

oct **oct** **EXPR** **oct**

这个函数把 **EXPR** 当作一个八进制字符串并且返回相等的十进制值。如果 **EXPR** 碰巧以 “0x” 开头，那么它就会被当作一个十六进制字符串看待。如果 **EXPR** 以 “0b” 开头，那么它就解释成一个二进制数的字符串。下面的代码将把任何以标准的 C 或 C++ 符号写的十进制，二进制，八进制，和十六进制输入字符串转换成数字：

```
$val = oct $val if $val =~ /^0/;
```

要实现相反的功能，使用对应格式的 **sprintf**：

```
$perms = (stat("filename"))[2] & 07777; $oct_perms = sprintf "%lo", $perms;
```

oct 函数常用于这样的场合，比如你需要把一个 “644” 这样的字符串转换成一个文件模式等等。尽管 Perl 会根据需要自动把字符串转换成数字，但是这种自动转换是以 10 为权的。

open **open** **FILEHANDLE**, **MODE**, **LIST** **open** **FILEHANDLE**, **EXPR** **open** **FILEHANDLE**

open 函数把一个内部的 **FILEHANDLE** 与一个 **EXPR** 或 **LIST** 给出的外部文件关联起来。你可以用一个，两个，或者三个参数调用它（或者更多参数——如果第三个参数是一条命令，而且你运行的 Perl 至少是 5.6.1）。如果出现了三个或者更多个参数，那么第二个参数声明这个文件打开的访问模式 **MODE**，而第三个参数（**LIST**）声明实际要打开的文件或者要执行的命令——具体是什么取决于模式。如果是一条命令，而且你想直接调用该命令而不调用 **shell**（象 **system** 或者 **exec** 那样），那么你还可以提供额外的参数。或者该命令可以作为一个参数提供（第三个），这个时候是否调用 **shell** 取决于该命令是否包含 **shell** 元字符。（如果这些参数是普通文件，不要使用超过三个参数的形式；那样没有作用。）如果无法识别 **MODE**，那么 **open** 抛出一个例外。

如果只提供了两个参数，那么就假设模式和文件名/命令一起组合在第二个参数里。（并且如果你没有在第二个参数里声明模式，而只是一个文件名，那么该文件就会以只读方式打开，安全第一。）

如果只有一个参数，和 **FILEHANDLE** 同名的包标量变量必须包含文件名和可选的模式：

```
$LOG = ">logfile"; # $LOG 不能是定义过的 my! open LOG or die "Can't open logfile: $!";
```

不过别干这种事。它不合风格。别记着它。

在操作成功的时候 **open** 返回真，否则返回 **undef**。如果 **open** 打开一个到子进程的管道，那么它的返回值将是那个新进程的进程 ID。和所有的系统调用一样，一定要记得检查 **open** 的返回值，看看它是否运转正常。不过我们不是 C 也不是 Java，所以如果 **or** 操作符能用的时候不要使用 **if** 语句。你还可以使用 **||**，不过如果你用 **||**，那么在 **open** 上加圆括弧。如果你省略了函数调用上的圆括弧，把它变成一个列表操作符，那么要注意在该列表后面用 “or die” 而不是 “|| die”，因为 **||** 的优先级比象 **open** 这样的操作符高，因此 **||** 会绑定你的最后一个参数，而不是整个 **open**：

```
open LOG, ">logfile" || die "Can't create logfile: $!"; # 错
open LOG, ">logfile" or die "Can't create logfile: $!"; # 对
```

上面的代码看起来太紧密了，不过通常你都会用一些空白来告诉你的眼睛该列表操作符在哪里终结：

```
open LOG, ">logfile" or die "Can't create logfile: $!";
```

正如本例显示的那样，该 **FILEHANDLE** 参数通常只是一个简单的标识符（通常是大写），但是它也可以是一个表达式，该表达式的值提供一个指向实际文件句柄的引用。（该引用可以是一个指向文件句柄名字的符号引用，也可以是一个指向任何可以解释成一个文件句柄的对象的硬引用。）这种文件

句柄叫做间接文件句柄，并且任意拿一个 **FILEHANDLE** 做其第一个参数的函数都可以象操作直接文件句柄那样操作间接文件句柄。不过 **open** 有一个特殊的地方，就是如果你给它一个未定义的变量做间接文件句柄，那么 **Perl** 会自动为你定义那个变量，也就是自动把它激活，使它包含一个合适的文件句柄引用。这样做的一个好处就是如果没有谁再引用它，那么该文件句柄将被自动关闭，一般是在该变量超出了范围之后：

```
{ my $fh; # (未初始化) open($fh, ">logfile") # $fh 被自动激活 or die "Can't create logfile: $!"; ... # 干点别的 } # $fh 在这里关闭
```

my \$fh 声明可以在保证可读性的前提下集成到 **open** 里：

```
open my $fh, ">logfile" or die ...
```

你在这里看到的文件名字前面的 **>** 符号就是一个模式的例子。从历史来看，首先出现的是两个参数的 **open** 的形式。最近新增加的三个参数的形式让你把模式和文件名分隔开，这样的好处就是避免在它们之间任何可能的混淆。在随后的例子里，我们知道用户不是想打开一个碰巧是以 **>** 开头的文件名。我们可以确信他们说的是一个 **MODE** **>**，这个模式是打开名字是 **EXPR** 的文件用于写，如果该文件不存在则创建之，如果存在则先把它截断成零长度：

```
open(LOG, ">", "logfile") or die "Can't create logfile: $!";
```

在上面的短一些的形式（两个参数的那个）里，文件名和模式在同一个字符串里。该字符串是用类似典型 **shell** 处理文件和管道重定向的方法分析的。首先，删除字符串里任何前导的和后跟的空白。然后根据需要在字符串两端搜索那些声明该文件应该如何打开的字符。在文件名和空格之间是允许空白的。

表示如何打开一个文件的模式是类 **shell** 的重定向符号。在表 29-1 里有一个这样的符号的列表。

（如果要用某种此表没有提到的组合模式访问文件，那么请参阅低层的 **sysopen** 函数。）

表 29-1

模式 | 读访问 | 写访问 | 只附加 | 不存在时创建 | 删除现有的

... (略)

如果模式是 **<** 或者什么都没有，那么则打开一个现有的文件做输入。如果模式是 **>**，那么该文件打开用于输入，会清空现有文件并且创建不存在的文件。如果模式是 **>>**，那么根据需要创建该文件并且为附加数据而打开，并且所有输出都自动放到文件结尾。如果因为你使用了 **>** 或 **>>** 这样的模式创建了一个新的文件，而且该文件原先并不存在，那么访问该文件的权限将取决于该进程当前的 **umask**，并且遵守该函数 (**umask**) 描述的规则。

