例如，要把字母数字字符用_替换： *#单引号* *my $server_= '()/\@#$%%^&*123.ADDDasdf/\.3';* *#元字符取反* *$server_ =~s/\P{Alnum}/_/g;* *print "ok\n" if ($server_ eq '_____123_ADDDasdf___3');*
ASCII	ASCII 字符
Cntrl	控制字符

Digit	数字字符
Graph	图形字符
HAN	汉字字符
Lower	小写字符
Print	可打印字符
Space	空白字符
Upper	大写字符
Word	单词字符
XDigit	十六进制数

在正则表达式中匹配汉字仍然是一个问题，也许模块 Unicode::EastAsianWidth 有希望解决此问题：

```
use Unicode::EastAsianWidth;
$a="汉字abcd";

$Unicode::EastAsianWidth::EastAsian = 1;
print $&."\n" while $a =~ /\p{InFullwidth}+/g;
```

锚

经常需要声明模式发生的位置，这叫做锚定（anchoring）模式。

锚	说明
^	匹配字符串的开始(或行，如果使用/m)。
$	匹配字符串的结束(或行，如果使用/m)。
\<	匹配单词的开始。
\>	匹配单词的结束。

锚	说明
\b	匹配单词的边界(between \w and \W)。 例如模式： /\bJoe\b/ 匹配： "This is Joe" 不匹配： "That is Joe's bazooka"
\B	匹配除了单词的边界。
\A	仅匹配字符串的开始。
\Z	匹配字符串的结束或一个新行之前。
\z	仅匹配字符串的结束。
\G	匹配在 pos()，即匹配前面 m//g 剩下的。
\c	当和\g 一起使用时，抑制重置搜索位置。没有\c 时，搜索位置重置到字符串的开始。
(?=regexp)	向前看。如果它紧跟着 regexp，就匹配在它前面的东西。 例如： *$x = "I catch the housecat 'Tom-cat' with catnip";* *$x =~ /cat(?=\s+)/; # 匹配'housecat'中的'cat'*
(?!regexp)	向前看。如果匹配前面的字符串并且不跟着 regexp 就匹配前面的字符串。 例如： *$x = "foobar";* *$x =~ /foo(?!bar)/; # 不匹配，因为'bar' 在 'foo'之后。* *$x =~ /foo(?!baz)/; # 匹配， 'baz' 不在 'foo' 之后。*

锚	说明
(?<=regexp)	向后看。如果匹配后面的字符串并且以 regexp 开头就匹配后面的字符串。 例如： *$x = "I catch the housecat 'Tom-cat' with catnip";* *@catwords = ($x =~ /(?<=\s)cat\w+/g); # 匹配,* *# $catwords[0] = 'catch'* *# $catwords[1] = 'catnip'*

2.16.3 扩展使用

提取匹配的文本

返回正则表达式匹配的文本有两种方法：特殊变量和圆括号。

特殊变量	含义
$&	匹配的文本
$`	匹配前的所有文本
$’	匹配后的所有文本

使用圆括号，我们可以访问编号变量$1,$2,$3 等，它们在匹配结束时定义。编号变量比特殊变量不仅更灵活而且更有效。注意，这些变量是从$1 开始，而不是从$0 开始。$0 用于保持程序的名字。

例如要提取<product>和</product>间的值：

```
my $src = '<product>wid100</product>';
$src=~/^<product>(.+)<\/product>$/;

print "$1\n";
```

又如要解析如下这段放在文件中的文字：

1 小时 2 秒

1 小时

1 小时

59 分钟 58 秒

59 分钟 58 秒

59 分钟 58 秒

30 秒

可以使用这段代码：

```
foreach(`cat filename`)
{
    chomp;
    print /(\d*)小时/?"$1":"00", ":";
    print /(\d*)分钟/?"$1":"00", ":";
    print /(\d*)秒/?"$1":"00", "\n";
}
```

防止字符串被插值

主要有两种方法：

方法	例子
quotemeta	*$a="123a.1213*b123c.123*d";* *$b=".";* *$b= quotemeta $b;* *@match =$a=~/$b/g;* *print "匹配次数 :".@match."\n";*
\Q	在要保护的文本开始处加上\Q，结束处加上\E。 *$a="123a.1213*b123c.123*d";* *$b=".";* *@match =$a=~/\Q$b/g;* *print "match :".@match."\n";*

找出所有匹配的位置

```
my $target = "abcmnabILOVEcjhabcILOVEiuabILOVEccabc";
my $query    = "ILOVE";

my $i = 0;
while($target =~ m/$query/g) {
    printf "Match %d begins at %d\n",++$i, $-[0];
}
```

正则表达式的返回值可以用来统计匹配的次数，例如统计 alex|0||tom|1||jerry|1|| 中 |1|| 的次数：

```
$x= 'alex|0||tom|1||jerry|1||';
@test = $x =~ /\|1\|\|/g;
print scalar(@test);
```

2.17 格式

格式由多行构成。这些行可分为三类：描述、数据、注释。其中注释是可选的。

```
format STDOUT=

say @<<<<<<<< @<<<<<<<.

$hello, $world

.
```

上面的例子中，第一行定义了格式的名称。格式名称的作用是：我们将会通过 write() 函数指定格式名称来输出它。第二行是一个描述，定义了两列。每一列以一个@开头。第三行是数据，把变量名称和列按位置联系起来。最后一行总是一个独立的句点。注意，结尾的句点非常必要，因为它是格式的结束标记。

可以这样使用格式：

```
$hello = ‘hello’;
$world = ‘world’;
write STDOUT;
```

控制台将打印出：

```
say hello        world     .
```

因为 write 将 format 的内容实行占位符替换，输出到同名输出句柄 STDOUT。因此最后一句也可以这样写：

```
$~ = ‘STDOUT’;
write ;
```

或

```
write;
```

如果使用 IO::Handle 派生出来的文件句柄$fh，也可以这样输出格式：

```
$fh -> format(STDOUT);
$fh -> format_write();
```

占位符说明如下：

占位符	说明
<	左对齐。例如，如下是长度为 5 的左对齐列： @<<<<
>	右对齐。例如，如下是长度为 5 的右对齐列： @>>>>
|	居中对齐。例如，如下是长度为 5 的居中对齐列： @||||
#	右对齐数字。例如，如下是长度为 5 的右对齐数字列： @####

2.18 POD

Pod(plain old document)是一个专门用于给 Perl、Perl 程序、Perl 模块写文档的简单易用的标记语言。它用于把文档与相关的代码集成到一起。这个概念叫做自归档。我们也可以使用工具把它从源代码中提取出来转换成说明文档，以简单文本，html，man 手册等形式显示。Perl 自带了这样的一些工具，如：pod2text ，pod2html，pod2man，pod2latex，pod2usage。

Pod 由三类基本段组成：普通段、代码段和命令段。

普通段是基本的文字段。在普通段可以使用格式化代码，格式化代码列表如下：

格式化代码	说明
I<text>	斜体
B<text>	加粗
C<code>	代码
L<name>	超级连接
E<escape>	字符转义
F<filename>	用于文件名
S<text>	跨行

格式化代码	说明
X<topic name>	索引
Z<>	空格式代码

代码段用于代表一个代码块或其它不需要任何特殊解析或格式化的文本。一个代码段的开始一个字符是空格或制表符。

命令段用于特殊处理整段文本，例如标题或列表。命令段以 = 开始

命令段	说明
=head1 Heading Text **=head2 Heading Text** **=head3 Heading Text** **=head4 Heading Text**	标题
=over indentlevel **=item stuff...** **=back**	列表
=cut	结束一个 Pod 块，同时在前后还要附加一个空行。
=pod	Pod 块开始标记。
=begin formatname **=end formatname** **=for formatname text...**	开始一个特殊的格式，例如 html。当前支持的格式有``roff"，``man"，``latex"，``tex"，``text"和``html"。 下面是一个 html 的例子： =begin html <hr> <img src="thang.png"> This is a raw HTML paragraph =end html

2.19 模块

模块是 Perl 中可重用代码的基本单元。它可以是一个相关例程的集合或一个类。Perl

中的很多功能都是由模块提供的。Perl 中的模块相当于 java 中的 jar 包。perl 模块可以全部由 Perl 脚本实现。也可以是 Perl 脚本加上底层的 C 语言实现。

perl 的模块文件一般是以 pm 结尾（pm 是 Perl Module 的缩写）。它有时候也叫做库文件。相当于 c 的库文件。

本节将介绍模块的导出与导入，以及程序块与自动加载。创建可安装模块可以借助 Perl 中的工具软件 H2xs，细节将在后面的第七章中介绍。

2.19.1 导出

Exporter 模块提供了一个通用的 import 例程，可以通过继承来使用它。使用 Exporter 只需要修改它的类变量值即可导出。Exporter 类属性描述如下：

@EXPORT	导出子例程列表
@EXPORT_OK	显式导出子例程列表
%EXPORT_TAGS	导出列表分类
@EXPORT_FAIL	禁止导出列表
$Verbose	调试导出

方法描述如下：

import	导入
require_version	要求版本
export_fail	
export_tags	%EXPORT_TAGS 中的定义导出
export_ok_tags	%EXPORT_TAGS 中的定义显式导出

2.19.2 导入

导入模块的方法有：

方法	说明
do ‘perl source file’;	

require Module;	require 5.6.0; #测试 Perl 版本

use Module;	等价于：
	BEGIN{
	require Module;
	Module->import;
	}

use Module LIST;	等价于：
	BEGIN {
	require Module;
	import Module LIST;
	}

导入模块搜索路径来源于@INC 变量：

perl -I	perl –I/home/httpd/perl/lib

直接修改@INC	BEGIN{
	unshift @INC , ‘/home/httpd/perl/lib’;
	}

use lib	use lib ‘/home/httpd/perl/lib’;

2.19.3 程序块

按程序块的执行先后次序列表如下：

BEGIN	当 Perl 执行一个程序时，它会先编译源代码，如果编译成功，就开始执行源代码的第一个句子。有时你会想在源代码进行编译之前执行一些初始化的工作，例如在@INC 加入目录，以便在编译之前告诉 Perl 在哪里找模块文件。 你可以用 BEGIN 子程序来做到这种效果，当

	Perl 编译完这个子程序后，就会立即执行它。
CHECK	在编译结束之后执行。其作用是执行类型检查。
INIT	在编译结束之后执行。其作用是在主运行阶段开始之前初始化变量和数据结构。
END	你可以用 END 子程序，在程序执行完毕时执行一些工作。

2.19.4 线程安全

Perl 自从 5.6.0 支持解释线程。Perl 自从 5.7.2 支持 CLONE 特殊子例程。在创建新线程时调用 CLONE。

2.19.5 自动加载

通过定义一个 AUTOLOAD 例程，根据需要我们可以在运行时截取不存在的调用并以我们自己的方式处理。

我们可以伪装成不同的例程处理特定情况的许多例程。我们还可以通过使用 eval 和 sub 关键字即时生成例程。这是用来将一个模块变得强大且灵活但同时最小化内存占用的一种强有力的途径。

未找到的例程的名字放在特殊的变量 Packagename::$AUTOLOAD 中。

第3章 面向对象编程

Perl 使用“包”来支持面向对象编程。所以这里首先介绍 Perl 中的包，然后是对象。tie 是一个很妙的东西，它能在变量和类之间建立联系。最后深入挖掘的是实际的面向对象编程中不可缺少的设计模式。

3.1 包

Perl 中有两种变量类型：

包变量	属于某个包的变量可以在任何地方使用。在包内部使用包变量不需要加包名，在包外部使用包变量需要加包名。 例如下面这段代码中的变量全部是包变量： *package main;* *for ($i=0; $i<100; $i++)* *{* *$Other_package::time = localtime();* *print "$i at $Other_package::time\n";* *}* *package Other_package;* *print "last time was: $time\n";* *print "last index was: $main::i\n";*
词法变量	只能在代码的词法边界内使用的变量。 例如下面这段代码中的变量全部是词法变量： *package main;* *my $i;* *for ($i=0; $i<100; $i++)* *{* *my $time = localtime();* *print "$i at $time\n";* *}*

声明	*package Logger;*
使用严格的变量声明	*use strict;*
声明全局包变量	*use vars qw($ls_config_file $line $current_path $ls_compute_name);*
使用 local 局部化包变量	*local $, = $separator;*

声明词法变量	*my $scalar;*
结束	*1; #用于正确导入返回*

3.2 对象

正象 c++中，一个对象可以看成是一个特殊的结构体。在 perl 中，一个对象类只不过是一个包，而一个对象实例只不过是一个引用。不过这个引用不是通过\或[…]或{…}创建的，而是通过特殊函数 bless 创建的一类特殊引用。

它支持的面向对象特性有：

类方法

对象方法

多重继承

动态继承

多态

方法重载。

不支持私有属性及方法。

3.2.1 使用对象

创建对象	*$object = new My::Object::Class(@args);*
使用对象	*$object_result = $object->method(@args);*
访问属性	*$value = $object -> {'property_name'};* *$object->{'property_name'} = $value;*
调用类方法（静态方法）	*$result = My::object::class->classmethod(@args);*
调用对象方法	*$result = $object->method(@args);*
确定一个对象的类名称	*$classname = ref($obj);*

确定一个对象的祖先	*if ($object->isa("My::Object::Class")) {;}else{;}*
确定一个对象的方法	*if($object->can('method')) {return $object->method(@args);}*
确定一个对象的版本	*$version = $object->VERSION;*

3.2.2 创建对象

构造函数	*sub new{return bless{},shift;}* *sub new{* *my ($class,$name,$suit) = @_;* *my $self = bless{} , $class;* *$self->{'name'} = $name;* *$self->{'suit'} = $suit;* *return $self;* *}*
析构函数	*sub DESTORY{* *$self = shift;* *close $self->{'filehandle'};* *shutdown $self->{'socket'};* *}*
类方法 （静态方法）	类方法使用类名作为方法的第一个参数。 例子如下： *sub method {* *my $class = shift;* *...* *}*
对象方法	对象方法使用对象作为第一个参数。 例如： *sub method {* *my $self = shift;* *...* *}*

3.2.3 底层数据类型

对象依据引用来实现，可以选择任何类型的引用作为对象的基础。

数组	
标量	如下是一个类实现了一个加密的口令作为一个标量数组。 *package Password;* *use strict;*

	my @salt = ("A".."Z","a".."z","0".."9","/","."); *sub new* *{* *my ($class, $cleartext) = @_;* *my $salt = $salt[rand @salt].$salt[rand @salt];* *my $pw = crypt($cleartext,$salt);* *return bless \$pw, ref($class)||$class;* *}* *sub verify* *{* *my ($self, $candidate) = @_;* *my $salt = substr($$self,0,2);* *return crypt($candidate,$salt) eq $$self;* *}* 需要注意的是，尽管$pw 是一个词法变量，它并不是在调用 Password::new 结束时停止存在，因为 bless 返回一个$pw 的引用，而引用作为 new 的结果返回。
typeglob	以下是由 IO::Handle 使用的实际的构造函数。 *sub new {* *#判断传入的类，通过类方法，* *#对象方法或表示成"IO::Handle"，如果当成例程调用。* *my $class = ref($_[0]) || $_[0] || "IO::Handle";* *@_ == 1 or croak "usage: new $class";* *#创建一个匿名的 typeglob(从'Sybmol'模块)* *my $io = gensym;* *bless $io, $class;* *}*

3.2.4 继承

继承在 Perl 中的含义是：如果你不能发现在类实例化的对象中发现方法，就从该类继承的类中查找。

声明一个类的继承关系是通过把父类名称加到该类的@ISA 包变量中实现。例如，类 PerlGuru 声明它继承 PerlHacker 如下：

```
package PerlGuru;

@ISA = ( "PerlHacker" );
```

3.3 tie

它把一个变量和一个类的方法绑到一起。一旦绑定后，对绑定变量的操作将转化成对对象方法的调用。这样方法调用的复杂性就隐藏在我们熟悉的变量操作后了。

它的语法如下：

tie VARIABLE,CLASSNAME,LIST

这里，VARIABLE 代表要改造的变量名，CLASSNAME 代表对象名称。LIST 代表传递给构造函数的参数。tie()函数的返回值是新创建的对象的引用。以后也可以使用 tied() 函数取得该对象。

举例如下：

```
use IO::Dir;

#把哈西变量和 IO::Dir 变量绑定到一起，并使用当前路径初始化该对象。
#底层调用对象的 TIEHASH 方法。
tie %dir, IO::Dir, ".";

#遍历%dir 哈西变量。
#底层调用对象的 FIRSTKEY this 和 NEXTKEY this, lastkey 方法。
foreach (keys %dir) {

    #取得该目录下的对象大小。
    #$dir{$_}是通过 IO::Dir 的对象的 FETCH 例程实现的。
    #它返回的是一个 lstat 函数的引用。
    #lstat 函数返回文件的 13 个属性，所以程序可以通过$dir{$_}->size 来访问文件的大小。
    print $_, " " , $dir{$_}->size,"\n";
}
```

3.3.1 标量

TIESCALAR classname, LIST	tie $scalar, Class::Name, @args;
FETCH this,	$value = $scalar;
STORE this, value	$scalar = $value;
DESTROY this	undefine $scalar;
UNTIE this	

3.3.2 数组

TIEARRAY classname, list	tie @array, Class::Name, @list;
FETCH this, key	$value = $array[$key];
STORE this, key, value	$array[$key] = $value;

FETCHSIZE this	$size = $#array;
STORESIZE this, count	$#array = $size;
CLEAR this	@array = ();
PUSH this, list	push @array,@list;
POP this	$value = pop @array;
SHIFT this	$value = shift @array;
UNSHIFT this, LIST	unshift @array, @list;
SPLICE this, offset, length, LIST	splice @array, $offset, $length, @list;
EXTEND this, count	$array[$count] = $value;
DESTROY this	undef @array;
UNTIE this	

3.3.3 哈希表

TIEHASH classname, LIST	tie %hash, Class::Name, @list;
FETCH this, key	$vaule = $hash{$key};
STORE this, key, value	$hash{$key} = $value;
DELETE this, key	$done = delete $hash{$key};
CLEAR this	%hash = ();
EXISTS this, key	$exists = exists $hash{$key};
FIRSTKEY this	($key, $value) = each %hash; #第一次
NEXTKEY this, lastkey	($key, $value) = each %hash; #第一次以后
DESTROY this	undef %hash;
UNTIE this	

3.3.4 文件句柄

TIEHANDLE classname, LIST	tie $fh, Class::Name, @args;
READ this, scalar, length, offset	read $fh, $in, $size, $from;
READLINE this	$line = <$fh>;
GETC this	$char = getc $fh;
WRITE this, scalar, length, offset	write $fh, $out, $size, $from;
PRINT this, LIST	print $fh @args;
PRINTF this, format, LIST	printf $fh $format @values;
BINMODE this	
EOF this	
FILENO this	
SEEK this, position, whence	
TELL this	
OPEN this, mode, LIST	
CLOSE this	close $fh;
DESTROY this	
UNTIE this	

3.4 设计模式

3.4.1 Iterator(遍历)

很多情况下需要每次遍历结构中的一个元素。这些东西包括简单的如数组，中等复杂的如 hash 中的键值，和复杂的东西比如树中的节点。

Iterator 涉及到三部分：数据，知道如何取得下一个 item 的 iterator，还有调用 iterator

的控制器。

在 java 中通常一个能遍历的对象返回一个 iterator 对象。例如，考虑如下这段代码，使用一个 iterator 遍历一个 Java 中的哈希表的关键字。

```
for (Iterator iter = hash.keySet().iterator(); iter.hasNext();) {
        Object key   = iter.next();
        Object value = hash.get(key);
        System.out.println(key + "\t" + value);
}
```

HashMap 对象有一个能遍历的对象：它的关键字。通过调用 keySet 的 iterator 方法，keySet 集合会给你一个 Iterator。如果有更多的东西待遍历，The Iterator 的 hasNext 方法将返回 true ，否则返回假。它的下一个方法以 Iterator 管理的顺序给出下一个对象。用这个 key，HashMap 给出 get(key)返回的下一个值。

在 Perl 中，任何内在的或用户自定义的能够遍历的对象都有一个方法，能够返回一个遍历项的序列。要遍历这个列表，只需要简单的把它放在一个 foreach 循环的括号中。因此，上边的 hash key 遍历者的 Perl 版本是：

```
foreach my $key (keys %hash) {
    print "$key\t$hash{$key}\n";
}
```

上面的例子来源于语言的核心。为了看到 foreach 在用户定义模块上完全实现 iterator 模式，可以参见一个源于 CPAN: XML::DOM 的例子。XML 的 DOM 接口是在 Java 中声明的 。DOM 上可以调用的一个方法是 getElementsByTagName。在 DOM 声明中，它返回一个 NodeList，是一个 Java 集合类。然而，NodeList 象上面的 Java 集合类一样工作。你必须请求一个 Iterator，然后遍历这个 Iterator。

在 Perl 中实现 DOM 时，getElementsByTagName 返回一个 Perl 列表。只需要这样要遍历：

```
foreach my $element ($doc->getElementsByTagName("tag")) {
    # ... process the element ...
}
```

这和冗余的 Java 版本形成鲜明的对比：

```
NodeList elements = doc.getElementsByTagName("tag");
for (Iterator iter = elements.iterator(); iter.hasNext();) {
        Element element = (Element)iter.next();
        // ... 处理元素 ...
}
```

Perl 的一个优美之处在于它能够以强大的方式组合过程化，面向对象和核心概念。

在 Perl 中使用了 iterator 模式的有：

- XML::*, HTML::*：有一些 XML 和 HTML 模块可以看成是流式的解析器。当我提供一段文本，这些模块把他们分成小段，提供给我与这些小段交互的机会。解析器作为 iterator 和控制者，它们知道如何取得下一段文本，我提供碰到这些文本的行为。

- Tie::DirHandle：Perl 内建的函数 readdir() 是一个 iterator。在标量上下文中，它提供给我指定路径下的下一个文件名 ，在数组上下文中提供所有的上下文。它定义了遍历的逻辑而由我提供控制。这个模块隐藏了 readdir()，我能够使用文件句柄 iterator 和路径句柄交互。我用行输入操作符而不是 readdir()取得下一个文件名。

3.4.2 Decorator(修饰)

在通常的操作中，一个 decorator 包装一个对象，和被包装的对象一样响应同样的 API 。例如，假如我添加一个压缩 decorator 到一个写文件对象。调用者传递一个文件写入器给 decorator 的构造函数，而且调用 decorator 的写方法。Decorator 的写方法首先压缩数据，然后调用它所包装的文件写入器的写方法。只要所有的写入器响应同样的 API，任何其他类型的写入器都能用同一个 decorator 包装。其他的 decorators 也可以在链接中使用。文本能够被一个 decorator 从 ASCII 转换到 unicode 而且被另一个压缩。 在这里，decorator 的顺序是重要的。

在 Perl 中，我们可以使用对象来实现这个模式。有时候，可以直接依赖内部语法实现。

I/O 是最常用的 decoration。Perl 直接提供了 I/O decoration。考虑上面的例子：压缩并写入 。有两个方法做这件事情。

使用管道方法。

在 Perl 中，当我打开一个文件用于写入，可以通过管道方法 decorator。 例子如下：

```
open FILE, "| gzip > output.gz" or die "Couldn't open gzip and/or output.gz: $!\n";
```

现在，写的每一行都通过 gzip 压缩后到达 output.gz 文件。 为了达到目的，需要有 gzip 这样的工具支持管道方法。这里还有一个效率问题，为了 gzip 这步，操作系统将创建一个新的进程。而进程创建是除了 I/O 操作以外最慢的动作。

如果你需要更多的控制数据流，就需要用 Perl 的 tie 机制自己 decorate。 在 Perl 6 中，它会更快，更容易使用，而且更强大，但是它在 Perl 5 中也能工作。它在 Perl 的面向对象框架内工作。

假设我需要在输出的每一行前面加一个时间戳。如下是一个实现它的 tie 类。

```
package AddStamp;
use strict; use warnings;

sub TIEHANDLE {
    my $class    = shift;
```

```
    my $handle = shift;
    return bless \$handle, $class;
}

sub PRINT {
    my $handle = shift;
    my $stamp  = localtime();

    print $handle "$stamp ", @_;
}

sub CLOSE {
    my $self = shift;
    close $self;
}
```

但是这个类是最小的，在现实生活中，需要更多的代码使 decorator 更稳定和完整。例如，上面的代码不检查文件句柄是可写的，也没有提供 PRINTF，因此调用 printf 函数会失败。

它的工作原理是：tie 文件句柄类的构造函数叫做 TIEHANDLE。它的名字是固定的而且是大写。这是一个类方法，因此第一个参数是类名。另外一个参数是一个大开的输出句柄。构造函数仅仅 bless 一个这个句柄的引用，然后返回这个引用。

PRINT 方法接受 TIEHANDLE 中创建的对象加上提供给 print 的所有参数。它计算时间戳，然后使用真正的 print 函数发送它和其他的参数。这是典型的 decoration 的工作方式，decorating 对象就像原来的 print 函数一样响应 print 调用。

CLOSE 方法关闭句柄。可以继承 Tie::StdHandle 让这个方法有更多的功能。

一旦把 AddTimeStamp.pm 放到 lib 路径，就能像这样使用它：

```
use strict;
use warnings;

use AddStamp;

open LOG, ">output.tmp" or die "Couldn't write output.tmp: $!\n";
tie *STAMPED_LOG, "AddStamp", *LOG;

while (<>) {
        print STAMPED_LOG;
}

close STAMPED_LOG;
```

在通常的写入方式打开文件后，使用内建的 tie 函数绑定 LOG 句柄到 AddStamp 类，返回 STAMPED_LOG 句柄。以后，就可以只使用 STAMPED_LOG。

如果有其他的 tied decorator，还可以把 tied 句柄传递给它们。Perl 5 ties 唯一不好的地方是速度比普通的操作慢。然而，一般来说，磁盘和网络是瓶颈，因此像这样内存中的低效无关紧要。

这项技术能与其它很多内部类型工作：标量，数组，哈希和文件句柄。

3.4.3 Flyweight(享元)

需要达到的效果是：

对与实例无关的对象（例如实例是常量或随机的），可能的时候，请求一个新对象，应该返回他们已经收到的同样的一个对象。

Flyweight 模式的基本思想是重用对象。Perl 远远超出了 GoF 考虑的。Larry Wall 经过改进后把它引入到 Perl 6 的内核中去了。

如果对象是与实例相关的则不能应用该模式。在能应用模式的地方使用 Flyweight 可以节约时间和内存。

如下是一个 Perl 例子。 假如我需要为掷骰子游戏提供一个骰子类。骰子类看起来象这样：

```
package CheapDie;
use strict;
use warnings;

use Memoize;
memoize('new');

sub new {
        my $class = shift;
        my $sides = shift;
        return bless \$sides, $class;
}

sub roll {
        my $sides = shift;
        my $random_number = rand;
```

```
        return int ($random_number * $sides) + 1;
}

1;
```

第一眼看来，它和其他类并无区别。它有一个叫做 new 的构造方法。构造方法把接收的骰子面数数字存储到一个对象成员变量，返回一个对象的 blessed 引用。Roll 方法计算 一个随机的数字，根据面数返回结果。

唯一陌生的就是这两行：

```
use Memoize;
memoize('new');
```

memoize 函数修改了调用者包的符号表以便把 new 包裹起来。包裹函数检查输入参数(在这里是骰子面数)。如果以前没有看见这些参数，它就调用该函数，把结果存储到缓存并返回给用户。这会比不使用 Memoize 花更多的时间和内存。

当该方法再次调用时就会节省了。当 wrapper 注意到一个使用同样参数调用，它就不会调用方法，而是发送缓存的对象。 作为对象的实现者，我们不需要做任何特殊的工作。如果你的对象很大，或构造起来很慢，这个技巧就可能节省时间和内存。

需要注意的是，一些方法不能使用这种技巧。例如，如果我 memoize roll 方法，它就会每次都返回同样的数字, 而不是随机的。

Memoize 也能用在非面向对象的场合。

3.4.4 Singleton(孤子)

在 flyweight 模式，我们看到有时候所有人能共享资源。GoF 把每一个人都需要共享的单一资源叫做 singleton 模式。这样的资源可能是一个配置参数的哈希表。每一个人都要能够看到它，但是它只能在启动时建立（也可能在重新配置时重建）。

在大多数情况下，可以只使用 Memoize(参见 flyweight 模式)。 在这种情况下，任何想要取得资源的人可以调用构造函数。第一个这样做的人引起构建对象并收到对象。随后人们调用构造函数，但是他们接收到的是原先构造好的对象。

有很多其他的方法达到同样的效果。例如，如果你的调用者可能传递给你没有想到的值，Memoize 就可能产生多个实例，每一个参数集一个实例。 在这时候，用 CPAN 上的 Cache::FastMemoryCache 管理 singleton 可能更有意义。借助 BEGIN 块也可以制作 singleton 。因为 bless 不一定要用在方法上。你可以这样：

```
package Name;
my $singleton;
BEGIN {
```

```
        $singleton = {
            attribute => 'value',
            another     => 'something',
        };
        bless $singleton, "Name";
}

sub new {
        my $class = shift;
        return $singleton;
}
```

这样做更直接，避免了 Memoize 的过度使用。

这种方法的另外一种变种举例，例如需要一个全局计数器。如果程序的一部分增加计数器，使用计数器的部分能够看到新值，代码如下：

```
package Counter;

my $singleton = undef;

sub new
{
my( $class ) = @_;

return $singleton if defined $singleton;

my $self = 0;
$singleton = bless \$self, $class;

return $singleton;
}

sub value
{
my $self = shift;
return $$self;
}

sub increment
{
my $self = shift;
return ++$$self;
}
```

```
1;
```

私有的类变量$singleton 存储唯一的类实例。

使用它的代码如下：

```
use Counter;

my $count = Counter->new();
my $count2 = Counter->new();

print "The counters are the same!\n" if $count eq $count2;

print "Count is now ", $count->increment, "\n";
print "Count is now ", $count2->increment, "\n";
print "Count is now ", $count->increment, "\n";

print "Count is now ", $count2->value, "\n";

print "The counters are the same!\n" if $count->value eq $count2->value;
```

假设我需要在程序的不同部分和数据库打交道——可能是在启动时加载配置信息，选择一些列，或在程序结束时插入日志信息。在一个大型程序中，这些功能可能存在于不同的模块中，这时以解耦合的方式使用数据库连接是有必要的。

3.4.5 Façade(外观)

Facade 是一个提供给应用程序的接口，它封装了功能的实现细节。Facade 解耦合了调用层和实现层，使它们不互相依赖，容易开发和使用，并促进了代码重用。

CPAN 上的一些 Perl 模块 可作为 façade 的使用例子。

为了从 web 站点请求一个简单的文件，我必须建立一个到 web 站点的连接，使用合适的 HTTP 协议请求资源，接受 HTTP 响应，解析响应，最后处理数据。，如果我想处理公共的 web 特性如 cookies、forms 和缓存，我不得不作更多的工作。如果我想从 FTP 服务器取得资源而不是 web 服务器，我需要处理一个完全不同的协议。

如果我思考这个问题，可能需要一些对象：连接，请求，响应和资源。然而，我只是想取得资源，然后继续我的实际工作。进行这样的编码会是相当耗时的。LWP 模块 (Library for WWW in Perl) 为做这些事情提供了一个 façade。我只需要告诉 LWP 取得资源，它做所有其他的事情，包括所有与 HTTP，FTP 协议相关的细节。

在下面的代码中，我使用了 LWP::Simple ，不需要说明协议，与服务器建立连接的方法，或解析服务器响应。只要我知道 URL，就能取得资源。

```
use LWP::Simple qw(get);
my $url = 'http://www.perl.org';
my $data = get( $url );
```

LWP::Simple 模块是一个 façade——它提供一个简单的接口统一协议，网络和解析方面的问题，以便达到我需要做的一件事情——取得资源。如果有人改变了 LWP 或底层的实现，我不需要改变代码就能从改进中获益。

3.4.6 Abstract Factory(抽象工厂)

如果你想做一个独立于平台的程序，你需要一个方法和底层的系统打交道而不需要记录每一个平台的 API。这里就可以用到 factory。用户代码请求一个类的实例。该类导出一个用于当前平台的合适的子类实例。该类就叫做抽象工厂(或简称为 factory)。就像我们下面看到的，平台可能是一个数据库。因此 factory 可能返回一个适用于特定数据库的对象，但是所有的对象有相同的 API。

为了展现基本思想，下面是一个例子导出两个类型中的一个。这个例子中有四个代码文件。头两个是问候者类。

```
package Greet::Repeat;

sub new {
    my $class       = shift;
    my $self        = {
        greeting => shift,
        repeat     => shift,
    };
    return bless $self, $class;
}

sub greet {
    my $self = shift;
    print ($self->{greeting} x $self->{repeat});
}

1;
```

这个问候者构造函数请求一个问候语和重复次数。它把参数存到一个 hash，返回一个它的 blessed 引用。当请求问候时，它重复的打印问候。

```
package Greet::Stamp;
use strict; use warnings;

sub new {
        my $class       = shift;
```

```
        my $greeting = shift;
        return bless \$greeting, $class;
}

sub greet {
        my $greeting = shift;
        my $stamp    = localtime();
        print "$stamp $$greeting";
}

1;
```

这个问候者仅请求一个问候字符串，问候者仅仅需要一个问候字符串，然后它 blesse 一个引用。当要求问候的时候，它打印出当前的时间和问候语。

如下是 factory：

```
package GreetFactory;
use strict; use warnings;

sub instantiate {
        my $class          = shift;
        my $requested_type = shift;
        my $location       = "Greet/$requested_type.pm";
        my $class          = "Greet::$requested_type";

        require $location;

        return $class->new(@_);
}

1;
```

一个 Perl factory 看起来和其它语言中的 factory 一样。这个 factory 仅有一个方法。它返回请求的类型给调用者。它使用调用者的请求类型作为类名称和类所在的 Perl 模块名称。

最后，你能像这样使用这个 factory：

```
#!/usr/bin/perl
use strict; use warnings;

use GreetFactory;

my $greeter_n = GreetFactory->instantiate("Repeat", "Hello\n", 3);
$greeter_n->greet();
```

```
my $greeter_stamp = GreetFactory->instantiate("Stamp", "Good-bye\n");
$greeter_stamp->greet();
```

可调用 GreetFactory 的 instantiate 方法取得每个 greeter 对象，传递你想要的类名称给它，以及类构造器需要的任何参数。

这个例子展示了基本的想法。任何新加的问候者类必须有一个像 Greet::Name 这形式的名字，而且实现文件是 Name.pm 文件放到一个 Greet 子路径。然后，调用者就可以使用它而不需要改变 factory。

在 Perl 中使用了 Abstract Factory 模式的有：

- DBI (DataBase Interface)： DBI 是一个提供各种数据库联接的类工厂。用户代码调用 DBI 的 connect 方法，提出 database 类型和与数据库建立联接需要的各种信息。DBI 能够根据请求加载任何系统中已安装的 DBD (DataBase Driver) 。用户代码能够通过同样的 DBI API 使用它们。如下例：

```
use DBI;
my $dbh         =
    DBI->connect("dbi:mysql:mydb:localhost", "user", "password");
...
my $sth         = $dbh->prepare('select * from table');
...
```

第4章 常用模块

Perl 中正是因为有了很多有用的模块，它的功能才变得如此强大。Perl 自带了很多常用的模块，但是更多的模块需要我们自己安装。因此本章首先介绍安装模块，然后是按功能分类的模块介绍。关于数据库相关的模块将单独在下一章介绍。

4.1 手动安装模块

本节将首先介绍模块安装的原理及 Makefile 文件，然后介绍在 Active Perl 中安装模块的方法——即采用 Active Perl 的安装工具 ppm（Programmer's Package Manager）安装模块的方法，最后介绍各种 perl 版本通用的手工安装模块的方法。

4.1.1 Makefile

了解 makefile 是安装模块的预备知识。一个 makefile 由一系列变量定义和依赖规则组成。makefile 中的一个变量是一个字符串变量。它就象 C 预编译的宏替换一样工作。变量长用来表示搜索路径，编译选项，运行程序的名字等。像在 Perl 中一样，变量不需要预定义，直接赋值即可。例如：

```
CC = gcc
```

将创建一个名叫 CC 的变量。把它赋值为 gcc。变量的名称是大小写敏感的。一般的命名规范是所有的变量名都大写。

可以定义自己的变量名，但是有一些标准名称和习惯用法能使写一个 makefile 更加容易些。最重要的变量是：CC, CFLAGS, 和 LDFLAGS。

CC	C 编译器的名称。在多数 unix 的 make 版本中，缺省是 cc 或 gcc，windows 中是 cl。
CFLAGS	一个传给 C 编译器的选项列表。通常用来设置包含路径(-I) 或构建调试版本 (-g)。
LDFLAGS	传给 linker 选项列表。通常用来包含应用相关的库文件(-l) 和设置库搜索路径(-L)。

要引用一个变量的值，可在美元符($)后跟着圆括号或花扩号括起来的名字。

```
CFLAGS = -g -I/usr/class/cs107/include

$(CC) $(CFLAGS) -c binky.c
```

第一行设置变量 CFLAGS 的值，打开调试信息并增加路径/usr/class/cs107/include 到头文件搜索路径。第二行使用 CC 取得编译器的名字和 CFLAGS 变量取得编译选项。没有赋值的变量是空字符串。

makefile 的第二个组成部分是依赖/构建规则。一个规则告诉如何基于一列选定文件的改变构建一个目标文件。规则的顺序没有任何关系，除了第一个规则是缺省规则。当没有参数调用 make 时将应用缺省规则（最常用的方式）。

一个规则通常由两类行组成：一个依赖行，接着是一条或多条命令行。如下是一个规则的例子：

```
binky.o : binky.c binky.h akbar.h

    $(CC) $(CFLAGS) -c binky.c
```

这个规则的意思是当文件 binky.c, binky.h 或 akbar.h 改变时，object 文件 binky.o 必须重建。目标 binky.o 依赖于这三个文件。程序员通过 makefile 表示源文件间的相互依赖关系。在上面的例子中，binky.c 源代码 #includes binky.h 和 akbar.h——如果任何.h 文件改变了，binky.c 都要重新编译。 借助这一点，有一些工具通过扫描源文件可以自动生成 makefile 文件，比如 Perl(还有 autoconf 和 Ant 等)。命令行列出构建 binky.o 的命令。