下面是几个常见的例子：

... (略)

如果你喜欢标点少的版本，你可以写：

... (略)

如果打开用于读取，那么特殊的文件名 **<-** 指的是 **STDIN**。如果用于写而打开，那么这个特殊的文件名指的是 **STDOUT**。通常，它们可以分别声明为 **<-** 和 **>-**：

```
open(INPUT, "<-") or die; # 重新打开标准输入用于读取 open(INPUT, "<-") or die; # 同样的事情，不过是明确声明 open(OUTPUT, ">-") or die; # 重新打开标准输出用于写
```

这样，用户就可以使用一个带文件名的程序，该程序可以使用标准输入或者标准输出，但程序的作者并不需要写特殊的代码来为此做准备。

你还可以在任何这三种模式前面加一个“+”以请求同时的读和写。不过，该文件是清空还是创建，以及是否它必须已经存在仍然是由你选用的大于号或者小于号来决定的。这就意味着“+<”几乎总是会读/写更新，而不确定的“+>”模式会在你能从文件中读取任何东西之前先清空该文件。（只有在你只想重新读取你刚刚写进去的东西的时候使用这个模式。）

```
open(DBASE, "+< database") or die "can't open existing database in update mode: $!";
```

你可以把一个打开了准备更新的文件当作一个随机访问的数据库，并且使用 `seek` 移动到特定的字节数处，但是普通文本文件里记录是变长的性质，通常会让你不可能利用读写模式更新这样的文件。参阅第十九章里的 `-i` 命令行选项获取一种更新的不同的方法。

如果 `EXPR` 里的前导字符是一个管道符号，`open` 启动一个新的进程并且把一个只写的文件句柄联接到该命令。这样你就可以写到那个句柄，并且你写的东西将在那个命令的标准输入里显示。比如：

```
open(PRINTER, "| lpr -Plp1") or die "can't fork: $!"; print PRINTER "stuff\n"; close
(PRINTER) or die "lpr/close failed: $?/$!";
```

如果 `EXPR` 的后跟的字符是一个管道符号，`open` 还是会启动一个新的进程，不过这次是用一个只读的文件句柄与之相联。这样就允许把该命令写到它的标准输出的东西在你的句柄里显示出来用于读取。比如：

```
open(NET, "netstat -i -n |") or die "can't fork: $!"; while () { ... } close(NET) or die
"can't close netstat: $!/$?";
```

明确地关闭任何任何管道文件句柄都导致父进程等待子进程完成并且在 `$? ($CHILD_ERROR)` 里返回状态码。我们也可以让 `close` 设置 `$! ($OS_ERROR)`。参阅 `close` 和 `system` 里的例子获取如何解释这些错误代码的信息。

任何包含 `shell` 元字符（比如通配符或 `I/O` 重定向字符）的管道命令都会传递给你的系统规范 `shell`（在 `Unix` 里是 `/bin/sh`），所以那些与管道相关的构造可以先处理。如果没有发现元字符，`Perl` 就自己启动新进程，而不调用 `shell`。

你还可以使用三个参数的形式来启动管道。使用该风格的和前面几个管道打开等效的代码是：

```
open(PRINTER, "|-", "lpr -Plp1") or die "can't fork: $!"; open(NET, "-|", "netstat -i -n")
or die "can't fork: $!";
```

在这里，第二个参数里的负号代表在第三个参数里的命令。这些命令碰巧不会调用 `shell`，但是如果你想保证不会调用 `shell`，你可以在新版本的 `Perl` 里说：

```
open(PRINTER, "|-", "lpr", "-Plp1") or die "can't fork: $!"; open(PRINTER, "-|",
"netstat", "-i", "-n") or die "can't fork: $!";
```

如果你使用两个参数的形式打开一个管道读进或写出这个特殊的命令“-”，（注：或者你可以把它当作在上面的三个参数形式中没有写命令。）那么先会隐含到做一个 `fork`。（在那些不能 `fork` 的系统上，这样会抛出一个例外。在 `Perl 5.6` 以前，`Microsoft` 系统不支持 `fork`。）在本例中，负号代表你新的子进程，它是父进程的一个拷贝。如果在父进程里看，从这个派生式的 `open` 返回的返回值是子进程的 `ID`，而从子进程里看是 `0`，而如果 `fork` 失败则返回 `undef`——这个时候，没有子进程存在。比如：

```
defined($pid = open(FROM_CHILD, "-|")) or die "can't fork: $!";
```

```
if ($pid) { @parent_lines = ; # 父进程代码 } else { print STDOUT @child_lines; # 子进程
代码 }
```

这个文件句柄的行为对于父进程来说是正常的，但对于子进程，父进程的输入（或输出）是取自（或者送到）子进程的 **STDOUT**（或者 **STDIN**）的。子进程看不到父进程的文件句柄的打开。（用 **PID 0** 就方便标出。）通常，如果你想对管道彼端的命令是如何执行的做更多的控制（比如你运行着 **setuid**），或者你不象对 **shell** 脚本扫描查找元字符，那么你就会愿意用这个构造取代普通的管道 **open**。下面的管道打开基本上是相同的：

```
open FH, "| tr 'a-z' 'A-Z'"; # 管道输出给 shell 命令
open FH, "|-", 'tr', 'a-z', 'A-Z'; # 管道输出给光命令
open FH, "|-" or exec 'tr', 'a-z', 'A-Z'; # 管道输出给子进程
```

以及这些：

```
open FH, "cat -n 'file' |"; # 从 shell 命令管道取来
open FH, "-|", 'cat', '-n', 'file'; # 从光命令管道取来
open FH, "-|" or exec 'cat', '-n', 'file' or die; # 从子进程取来
```

有关派生打开的更灵活的使用的信息，参阅第十六章的“自言自语”一节和第二十三章，安全性，里的“清理你的环境”一节。

如果用 **open** 开始一条命令，你必须选择输入还是输出：“**cmd|**”是读取，“**|cmd**”是写出。你不能用 **open** 打开一个同时用管道定向输入和输出的命令，象下面这样（目前）非法的符号，

“**|cmd|**”，写的那样。不过，标准的 **IPC::Open2** 和 **IPC::Open3** 库过程给你一个非常接近的等价物。有关双头管道的细节，请参阅第十六章里的“双响通讯”一节。

你还可以象在 **Bourne shell** 里的传统一样，声明一个以 **>&** 开头的 **EXPR**，这种情况下该字符串其他的部分解释成一个将要调用 **dup2(2)** 系统调用（注：目前这个方法不能用于通过自动激活文件句柄引用的类型团 **I/O** 对象，不过你总是可以用 **fileno** 取出文件描述符并且复制那个东西。）复制的文件句柄的名字（或者文件描述符，如果它是数字）。你可以在 **>**，**>>**，**<**，**++**，**++>**，和 **++<** 后面使用 **&**。（声明的模式应该与最初的文件句柄的模式相匹配。）

你想这么做的一个原因可能是因为你已经有一个打开的文件句柄，并且想做另外一个句柄，而该句柄是前面一个的真正的复制品。

```
open(SAVEOUT, ">&SAVEERR") or die "couldn't dup SAVEERR: $!";
open(MHCONTEXT, "<&4") or die "couldn't dup fd4: $!";
```

这意味着如果一个函数期待一个文件名，但你不给它一个文件名，因为你已经有一个打开的文件了，那么你只要把文件句柄前面带一个与号传递给他好了。不过，你最好用一个全称的句柄，以防该函数偏巧在另外一个包里：

```
somefunction("&main::LOGFILE");
```

另外一个“**dup**”文件句柄的理由是临时重定向一个现有的文件句柄而不用丢失最初那个的目的地。下面是一个保存，重定向，并恢复 **STDOUT** 和 **STDERR** 的脚本：

! /usr/bin/perl

```
open SAVEOUT, ">&STDOUT"; open SAVEERR, ">&STDERR";
```

```
open STDOUT, ">foo.out" or die "Can't redirect stdout";
open STDERR, ">&STDOUT" or die "Can't dup stdout";
```

```
select STDERR; $| = 1; # 允许自动冲刷
select STDOUT; $| = 1; # 允许自动冲刷
```

```
print STDOUT "stdout 1\n"; # 这些 I/O 流也传播到
print STDERR "stderr 1\n"; # 子进程
```

```
system("some command"); # 使用新的stdout/stderr
```

```
close STDOUT; close STDERR;

open STDOUT, ">&SAVEOUT"; open STDERR, ">&SAVEERR";

print STDOUT "stdout 2\n"; print STDERR "stderr 2\n";
```

如果该文件句柄或描述符号是前导一个 `&=`，而不是单单一个 `&`，那么这次不是创建一个完全新的文件描述符，而是 Perl 把 `FILEHANDLE` 作成一个现有的描述符的别名，用的是 C 库调用 `fdopen` (3)。这样稍微更节约系统资源一些，尽管现在人们已经不太关心这个了。

```
$fd = $ENV{"MHCONTEXTFD"}; open(MHCONTEXT, "<&=fdnum") or die "couldn't
fdopen descriptor $fdnum: $!";
```

文件句柄 `STDIN`，`STDOUT`，和 `STDERR` 在跨 `exec` 中总是保持打开状态。缺省时，其他文件句柄不是这样的。在那些支持 `fcntl` 函数的系统上，你可以为一个文件句柄修改 `exec` 时关闭的标志。

```
use Fcntl qw(F_GETFE F_SETFD); $flags = fcntl(FH, F_SETFD, 0) or die "Can't clear
close-on-exec flag on FH: $!\n";
```

又见第二十八章里的特殊变量 `$^F` (`$SYSTEM_FD_MAX`)。

对于一个或者两个参数的 `open` 形式，当你拿一个字串变量做文件名的时候必须小心，因为该变量可能包含任何古怪的字符（特别是当文件名是通过互联网获取的时候，这时它包含许多古怪的字符。）如果你不仔细，那么该文件名的一部分可能解释成一个 `MODE` 字串，或者一个可忽略的空白，一个复制声明，或者一个负号。下面是一个历史上很有趣的隔离自己的方法：

```
$path =~ s#^(\\s)#.\\$!#; open(FH, "< $path\\0") or die "can't open $path: $!";
```

不过这个方法仍然在许多方面有缺陷。你应该使用三个参数的 `open` 清晰地打开任何文件名，而又不担心有任何安全问题：

```
open(FH, "<", $path) or die "can't open $path: $!";
```

另一方面，如果你想要的是一个真正的 C 风格的 `open(2)` 系统调用，以及还有它的所有问题和警告，那么可以使用 `sysopen`：

```
use Fcntl; sysopen(FH, $path, O_RDONLY) or die "can't open $path: $!";
```

如果你在那些区分文本和二进制文件的系统上运行，你可能需要把你的文件句柄置于二进制模式——或者干脆别这么干，因为这个模式可能破坏你的文件。在这样的系统上，如果你在文本模式上操作二进制文件，或者在二进制模式上操作一个文本文件，那么你可能不会喜欢看到的结果。

那些需要 `binmode` 函数的系统和那些不需要该函数的系统的区别是在文本文件使用的格式上。那些不需要它的系统用一个字符结束每一行，这个字符对应 C 里面的换行符，`\n`。Unix 和 Mac 落在这个范围里。VMS，MVS，MS-XXX，以及 S&M 操作系统以及其他变种认为在文本文件上的 I/O 和二进制文件上的是不同的，所以它们需要 `binmode`。

或者其等价物。在 Perl 5.6 里，你可以在 `open` 函数里声明二进制模式而不用单独调用 `binmode`。作为 `MODE` 参数的一部分（但是只在三个参数的形式里），你可以声明各种输入和输出纪律。要实现和 `binmode` 一样的功能，使用三个参数的 `open` 形式，并且在其他 `MODE` 字符后面放上一条纪律 `:raw`：

```
open(FH, "<:raw", $path) or die "can't open $path: $!";
```

因为这是一种非常新的特性，所以到你阅读到这些的时候可能已经比我们写这些的时候有更多纪律了。不过，我们可以合理地推测所有纪律可能是表 29-2 的部分或全部：

表29-2 I/O 纪律

纪律 | 含义

:raw | 二进制模式；不做处理 **:text** | 缺省文本处理 **:def** | **use open** 的缺省声明 **:latin1** | 文件应该是 ISO-8859-1 **:ctype** | 文件应该是 LC_CTYPE **:utf8** | 文件应该是 UTF-8 **:utf16** | 文件应该是 UTF-16 **:utf32** | 文件应该是 UTF-32 **:uni** | 直观 Unicode (UTF-*) **:any** | 直观 Unicode/Latin1/LC_CTYPE **:xml** | 使用文件中声明的编码 **:crlf** | 直观换行符 **:para** | 段落模式 **:slurp** | 吞噬模式

你应该可以堆叠那些堆叠在一起有意义的纪律，所以，你可以说：

```
open(FH, "<:para:crlf:uni", $path) or die "can't open $path: $!"; while ($para = )
{ ... }
```

这样将设立纪律如下：

- * 如果该文件已经是 UTF-8，则读进一些 Unicode 的数据并转换成 Perl 内部的 UTF-8 格式。
- * 寻找行结尾的序列变体，把它们转换成 \n，并且
- * 把文件处理成段落大小的块，类似 `$/= ""` 干的事情。