也就是说，依赖行说明什么时候去做，命令行说明去做什么。

命令行由一个 tab 缩进开头。千万不要以空格开头，即使那样看起来是一样的。这样做据说是为了向后兼容。

命令行也是可以忽略的，make 将根据源文件的扩展名使用一个缺省的构建规则。例如.c 看成 C 文件, .f 看成 Fortran 文件。C 文件的缺省构建规则如下：

```
$(CC) $(CFLAGS) -c source-file.c
```

依赖上面的缺省构建规则的情况很常见，大多数调整都可以通过改变 CFLAGS 达到。如下是一个简单但是典型的 makefile。它编译包含在 main.c, binky.c, binky.h, akbar.c,akbar.h, and defs.h.中的 C 源文件。这些文件将产生中间结果文件 main.o,binky.o, and akbar.o。这些文件将链接到一起产生可执行程序。在 makefile 文件中忽略空行，而注释和 Perl 中一样以#开头。

```
## A simple makefile

CC = gcc

CFLAGS = -g -I/usr/class/cs107/include

LDFLAGS = -L/usr/class/cs107/lib -lgraph
```

```
PROG = program

HDRS = binky.h akbar.h defs.h

SRCS = main.c binky.c akbar.c

## This incantation says that the object files

## have the same name as the .c files, but with .o

OBJS = $(SRCS:.c=.o)

## This is the first rule (the default)

## Build the program from the three .o's

$(PROG) : $(OBJS)

tab$(CC) $(LDFLAGS) $(OBJS) -o $(PROG)

## Rules for the source files -- these do not have

## second build rule lines, so they will use the

## default build rule to compile X.c to make X.o

main.o : main.c binky.h akbar.h defs.h

binky.o : binky.c binky.h

akbar.o : akbar.c akbar.h defs.h

## Remove all the compilation and debugging files

clean :

tabrm -f core $(PROG) $(OBJS)

## Build tags for these sources

TAGS : $(SRCS) $(HDRS)

tabetags -t $(SRCS) $(HDRS)
```

第一个(缺省的)目标从三个.o 文件构建程序 。接下来的三个目标例如"main.o : main.c binky.h akbar.h defs.h"标识构建.o 文件所依赖的源文件。这些规则声明了需要构建什么，但是忽略了命令行。因此将使用缺省的规则从同名的.c 文件构建.o 文件。最后，make 自动的知道 X.o 总是依赖它的原文件 X.c，因此 X.c 能从规则中省略。第一个规则也可以写成没有 main.c ： "main.o : binky.h akbar.h defs.h"。

然后的目标 clean 和 TAGS，执行其它的方便操作。clean 目标用来删除所有的 object 文件，可执行文件，和一个 core 文件（调试时产生的），这样就可以从头再次执行构建操作了。 TAGS 规则创建一个 tag 文件便于 Unix 编辑器查找符号定义。

4.1.2 Makefile.PL

Makefile.PL 是为整个项目生成 Makefile 的 Perl 脚本。这是通过调用包含在 ExtUtils::MakeMaker 库中的 WriteMakefile 函数实现的。简单的 Makefile.PL 类似于：

```
use ExtUtils::MakeMaker;
WriteMakefile(
    'NAME'              => 'myproject',
    'VERSION_FROM'      => 'lib/MyModule.pm', # finds $VERSION
);
```

在这个示例中，我们将所需的两段信息传递给了 WriteMakefile：NAME 和 VERSION。Name 是显式指定的。Version 是通过使用 VERSION_FROM 变量隐含指定的。这向 MakeMaker 表明，哪一个模块包含代表整个项目的$VERSION 变量。

构造 Makefile 所需的所有其它变量将取自 Perl 解释器缺省值。这些包含类似 PREFIX 的东西，它是应该在其中安装的应用程序的路径（通常缺省设为 /usr 或 /usr/local），MAN1PATH 是应该安装第一节（用于命令）帮助手册的位置，INSTALLSITELIB 是应该安装库的位置。

项	说明
ABSTRACT	描述模块的一行。将包含在 PPD 文件中。
ABSTRACT_FROM	包含包描述的文件名。MakeMaker 在 POD 中查找匹配 /^($package\s-\s)(.*)/的一行。典型的，这是``=head1 NAME" 部分的第一行。. $2 成为 abstract。
AUTHOR	包含包作者的名字(和 e-mail 地址)的字符串。用在 PPM(Perl Package Manager)的 PPD (Perl Package Description) 文件。
BINARY_LOCATION	用于为二进制包创建 PPD 文件。可能设置成专用于某种结构的二进制压缩包的一个绝对或相对路径或 URL。例如： perl Makefile.PL BINARY_LOCATION=x86/Agent.tar.gz

项	说明
	构建一个 PPD 包 that references a binary of the Agent package, 位于相对于 PPD 本身的 x86 路径。
INC	包含文件路径，例如："-I/usr/5include -I/path/to/inc"。
NAME	Perl 模块名 (DBD::Oracle)。缺省是路径名，但是应该在 Makefile.PL 中显式定义。
PREREQ_PM	哈希引用：运行本模块需要的模块的名字是键值，对应的版本是值。如果要求的版本号是 0，代表仅仅检查该模块是否存在，忽略版本检查。 例如： PREREQ_PM => {Data::Dumper=>q[2.09], Parse::RecDescent=>q[1.8] }
VERSION	发布包的版本号。缺省是 0.1。
VERSION_FROM	除了在 Makefile.PL 中指定版本，还可以让 MakeMaker 解析一个文件决定版本号。解析例程要求 VERSION_FROM 指定的文件包含唯一的一行计算版本号。文件中的第一行包含正则表达式： /([\$*])(([\w\:\']*)\bVERSION)\b.*\=/ 会用 eval()计算，而 eval()后命名变量的值将会赋值给 MakeMaker 对象的 VERSION 属性。如下行将会成功解析。 $VERSION = '1.00'; *VERSION = \'1.01'; ($VERSION) = '$Revision: 1.63 $ ' =~ /\$Revision:\s+([^\s]+)/; $FOO::VERSION = '1.10'; *FOO::VERSION = \'1.11'; our $VERSION = 1.2.3; # new for perl5.6.0 如下则是错误的写法： my $VERSION = '1.01'; local $VERSION = '1.02';

项	说明
	local $FOO::VERSION = '1.30'; 在 VERSION_FROM 中的文件名不增加成为 Makefile 的依赖。这样做是不正确的，但是如果该文件任何小的改动都会导致重写 Makefile 也是一件很痛苦的事情。 如果你想保证 Makefile 总是包含正确的 VERSION 宏，可以手工添加依赖关系如下： depend => { Makefile => '$(VERSION_FROM)' }
clean	{FILES => "*.xyz foo"}
depend	{ANY_TARGET => ANY_DEPENDECY, ...}(ANY_TARGET must not be given a double-colon rule by MakeMaker.)
dist	{TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => '.gz', SHAR => 'shar -m', DIST_CP => 'ln', ZIP => '/bin/zip', ZIPFLAGS => '-rl', DIST_DEFAULT => 'private tardist' } 如果你指定 COMPRESS，SUFFIX 也应该修改，因为需要告诉压缩的目标文件。如果你想保留你的文件上的时间戳，把 DIST_CP 设置成 ln 可能是有用的。DIST_CP 能接受值'cp'，它拷贝文件，'ln'链接文件，而'best' 会拷贝符号链接而链接其他的。缺省是'best'。
dynamic_lib	{ARMAYBE => 'ar', OTHERLDFLAGS => '...', INST_DYNAMIC_DEP => '...'}
test	{TESTS => 't/*.t'}

4.1.3 在 Unix 下安装

以 DBI 模块为例，首先从 cpan(http://www.cpan.org)上下载模块文件 DBI-1.38.tar.gz，然后按如下步骤：

1．解压

>gzip -dc DBI-1.38.tar.gz | tar -xof –

2．构建

生成 make 使用的 Makefile 文件：

>perl Makefile.PL

生成可安装文件：

```
>make
```

生成测试：

```
>make test
```

3．安装（或重新安装）

```
>make install
```

4．先卸载然后重新安装

```
>make install UNINST=1
```

4.1.4 CPAN 安装

可以使用 CPAN.pm 模块来安装。这种安装方式在 Unix 和 Windows 下都可以使用。从防火墙后使用 CPAN 安装模块时需设置环境变量：

```
http_proxy = http://192.168.0.1:3128

ftp_proxy= http://192.168.0.1:3128
```

安装一个模块可以输入以下命令：

```
>perl -MCPAN -e “install SQL::Translator”
```

这时会进入一个交互界面。在一般情况下，不需要更改缺省值，使用回车就行了。

4.1.5 ppm 安装

ppm 是 Active perl 特有的管理、安装模块的命令行接口程序。它的基本过程是从指定的存储库中搜索编译好的安装模块（打包成 zip），然后安装。

从防火墙后使用 ppm 安装模块时需设置环境变量：

```
HTTP_proxy = http://proxy:8080/

HTTP_proxy_user = username
```

HTTP_proxy_pass = password

安装模块：Authen-NTLM 和 LWP:Authen:Ntlm.pm。其中 Authen-NTLM 有两个相同的模块，必须安装正确的（Mark J Bush 的版本）。然后就可以通过输入命令 ppm 使用了。

正常情况下，安装的过程很简单。输入如下命令即可：

```
>PPM

>INSTALL PACKAGENAME

>QUIT
```

也可以直接用一个命令：

```
>ppm install PACKAGENAME
```

如：

```
>ppm install Parse-Tokens
```

此外还可以 search 所需的模块，看它是否存在。

可以从 http://ppm.activestate.com/PPMPackages/zips/8xx-builds-only/Windows/下载到本地安装包。专门编译的一般已压缩成.zip 文件。解压.zip 文件，直接安装即可。如果不正确，可先检查 PPD 文件中的描述，必要时修改 ppd 文件。一个 ppd 文件就是一个 html 格式的文件。它记录了对模块的一些简单说明，如名称，版本号，依赖其它的包名称，操作系统名称等。样例如下：

```
<SOFTPKG NAME="DateTime" VERSION="0,08,0,0">

    <TITLE>DateTime</TITLE>

    <ABSTRACT>DateTime base objects</ABSTRACT>

    <AUTHOR>Dave Rolsky &lt;autarch@urth.org&gt;</AUTHOR>

    <IMPLEMENTATION>

        <DEPENDENCY NAME="Class-Factory-Util" VERSION="1,3,0,0" />

        <DEPENDENCY NAME="DateTime-TimeZone" VERSION="0,1,0,0" />

        <DEPENDENCY NAME="Params-Validate" VERSION="0,52,0,0" />
```

```
        <DEPENDENCY NAME="Test-More" VERSION="0,0,0,0" />

        <DEPENDENCY NAME="Time-Local" VERSION="0,0,0,0" />

        <OS NAME="MSWin32" />

        <ARCHITECTURE NAME="MSWin32-x86-multi-thread-5.8" />

        <CODEBASE HREF="D:\lg\perl\perl_datetime\DateTime-0.08.tar.gz" />

    </IMPLEMENTATION>

</SOFTPKG>
```

其中的<ARCHITECTURE NAME>是比较关键的一项，声明了只能在 Perl 的哪个版本安装。例如<ARCHITECTURE NAME="MSWin32-x86-multi-thread-5.6" />表示只能在 Perl5.6 中安装。如果在 Perl5.8 中安装则会报错：

```
Error installing package 'xxx.ppd': Read a PPD for 'xxx.ppd', but it is not intended for this build of Perl (MSWin32-x86-multi-thread-5.8)
```

如果是<ARCHITECTURE NAME="MSWin32-x86-multi-thread" />，在 Active Perl 的不同版本可能能够正常使用。如果是在 Perl5.8 中则需要修改此项成为<ARCHITECTURE NAME="MSWin32-x86-multi-thread-5.8" />即可装上。

4.1.6 构建模块

这种安装方式至少需要 nmake(Visual C++中包含)工具。但是实际操作中，很多复杂的包安装往往需要用到 Visual C++，如果没有 Visual C++，用免费的 Windows SDK 替代也是可以的。

首先利用 Perl 本身构建 Makefile。

```
>perl Makefile.PL
```

然后用 nmake 编译 C 语言源程序。

```
>nmake
```

编译时，有的包还可能允许带一些附加的参数，

还可以测试编译后的程序。

>nmake test

最后是安装它。

>nmake install

如果需要的话，可以构建它的发布包，以便下次安装时使用。

>nmake ppd

要生成的 Makefile 可以通过在命令行增加参数，以 KEY=VALUE 的形式。例如：

perl Makefile.PL PREFIX=/tmp/myperl5

对生成的 Makefile 其他可做的操作包括：

make config # 检查 Makefile 是否是最新的。

make clean # 删除本地临时文件

make realclean # 删除导出的文件(包括 ./blib)

make ci # 检查在 MANIFEST 文件中的所有文件。

make dist # see below the Distribution Support section

4.1.7 制作 PPM 安装包

因为很多模块（如 DBD-Oracle，DBD-Sybase 等）没有提供现成的 PPM 安装包。为了方便发布，可以自己制作安装。为了制作首先需要 windows 下的 tar 和 gzip 工具。可以从 http://www.weihenstephan.de/~syring/win32/UnxUtils.html 下载 tar 和 gzip。如下以制作 DBD-Oracle 包示例。

首先是正常的编译。

>Makefile.PL

>nmake

结果文件放在 blib 目录下，

```
>tar cvf DBD-Oracle-1.14.tar blib

>gzip --best DBD-Oracle-1.14.tar

>nmake ppd
```

生成的 DBD-Oracle.ppd 文件，然后修改其中的 <CODEBASE HREF="MSWin32-x86-multi-thread-5.8/DBD-Oracle-1.14.tar.gz" />，最后该文件内容如下：

```
<SOFTPKG NAME="DBD-Oracle" VERSION="1,14,0,0">
    <TITLE>DBD-Oracle</TITLE>
    <ABSTRACT>Oracle database driver for the DBI module</ABSTRACT>
    <AUTHOR>Tim Bunce (dbi-users@perl.org)</AUTHOR>
    <IMPLEMENTATION>
        <DEPENDENCY NAME="DBI" VERSION="0,0,0,0" />
        <OS NAME="MSWin32" />
        <ARCHITECTURE NAME="MSWin32-x86-multi-thread-5.8" />
        <CODEBASE HREF="DBD-Oracle-1.14.tar.gz" />
    </IMPLEMENTATION>
</SOFTPKG>
```

文件 DBD-Oracle.ppd 和 DBD-Oracle-1.14.tar.gz 就是我们得到的结果。把它们拷到我们自己的存储库目录下就可以安装了。

4.1.8 查找已安装模块

要察看单个模块是否已安装，可以在命令行输入：

```
> perl -MModuleName -e " print "
```

如果存在就不会出错，否则会报错。

或输入：

> perl -e "use ModuleName"

可以通过编程产生一个所有可用模块的列表：

```
use File::Find;
# Module list
foreach my $dir (@INC){
   find sub {
      print "$File::Find::name\n" if /\.pm$/;
   }, $dir;
}
```

或者使用 CPAN 模块察看所有的安装模块：

>perl -MCPAN -e autobundle

CPAN 还可以通过交互界面使用：

>perl -MCPAN -e shell

cpan> autobundle

4.2 文件

文件句柄是一种独立的数据类型。它不同于标量。

Perl 自动为所有的 Perl 程序提供了三种标准的文件句柄，它们分别是 STDIN,STDOUT 与 STDERR。

名称	特性	缓存否
STDIN	缺省的输入文件句柄，通常与键盘连接。在使用 getc 函数进行读取操作时使用的就是这个文件句柄。	缓存
STDOUT	缺省的输出文件句柄，通常与显示屏连接。在使用 print 函数时，使用的就是这个文件句柄。	缓存
STDERR	缺省的错误输出文件句柄，STDERR 连接到显示屏上。	不缓存

4.2.1 IO::Handle 对象

这个类有两个构造函数。

构造函数	说明
new ()	创建一个新的 IO::Handle 对象。
new_from_fd (FD, MODE)	创建一个新的 IO::Handle 对象。需要两个参数，用于传给 fdopen 函数。如果 fdopen 失败，对象将被销毁。否则对象将返回给调用者。

内置函数方法列表如下：

方法	说明
$io->close	关闭与文件句柄联系的文件或管道，如果 IO 缓存成功清空而且关闭了文件描述符将返回真。例子如下： *open(OUTPUT, '|sort >foo') # 管道输出到 sort* *or die "Can't start sort: $!";* *#... # print 输出* *close OUTPUT #等待 sort 结束* *or warn $! ? "Error closing sort pipe: $!"* *: "Exit status $? from sort";* *open(INPUT, 'foo') #取得 sort 的结果* *or die "Can't open 'foo' for input: $!";*
$io->eof	如果文件达到末尾处或文件句柄没有打开，则返回 1。 在一个 while (<>) 循环中，可以用 eof 或 eof(ARGV)来监测每一个文件的结束，eof() 仅能监测最后一个文件的结束。 例子如下： *#重置每一个输入文件的行号* *while (<>) {* *next if /^\s*#/; # 跳过注释* *print "$.\t$_";* *} continue {* *close ARGV if eof; # 不是 eof()!* *}* *#在最后一个文件的最后一行插入短横* *while (<>) {* *if (eof()) { # 检查当前文件的结束* *print "----------\n";* *close(ARGV);* *}* *print;* *}*

方法	说明
$io->fileno	返回文件句柄的文件描述或 undefined 如果文件句柄没有打开。 但是联结到内存对象的文件句柄可能返回 undefined 即使它已经打开。 能使用这个函数发现两个文件句柄是否指向同一个底层描述符： *if (fileno(THIS) == fileno(THAT)) {* *print "THIS and THAT are dups\n";* *}*
$io->format_write ([FORMAT_NAME])	声明 write 函数使用的 format。例如： *format Something =* *Test: @<<<<<<<< @||||| @>>>>>* *$str, $%, '$' . int($num)* . *$str = "widget";* *$num = $cost/$quantity;* *$~ = 'Something';* *write;*
$io->getc	从文件句柄相关的输入文件中返回下一个字符，如果已到文件末尾或出错，返回 undefined。如果忽略 FILEHANDLE，将从 STDIN 读取。但是这样做并不是特别高效。特别的，当从 STDIN 读取时，必须等待用户输入回车。例子如下： $key = getc(STDIN);
$io->read (BUF, LEN, [OFFSET])	尝试从指定的 FILEHANDLE 读 LEN 长度的字符到标量变量 BUF。返回实际读入的字符个数。当到达文件结束处时返回 0 ，或者 undef 如果出错。 BUF 将增长或缩小到实际读入的长度。如果需要填充长度，新字节内容会是 0。OFFSET 用来把读入的数据放到标量的其他地方，而不是开始。这个函数实际是调用 Perl 的或系统的 fread()。 注意：读入方式依赖于文件句柄的状态，按 8 位的字节或字符的方式读入。缺省情况下，所有的文件句柄按字节操作。但是如果文件句柄以:utf8 在 I/O 层打开，I/O 会按字符操作，而不是字节。

方法	说明
$io->print(ARGS) 或 print FILEHANDLE LIST	打印一个字符串或字符串的列表。如果成功返回 true。 FILEHANDLE 可能是一个标量变量名，在此时变量名包含 filehandle 的名字或者 filehandle 的引用。但是这样就引入了一级间接。注意：如果 FILEHANDLE 是一个变量而下一个 token 是一个项目，它会被误解成一个操作符除非你在前面放一个+或用圆括号把参数括起来。如果忽略 FILEHANDLE ，打印到缺省的标准输出(或最后选择的输出通道)。如果忽略 LIST，打印$_到当前选择的输出通道。要设置缺省输出可使用 select 操作。在每个 LIST 项目间和整个输出结束时将打印$\的当前值(如果有的话)。 因为 print 接受一个列表值，在 LIST 中的任何参数都在列表上下文中。 注意，如果把 FILEHANDLES 存在数组或其它的表达式中，就要使用 block 返回它的值，例如： print { $files[$i] } "stuff\n"; print { $OK ? STDOUT : STDERR } "stuff\n";
$io->printf (FMT, [ARGS])	等价于 print FILEHANDLE sprintf(FORMAT, LIST)，除了不附加$\ (输出记录分隔符)。列表中的第一个参数解释成 printf 格式。如果 use locale 起作用，用作小数点的字符受 LC_NUMERIC locale 的影响。 当可以使用简单的 print 时，不要使用 printf。Print 效率更高而且错误更少。
$io->stat	用于状态检测，具体用法参见下文。
$io->sysread (BUF, LEN, [OFFSET])	试图从指定的 FILEHANDLE 读取 LEN 长度的字符到变量 SCALAR 。读取使用系统调用 read(2)。它忽略缓存的 IO，因此把这个和其它类型的 reads, print, write, seek, tell, 或 eof 可能导致混淆，因为 stdio 通常使用缓存数据。 返回实际读入的字符数量，在文件结束时返回 0，如果出现错误返回 undef。SCALAR 会增加或减少到实际读入的长度。 注意：字符的读入依赖于文件句柄的状态，可能是读入一个字节或一个字符。缺省所有的文件句柄按字节操作，但是例如如果文件句柄已经使用:utf8 I/O 层打开，I/O 会按字符操作，而不是字节。 可以声明 OFFSET 来把读入的数据放置在字符串的某个位置而不是开始。一个负值的 OFFSET 声明从字符串的结尾处倒数。一个大于标量长度的 OFFSET 导致字符串在读入之前附加要求长度的"\0"字节。 没有 syseof() 函数，因为 eof() 在设备文件上工作的不是很好。可使用 sysread() 然后检查返回值，如果是 0 就表示已经读完了。

方法	说明
$io->syswrite (BUF, [LEN, [OFFSET]])	试图写入 LENGTH 个字符长度的数据从标量 BUF 到指定的 FILEHANDLE，使用系统调用 write(2)。如果 LENGTH 没有声明，写入整个 BUF。返回实际写入的字符的个数，如果有错误则返回 undef。如果 LENGTH 大于 BUF 的偏移量后 （OFFSET）可得到的数据，只有尽可能多的数据写入。 它忽略缓存的 IO，因此把它和读入混合在一起 (除了 sysread()可以正常工作外)，print, write, seek, tell, 或 eof 可能导致混淆 ，因为 stdio 通常使用缓存的数据。 可以声明一个偏移量（OFFSET） 来从字符串的某个部分开始写入数据而不是开始。一个负值的 OFFSET 声明从字符串的结尾处倒数。当 BUF 是空时，唯一能用的 OFFSET 是 0。 注意：字符的写入依赖于文件句柄的状态，可能是写入一个字节或一个字符。缺省所有的文件句柄按字节操作，但是例如如果文件句柄已经使用:utf8 I/O 层打开，I/O 会按字符操作，而不是字节。
$io->truncate (LEN)	截短文件到指定的长度。如果在你的系统上 truncate 没有实现，将产生一个致命错误。如果成功，返回 true，否则返回 undefined。 如果 LENGTH 大于文件的实际长度，其结果不可知。

相关变量：

方法	变量
$io->autoflush ([BOOL])	$\|，$|变量设定为非 0 值时，不缓存输出。
$io->format_page_number([NUM])	$%
$io->format_lines_per_page([NUM])	$=
$io->format_lines_left([NUM])	$-
$io->format_name([STR])	$~
$io->format_top_name([STR])	$^
$io->input_line_number([NUM])	$.
$io->autoflush ([BOOL])	$|

IO::Handle->format_line_break_characters([STR])	$:
IO::Handle->format_formfeed([STR])	$^L
IO::Handle->output_field_separator([STR])	$,
IO::Handle->output_record_separator([STR])	$\
IO::Handle->input_record_separator([STR])	$/

方法	说明
$io->fdopen (FD, MODE)	Fdopen 像通常的 open 除了它的第一个参数不是易个文件名而是文件句柄名，一个 IO::Handle 对象或一个文件描述符。
$io->opened	如果对象当前是一个有效的文件描述符，返回真，否则返回 false。
$io->getline	这个方法和<$io>一样工作，只是可读性更好，能够安全地在数组上下文中调用时仍然返回一行。
$io->getlines	这个方法和<$io>一样工作，只是可读性更好。当在数组上下文中调用时，读取文件中剩下的所有行， 如果在标量上下文中调用它，它会调用 croak()。
$io->ungetc (ORD)	该方法把 ORD 定义的字节压栈到输入流。压栈的字节将在随后的读入中以压栈相反的顺序返回。
$io->write (BUF, LEN [, OFFSET])	这个 write 方法像 C 语言中的 write 函数，与 read 对应。
$io->error	返回 true 如果给定的句柄从打开以来，或自从最后一次调用以来有任何错误 clearerr 以来，如果句柄是无效的也返回 true。 仅当一个有效的句柄没有明显的错误时返回 false。
$io->clearerr	清除给定的句柄错误指示。返回-1 如果句柄是无效的，否则返回 0。

方法	说明
$io->sync	同步一个文件在内存中的状态和在物理介质上的状态。同步不会在 perlio api 层操作，但是在文件描述符上操作(类似 sysread，sysseek 和 systell)。这意味着不会同步在 perlio api 层存在的任何数据。要同步缓存在 perlio api 层的任何数据必须使用 flush 方法。Sync 并没有在所有的平台实现。成功时返回 0 而不是真。错误时返回 undef，句柄无效时返回 undef。
$io->flush	在 perlio api 层上清空缓存数据。将会清除在缓冲区中任何没有读入的数据，将把任何没有实际写入的数据写到底层的文件描述符。成功时返回 0 而不是真，错误时返回 undef。
$io->printflush (ARGS)	打开 autoflush，print ARGS 然后恢复 IO::Handle 对象的 autoflush 状态。返回从 print 返回的值。
$io->blocking ([BOOL])	是否使用阻塞式 IO。 该方法返回以前的设置值，如果不给出参数 BOOL 则返回当前的设置值。

4.2.2 IO::Seekable

没有构造函数，用于给其它基于 IO::Handle 的类提供继承。

方法	说明
$io->getpos	返回一个不透明的值代表 IO::File 的当前位置，如果得不到则返回 undef(例如一个像终端，管道或 socket 这样不可搜索的流)。如果 C 库函数中 fgetpos()函数可用，则使用它实现 getpos，否则 perl 使用 C 的 ftell()函数模拟 getpos。

$io->setpos	使用以前调用 getpos 返回的值给它来回到一个以前访问的点。成功时返回"0 but true"，失败时返回 undef。例如：
	
	my $fp=$file->getpos;
	
	print $file->setpos($fp);

$io->seek (POS, WHENCE)	搜索 IO::File 到位置 POS，相对于 WHENCE： WHENCE=0 (SEEK_SET) POS 是绝对位置(Seek 相对于文件的开始)。 WHENCE=1 (SEEK_CUR)

方法	说明
	POS 是一个当前位置的偏移量(Seek 相对于当前)。 WHENCE=2 (SEEK_END) POS 是一个从文件结束处的偏移量(Seek 相对于结束)。 如果不想在代码中使用数字 0，1 或 2，可以从 Fcntl 模块输入 SEEK_* 常量。 成功返回 1，否则返回 0。 例如，下例跳到文件的开始处： $io ->seek(0,0);
$io->sysseek (POS, WHENCE)	类似$io->seek，但是直接通过系统调用 lseek 设置 IO::File 的位置。因为会与大部分 perl IO 操作混淆，所以一般与 sysread 和 syswrite 一起使用。 成功时返回新的位置，失败时返回 undef。位置零返回字符串 "0 but true"。
$io->tell	返回 IO::File 的当前位置，如果出错则返回-1。

4.2.3 IO::File

通过 IO::File，我们可以采用面向对象的方式读写文件。一个简单的例子如下：

```
use strict;
use IO::File;

my $file=IO::File->new();
my $path='d:/test.sql';
$file->open($path, O_RDONLY) or die('unable to open "', $path, '": ', $!, "\n");
while (defined(my $line=$file->getline())) {
    print $line;
}
$file->close();
```

构造函数

构造函数	说明
new (FILENAME [,MODE [,PERMS]])	创建一个 IO::File。如果它接收到任何参数，这些参数就会传给 open 方法；如果 open 失败，该对象就会销毁。否则，返回对象给调用者。
new_tmpfile	创建一个临时文件并打开用于读/写。临时文件是匿名的(即：在创建以后，它被 unlinked ，但是保持打开)。如果不能创建或打开临时文件，IO::File 对象将被销毁。否则，返回对象给调用者。

附加方法

方法	说明
Open (FILENAME [,MODE [,PERMS]])	Open 函数接受一个，两个或三个参数。如果是一个参数，它就是内在的 open 函数的前端。如果是两个或三个参数，第一个参数是文件名，第二个是打开模式，可选的，跟着一个文件权限值。 如果 IO::File::open 接受一个 Perl 模式字符串(``>", ``+<", 等)或一个 ANSI C fopen()模式字符串(``w", ``r+"，等)，它使用基本的 Perl open 操作符(but protects any special characters)。 如果传给 IO::File::open 一个数字模式，它传递该模式和可选的权限值给 Perl sysopen 操作。权限值缺省是 0666。 为了方便，IO::File 从 Fcntl 模块输出 O_XXX 常量，只要该模块可得到。

Perl 模式字符串列表

符号	方式	说明
<	读	open(INFO, "< datafile") || die("can't open datafile: $!");
>	写	open(RESULTS,"> runstats") || die("can't open runstats: $!");
>>	追加	open(LOG, ">> logfile ") || die("can't open logfile: $!");
+<	读-更新	open(WTMP, "+< /usr/adm/wtmp") || die "can't open /usr/adm/wtmp: $!";

符号	方式	说明
+>	写-更新	open(SCREEN, "+> /tmp/lkscreen")|| die "can't open /tmp/lkscreen: $!";
+>>	追加-更新	open(LOGFILE, "+>> /tmp/applog" || die "can't open /tmp/applog: $!";

sysopen HANDLE, PATH, FLAGS, [MASK]

这个函数接受的标记（FLAGS）说明如下：

标记	说明
O_RDONLY	只读。
O_WRONLY	只写。（windows 上不支持）
O_RDWR	读写。
O_CREAT	当文件不存在时创建文件。
O_EXCL	如果文件存在时返回失败。
O_APPEND	添加到文件。
O_TRUNC	截断文件。
O_NONBLOCK	非阻塞方式。

通过模式组合，我们可以精细的控制文件打开方式。例如，不管一个文件是否存在，我们都是打开一个文件，并重新写入内容：

```
$file->open($path, O_CREAT|O_TRUNC|O_RDWR);
```

在 Perl 5.8.0 中引进了一个新的 I/O 框架叫做``PerlIO"。这是 Perl 中的 I/O 的一个新的工具。对大多数部分而言，还是像以前那样工作。但是 PerlIO 引进了一些新的特性，像可以把 I/O 看成``层"。一个 I/O 层可以除了传输数据外，还做数据转换。这样的转换包括压缩和解压缩，加密和解密，还有在各种字符编码之间转换等等。

4.2.4 文件测试

选项	说明

选项	说明
-r	对于有效用户，文件可读。
-w	对于有效用户，文件可写。
-x	对于有效用户，文件可执行。
-o	对于有效用户，拥有该文件。
-R	对于真实用户，文件可读。
-W	对于真实用户，文件可写。
-X	对于真实用户，文件可执行。
-O	对于真实用户，拥有该文件。
-e	该文件存在。
-z	该文件为空。
-s	该文件非空(返回字节大小长度)。
-f	文件是普通文件。
-d	文件是目录。
-l	文件是符号链接。
-p	文件是命名管道(FIFO), 或者文件句柄是管道。
-S	文件是套接字。
-b	文件是块设备。
-c	文件是字符设备。
-t	文件是可交互的。
-u	文件设置了 setuid 位。
-g	文件设置了 setgid 位。
-k	文件设置了 sticky 位。

选项	说明
-T	文件是 ASCII 码文本文件。
-B	文件是二进制文件（与-T 相反）。
-M	脚本开始时间减去文件修改时间，单位是天。
-A	文件的最后一次访问时间。
-C	在 Unix 返回 inode 修改时间，其它平台返回文件创建时间。

例如：

```
while (<>) {
        chomp;
        next unless -f $_;        # 不是普通文件就忽略
        #...
}
```

取得文件状态也可以通过 stat 函数实现，语法如下：

stat FILEHANDLE

stat EXPR

stat

位置	代号	说明
0	dev	文件系统的设备号
1	ino	inode 号
2	mode	文件模式(类型和权限)。
3	nlink	文件的链接数。
4	uid	文件所有者的数字用户 ID。
5	gid	文件所有者的数字组 ID。
6	rdev	设备标识符(仅对于特殊文件)。
7	size	以字节位单位的文件大小。

位置	代号	说明
8	atime	以秒位单位的最后访问时间。
9	mtime	以秒位单位的最后修改时间。
10	ctime	以秒位单位的 inode 最后修改时间。
11	blksize	文件系统 I/O 建议的块大小。
12	blocks	实际分配给文件的块数。

例如，取得文件修改时间：

```
use POSIX qw(strftime);

my $opt_f= 'c:/ppes_audit.log';
$file_modify_time = strftime('%Y-%m-%d', localtime( (stat($opt_f))[9] ));
print "$file_modify_time";
```

4.2.5 glob

提供通过文件名查找文件的功能。File::Glob::bsd_glob() 函数提供了该功能，它的原型如下：

```
@filenames = bsd_glob(pattern, [flags]);
```

编程举例如下：

```
use File::Glob ':glob';
@list = bsd_glob('c:\*.pl');

foreach (@list) {
    print "This element is $_\n";
}
```

如下是对 bsd_glob 中使用的 flag 标志说明：

标志	功能
GLOB_ERR	如果遇到一个不能打开或读取的路径，强迫 bsd_glob()返回一个错误。通常情况， bsd_glob() 会继续查找匹配项。

标志	功能
GLOB_LIMIT	限制匹配使用的内存量。让 bsd_glob() 返回错误(GLOB_NOSPACE) 当模式大小扩展到大于系统常量 ARG_MAX 时。ARG_MAX 通常定义在 limits.h 中。如果系统中没有定义这个常量， bsd_glob() 按先后顺序使用 sysconf(_SC_ARG_MAX)或_POSIX_ARG_MAX 的值。
GLOB_MARK	匹配模式的每个路径后附加一个斜线。
GLOB_NOCASE	缺省情况，在 unix 下文件名是大小写敏感的，windows 是大小写不敏感的。这个标记让 bsd_glob() 执行大小写不敏感的匹配。
GLOB_NOCHECK	如果没有文件与此模式匹配，那么 bsd_glob()返回一个仅包括模式本身的列表。如果设置了 GLOB_QUOTE，它的影响就会出现在模式的返回值中。
GLOB_NOSORT	缺省情况，路径名按照 ASCII 码升序排列。这个标记防止排序，因此加速了 bsd_glob()的执行速度。
GLOB_BRACE	像 csh 一样预处理{}中的字符串。由于历史的原因模式'{}'被留下来不处理 。例如： glob ("{foo/{,bar,biz},baz}", GLOB_BRACE, NULL, &result) 等价于如下四句： glob ("foo/", GLOB_BRACE, NULL, &result) glob ("foo/bar", GLOB_BRACE|GLOB_APPEND, NULL, &result) glob ("foo/biz", GLOB_BRACE|GLOB_APPEND, NULL, &result) glob ("baz", GLOB_BRACE|GLOB_APPEND, NULL, &result)
GLOB_QUOTE	使用反斜线('\')作为转义符。
GLOB_TILDE	扩展以'~'开始的模式作为用户的 home 路径。
GLOB_CSH	LOB_CSH 作为 GLOB_BRACE | GLOB_NOMAGIC | GLOB_QUOTE | GLOB_TILDE | GLOB_ALPHASORT 的缩略语。
GLOB_ALPHASORT	如果没有 GLOB_NOSORT，排序文件名按字母顺序(不区分大小写)而不是按 ASCII 顺序。

标志	功能
GLOB_APPEND	未实现
GLOB_DOOFFS	未实现
GLOB_ALTDIRFUNC	未实现
GLOB_MAGCHAR flags	未实现

4.2.6 管道操作

输出命令管道的语法如下：

```
open(FILEHANDLE,|COMMAND);
```

举例如下：

```
open(MYPIPE,"|wc -w");
print MYPIPE "apple pears peaches";
close(MYPIPE);
```

输入过滤器的语法如下：

```
open(FILEHANDLE, COMMAND|);
```

举例如下：

```
open(INPIPE,"date/T |");
$today = <INPIPE>;
print $today;
close(INPIPE);
```

执行结果：

```
2003-05-18 星期日
```

4.3 目录

功能	方法

功能	方法
创建目录	*use File::Path;* *mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);*
删除目录	*use File::Path;* *rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);*
遍历目录	下例仅遍历当前目录： *my($file);* *my $root_dir="c:/temp";* *opendir(D, $root_dir);* *while ($file = readdir D) {* *next if ($file =~/^\.\.?$/);* *$file = $root_dir . "/" . $file;* *print "$file\n" if (-d $file && $file ne "." && $file ne "..");* *}*

采用递归的方法遍历目录：

```
my($root) = "C:\\temp";
DoDir($root);

sub DoDir {
   my($dir) = shift;
   my($file);
   opendir(DIR, $dir) || die "Unable to open $dir :$!";
   my(@files) = grep {!/^\.\.?$/ } readdir(DIR);
   closedir(DIR);
   foreach (@files) {
        if (-d ($file = "$dir\\$_")) {
                print "Found a directory: $file\n";
               DoDir($file);
      } else {
                print "File $file\n";
      }
   }
}
```

4.4 数据结构

4.4.1 Data::Dumper

Data::Dumper 让存储、传递、检索复杂的数据结构变得容易。当给定一个标量的列表或一个变量的引用，以 perl 语法的方式写出它们的内容。引用可以是对象。

它的通常用处是可以把结果打印出来或使用 eval 复制一个同样的数据结构。

缺省情况下，Data::Dumper 输出它的 Dumper 例程。

4.5 命令行

4.5.1 命令行约定

很多程序都有命令行参数来控制程序的运行或作为程序交互的简单接口。一般采用选项与值的思想。

常用的有两种约定：

单字符选项约定：选项是一个参数，用一个字符长表示，并且这个字符前加一个负号。例如：

>tar –xvf a.bak

在 windows 中”-”，也可能是”/”。

长选项约定：长选项名可能是描述性的单词，而不是单个字符。例如：

>program --option1=1

4.5.2 单字符选项约定处理

可以使用 Getopt::Std 进行简单的命令行处理。

getopt($options, \%list)	第一个参数代表可选项的列表，任何不存在于列表中的选项都是布尔标志。第二个参数代表一个哈希变量，参数可保存于其中。如果第二个参数不存在，则值保存在 $opt_* 中。任何选项的缺省值都是 1。

	例如：
	
	# 可设置 -o, -D & -I 选项。值保存在$opt_o, $opt_D, $opt_I 中。
	
	getopt('oDI');
	
	#值保存在%opts 中，其它同上。
	
	getopt('oDI', \%opts);

getopts($options, \%list)	第一个参数代表可选项的列表，任何不存在于列表中的选项都视为非法。非布尔选项需加后缀:。其余与 getopt 相同。 例如： # -o 和 -i 是布尔标记，-f 接受一个字符串参数 getopts('oif:');

例子代码：

```
sub usage
{
    print STDERR << "EOF";

    用法：$0 [-hvd] [-f file] -s path
     -h          ：显示帮助信息
     -v          ：详细输出
     -d          ：打印调试信息到 stderr
     -f file     ：文件名
     -s path     ：绝对路径

例子：  $0 -v -f include.staff
EOF
}

use Getopt::Std;                # 命令行选项处理
my $opt_string = 'hvdf:s:';
getopts( "$opt_string", \my %opt ) or usage() and exit;

usage() and exit if $opt{h};

$VERBOSE = 1 if $opt{v};
print STDERR "详细输出模式开。\n" if $VERBOSE;

$DEBUG     = 1 if $opt{d};
```

print STDERR "调试模式开。\n" if $DEBUG;

die "${opt{s}} 不是可执行文件" unless -x ${opt{s}};

4.5.3 长选项约定处理

可以使用 Getopt::Long 进行更复杂的命令行处理。Getopt::Long 模块执行与 Getopt::Std 一样的动作，但 Getopt::Long 除了可以处理单字符选项外，同时还支持长选项（GNU Long），还可以执行更好的解析、进行错误检查。模块函数列表如下：

函数	说明
Parser	构造函数
Configure	控制处理命令行参数的方法
GetOptions(\%list, $option1, $option2,…)	取得命令行参数到指定变量。

传递配置的方法有如下几种：

方法	举例
直接调用 Configure 方法	*configure(...configuration options...);*
通过面向对象方法	*use Getopt::Long;* *$p = new Getopt::Long::Parser;* *$p->configure(...configuration options...);*
通过构造函数	*$p = new Getopt::Long::Parser config => [...configuration options...];*
use	*use Getopt::Long qw(:config no_ignore_case bundling);*

配置选项

选项	说明	缺省状态	POSIXLY_CORRECT 缺省状态
default	所有的配置选项重置到默认状态。		
posix_default	这个选项重置所有的配置选项到 POSIXLY_CORRECT 缺省状态。通过设置 POSIXLY_CORRECT 环境变量		

选项	说明	缺省状态	POSIXLY_CORRECT 缺省状态
	可以达到同样的效果。		
auto_abbrev	只要缩写后的长选项名是唯一的，就允许缩写长选项名。	Y	N
getopt_compat	允许以 "+" 开始选项。缺省允许。	Y	N
gnu_compat	"gnu_compat" 控制是否允许 "--opt=" ，以及它应该做什么。没有"gnu_compat"，"--opt=" 给出一个错误。有了"gnu_compat"，"--opt="会给选项"opt"一个空值。		
gnu_getopt	这是同时设置四个选项 "gnu_compat"， "bundling"，"permute" 和 "no_getopt_compat" 的快捷方式。使用 "gnu_getopt", 命令行处理应该与 GNU getopt_long()完全兼容。		
require_order	是否允许命令行参数和选项混合在一起。 可参见 "permute"，它是 "require_order"的反义词。	N	Y
permute	是否允许命令行参数和选项混合在一起。缺省是允许，但是，设置环境变量 POSIXLY_CORRECT 将禁止 "permute"。注意 "permute" 是 "require_order"的反义词。 如果允许"permute"，就意味着： --foo arg1 --bar arg2 arg3 等价于 --foo --bar arg1 arg2 arg3 如果声明了一个参数回调例程，如果 GetOptions() 成功返回，@ARGV 会总是空。因为所有的选项都已经处理过了。唯一的例		

选项	说明	缺省状态	POSIXLY_CORRECT 缺省状态
	外是当使用了"--" 时： --foo arg1 --bar arg2 -- arg3 这会为 arg1 和 arg2 调用回调例程，然后结束 GetOptions() 把"arg2"留在@ARGV 数组中。 如果打开"require_order"，当碰到第一个非选项时选项处理结束。 --foo arg1 --bar arg2 arg3 等价于： --foo -- arg1 --bar arg2 arg3 如果也打开了"pass_through"，会在第一个不认识的选项或非选项处结束选项处理。		
bundling_override	如果打开"bundling_override"，就会允许绑定因为伴随着"bundling"，但是长选项名覆盖选项绑定。 注意：关闭"bundling_override"也会关闭"bundling"。 注意：使用选项绑定很容易导致不想要的结果出现，特别是当混合长选项和绑定时。	N	
bundling	打开此选项允许单个字符选项绑在一起。为了区分绑定与长选项名，长选项必须使用"--"引进，而绑定使用"-"。 举例，如有选项"a"，"l"和"all"，而且打开了auto_abbrev，可能的参数和选项设置是： -a,	N	

选项	说明	缺省状态	POSIXLY_CORRECT缺省状态
	--a a -l, --l l -al, -la, -ala, -all, ... a, l --al, --all all 要注意的是"--a"设置选项"a" (因为自动完成)，而不是"all"。 注意：关闭"bundling"的同时也关闭了"bundling_override"。		
ignore_case	如果打开，当匹配长选项名时忽略大小写。 然而，如果绑定是允许的，单个字符选项将看成是大小写敏感的。 使用"ignore_case"，仅仅是大小写不同的选项声明会被认为是重复的，例如"foo"和 "Foo"。 注意：关闭"ignore_case"也会关闭"ignore_case_always"。	Y	Y
ignore_case_always	当绑定其作用时，仍然忽略单个字符选项的大小写。 注意：关闭"ignore_case_always"也关闭"ignore_case"。	N	
pass_through	不知道或有岐义的选项，或者提	N	

选项	说明	缺省状态	POSIXLY_CORRECT 缺省状态
	供了一个无效的选项值被略过仍然在@ARGV 而不是标志成错误。这就可能写包装脚本仅处理部分用户提供的命令行数，而传递剩下的选项给其他的程序。 如果允许"require_order"，选项处理会结束在第一个不认识的选项或非选项处。然而如果允许"permute"，结果就变得模糊了。		
prefix	开始选项的字符串。如果常量字符串仍不够，可使用"prefix_pattern"。		
prefix_pattern	一个 Perl pattern 标识选项先导符字符串。	"(--|-|\+)"	"(--|-)"
debug	允许调试输出。	N	

如果需要显示帮助信息，可以这样：

```
use Getopt::Long;
GetOptions ("help|?");      # -help 和 -? 都会设置$opt_help 变量
```

4.6 配置

4.6.1 AppConfig

程序的首要需求之一就是其可配置性。基于文件的配置和命令行的选项可以满足可配置性的基本需要。在 Perl 中 Getopt::Std 和 Getopt::Long 可以用于命令行选项，而 AppConfig 可以解决文件配置的问题。

AppConfig 支持的特性有：

- AppConfig 能够忽略在配置文件中的空行和注释。
- 什么是重要的以及什么能在错误行忽略 (像拼错关键字)：你能够设置 AppConfig 的敏感度以忽略不好的设置或终止程序。关键字也能设置同名，以便其他可能的拼写(例如国际化设置)。
- 当需要嵌套的数据结构时，可以使用 XML::Simple 这样的模块。AppConfig 没有直接提

供解析嵌套的数据结构的支持。

- 多个配置文件：AppConfig 将能够处理多个配置文件，按顺序加载设置。你能够辅助 AppConfig 重置数组和 hash 以便 values inserted at the bottom of the stack don't have to come out at the top.

- 变量缺省：AppConfig 提供可变的缺省值。"-variable"语法重置一个在配置文件中的变量到它的缺省状态。

- 控制命令行选项和用文件配置集成它们：AppConfig 为 Getopt::Std 和 Getopt::Long 命令行选项解析提供支持。解析能够在文件读入前或以后进行。

- 教会用户另外一种配置文件格式：AppConfig 使用一个标准的，灵活的格式。对于标量"KEYWORD value" 和"KEYWORD=value"都是可接受的。因为数组向上增长，"ARR=1" 接着"ARR=2" 将会产生一个包括元素 1 和 2 的数组。也可以声明 boolean 选项成"bool"，"nobool"，"!bool"，"bool on"，"bool off"，"bool yes" (不要用 "bool no")，"bool=1"，"bool=0"。Hash 选项声明成"KEYWORD PARAMETER = value"，这里相当于 $h{ PARAMETER }= ‘value’。

先看一个 AppConfig 的简单使用。能够从命令行设置其中的变量。从命令行设置时，对于标量，布尔量和数组使用"-varname value"，对于 hash 用"-varname key=value"。文件 config.pl 的内容如下：

```
#!/usr/bin/perl -w
# 此程序只能在 Perl 5.005 或以上版本运行

use AppConfig qw/:argcount/;
use strict;                    # 良好风格的程序
use Data::Dumper;              # 用于哈西和数组引用

$| = 1;                        # 立即刷新全部的输出

my $config = AppConfig->new();

# 定义我们使用的全部变量，需要时就赋给缺省值。
$config->define(
                    'DEBUG' => { ARGCOUNT => ARGCOUNT_NONE, DEFAULT => 0 },
                    'NAME'  => { ARGCOUNT => ARGCOUNT_ONE,  DEFAULT =>
'Ishmael' },
                    'HOSTS' => { ARGCOUNT => ARGCOUNT_LIST },
                    'PHONE' => { ARGCOUNT => ARGCOUNT_HASH },
                  );

# 在启动时，读取多个配置文件。
foreach my $file ('example.conf', 'example2.conf')
```

```
{
 # 当文件存在时，就处理文件。
 $config->file($file) if -f $file;
}

# 注意 args() 也可以在处理文件之前调用，
#在这时，要加载的文件作为命令行选项(例如"-f")传入。

$config->args();                #处理命令行选项

my %varlist = $config->varlist('.*');
foreach my $varname (keys %varlist)
{
 print "Variable name $varname, value = ", Dumper($config->get($varname)), "\n";
}

# 在 AppConfig 对象中，变量名称是自动加载的方法。
print 'Accessing variables directly as a method is easy too: name = ',
 $config->NAME, "\n";
```

如下是一个配置文件的例子：

```
# 忽略空行

# 设置一个布尔值

debug

# 设置一个标量

name=E.T.

# 设置一个数组

hosts = dbhost

hosts = backuphost

# 重置 hosts 数组

-hosts

# 增加一个新值到 hosts

hosts = firewall
```

hosts = farewell

设置一个哈西

phone joe = 222-333-4444

phone marge = 555-666-7777

下面介绍 AppConfig 高级用法。AppConfig 能够在几个级别做扩展，依赖于 EXPAND 设置。

```
# 全局性的扩展所有的变量
my $config = AppConfig->new({ GLOBAL => { EXPAND => EXPAND_ALL } });
# 把 HOME_DIR 扩展成 UID，这样"~username"的作用就会像在 shell 中一样了。
$config->define('HOME_DIR => { ARGCOUNT => ARGCOUNT_ONE, EXPAND
=> EXPAND_UID });
```

INI 风格的部分是 AppConfig 另外一个有用的特性。在一个配置文件使用[section]的等价方法是，在所有的关键字前面加上一个节名和一个下划线'_'。 例如：

[file]

location = /tmp

type = txt

name = accounts.txt

[database]

host = wyrm

user = slayer

password = amethyst

等价于：

file_location = /tmp

file_type = txt

file_name = accounts.txt

database_host = wyrm

```
database_user = slayer

database_password = amethyst
```

能够用 varlist()函数检查一个 AppConfig 配置对象。如下的代码打印出 AppConfig 对象的每一个变量的内容。 注意，varlist()函数接受一个正则表达式作参数(不能使用空字符串)。

```
use Data::Dumper;   # 用于哈西和数组引用
my %varlist = $config->varlist('.*');
foreach my $varname (keys %varlist)
{
  print "Variable name $varname, value = ", Dumper $config->get($varname), "\n";
}
```

如下是一个 AppConfig 中的 Getopt::Long 接口，通过它可以取得 Getopt::Long 模块的全部的功能。如下代码定义 Getopt::Long 的变量参数，调用它从命令行解析参数。无效的值会产生错误。

```
$config->define("help|h|!"); # 定义一个布尔值
$config->define("code|c|=i"); # 定义一个标量整型值
$config->define("list|l|=f@"); # 定义一个浮点数的数组
$config->define("uids|u|=f%"); # 定义一个浮点数的哈西表
$config->getopt();   # 不使用 args()，而是使用 Getopt::Long 选项
```

在 AppConfig 中也有变量校验。 这意味着变量可以拒绝恶意的设置值，或者无意义的指向一个正则表达式(或者一段代码)。

```
# 仅仅当它是"joe"时，用户名校验成功。
# 当包含"joe"或"joE"时，密码校验成功。
$config->define(
    'USERNAME' => { ARGCOUNT => ARGCOUNT_ONE,
                                        VALIDATE => sub # 例程校验
                        {
                        my $varname = shift @_;
                        my $value = shift @_;
                        print "$varname = $value\n";
                        return ($value eq "joe");
                        }
                    },
    'PASSWORD' => { ARGCOUNT => ARGCOUNT_ONE,
                                        VALIDATE => "jo[Ee]" # 正则表达式校验
                    }
                );
```

在 AppConfig 中有自动触发动作，每次一个变量的值改变以后，就会执行该动作。注意，一个 AppConfig 的引用传入例程，因此，一个改变可以触发其它的变量改变。

```
$config->define(
    'USERNAME' => { ARGCOUNT => ARGCOUNT_ONE,
                                    ACTION => sub # autoaction
                            {
                            my $config = shift @_;
                            my $varname = shift @_;
                            my $value = shift @_;
                            print "$varname = $value\n";
                            }
                    }
);
```

AppConfig 的限制：

AppConfig 不处理变量中包含的代码，因为其作者认为这是一个不好的特性。AppConfig 不提供自动计算变量，尽管校验和和一个变量关联的自动例程能够方便的完成这个任务。如果你想更直接的解决，可以选取该变量然后自己在上面运行 eval() (如下例所示)。

```
foreach my $varname ('username', 'password')
{
  $config->set($varname, eval $config->get($varname));
}
```

使用 AppConfig 和 Persistent::DBI 加载配置到数据库

这个代码例子使用 MySQL 数据库和对应的持久化模块。如果你使用其它的数据库(例如 Postgres 或 Oracle)，就要寻找其它的持久化模块。

在数据库中使用配置，首先要设计数据库模式。如下这个例子把布尔值，标量和数组，以及 hashe 存储在不同的表中。你也可以让一个表用于所有的数据类型，或者根据用途分成不同的表。

这里展示的数据库模式对大多数目的都足够了。它对键和值的长度的确有一些限制，但是这些很容易在代码中调整。然而，array 和 hash 元素 ID 创建的方法可能导致问题。这里没有完美的解决方法。

我们将要使用 AppConfig::State 中的_argcount() 方法。只要给出变量的名字，它告诉我们在处理哪种变量。

这里是代码例子，它使用了 MySQL 数据库和对应的持久化模块：persistent-config.pl。

```
#此程序只能在 Perl 5.005 或以上版本运行

#
#该程序使用的表可以用如下的 SQL 语句创建

# create table booleans (name varchar(80) NOT NULL, val int(1), PRIMARY KEY(name));
# create table scalars (name varchar(80) NOT NULL, val varchar(255), PRIMARY KEY(name));
# create table hashes (name varchar(80) NOT NULL, parameter varchar(40), var varchar(40), val
varchar(255), PRIMARY KEY(name));
# create table arrays (name varchar(80) NOT NULL, var varchar(40), i int(10), val varchar(255),
PRIMARY KEY(name));

use Persistent::MySQL;
use AppConfig qw/:expand :argcount/;
use English;
use strict;

$| = 1;

my $config = AppConfig->new();        # AppConfig 对象
# define all the variables we will use
$config->define(
                    'DEBUG'                 => { ARGCOUNT => ARGCOUNT_NONE,
                        DEFAULT => 0 },
                    'USERNAME'              => { ARGCOUNT => ARGCOUNT_ONE,
                        DEFAULT => 'nobody' },
                    'HOSTS'                 => { ARGCOUNT => ARGCOUNT_LIST },
                    'PROPERTIES'            => { ARGCOUNT => ARGCOUNT_HASH },
                );

$config->args();                #处理命令行选项

# 建立起一些值
$config->HOSTS()->[0] = 'backuphost';
$config->HOSTS()->[1] = 'dbhost';

$config->PROPERTIES()->{joe} = '222-333-4444';
$config->PROPERTIES()->{marge} = '555-666-7777';

# 使用 eval，防备抛出一个异常
eval
{
  # create the persistent objects (one table per data type)
  my $database = 'persistent';
```

```
my $host     = 'xyz';
my $user     = 'xyz';
my $password = 'xyz';

my $booleans = new Persistent::MySQL(
                        "DBI:mysql:database=$database;host=$host",
                        $user,
                        $password,
                        'booleans');
my $scalars = new Persistent::MySQL(
                       "DBI:mysql:database=$database;host=$host",
                       $user,
                       $password,
                       'scalars');
my $arrays = new Persistent::MySQL(
                      "DBI:mysql:database=$database;host=$host",
                      $user,
                      $password,
                      'arrays');
my $hashes = new Persistent::MySQL(
                      "DBI:mysql:database=$database;host=$host",
                      $user,
                      $password,
                      'hashes');

#这是主要对象标识符键，一个 80 个字符的 id
$booleans->add_attribute('name', 'ID', 'VarChar', undef, 80);
$scalars->add_attribute('name', 'ID', 'VarChar', undef, 80);
$arrays->add_attribute('name', 'ID', 'VarChar', undef, 80);
$hashes->add_attribute('name', 'ID', 'VarChar', undef, 80);

#这是一个 255 个长度的字符串的值(除了布尔值)
$booleans->add_attribute('val', 'Persistent', 'Number', undef, 1);
$scalars->add_attribute('val', 'Persistent', 'VarChar', undef, 255);
$arrays->add_attribute('val', 'Persistent', 'VarChar', undef, 255);
$hashes->add_attribute('val', 'Persistent', 'VarChar', undef, 255);

# 对于哈西表，总是需要一个变量名和键
# (40 chars. max each)
$hashes->add_attribute('var', 'Persistent', 'VarChar', undef, 40);
$hashes->add_attribute('parameter', 'Persistent', 'VarChar', undef, 40);

# 对于数组，总是需要一个索引和变量名
# (40 and 10 chars max, respectively)
```

```
$arrays->add_attribute('var', 'Persistent', 'VarChar', undef, 40);
$arrays->add_attribute('i', 'Persistent', 'Number', undef, 10);

# 清空所有的表——也可以在SQL 中实现，但是对于小表也可以这样。
foreach my $p ($booleans, $scalars, $arrays, $hashes)
{
 $p->restore_all();
 $p->delete while $p->restore_next();
}

# 取得AppConfig 变量的列表，并存储它们。
my %varlist = $config->varlist('.*');
foreach my $varname (keys %varlist)
{
 # 这里 $varname 是我们感兴趣的变量名称。
 $config->DEBUG && print "Storing variable $varname\n";

 if (ARGCOUNT_NONE == $config->_argcount($varname)) # 存储布尔值
 {
  $booleans->clear;
  $booleans->name($varname);
  #我们把值设置成1 或0，而不考虑在变量里存了什么。
  $booleans->val( (1 == $config->get($varname)) );
  $booleans->save();
 }
 elsif (ARGCOUNT_ONE == $config->_argcount($varname)) # 存储标量
 {
  $scalars->clear;
  $scalars->name($varname);
  $scalars->val( $config->get($varname) );
  $scalars->save();
 }
 elsif (ARGCOUNT_LIST == $config->_argcount($varname)) # 存储数组
 {
  #存储数组的每一个值到它自己的$arrays 实例。
  my $counter = 0;
  foreach my $value (@{$config->get($varname)})
  {
   $arrays->clear;
   $arrays->name($varname . $counter); #这个ID 可能更好
   $arrays->var($varname);
   $arrays->val($value);
   $arrays->i($counter);
   $arrays->save;
```

```
    $counter++;
   }
  }
  elsif (ARGCOUNT_HASH == $config->_argcount($varname)) #存储哈西表
  {
   my $hash = $config->get($varname);
   foreach my $key (keys %$hash)
   {
    $hashes->clear;
    $hashes->name($varname . $key); # 这个 ID 可能更好
    $hashes->var($varname);
    $hashes->val($hash->{$key});
    $hashes->parameter($key);
    $hashes->save;
   }
  }
  else
  {
   die "Impossible argcount($varname) = " . $config->_argcount($varname) . ", quitting";
  }
}

# 创建新的 AppConfig
my $config = AppConfig->new();       # AppConfig 对象
# 定义我们使用的所有变量。
# 注意，可以由数据库表自动实现，但是这样做更容易。
$config->define(
            'DEBUG'                 => { ARGCOUNT => ARGCOUNT_NONE,
                                         DEFAULT => 0 },
            'USERNAME'              => { ARGCOUNT => ARGCOUNT_ONE,
                                         DEFAULT => 'nobody' },
            'HOSTS'                 => { ARGCOUNT => ARGCOUNT_LIST },
            'PROPERTIES'            => { ARGCOUNT => ARGCOUNT_HASH },
           );


# 恢复布尔值
$booleans->restore_all();
while ($booleans->restore_next())
{
 $config->DEBUG && print "Restoring boolean ", $booleans->name, "\n";
 $config->set($booleans->name, $booleans->val);
}
```

```
 # 恢复标量
 $scalars->restore_all();
 while ($scalars->restore_next())
 {
  $config->DEBUG && print "Restoring scalar ", $scalars->name, "\n";
  $config->set($scalars->name, $scalars->val);
 }

 # 恢复数组
 $arrays->restore_all();
 while ($arrays->restore_next())
 {
  $config->DEBUG && print "Restoring array element ", $arrays->name, "\n";
  my $array = $config->get($arrays->var());
  $array->[$arrays->i()] = $arrays->val;
 }

 #恢复哈西表
 $hashes->restore_all();
 while ($hashes->restore_next())
 {
  $config->DEBUG && print "Restoring hash element ", $hashes->name, "\n";
  my $hash = $config->get($hashes->var());
  $hash->{$hashes->parameter()} = $hashes->val();
 }
};

print "An error occurred: $EVAL_ERROR\n" if $EVAL_ERROR;
```

4.7 XML

从根本上讲，XML 是一种描述型的标记语言。标记语言起源于 GML，后来发展成 SGML（标准通用标记语言）。SGML 的功能十分强大，同时也非常复杂，从二十世纪八十年代开始就已经广泛应用于出版等专业领域。HTML 就是对 SGML 进行了大量简化后的一个应用，而 XML 也是 SGML 的一个应用子集，只不过鉴于 HTML 的教训，XML 在减少复杂性的同时，保留了 SGML 最有活力的部分——可扩展能力。

一篇 XML 文档由标记和内容组成，看起来和 HTML 文档很相似。元素是 XML 最主要的标记，与 HTML 本质的不同是 XML 中元素没有预定义，而是由用户对自己文档中使用的元素进行定义，这就需要加入“文档类型声明”（DTD）。通过 DTD，一方面用户可以在文档中定义自己的标记，另一方面分析器也可以得到关于文档内容和结构方面的元信息。这样，XML 文档就具有了可扩展性、结构性和可验证性，同时，XML 文档也具备了存储结构化数据的能力。考虑到与 HTML 的兼容，DTD 不是 XML 文档必须的成份，具有 DTD 的 XML 文档称作“Valid”文档，否则是“Well - formed”文档。

然而，由于 DTD 采用的是与 XML 文档完全不同的语法，也就是说，需要同时有两套分析器来处理 DTD 和 XML 文档本身；另外，DTD 的语法也相对复杂和古怪，不大易用，所以，又产生了 XML Schema。其实，XML Schema 本身就是一个 XML 文件。不同的是，Schema 文件所描述的是引用它的 XML 文件中的元素和属性的具体类型。XML Schema 利用 Namespace 将文档中特殊的结点与 Schema 说明相联系，一个 XML 文件可以有多个对应的 Schema，但只能有一个对应的 DTD。XML Schema 内容模型是开放的，可以随意扩充，而 DTD 无法解析扩充的内容。DTD 只能把内容类型定义为一个字符串，而 XML Schema 允许把内容类型定义为整型、浮点型、布尔型或者许多其它简单数据类型。

解析 XML 有两种接口：DOM 和 SAX。

DOM 模型需要对整个 XML 文档进行扫描，然后解析生成一个对象树，XML 文档中的所有标签和属性都是用对象来表示，而不是一个孤立的文本。

SAX 是事件驱动的。就是说，它可以在处理的过程中生成不同的消息，并调用相应的函数进行处理。

关于 XML 标准本身请参见 http://www.w3.org/TR/REC-xml。

为了方便 xml 文档的编写，校验等，可以使用 xml 文档编辑器，例如 xmlspy，xmlwrite 等。

4.7.1 XML::Simple

结构复杂的配置文件一般都是用 XML 格式写成的，因为它能很方便的处理嵌套结构，例如下面这个 xml 文件：

```
<config logdir="/var/log/foo/" debugfile="/tmp/foo.debug">
  <server name="sahara" osname="solaris" osversion="2.6">
    <address>10.0.0.101</address>
    <address>10.0.1.101</address>
  </server>
  <server name="gobi" osname="irix" osversion="6.5">
    <address>10.0.0.102</address>
  </server>
```

```
  <server name="kalahari" osname="linux" osversion="2.0.34">
    <address>10.0.0.103</address>
    <address>10.0.1.103</address>
  </server>
</config>
```

简单的解析 xml 文件的脚本如下：

```
use XML::Simple;
my $config = XMLin();                                  #加载文件
print $config->{logdir};                               # 日志路径
print $config->{server}->{kalahari}->{address}->[1];   #在服务器'kalahari'上的第二个地址
                                                       # (name 是一个 key attribute)
$config->{debugfile}= "/dev/null";                     # 改变调试文件
XMLout();                                              # 输出更新的 XML 文件
```

当前这个类不支持简体中文编码(gb2312)，是因为其底层解析模块 expat 不支持简体中文编码（gb2312），修改此模块后就可以支持。相关的修改在 http://www.hunterpro.net/projects/xml/index.html 可以找到。

函数	说明
XMLin	解析 XML 格式化的数据，而且返回一个数据结构的引用。这个数据结构以更容易处理的形式，包含同样的信息。 XMLin() 接受一个可选的 XML 标识符跟着 0 个或多个'name => value' 选项对。The XML 标识符可以是如下任意一个： ● 一个文件名：如果文件名不包含路径部分。XMLin() 会寻找该文件在每一个搜索路径(参见下面的选项)。例如： $ref = XMLin('/etc/params.xml'); 注意，文件名'-'能用来从 STDIN 解析。 ● undef：如果没有标识符，XMLin()会检查脚本路径和每一个查找路径，找与脚本名称相同但是以扩展名是'.xml'的文件。注意：如果想声明选项，就必须提供'undef'的值，例如： $ref = XMLin(undef, forcearray => 1);

函数	说明
	● 一个 XML 的字符串：一个包含 XML 的字符串会直接解析，例如： $ref = XMLin('<opt username="bob" password="flurp" />'); ● 一个 IO::Handle 对象：一个 IO::Handle 对象会被读入直到碰到 EOF，而它的内容被解析，例如： $fh = new IO::File('/etc/params.xml'); $ref = XMLin($fh);
XMLout	接受一个数据结构(通常是一个 hash 引用) 返回一个该结构的 XML 编码 。如果使用 XMLin()的输入，将返回一个与原来相同的数据结构。 当把 hash 翻译成 XML，hash 键值以'-'开头的将被忽略。注意：如果不忽略，键值将看成元素或属性名，元素或属性名以'-'开始就不会是一个有效的 XML。

主要选项有：

选项	使用
keyattr	这个选项控制'array folding'特性，将把嵌套的元素从一个数组翻译到一个 hash。例如，这个 XML： <opt> <user login="grep" fullname="Gary R Epstein" /> <user login="stty" fullname="Simon T Tyson" /> </opt> 在缺省状态下转换成： { 'user' => [{ 'login' => 'grep', 'fullname' => 'Gary R Epstein' },

选项	使用

	{
	'login' => 'stty',
	'fullname' => 'Simon T Tyson'
	}
	]
	}
	如果使用选项'keyattr => ``login''' 声明'login'属性是一个键，同样的 XML 会解析成：
	{
	'user' => {
	'stty' => {
	'fullname' => 'Simon T Tyson'
	},
	'grep' => {
	'fullname' => 'Gary R Epstein'
	}
	}
	}
	如果有多于一个属性名，键属性名应该在数组中提供。XMLin()会以提供的顺序试图匹配属性名 。当把哈西折成数组时，XMLout()会使用提供的第一个属性名。
	注意： the keyattr 选项控制数组的折叠。缺省情况下，单个的嵌套元素会被卷进一个标量而不是一个数组因此不会被折叠。使用下面将要介绍的'forcearray'选项将强迫嵌套元素解析成数组，因此有可能折叠进哈西。
	'keyattr'的缺省值是['name', 'key', 'id']。设置这个选项成一个空的列表会禁止数组折叠特性。

选项	使用
searchpath	当 XML 是从文件读入而且没有声明文件路径时，这个属性允许你声明查找哪个路径。 如果第一个参数没有定义，那么缺省的查找路径仅包含脚本文件所在的路径， 主意：不查找当前路径('.')，除非它正好是包含脚本的路径。

forcearray	如果想强迫嵌套元素表示成数组，即使只有一个元素，就把这个选项设置成 1。例如，使用 forcearray 允许选项，这段 XML：
	
	<opt>
	
	<name>value</name>
	
	</opt>
	
	将解析成：
	
	{
	
	'name' => [
	
	'value'
	
	]
	
	}
	
	而不是这个（缺省的）：
	
	{
	
	'name' => 'value'
	
	}
	
	当数据结构可能回写成 XML 时，它特别有用，而缺省的行为把单个嵌套的元素卷成属性不是我们想要的结果。

4.7.2 XML::Parser::PerlSAX

XML::Parser::PerlSAX 包含在 libxml-perl 中。以下面的 xml 文件为例：

```
<?xml version="1.0"?>
```

```
<config logdir="/var/log/foo/" debugfile="/tmp/foo.debug">
<server name="sahara" osname="solaris" osversion="2.6">
<address>10.0.0.101</address>
<address>10.0.1.101</address>
</server>
<server name="gobi" osname="irix" osversion="6.5">
<address>10.0.0.102</address>
</server>
<server name="kalahari" osname="linux" osversion="2.0.34">
<address>10.0.0.103</address>
<address>10.0.1.103</address>
</server>
</config>
```

可以使用 sax 接口统计，比如说<address>出现的次数。

```
use XML::Parser::PerlSAX;

my $my_handler = MyHandler->new;
my $parser = XML::Parser::PerlSAX->new( Handler => $my_handler );
my $instance = 'c:/server.xml';
$parser->parse(Source => { SystemId => $instance });
print $my_handler->{'count'} ;

1;

 package MyHandler;

sub new {
        my ($type) = @_;
    my $self = bless{} , $type;
```

```
    $self->{'count'} = 0;
        return $self ;
}

sub start_element {
        my ($self, $element) = @_;
    $self->{'count'}= $self->{'count'}+1 if ($element->{Name} eq 'address')    ;

        print "Start element: $element->{Name}\n";
}

sub end_element {
        my ($self, $element) = @_;

        #~ print "End element: $element->{Name}\n";
}

1;
```

4.7.3 XML::UM

这个模块提供方法把 UTF-8 字符串转换成任何 XML::Encoding 支持的 XML 编码。因为 XML::Parser 把所有的字符串转换成 UTF-8，但却没有相对应的功能，所以这个模块处理 XML 很方便。

它从 XML::Encoding 使用的映射相关的.xml 文件创建映射。注意，XML::Encoding 使用的是在 perl 路径中的.enc 文件，而不是.xml 文件。.enc 是由.xml 文件创建的。因此，必须声明$ENCDIR。

当前的实现使用了 XML::Encoding 类解析.xml 文件并创建一个哈西映射 UTF-8 字符串(每个由多达 4 字节组成)到他们在指定编码中的等价字节序列。注意，大的映射是很耗内存的！这个实现的进一步改进方法是直接解析.enc 文件，或者在 XS (即 C 代码)中作转换。

使用例子如下：

```
use XML::UM;

# 设置路径到 XML::Encoding 自带的.xml 文件。
# 尾部的斜线是必需的!
$XML::UM::ENCDIR = '/home1/enno/perlModules/XML-Encoding-1.01/maps/';

# 创建 encoding 例程
my $encode = XML::UM::get_encode (
    Encoding => 'ISO-8859-2',
```

```
    EncodeUnmapped => \&XML::UM::encode_unmapped_dec);

# 把字符串从 UTF-8 转换成指定的编码。
my $encoded_str = $encode->($utf8_str);

# 为了垃圾回收，删除循环引用。
XML::UM::dispose_encoding ('ISO-8859-2');
```

相关的方法如下：

方法	说明
get_encode (Encoding => STRING, EncodeUnmapped => SUB)	它把调用传递给对象 $XML::UM::FACTORY，它缺省定义成 XML::UM::SlowMapperFactory 的实例。覆盖此变量插入你自己的 mapper factory。 XML::UM::SlowMapperFactory 创建一个 XML::UM::SlowMapper 的实例(并缓存它用于以后的使用)。 XML::UM::SlowMapper 读入.xml 编码文件并创建一个 hash 映射 UTF-8 字符成编码字符。 最后调用 XML::UM::SlowMapper 的 get_encode() 方法。XML::UM::SlowMapper 生成一个匿名过程使用 hash 转换多字节的 UTF-8 成合适的编码。 参数说明： * Encoding 字符编码名称，例如：'ISO-8859-2'。 * EncodeUnmapped (缺省：\&XML::UM::encode_unmapped_dec) 定义在映像文件中没有发现的 Unicode 字符(属于指定的编码) 如何打印。缺省把它们转换成 decimal 实体引用，像'{'。 使用 \&XML::UM::encode_unmapped_hex 用于 hexadecimal 常量，像'«'。
dispose_encoding ($encoding_name)	调用它释放 SlowMapper 为特定编码使用的内存。注意，为了释放大的转换 hash，用户不应有对 get_encode()生成的例程的引用。

4.8 时间

4.8.1 Date::Manip

Date::Manip 是 CPAN 中进行日期及时间编程功能最强大的模块。提供了日期解析、日期比较、确定时间间隔、日期解析等功能。但是由于它完全由 Perl 编写而成，也带来了程序运行效率的问题。

注意，在 windows 下安装的时候，因为该模块无法取得时区，所以要手工设置时区变量。例如，国内的时区一般是正 8 区，要修改 Manip.pm 文件中的$Cnf{"TZ"}变量如下：

$Cnf{"TZ"}=" +0800";

Date::Manip 中的基本类型如下：

类型	说明
DATE	代表日期和时间 (year, month, day, hour, minute, second and weeks when appropriate). 它不能处理秒以下的单位。 在 windows 下的时区支持还不完全。
DELTA	一段时间的长短。它并不包括开始时间或结束时间的信息。
RECURRENCE	recurrence 只是一个定义循环事件发生的概念。例如，如果一个事件发生每隔一个星期五或 4 小时，就能把它定义成一个循环。 用一个 recurrence 再加上开始时间和结束时间，你可以得到一个在此期间循环发生的日期的列表。
GRAIN	时间的粒度基本上指你想多精确的看待时间。例如，如果你想要比较两个日期，看他们以天为粒度上是否相等，那么它们就只需要在同一天发生就是相等了。如果在小时的粒度上，它们必须发生在同一个小时上。 注意：将来会增加对它的支持。
HOLIDAYS 和 EVENTS	一个命名的时间。节假日（HOLIDAYS）用于商业模式的计算。事件（EVENTS）允许日历和计划应用程序更容易设计。

所有的方法列表如下：

方法	说明
ParseDate	返回含有指定日期/时间表示形式的标量。 例如： *PaseDate(“today”);* *PaseDate(“05/29/2003”);*
ParseDateString	这个例程是供 ParseDate 调用的。但你也可以直接调用它。
UnixDate	可用来格式化输出字符串。 *print UnixDate("today",* *"It is now %T on %A the %E of %B, %Y.");*
Delta_Format	这个函数类似于 UnixDate 例程，只是它从一个差值取得信息。
DateCalc	可以加、减时间。例如，要取得两天后的时间： *$date = DateCalc($date,"+2 days");* *$date = DateCalc("today","+ 3hours 12minutes 6 seconds",\$err);*
ParseRecur	用于查找一个循环的事件发生的日期的列表。例如，要查找每个月的第二个星期二。 *@date = ParseRecur("0:1*2:2:0:0:0",$base,$start,$stop);*
Date_Cmp	用于比较两个日期，例如： *$date1 = ParseDate($string1);* *$date2 = ParseDate($string2);* *$flag = Date_Cmp($date1,$date2);* *if ($flag<0) {* *# date1 早* *} elsif ($flag==0) {* *# 两个日期是相同的* *} else {* *# date2 早* *}*
DateCalc ($d1,$d2 [,\$err] [,$mode]);	估算两日期之差。在商务模式下，可仅计算营业日。例如： *my $date = PaseDate(“today”);* *my $ tomorrow = DateCalc($date, “+3 days”, 2);* 计算上个月： *$date = DateCalc("today","- 1month");*
Date_SetTime	它接受一个日期 (或可以通过 ParseDateString 解析的任何字符串)而且设置日期的时间部分。例如，取得明天 7:30 的时间的一个方法是： *$date = ParseDate("tomorrow");* *$date = Date_SetTime($date,"7:30");*

方法	说明
Date_SetDateField($date,$field,$val[,$nocheck]);	它接受一个时间域，并把它设置成一个新值。$field 是如下任一字符串 ``y", ``m", ``d", ``h", ``mn", ``s" (大小写不敏感) ，而$val 是新值。 如果$nocheck 是一个非零值，就不校验数据。
Date_GetPrev	该函数接受一个日期(任何字符串都会由 ParseDateString 解析成日期) 并找出上一个时间点。
Date_GetNext	和 Date_GetPrev 类似。
Events_List	该函数返回一个事件的列表。事件定义在配置文件中。
Date_IsHoliday	判断是否节假日。如果输入日期不是一个节假日，就返回 undef。如果输入日期是假日就返回包含假日名称的字符串。如果是一个无名的假日就返回长度为 0 的字符串。

UnixDate 例程接受的格式符号说明如下：

年

%y year - 00 to 99

%Y year - 0001 to 9999

%G year - 0001 to 9999 (see below)

%L year - 0001 to 9999 (see below)

月，周

%m month of year - 01 to 12

%f month of year - " 1" to "12"

%b,%h month abbreviation - Jan to Dec

%B month name - January to December

%U week of year, Sunday

as first day of week - 01 to 53

%W week of year, Monday

as first day of week - 01 to 53

日

%j day of the year - 001 to 366

%d day of month - 01 to 31

%e day of month - " 1" to "31"

%v weekday abbreviation - " S"," M"," T"," W","Th"," F","Sa"

%a weekday abbreviation - Sun to Sat

%A weekday name - Sunday to Saturday

%w day of week - 1 (Monday) to 7 (Sunday)

%E day of month with suffix - 1st, 2nd, 3rd...