如果你想设置缺省打开模式（**:def**）来做一些 **:text** 以外的事情，你可以用 **open** 用法在你的文件的顶端声明它：

```
use open IN => ":any", OUT => ":utf8";
```

实际上，如果哪一天这就是缺省的 **:text** 纪律，那就太好了。因为它完美地体现了“严于律己，宽以待人”的概念。

opendir opendir DIRHANDLE, EXPR

这个函数打开一个叫 **EXPR** 的目录给 **readdir**, **telldir**, **seekdir**, **rewinddir**, 和 **closedir** 处理。如果成功该函数返回真。目录句柄有自己的名字空间，和文件句柄是分离的。

ord ord EXPR ord

这个函数返回 **EXPR** 的第一个字符的数字值（ASCII, Latin-1, 或者 Unicode）。返回值总是无符号的。如果你需要有符号值，使用 **unpack('c', EXPR)**。如果你希望字符串中的所有字符都转换成一系列数字，使用 **unpack('U*', EXPR)**。

our our TYPE EXPR : ATTRIBUTES our EXPR : ATTRIBUTES our TYPE EXPR our EXPR

一个 **our** 声明一个或多个在当前闭合块，文件或者 **eval** 里有效的全局变量。也就是说，**our** 和 **my** 声明在可视性判断上有着一样的规则，只不过不创建一个新的私有变量；它仅仅是允许对现有包全局变量的无限制的访问。如果列出了多于一个数值，那么该列表必须放在圆括弧里。

使用 **out** 声明的一个主要用途就是隐藏变量，使之免于 **use strict "vars"** 声明的影响；因为该变量伪装成了一个 **my** 变量，所以就允许你使用声明了的全局变量而不必使用带着包名的全称。不过，和 **my** 声明一样，它只能在 **our** 声明的词法范围里生效。在这方面，它和 **use vars** 不一样，后者会影响整个包，而不仅仅是词法范围。

our 和 **my** 类似的地方还有就是你可以带着 **TYPE** 和 **ATTRIBUTE** 声明变量。下面是语法：

```
our Dog $spot :ears(short) :tail(long);
```

到我们写这些的时候，它的含义还不是非常清楚。属性可以影响 `$spot` 的全局的或者局部的含义。一方面，它又很象用属性的 `my` 变量，把当前 `$spot` 的局部外观封装起来而又不干涉该全局量在其他地方的外观。另一方面，如果一个模块把 `$spot` 声明为 `Dog`，而另外一个模块把 `$spot` 声明为 `Cat`，那么最后你就可能会有喵喵叫的狗和汪汪叫的猫。这是一个需要研究的课题，当然这只不过是不知道应该说些什么的有趣的说法而已。（除了一方面，就是我们的确知道在该变量指向一个伪散列的时候，我们应该如何处理 `TYPE` 声明——参阅第十二章的“管理实例数据”。）

`our` 和 `my` 相似的另外一个地方就是可视性。一个 `our` 声明一个全局变量，该变量将在其整个词法范围都可见，甚至是跨包的边界。该变量位于哪个包是用声明的位置判断的，而不是在使用的位置。这意味着下面的行为是有问题的并且是注定要失败的：

```
package Foo; our $bar; # 对于剩下的词法范围而言，$bar 是 $Foo::bar $bar = 582;
```

```
package Bar; print $bar; # 打印 582，就好象“our”是“my”一样
```

不过，`my` 会创建一个新的，私有的变量而 `our` 暴露一个现有的全局变量，这个区别非常重要，特别是在赋值的时候。如果你把一个运行时赋值和一个 `our` 声明结合起来，那么该全局变量的值不会在 `our` 超出范围之后消失。如果希望它消失，你应该使用 `local`：

```
($x, $y) = ("one", "two"); print "before block, x is $x, y is $y\n"; { our $x = 10; local
our $y = 20; print "in block, x is $x, y is $y\n"; } print "past block, x is $x, y is $y\n";
```

打印出：

```
before block, x is one, y is two in block x is 10, y is 20 past block, x is 10, y is two
```

在同一个词法范围里多个 `our` 声明是允许的，条件是它们在不同的包里。如果它们碰巧在同一个包里，Perl 会应你之邀发出警告。

```
use warnings; package Foo; our $bar; # 为剩余的词法范围声明 $Foo::bar $bar = 20;
```

```
package Bar; our $bar = 30; # 为剩余的词法范围声明 $Foo::bar print $bar; # 打印 30
```

```
our $bar; # 发出警告
```

又见 `local`，`my`，和第四章的“范围声明”一节。

`pack` `pack` `TEMPLATE`, `LIST`

这个函数接收一个普通 Perl 数值的 `LIST` 并且根据 `TEMPLATE` 把它们转换成一个字节串并且返回该字串。参数列表在必要时会被填充或者截除。也就是说，如果你提供的参数比 `TEMPLATE` 要求的少，`pack` 假设缺的都是空值。如果你提供的参数比 `TEMPLATE` 要求的多，那么多余的参数被忽略。在 `TEMPLATE` 里无法识别的格式元素将会抛出一个例外。

这个模板把该字串的结构描述成一个域序列。每个域都由一个字符代表，描述该值的类型和其编码。比如，一个格式字符 `N` 声明一个四个字节的无符号整数，字节序是大头在前。

域是以模板中给出的顺序包装的。比如，如果要把一个一字节的无符号整数和一个单精度浮点数值包装到一个字串里，你要说：

```
$string = pack("CF", 244, 3.14);
```

返回的字串的第一个字节的值是 244。剩下的字节是 3.14 作为单精度浮点数的编码。浮点数的具体编码方式取决于你的计算机的硬件。

有些包装时要考虑的重要事情是：

- * 数据的类型（比如是整数还是浮点还是字符串），
- * 数值的范围（比如你的整数是否放在一个，两个，四个，或者甚至八个字节里；或者你包装的是一个 8 位字符还是 Unicode 字符。），
- * 你的整数是有符号还是无符号，以及
- * 使用的编码（比如说本机，包装位和字节时小头在前，或者是大头在前）。