小时

%H hour - 00 to 23

%k hour - " 0" to "23"

%i hour - " 1" to "12"

%I hour - 01 to 12

%p AM or PM

分钟，秒，时区

%M minute - 00 to 59

%S second - 00 to 59

%s seconds from 1/1/1970 GMT- negative if before 1/1/1970

%o seconds from Jan 1, 1970

in the current time zone

%Z timezone - "EDT"

%z timezone as GMT offset - "+0100"

时间，日期

%c %a %b %e %H:%M:%S %Y - Fri Apr 28 17:23:15 1995

%C,%u %a %b %e %H:%M:%S %z %Y - Fri Apr 28 17:25:57 EDT 1995

%g %a, %d %b %Y %H:%M:%S %z - Fri, 28 Apr 1995 17:23:15 EDT

%D,%x %m/%d/%y - 04/28/95

%l date in ls(1) format

%b %e $H:$M - Apr 28 17:23 (if within 6 months)

%b %e %Y - Apr 28 1993 (otherwise)

%r %I:%M:%S %p - 05:39:55 PM

%R %H:%M - 17:40

%T,%X %H:%M:%S - 17:40:58

%V %m%d%H%M%y - 0428174095

%Q %Y%m%d - 19961025

%q %Y%m%d%H%M%S - 19961025174058

%P %Y%m%d%H%M%S - 1996102517:40:58

%F %A, %B %e, %Y - Sunday, January 1, 1996

%J %G-W%W-%w - 1997-W02-2

%K %Y-%j - 1997-045

其它格式

%n insert a newline character

%t insert a tab character

%% insert a `%' character

%+ insert a `+' character

Date::Manip 模块可以自定制，可以自定制的方面包括：

- 时区；
- 语言；
- US/非 US 时间格式 (mm/dd/ 或 dd/mm)；
- 商业小时；
- 假期。

4.8.2 HTTP::Date

主要特点是认识很多种类型的日期表示，可用于日期类型字符串的格式化转换。

```
use HTTP::Date qw(time2str str2time time2iso time2isoz);

my $stringGMT = time2str(time);     # 格式化成 GMT ASCII 时间。
my $time = str2time($stringGMT);    # 把 ASCII 日期转换成机器时间。

my $stringISO = time2iso($time);
print "$stringISO\n";
```

方法	说明
time2str([$time])	把机器时间(seconds since epoch)转换成字符串。如果不输入参数时，缺省使用当前时间。

方法	说明
str2time($str [, $zone])	把字符串转换成机器时间。如果不认识$str 的格式或时间超出可表示的范围将返回 undef。它认识的时间格式与parse_date()相同。 认识的时间格式举例如下： \"Wed, 09 Feb 1994 22:23:32 GMT\" \"Thu Feb 3 17:03:55 GMT 1994\" \"Thu Feb 3 00:00:00 1994\" \"Tuesday, 08-Feb-94 14:15:29 GMT\" \"Tuesday, 08-Feb-1994 14:15:29 GMT\" \"03/Feb/1994:17:03:55 -0700\" \"09 Feb 1994 22:23:32 GMT\" \"08-Feb-94 14:15:29 GMT\" \"08-Feb-1994 14:15:29 GMT\" \"1994-02-03 14:15:29 -0100\" \"1994-02-03 14:15:29\" \"1994-02-03\" \"1994-02-03T14:15:29\" 这里 \"T\" 可以是 t，T，或空格。 \"19940203T141529Z\" 这里 \"T\" 可以是 t，T，或空格。 \"19940203\" \"08-Feb-94\" \"08-Feb-1994\" \"09 Feb 1994\" \"03/Feb/1994\" \"Feb 3 1994\" \"Feb 3 17:03\" \"11-15-96 03:52PM\"

方法	说明
time2iso([$time])	与 time2str()相同，但是返回``YYYY-MM-DD hh:mm:ss"格式代表本地时区时间的字符串。当省略输入参数时，该函数返回当前时间。 缺省没有导入。
time2isoz([$time])	与 time2str()相同，但是返回``YYYY-MM-DD hh:mm:ssZ"格式代表通用时间的字符串。 缺省没有导入。
parse_date($str)	这个函数会解析日期字符串，然后返回一个数字的列表，可能跟着一个 (可能没定义) 时区标识符($year, $month, $day, $hour, $min, $sec, $tz)。$year 是四位数的，$month 从 1 开始。

4.8.3 Date::Simple

Date::Simple 是一个处理日期简单有效的模块。但是它只能处理日期，不能处理时间。

```
use Date::Simple;

my $today = Date::Simple->new;

my $tomorrow = $today + 1;
my $yesterday = $today - 1;
my $diff = $tomorrow - $yesterday;

print "Tomorrow's date (in ISO 8601 format) is $tomorrow.\n";
print "yesterday's date (in ISO 8601 format) is $yesterday.\n";
print "diff = $diff \n"
```

运行结果输出如下：

```
Tomorrow's date (in ISO 8601 format) is 2003-06-11.

yesterday's date (in ISO 8601 format) is 2003-06-09.

diff = 2
```

方法	说明
new	构造函数。 *my $date = Date::Simple->new('1972-01-17');* *my $otherdate = Date::Simple->new(2000, 12, 25);*
next	返回代表明天的对象。 *my $tomorrow = $today->next;*
prev	返回代表昨天的对象。 *my $yesterday = $today->prev;*
year	返回日期对象的年。 *my $year = $date->year;*
month	返回日期对象的月。 *my $month = $date->month;*
day	返回日期对象的天。 *my $day = $date->day;*
format (arg)	返回代表日期的格式化字符串。如果不传递参数，将返回 ISO 8601 格式化的日期。 *my $change_date = $date->format("%d %b %y");* *my $iso_date1 = $date->format("%Y-%m-%d");* *my $iso_date2 = $date->format;* 格式化参数和 strftime 类似。因为这个方法底层是用 strftime 实现的。

格式化参数说明如下：

格式	说明
%a	本地缩写周名。
%A	本地周名全称。
%b	本地缩写月名。
%B	本地月名全称。
%c	本地日期和时间表示。
%C	世纪号(年除以 100 后取整) ， 范围[00-99]。
%d	一个月中的第几天，取值范围[01,31]。

格式	说明
%D	与%m/%d/%y 相同。
%e	一个月中的第几天，取值范围[1,31]，长度仍然是两位，不足两位前面以空格补齐。
%h	与%b 相同，本地缩写月名。
%H	小时数(24 小时) ，取值范围[00,23]。
%I	小时数(12 小时) ，取值范围[01,12] 。
%j	年中的天数，取值范围[001,366]。
%m	月份 [01,12]。
%M	分钟数[00,59]。
%n	新行。
%p	本地等价的上午 a.m.或下午 p.m。
%r	等价于 %I:%M:%S %p。
%R	等价于%H:%M。
%S	分钟[00,61]。
%t	制表符。
%T	相当于%H:%M:%S 的缩写。
%u	星期几 [1,7]，其中 1 代表星期一。
%U	年中的第几周(星期天作为一周的第一天) [00,53]。
%V	年中的第几周(星期一作为一周的第一天)，[01,53]。如果包含 1 月 1 日的周有 4 天或以上在新的一年，就把它看成第一周，否则它就是上一年的最后一周，而下一周是第一周。
%w	星期几 [0,6]。0 代表星期天。
%W	年中的第几周(星期一作为一周的第一天)， [00,53]。在新的一年中的第一个星期一以前的天认为是第 0 周。

格式	说明
%x	本地的日期表示。
%X	本地的时间表示。
%y	两位数表示的年 [00,99]。
%Y	四位数表示的年。
%Z	时区名或缩写，如果没有时区信息存在，就用长度为 0 的字符串代替。
%%	%是转义符是，%%代表一个%。

操作

操作符	说明
+= -=	可以使用 += 和 - = 操作符以天数增加或减少日期。
+ -	通过使用+ 和 – 操作符，可以按增加或减少天数构造新的日期偏移。
-	Date 类型减操作，返回之间的天数。
	可以使用算术比较操作，比较两个日期。
	能够插值一个 date 实例成一个字符串。以 ISO 8601 格式(例如：2000-01-17)。

4.9 日志

4.9.1 Log::LogLite

Log::LogLite 是一个简单的日志类。

方法	说明
new	构造函数
template	日志模版

方法	说明
write	写日志
default_message([MESSAGE])	用于操作 DEFAULT_MESSAGE 成员变量的方法。如果定义了 MESSAGE，DEFAULT_MESSAGE 将会返回这个值。
log_line_numbers([BOOLEAN])	如果这个变量设成真，<called_by> 字符串将会保存 调用例程的文件名和调用行。缺省是假。

相关包 Log::LogLite, IO::LockedFile。

举例如下：

```
use Log::LogLite;
my $LOG_DIRECTORY = "/where/ever/our/log/file/should/be";
my $ERROR_LOG_LEVEL = 6;

# 创建一个新的 Log::LogLite 对象
my $log = new Log::LogLite($LOG_DIRECTORY."/error.log", $ERROR_LOG_LEVEL);
$log->template('[<date>] <default_message><message>');
# 出错了
$log->write("Could not open the file ".$file_name.": $!", 4);
```

4.9.2 Log::Log4perl

Log4perl 是 Java 中的日志管理系统 Log4J 的 Perl 移植版本。Log::Log4perl 提供了一个强大的 logging 接口。简单例子如下：

```
use Log::Log4perl qw(get_logger);

my $conf = q(
        log4perl.category.ETL              = WARN, Logfile
        log4perl.appender.Logfile          = Log::Log4perl::Appender::File
        log4perl.appender.Logfile.filename = test.log
        log4perl.appender.Logfile.layout = \
            Log::Log4perl::Layout::PatternLayout
        log4perl.appender.Logfile.layout.ConversionPattern = [%H|%F{1}|%d{yyyy-M-d
HH:mm:ss.SSS}]%n %m %n
);

Log::Log4perl::init(\$conf);

my $logger = get_logger("ETL");
```

$logger->error("Blah");

它会记录如下日志：

[CSC-GY006|sample.pl|2003-6-10 22:10:56.433]

Blah

添加到日志文件 "test.log"。如果该文件不存在 Log4perl 将会创建它。

格式	说明
%c	记录日志事件的类型。
%C	调用者的包名（或类名）全称。
%d	当前日期的 yyyy/MM/dd hh:mm:ss 格式。
%F	记录事件发生的文件名。
%H	主机名。
%l	调用方法的全名，接着是调用者的源文件名和用括号括起来的行号。
%L	记录日志语句所在文件中的行号。
%m	将要记录的消息。
%M	发出日志请求的方法或函数。
%n	新行 (独立于操作系统)。
%p	日志事件的优先级。
%P	当前进程的 pid。
%r	从程序开始日志事件后的毫秒数。
%x	The elements of the NDC stack (see below)
%X{key}	The entry 'key' of the MDC (see below)
%%	百分符号 (%) 。

日期格式如同 java（http://java.sun.com/j2se/1.3/docs/api/java/text/SimpleDateFormat.html）

格式	说明
G	指定公元 (文本) ，例如 AD。
y	年(数字)，例如 1996。
M	月份(数字和文本类型) ，例如 July & 07。
d	月中的天数 (数字)。
h	上下午的小时数 (1~12) (数字)。
H	一天中的小时数 (0~23) (数字) 。
m	分钟 (数字) 。
s	秒数 (数字) 。
S	毫秒 (数字)。
E	星期几(文本)。
D	年中的天数(数字)。
F	月中的第几个星期(数字)。例如 2 (七月的第二个星期三)。
w	年中的星期数(数字)。
W	月中的星期数(数字)。
a	上/下午标记(文本)，例如 AM。
k	天中的小时数(1~24) (数字)。例如 24。
K	上午或下午的小时数(0~11) (数字)。例如 0。
z	时区(文本)。
'	文本的转义符(分隔符)。
''	单引号(字面上的) '。

例如，使用中国的 Locale：

Format Pattern	Result
"yyyy.MM.dd G 'at' hh:mm:ss z"	->> 1996.07.10 AD at 15:08:56 PDT
"EEE, MMM d, "yy"	->> Wed, July 10, '96
"h:mm a"	->> 12:08 PM
"hh 'o"clock' a, zzzz"	->> 12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	->> 0:00 PM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"	->> 1996.July.10 AD 12:08 PM

在 perl 中配置 log4perl。

初始化 logger 工作能在 Perl 中完成。这也是 Log::Log4perl::Config 在幕后所做的工作。在 Perl 层上，我们声明哪个 logger 与哪个 appender 工作，使用哪个 layout。

常见的情况是，需要把日志输出到多个 Appender 去。现在假设我们想把在 My::Category 类的 info 级别或以上的日志记录到 STDOUT 和一个日志文件，比如/tmp/my.log。在你的系统初始化阶段，只需要使用已有的 Log::Log4perl::Appender::File 和 Log::Log4perl::Appender::Screen 模块，并通过 Log::Log4perl::Appender 包裹器定义两个 appenders。代码如下：

```
use Log::Log4perl;
use Log::Log4perl::Layout;
use Log::Log4perl::Level;

# 定义一个日志种类
my $log = Log::Log4perl->get_logger("My::Category");

# 定义一个布局
my $layout = Log::Log4perl::Layout::PatternLayout->new("[%r] %F %L %m%n");

# 定义一个文件附加器
my $file_appender = Log::Log4perl::Appender->new(
                                "Log::Log4perl::Appender::File",
                                name          => "filelog",
                                filename   => "/tmp/my.log");
#定义一个标准输出附加器
my $stdout_appender =   Log::Log4perl::Appender->new(
                                "Log::Log4perl::Appender::Screen",
```

```
                                name            => "screenlog",
                                stderr          => 0);
# 让两个附加器使用同样的布局(当然也可以是不同的)
$stdout_appender->layout($layout);
$file_appender->layout($layout);
$log->add_appender($stdout_appender);
$log->add_appender($file_appender);
$log->level($INFO);
```

4.10 中文与 unicode

4.10.1 Unicode::Map

这个模块能实现本地编码和 2 字节的 Unicode UCS2 格式的字符串间的双向转换。所有的映射都是通过 2 字节的 UTF16 编码，而不是通过 1 字节的 UTF8 编码实现的。可以使用 Unicode::String 实现 UTF8 到 UTF16 的转换。

```
use Unicode::Map();

$Map = new Unicode::Map("GB2312");
$utf16 = $Map -> to_unicode ("测试 test");
$locale = $Map -> from_unicode ($utf16);
print "$locale";
```

方法	说明
new	返回一个 GB2312-80 编码的新的 Map 对象。 $Map = new Unicode::Map("GB2312-80")
from_unicode	从 utf16 编码字符串$src 创建一个本地字符集表示的字符串。 $dest = $Map -> from_unicode ($src)
to_unicode	从$src 创建一个 utf16 表示的字符串。 $dest = $Map -> to_unicode ($src)

4.10.2 Unicode::String

这个模块可以进行各种编码之间的转换，同时可以用它实现支持中文的字符函数，如 substr，index，和 chop。

```
use Unicode::String qw(utf8 latin1 utf16);
$u = utf8("The Unicode Standard is a fixed-width, uniform ");
$u .= utf8("encoding scheme for written characters and text");

 # 转换成各种外部的格式
 print $u->ucs4;          # 4 byte characters
 print $u->utf16;         # 2 byte characters + surrogates
 print $u->utf8;          # 1-4 byte characters
 print $u->utf7;          # 7-bit clean format
 print $u->latin1;        # lossy
 print $u->hex;           # a hexadecimal string

 # 如下的方式都可以当作构造函数
 $u->latin1("? v?re eller ? ikke v?re");
 $u = utf16("\0?\0 \0v\0?\0r\0e");

 # 字符串操作
 $u2 = $u->copy;
 $u->append($u2);
 $u->repeat(2);
 $u->chop;

 $u->length;
 $u->index($other);
 $u->index($other, $pos);

 $u->substr($offset);
 $u->substr($offset, $length);
 $u->substr($offset, $length, $substitute);

 # 重载运算符
 $u .= "more";
 $u = $u x 100;
 print "$u\n";

 # string <--> array of numbers
 @array = $u->unpack;
 $u->pack(@array);

 # 杂项
 $u->ord;
```

问题举例：

因为中文的汉字是用两个字节的，而普通的 asiic 码是一个字节的。用普通方式截取一

段字符串时，会出现乱码的问题。

譬如：

```
$a="lunar 我";
$b=substr(a ,0,6);
```

此时$b 的末尾就是乱码了。我现在怎么才能正确的拆分一个有汉字的字符串呢？答案是借助 Unicode::String 的 substr 函数可以实现，但是要让 Unicode::String 接受汉字，还必须借助 Unicode::Map 转换成 unicode。

```
use Unicode::String qw(utf8 utf16);
use Unicode::Map();

$Map = new Unicode::Map("GB2312");
$utf16 = $Map -> to_unicode ("lunar 我");
$us_16 = utf16($utf16);

$us_16 =$us_16->substr(0, 6);

$locale = $Map -> from_unicode ($us_16->utf16);
print "$locale";
```

但是随后介绍的 encoding 模块将展示一种更简单的方法。

4.10.3 encoding

利用 encoding 模块, 你可以轻易写出以字符为单位的程序：

```
# 启动 euc-cn 字符串解析;
use encoding 'euc-cn', STDIN => 'euc-cn', STDOUT => 'euc-cn';

print length("骆驼")."\n";                 #  2
print substr("骆驼",1,1)."\n";             #  驼
print index("谆谆教诲", "蛔唤")."\n";      # -1 (不包含此子字符串)
```

在最后一列例子里, ``谆" 的第二个字节与 ``谆" 的第一个字节结合成 EUC-CN 码的 ``蛔"; ``谆" 的第二个字节则与 ``教" 的第一个字节结合成 ``唤". 这解决了以前 EUC-CN 码比对处理上常见的问题。

4.10.4 Lingua::ZH::TaBE

这是一个能实现中文断句的模块。

4.11 解析文本

4.11.1 Parse::RecDescent

解析文本是我们通常需要碰到的问题。我们可能需要检查一段文本是否有效（如报表中的一段自定义 sql 语句或自定义脚本的有效性）。lex 和 yacc 就是这方面最著名的工具。但是在 Perl 中我们还可以使用更新、更好的工具 Parse::RecDescent 模块。

文法

编写一个 Parse::RecDescent 程序的一个重要基础是构造文法（定义规则）。

- 一个文法是一个 *4 元组* G = (N, E, P, S)。
- N 是一个*非终结符*的有限集合。
- E 是一个*终结符*的有限集合。
- P 是一个(N + E)*N(N + E)* x (N + E)*的有限子集。
- P 中的元素 (a, b)可写成 a -> b，叫做*产品*。
- S 是 N 中的一个元素叫做*开始符号*。

例如，如下是一个文法：

S -> 0 A 1

0 A -> 0 0 A 1

A -> e

非终结符是 A 和 S，终结符是 0 和 1。开始符号是 S。e 是一个特殊符号，表示空字符串。

Parse::RecDescent 中使用的概念是：

- 文法由规则的集合组成。
- 每一个规则以一个标识符开始 (对应一个非终结符), 接着一个冒号，接着 0 个或多个产品，产品间以 | 分隔。

- 如果一个规则有一个多于一个产品，它们就是可替换的。让几个规则以同样的标识符开始可以达到类似的效果。

- 每个产品由 0 个或多个项目组成。

- 项目是：

  o子规则：要匹配的另外一个规则。

  o令牌(Token)：一个要匹配的字符串或一个正则表达式。该匹配跳过任何空格。这对应所谓的终结符。

  o动作：一段要执行的 Perl 代码。

  o 指示器：给解析器的一个特殊的指令。

  o 注释：一个标准的 Perl 注释。

通俗的说，文法由一系列规则构成，它说明了什么是有效的 token 以及它出现的次序。Token 就是一个占位符的意思，也就是当成一个整体看待的字符串。如果学过 c 语言，也可以把文法看成宏语言。

macro_name : macro_body

宏的名称和定义之间是由冒号分开,而且宏定义能够是如下的任意组合:显式的字符串，正则表达式或在源文件中定义的其它的宏。

例如我们要计算一个简单的加法表达式：

+ 10 2

它是一个加号跟着两个数字。写成文法是这样的：

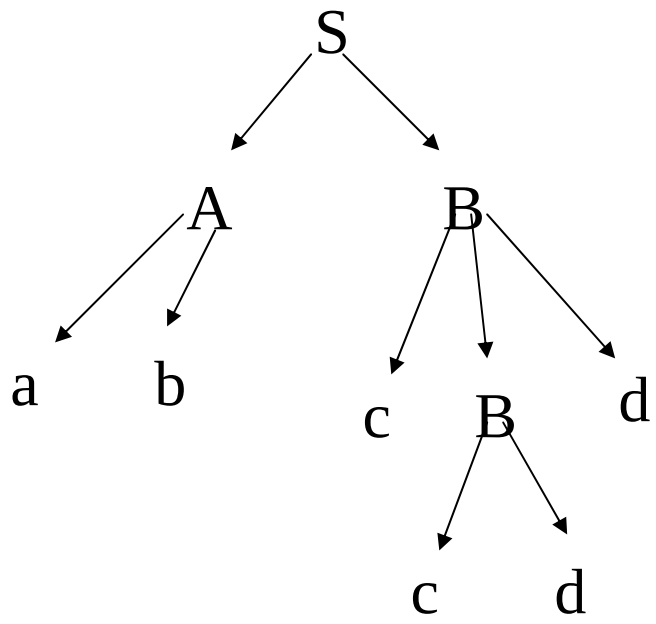
addition_expr: '+' number number

number: /-?\d+/

它匹配 + 11 –2 ， + 1 1 等。

可以借助语法树构造文法。语法树是这样的一个语法结构，它的结点由符号组成。根结点对应于识别符号。只有非终结符号对应的结点有子结点。并且，一个结点和它的子结点分别对应于文法中的一个规则的左部和右部。

一个语法树的例子如下图所示：



它的文法可以写成下面这样。

S::=AB

A::=aAb | ab

B::=cBd | cd

RecDescent 是 Recursive-Descent 的缩写，也就是递归下降的意思。它是一种自顶向下的语法分析技术。与之对应的是自底向上分析技术，如 LR(K)。

递归下降的实现思想：识别程序由一组过程组成。每个过程对应于一个非终结符号。每一个过程的功能是：选择正确的右部，扫描完相应的字。在右部中有非终结符号时，调用该终结符号对应的过程来完成。需要注意的是：递归下降对规则的要求是不能有左递归。

编程步骤

使用 Parse::RecDescent 一般分如下几步：

- 定义解析器使用的文法（grammar）。
- 创建解析器对象处理文法。
- 传递要解析的文本给解析器。

程序的基本结构如下：

```
use Parse::RecDescent;
my $grammar = q(
#  定义文法
);
my $parser = Parse::RecDescent->new($grammar);
my $text = q(
#  要解析的文本
);
# top_rule  是你的语法中的顶级规则的名字。
$parser->top_rule($text);
```

首先使用它搭建一个简单的测试框架：

```
use strict;
use Parse::RecDescent;
use Math::BigInt;

die "Usage $0 <file>\n" unless @ARGV;

my $grammar    = do {local $/; <ARGV>};

$::RD_HINT     = 1;
$::RD_WARN     = 1;
$::RD_ERRORS = 1;

my $parser     = Parse::RecDescent -> new ($grammar)
                    or die "Compilation error!\n";

for (my $i = 1, print "In [1] := "; $_ = <STDIN>;     ++ $i,   print "In [$i] := ")
{
    my $result = $parser -> start ($_);
    $result = "<<UNDEF>>"     unless defined $result;
    $result = "<<MATCHED>>" unless $result =~ /\S/;
    print "Out [$i] = $result\n";
}
print "\n";
```

它接受一个文件，文件中存储文法描述。start 是第一个顶层规则。用户输入一行，返回解析结果。

下面我们完成计算表达式的例子：

```
use Parse::RecDescent;
#定义文法
my $grammar = <<'EOG';
```

```
 addition_expr: '+' number number
                    { print "successfully matched addition_expr rule\n"; }
                  | <error: illegal expression>
   number: /-?\d+/
EOG
#读入要解析的文本
my @lines = <DATA>;
my $text = join('', @lines);
#建立一个解析器
my $parserRef = new Parse::RecDescent($grammar);
my $ret = $parserRef->addition_expr($text);
die $@ if $@;
```

其中，规则中的大括号中的是一个执行语句，当执行到这个规则时执行该语句。

它的全局变量介绍如下：

全局变量	说明
$::RD_ERRORS	是否报告致命错误。
$::RD_WARN	是否报告警告信息。
$::RD_HINT	是否显示建议。
$::RD_TRACE	是否跟踪解析行为。
$::RD_AUTOSTUB	为没定义的规则生成 "stubs"。
$::RD_AUTOACTION	附加 action 到产品。当一个规则没有它自己的相关动作代码时就会运行这段代码。 例如，如下代码将打印规则的匹配情况： $::RD_AUTOACTION = q { print "$item[0]: @item[1..$#item]\n"; 1 };

内部变量介绍如下：

内部变量名	含义
@item	文法中的每一个分隔符分别代表 item[1]..item[n]，而 item[0] 则代表当前使用的规则名。这种处理方法类似于正则表达式中的圆括号。 当匹配一个重复的子规则时，@item 包含一个对数组的引用，而在

内部变量名	含义
	已经匹配一个非重复的子规则时，@item 包含值 1（即 true）。
@arg	数组@arg 和哈希变量%arg 存储从其它规则传给该规则的参数。
$return	如果一个规则正确匹配，则返回$return 的值。$return 是 yacc 文法中的$$的等价物。如果让$return=0，则它会立即正确返回。 从顶层规则返回的值将返回给顶层规则方法的调用者。
$commit	当前规则的当前 commit 状态。
$skip	当前终结符的前缀。
$text	剩下未解析的文本。改变$text 不会导致失败的匹配，只会挽救成功的匹配。因此，可能动态的改变要解析的文本—— 例如，提供一个象#include 的功能。如果让$text="，则该规则会返回，而且成功匹配。
$thisline	保存当前解析的当前行号 (从 1 开始)。
prevline	保存已经成功解析的行号(其值在每行结束处会与$thisline 有所不同)。
$thiscolumn	存储当前行解析的当前列号 (从 1 开始)。
$prevcolumn	$prevcolumn 存储上一个成功解析的字符的列号。通常，$prevcolumn == $thiscolumn-1，除了在行结尾处。
$thisoffset	$thisoffset 存储当前解析位置在全部被解析文本的偏移量(从 0 开始)。
$prevoffset	$prevoffset 存储上一个成功解析的字符的偏移量。在任何情况下，$prevoffset == $thisoffset-1。
@itempos	数组@itempos 存储对应每一个@item 的元素的一个 hash 引用。
$thisparser	Parse::RecDescent 对象的引用。
$thisrule	一个对应当前正在匹配的规则的 Parse::RecDescent::Rule 对象的引用。
$thisprod	对应当前正在被匹配的产品的 Parse::RecDescent::Production 对象的引用。
$score	$score 存储当前最大的产品 score，由先前声明的<score:...> 指示器

内部变量名	含义
	决定。 当有多个规则能匹配文本时，将采用 score 最大的规则成功返回。
$score_return	$score_return 存储成功产品对应的返回值。

现在让我们把文法变得稍微复杂一点，并改用测试框架。

```
start: '+' number number
                    { $return = $item[2] + $item[3];    }
number: /-?\d+/
```

测试结果是：

```
In [1] := + 1 2
Out [1] = 3
In [2] := + 12 -3
Out [2] = 9
In [3] := - 1 2
Out [3] = <<UNDEF>>
```

文法也可以简写成：

```
start: '+' number number
                    { $item[2] + $item[3];    }
number: /-?\d+/
```

让我们把这个计算器变得更真实一点：

```
                                        {my %variables}
```

```
start:          statement    /^\Z/            {$variables {'.'} = $item [1]}

statement:      variable '=' statement     {$variables {$item [1]} = $item [3]}
                | expression

expression:     term '+' expression        {$item [1] + $item [3]}
                | term '-' expression      {$item [1] - $item [3]}
                | term

term:           factor '*' term            {$item [1] * $item [3]}
                | factor '/' term          {$item [1] / $item [3]}
                | factor

factor:         number
                | variable                 {$variables {$item [1]} ||=
                                                Math::BigInt -> new (0)}
                | '+' factor               {$item [2]}
                | '-' factor               {$item [2] * -1}
                | '(' statement ')'        {$item [2]}

number:         /\d+/                      {Math::BigInt -> new ($item [1])}
```

```
variable:       /[a-z]+/i

                | '.'
```

这个文法可以处理赋值语句，并可定义变量。这里的.作为一个特殊的变量保存上一次计算的结果。

```
In [1] := 10 + 4

Out [1] = +14

In [2] := 3276581927324128 * 1234176413461234

Out [2] = +4043880131476789972060280853952

In [3] := Larry = 1

Out [3] = +1

In [4] := Tom = 2

Out [4] = +2

In [5] := Randal = 3

Out [5] = +3

In [6] := Larry + Tom + Randal

Out [6] = +6

In [7] := 3 * (4 + 5)

Out [7] = +27

In [8] := - Larry + . * (Tom ---- 23) + + + + + + Randal

Out [8] = +677

In [9] := .

Out [9] = +677

In [10] := x = (y = 3) + 4 + (z = 6)
```

Out [10] = +13

In [11] := y

Out [11] = +3

In [12] := z

Out [12] = +6

In [13] := x

Out [13] = +13

重复

规则是可重复的用末尾加(s)表示。又如，规则：

BaseRequest: Preface(s?) Name /(is)?/

声明 name 前面可以有 0 或多个 prefaces。也可以把重复次数加一个限制：

BaseRequest: Preface(0..10) Name /(is)?/

规则的附加修饰列表如下：

修饰	说明
s	强制重复规则。出现 1 或多次。
?	可选的子规则。出现 0 或 1 次。
(s?)	可选的重复子规则。出现 0 或多次。
(N)	重复子规则。必须出现正好 N 次。
(m..n)	重复子规则。从 m 到 n 次重复。
(..M)	重复子规则。从 1 到 M 次重复。
(N..)	重复子规则。必须出现至少 N 次。

指示器

指示器列表如下：

指示器	说明
<commit>	为了提高效率，允许解析树的递归下降剪枝。在一个规则中，一个<commit> 指示器告诉规则，如果当前产品失败，则忽略随后的匹配。
<uncommit>	
<reject>	立即导致当前的产品失败(等价于动作{undef}，只是看起来更明显)。一个<reject>用于想得到一个产品中动作的副作用，而不想预判断规则中其他产品的匹配。 <reject: expression> 仅当条件是真时拒绝。
<skip>	忽略终结符之间的东西。
<resync>	这个指示提供了一个消耗一些正待解析的文本的与众不同的含义。通常用来跳过大量的输入。它最简单的形式<resync>简单地消耗掉文本直到而且包括下一个新行("\n") 。 仅当发现一个新行时匹配成功。在这种情况下，它的周围规则成功时返回 0。
<error>	错误处理。 这个指示提供解析时自动的或用户定义的错误消息生成。它的最简单的形式<error> 基于期望的最后一个项目和导致它失败的文本之间的错误匹配，准备一个错误消息。 <error?>仅当规则已经提交时发出错误。 <error: message> 用于客户化错误消息。
<defer>	一个<defer> 指示器像一个代码块，除了它只是在当前产品使用到最后的匹配中。

动态规则

Token 使用双引号或正则表达式做插值。因此，通过引用产品规则中的其它项目可以把它们做成上下文相关的。例如，可以通过插值$item。<matchrule:...>是另外一个构建动态规则的方法。每次碰到指示器时，冒号后的部分当成为一个双引号字符串计算，而它的值作为要匹配的子规则的名字，如：

```
statement:    "if"       if_statement        "end if"
```

```
        |   "while" while_statement    "end while"

        |   "for"     for_statement       "end for"
```

等价于：

```
keyword:       "if" | "while" | "for"

statement:    keyword <matchrule: "$item[1]_statement"> "end $item[1]"
```

子规则参数

可以使用参数调用子规则。参数放在方括号中。参数可以通过@arg 或 %arg 访问，可以选择其中任何一种方式访问参数。参数出现在重复描述符之前：rule: sub other[$item[1]](s)。

带参数的子规则例子如下：

```
start:     word    rev[$item[1]]    {1}



rev:         word    {$return = 1 if $arg [0] eq reverse $item [1]; undef}



word:       /\w+/
```

测试结果是：

```
In [1] := foo oof

Out [1] = 1

In [2] := foo bar

Out [2] = <<UNDEF>>
```

使用举例

要把如下这段文本按 id 整理成一行的格式。

```
id:line1
```

272752574525724572457

24625762572456724724724262346363246326363666346363636

id:line2

795905634562357254572725473473457347573734352547457343457457435734546435

id:line3

42346243642

43262342

42623466

43623432

4543

即这样：

id:line1
272752574525724572457424625762572456724724724262346363246326363666346363636

id:line2 795905634562357254572725473473457347573734352547457343457457435734546435

id:line3 42346243643262342442623466643623432

先用文法描述输入文本：

addition_expr: a_line(s)

a_line:'id:' line_id number(s)

line_id:/.+\n/

number:/\d+\s*\n/

定义好触发动作就可以达到目的。程序如下：

```
use Parse::RecDescent;
use Data::Dumper;

my $grammar = <<'EOG';
 addition_expr: a_line(s)
                    | <error: illegal expression>
   a_line:'id:' line_id number(s)
                      { print "\n"; }
   line_id:/.+\n/
                      { $item[1] =~ s/\s+$//;print "id:$item[1] "; }
   number:/\d+\s*\n/
                      { $item[1] =~ s/\s+$//;print $item[1]; }
EOG

open(DATA,"e:/perl/code/t.txt");

my @lines = <DATA>;

my $text = join('', @lines);
my $parserRef = new Parse::RecDescent($grammar);
my $ret = $parserRef->addition_expr($text);
die $@ if $@;
```

4.12 网络

4.12.1 Net::FTP

可以用 perl 内在的模块 Net::FTP 方便的实现 ftp 客户端功能。取得一个文件的例子如下：

```
use Net::FTP;

$ftp = Net::FTP->new("10.120.130.88", Debug => 0) or die $!;
$ftp->login("hero",'hero') or die $!;
@list =$ftp->ls ("/OAM" ) or die $!;

$ftp->binary();
$ftp->get('/OAM/RTCS_Log/QosT.log');
$ftp->quit;

foreach $line (@list) {
   print "$line\n";
}
```

相关的方法列表如下：

方法	说明
new (HOST [,OPTIONS])	取得一个新的 Net::FTP 对象的构造函数。HOST 是远程主机的名字 。 OPTIONS 使用了关键字和值对的形式。可能的选项是： Firewall – 作为 FTP 防火墙的机器的名字。可以通过设置环境变量 FTP_FIREWALL 覆盖这个值。 FirewallType – 防火墙的类型。共有 8 种配置可能，具体请参见 Net::Config。 BlockSize –Net::FTP 传输时使用的块大小(缺省值 10240)。 Port – 用于 FTP 连接的端口号。 Timeout – 设置超时时间(缺省 120)。 Debug – 调试级别(0，关闭调试；1，打开调试)。 Passive – 如果设置一个非零值，所有的数据传输会使用被动模式。通常并不需要这样设置，除非一些哑服务器和一些防火墙配置。这个选项也可以通过环境变量 FTP_PASSIVE 设置。 Hash – 如果给一个对文件句柄的引用 (例如， *STDERR)，每传输 1024 字节就打出一个 # 到该文件句柄。它只是为你调用 hash()方法，以便为所有的传输做标记。也可以显式的调用 hash()方法。 如果构造函数失败，将返回 undef ，通过变量$@可得到错误信息。
login ([LOGIN [,PASSWORD [, ACCOUNT]]])	登录到FTP服务器，给出登录信息。如果不给出参数，Net::FTP 使用 Net::Netrc 包查找登录信息。如果没有发现信息，将使用 anonymous 登录。如果没有给出密码，将用 anonymous@ 作为密码。 如果是通过防火墙，将不带参数的调用 authorize 方法。
authorize ([AUTH [, RESP]])	这是一个一些防火墙 ftp 代理使用的协议。它用来认证发送数据出去的用户。如果两个参数都没有提供，就使用 Net::Netrc 认证。
site (ARGS)	发送一个 SITE 命令给远程的服务器并等待一个响应。返回一个响应码。SITE 命令是 FTP 服务器规定的扩展命令。