表 29-3 列出了格式字符以及它们的含义。（其他字符也可能在格式中出现；它们在稍后介绍。）

表29-3, pack/unpack 的模板字符

字符 | 含义

a | 一个填充空的字节串 **A** | 一个填充空格的字节串 **b** | 一个位串，在每个字节里位的顺序都是升序
B | 一个位串，在每个字节里位的顺序都是降序 **c** | 一个有符号 char（8位整数）值 **C** | 一个无符号 char（8位整数）值；关于 Unicode 参阅 **U** **d** | 本机格式的双精度浮点数 **f** | 本机格式的单精度浮点数 **h** | 一个十六进制串，低四位在前 **H** | 一个十六进制串，高四位在前 **i** | 一个有符号整数值，本机格式 **I** | 一个无符号整数值，本机格式 **l** | 一个有符号长整形，总是 32 位 **L** | 一个无符号长整形，总是 32 位 **n** | 一个 16位短整形，“网络”字节序（大头在前） **N** | 一个 32 位短整形，“网络”字节序（大头在前） **p** | 一个指向空结尾的字符串的指针 **P** | 一个指向定长字符串的指针
q | 一个有符号四倍（64位整数）值 **Q** | 一个无符号四倍（64位整数）值 **s** | 一个有符号短整数值，总是 16 位 **S** | 一个无符号短整数值，总是 16 位 **u** | 一个无编码的字符串 **U** | 一个 Unicode 字符数字 **v** | 一个“VAX”字节序（小头在前）的 16 位短整数 **V** | 一个“VAX”字节序（小头在前）的 32 位短整数 **w** | 一个 BER 压缩的整数 **x** | 一个空字节（向前忽略一个字节） **X** | 备份一个字节 **Z** | 一个空结束的（和空填充的）字节串 **@** | 用空字节填充绝对位置

你可以在你的 **TEMPLATE** 里自由地使用空白和注释。注释以惯用的 **#** 符号开头并且延伸到 **TEMPLATE** 里第一个换行符（如果存在）。

每个字母后面都可以跟着一个数字，表示 **count**（计数），解释成某种形式的重复计数或者长度，具体情况取决于格式。除了 **a**, **A**, **b**, **B**, **h**, **H**, **P**, 和 **Z** 之外，所有格式的 **count** 都是重复次数，因此 **pack** 从 **LIST** 里吃进那么多数值。如果 **count** 是一个 ***** 表示剩下的所有东西。

a, **A** 和 **Z** 格式只吃进一个数值，但是把它当作一个长度为 **count** 的字节串打包，并根据需要填充空或者空格。在解包的时候，**A** 抽去结尾的空格和空，**Z** 抽去第一个空后面的所有东西，而 **a** 原封不动地返回文本数据。打包的时候，**a** 和 **Z** 是相同的。

与之类似，**b** 和 **B** 格式打包一个长度为 **count** 的位串。输入域里的每个字节都以每个输入字节的最低位（也就是 `ord($byte) % 2`）为基础生成结果中的 1 个位。方便一点来说，这意味着字节 0 和 1 生成位 0 和 1。从输入字符串的开头开始，每 8 个字节转换成一个字节的输出。如果输入字符串的长度不能被 8 整除，那么余下的部分用 0 补齐。类似，在 **unpack** 的时候，任何额外的位都被忽略。如果输入字符串比需要的长，那么多出来的部分被忽略。**count** 如果是 ***** 意思是使用输入域的所有字节。在解包的时候，这些位转换成一个 0 和 1 组成的字符串。

h 和 **H** 格式打包一个由 **count** 个半字节（4 位组，常用于代表十六进制位。）组成的字符串。

p 格式打包一个指向一个空结尾的字符串。你有责任确保该字符串不是一个临时数值（因为临时值可能在你能使用打包的结果之前很可能被释放掉）。**P** 格式打包一个指向一个结构的指针，该结构的大小由 **count** 指明。如果对应的 **p** 或 **P** 的值是 **undef**，则创建一个空指针。

/ 字符允许对这样一个字符串进行打包或解包：这个打了包的结构包含一个字节数以及后面跟着字符串本身。你可以这样写 **length-item/string-item**。**length-item** 可以是任意 **pack** 模板字符，并且描

述长度值是如何打包的。最常用的可能是那些整数打包的东西，比如 **n**（用于打包 **Java** 字串），**w**（用于打包 **ASN.1** 或 **SNMP**）以及 **N**（用于 **Sun XDR**）。**string-item** 目前必须是 **A***，**a***，或者 **Z***。对于 **unpack** 而言，字串的长度是从 **length-item** 里获取的，但是如果你放 ***** 进去，那么它将被忽略。

```
unpack 'C/a', "\04Gurusamy"; # 生成 'Guru'
uppack 'a3/A* A*', '077 Bond J'; # 生成 ('Bond', 'J')
pack 'n/a* w/a*', 'hell', ',world'; # 生成 "hello, world"
```

length-item 不会从 **unpack** 明确地返回。向 **length-item** 字母加一个 **count** 不一定能干有用的事，除非该字母是 **A**，**a**，或 **Z**。用带 **length-item** 的 **a** 或 **Z** 进行打包可能会引入空（\0）字符，这时候，**Perl** 在会认为它不是合法数字字串。

整数格式 **s**，**S**，**l**，和 **L** 的后面可以紧跟一个 **!**，表示本机的短整数或者长整数，而不是各自准确的 **16** 位和 **32** 位。现在，这是一个在许多 **64** 位平台上的问题，在这些平台上本地 **C** 编译器看到本机短整数和长整数可能和上面的这些值不同。（**i!** 和 **I!** 也可以用，不过只不过是保持了完整；它们与 **i** 和 **I** 相同。）

你可以通过 **Config** 模块获取制作你的 **Perl** 的平台上的本机的 **short**，**int**，**long** 和 **long long** 的实际长度：

```
use Config; print $Config{shortsize}, "\n"; print $Config{intsize}, "\n"; print $Config{longsize}, "\n"; print $Config{longlongsize}, "\n";
```

这里只是说 **Configure** 知道一个 **long long** 的大小，但着并不意味着你就能用 **q** 和 **Q**。（有些系统能有，不过你用的系统很可能还没有。）

长度大于一个字节的整数格式（**s**，**S**，**i**，**I**，**l**，和 **L**）都是天生不能在不同处理器之间移植的，因为它们要遵从本机字节序和位权重序的规则。如果你需要可移植的整数，那么使用格式 **n**，**N**，**v**，和 **V**；因为它们是字节权重序并且是尺寸已知的。

浮点数只以本机格式存在。因为浮点格式的千差万别而且缺乏一种标准的“网络”表现形式，所以没有做任何浮点数交换的工具。这意味着在一台机器上打包了的浮点数数据可能不能在另外一台上读。甚至如果两台机器都使用 **IEEE** 浮点数算法的话，这都仍然是一个问题，因为与权重相关的内存表现形式不是 **IEEE** 规范的一部分。

Perl 在内部使用双精度数进行所有浮点数计算，所以从 **double** 转换成 **float**，然后又转换成 **float** 会损失精度。这就意味着 **unpack("f", pack("f", \$foo))** 可能不等于 **\$foo**。

你有责任为其他程序考虑任何对齐或填充的问题，尤其是那些带着 **C** 编译器自己的异质概念的 **C struct** 的程序，**C** 编译器在不同的体系结构上对 **C struct** 如何布局有着巨大的差别。你可能需要在打包时增加足够的 **x** 来弥补这个问题。比如，一个 **C** 声明：

```
struct foo { unsigned char c; float f; };
```

可以写成一个“**C x f**”格式，一个“**C x3 f**”格式，或者甚至是“**f C**”格式——而且这只是其中一部分。**pack** 和 **unpack** 函数把它们输入和输出当作平面的字节序列处理，因为它们不知道这些字节要去哪儿，或者从哪儿来。

让我们看一些例子，下面的第一部分把数字值包装成字节：

```
$out = pack "CCCC", 65, 66, 67, 68; # $out 等于"ABCD"
$out = pack "C4", 65, 66, 67, 68; # 一样的东西
```

下面的对 **Unicode** 的循环字母做同样的事情：

```
$foo = pack("U4", 0x24b6, 0x24b7, 0x24b8, 0x24b9);
```

下面的做类似的事情，增加了一些空：

```
$out = pack "CCxxCC", 65, 66, 67, 68; # $out 等于 "AB\0\0CD"
```

打包你的短整数并不意味着你就可移植了：

```
$out = pack "s2", 1, 2; # 在小头在前的机器上是 "\1\0\2\0" # 在大头在前的机器上是 "\0\1\0\2"
```

在二进制和十六进制包装上，**count** 指的是位或者半字节的数量，而不是生成的字节数量：

```
$out = pack "B32", "... (略)"; $out = pack "H8", "5065726c"; # 都生成 “Perl”
```

a 域里的长度只应用于一个字串：

```
$out = pack "a4", "abcd", "x", "y", "z"; # "abcd"
```

要绕过这个限制，使用多倍声明：

```
$out = pack "aaaa", "abcd", "x", "y", "z"; # "axyz" $out = pack "a" x 4, "abcd", "x",  
"y", "z"; # "axyz"
```

a 格式做空填充：

```
$out = pack "a14", "abcdefg"; # " abcdefg\0\0\0\0\0"
```

这个模板打包一个 **C** 的 **struct tm** 记录（至少在一些系统上如此）：

```
$out = pack "i9pl", gmtime(), $tz, $toff;
```

通常，同样的模板也可以在 **unpack** 函数里使用，尽管一些模板的动作不太一样，特别是 **a**，**A**，和 **Z**。

如果你想把定长文本域连接到一起，可以用 **TEMPLATE** 是多个 **a** 或者 **A** 的 **pack**：

```
$string = pack("A10" x 10, @data);
```

如果你想用一个分隔符连接变长文本域，那么可以用 **join** 函数：

```
$string = join(" and ", @data); $string = join ("", @data); # 空分隔符
```

尽管我们所有的例子都使用文本字符串做模板，但我们没有理由不让你使用来自磁盘文件的模板。你可以基于这个函数做一个完整的关系型数据库。（我们不会想知道那样能证明你什么东西。）

package package NAMESPACE package

它并不是一个真正的函数，只是一个声明，表示剩余的最内层的闭合范围属于它指明的符号表或者名字空间。（因此 **package** 声明的范围和 **my** 或 **our** 声明的范围是一样的。）在它的范围里，该声明令编译器把所有唯一的全局标识符都解析为到这个声明的包的符号表中寻找。

一个 **package** 声明只影响全局变量——包括那些你用了 **local** 的变量——而不包括你用 **my** 创建的词法变量。它只影响没有修饰的全局变量；那些带有自身包名字修饰的全局变量忽略当前所声明的包。用 **our** 声明的全局变量是没有修饰的，因此它们要受当前包影响，但是只是在声明的那个点才受影响，然后它的行为就好像 **my** 变量一样。也就是说，对于它们的词法范围的剩余部分，**our** 变量是在声明的地方“钉进”正在使用中的包中去的，甚至随后的包声明干涉进来也如此。

通常，你会把 `package` 声明作为一个文件里的要被 `require` 或者 `use` 操作符包括的第一个东西，但是你可以把 `package` 放在任何语句也可以合法放置的地方。当我们创建一个传统的或面向对象的模块文件的时候，我们习惯上把包的名字命名成和文件的名字相同，以此来避免混淆。（习惯上还喜欢把这样的包名字命名成以大写字母开头，因为小写字母模块通常会被解释成用法模块。）

你可以在多于一个地方切换到一个给定的模块中；它只是影响编译器对该块的剩余部分使用哪个符号表的决策。（如果编译器在同一个层次看到另外一个 `package` 声明，那么新的声明覆盖前面的那个。）Perl 假设你的主程序以一个不可见的 `package main` 声明开始。

你可以通过用包名字和双引号修饰对应标识符的方法来引用其它包的变量，子过程，句柄，和格式等：`$Package::Variable`。如果包名字是空，那么就假设是主包（`main package`）。也就是说，`$::sail` 等效于 `$main::sail`，也等效于 `$main'sail`，在那些老一些的代码里还会看到这样的标识符。

下面是一个例子：

```
package main; $sail = "hale and hearty"; package Mizzen; $sail = "tattered"; package
Whatever; print "My main sail is $main::sail.\n"; print "My mizzen sail is
$Mizzen::sail.\n";
```

它打印出：

```
My main sail is hale and hearty. My mizzen sail is tattered.
```

一个包的符号表是存储在一个散列里的，它的名字是以双冒号结尾的。比如，主包的符号表叫 `%main::`。因此现存的包符号 `*main::sail` 也可以通过 `$main::{"sail"}` 来访问。

如果省略了 `NAMESPACE`，那么就没有当前包存在，因此所有标识符必须是全称或者声明为词法范围。它比 `use strict` 更严格，因为它的范围还扩展到函数名字。

参阅第十章，包，获取有关包的更多信息。参阅本章早些的 `my` 获取其他范围相关的问题的信息。

pipe pipe READHANDLE, WRITEHANDLE

和对应的系统调用类似，这个函数打开一对相互联接的管道——参阅 `pipe(2)`。通常这个函数在 `fork` 之前调用，然后管道的读端关闭 `WRITEHANDLE`，而写端关闭 `READHANDLE`。（否则这个管道在写端关闭它以后不会告诉读端 `EOF`。）如果你设立了一个管道化的进程的循环，那么除非你非常细心，否则很有可能发生死锁。另外，请注意 Perl 的管道使用标准的 `I/O` 缓冲，所以你可能需要你的 `WRITEHANDLE` 上设置 `$|`（`$OUTPUT_AUTOFLUSH`）以保证在每次输出操作之后冲刷数据，具体情况取决于应用——参阅 `select`（输出文件句柄）。