方法	说明
type (TYPE [, ARGS])	发送一个 TYPE 命令给远程的 FTP 服务器以改变数据传输的类型。返回值是前一个值。
ascii ([ARGS]) binary([ARGS]) ebcdic([ARGS]) byte([ARGS])	type 的同义语，已经设置第一个参数。注意：不完全支持 ebcdic 和 byte。
rename (OLDNAME, NEWNAME)	重命名在远程 FTP 服务器上的一个文件。把它从 OLDNAME 命名为 NEWNAME。这个是通过发送 RNFR 和 RNTO 命令实现的。
delete (FILENAME)	发送一个删除 FILENAME 的请求给服务器。
cwd ([DIR])	改变路径到$dir。如果$dir 是 ".."，则使用 FTP 的 CDUP 命令上移到上级目录。如果不给出路径则改变路径到根目录。
cdup ()	进入 FTP 服务器当前目录的父目录。
pwd ()	显示 FTP 服务器的当前工作目录。
restart (WHERE)	设置开始接下来的数据传输的字节偏移量。Net::FTP 简单的纪录此值，然后当下一次传输数据时使用它。因此，这个方法不会返回错误，但是设置它可能导致随后的数据传输失败。
rmdir (DIR)	删除 FTP 服务器目录 DIR。
mkdir (DIR [, RECURSE])	使用名称 DIR 创建一个新路径。如果 RECURSE 是 true，则 mkdir 会试图创建给定路径下的所有路径。 返回新路径的完全路径。
ls ([DIR])	取得 DIR 或者当前路径的的路径列表。 在一个数组上下文中，返回一个服务器返回的数组的列表；在标量上下文中，返回一个列表的引用。
dir ([DIR])	取得一个 DIR 的路径列表或当前路径的列表。 在一个数组上下文中，返回一个服务器返回的数组的列表；在标量上下文中，返回一个列表的引用。

方法	说明
get (REMOTE_FILE [, LOCAL_FILE [, WHERE]])	从服务器取得文件 REMOTE_FILE，并存储在本地。LOCAL_FILE 可能是一个文件名或者一个文件句柄。如果不加声明，该文件将存储在当前路径使用远程文件同样的文件名。 如果给出 WHERE，那么就不会传送文件的第一个 WHERE 字节，而剩下的字节将会附加到已经存在的本地文件。简单的说就是指出续传的断点。 返回 LOCAL_FILE。没有给出 LOCAL_FILE 就返回生成的本地文件名。当出错时就返回 undef。 LOCAL_FILE 参数有时无法正常工作，这时可以用 chdir 把当前路径改到 LOCAL_FILE 所在的路径。
put (LOCAL_FILE [, REMOTE_FILE])	把一个文件上传到服务器上。LOCAL_FILE 可以是一个文件名或者文件句柄。如果 LOCAL_FILE 是一个文件句柄，则必须提供 REMOTE_FILE。如果没有声明 REMOTE_FILE，则会使用和 LOCAL_FILE 同样的名称保存到当前路径。 返回 REMOTE_FILE，或生成的远程文件名，当没有给出 REMOTE_FILE 时。 注意：当由于某种原因，传输没有完成时，将返回一个错误，但是不会自动删除已经传输的内容。
put_unique (LOCAL_FILE [, REMOTE_FILE])	和 put 相同，但是使用 STOU 命令。 返回服务器上的文件名。
append (LOCAL_FILE [, REMOTE_FILE])	和 put 相同，但是附加到服务器上的文件。 返回 REMOTE_FILE，或者生成的远程文件名，如果没有给出 REMOTE_FILE。
unique_name ()	返回最近一个使用 STOU 命令存储到服务器上的文件名。
mdtm (FILE)	返回给定文件的修改时间。
size (FILE)	返回服务器上文件的字节大小。
supported (CMD)	返回 TRUE，如果服务器支持给定的命令。

方法	说明
hash ([FILEHANDLE_GLOB_REF],[BYTES_PER_HASH_MARK])	如果不使用参数的方式调用，或者第一个参数是 false，将会关闭 hash 标志输出。如果第一个参数是 true 但是不是指向一个文件句柄的引用，就用*STDERR 代替。第二个参数是每个 hash 标志打印的字节数，缺省是 1024。返回值是一个包含两个值的数组的引用：文件句柄引用和每个 hash 标志打印的字节数。
nlst ([DIR])	发送一个 NLST 命令（返回当前路径的列表）给服务器，伴随一个可选的参数。
list ([DIR])	和 nlst 相同，但是使用 LIST 命令（返回当前路径的列表或文件的说明）。
retr (FILE)	开始返回一个服务器上叫做 FILE 的文件的拷贝。
stor (FILE)	告诉服务器，你要上传一个文件到服务器。FILE 是要创建的文件名。
stou (FILE)	和 stor 相同，但是使用 STOU 命令。服务器创建文件的唯一名称，当数据连接关闭后，可以通过 unique_name 方法得到文件名。
appe (FILE)	告诉服务器，附加数据到 FILE 文件的结尾。如果该文件不存在，就创建它。
port ([PORT])	发送一个 PORT 命令给服务器。如果声明了 PORT ，就把它发送给服务器。如果没有，就建立一个监听 socket，发送正确的信息给服务器。
pasv ()	告诉服务器进入被动模式。返回该服务器监听的文本，这段文本适合使用 port 方法发送到另外一个 ftp 服务器。
pasv_xfer (SRC_FILE, DEST_SERVER [, DEST_FILE])	这个方法将会在两个远程 ftp 服务器间做文件传输。如果忽略 DEST_FILE，则将会使用 SRC_FILE 中的文件名。
pasv_xfer_unique (SRC_FILE, DEST_SERVER [, DEST_FILE])	像 pasv_xfer 但是存储在远程服务器上的文件使用 STOU 命令。
pasv_wait (NON_PASV_SERVER)	用这个方法等待主动和被动服务器间的传输结束。这个方法应该使用 Net::FTP 对象在被动服务器上调用，并使用主动服务器作为参数。
abort ()	中止当前的数据传输。

方法	说明
quit ()	发送 QUIT 命令给远程 FTP server 并关闭 socket 连接。

4.12.2 Net::Telnet

Net::Telnet 包装了用于登录终端的 telnet 协议，使用这个模块相当直接：

```
use Net::Telnet;

$telnet = new Net::Telnet ( Timeout=>10,
                            Dump_Log     => 'dump.log',
                            Errmode=>'die');
#与机器建立连接
$telnet->open('130.59.1.15');
#登录（等待“login:”的出现）
$telnet->waitfor('/login: $/i');
#输入用户名（返回“interf”）
$telnet->print('interf');
#输入密码（等待“password:”的出现）
$telnet->waitfor('/password: $/i');
#输入口令（返回“interface”）
$telnet->print('interface');
#完成登录（等待提示符“#”的出现）
$telnet->waitfor('/\# $/i');
#发送命令“who”
$telnet->print('who');
#命令“who”执行结束（等待提示符“#”的出现）
while ($line = $telnet->getline()){
    print $line;
}
```

现在把这段代码写的更简洁一点：

```
use strict;
use Net::Telnet ();

my $username= 'interf';
my $passwd ='interface';

my $t = new Net::Telnet (Timeout => 10,
                         Errmode=>'die',
                         Prompt => '/\# $/i');
$t->open("130.59.1.15");
$t->login($username, $passwd);
```

```
my @lines = $t->cmd("who");
print @lines;
```

4.12.3 WebService

Perl 本身也需要安装附加的扩展包来提供对 Web Service 的支持，您可以按照下面的网址下载相关软件。

Perl Web Service 软件包可以从 ftp://ftp.activestate.com/Web Service/Perl/WebService-0.01.tar.gz 得到。Perl 介绍可以从 http://aspn.activestate.com/ASPN/Web Services/SWSAPI/perltut 得到。

下载后的安装程序在WINDOWS平台上如下（安装了 nmake.exe 工具程序，Visual Studio 开发包软件附带）：

1. 解压缩下载的软件包，进入解压后建立的目录中。

2. 执行如下 3 个命令， 要保证 PATH 环境变量中可以访问 perl 和 nmake 程序：

```
>perl makefile.pl

>nmake

>nmake install
```

代码演示如下：

```
use Web Service::ServiceProxy;
my $wsdl = "http://localhost/userx/biz.asmx?WSDL";
my $get_count = Web Service::ServiceProxy->new($wsdl);
print $get_count->get_count("111", "s111");
```

4.13 提取网页

4.13.1 HTTP::Request

一个提取网页的例子如下：

```
use LWP::UserAgent;
use HTTP::Request::Common;

$protein="MSSSTPFDPYALSEHDEERPQNVQSKSRTAELQAEIDDTVGIMRDNINKVAERGERL
TSI";
```

```
my $agent=LWP::UserAgent->new;

my $SUSUI_URL="http://sosui.proteome.bio.tuat.ac.jp/cgi-bin/adv_sosui.cgi";

my $req = HTTP::Request->new(POST => "$SUSUI_URL");

$req->content("query_seq=$protein");

my $res = $agent->request($req);

# 检查响应的输出
if ($res->is_success) {
        print $res->content;
} else {
        print "Bad luck this time\n";
}
```

取得文本文件的一个例子如下(http, ftp)：

```
use LWP::UserAgent;
use CGI qw(header -no_debug);

my $URL = 'http://www.yahoo.com/';
my $res = LWP::UserAgent->new->request(new HTTP::Request GET => $URL);
print header, $res->is_success ? $res->content : $res->status_line;
```

取得 jpeg/gif/bmp 文件并返回它。

```
use LWP::UserAgent;
use CGI qw(header -no_debug);
$URL =
'http://a100.g.akamaitech.net/7/100/70/0001/www.fool.com/art/new/butts/go99.
gif';
my $res = LWP::UserAgent->new->request(new HTTP::Request GET => $URL);
binmode(STDOUT);
print $res->is_success ? (header('image/gif'), $res->content)
                       : (header('text/html'), $res->status_line);
```

取得密码保护的文件：

```
BEGIN {
   package RequestAgent;
   use LWP::UserAgent;
   @ISA = qw(LWP::UserAgent);

   sub new { LWP::UserAgent::new(@_); }
```

```
   sub get_basic_credentials { return 'user', 'password' }
}
use CGI qw(header -no_debug);

my $res = RequestAgent->new->request(new HTTP::Request GET => $URL);
print header, $res->is_success ? $res->content : $res->status_line;
```

建立 REFERER 和其他的 HTTP 头参数：

```
use LWP::UserAgent;
use HTTP::Headers;
use CGI qw(header -no_debug);

my $URL = 'http://localhost/cgi-bin/hello.cgi';
my $res = LWP::UserAgent->new->request(
   new HTTP::Request(
      GET => $URL,
      new HTTP::Headers referer => 'http://www.yahoo.com'),
);
print header, $res->is_success ? $res->content : $res->status_line;
```

取得文件的指定部分(第一个 MAXSIZE 字节)：

```
use LWP::UserAgent;
use CGI qw(header -no_debug);

my $URL = 'http://www.yahoo.com/';
my $MAXSIZE = 1024;

print header;
my $res = LWP::UserAgent->new->request(
   new HTTP::Request(GET => $URL), \&callback, $MAXSIZE);

sub callback { my($data, $response, $protocol) = @_; print $data; die }
```

取得和建立 cookies：

```
use LWP::UserAgent;
use CGI qw(header -no_debug);
use HTTP::Cookies;

my $URL = 'http://mail.yahoo.com/';

my $ua = new LWP::UserAgent;
my $res = $ua->request(new HTTP::Request GET => $URL);
my $cookie_jar = new HTTP::Cookies;
$cookie_jar->extract_cookies($res);
```

```
print header;
if ($res->is_success) {
   my $req = new HTTP::Request GET => $URL;
   $cookie_jar->add_cookie_header($req);
   $res = $ua->request($req);
   print $res->is_success ? $res->as_string : $res->status_line;
} else {
   print $res->status_line;
}
```

声明代理服务器：

```
use LWP::UserAgent;
use CGI qw(header -no_debug);

my $URL = 'http://www.yahoo.com/';
my $ua = new LWP::UserAgent;

$ua->proxy(['http', 'ftp'], 'http://proxy.sn.no:8001/');
$ua->proxy('gopher', 'http://proxy.sn.no:8001/');

my $res = $ua->request(new HTTP::Request GET => $URL);
print header, $res->is_success ? $res->content : $res->status_line;
```

检查重定向：

```
use LWP::UserAgent;
use CGI qw(header -no_debug);

my $URL = 'http://www.yahoo.com/';
my $res = LWP::UserAgent->new->request(new HTTP::Request GET => $URL);

print header;
print $res->request->url if $res->previous->is_redirect;
```

为 POST 方法创建参数：

```
use URI::URL;
use HTTP::Request;
use LWP::UserAgent;
use CGI qw(header -no_debug);

my $URL = 'http://yahoo.com/?login=mylogin&password=mypassword';
my $uri = new URI $URL;
my $method = 'POST';
```

```
my $request;
if (uc($method) eq 'POST') {
   my $query = $uri->query;
   (my $url = $uri->as_string) =~ s/\?$query$//;
   $request = new HTTP::Request ($method, $url);
   $request->header('Content-Type' => 'application/x-www-form-urlencoded');
   $request->content($query);
} else {
   $request = new HTTP::Request ($method, $uri->as_string);
};

# 增加 Host 域，因为 HTTP/1.1 需要它。
$request->header(Host => $uri->host_port) if $uri->scheme ne 'file';

my $res = LWP::UserAgent->new->request($request);

print header, $res->is_success ? $res->content : $res->status_line;
```

上传文件：

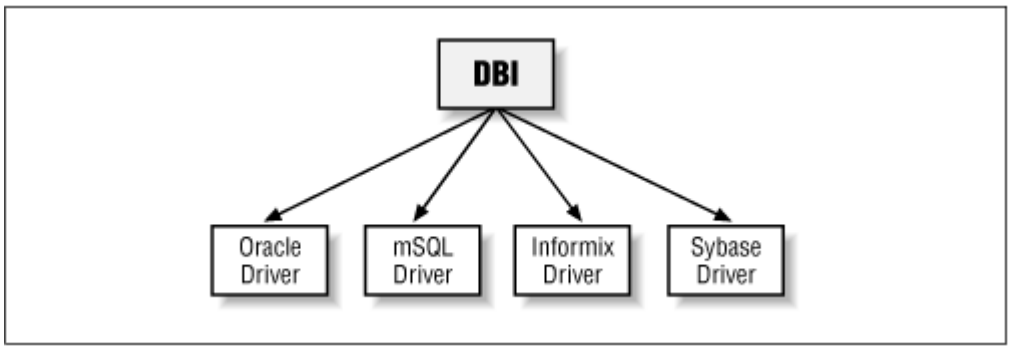
```
use HTTP::Request::Common;
use LWP::UserAgent;
use CGI qw(header -no_debug);

my $URL = 'http://localhost/cgi-bin/survey.cgi';
my $req = POST $URL,
   Content_Type => 'form-data',
   Content      =>
   [
      name       => 'Paul Kulchenko',
      email      => 'paulclinger@yahoo.com',
      surveyfile => ['./survey.dat'], # this file will be uploaded
   ];

my $res = LWP::UserAgent->new->request($req);

print header, $res->is_success ? $res->content : $res->status_line;
```

第5章 数据库 DBI

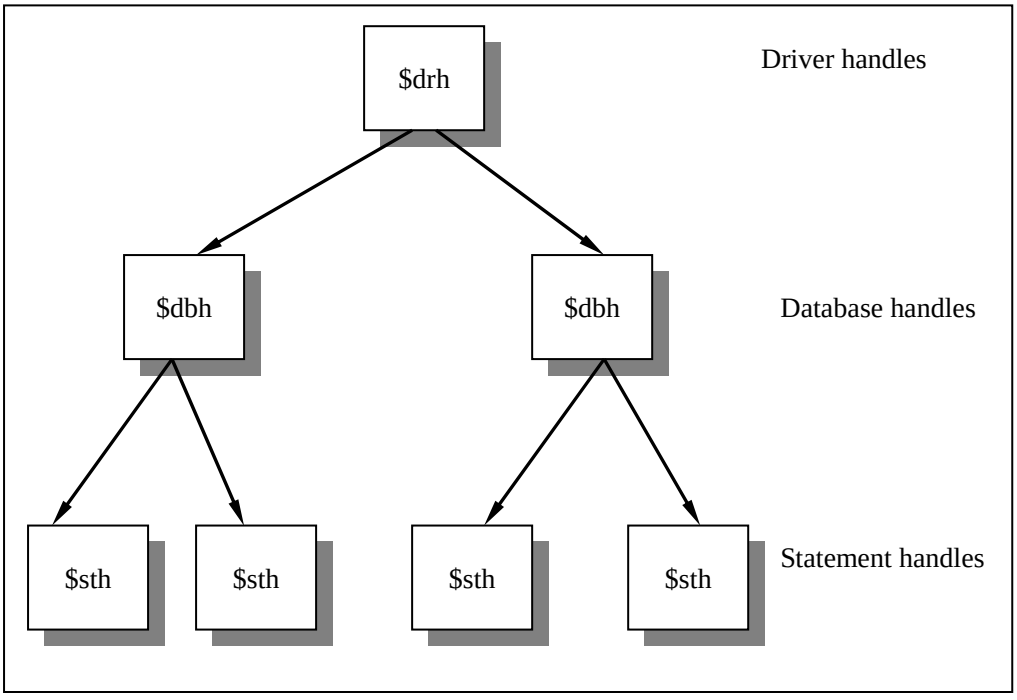
本章首先概要介绍 Perl 中的数据库编程接口 DBI。然后介绍 DBI 编程的相关技术：调试，DBD::Proxy 和内存数据库 DBD::AnyData 以及封装 DBI 的方法 Tie::DBI。虽然 DBI 是一个编程的统一接口，但是各种数据库还是存在一定的差别，所以最后是按数据库类型介绍 Perl 中的数据库编程，而且虽然是以各种 DBD-*介绍为主，但是这里并没有完全限定于 DBI，为实际编程提供了选择的余地。

5.1 概述

DBI（DataBase Interface）是 Perl 独立于数据库的编程。它的体系结构类似于 ODBC 的结构。DBI 是一个编程接口，对应于 ODBC 编程接口，DBD（DataBase Driver）是数据库驱动对应于 ODBC 驱动。图示如下：



Driver handle、Database handle 与 Statement handle 之间的关系如下图所示：



Driver handle封装了一个DBD，对每一个加载的DBD唯一对应一个Driver handle。Driver handle 只供 DBI 内部使用，用户程序并不直接使用。Statement handle 封装了发给数据库的语句，也是通过它返回结果集。这三类句柄的一般的命名约定如下：

$drh	数据库驱动句柄(Driver handles)。
$dbh	数据库句柄(Database handles)。
$sth	语句句柄(Statement handles)。

DBI 相关方法：

方法	说明
connect($data_source, $username, $ password, \%attr)	构造函数，与数据库建立连接。DBI 将自动加载相应的驱动。
Available_drivers()	检测所有的数据库驱动。
Data_sources($drive_name)	列出所有 DBD 能检测到的数据源。

通过 connect 函数与数据库建立连接时，其中连接参数的设置方法是通过关键字——值对的方法设置的。连接参数的选项有：

连接参数	说明
AutoCommit	是否自动提交事务。
ChopBlanks	从 CHAR 类型的列截短尾部空格。
PrintError	告诉 DBI 调用 Perl warn( ) 函数(典型的导致打印错误到屏幕)。
RaiseError	缺省关闭。如果打开，将导致 DBI 使用 die 提出一个异常。能和 eval 一起使用捕捉异常。
Warn	打开有用的警告。

可以通过，$DBI::errstr 和$dbh->errstr 返回错误消息，通过$dbh->err 返回错误代码。

dbh 方法如下：

方法	说明

方法	说明
prepare($sql_select)	返回一个 sth。
disconnect()	断开连接。
do($statement, [\%attr], [@bind_values])	返回值，sql 语句影响的行数。
begin_work()	开始事务。
commit()	提交事务。
rollback()	回滚事务。
quote($name)	在数据两边加上引号，并对数据本身内部的引号进行转义，避免数据本身的引号把数据截断。
selectrow_hashref($statement)	相当于一次捆绑式调用 prepare + execute + fetch + finish 的方法。它返回结果集的第一行。$statement 参数可以是一个已经 prepare 的 sth，此时执行时跳过 prepare 步骤。
selectrow_arrayref($statement)	相当于一次捆绑式调用 prepare + execute + fetch + finish 的方法。它返回结果集的第一行。$statement 参数可以是一个已经 prepare 的 sth，此时执行时跳过 prepare 步骤。
selectcol_arrayref($statement, [\%attr])	相当于一次捆绑式调用 prepare + execute + fetch + finish 的方法。返回一列数据的数组引用。例如： my @Name = @{$dbh->selectcol_arrayref("SELECT col3 FROM table1 ")}; foreach (@Name){print "$_\n"}
selectall_arrayref($statement)	批量取得行。相当于一次捆绑式调用 prepare + execute + fetchrow_arrayref + finish 的方法。返回的是 fetchrow_arrayref 的调用结果。例如： *my $rows = $dbh->selectall_arrayref($sql);* *for my $row (@$rows) {* *print $$row[0], $$row[1],"\n";* *}*
selectall_hashref($statement, $keyfield, ...)	批量取得行。相当于一次捆绑式调用 prepare + execute + fetchall_hashref + finish 的方法。返回的是 fetchall_hashref 的调用结果。

方法	说明
clone(\%attr)	返回一个新的数据库句柄$dbh2，是$dbh1 的复制。即使$dbh1 当前是断开连接的也可以使用该方法。
Tables()	返回当前数据库中所有表名。

sth 方法：

方法	说明
bind_param($num, $name, options)	绑定参数。
can($method_name)	检查方法是否已实现。如果 DBI 已经提供了一个存根方法则返回 false。
execute()	执行语句。
fetchrow()	取得一个查询语句的返回值。例如： @array = $query->fetchrow();
fetchrow_array()	取得数据的下一行。把各列值作为一个列表返回。列中的 Null 值返回成`undef'。
fetchrow_arrayref()	取得数据的下一行。返回一个数组的引用。列中的 Null 值返回成`undef'。这是取得数据最快的方法，特别是当和`$sth->`bind_columns'一起使用时。
fetchrow_hashref()	取得数据的下一行。返回一个 hash 的引用。该引用中包含列名称和列值对。
fetchall_arrayref($slice, $max_rows)	允许批量取得行。例如： *my $records =$sth->fetchall_arrayref;* *for my $record (@$records) {* *print "column1 = ", $record->[0], "\n";* *}*

方法	说明
fetchall_hashref($keyfield, ...)	可用于返回所有的数据。返回一个 hash 引用。每行一个入口。参数$keyfield 声明哪一个列值用作键值。也可以使用整数列号(从 1 开始)。在 hash 中的每一个入口是一个 hash 的引用，保存该行的列值。 例如： *my $records = $sth->fetchall_hashref(1);* *for my $id (keys %$records) {* * my $record = $records->{ $id };* * print $record->{'OBJECT_NAME'}, "\n";* *}*
finish()	结束语句。

5.2 调试

调试 DBI 使用环境变量 DBI_PROFILE。

级别	说明
1	执行时间总体情况。
2	执行时间按语句汇总。
4	执行时间按方法汇总。
6	执行时间按方法和语句汇总。
8	按方法和语句汇总。输出方法的全称。
10	最详细的输出语句然后方法的全称。

例如在脚本中使用：

$ENV{DBI_PROFILE} = 1;

将显示总共的执行时间。

DBI::Profile: 11.250000 seconds 93.99% (1025 method calls)

5.3 DBI 代理 DBD::Proxy

使用独立于数据库的协议连接远程数据库。dbiproxy 是一个小的 dbi proxy 服务器。能够用端口号做为参数：

>dbiproxy –localport 4444

客户端建立连接的方式如下：

$dbh = DBI->connect(“dbi:Proxy:hostname=test;port=4444\

dsn=dbi:ODBC:datasource”,”,”);

5.4 DBD::AnyData

DBD::AnyData是内存数据库DBD::RAM的升级版本，请先确保已安装DBI，然后按顺序安装AnyData，SQL::Statement和DBD::AnyData。这些包都可以在www.cpan.org找到。

它除了支持内存数据库，还支持把各种文件看成数据库。

内存支持的数据类型包括：INTEGER,CHAR(n),VARCHAR(n),REAL,BLOB。

如下是一个简单的例子。首先，创建一个表，然后插入字符串，使用 SQL 语句检索该记录，然后打印它。

```
use DBI;

my $dbh = DBI->connect('dbi:AnyData(RaiseError=>1):');
$dbh->do("CREATE TABLE test (id TEXT,phrase TEXT)");
$dbh->do("INSERT INTO test VALUES (1,'foo')");
$dbh->do("INSERT INTO test VALUES (2,'bar')");
$dbh->do("UPDATE test SET phrase='baz' WHERE id = '2'");
$dbh->do("DELETE FROM test WHERE id = '1'");

my $classes_sth = $dbh->prepare( "SELECT id,phrase FROM test" );

$classes_sth->execute;
while (my($id,$class_title) = $classes_sth->fetchrow_array) {
    print "$id,$class_title \n";
}
```

它认识的文件类型有 CSV（逗号分割），固定长度等。还支持 XML, Mp3, HTML 表等特殊的格式。

例如，有这样一个 TAB 分隔的文件（data.txt）。

Li_Hong girl

Wang_Wei boy

Zhang_Qian girl

和（datb.txt）：

Li_Hong 866-7544

Wang_Wei 521-4238

Zhang_Qian 346-1233

Huang_Mi 221-33323

…

要的结果是取 data.txt 中存在的 datb.txt 的电话号码，由于当前这个模块不支持联结，只能用循环模拟联结。代码如下：

```
use DBI;

my $dbh = DBI->connect('dbi:AnyData(RaiseError=>1):');
$table='students';
$format='Tab';
$file='data.txt';
$flags = { col_names => 'stu_name,sex'};

$dbh->func( $table, $format, $file, $flags, 'ad_import');

$table='students_tel';
$format='Tab';
$file='datb.txt';
$flags = { col_names => 'stu_name,tel_no'};

$dbh->func( $table, $format, $file, $flags, 'ad_import');

my $tel_sth = $dbh->prepare( "SELECT stu_name,tel_no FROM students_tel " );
my $stu_sth     = $dbh->prepare( "SELECT stu_name FROM students WHERE stu_name = ?" );
#~ my $classes_sth = $dbh->prepare( "SELECT a.stu_name,a.tel_no FROM students_tel  a , students b  where a.stu_name = b.stu_name " );
$tel_sth->execute;
```

```
while (my($id,$tel_no) = $tel_sth->fetchrow_array) {
        $stu_sth->execute($id);
        my $row = $stu_sth->fetchrow_arrayref;
        my $stu_name = $row ? $row->[0] : '';
        print "$tel_no : $stu_name\n";
}
```

但是缺省安装的 DBD::AnyData 不支持连接 join。据模块作者 Jeff 说，把 SQL::Statement 升级到 1.005，该模块就支持连接了。

5.5 Tie::DBI

Tie::DBI 把 hash 表和关系数据库通过 DBI 接口关联起来了。

选项	说明	缺省值
CLOBBER	它控制是否数据库可通过绑定的 hash 写入。值越高权限越大。 0 使数据库成为只读的。试图存入 hash 将导致一个致命错误。 0 所有的读操作。 1 更新列。 1 增加记录。 2 删除记录。 3 清除整个表。	0
AUTOCOMMIT	是否自动提交事务。	1
DEBUG	调试。当调试选项设置成一个非 0 值，该模块会打印内容 SQL 语句和其他的调试信息到标准错误。调试选项更大的值导致更详细的输出。	0
WARN	如果设置成一个非 0 值。警告非法操作，例如试图删除关键列的值。如果设置成 0，将会安静的忽略这些错误。	1

下面是使用它操作 Oracle 数据库的例子：

```
use Tie::DBI;
$username = 'test';
```

```
$password = ‘test’;
$database = ‘databasename’;
my $dbh = DBI->connect("dbi:Oracle:$database ", $username, $password,{AutoCommit => 0})
|| die "Unable to connect to xxx";

tie %h,Tie::DBI,{db          => $dbh,
                 table       => 'test',
                 key         => 'id',
                 CLOBBER     => 1};

# 取得键和值
@keys = keys %h;
@fields = keys %{$h{$keys[0]}};
print $h{'id1'}->{'field1'};
while (($key,$value) = each %h) {
    print "Key = $key:\n";
    foreach (sort keys %$value) {
        print "\t$_ => $value->{$_}\n";
    }
}

# 改变数据
$h{'id1'}->{'field1'} = 'new value';
$h{'id1'} = { field1 => 'newer value',
              field2 => 'even newer value',
              field3 => "so new it's squeaky clean" };

# 其他函数
tied(%h)->commit;
tied(%h)->rollback;
tied(%h)->select_where('price > 1.20');
@fieldnames = tied(%h)->fields;
 $dbh = tied(%h)->dbh;
dbh->disconnect();
```

5.6 MS SqlServer

5.6.1 WIN32:ODBC

WIN32:ODBC 并不是一个与 DBI 兼容的标准，但是它能在各种 WIN32 平台运行。安装方法如下：

- 从 http://www.roth.net/pub/ntperl/ODBC/取得最新的安装包，如 Win32ODBC_v970208.zip

以及。

- 解压 Odbc.pm (从 Win32ODBC_v970208.zip)到 C:\Perl\lib\Win32。
- 解压 odbc.pll (从 Win32_ODBC_Build_306.zip)到 C:\Perl\lib\Auto\Win32\odbc。

检索数据例子如下：

```
use Win32::ODBC;

if (!($db = new Win32::ODBC("DSN=ADOSamples;UID=sa;PWD="))){
    print "Error connecting to $DSN\n";
    print "Error: " . Win32::ODBC::Error() . "\n";
    exit;
}

$SqlStatement = "SELECT * FROM jobs";
if ($db->Sql($SqlStatement)){
    print "SQL failed.\n";
    print "Error: " . $db->Error() . "\n";
    $db->Close();
    exit;
}

while($db->FetchRow()){
    undef %Data;
    %Data = $db->DataHash();
    print Dumper(%Data);
}

$db->Close();
```

修改数据：

```
use win32::odbc;

my($DSN,$data,$table,$sql);
print " connecting to the database..... ";
$DSN ="win008";
$data = new Win32::ODBC($DSN);
$table =bbf;
$sql ="INSERT INTO $table(type,time,home,result,away,ht,status)
VALUES('$mtype','$mtime','$mhome','$mresult','$maway','$mht','$mstatus')";
$data->Sql($sql);
$data->Close();
print "ok!\n";
```

5.6.2 Win32::ADO

除了 ODBC，还可以使用 ADO 连接 SQL Server 数据源。Win32::ADO 可以用来检查数据库返回错误。

```
use Win32::ADO qw/CheckDBErrors/;
use Win32::OLE;

my $Conn = new Win32::OLE('ADODB.Connection');
#~ $Conn = $Server->CreateObject("ADODB.Connection");
$Conn->Open( "DSN=ADOSamples;UID=sa;PWD=" );
CheckDBErrors($Conn, \@DBErrors) or die "SQL Failed at ", __LINE__, "\n", @DBErrors;

my $RS = $Conn->Execute( "SELECT * FROM jobs" );

CheckDBErrors($Conn, \@DBErrors) or die "SQL Failed at ", __LINE__, "\n", @DBErrors;

my $count = $RS->Fields->{Count};
for (my $i = 0; $i < $count; $i++ ) {
    print $RS->Fields($i)->{Name},"\n";
}
while ( ! $RS->{EOF} ) {
    for (my $i = 0; $i < $count; $i++ ) {
        print $RS->Fields($i)->{Value},"\n";
    }
    $RS->MoveNext;
}
$RS->Close;
$Conn->Close;
```

调用存储过程示例如下：

```
use Win32::OLE;
use Win32::ADO qw/CheckDBErrors adVarChar adParamInput/;

$con = Win32::OLE->new("ADODB.Connection");
$cmd = Win32::OLE->new("ADODB.Command");
$rs = Win32::OLE->new("ADODB.Recordset");
$con->Open("DSN=ADOSamples;UID=sa;PWD=");
$con->Execute("drop table texttable");
$con->Execute("drop procedure textproc");

$con->Execute("create table texttable(text varchar(255))");
$con->Execute("create procedure textproc \@text varchar(255) as insert texttable values(\@text)");
```

```
$cmd->{ActiveConnection} = $con;
$cmd->{CommandText} = "textproc";
$cmd->{CommandType} = adCmdStoredProc;
$cmd->Parameters->Append($cmd->CreateParameter("foo", adVarChar, adParamInput, 255));

$cmd->Parameters->Item(0)->{value} = ("A" x 200);
$cmd->Execute;

$cmd->Parameters->Item(0)->{value} = "B" x 100;
$cmd->Execute;
$rs = $con->Execute("select text from texttable");
while (! $rs->EOF) {
    $v = $rs->Fields(0)->value;
    print length($v)."\n";
    print $v."\n";
    $rs->MoveNext;
}
```

5.6.3 DBD::ODBC

因为微软没有提供从非 windows(dos)平台访问 MS Sql Server 数据库的方法。这使得要在 UNIX 环境下访问 MS Sql Server 成了一个问题。

UNIX 环境下就没有 MS Sql Server 数据库的 DBI 驱动。可以从 http://www.freetds.org 取得 unix 下的 Sql Server 数据库的驱动程序。

freetds 使用 TDS 协议来和 MS Sql Server(Sybase)数据库打交道。TDS 协议有不同的版本，而且它们之间是互不兼容的：

版本及厂家	说明
TDS 4.2 Sybase 和 Microsoft	当 Sybase/Microsoft 分家时，使用该版本。
TDS 5.0 Sybase	由 Sybase 引进。因为 TDS 5.0 协议是可扩展的。因此该协议可能是 Sybase 的最后的版本了。
TDS 7.0 Microsoft	为 SQL Server 7.0 引进。包括支持 SQL Server 7.0 中的扩展数据类型(例如长度大于 255 的 char/varchar 列)。也包括了对 Unicode 的支持。
TDS 8.0 Microsoft	为 SQL Server 2000 引进。包括支持大 integer (64 位的 int)和 "variant" 数据类型。

安装 freetds：

$ ./configure --prefix=/usr/local/freetds --with-iodbc=/usr/local

如果系统已经安装了 iodbc 或 unixodbc，可以加上选项支持 iodbc（--with-iodbc）或 unixodbc（--with-unixodbc）。

$ make

$ su root

Password:

$ make install

产品	支持版本	说明
Sybase System 10 以前，Microsoft SQL Server 6.x	4.2	仍然能够和所有的产品一起工作，只是有局限。
Sybase System 10 及以上	5.0	Sybase 当前使用的协议。
Sybase System SQL Anywhere	仅 5.0	源于 Watcom SQL Server，它基于一个完全不同的代码基础。我们的 SQL Anywhere 在版本 5.5.03 使用 OpenServer Gateway 第一次实现了支持 TDS，而在版本 6.0 实现了本地支持。
Microsoft SQL Server 7.0	7.0	包括支持 SQL Server 7.0 中的扩展数据类型(例如长度大于 255 的 char/varchar 列)。也包括了对 Unicode 的支持。
Microsoft SQL Server 2000	8.0	支持大 integer (64 位的 int) 和所有列上的 variant 和 collation 数据类型。FreeTDS 对 8.0 的支持仍然不太成熟。它不支持变量，没有广泛的使用 collation。如果在使用中碰到了问题，请使用 TDS 7.0 (或 4.2)。

FreeTDS 使用的配置文件叫做 freetds.conf。该文件缺省在/usr/local/etc 目录下：

可以通过 FreeTDS 附带的命令行工具软件 tsql 检查配置是否成功，tsql 的语法是：

tsql [-S server [-H hostname | -p port]] -U username [-P password]

例如：

>/usr/local/freetds/bin/tsql -S MyServer70 -U sa

5.7 Oracle 数据库

ORACLE 数据库有两个相关模块：DBD::Oracle 和 Oracle::OCI。前者用于一般性用途，后者对 ORACLE 数据库提供了精细的控制。

5.7.1 DBD::Oracle

如下是一个简单的测试程序：

```
use DBD::Oracle;
$dbh = DBI->connect("dbi:Oracle:dwdb", 'dwdb', 'dwdb', {AutoCommit=>0}) || die 'Could not connect to database.';
print "connect ok!\n";

my $sth = $dbh->prepare("select 1 from dual");
$sth->execute;
my ($test) = $sth->fetchrow();

$sth->finish;

if ($test == 1)
{
    print "select ok!\n" ;
}
else
{
    print "select error!\n";
}

$dbh->disconnect;
```

建立连接

典型的连接到 Oracle 数据库的方式有两种：

连接方式	例子
使用 TNS	$dbh = DBI->connect(”dbi:Oracle:tnsname”, $username,$password);
不使用 TNS 名称	$dbh = DBI->connect(”dbi:Oracle:host=foo.com;sid=ORCL” , $username,$password);

	或 $dbh = DBI->connect(”dbi:Oracle: (DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(foo.com)(PORT=1526))) (CONNECT_DATA=(SID=ORCL)))” , $username,$password);

connect 函数中可设置的连接参数有：

连接参数	例子
ora_session_mode	用 OPER 或 DBA 模式建立连接： $dbh = DBI->connect(”dbi:Oracle:”, $username,$password, { ora_session_mode => $mode }); 这里 $mode = 2 代表 SYSDBA，或 = 4 代表 SYSOPER。
ora_module_name	用于给 SESSION 起一个名称： $dbh = DBI->connect(”dbi:Oracle:”, $username,$password, { ora_module_name => $0 });

绑定参数

缺省方式下 DBD::Oracle 把任何东西都绑定成字符串，包括数字。

可以在 ORACLE 中声明的特定的绑定类型有 LONG/LOB，CURSOR 和定长的 CHAR 值。例子如下：

```
use DBD::Oracle qw(:ora_types);
$sth = $dbh->prepare("UPDATE tablename SET foo=? WHERE bar=?");
$sth->bind_param(1, "dummy", { ora_type => ORA_CLOB });
$sth->bind_param(2, "dummy", { ora_type => ORA_CHAR });
$sth->execute(@$_) foreach (@updates);
```

绑定字符串也可以这样，{ TYPE=>SQL_CHAR }。

返回 cursors 对象

```
$sth = $dbh->prepare("BEGIN :csr := func_returning_cursor(:arg); END;");
$sth->bind_param(":arg", $arg);
$sth->bind_param_inout(":csr", \my $sth2, 0, { ora_type=>ORA_RSET } );
$sth->execute;
my $data = $sth2->fetchall_arrayref;
```

但不是所有的生成 cursor 的方式都支持。而且 cursor 需要显式的关闭，如下所示：

```
$sth3 = $dbh->prepare("BEGIN CLOSE :cursor END");
$sth3->bind_param_inout(":cursor", \$sth2, 0, { ora_type => ORA_RSET } );
$sth3->execute;
```

绑定返回值

能使用 Oracle 的 RETURNING 子句。

```
$sth = $dbh->prepare(q{       UPDATE anothertable SET X=? WHERE Y=? RETURNING Z});
$sth->bind_param(1, $x);
$sth->bind_param(2, $y);
$sth->bind_param_inout(3, \$z, 100);
$sth->execute;
```

但是当前绑定返回值仅支持返回单个值，即仅更新一行。如下是一个绑定 date 类型的例子：

```
use DBD::Oracle;
use DBI qw(:sql_types);

my $dbh = DBI->connect("dbi:Oracle:dwdb", 'dwdb', 'dwdb', {AutoCommit=>0}) || die 'Could not connect to database.';

my $sth = $dbh->prepare("ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD'");
$dbh->{RaiseError} = 1; # 如果这里发生了错误就不再继续处理

$sth->execute();

$sql ="insert into F_QXXX_SS (TQ,RQ_ID) values (?,?)";

@bind_variables=('newe','2003-03-03');

$sth = $dbh->prepare($sql);

$sth->execute(@bind_variables);

$sth->finish;

$dbh->disconnect;
```

如下程序调用一个存储过程，有一个输入参数，没有返回值。假设存储过程不调用 commit。注意，其中使用了 eval 块：如果 Oracle procedure 返回一个异常，就会导致 Perl 程序调用 die，错误信息放在$@ 和$DBI::errstr 中。

```
use strict;
use DBI;
```

```
my $dbh = DBI->connect(
    'dbi:Oracle:orcl',
    'jeffrey',
    'jeffspassword',
    {
        RaiseError => 1,
        AutoCommit => 0
    }
) || die "Database connection not made: $DBI::errstr";

eval {
    my $func = $dbh->prepare(q{
        BEGIN
            jwb_function(
                parameter1_in => :parameter1
            );
        END;
    });

    $func->bind_param(":parameter1", 'Bunce'); #使用命名的占位符很方便。
    $func->execute;

    $dbh->commit;
};

if( $@ ) {
    warn "Execution of stored procedure failed: $DBI::errstr\n";
    $dbh->rollback;
}

$dbh->disconnect;
```

这个程序调用一个有一个返回值的存储过程。为了从函数返回一个值，我们使用 bind_param_inout 绑定一个占位符。当时用此方法，我们必须告诉 DBD::Oracle 为返回值分配多少字节 。

```
use strict;
use DBI;

my $dbh = DBI->connect(
    'dbi:Oracle:orcl',
    'jeffrey',
    'jeffspassword',
    {
```

```
            RaiseError => 1,
            AutoCommit => 0
        }
) || die "Database connection not made: $DBI::errstr";

my $rv; #用这个变量保存 Oracle 存储过程的返回值
eval {
        my $func = $dbh->prepare(q{
            BEGIN
                :rv := jwb_function(
                    parameter1_in => :parameter1
                );
            END;
        });

        $func->bind_param(":parameter1", 'Bunce');
        $func->bind_param_inout(":rv", \$rv, 6);
        $func->execute;

        $dbh->commit;
};

if( $@ ) {
        warn "Execution of stored procedure failed: $DBI::errstr\n";
        $dbh->rollback;
}

print "Execution of stored procedure returned $rv\n";

$dbh->disconnect;
```

row cache 调优

Oracle OCI 支持透明的客户端 row cache，缺省仅 2 行。它把网络循环减半，当选择一行时特别有好处。

DBD::Oracle 走的更远一点，它能够自动度量 row cache 到正好适合 10 个以太包，而且基于一个估计的平均行宽度和 SQL*Net 协议。

但是，有时候也需要调整，例如 你正在查询大量数据，正好有一个大的缓存，或者估计的平均行宽度不准确 (可打开 trace 查看它)。

手工调优方法如下：

$dbh->{RowCache} = $n;

这里 $n > 0 声明行数，$n < 0 声明使用的内存数。

5.7.2 Oracle::OCI

用途

通过使用 OCI 能够实现 DBD::Oracle 不能实现的如下功能：

- 大块读写大数据对象 LOB。包括通过回调函数流式化存取 LOB。
- 创建和操作 collections、iterators、用户定义类型、cursors, 变长数组、嵌套表等。
- 多个用户共享同样的数据库连接，可用于 web server。
- 多个进程共享同一事务，大批量数据加载时很有用。
- OCI 函数调用的非阻塞模式，可用于 GUI。
- 通过数组或 Direct Path Loading 批加载。
- 以最详细的方式描述模式元数据。
- 使用 Oracle 的数据操作和格式化工具：dates, numbers, character set conversions 等。
- 高级队列：包括 Publish / Subscribe 和异步 event notifications。
- 在一个调用中返回相关对象树。
- 管理并行服务器的宕机重起。
- 线程安全。

Oracle::OCI 模块当前不能在 windows 平台安装使用，可以在 unix 上使用。

如下是一个从 DBI 句柄取得 OCI 属性的例子：

```
use DBI;
use Oracle::OCI qw(:all);
$dbh = DBI->connect("dbi:Oracle:", $user, $password);
OCIAttrGet($dbh, OCI_HTYPE_handle, my $attrib_value, 0, OCI_ATTR_name, $dbh);
```

处理大对象

首先使用 DBI 取得一个 LOB 'locator' (不是内容)。

```
my $lob_locator = $dbh->selectrow_array(
"select my_lob from table_name where id=1 for update",
{ ora_auto_lob => 0 }  # 返回 LOB locator 而不是内容
);
```

然后使用 Oracle::OCI 操作它。

```
OCILobGetLength($dbh, $dbh, $lob_locator, my $lob_len=0);
OCILobTrim($dbh, $dbh, $lob_locator, $lob_len - 2);
```

获取，编辑和更新其中的一些字节。

```
my ($offset, $amount, $buffer) = ($lob_len/2, 44, '');
OCILobRead($dbh, $dbh, $lob_locator,
$amount, $offset, oci_buf_len($buffer, 200, \$amount), 0,0, 0,0 );
$buffer =~ s/ATGC/ACTG/g;
OCILobWrite($dbh, $dbh, $lob_locator, $amount, $offset,
oci_buf_len($buffer), OCI_ONE_PIECE, 0,0, 0, 1 );
```

5.8 Sybase 数据库

Sybase 有两种 c 语言编程接口：OpenClient 和 DBClient。因为 DBClient 是一种比较老的接口，而且 Sybase 已经不再升级 DBClient，因此推荐采用 OpenClient。

有两类模块可以和 Sybase 数据库打交道：sybperl 和 DBD-Sybase。DBD-Sybase 和 sybperl 都是构建在 OpenClient 之上的。

sybperl 是一个模块集合的简称，这个模块集合使用 Sybase::名字空间。sybperl 包含四个模块：Sybase::DBlib，Sybase::CTlib，Sybase::BCP 和 Sybase::Sybperl。头两个分别实现了 Sybase DB-Library 和 Client Library API 的简单包装。Sybase::BCP 是一个特殊的模块，其目的是做 Bulk-Copy 操作，而 Sybase::Sybperl 是一个与 sybperl 1.xx (即 perl 4.x 版本)兼容的模块。

其中 Sybase::CTlib 能够异步调用数据库请求，能够有限的调用 bulk-copy API 等，所以它最引人注目。

DBD-Sybase 是符合 DBI 接口的 Sybase 数据库驱动。一般推荐使用 DBD-Sybase 模块，因为它有更好的可移植性。

但是有一些领域 DBI API 和 Sybase 不是特别匹配。一个例子是，Sybase (以及 MS-SQL) 能够返回从单个存储过程中返回列类型不一样的多个结果集。为了处理这种情况，perl 代码

必须包含附加的循环保证所有的结果集已经得到处理：

```
$sth = $dbh->prepare("exec some_proc $foo, $bar");
$sth->execute;
do {
    while($d = $sth->fetch) {
....
}
}
while($sth->{syb_more_results});
```

另外一个问题是 AutoCommit。DBD::Sybase 处理 AutoCommit off 以两种方式：它可以使用"chained transaction"模式，或者当第一个请求发出时，发出一个"begin transaction" ，但是两者都有副作用。在"chained transaction"模式，如果你恰好执行了一段代码发出了一个显式的"begin tran"，你会得到一个错误，而在另外一个模式下，如果执行任何 DDL 语句(包括"select into")，你都会得到一个错误，因为锁住系统表问题。

5.8.1 DBD-Sybase

首先注意，不要把不同版本的 OpenClient 安装到一起。否则 DBD-Sybase 初始化时会出现错误：

The context allocation routine failed.

The following problem caused the failure:

Invalid context version.

DBD-Sybase 模块的使用样例如下：

```
use DBI;
use strict;

my $Srv='ubis_ase';
my $Db='buffer';

my $dbh = DBI->connect("dbi:Sybase:server=$Srv;database=$Db", 'sa', 'sasasa');

$dbh->do("create table slav(one int primary key, two int)");
$dbh->{AutoCommit} = 0;
my $sth = $dbh->prepare("insert slav values(?, ?)");
for (1 .. 10) {
    my $ret = $sth->execute($_ % 8, $_);

    if (!$ret) {
```

```
    print "Failed with ", $dbh->err, "\n";
    die $dbh->errstr;
    }
    else
    {
    print "\$ret = $ret\n";
    }
}

$dbh->commit;

my $ret =$dbh->do("drop table slav");
if (!$ret) {
    print "Failed with ", $dbh->err, "\n";
    die $dbh->errstr;
}
```

建立连接

SYBASE 中的逻辑服务器名称和物理的 IP 地址，端口号的对应关系保存在 sql.ini 文件中（或 interfaces 文件，如果是 UNIX 平台）。OpenClient （DBD-Sybase 使用的接口）使用环境变量 SYBASE 指定的路径查找 sql.ini 文件。SYBASE 环境变量是 Sybase 的安装路径(例如：’/usr/local/sybase’)。如果你需要在你的脚本中设置，你必须在 BEGIN 块中指定：

```
BEGIN {
$ENV{SYBASE} = ’/opt/sybase/11.0.2’;
}
$dbh = DBI->connect(’dbi:Sybase:’, $user, $passwd);
```

建立连接时，可声明一个或多个连接参数，用分号分隔。

连接参数	说明
server	DBD::Sybase 连接的服务器名称，缺省值是“SYBASE”，可以用两种方式指定。第一种方式是设置“DSQUERY”环境变量： $ENV{DSQUERY} = "ENGINEERING"; $dbh = DBI->connect(’dbi:Sybase:’, $user, $passwd); 或者，你可以连接时传递服务器名称，如： $dbh = DBI->connect("dbi:Sybase:server=ENGINEERING", $user, $passwd);

连接参数	说明
database	声明缺省数据库，例如： $dbh = DBI->connect("dbi:Sybase:database=sybsystemprocs",$user, $passwd); 它等价于： $dbh = DBI->connect('dbi:Sybase:', $user, $passwd); $dbh->do("use sybsystemprocs");
charset	声明客户端使用的字符集。 $dbh = DBI->connect("dbi:Sybase:charset=iso_1",$user, $passwd);
language	声明客户端使用的语言，例如： $dbh = DBI->connect("dbi:Sybase:language=us_english",$user, $passwd);
packetSize	连接使用的网络包大小。使用大的网络包大小能够增加某些类型查询的性能。要让服务器支持这个特性，请参见 Sybase 文档。 $dbh = DBI->connect("dbi:Sybase:packetSize=8192",$user, $passwd);
interfaces	声明一个可选的接口文件的位置，例如： $dbh = DBI->connect("dbi:Sybase:interfaces=/usr/local/sybase/interfaces", $user, $passwd);
loginTimeout	声明 DBI->connect()将要等待服务器响应的秒数。如果服务器在指定秒数以前没有响应，DBI->connect() 调用将因为 timeout 错误而失败。缺省值是 1 分钟，，通常应该足够了，但是对于一个忙的服务器，有时候需要增加该值： $dbh = DBI->connect("dbi:Sybase:loginTimeout=240", # wait up to 4 minutes $user, $passwd);
timeout	声明任何 Open Client 调用将要超时的秒数。一旦收到一个连接上的超时错误，要有进一步的处理的话，连接应该关闭然后再打开。把这个值设置成 0 或一个负值会导致一个无限的 timeout 值。下例设置等待 4 分钟： $dbh = DBI->connect("dbi:Sybase:timeout=240", $user, $passwd);

连接参数	说明
scriptName	声明将要显示在 sp_who 中的连接的名字(即在 sysprocesses 表的“program_name”列)。例如： $dbh->DBI->connect("dbi:Sybase:scriptName=myScript", $user, $password);
hostname	声明将要显示在 sp_who 中的 hostname (即在 sysprocesses 表的“hostname”列)。例如： $dbh->DBI->connect("dbi:Sybase:hostname=kiruna", $user, $password);
tdsLevel	声明当连接到服务器时要使用的 TDS 协议层。可能的值是 CS_TDS_40, CS_TDS_42, CS_TDS_46, CS_TDS_495 和 CS_TDS_50。通常，这是由客户端和服务器自动选择的，但是某些情况下，需要客户端强制指定底层协议。例如： $dbh->DBI->connect("dbi:Sybase:tdsLevel=CS_TDS_42", $user, $password); 注意：设置 tdsLevel 成 CS_TDS_495 会禁止一些功能，如 ?-风格的占位符和 CHAINED 的非 AutoCommit 模式等。

绑定参数

DBD::Sybase 缺省假定所有的参数都是 SQL_CHAR，你必须使用 bind_param()指定参数为其它类型。然后，DBD::Sybase 会记住这个参数的类型，当你下次使用时，就不用再指定了：

```
my $sth = $dbh->prepare("exec my_proc \@p1 = ?, \@p2 = ?");
$sth->bind_param(1, 'one', SQL_CHAR);
$sth->bind_param(2, 2.34, SQL_FLOAT);
$sth->execute;
....
$sth->execute('two', 3.456);
....
```

当前不支持 TEXT 或 IMAGE 类型的绑定。

存储过程

存储过程的调用和普通 SQL 一样，能够返回和 SELECT 同样类型的结果集(即存储过程的一个 SELECT 能够使用$sth->fetch 检索)。

如果存储过程通过 OUTPUT 参数返回数据，必须首先声明这些输出参数：

```
$sth = $dbh->prepare(qq[
        declare \@name varchar(50)
        exec getName 1234, \@name output
]);
```

存储过程不能使用绑定参数（?）的方法调用，如下是非法的：

```
$sth = $dbh->prepare("exec my_proc ?");
$sth->execute('foo');
```

可以使用下面的方法调用：

```
$sth = $dbh->prepare("exec my_proc 'foo'");
$sth->execute;
```

因为 Sybase 存储过程往往返回多于一个结果集，所以应该使用一个循环保证 syb_more_results 是 0：

```
do {
    while($data = $sth->fetch) {
        ...
    }
}
while($sth->{syb_more_results});
```

5.9 PostgreSQL 数据库

5.9.1 PL/perl

PL/Perl 是一个可加载的过程语言。允许你用 Perl 脚本语言书写可以用于 SQL 查询语句的函数，就好象它们是内建在 Postgres 里一样。这样扩展了 SQL 语句的功能。

例如，假设你有下面的一个员工表：

```
CREATE TABLE EMPLOYEE (

    name text,

    basesalary int4,

    bonus int4 );
```

下面这个函数获取员工所有的薪水（本金+奖励）：

```
CREATE FUNCTION totalcomp(int4, int4) RETURNS int4
```

```
AS 'return $_[0] + $_[1]'

LANGUAGE 'plperl';
```

请注意参数会象想象的那样以 @_ 形式传递给函数．还有，因为 SQL 创建函数的引号规则，你会发现你要比平常更频繁地使用扩展的引号函数（qq[]，q[]，qw[]）。

我们现在可以象下面这样使用我们的函数：

```
SELECT name, totalcomp(basesalary, bonus) from employee
```

我们还可以把表的整个记录传递给函数：

```
CREATE FUNCTION empcomp(employee) returns int4

AS 'my $emp = shift;

    return $emp->{'basesalary'} + $emp->{'bonus'};'

LANGUAGE 'plperl';
```

记录是作为一个散列（哈希 hash）数组的引用传递的。脚标是记录里面的字段名称，值是记录里面对应的字段的值。也许从函数返回多个值是有用的，这些值可以放在复合类型中，但是当前还不支持返回复合类型。

新函数 empcomp 可以这样使用：

```
SELECT name, empcomp(employee) from employee;
```

PL/Perl 解释器是一个完整的 Perl 解释器。不过，有些操作出于安全性的考虑被屏蔽掉了。通常，被限制的操作是那些与外部环境交互的动作。包括文件句柄操作，请求和使用（对于外部模块而言）。要知道，安全不是绝对的．实际上，有几种拒绝服务攻击仍然是可能的－存储器耗尽和无限循环就是两个例子。

安装方法

假设 Postgres 源代码树的根是 $PGSRC，那么 Pl/perl 源代码位于 $PGSRC/src/pl/plperl。

要制作，只需要按照下面步骤处理：

```
$cd $PGSRC/src/pl/plperl
```

$perl Makefile.PL

$make

这样做将创建共享库文件．在一个 Linux 系统里，它将被叫做 plperl.so。在其他系统里的扩展名可能不同。

然后应该把该共享库拷贝到 postgres 安装的 lib 子目录下。

createlang 命令用于将该语言安装到一个数据库中。如果它被安装到 template1 里面，以后创建的所有数据库将自动安装这个语言。

5.10 MySQL

5.10.1 DBD::mysql

下面这个使用 Mysql 模块的例子打印出 Mysql 服务器上的所有数据库：

```
use DBD::mysql;

$dsn = "DBI:mysql:database=mysql";
$user = "sa";
$password = "";

$dbh = DBI->connect($dsn,$user,$password);
# 以 $dsn $user $password 等参数登录数据库。

@dbs = $dbh->func('_ListDBs');

foreach  (@dbs)
{
    print "database name : $_\n" ;
}

print "database count ".scalar(@dbs);
```

建立连接

如果通过 Shell 来连接到 MySQL，通常要求用户输入 mysql 命令时跟上用户名和密码，在有些情况下需指定主机名。

命令如下：

mysql -u username -p password

登陆之后，在访问数据表之前，必须列出欲使用的数据库的名称，方式如下：

USE bookstore;

当不使用 Perl DBI 模块的时候，要实现这两个简单的任务也会变得很复杂。使用 Perl DBI 模块来建立与数据库 bookstore 的连接，仅需在 Perl 脚本中填入如下代码：

```
use DBI;
my $dbh = DBI->connect ("DBI:mysql:bookstore:localhost",
                        "username","password")
        || die "Could not connect to database: "
         . DBI-> errstr;
```

第一行代码，调用 DBI 模块，通常，该行置于脚本的开始部分。第二行（在版面上占据超过了一行，但实际是一行代码）创建了一个数据库的句柄，并指明采用的数据库引擎（在这里是 mysql）、数据库的名称（bookstore）、主机名(localhost)、用户名及密码。接下来的一行，||（或运算符）提供了当连接失败时可选择的执行操作。也即，执行脚本退出并输出 Perl 所产生的错误信息，“.”操作符用来将输出信息合在一起。如果是在 Shell 下执行脚本，错误信息将输出到显示器屏幕（标准输出设备）。如果是在 Apache 下（也即是在浏览器下）执行，错误信息将记到 Apache 的错误日志上，通常是在 /var/log/httpd/error_log。

上面的代码，只是建立了数据库的句柄，只有同 SQL 语句关联并执行 SQL 语句才能得到我们想要的操作（操作数据库中数据），你可以加上以下的代码：

```
my $sql_stmnt = "SELECT book_id, title, publisher
                 FROM books
                 WHERE author = 'Olympia Vernon'";
my $sth = $dbh->prepare($sql_stmnt);
$sth->execute();
```

第一行（以分号“;”结尾）创建了一个变量($sql_stmnt)来存放 SQL 语句，在本例中，使用一个简单的 SELECT 语句来从一系列书的目录下找到作者是 Olympia Vernon 的书。第二行通过将这个 SQL 语句($sql_stmnt)与前面建立的数据库句柄($dbh)联系起来，构建一个 SQL 句柄($sth)。最后第三行代码利用 DBI 模块的 execute()函数来执行这个 SQL 句柄。

关于连接参数的一些主要选项有：

连接参数	说明

连接参数	说明
Host 或 hostname port	主机名如果不加声明或者定义成"时，指运行在本机缺省端口(3306)的 MySQL 服务器。如果 MySQL 服务运行在一个非标准的端口，可以通过把 host:port 形式的参数作为 hostname 指定端口。也可以使用 port 参数。例如： $dsn = "DBI:mysql:database=mysql;hostname=localhost:3306";
mysql_client_found_rows	使能 (TRUE value) 或关闭 (FALSE value) 标记 CLIENT_FOUND_ROWS。当连接到 MySQL 服务器时，如果没有打开 mysql_client_found_rows，如果你执行如下语句： UPDATE $table SET id = 1 WHERE id = 1 MySQL 将总是返回 0，因为没有这条语句没有改变任何行。如果打开 mysql_client_found_rows ，就会返回 id 值是 1 的行数。
mysql_compression	在 MySQL 3.22.3 版本或以上，如果设置 "mysql_compression=1"，那么就会压缩通过网络传送的服务器和客户端之间的数据传递。
mysql_connect_timeout	如果在给定的秒数连接请求没有成功就会返回超时错误。例如： $dsn = "DBI:mysql:database=mysql;mysql_connect_timeout=6";

连接参数	说明
mysql_read_default_file mysql_read_default_group	这些选项用于读取一个配置文件，如 /etc/my.cnf 或者 ~/.my.cnf。缺省情况下 MySQL 的 C 客户端库不使用任何配置文件。但是仍然可以声明读取一个配置文件，例如： $dsn = "DBI:mysql:test;mysql_read_default_file=/home/joe/my.cnf"; 选项 mysql_read_default_group 可以声明在配置文件中的缺省组。通常是 client 组，但是请看下面这个例子： [client] host=localhost [perl] host=perlhost 如果使用这个配置文件，缺省就会连接到<localhost>。然而，如果使用： $dsn = "DBI:mysql:test;mysql_read_default_group=perl;" . "mysql_read_default_file=/home/joe/my.cnf"; 就会连接到 perlhost。注意，如果只是声明一个缺省组而不声明一个配置文件，就会读入缺省配置文件。
=item mysql_socket	在 MySQL 3.21.15 版本或以上，可以选择一个连接到服务器的 Unix socket。例如： mysql_socket=/dev/mysql 通常不需要这个选项，除非要使用其他位置的 socket。

连接参数	说明
mysql_ssl	当连接到 MySQL 数据库时的一个真值打开 CLIENT_SSL 标记： mysql_ssl=1 这样会加密和服务器间的通讯。 当打开 mysql_ssl 时，如下这些标记是可选的： mysql_ssl_client_key mysql_ssl_client_cert mysql_ssl_ca_file mysql_ssl_ca_path mysql_ssl_cipher
mysql_local_infile	在 MySQL 3.23.49 或以上版本，加载本地数据的能力缺省是关闭的。如果 DSN 参数中包括： mysql_local_infile=1 就会打开 LOAD DATA LOCAL。即使服务器关闭 LOCAL，这个选项也是有效的。

取得数据

已经连接到 MySQL 并且调用了 SQL 语句，现在你就可以获得数据结果了。MySQL 将请求的数据结果以行和列的形式返回给 Perl，形式和利用 mysql 客户端获得的一样，但没有按照表格的格式。在 Perl 中，每一条记录中的任何一项都是作为数组的一个成员，每行是一个数组。对每个数组，可以用变量得到每个单元的值，在接收或者处理下一记录之前，用来打印变量的值或者进行其他操作。我们通过 while 循环来进行操作，代码如下：

```
while (my($book_id, $title, $publisher) = $sth->fetchrow_array() {

    print $q->a({href=>"book_detail.cgi?book_id=$book_id"},

                "$title ($publisher)");

}
```

这段代码的核心是 DBI 模块的 fetchrow_array()函数。正如函数的名称代表的那样，它取得每一行或者一组字段，每一次获得一行。While 语句执行到没有记录时结束。数组里每一单元的值存放在$book_id, $title, $publisher 这 3 个变量中。然后在 CGI 输出中可以将变量的值打印出来(未在上述代码中展示)。代码中$q->a({...},"...")是用来创建超连接文本的语法。超连接（href）包含文本连接的 CGI 脚本，在记录标识数 book_id 之后，跟着的是要显示的文本（也即，放在“()”中的书名和出版商名称）。这是一种简单的方法来获取 MySQL 中的数据，在 Perl 中还可以用其他方法来实现类似功能。

在开始讲更复杂的方法之前，需要指出开始跳过的两个内容，第一，使用 CGI 模块，必须先在脚本中声明它，通常在脚本的开头处比较合适，代码如下：

```
use CGI;
my $q = new CGI;
```

第一行声明调用 CGI 模块，CGI 模块是 Perl 的默认模块。第二行基于该模块创建了一个新对象，通过变量$q 来引用和调用，顺便提一下，任何变量名称都可以。

然后，必须在数据访问完成之后结束 MySQL 的会话，只需要敲入以下代码即可：

```
$sth->finish;
$dbh->disconnect;
```

第一行关闭 SQL 语句句柄，只要不断开与 MySQL 的连接(如在第二行作的那样)，不需要重新连接到 MySQL 就可以执行新的 SQL 语句，如果连接持续不操作时间比较长，MySQL 自己会释放该连接,来减少系统资源的消耗。当操作完成了终止会话是一个比较好的做法。

完整的例子

一个比上一节介绍的方法更有效的从MySQL获得数据的方法是通过将所有数据放置在内存中以供将来使用，这样就可以在处理和显示这些数据之前关闭与 MySQL 的连接。保留与 MySQL 的连接，在处理每一条记录并从一个复杂的数据结构中传递数据（一个数组的数组）将很大的影响脚本的效率。因此，将数据放在内存中，传递它在内存地址的一个引用通常更好一些。所以，不再是调用 fetchrow_array()，现在调用的是 fetchall_arrayref()。正如函数的名称的含义，它一次获得所有的数据，把它们放在一个数组里，获得它在内存中的地址引用。为使读者能完整的了解，以下给出完整的例子，同以前章节中给出的代码不同在于采用了 fetchall_array()和相应的改变。

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

# Load Modules
use DBI;
use CGI;

my $q = new CGI;
```

```
# Connect to MySQL
my $dbh = DBI->connect("DBI:mysql:bookstore:localhost",
                       "username","password")
          || die "Could not connect to database: "
          . DBI->errstr;

my $sql_stmnt = "SELECT book_id, title, publisher
                 FROM books
                 WHERE author = 'Olympia Vernon'";
my $sth = $dbh->prepare($sql_stmnt);
$sth->execute();

# Retrieve results and reference number
my $results = $sth->fetchall_arrayref();
$sth->finish();
$dbh->disconnect();

# Web page

print
    $q->header(-type=>'text/html'),
    $q->start_html;

    # Loop through array of arrays

    foreach my $record (@$result){

        # Parse each record array and display
        my ($book_id, $title, $publisher) = @$record;
        print $q->a({href=>"book_detail.cgi?book_id=$book_id"},
              "$title ($publisher)"), $q->br, "\n";
    }

    print $q->end_html;

    exit;
```

SQL 语句执行的结果存储在内存中而不是通过在循环控制语句中嵌入 fetch 函数，结果数据的地址存放在引用变量$results 中，同 MySQL 的连接随后释放。在 Web 页面开始的时候，作一个 foreach 循环来从整个数组中提出每一条记录（每一行）。每一条记录然后在分拆为单独的变量，随后用变量的值建立了一个超连接。

管理函数

DBD::mysql 中独有的一些特殊的函数有：

函数	说明
createdb	创建数据库，调用形式是： $rc = $drh->func('createdb', $database, $host, $user, $password, 'admin'); 或 $rc = $dbh->func('createdb', $database, 'admin');
dropdb	删除数据库，调用形式是： $rc = $drh->func('dropdb', $database, $host, $user, $password, 'admin'); 或 $rc = $dbh->func('dropdb', $database, 'admin');
shutdown	关闭数据库，调用形式是： $rc = $drh->func('shutdown', $host, $user, $password, 'admin'); 或 $rc = $dbh->func('shutdown', 'admin');
reload	重新装载数据库，调用形式是： $rc = $drh->func('reload', $host, $user, $password, 'admin'); 或 $rc = $dbh->func('reload', 'admin');

5.11 ODBC

5.11.1 iODBC

DBD:ODBC 是联系 ODBC 和 DBI 的桥梁。

unix 一般不会自带 ODBC 驱动管理程序，而 iodbc 是一个开放源码，应用广泛的 ODBC 驱动管理程序。可以从 http://www.iodbc.org/得到它。它缺省安装在 /usr/local/bin 目录下。

/etc/odbcinst.ini

/etc/odbc.ini

下面以 Sybase IQ12.5 为例，说明使用情况。在 UNIX 下会碰到一些 windows 碰不到的问题。例如驱动程序 64 位和 32 位的问题。64 位的应用程序，包括许多第三方工具，可以使用 64 位的 ODBC 驱动连接到 64 位的数据库服务器。32 位的应用程序可以使用 32 位的 ODBC 驱动连接 64 位的数据库服务器。但是，32 位的应用程序不能使用 64 位的驱动程序连接到 64 位的数据库服务器。

安装好驱动程序后，根据驱动程序的说明，配置环境变量：

export ODBCINI=/etc/odbc.ini

export LD_LIBRARY_PATH=/tmp/12.5-odbc/ASIQ-12_5-odbc/lib:/iq/ASIQ-12_5/lib

参照 windows 注册表中的相关配置，odbc.ini 文件内容如下例：

[ubis_iqiq]

AutoStop=YES

CommLinks=TCPIP{host=130.59.1.15:6600},SharedMemory

Driver=/iq/ASIQ-12_5/lib/dbodbc7.so

EngineName=ubis_iqiq

PWD=SQL

UID=DBA

Intergrated=NO

我们可以通过 iodbctest 测试该数据源。

iodbctest "DSN=ubis_iqiq;UID=DBA;PWD=SQL"

返回结果

iODBC Demonstration program

```
This program shows an interactive SQL processor

Driver Manager: 03.51.0001.0908

Driver: 07.00.0004

SQL>select getdate()

 result set 1 returned 1 rows.

SQL>quit
```

表明已经成功连接上。

DBD::ODBC 的安装方法和其他模块一样，但 DBD::ODBC 1.06 有一个小 bug。生成的 make 文件在第 312 行有错误。应该把@$(NOOP)前面的空格改成制表符。

```
config :: $(changes_pm)

        @$(NOOP)
```

要测试该模块需要设置三个环境变量：

```
export DBI_DSN=dbi:ODBC:your_dsn

export DBI_USER=DBA

export DBI_PASS=SQL
```

第6章 调试

这一章主要介绍保证程序正常运行的两种技术：单元测试与异常处理。

6.1 单元测试

在软件开发中软件开发中，单元测试是从代码编写完成到整体测试中间的很重要一个过渡步骤。

开发一个可重用的模块时，它几乎是一个必需的过程。一个可重用的模块的开发过程一般是：

- 设计模块；
- 编写模块使用模块的用户代码；
- 编写单元测试脚本；
- 编写模块代码；
- 进行单元测试，验证模块正确性。

其中编写模块代码往往是增量进行的，因此后两个步骤往往需要重复多次才能最终完成一个模块的开发。

如下是一个基本的测试脚本：

```
print "1..1\n";

print 1 + 1 == 2 ? "ok 1\n" : "not ok 1\n";
```

因为 1 + 1 = 2, 它打印出：

```
1..1

ok 1
```

它说的意思是：

- 1..1：就要运行一个测试。

- ok 1：第一个测试通过了。

你的基本测试单元是 ok。对每个你测试的东西，成功则打印出一个 ok，否则打印出一个 not ok。如果自己写这些打印语句会很麻烦，幸好有 Test::Simple 和 Test::Unit 这样的模块。

6.1.1 Test::Simple 与 Test::More

Test::Simple 与 Test::More 有 100%的向后兼容。也就是说，你能在你的程序中完全使用 Test::More 替换 Test::Simple。Test::More 是 Test::Simple 的一个超集。Test::Simple 只有一个函数 ok()。

Test::Harness 解释你的测试结果决定是否成功或失败。

```
use Test::Simple tests => 1;
ok( 1 + 1 == 2 );
```

像上面的代码一样做同样的事情。ok()是 Perl 测试的中心。如果 ok()返回一个 true，则表示测试通过了，否则表示失败了。

```
use Test::Simple tests => 2;
ok( 1 + 1 == 2 );
ok( 2 + 2 == 5 );
```

这段代码输出：

```
1..2

ok 1

not ok 2
```

测试的输出解释如下：

- 1..2：将要运行两个测试。号码用来保证你的测试程序运行时没有死掉或跳过某些测试。
- ok 1：第一个测试通过了。
- not ok 2：第二个测试失败了。

Test::Simple 友好的打印出一些关于你的测试的额外的注释。

我们将要给出一个测试一个模块的例子。在我们的例子中，我们将测试一个日期库，Date::ICal。可以从 CPAN 上下载它。要安装这个模块还必须先安装因此下载一个拷贝然后接着做。

从哪里开始

测试最难的部分是，要从哪里开始？人们经常被测试一整个模块的巨大任务所吓倒。最好的出发地是程序开始处。Date::ICal 是一个面向对象的模块，意味着你从创建一个对象开始。因此我们测试 new()。

```
use Test::Simple tests => 2;
use Date::ICal;
my $ical = Date::ICal->new;          # 创建一个对象
ok( defined $ical );                 # 检查对象是否创建成功
ok( $ical->isa('Date::ICal') );      # 类名是否正确
```

运行它，你应该得到：

```
1..2

ok 1

ok 2
```

好的，现在你已经写出了第一个有用的测试了。

名字

输出不具有描述性，是吗？当你只有两个测试时，你能指出哪一个是#2，但是如果你有 102 个呢？

ok()函数的第二个参数能够让每个测试有一个描述性的名字。

```
use Test::Simple tests => 2;
ok( defined $ical,                  'new() returned something' );
ok( $ical->isa('Date::ICal'),     "   and it's the right class" );
```

因此，现在你会看见：

```
1..2

ok 1 - new() returned something

ok 2 -      and it's the right class
```

测试手册

建立一个完整测试的最简单的方法是按照手册说明的测试功能测试。让我们从 Date::ICal Date::ICal 手册大纲中拿出一些东西，进行全面的测试。

```
use Test::Simple tests => 8;
use Date::ICal;
$ical = Date::ICal->new( year => 1964, month => 10, day => 16,
                         hour => 16, min => 12, sec => 47,
                         tz => '0530' );
ok( defined $ical,                'new() returned something' );
ok( $ical->isa('Date::ICal'), "   and it's the right class" );
ok( $ical->sec   == 47,           '   sec()'   );
ok( $ical->min   == 12,           '   min()'   );
ok( $ical->hour  == 16,           '   hour()'  );
ok( $ical->day   == 17,           '   day()'   );
ok( $ical->month == 10,           '   month()' );
ok( $ical->year  == 1964,         '   year()'  );
```

运行它，你会得到：

```
1..8

ok 1 - new() returned something

ok 2 -    and it's the right class

ok 3 -    sec()

ok 4 -    min()

not ok 5 -    hour()

#         Failed test (test.pl at line 10)

not ok 6 -    day()

#         Failed test (test.pl at line 11)

ok 7 -    month()

ok 8 -    year()

# Looks like you failed 2 tests of 8.
```

哎呦，错了！Test::Simple 让我们知道那一行发生了错误，但是没有更多的了。我们预期得到 17，但是却没有得到。我们得到什么了呢？不知道。我们必须在调试器中再运行测试或者写一些打印语句找到。

但是我们不这样做，我们从 Test::Simple 切换到 Test::More。Test::More 能够实现 Test::Simple 做的每一件事情，而且更多！事实上，Test::More 和 Test::Simple 的实现方法是一样的。你可以等价的把 Test::Simple 替换成 Test::More。这正是我们要做的事情。

Test::More 比 Test::Simple 做的更多。在这里最重要的差别是它比打印``ok"提供更多的信息量的方法。尽管你可以使用一个通用的 ok()写几乎任何测试，但是它不能告诉你什么出错了。我们使用 is()函数，它让我们声明支持的东西和某个东西等价：

```
use Test::More tests => 8;
use Date::ICal;
$ical = Date::ICal->new( year => 1964, month => 10, day => 16,
                         hour => 16, min => 12, sec => 47,
                         tz => '0530' );
ok( defined $ical,                'new() returned something' );
ok( $ical->isa('Date::ICal'), "   and it's the right class" );
is( $ical->sec,       47,         '   sec()'     );
is( $ical->min,       12,         '   min()'     );
is( $ical->hour,      16,         '   hour()'    );
is( $ical->day,       17,         '   day()'     );
is( $ical->month,     10,         '   month()'   );
is( $ical->year,      1964,       '   year()'    );
```

$ical->sec 是 47 吗?$ical->min 是 12 吗?用 is()替换，你可以得到一些更多的信息。

```
1..8

ok 1 - new() returned something

ok 2 -    and it's the right class

ok 3 -    sec()

ok 4 -    min()

ok 5 -    hour()

not ok 6 -    day()