（和 `open` 一样，如果两个文件句柄都没有定义，那么它们将被自动激活。）

下面是一个小例子：

```
pipe(README, WRITEME); unless ($pid = fork) { # 子进程 defined $pid or die "can't
fork: $!"; close(README); for $i (1..5) { print WRITEME "line $i\n" } exit; } $SIG
{CHLD} = sub { waitpid($pid, 0) }; close(WRITEME); @strings = ; close(README);
print "Got:\n", @strings;
```

请注意这里写入端是如何关闭读端以及读取端是如何关闭写入端的。你不能使用一条管道进行双向交流。要么使用两个不同的管道，要么使用 `socketpair` 系统调用。参阅第六章里的“管道”一节。

pop pop ARRAY pop

这个函数把一个数组当作一个堆栈对待——它弹出（删除）并返回数组的最后一个值，把数组缩短一个元素长度。如果省略了 **ARRAY**，那么该函数在子过程或者格式的词法范围里弹出 **@_**；在文件范围（通常是主程序），或者用 **eval STRING, BEGIN{}**，**CHECK{}**，**INIT{}**，和 **END{}** 构造建立的词法范围里，它弹出 **@ARGV**。它和下面的东西有同样的效果：

```
$tmp = $ARRAY[$#ARRAY--];
```

或者：

```
$tmp = splice @ARRAY, -1;
```

如果在数组里没有元素，那么 **pop** 返回 **undef**。（不过，如果你的数组包含 **undef** 数值，那么可不要依赖这个特性猜测数组是否为空！）又见 **poshi** 和 **shift**。如果你想弹出超过一个元素，那么使用 **splice**。

pop 要求它的第一个参数是一个数组，而不是一个列表。如果你只想一个列表的最后一个元素，那么用：

```
( LIST )[-1]
```

pos pos SCALAR pos

这个函数返回上一次 **m//g** 对 **SCALAR** 搜索在 **SCALAR** 中留下来的位置。它返回在最后一个匹配字符后面的字符的偏移量。（也就是说，它等效于 **length(\$`) + length(\$&)**。）这个位置是下一次在该字串上 **m//g** 开始的偏移量。请注意字串开头的偏移量是 **0**。比如：

```
$graffito = "fee fie foe foo"; while ($graffito =~ m/e/g) { print pos $graffito, "\n"; }
```

打印 **2**，**3**，**7**，和 **11**，就是每个跟在“e”后面的字符的偏移量。**pos** 函数可以赋予一个数值以告诉下一次 **m//g** 开始的位置：

```
$graffito = "fee fie foe foo"; pos $graffito = 4; # 忽略 fee，开始 fie while ($graffito =~ m/e/g) { print pos $graffito, "\n"; }
```

这样只打印出 **7** 和 **11**。正则表达式断言 **\G** 只匹配当前 **pos** 为被搜索字串声明的位置。参阅第五章的“位置”一节。

print print FILEHANDLE LIST print LIST print

这个函数打印一个字串或者一系列用逗号分隔的字串。如果设置了

\$\ (\$OUTPUT_RECORD_SEPARATOR) 变量，那么它将在列表结尾隐含地打印出来。如果成功，该函数返回真，否则返回假。**FILEHANDLE** 可以是一个标量变量的名字（未代换的），这个时候该变量要么包含实际文件句柄的名字或者包含一个指向某种文件句柄对象的引用。和任何其他间接对象一样，**FILEHANDLE** 也可以是一个返回这种数值的块：

```
print { $OK ? "STDOUT" : "STDERR" } "stuff\n"; print { $iohandle[$i] } "stuff\n";
```

如果 **FILEHANDLE** 是一个变量而下一个记号是一个项，那么它可能就会错误地解释成一个操作符，除非你在它们中间放一个 **+** 或者在参数周围放上圆括弧。比如：

```
print $a - 2; # 向缺省文件句柄（通常是 STDOUT）打印 $a - 2 print $a (-2); # 向在 $a 中声明的文件句柄中打印 -2 print $a -2; # 也打印 -2（古怪的分析规则 :-）
```

如果省略了 **FILEHANDLE**，那么该函数打印到当前选择的输出文件句柄，初始时是 **STDOUT**。要把缺省的输出文件句柄设置为 **STDOUT** 之外的东西，请使用 **select FILEHANDLE** 操作。（注：从此以后，**STDOUT** 就不再是 **print** 的缺省输出文件句柄了。它只是缺省的缺省文件句柄。）如果还省

略了 **LIST**，那么该函数打印 **\$_**。因为 **print** 接受 **LIST**，所以在 **LIST** 里的任何东西都在列表环境中计算。因此，当你说：

```
print OUT ;
```

它不会从标准输入中打印出下一行，而是来自标准输入的所有剩余的行知道文件结尾（**end-of-fiel**），因为那些东西是 在列表环境中返回的东西。如果你需要另外的结果，请说：

```
print OUT scalar ;
```

同样，请记住“如果它看起来象函数，那它就是函数”的规则，要注意不要在 **print** 关键字后面跟一个左圆括弧，除非你想用对应的右圆括弧结束 **print** 的参数——在中间插入一个 **+** 或者在所有参数周围放上圆括弧：

```
print (1+2)*3, "\n"; # 错 print +(1+2)*3, "\n"; # 对 print ((1+2)*3, "\n"); # 对
```

```
printf printf FILEHANDLE FORMAT, LIST printf FORMAT, LIST
```

这个函数向 **FILEHANDLE** 输出一条格式化的字串或者如果省略的时候，则是向当前选定的输出文件句柄打印，输出文件句柄初始时是 **STDOUT**。在 **LIST** 里的第一个项必须是一个字串，该字串声明如何格式化其他的项。它和 C 库的 **printf(3)** 和 **fprintf(3)** 函数类似。该函数等效于：

```
print FILEHANDLE sprintf FORMAT, LIST
```

只不过上面的句子中没有附加 **\$\ (\$OUTPUT_RECORD_SEPARATOR)**。如果 **use locale** 起作用，那么用于表示格式化的浮点数的小数点的字符将受到 **LC_NUMERIC** 的区域设置的影响。

只有当一个非法的引用类型用做 **FILEHANDLE** 参数的时候才会抛出一个例外。不识别的格式都丝毫未动地传递出去。如果打开了警告，那么两种情况都会触发警告。

参阅本章其他地方的 **print** 和 **sprintf** 函数。**sprintf** 的描述中包括格式声明列表的描述。我们本可以在这里复制一份，不过这本书已经是生态杀手了。

如果你省略了 **FORMAT** 和 **LIST**，则使用 **\$_**——但是在那些情况里，你应该使用 **print**。如果简单的 **print** 就可以用，不要使用 **printf**。**print** 函数更加高效并且也少一些出错机会。

```
prototype prototype FUNCTION
```