#         Failed test (- at line 16)

#                 got: '16'
```

```
#          expected: '17'

ok 7 -     month()

ok 8 -     year()

# Looks like you failed 1 tests of 8.
```

让我们知道$ical->day 返回 16，但是我们需要 17。一个快速的检查显示代码工作正常，我们在写测试时犯了一个错误。把它改成：

```
is( $ical->day,        16,          '   day()'     );
```

一切正常了。

因此任何时候你正在做一个``这个等于那个"此类的测试时，使用 is()。它甚至能在数组上工作。这个测试总是在标量上下文中，因此你可以测试一个数组中有多少元素用这种方法

```
is( @foo, 5, 'foo has 5 elements' );
```

有时候测试本身错了

它提供给我们一个非常重要的教训。是代码就会有错误，而测试也是代码。因此，测试会有 bug。一个失败的测试可能意味着代码中的错误，但是不会发现测试是错的这种可能。

在另一方面，不要轻易的声明一个测试是不正确的仅仅因为你不能发现错误。不应该轻易让一个测试失效，也不要把它用作逃避工作的托词。

测试许多值

在这里，我们将要测试很多值，用很多边界值测试代码。它在 1970 年以前能正常运行吗?2038 年以后呢? 1904 年以前呢?一万年以后会有问题吗?它能正确处理润年吗?我们能够我们能够循环上面的代码，我们可以用一个循环测试。

```
use Test::More tests => 32;
use Date::ICal;
my %ICal_Dates = (
                  # An ICal string        And the year, month, date
                  #                           hour, minute and second we expect.
                  '19971024T120000' =>      # from the docs.
                                            [ 1997, 10, 24, 12,   0,   0 ],
                  '20390123T232832' =>      # after the Unix epoch
                                            [ 2039,   1, 23, 23, 28, 32 ],
```

```
        '19671225T000000' =>      # before the Unix epoch
                                  [ 1967, 12, 25,   0,   0,   0 ],
        '18990505T232323' =>      # before the MacOS epoch
                                  [ 1899,   5,   5, 23, 23, 23 ],
);
while( my($ical_str, $expect) = each %ICal_Dates ) {
        my $ical = Date::ICal->new( ical => $ical_str );
        ok( defined $ical,                "new(ical => '$ical_str')" );
        ok( $ical->isa('Date::ICal'), "   and it's the right class" );
        is( $ical->year,      $expect->[0],       '   year()'   );
        is( $ical->month,     $expect->[1],       '   month()' );
        is( $ical->day,       $expect->[2],       '   day()'     );
        is( $ical->hour,      $expect->[3],       '   hour()'   );
        is( $ical->min,       $expect->[4],       '   min()'     );
        is( $ical->sec,       $expect->[5],       '   sec()'     );
}
```

现在我们可以通过增加%ICal_Dates 关联数组的元素测试日期的分支了。既然测试更多的数据已经很容易了，我们可以考虑一些相关的问题了。一个问题是，每一次我们增加测试时，我们不得不调整

Test::More tests => ## line;

这项工作很让人恼火。我们可以使用 no_plan。这意味着我们正要运行一些测试，而不知道有多少。

```
use Test::More 'no_plan';     # instead of tests => 32
```

现在我们能够仅仅增加测试，而不必指出测试的数目了。

忽略测试项

探究已有的 Date::ICal 测试，我在 t/01sanity.t 中发现了这个。

```
use Test::More tests => 7;

use Date::ICal;

# Make sure epoch time is being handled sanely.

my $t1 = Date::ICal->new( epoch => 0 );

is( $t1->epoch, 0,                "Epoch time of 0" );
```

```
# XXX This will only work on unix systems.

is( $t1->ical, '19700101Z', "    epoch to ical" );

is( $t1->year,    1970,             "    year()"    );

is( $t1->month, 1,                  "    month()" );

is( $t1->day,     1,                "    day()"     );

# like the tests above, but starting with ical instead of epoch

my $t2 = Date::ICal->new( ical => '19700101Z' );

is( $t2->ical, '19700101Z', "Start of epoch in ICal notation" );

is( $t2->epoch, 0,                  "    and back to ICal" );
```

纪元的开始点在大多数非 Unix 的操作系统中都不同。即使 Perl 在大多数部分消除了不同，某些 Perl 版本的确不同。 MacPerl 是其中之一。我们知道它永远不会在 MacOS 上成功运行。因此，最好的方法是我们通过判断跳过执行测试。

```
use Test::More tests => 7;
use Date::ICal;
# Make sure epoch time is being handled sanely.
my $t1 = Date::ICal->new( epoch => 0 );
is( $t1->epoch, 0,                  "Epoch time of 0" );
SKIP: {
        skip('epoch to ICal not working on MacOS', 6)
            if $^O eq 'MacOS';
        is( $t1->ical, '19700101Z', "    epoch to ical" );
        is( $t1->year,    1970,             "    year()"    );
        is( $t1->month, 1,                  "    month()" );
        is( $t1->day,     1,                "    day()"     );
        # like the tests above, but starting with ical instead of epoch
        my $t2 = Date::ICal->new( ical => '19700101Z' );
        is( $t2->ical, '19700101Z', "Start of epoch in ICal notation" );
        is( $t2->epoch, 0,                  "    and back to ICal" );
}
```

这里有一点小技巧。当运行在除了 MacOS 外的其它操作系统时，所有的测试正常运行。但是当在 MacOS 运行时，skip()导致 SKIP 块的整个内容体跳出。不会执行 SKIP 块，但是会打印一些特殊的输出，告诉 Test::Harness 已经跳过测试。

```
1..7
```

ok 1 - Epoch time of 0

ok 2 # skip epoch to ICal not working on MacOS

ok 3 # skip epoch to ICal not working on MacOS

ok 4 # skip epoch to ICal not working on MacOS

ok 5 # skip epoch to ICal not working on MacOS

ok 6 # skip epoch to ICal not working on MacOS

ok 7 # skip epoch to ICal not working on MacOS

这意味着在 MacOS 上你的测试不会失败。你必须小心这些跳过的测试。它不是用来跳过真正的错误的。

如下这个测试将全部跳过。

```
SKIP: {
        skip("I don't wanna die!");
        die, die, die, die, die;
}
```

Todo 测试

Date::ICal 手册中介绍了这个功能：

ical

$ical_string = $ical->ical;

Retrieves, or sets, the date on the object, using any valid ICal date/time string.

但是没有在 Date::ICal 测试包中看到任何使用 ical()方法的测试。因此我要写一个。

```
use Test::More tests => 1;
my $ical = Date::ICal->new;
$ical->ical('20201231Z');
is( $ical->ical, '20201231Z',     'Setting via ical()' );
```

运行它，我得到了：

```
1..1

not ok 1 - Setting via ical()

#       Failed test (- at line 6)

#                 got: '20010814T233649Z'

#       expected: '20201231Z'

# Looks like you failed 1 tests of 1.
```

看起来它还没有实现。如果我们没有时间去改正这个错误，我们将要把它包含在一个 TODO 块中，说明这个测试会失败。

```
use Test::More tests => 1;
    TODO: {
        local $TODO = 'ical($ical) not yet implemented';
        my $ical = Date::ICal->new;
        $ical->ical('20201231Z');
        is( $ical->ical, '20201231Z',     'Setting via ical()' );
}
```

现在我运行时，输出结果就会不同：

```
1..1

not ok 1 - Setting via ical() # TODO ical($ical) not yet implemented

#                 got: '20010822T201551Z'

#       expected: '20201231Z'
```

Test::More 不说``Looks like you failed 1 tests of 1"。语句'# TODO'告诉 Test::Harness ``this is supposed to fail" 而且它把一个失败当成一个成功的测试。因此即使在底层代码修改之前，你都能写测试了。

如果一个 TODO 测试通过了，Test::Harness 会报告它``UNEXPECTEDLY SUCCEEDED"。此时，你只要简单的把 TODO 代码块和 local $TODO 删除，把它变成一个真正的测试即可。

6.1.2 Test::Unit

它是一个高度面向对象的一个测试框架。它的使用接口与 java 中的 Junit 很相似。为了与实际的脚本编程相适应，Test::Unit 并不提供很多单元测试的面向对象的方法——如果你想要这种方式，请看 Test::Unit::TestCase (also included in this install)。

一个关于面向对象方法的简单的介绍能够在 Test::Unit::TestCase(test 基类)的文档中找到。Test::Unit 自我测试包(包含在 Test::Unit::tests::AllTests 中)是一个这个方法的好的例子。

这个测试框架也有一个基于 GUI 的接口 。"TkTestRunner.pl"脚本显示了如何调用它。

这个测试框架也描述了从 Test::Harness 风格的测试到单元测试框架的适配器，反之也有。参见 Test::Unit::HarnessUnit 和 Test::Unit::UnitHarness。这个方法的一个例子是单元测试框架的自我测试，你可以通过'make test' 命令开始 (参见 t/all_tests.t)。

6.2 异常处理

这里主要讨论在 Perl 中异常处理的有关细节，和如何使用 Error.pm 实现它。我们将会讨论使用异常处理优于传统的错误处理机制之处。使用 eval {}进行错误处理，和 eval {}的问题以及 Fatal.pm 中可得到的功能。但是，主要焦点将集中在使用 Error.pm 处理异常。

6.2.1 定义

什么是异常?

一个异常可以定义成一个发生在程序执行期间的事件,这个事件使它背离正常的执行路径。很多类型的错误能导致异常。严重的错误如虚拟内存溢出，简单的错误如试图读取一个空的堆栈或打开一个无效的文件。

一个异常通常伴随三个重要的消息：

- 异常的类型——由异常对象的类决定。
- 异常发生的地点——堆栈跟踪。
- 上下文信息——错误消息和其他的状态信息。

一个异常处理器是一段代码，用于优雅的处理异常。

通过使用异常管理错误，应用程序可以获得很多传统错误处理机制得不到的好处。

6.2.2 使用面向对象异常处理的好处

面向对象的异常处理允许你分离错误处理代码和普通的代码。作为一个结果，代码变得更简单，更可读而且往往更有效。代码更有效是因为通常的执行路径不必检查错误引起了CPU CPU 循环的减少。

面向对象的异常处理另外一个重要的优点是延调用栈传播错误沿调用栈传播错误的能力。 这些都是自动发生的而不需要程序员显式的检查返回值和返回它们给调用者。而且，层层传递返回值的方法是容易导致错误的，因为伴随着代码的跳跃倾向于失去重要的信息。

大多数时间，错误发生的地点往往不是处理错误的最好地点。因此，错误需要返回给调用者。但是当错误达到能够处理的地方时，很多错误上下文丢失了。这是传统的错误处理机制的一个常见问题 (例如，检查返回值并传递给调用者)。异常处理能弥补这个缺点，它能允许错误发生点的上下文相关信息而且传递它到能有效处理的点。

例如，如果你需要一个函数 processFile()，你的应用程序调用的第四个方法。而且 func1() 是仅有的对发生在 processFile()中的错误感兴趣感兴趣的方法。使用一个传统的错误处理机制，你可能像下面这样做来传递错误代码直到 func1()。

```
sub func3
{
    my $retval = processFile($FILE);
    if (!$retval) { return $retval; }
    else {
       ....
    }
}

sub func2
{
    my $retval = func3();
    if (!$retval) { return $retval; }
    else {
       ....
    }
}

sub func1
{
    my $retval = func2();
    if (!$retval) { return $retval; }
    else {
       ....
    }
```

```
}

sub processFile
{
    my $file = shift;
    if (!open(FH, $file)) { return -1; }
    else {
       # Process the file
       return 1;
    }
}
```

使用 OO 异常处理，你要做的所有的事情是把对 func2()的调用包裹在 try 代码块中，并用一个合适的异常处理器处理(catch 代码块).从该代码块扔出的异常。使用异常处理的等价的代码如下：

```
sub func1
{
    try {
        func2();
    }
    catch IOException with {
        # Exception handling code here
    };
}

sub func2 { func3(); ... }
sub func3 { processFile($FILE); ... }

sub processFile
{
     my $file = shift;
     if (!open(FH, $file)) {
        throw IOException("Error opening file <$file> - $!");
     }
     else {
        # Process the file
        return 1;
     }
}
```

因为 func1() 是处理 catch 代码块的唯一的函数，在函数 processFile()中扔出的异常被一路传给 func1()，在那里它会被 catch 代码块合适的处理掉。这两个错误处理技术比较起来，面向对象的异常处理的优点是显而易见的。

最后，异常处理能用来聚集相关的错误。通过这样做，你能够使用一个异常处理器处理

相关的异常。能使用异常类的继承层次关系给异常分组。然而，一个异常处理器能够通过它的参数捕获该类的异常的某些，或捕获它的子类的任何异常。

6.2.3 在 Perl 中实现

我们首先复习一下 Perl 的内在异常处理机制。

Perl 有一个内建的异常处理机制：eval {}模块。它通过把需要执行的代码包裹在 eval 块中，然后检查 $@ 变量，看是否发生异常。典型的语法如下：

```
eval {

    ...

};

if ($@) {

    errorHandler($@);

}
```

在 eval 块中，如果一个语法错误或者运行时错误，或者执行了一个 die 语句，eval 就返回一个 undefined 的值，而$@设置成错误信息。如果没有错误，$@就是一个空字符串。

什么错了？因为错误信息存储在$@ ，是一个简单的标量，检查已经发生错误的类型容易出错。而且$@没有告诉我们异常发生在哪儿。为了解决这些问题，Perl 5.005 引入了异常对象。

从 Perl 5.005 以后，你可以这样做：

```
eval {
    open(FILE, $file) ||
    die MyFileException->new("Unable to open file - $file");
};

if ($@) {
  # 现在$@ 包含一个异常对象，是 MyFileException 的实例。
  print $@->getErrorMessage();
# 这里 getErrorMessage()是类 MyFileException 中的一个方法
}
```

异常类(MyFileException)能够构建你想要的任何功能。例如：如果你想得到调用上下文，通过在异常类的构造器中(典型的如 MyFileException::new())使用 caller()。

测试异常类型也变成可能：

```
eval {
    ....
};

if ($@) {
if ($@->isa('MyFileException')) {
  # 声明异常处理器
      ....
    }
else {
  # 通用的异常处理器
      ....
    }
}
```

如果异常对象实现字符串化，通过重载字符串操作，然后当$@用在字符串上下文中时，对象的字符串化的版本就可得到。通过合适的构建重载方法，在字符串上下文中的$@的值就可以按要求裁减。

```
package MyException;

  use overload ('""' => 'stringify');
  ...
  ...
  sub stringify
  {
    my ($self) = @_;
    my $class = ref($self) || $self;

    return "$class Exception: " . $self->errMsg() . " at " .
                                          $self->lineNo() . " in " .
                                          $self->file();
    # 假设 errMsg()，lineNo() 和 file() 是异常类中的方法，
    # 分别存储返回错误消息，行号和源文件。
}
```

当重载字符串操作""，重载方法(在我们这里是 stringify())期望返回一个字符串表示对象的字符串形式。 stringify()方法能够返回关于异常对象的各种上下文/状态信息。

6.2.4 eval 的问题

如下是使用 eval {}建造的一些问题：

- 依赖于上下文，类似的返回结果可能意味着不同的事情。
- eval 代码块能用作构建动态代码块和异常处理 。
- 对于清理处理，即 finally 代码块，没有内在的规定。
- 如果需要跟踪堆栈，需要通过手写的代码维护。
- 不美观(尽管这是非常主观的看法)。

6.2.5 使用 Error.pm

Error.pm module 实现了面向对象的异常处理机制。它模拟其它的面向对象语言(如 Java 和 C++)中的 try/catch/throw 语法。它也避免了使用 eval 固有的所有问题。因为它是一个纯 perl 模块，它能够运行在 Perl 运行的几乎所有模块。

这个模块提供两个接口：

- 用于异常处理的过程化的接口；
- 其它异常类的基类。

该模块输出各种函数执行异常处理。如果在 use 语句上使用 :try 标记就会输出这些函数。

如下是一个典型的调用：

```
use Error qw(:try);

try {
    some code;
    code that might thrown an exception;
    more code;
    return;
}
catch Error with {
    my $ex = shift;     # Get hold of the exception object
    handle the exception;
}
finally {
    cleanup code;
}; # <-- 记住这个分号，不要忘记在大括号后包括尾部的分号(;)。原因是：所有这些函数接受一个代码引用作为它们的第一个参数 。例如，在 try 代码块，紧接着 try 的代码作为一个代码引用(匿名函数)传递给函数 try()。
```

try 代码块

一个异常处理器通过把可能抛出异常的语句用一个 try 代码块包起来。如果一个异常发生在 try 代码块中，通过合适的与这个 try 代码块关联的异常处理器(catch 代码块)处理它。如果没有异常抛出，try 会返回代码块的结果。

语法是：try BLOCK EXCEPTION_HANDLERS

一个 try 代码块应该有至少一个（或多个）catch 代码块或一个 finally 代码块。

Catch Block

你用一个 try 代码块关联异常处理器：提供一个或多个 catch 代码块直接在 try 代码块后面。例子如下：

```
try {
    ....
}
catch IOException with {
    ....
}
catch MathException with {
    ....
};
```

语法是：

catch CLASS with BLOCK

它允许满足条件$ex->isa(CLASS)所有的错误可以通过执行 BLOCK 处理。

BLOCK 接受两个参数。第一个是被抛出的异常，第二个是一个标量引用。如果这个标量引用被设置成从 catch 代码块返回，则 try 代码块继续，就好像没有发生异常一样。

如果被第二个参数引用的这个标量没有设置，而没有异常被抛出 (在 catch 代码块中)，然后当前的 try 代码块将返回，伴随 catch 代码块的结果。

为了传播一个异常，catch 代码块可以选择通过调用$ex->throw()再次抛出异常。

异常处理器的顺序是重要的。如果你有在不同的继承层次上的处理器就更重要了。用来处理离根越远的错误的异常处理器应该放在 catch 代码块列表的前面。

一个处理指定类型对象的异常处理器可能被超类的异常处理器预清空。当超类的异常处理器出现在前面时，就会发生这种情况。

例如：

```
try {
    my $result = $self->divide($value, 0);
    # divide() throws DivideByZeroException
    return $result;
}
catch MathException with {
    my $ex = shift;
    print "Error: Caught MathException occurred\n";
    return;
}
catch DivideByZeroException with {
    my $ex = shift;
    print "Error: Caught DivideByZeroException\n";
    return 0;
};
```

假设如下的层次结构：

MathException 是 Error 的子类

[@MathException::ISA = qw(Error)]

DivideByZeroException 是 MathException 的子类

[@DivideByZeroException::ISA = qw(MathException)]

在上面的代码列表中，DivideByZeroException 被第一个 catch 代码块捕获了，而不是第二个。这是因为 DivideByZeroException 是 MathException 的子类。换句话说，$ex->isa('MathException')返回 true。因此，第一个 catch 代码块处理了这个异常。反转 catch 代码块的顺序可以保证异常被正确的异常处理器捕获。

Finally 代码块

设置异常处理器的最后一步是在控制传递到程序的其他部分以前为清理提供一个机制。这可以通过把清理逻辑包含在一个 finally 代码块中实现。在 finally 块中的代码的执行与 try 代码块中的执行无关。finally 代码块的典型使用例如关闭文件或通常的释放系统资源等。

如果没有异常被抛出，catch 块中的代码就不会执行。但是在 finally 块中的代码总是会执行。

如果抛出一个异常，就会执行合适的 catch 块中的代码。一旦代码执行完成，就执行 finally 块。

```
try {
    my $file = join('.', '/tmp/tempfile', $$);

    my $fh = new FileHandle($file, 'w');
    throw IOException("Unable to open file - $!") if (!$fh);

    # 可能抛出异常的代码
    return;
}
catch Error with {
    my $ex = shift;
    #异常处理代码
}
finally {
    close($fh) if ($fh);      # 关闭临时文件
    unlink($file);            # 删除临时文件
};
```

在上面的代码中，在 try 代码块中创建一个临时文件而且这个块中的一些代码有可能抛出一个异常。与 try 块是否执行成功无关，临时文件必须关闭并且从文件系统中删除。这通过在 finally 块中关闭和删除文件实现。

注意，每个 try 块只允许一个 finally 块。

Throw 语句

throw() 创建一个新的"Error"对象并抛出一个异常。这个异常可以通过包裹在一个 try 代码块中捕获。否则程序将会退出。

也可以在一个存在的异常对象上调用 throw()，再次抛出它。下面列出的代码解释了如何再次抛出一个异常：

```
try {
    $self->openFile();
    $self->processFile();
    $self->closeFile();
}
catch IOException with {
    my $ex = shift;
    if (!$self->raiseException()) {
       warn("IOException occurred - " . $ex->getMessage());
       return;
    }
    else {
```

```
        $ex->throw(); # 再次抛出异常
    }
};
```

构建你自己的异常类

设置$Error::Debug 包变量的值成 true 将打开捕获堆栈追踪，以后可以使用 stacktrace() 方法查询该值。stacktrace 将按顺序显示所有的达到异常的方法列表。

下面的代码片断创建异常类 MathException，DivideByZero 和 OverFlowException。而后两者是 MathException 的子类，而 MathException 它自己又是从 Error.pm 继承的。

```
package MathException;

use base qw(Error);
use overload ('""' => 'stringify');

sub new
{
    my $self = shift;
    my $text = "" . shift;
    my @args = ();

    local $Error::Depth = $Error::Depth + 1;
    local $Error::Debug = 1;   # Enables storing of stacktrace

    $self->SUPER::new(-text => $text, @args);
}
1;

package DivideByZeroException;
use base qw(MathException);
1;

package OverFlowException;
use base qw(MathException);
1;
```

更多...

Fatal.pm

如果你有一些函数在错误时返回 false 而在成功时返回 true ，那么你就可以使用 Fatal.pm 把它们转换成失败时扔出异常的函数。对用户定义的函数或者内建的函数(有一些

特例)都可以这样处理。

```
use Fatal qw(open close);

eval { open(FH, "invalidfile") };
if ($@) {
    warn("Error opening file: $@\n");
}

....
....

eval { close(FH); };
warn($@) if ($@);
```

缺省情况下，Fatal.pm 捕捉每一个致命函数的使用，即：

```
use Fatal qw(chdir);
if (chdir("/tmp/tmp/")) {
    ....
}
else {
    # 执行流程永远不会到达这里。
}
```

如果你使用 Perl 5.6 或以上版本，你可以通过在 import 列表中增加:void 包围它。在 import 列表中的所有函数都会导致一个错误，仅当在 void 上下文调用它们时，即当他们的返回值被忽略时。

通过改变 use 语句成下面这样，我们可以肯定当 chdir() 失败时在 else 中的代码能够执行。

```
use Fatal qw(:void chdir);
```

如下代码演示了联合使用 Fatal.pm 和 Error.pm：

```
use Error qw(:try);
use Fatal qw(:void open);

try {
    open(FH, $file);
    ....
    openDBConnection($dsn);
    return;
}
catch DBConnectionException with {
    my $ex = shift;
    # 数据库连接失败
```

```
}
catch Error with {
    my $ex = shift;
    #如果 open()失败，我们会到这里
};
```

因为在 Perl 6 中的异常处理语法将要设计成和 Error.pm 很接近，对于开发者来说，在代码中使用面向对象风格的异常处理，然后在 Perl 6 推出的时候把它移植到 Perl 6 中的异常处理语法可能是有意义的。

6.2.6 结论

如下是选择异常-处理机制而不是传统的错误处理机制的一些关键原因：

- 错误处理代码能够与普通的代码分开；
- 更简单，可读性更好而且代码更有效；
- 能够沿调用栈传播错误信息；
- 能够为异常处理器保留上下文信息；
- 把错误类型按逻辑分组。

那么，停止返回错误代码，开始抛出异常吧。

第7章 Perl 扩展

7.1 制作可执行文件

7.1.1 使用 perlcc 制作 exe

首先修改批处理文件 perlcc.bat 中的 perl57.lib ,改为 perl58.lib。该文件一般在 C:\Perl\bin 目录下。

以 sample.pl 为例，执行如下命令：

```
>perlcc -o sample.exe sample.pl
```

将生成 sample.exe 可执行文件。注意：发布 sample.exe 时需要附带动态连接库 perl58.dll。该文件一般在 c:\perl\lib\core\目录下。

大家知道 Perl 很多的模块是本身 Perl 的语言和内部函数编写的。但是有一部分包括 IO::Socket DBD DBI 等这些常用的模块，由于 Perl 本身内置函数限制，采用了 PerlXS 接口通过 C 程序达到目的的。这些是通过第三方程序达到目的模块是无法通过 perlcc 成功编译的，但是使用 Perl Bytecode 能解决这一问题。

关于 Perl 编译器可以参考 CPAN 的 Authors/Malcolm_beattie。

7.2 从 c 调用 perl

7.2.1 准备工作

找到 perl 库所在路径：

```
>perl -MConfig -e "print $Config{archlib}"
```

```
C:\Perl\lib
```

perl library 和头文件 EXTERN.h 、perl.h 在 C:\Perl\lib\CORE 路径。

找到合适的 c 编译器。

```
>perl -MConfig -e "print $Config{cc}"
```

cl

找到应该包含的库

>perl -MConfig -e "print $Config{libs}"

oldnames.lib kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib netapi32.lib uuid.lib wsock32.lib mpr.lib winmm.lib version.lib odbc32.lib odbccp32.lib msvcrt.lib

7.2.2 添加 Perl 解释器

编译

```
#include <EXTERN.h>                    /* from the Perl distribution     */
#include <perl.h>                      /* from the Perl distribution     */
static PerlInterpreter *my_perl;   /***     The Perl interpreter     ***/
int main(int argc, char **argv, char **env)
{
        my_perl = perl_alloc();
        perl_construct(my_perl);
        PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
        perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
        perl_run(my_perl);
        perl_destruct(my_perl);
        perl_free(my_perl);
}
```

7.3 使用 Perlscript

什么是 PerlScript 呢？PerlScript 是一个 ActiveX 脚本引擎，使你可以在任何 ActiveX scripting host 上运行 Perl 程序。在安装 Active Perl 时，PerlScript 是作为可选组件安装的。

你可以在 WSH 中使用它。下面是一个 Hello World 的例子：

```
$WScript->Echo('Hello World !');
```

更常见的是，你可以在 ASP 中使用它。这样做最大的好处是，在 ASP 环境之内仍然能够使用到 ActivePerl 和 PerlScript 的长处：模式匹配和正则表达式，Perl 内部函数，并且能够在您的代码中使用自由和便宜的巨大的 Perl 模块库。

你可以把如下代码包含到你要使用 PerlScript 的 ASP 页面中去：

<% @LANGUAGE="PerlScript" %>

例如要输出"Hello, world!"，你可以这样做：

```
<% @LANGUAGE="PerlScript" %>
<%
  $Response->Write("Hello, world!");
%>
```

7.3.1 从 PerlScript 访问 ASP 内在对象

为了帮助开发者，PerlScript 提供只在 ASP 环境内能使用的几个对象。它们是 Application，Session，Response，Request，和 Server 对象。

当 ASP 应用首先开始时就创建了 Application 对象, 并且持续应用的终身。对象有两个 COM 集合, 都可以从脚本中取得：StaticObjects 集合和 Contents 集合。在 global.asa 应用程序文件中使用<OBJECT> 标记设置 StaticObjects 的值。内容集合包含在运行时间设置和检索的值。 一个 COM 集合是一个作为一个整体看待的相似对象的小组。它们有相关的属性以便让开发人员直接地访问各自的元素，或通过整个集合遍历。

除了这两个集合之外, Application 对象还有两个方法, Lock 和 UnLock, 用于保护对象同时被多于一个应用程序用户改变它的值。

我们通过例子看一下使用 Lock 和 UnLock 的方法。在一个 ASP 页面中设置集合中的值， 然后从另一页检索相同的值。

首先，下文中包含一个页面设置一个新值给 Application 内容，通过首先锁住对象，设置值，然后解锁对象。

```
<%@ LANGUAGE = PerlScript %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HEAD>
<TITLE> Application 对象</TITLE>
</HEAD>

<BODY BGCOLOR=#FFFFFF>
<%
# 访问 ASP Application 对象
#取得和设置值

$Application->Lock;
$Application->Contents->SetProperty('Item', 'test', 0);
$Application->UnLock;
%>
```

```
</BODY>
</HTML>
```

注意，你不必创建 Application 对象—它是系统内部创建的(所有的 ASP 对象也都是如此)。

如果你已经用过 VBScript，你或许已经注意到你必须使用一种与 PerlScript 不同的方法设置集合中的值。 VBScript 允许使用下面这样快速的方法直接设置值：

```
Application.Contents("test") + val
```

但是 PerlScript 不支持这种方式。你只能使用 SetProperty 方法设置 Content 项：

```
$Application->Contents->SetProperty('Item', 'test', $val);
```

而且，你只能使用 SetProperty 设置 ASP 的内建对象属性。或者也可以用 Perl 的 hash 解引用设置(取得)值，如下所示：

```
my $lcid = $Session->{codepage};

<%@ LANGUAGE = PerlScript %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HEAD>
<TITLE> Application 对象 </TITLE>
</HEAD>
<BODY BGCOLOR=#FFFFFF>
<%
# 取得 ASP Application 对象
# 取得和设置值

# 测试值
my $val = $Application->Contents('test');
$Response->Write("Before value: $val<p>");

# 增加值
$val++;

# 重置值
$Application->Lock;
$Application->Contents->SetProperty('Item','test',$val);
$Application->UnLock;

# 再次测试值
my $val = $Application->Contents('test');
$Response->Write("After value: $val");
```

```
%>
</BODY>
</HTML>
```

包含另一 ASP 页面访问那些变量集，打印出值，增加值，并且将它设置回 Application 对象。然后访问这个值，再一次将它打印出来。 从许多浏览器访问这个页面，并且从许多分离的客户 session，它们增加的是相同的 Application 项目，因为所有的应用物体的进入的相同应用项目。

除了上面介绍的 Application 对象，还存在 Session 对象，Session 对象持续存在于用户会话的整个过程。

当某一用户第一次访问 ASP 程序的时候创建该 Session 对象，并一直持续到会话结束或者超时，或者用户以某种方式断开于程序的连接。Session 也具有 StaticObjects 和 Contents 集合，但是同 Application 对象不同的是，你设置 Contents 的值时不需要 lock 或者 unlock Session 对象。但设置 Contents 中的值或从中得到值的时候，需采用同 Application 相同的方式，示意代码如下：

```
$Session->Contents->SetProperty('Item', 'test', 0);

my $val = $Session->Contents('test');
```

7.3.2 其它的选择

Session 还有其它属性与方法，包括 Timeout 属性（用来设定计算超时的值），Abandon 方法，用来放弃当前的会话。每一个会话都赋予一个唯一的 SessionID 标识并且这个值可以在脚本中访问。如果想识别一个特别的用户，访问 SessionID 时应谨慎，这个值仅对一个特定的会话是唯一并且有意义的。

在支持国际化方面，有一些 Session 属性可以控制 web 页面、CodePage 中使用的字符集，并指定区域标识（LCID）。LCID 是系统定义的区域的通用简称，控制比如货币符号等（比如，美元$或者英镑£）。

```
<%@ LANGUAGE = PerlScript %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HEAD>
<TITLE> Session object </TITLE>
</HEAD>
<BODY BGCOLOR=#FFFFFF>
<%
# 将超时时间设置为 60 分钟
$Session->{Timeout} = 60;
# 打印代码页
my $cdpg = $Session->{codepage};
```

```
$Response->Write("codepage is: $cdpg<p>");
#打印 LCID
my $lcid = $Session->{LCID};
$Response->Write("LCID is: $lcid");
%>
</BODY>
</HTML>
```

上面这段代码显示了设置 Session 对象的 Timeout 属性，读取并打印 CodePage 和 LCID 的值。在我自己的环境返回 ASP 页面并且在这个环境测试，CodePage 打印出的值是 1252－美国英语以及很多欧洲语言映射到这种字符集。LCID 打印出的值是 2048，这是美国的区域标识。

Application 和 Session 示例用到了第三个 ASP 内建的对象，Response 对象。该对象负责所以从服务器端到客户端的通讯。包括在客户端设置 Web cookies（用 Cookies collection），以及用 Buffer 属性来控制是否在传到客户端之前作缓冲，或者生成就马上发送给客户端。

```
$Response->{Buffer} = 1;
```

可以使用 Buffer 属性同 End、Flush 和 Clear 方法一起来控制返回给客户端的内容。将 Buffer 设置为 true (Perl 中的值是 1)，直到整个 ASP 页面完成才将页面内容返回给客户端。如果页面上发生错误，调用 Clear 来清除缓冲中的内容（一直到调用点处）。调用 End 方法在调用处结束脚本的处理并返回缓冲区的内容。调用 Flush 方法输出缓冲区的内容，并结束缓冲。

```
<%@ LANGUAGE = PerlScript %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HEAD>
<TITLE> Request </TITLE>
</HEAD>
<BODY BGCOLOR=#FFFFFF>
<%
my $firstname = $Request->Form('firstname')->item;
my $lastname = $Request->Form('lastname')->item;
my $email = $Request->Form('email')->item;
$Response->Write("Hello, $firstname $lastname<p>");
$Response->Write("Email yourself by clicking <a href='mailto:$email'>here</a>");
%>
</BODY>
</HTML>
```

上述代码演示 ASP 页面处理 POST 过来的 HTML 表单。

示例中 ASP 页面以表格：lastname、firstname 和 email 的 3 个文本字段形式给出。然后用这生成问候页面，包括为 email 地址添加超连接。如果表单已经调用了 GET 方法（表单中字段名、字段值对连接到处理页面的 URL），就可以利用 QueryString 来访问页面内容，采用下述代码：

my $firstname = $Request->QueryString('firstname')->item;

除 Form 集合和 QueryString 集合，Request 对象还具有 SeverVariables 集合，它记录服务器和客户端的环境配置信息。ServerVariables 集合实现的功能和 CGI 中访问%ENV 类似。

通过指定确切的变量名，可以访问 SeverVariables 集合的每一个元素，代码如下：

my $val = $Request->ServerVariables('PATH_TRANSLATED')->item;

或者你可以在整个 collection 中迭代。要实现迭代，可以利用 Win32:OLE:Enum Perl 模块来帮助实现。Enum 类是随 ActivePerl 装入的模块之一，提供一个枚举专门来对像 ServerVariables 这样的 COM 集合作迭代查询。

可以利用 Perl 的 foreach 语句来迭代查询每一个 ServerVariables 元素，打印出元素的名称和关联的值。

如果一个 HTML 表单包含一个文件输入元素－－用来随表单上传一个文件，可以采用 Request 对象的 BinaryRead 方法来处理表单的内容。TotalBytes 属性提供关于总共上传的字节量信息。

7.4 其它语言中使用 Perl

ActivePerl 使用 PerlSE.dll 注册的 PerlScript 实现了一个 COM 组件。它本来是用在 WSH 中的。可以修改 Perl 文件后缀为 pls 使用它，因为 WSH 依靠它来分辨该文件为 PerlScript。执行方法如下：

>cscript test.pls

或

> wscript test.pls

我们可以利用这个 PerlScript 的 COM 组件来把 Perl 集成到其它语言中。例如在 Power Builder 中的使用样例：

```
OleObject wsh
Integer   li_rc, i, j , k

wsh = CREATE OleObject
li_rc = wsh.ConnectToNewObject( "MSScriptControl.ScriptControl" )
wsh.language = "perlscript"

i = 1
j = 2
```

```
k = integer(wsh.Eval( string(i) + " ^ " + string(j)))

MessageBox( "Result" , string(i) + " xor " + string(2) + " = " + string(k))
```

7.5 Perl 中使用 c

7.5.1 Inline

Inline 会把你不需要知道的所有的乱七八糟的实现细节都屏蔽起来，让调用它的代码保持到最少。它分析你的代码，如果需要的话，就编译它，把整个事情运作起来。借助它你能够用其它语言写函数，过程，类和方法，并且调用它们，就好像它们都是用 Perl 写的一样。

下面介绍一个 C 语言的简单例子。让我们从 Hello, world 开始欣赏 Inline 的表演。

```
use Inline C => <<'END_C';
void greet() {
    printf("Hello, world\n");
}
END_C
greet;
```

从命令行执行此脚本，它会打印出：

Hello, world

在 windows 下，可以调用 windows API：

```
use Inline C => <<'END_C';
void greet() {
    MessageBox(    NULL,    "Hello, world",    "windows",    MB_OK    );
}
END_C
greet;
```

在这个例子中，Inline.pm 被编程语言的名字”C"激活。一个字符串包含一段语言的源代码。 C 代码定义了一个函数叫做 greet() ，它会绑定到 Perl 例程&main::greet。因此，当我们调用 greet() 例程，程序会把信息输出到显示器。

你可能想知道为什么没有像#include <stdio.h> 这样的语句呢? 因为 Inline::C 自动添加了如下行到你的代码顶端：

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "INLINE.h"
```

这些头文件包括所有的标准系统头文件，所以你只需要在包括非标准的库时才使用#include。这是 Inline 的哲学：让简单的事情保持简单。

下一个问题是，如何在 Perl 与 C 之间传递数据？ 在这个例子中我们传递一个字符串给 C 函数，让它回传一个 新的 Perl 标量。

```
use Inline C;
print JAxH('Perl');

__END__
__C__
SV* JAxH(char* x) {
    return newSVpvf("Just Another %s Hacker\n", x);
}
```

这个程序的执行结果是，打印出：

```
Just Another Perl Hacker
```

你可能已经注意到了，这个例子编码不同于上一个。use Inline 语句声明使用的语言，而不是源代码。这时 Inline 把记号'__C__'看成结束。

是用缺省的 Perl 类型约定，Inline 能够很容易的在基本的 Perl 数据类型和 C 之间转换。

这个例子使用了 Inlining 的高级概念。它返回的值是 SV* 类型的(标量值)。 标量值是最常见的 Perl 内部类型。Perl 函数 newSVpfv() 使用了熟悉的 sprintf()语法用来从字符串创建一个新的标量值。

不仅仅包括 C and C++，Inline 支持好几种编程语言，当前 Inline 支持 C, C++, Python, 和 CPR。

Inline C 可以在所有常用的 Unix 和 Microsoft Windows 下运行。Inline 已经成功用于 Linux, Solaris, AIX, HPUX,和 BSD。

在 Windows 上使用 Inline 通常有两种方法：第一种是使用 ActiveState's ActivePerl for MSWin32。为了使用 Inline 环境，还需要 MS Visual C++ 6.0，需要运行 vcvars32.bat，注册环境变量。可以在命令行使用 cl.exe 编译和 nmake make 工具。另一种方案是使用 Cygwin 工具。它是 Windows 上的 Unix 移植层。它包含所有的大多数 Unix 工具，例如 bash, less, make, gcc，当然还有 perl。

Inline 语法

Inline 和你用过的大多数 Perl 模块不同，它既不输入任何函数到你的名字空间，也没有任何面向对象的方法。 它唯一一个接口是通过'use Inline ...' 命令。 Inline 用法如下：

use Inline C => source-code,

config_option => value,

config_option => value;

这里 C 是一个编程语言。source-code 是一个字符串或关键字'DATA'。接下来是一些可选的配置选项以 'keyword => value' 的形式出现。 如果使用'DATA'选项，没有配置参数，可以直接写：

use Inline C;

Inline 最通常的用法是使用它调用外部函数库，你可以为每个 Perl 中需要使用的函数提供一个包裹函数。包裹函数调用真正的函数，它同时负责参数的传入与传出。这里是一个简短的 Windows 例子，显示一个带标题的文本消息框，和一个确定按扭：

```
use Inline C => DATA =>
             LIBS => '-luser32',
             PREFIX => 'my_';
MessageBoxA('Inline Message Box', 'Just Another Perl Hacker');

__END__
__C__
#include <windows.h>
int my_MessageBoxA(char* Caption, char* Text) {
   return MessageBoxA(0, Text, Caption, 0);
}
```

这段程序从 MSWin32 的 user32.dll 调用了一个函数。包裹器决定了从 Perl 传递的参数的类型和顺序。尽管真的 MessageBoxA()函数需要四个参数，我们可以仅暴露两个给 Perl，并且我们能够改变顺序。为了避免在 C 中的名称空间冲突，包裹器必须有一个不同的名字。通过使用 PREFIX 选项 (和 XS PREFIX 选项相同)我们能够绑定它到在 Perl 中的原来的名字。

它接受所有的类型

Inline 的旧版本仅支持 5 种数据类型。它们是： int, long, double, char* 和 SV*。这些是你所需要的全部类型。所有的基本 Perl 标量类型都通过这些表示。 像引用等可以通过使用通用的 SV* (scalar value) 类型处理，然后你自己在 C 函数中做这些映射代码。

在 Perl 的 SV*类型和 C 类型间的转换过程叫做类型映射。在 XS 中，通常使用类型映射文件做这个。一个缺省的类型映射文件存在于每一个 Perl 安装，是一个叫做 C:\Perl\lib\ExtUtils\typemap 或类似名称的文件。 这个文件包括超过 20 种不同的 C 类型的转换代码，包括所有的 Inline 缺省类型。

从版本 0.30 开始，Inline 不再有任何内建的类型。它只从 typemap 文件取得所有的类型。因为它使用 Perl 的缺省 typemap 文件用作它自己的，它实际上自动有更多的类型可供使用。

这个版本提供了许多灵活性。你可以通过使用 TYPEMAPS 配置选项，声明你自己的类型映射文件。这样就不仅允许你用自己的变换代码覆盖缺省的类型变换，同时意味着你可以增加新的类型到 Inline。用这种方法扩展 Inline 语法的主要的优点是已经存在许多用于各种 API 的类型映射。如果你过去使用自己的 XS 代码，你就可以使用你已有的类型映射文件。不需要任何改变。

让我们看一个自己编写类型映射的小例子。由于某种原因，C 类型 float 没有出现在缺省的 Perl 类型映射文件中。可能的原因是 Perl 的浮点数总是存成双精度类型，它的精度比 浮点数更高。下面就来写一个支持浮点数的类型映射文件：

```
float                           T_FLOAT



INPUT

T_FLOAT

        $var = (float)SvNV($arg)

OUTPUT

T_FLOAT

        sv_setnv($arg, (double)$var);
```

先看结构，这个文件提供了两个代码片断。一个转换 SV*到浮点数，另外一个相反。我们写一个脚本测试一下：

```
use Inline C => DATA =>
            TYPEMAPS => './typemap';

print '1.2 + 3.4 = ', fadd(1.2, 3.4), "\n";
__END__
__C__
```

```
    float fadd(float x, float y) {
        return x + y;
}
```

超越 C 的部分

Inline 的主要目标是让在 Perl 中使用其它编程语言变得容易。并不仅限于 C。Inline 初始的实现仅支持 C，对语言的支持直接在 Inline.pm 中。Inline 现在的版本支持多种编译和解释型的语言，而且采用了面向对象的方式实现。每一种面向对象的语言有自己独立的模块，并且都继承于基本的 Inline 模块。

Inline 现在支持的语言：C ，C++和 Python ，分别在 Inline::C，Inline::CPP 和 Inline::Python 模块中实现。

如下是一个使用 Inline Python 的简单程序：

```
use Inline Python;
my $language = shift;
print $language,
        (match($language, 'Perl') ? ' rules' : ' sucks'),
        "!\n";
__END__
__Python__
import sys
import re
def match(str, regex):
    f = re.compile(regex);
    if f.match(str): return 1
    return 0
```

这个程序使用了 Python regex 显示``Perl rules!"。

因为 Python 支持它自己的 Perl 标量，数组和 hash 版本，所以可以从 Python 代码中反过来调用 Perl 代码。如果传递一个 hash 引用给 python，python 会把它转换成字典，反过来也一样。

CPR

如果你想把 Perl 解释器嵌入到 C 程序中去，那么 CPR 则提供了解决问题的另外一种方式。

CPR 的诞生源于这样一个想法：把你的 C 程序传递给 perl 程序，它再传递给 Inline。那样就可以写出这样的程序：

```
#!/usr/bin/cpr
int main(void) {
    printf("Hello, world\n");
}
```

然后就可以从命令行执行它。一个解释性的 C 语言！CPR 就是``C Perl Run"的缩写，对应的 Perl 模块实现叫做 Inline::CPR。

当然，CPR 本身并不是新的语言，但是你可以这样认为。CPR 就像 C 一样，除了你在任何时候都可以直接调用 Perl5 API。CPR 用它自己的 CPR 包裹 API 重新定义了这个 API。

7.5.2 H2xs

h2xs 用来把 C 头文件转换成 Perl 扩展。使用 h2xs 工具也可以作为创建可安装模块的起点。例如，如果你想做一个叫做 Planets 和 Astronomy::Orbits 的模块，你可以输入：

```
>h2xs -XA -n Planets

>h2xs -XA -n Astronomy::Orbits
```

这里 -n 参数指定了模块的名称， -X 表示不需要创建 XS 部件，而-A 指模块不使用 AutoLoader。

这个命令将创建一个叫做. /Planets/ 和 . /Astronomy/Orbits/ 的路径。你会在这个路径下发现所有的部件。模块路径下的文件列表如下：

Makefile.PL	生成 Makefile 文件。
test.pl	测试用。
Changes	程序变更记录。
MANIFEST	文件列表。

其实写模块是件容易的事情。使用 h2xs 程序已经生成了框架模块文件。

在./Astronomy/Orbits/ 路径下的 Orbit.pm 文件内容如下：

```
package Astronomy::Orbits;

use 5.008;
use strict;
use warnings;

require Exporter;
```

```
our @ISA = qw(Exporter);

# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.

# This allows declaration       use Astronomy::Orbits ':all';
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.
our %EXPORT_TAGS = ( 'all' => [ qw(

) ] );

our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

our @EXPORT = qw(

);

our $VERSION = '0.01';


# Preloaded methods go here.

1;
__END__
```

第一行设置包名（缺省的前缀）修饰到文件中的所有的全局标识符 (变量，函数，文件句柄等)。因此变量 @ISA 的全称实际是：@Astronomy::Orbits::ISA。

最后介绍 h2xs 程序接受的参数列表如下：

选项	说明
-A,	不使用 AutoLoader。
-B,	beta 版本。使用 Use beta 版本号 0.00_01 (如果使用了-v 将忽略此选项)。
-C,	忽略改变。忽略创建 Changes 文件，增加 HISTORY 头到 POD 存根。
-F,	cpp 标记。附加的 C 预处理、编译标记。
-M,	函数掩码。不导入指定的 C 函数/宏(缺省是选择全部)。

选项	说明
-O, --overwrite-ok	允许覆盖事先存在的扩展路径。
-P,	忽略 pod。忽略 POD 节存根部分。
-X,	忽略 XS。忽略创建 XS 部件(implies both -c and -f).
-a, --gen-accessors	为结构体和联合体生成 get/set 成员函数(和-x 一起使用)。
-b, --compat-version	声明一个向后兼容的版本。缺省只向后兼容到当前的安装版本。
-c, --omit-constant	忽略 constant() 函数并 function and specialised AUTOLOAD from the XS file.
-d, --debugging	打开调试信息。
-f, --force	强迫创建扩展，即使 C 头文件不存在。
-g, --global	包含安全的存储静态的数据到.x 文件的代码。
-h, -?, --help	显示帮助信息。
-k, --omit-const-func	忽略函数参数中的'const' 属性(和-x 一起使用)。
-m, --gen-tied-var	为了访问声明的变量生成绑定的变量。
-n, --name	声明扩展的名称。
-o, --opaque-re	用于"opaque"类型的正则表达式。
-p, --remove-prefix	声明一个从 Perl 函数名中移走的前缀。
-s, --const-subs	为指定的宏创建例程。
-t, --default-type	自动加载常量的缺省类型(缺省是 IV)。
--use-new-tests	在向下兼容的模块中使用 Test::More。
--use-old-tests	使用模块 Test 而不是 Test::More。
--skip-exporter	不输出符号。
--skip-ppport	不使用可移植层。

选项	说明
--skip-autoloader	不使用模块 C<AutoLoader>。
--skip-strict	不使用 pragma C<strict>
--skip-warnings	不使用 pragma C<warnings>
-v, --version	为扩展指定版本号。
-x, --autogen-xsubs	使用 C::Scan 自动生成 XSUB。

第8章 Unicode 与中文

8.1 字符集

字符集是代表一个可交换的编码，而字符是一个符号的抽象描述。一个字符的代码点（codepoint）是一个与字符位置相关的数字。取得一个字符的代码点的 Perl 函数是 ord。

当处理一个象 ISO8859-1("Latin-1")这样有 256 个字符的字符集时，很容易在计算机中表示它——每个代码点简单的编码成一个字节就行了。当有 65,536 个字符或更多时，必须准确的说明如何把每个字符表示成字节序列。这就是字符的字符编码。

Unicode 通常使用叫做 Unicode Transformation Formats（UTFs）的字符编码。Unicode 是一个 16 位的字符集，使用 65536 个不同的"代码点"。

Unicode 有两个流行的的编码方式：

- UCS-2，每一个字符有两个字节。这样当编码 ASCII 时，大小将是原来的两倍。
- UTF-8，它是 ASCII 兼容的并且每个字符使用 1 到 6 个字节。英文并不使用比以前更多的空间，但是中文却会。

8.2 中文

8.2.1 编码

ASCII 制订的时候，并没有考虑对多语种，特别是对象中国汉字这样的象形文字的支持。为此后来又提出了不少解决方案，其中代码页体系(ISO2022)是现在普遍实行的方案，而 ISO10646/GB13000/Unicode 是今后发展的方向。

中国的汉字编码标准 GB2312 是 7bits 标准，具体说是双 7 位字节标准。 而 ASCII 是单 7 位字节标准,计算机怎么区分呢?一种是在第八位置"1", 提示计算机转入双字节编码,这是最常见的一种实现,也叫 EUC(Extended Unix Code)编码.另一种是用特殊标记提示计算机转入双字节编码,如 HZ 编码就是用开始，用结束的块标识双字节编码区.它们都是 GB2312 的一种实现.对象中国汉字这样的象形文字体系，代码页是根据各个国家，地区或行业标准，按照 EUC 方式编码。代码页向下兼容 ASCII，是一种不等长编码，会带来代码的复杂性，同时还会引起因代码页切换而带来的乱码问题。

Unicode 是一种多字节等长编码。ISO10646/GB13000/Unicode 现已在 UCS2 上实现一致，也就是已实现双字节编码标准。下面所讨论的 ISO10646/GB13000/Unicode，就只是指 UCS2

这种情况。Unicode 对 ASCII 采取前面加"0"字节的策略实现等长兼容。如"A"的 ASCII 码为 0x41, Unicode 码就为 0x00,0x41。

最常用的国家标准(GB)有：

- GB2312 是基本集，也就是目前最常用的标准。GB2312 系列经过两次修正和扩充，已和原始的 GB2312-1980 标准有些不同。

- GBK 是 GB2312 向 GB13000 过渡的一个中间产物。它是 GB2312 的一次大的扩展，编码向下兼容 GB2312 的 EUC 编码，字汇(字符集)和 GB13000 相同，是 GB2312 的 3 倍。所以说 GBK 也包含 BIG5，Shift-JIS，KSC 的字汇。注意只是包含字汇，而编码与原始的标准是不同的。在具体应用中，用 GBK 字体就可以同时显示 GB2312，BIG5，Shift-JIS，KSC 的字符串。但除了 GB2312 字符串，其它都要转换。

- ISO/IEC 10646 等同于 GB13000-1993/JIS0221-1995/KSC5000-1995 这些国家标准。 制订的目标是包容各语种的文字，其中以汉字最多(Unicode2.0 有 20902 个汉字)。

GB2312 的 EUC 编码范围是第一字节 0xA1~0xFE(实际只用到 0xF7)，第二字节 0xA1~0xFE。GBK 对此进行扩展。第一字节为 0x81~0xFE，第二字节分两部分，一是 0x40~0x7E，二是 0x80~0xFE。其中和 GB2312 相 同的区域，字完全相同。扩展部分大概是按部件(部首)和笔顺(笔画) 从 GB13000 中取出再排列入 GBK 中。因此 GBK 并不是 GB13000，虽然两者字汇相同，但编码体系不同。一个是 ISO2022 系列不等长编码，一个是等长编码，并且编码区域也不同。注意到 GBK 实际上不是国家标准。

8.3 XML 与中文

CPAN 上的 XML::Parser 模块并不包含 GB2312 encoding 支持。要添加支持，可按如下步骤做：

1. 从 ftp.unicode.org 下载 GB2312.TXT。

2. 下载 XML::Encoding 1.01 取得：make_encmap 和 compile_encoding。

3. 运行 make_encmap 如下：

```
make_encmap GB2312 GB2312.TXT > GB2312.encmap
```

4. 添加 expat='yes' 到 GB2312.encmap 第一行（实际上它是一个 xml 文件）。

5. 运行 compile_encoding:：

```
compile_encoding -o GB2312.enc GB2312.encmap
```

6. 把 GB2312.enc 拷贝到：

C:\Perl\site\lib\XML\Parser\Encodings

7．把支持 GB2312 的 libexpat.dll 拷贝到 perl 可以找到的路径。

8.3.1 Expat

Expat 是一个广泛使用的 xml 解析器，从手持设备到大型的 unix 机器都有使用。它是一个用 C 编写的函数库， Perl 的 XML::Parser 就是调用这个解析器分析 XML 文档的。

Expat 的一个缺点是，虽然它可以接受各种形式的字符编码，但是它返回的字符串始终是 UTF-8 或 UTF-16 (依赖于 Expat 的编译方式)形式的编码。 必须由应用程序把字符串转换成其它的编码。

即使如此，它解析中文编码仍然有问题。因为 Expat 不支持中文编码（GB2312）,而它又是 Perl 中 XML 的核心模块。有必要对它的源码进行仔细分析，给出解决的补丁。

按照 Parser/Encoding/README 的方法,使用 make_enmap 的时候需要改下载来的 GB2312.TXT。因为在 GB2312.txt 中,中文编码 0x2121 的应该为 0xa1a1，即第一列值全部需要加 0x8080,这样就可以了。修改 make_enmap 给需要处理的地方每个值加 0x80。

第9章 Perl6 简介

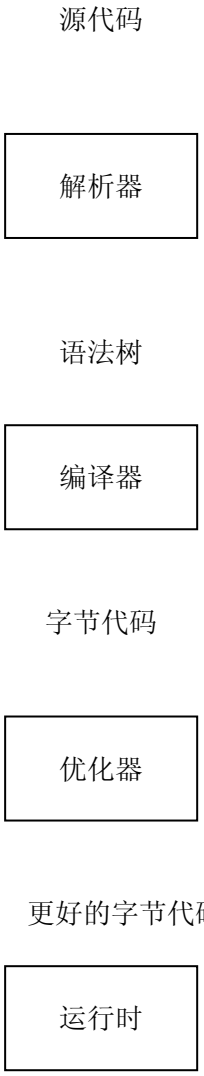
9.1 Perl6 体系结构

Perl6 是 Perl 程序设计语言的下一个版本。因为 Perl5 的解释器太混乱，影响了维护，阻碍了新特性的引进，吓跑了潜在的内部黑客。维护 Perl 程序和解释器变的困难重重。最后，Larry Wall 邀请整个 Perl 社团参与 Perl6 的设计与实现。

perl6 不是简单的 perl5 的重写。通过把解析、编译和运行时分开，这样就打开了多语言合作的大门。你能够用 perl6 或 perl5 或任何有解析器的其它语言。 内部可交换的运行时引擎让你解释你的字节码或把它转换语言(如 Java, C, 或反编译成 Perl)。

perl6 运行时叫做 Parrot，可以从 www.parrotcode.org 得到。Parrot 当前有三个以它为目标的编译器。Jako 和 Cola 是对应看起来象 C 和 Java 的语言，但它们只有有限的功能。

Perl6 体系结构如下图：


源代码
解析器
语法树
编译器
字节代码
优化器
更好的字节代码
运行时

9.2 Parrot

Parrot 是一个用于执行解释性语言的字节代码的高效虚拟机。Parrot 将会用于 Perl 6 编译器。同时，已经有了一个有部分的功能的 Perl 6 编译器和其他语言的编译器。

在最新的 Parrot 0.1.0 版本中，提供了对对象和多线程的支持。

所有的 parrot 变量都叫做 PMC(Parrot Magic Cookie)，它只在 parrot 内部可见。在 Parrot 外部只能通过 vtable 函数访问 PMC。vtable 函数的第一个参数是当前的解释器。第二个参数是 PMC 本身。

9.3 Perl6 语法

相对于 Perl5 而言，Perl6 语法的变化从基本的数据类型到操作符和正则表达式等。下面我们只介绍相对于程序结构上变化最重要的函数和对象。

9.3.1 函数

假定我们想要根据一些用户提供的标准能把一个列表划分成两个数组(然后就能知道它是"绵羊"或"山羊")。 我们将这个需要的子程序称为&part ——因为它划分一个列表成为两个部分。

在一般情况下，我们通过传入一个函数说明如何分开列表。然后 &part 为每个列表中的元素调用该子程序， 如果该子程序返回真就把元素放到"山羊"列表，否则就放到"绵羊"列表。 最后将返回一个指向两个结果数组的引用。

例如，调用：

($cats, $chattels) = part &is_feline, @animals;

将导致把@animals 中的猫科动物放到$cats， 而其他的动物放到$chattels。

函数&part 的 Perl 6 实现可以是：

```
sub part (Code $is_sheep, *@data) {
    my (@sheep, @goats);
    for @data {
        if $is_sheep($_) { push @sheep, $_ }
        else             { push @goats, $_ }
```

```
        }
        return (\@sheep, \@goats);
    }
```

和 Perl 5 中一样，关键字 sub 声明了一个函数。和 Perl 5 中一样，函数的名字紧跟着 sub - 假定名字不包括包的标识符 – 结果函数安装到了当前的包。

与 Perl 5 不同，在 Perl 6 中我们允许在函数的名字之后指定一张形参表。 这个列表由由零或更多的参数变量组成。 这些参数变量中的每一个实际上是真一个词法变量说明， 但是因为他们在一个参数表里，所以我们不需要(并且也不允许) 使用关键字 my。

正象用一个正式的变量，每个参数被给定一种贮存类型，注明它被允许存成哪种值。上例中，$is_sheep 形参是 Code 类型的， 表明第一个实参一定是一个函数或者块。

这些形参变量中的每个都自动限定到函数范围内，这样当函数调用时，通过它们来访问实参。

Perl 5 中也有参数，但是用户不能命名它们。它们总是叫做$_[0], $_[1], $_[2], 等。

9.3.2 对象

让我们举一个简单的例子来说明，Perl5 和 Perl6 在对象语法上的差别。

假设我们定义一个点（Point）对象，由于某种原因，只允许你调整 y 轴而不允许调整 x 轴。

```
class Point {
    has $.x;
    has $.y is rw;

    method clear () { $.x = 0; $.y = 0; }
}

my $point = Point.new(x => 2, y => 3);

$a = $point.y;          # 正确
$point.y = 42;          # 正确

$b = $point.x;          #正确
$point.x = -1;          #非法，缺省是只读的

$point.clear;           # 重置到原点 0,0
```

如果你比较它和 Perl 5 的写法，就会发现如下不同：

- 它使用关键词 class 和 method 而不是 package 和 sub。
- 属性是显式声明的而不是隐性的 hash 关键字。
- 因为额外的小圆点，将属性变量与普通的变量相混淆是不可能的。
- 或许是最重要的，我们不必为了为对象的值而不得不使用 hash (或者任何其他外部数据结构)。
- 我们不一定要写一个构造函数。
- 隐含的构造者自动知道怎样映射命名参数到属性名。
- 我们不一定要写附加的方法。
- 在对象外边，属性缺省是只读的。
- 请求 clear 方法是隐含的。
- 可能最明显的是，Perl 6 使用.而不是-> 来引用一个对象。

附录 A　命令行参数

开关	功能
-0[digits]	记录分割符。

-a	启用 autosplit，并使用空白作为缺省间隔符。
	例如：
	>perl -ane "print pop(@F), \"\n\";"
	等价于：
	while (<>) {
	@F = split(' ');
	print pop(@F), "\n";
	}
	如下命令打印文本文件的每行的第一列：
	perl -ane "print shift(@F), \"\n\"" < c:\test.txt

-C	启用本地宽字符系统接口。
-c	让 Perl 检查脚本的语法然后退出而不执行它。但是实际上，它会执行 BEGIN 和 END 块，和 use 块，因为 它们并不看成是程序的执行部分。
-d	调用调试程序。
-d:foo[=bar,baz]	在调试或跟踪模块的控制下运行脚本。例如，-d:DProf 使用 Devel::DProf 配置器执行脚本。
-Dletters	可以设置调试标志。
-Dnumber	可以设置调试标志。
-e commandline	执行命令行中开关之后的文本来写一行脚本。 例如，如下打印一行：

开关	功能
	>perl -e "print \"hello world\n\";" 在这里"前面的\代表转义。
-Fpattern	在分隔符之间使用指定的模式启用-a。 如下命令打印文本文件的每行的第一列，用,分隔： >perl –aF/,/ -ne "print shift(@F), \"\n\"" < c:\test.txt
-h	显示帮助信息。
-i[extension]	修改<>操作符。如果不提供 extension，就会覆盖当前文件。如果提供 extension，就会根据 extension 产生新文件，向新文件写入。 例如，如下命令备份文件 d:/temp.log 到 d:/temp.log.bak： perl -pi".bak" -e 1 d:/temp.log
-Idirectory	加一条搜索路径，用于搜索包含的文件。
-l[octnum]	自动化行结束处理，并可定义行结束符。打开此开关有两个影响：首先，当与-n 或-p 一起使用时，它自动的 chomp $/ (输入记录分隔符)；其次它给$\(the 输出记录分隔符)赋值成 octnum。如果忽略 octnum，将把$\设置成$/的当前值。 例如，如下代码把列截短成前 10 列。 perl -lpe "$_ = substr($_, 0, 10) ;" d:\temp.log
-m[-]module	导入给定模块。在执行程序之前执行 use module (); 。
-M[-]module	在执行程序之前执行 use module; 。
-M[-]'module ...'	如果 -M 或 –m 后的第一个字符是一个短横 (-) ，则 'use' 由 'no' 替换。
-[mM][-]module=arg[,arg]...	-mmodule=foo,bar 或者 -Mmodule=foo,bar 等价于 '-Mmodule qw(foo bar)'。
-n	使 Perl 在脚本上采用一个 while(<>){脚本}的循环。它等价于：

开关	功能

	while (<>) {
	
	... # your script goes here
	
	}
	
	下例相当于 type 命令：
	
	>perl -ne "print $_" filename

-p	使 Perl 采取一个 while(<>){脚本}的循环。这个选项和 -n 一样，只不过在每个循环后多了一个 print $_;
	
	while (<>) {
	
	... # your script goes here
	
	print;
	
	}
	
	因此
	
	>perl -pe ";" test.txt
	
	等价于
	
	>perl -ne "print $_" test.txt
	
	例如，下例替换一个单词(通过 stdout 输出，可以用来首先检查一下结果是否正确)：
	
	>perl -p -e "s/foo/bar/;" <myInputFile>

-P	使得程序在编译前通过 C 预处理程序运行。
-s	将变量的名字定义与紧跟命令行的开关相同。
-S	使用 PATH 环境变量寻找一个指定的程序文件。
-T	在执行不安全操作时中止数据进入程序。
-u	在程序编译后执行一个核心转储操作。
-U	允许不安全操作。

开关	功能
-v	打印当前使用的 Perl 的版本。
-V	打印出在编译中 Perl 用到的主要配置值的汇总纪录，同时打印出@INC 数组的值。
-V:name	打印出给定的配置变量的值。如： >perl -V:osname
-w	启用警告。如变量名在赋值前使用等。
-W	启用所有的警告。
-X	禁用警告。
-x directory	去掉以#!开头的程序行之前的无关文本。Perl 执行的代码以<CODE>__END__</CODE>结束。 例如下面这段代码保存在 test.txt 文件中，可以用 perl -x test.txt 执行： message #!perl print "aaa"; <CODE>__END__</CODE> message

附录 B　环境变量

变量	说明
HOME	如果调用 chdir 没有参数时使用该值。
LOGDIR	如果 chdir 没有参数而没有设置 HOME 环境变量时使用该值。
PATH	当执行子进程时使用，当使用-S 时用来发现脚本。
PERL5LIB	一个冒号分隔的路径的列表，在搜索标准库和当前路径之前通过它寻找 Perl 库文件。如果没有定义 PERL5LIB ，使用 PERLLIB。当运行 taint 检查时，两个变量都不用。脚本这时候应该写： use lib "/my/directory";
PERL5OPT	命令行选项 (开关)。这个变量中的开关比 Perl 命令行有更高的优先级。仅允许-[DIMUdmw] 开关。当运行 taint 检查时，忽略该变量。
PERLLIB	一个冒号分隔的路径的列表，在搜索标准库和当前路径之前通过它寻找 Perl 库文件。如果定义了 PERL5LIB ，就不用 PERLLIB。
PERL5DB	此命令用来加载调试代码。缺省是： BEGIN { require 'perl5db.pl' }
PERL5SHELL (仅限于 WIN32 版本)	可以设置一个可选的 shell，perl 内部必须使用来执行``backtick" 命令或 system()函数。在 WindowsNT 操作系统上，缺省是“cmd.exe /x/c”，而在 Windows95 上是“command.com /c”。这个值是空格分割的。可以使用一个反斜线对字符转义。
PERL_DEBUG_MSTATS	仅当如果 perl 编译时把 malloc 包含进来时使用 (即，如果“perl -V:d_mymalloc”是'define')。如果设置，它会导致在程序执行后 dump 内存统计。如果设置大于 1 的整数，也导致编译后 dump 内存统计。
PERL_DESTRUCT_LEVEL	仅当你的 perl 可执行文件使用-DDEBUGGING 编译时相关，它控制全局对象的析构和其它的引用的行为。

附录 C　特殊变量

缺省变量

变量	英文名	说明
$_	$ARG	缺省输入。 例如： $_ = "\$_ is the default for many operations including print().\n"; print;
@_		函数参数

正则表达式变量

这些变量都是局域变量，而且只读。

变量	英文名	说明
$<I<digit>> Short Name: $1, $2, ... $<N>	无	这些变量用于指和正则表达式匹配的字符串。在匹配的任何模式中，圆括号的集合用来表示子模式。这些子模式用数字从左至右标识。在一个匹配完成后，可以通过这些变量引用每一个子模式匹配。$1 是第一个子模式，$2 是第二个，依次类推，直到$<N>，指第 N 个子模式。 例如： $_ = "AlphaBetaGamma"; /^(Alpha)(.*)(Gamma)$/; print "$1 then $2 then $3\n";
$&	$MATCH	当字符串用于模式匹配时，字符串被分成了三部分：匹配以前的部分，匹配上的部分，匹配以后的部分。任何部分都可能是空，这个变量指最近一次匹配上的字符串。 例如：

变量	英文名	说明
		$_ = "AlphaBetaGamma"; /B[aet]*/; print "Matched: $&\n";
$'	$POSTMATCH	匹配部分以后的部分。 例如： $_ = "AlphaBetaGamma"; /Beta/; print "Postmatch = $'\n";
$`	$PREMATCH	最近一次匹配，匹配部分以前的部分。 例如： $_ = "AlphaBetaGamma"; /Beta/; print "Prematch = $`\n";
$+	$LAST_PAREN_MATCH	最后一个圆括号中的子表达式匹配的部分。大多数情况，只需要使用$1, $2, 等 ，而不需要用$+。当正则表达式中有一系列括号时，$+是有用的。 例如： $_ = "AlphaBetaDeltaGamma"; /Alpha(.*)Delta(.*)/; print "The last match was $+\n";
$*	$MULTILINE_MATCHING	缺省情况下，Perl 为了加快匹配速度，假设模式中不包括新行，也就是只执行单行匹配。如果要执行多行匹配，就要把此值设成 1。 例如： $_ = "Alpha\nBeta\nGamma\n";

变量	英文名	说明
		$* = 0; # Assume string comprises a single line /^.*$/; print "a) Assuming single line: $& (which is wrong - the assumption was wrong).\n"; $* = 1; # Do not assume string comprises a single line /^.*$/; print "a) Not assuming single line: $& (which is correct).\n"; $* = 0;
@+	@LAST_MATCH_END	这个数组保存当前匹配的最后成功子匹配的结尾的偏移量。$+[0]是整个匹配的偏移量。$+[1]是$1 结束的偏移量，$+[2]是$2 结束的偏移量。 例如： $_ = "AlphaBetaDeltaGamma"; /Alpha(.*)Delta(.*)/; print "The last match was @+\n";
@-	@LAST_MATCH_START	$-[0] 是最后一个成功的匹配的开始的偏移量。$-[n]是第 n 个子模式的偏移量，或 undef，如果没有匹配上的话。 $-[0]也可以看成是整个匹配开始的偏移量。$-[1]是$1 开始的地方，$-[2] 是$2 开始的地方，依次类推。 在对$var 做过匹配后： $` 等价于 substr($var, 0, $-[0]) $&等价于 substr($var, $-[0], $+[0] - $-[0]) $'等价于 substr($var, $+[0]) $1 等价于 substr($var, $-[1], $+[1] - $-[1]) $2 等价于 substr($var, $-[2], $+[2] - $-[2])

变量	英文名	说明
		$3 等价于 substr $var, $-[3], $+[3] - $-[3])

输入、输出变量

变量	英文名	说明
$.	$INPUT_LINE_NUMBER	最近一次执行读操作的当前行数。显式的关闭文件句柄重置行数。 例如： print "The last file read had $. lines\n";

$/	$INPUT_RECORD_SEPARATOR	输入记录分隔符，缺省值是新行。
		
		例如，有文件 infor.txt,内容是 owens 1001 love。想用 open(PF,'infor.txt')来读取，但是只想读取 owens，不取后边的两项，可以：
		
		open(PF,'infor.txt');
		
		$/= ' ';
		
		$line =<PF>;
		
		chop($line);
		
		print $line;
		
		函数 chomp 截短的回车符的定义也是取自变量”$/”。例如：
		
		$s = 'hello world';
		
		$/=' world';
		
		chomp $s;
		
		print $s;

$,	$OUTPUT_FIELD_SEPARATOR	print 操作的输出域分隔符。

变量	英文名	说明

$\	$OUTPUT_RECORD_SEPARATOR	print 操作的输出记录分隔符。通常用于省略换行符。
		$\ = "\n";
		#No need for an explicit newline now.";
		foreach my $item (@sql_lines)
		{
		print OUT "$item";
		}
		$\ = "";

$”	$LIST_SEPARATOR	当数组转换成字符串时，元素缺省以空格分隔(例如，当打印数组时)。这个变量即代表这个分隔符，缺省是空格。 例如： $" = ' ! '; @thisarray = (Alpha, Beta, Gamma); print "@thisarray.\n"; $" = ' ';
$^L	$FORMAT_FORMFEED	当执行一个进纸动作时输出的字符。缺省是 \f。
$:	$FORMAT_LINE_BREAK_CHARACTERS	就是目前可以作为折行的字符集合。缺省值是” \n“(也就是空白，换行字符，以及连字号)。
$^A	$ACCUMULATOR	格式化行的写收集器的当前值。 例如： $tmp = formline<<'FINISH', Alpha, Beta, Gamma; @<<<<<<<<<< @|||||||||| @<<<<<<<<<< FINISH

变量	英文名	说明
		print "Accumulator now contains:\n $^A\n"; $^A = "";

错误变量

变量	英文名	说明
$?	$CHILD_ERROR	包含了最近一次执行的外部程序结束状态。这些程序以办是通过管道，反小点 (') 或 system 函数执行的。
$!	$OS_ERROR, $ERRNO	包含了系统的错误。如果用在数值的地方，就是系统错误码；如果用在字符串的地方，就是错误信息字符串。
$^E	$EXTENDED_OS_ERROR	在某些平台，返回扩展错误信息。
$@	$EVAL_ERROR	从上一个 eval 命令的 Perl 语法错误信息。

系统变量

变量	英文名	说明
$$	$PROCESS_ID $PID	运行当前脚本的 Perl 进程的 pid。
$<	$REAL_USER_ID $UID	当前进程的实际用户标识符(uid)。
$>	$EFFECTIVE_USER_ID $EUID	当前进程的有效用户标识符。
$(	$REAL_GROUP_ID $GID	当前进程的实际组标识符(gid)。
$)	$EFFECTIVE_GROUP_ID $EGID	当前进程的有效组标识符。

变量	英文名	说明
$0	$PROGRAM_NAME	正在执行的 Perl 脚本的文件名称。这个参数与执行时输入有关。例如，当执行时输入： >perl c:/perl/test.pl 此时，$0 是 c:/perl/test.pl。 当执行时输入： >perl test.pl 此时，$0 是 test.pl。
$[		数组中第一个元素的序号或子串中第一个字符的序号。缺省是 0。
$]	$PERL_VERSION	返回版本号，加上补丁级别除以 1000。 例如： print("\nTest Perl Version ($])\n");
$^D	$DEBUGGING	调试标志的当前值。
$^F	$SYSTEM_FD_MAX	最大的系统文件描述符，通常是 2。
$^I	$INPLACE_EDIT	原地编辑扩展的当前值。可使用 undef 禁止原地编辑。
$^M		$M 的内容能用作紧急内存池，以便 Perl 出 out-of-memory 错误时使用。使用$M 要求 Perl 进行特殊的编译。
$^O	$OSNAME	编译 Perl 本身时的操作系统名称。
$^P	$PERLDB	是否打开调试。
$^T	$BASETIME	当前脚本开始运行的时间。以秒为单位，从 1970 年开始。
$^W	$WARNING	警告开关的当前值，真或假。
$^X	$EXECUTABLE_NAME	二进制 Perl 执行文件的名称。

变量	英文名	说明
$ARGV		当从< >读入时的当前文件名。

其它

变量	说明
@ARGV	命令行参数。
$ARGV	当前文件的文件名，代表标准输入<STDIN>。
@INC	寻找 Perl 脚本的地址表。
%INC	通过 do 或 requir 包含的文件名的目录。

附录 D 预编译指令

传递给编译器的指令叫做预编译指令。预编译指令可以打开，例如使用严格的语法检查：

use strict;

或者关闭，例如如下语句关闭 integer 预编译指令：

no integer;

名称	说明
integer	强制使用整型而不是浮点或双精度运算。
sigtrap	当碰到不可预期的信号时，允许堆栈返回。
strict	限制不安全的结构。强烈推荐采用此预编译指令。它包含如下三件事情：vars、subs 和 refs： use strict 'vars'; 你必须预声明变量，使用全称(如：$package::foo)或者 import 它。实际上，你会发现预声明变量是非常重要的。 use strict 'subs'; 限制使用裸词调用一个例程，除非已经定义了该例程 (例如，限制使用 "somesub" 而不是"&somesub" 或者 "somesub()")。 use strict 'refs'; 这个预编译指令禁止你使用符号引用。
subs	用于预定义函数名。

参考资源

9.4 书籍

Peter Wainwright .”Professional Perl Programming” 2001

“Professional Development with Perl” 2001

Tim Bunce. “Perl for Oracle-Tools and Technologies”2002

Larry Wall, Tom Christiansen, and Jon Orwant."Programming Perl", 3rd Edition

9.5 网址

美国

网址	说明
www.perl.com	Perl 主站点。
www.activestate.com	Windows 下的 perl。
www.cpan.org	Perl 模块。
www.perldoc.com	Perl 文档。
www.pm.org	Perl 用户组织。
use.perl.org	Perl 文章。
p5ee.perl.org	指导如何使用 perl 来构建企业级应用的网站。
perl.apache.org	mod_perl 的发源地。
www.perlfoundation.org	Perl 基金会。
www.perlmonks.org	Perl 论坛。
www.perlmonth.com	合并入 Open Source Digest (www.opensourcedigest.com)
dev.perl.org	开发 Perl 语言。

国内

网址	说明
www.perlchina.net	中国 perl 协会网站。
www.ilcatperl.org	专业的 Perl 技术文献。
www.chinaunix.com	有一个 perl 论坛。