把一个函数的原形当作字串返回（如果该函数没有原型返回，**undef**）。**FUNCTION** 是一个指向你知道原型的函数的引用或者名字。

如果 **FUNCTION** 是一个以 **CORE::** 开头的字串，那么其他就会当作一个 Perl 内建函数的名字，如果没有这样的内建函数，那么就抛出一个例外。如果这个内建函数不是可覆盖的（象 **qw//**）或者是它的参数不能用一个原型表达（比如 **system**），那么此函数就返回 **undef**，因为该内建函数的行为并不象 Perl 函数那样。否则，返回描述等效原型的字串。

```
push push ARRAY, LIST
```

这个函数把 **ARRAY** 当作一个堆栈并且把 **LIST** 的值推进 **ARRAY** 尾部。**ARRAY** 的长度增加 **LIST** 的长度。该函数返回新长度。**push** 函数和下面的代码有一样的效果：

```
foreach $value (listfunc()) { $array[++$#array] = $value; }
```

或者：

```
splice @array, @array, 0, listfunc();
```

但是这个函数更高效些（不管对你还是你的计算机）。你可以拿 **push** 和 **shift** 一起使用，制作一个时间上相当高效的移动寄存器或者队列：

```
for(;;) { push @array, shift @array; }
```

又见 **pop** 和 **unshift**。

q/STRING/ q/STRING/ qq/STRING/ qr/STRING/ qw/STRING/ qx/STRING/

通用引号。参阅第二章里的“选择自己的引号”有关 **qx//** 的状态注解，参阅 **readpipe**。有关 **qr//** 的状态注解，参阅 **m//**。又见第五章“内部控制”。

quotemeta quotemeta EXPR quotemeta

这个函数返回 **EXPR** 的值，并且把所有非字母数字的字符都前缀反斜杠。（也就是说，字串里所有不匹配 **/[A-Za-z_0-9]/** 的字符都会在前面前缀反斜杠，不管区域设置是什么。）它是在代换环境中内部实现 **\Q** 逃逸的函数（包括双引号的字串，反勾号，以及模式。）

rand rand EXPR rand

这个函数返回一个伪随机的浮点数数字，该数字大于等于 **0** 而小于 **EXPR** 的值。（**EXPR** 应该是正。）如果省略了 **EXPR**，该函数返回一个在 **0** 和 **1** 之间的浮点数（包括 **0**，但是不包括 **1**）。除非已经调用了 **srand**，否则 **rand** 自动调用 **srand**。又见 **srand**。

要获取一个整数值，比如说给一个退出的角色赋值，你可以把它和 **int** 组合在一起，比如：

```
$roll = int(rand 6) + 1; # $roll 现在是一个介于 1 和 6 之间的数字
```

因为 Perl 使用你自己的 C 库的伪随机数函数，比如说 **random(3)** 或者 **drand48(3)**等，所以我们不能保证生成的数值的分布质量。如果你需要更强壮的随机，比如说用于加密用途，你可能应该参考一下 **random(4)** 的文档（如果你的系统有 **/dev/random** 或者 **/dev/urandom** 设备），CPAN 模块 **Math::TrulyRandom**，或者一本关于伪随机数的计算生成的好教科书，比如 Knuth 的第二卷。

read read FILEHANDLE, SCALAR, LENGTH, OFFSET read FILEHANDLE, SCALAR, LENGTH

这个函数试图从声明的 **FILEHANDLE** 中读取 **LENGTH** 字节的数据到 **SCALAR** 变量中。该函数返回读取的字节的数量，到文件结尾时返回 **0**。它在出错的时候返回 **undef**。**SCALAR** 将增长或者缩短到实际读取的长度。如果声明了 **OFFSET**，则决定该变量从那里开始输出字节，这样你就可以在一个字串中间读取数据。

要从文件句柄 **FROM** 中拷贝数据到文件句柄 **TO**，你可以说：

```
while(read(FROM, $buf, 16384)) { print TO $buf; }
```

read 的反操作只是一个简单的 **print**，它已经知道你想写的字串长度并且可以写任意长度的字串。不要误用 **write**，因为它只用语 **format**。

Perl 的 **read** 函数是用标准 I/O 的 **fread(3)** 函数实现的，所以实际的 **read(2)** 系统调用可能会读取超过 **LENGTH** 字节的数据填进缓冲区，而且 **fread(3)** 可能会做多次 **read(2)** 系统调用以填充缓冲区。要获得更好的控制，用 **sysread** 声明真正的系统调用。除非你想自找麻烦，否则请不要混合 **read** 和 **sysread**。不管你使用哪个，都要注意如果你从一个包含 **Unicode** 或者任何其他多字节编码的文件里读取，那么缓冲区的边界可能落在字符的中间。

readdir readdir DIRHANDLE

这个函数从一个用 `opendir` 打开的目录句柄读取目录记录（它们就是文件名）。在标量环境中，这个函数返回下一个目录记录（如果存在的话）；否则，它返回 `undef`。在列表环境中，它返回在该目录中所有剩下的记录，如果剩下没有记录了，那么这个返回可能是一个空列表。比如：

```
opendir(THISDIR, ".") or die "serious dainbramage: $!"; @allfiles = readdir THISDIR;
closedir THISDIR; print "@allfiles\n";
```

上面的代码在一行里打印出当前目录的所有文件。如果你想避免“.”和“..”记录，使用下面的咒语中的一条（你认为最不好念的那条）：

```
@allfiles = grep { $_ ne '.' and $_ ne '..' } readdir THISDIR; @allfiles = grep { not /^
[.][.]?z/ } readdir THISDIR; @allfiles = grep { not /^\. {1,2}z/ } readdir THISDIR;
@allfiles = grep !/^\.?z/, readdir, THISDIR;
```

为了避免所有 `.*` 文件（象 `ls` 程序）：

```
@allfiles = grep !/^\. /, readdir THISDIR;
```

只拿出文本文件，说：

```
@textfiles = grep -T, readdir THISDIR;
```

不过我们再看看最后一个例子，因为如果 `readdir` 的结果不在当前目录里，那么我们需要在它的结果上把目录部分粘回去——象这样：

```
opendir(THATDIR, $path) or die "can't opendir $path: $!"; @dotfile = grep { /^\. / && -
f } map { "$path/$_" } readdir(THATDIR); closedir THATDIR;
```

`readline` `readline` `FILEHANDLE`

这个函数是在内部实现操作符的函数，但是你可以直接使用它。该函数从 `FILEHANDLE` 中读取下一条记录，`FILEHANDLE` 可以是一个文件句柄名称或者一个间接的文件句柄表达式，该表达式要么返回实际文件句柄的名字要么返回一个指向任何类似文件句柄的东西的引用（比如一个类型团）。

（早于 5.6 版本的 Perl 只接受一个类型团。）在标量环境里，每次调用读取并返回下一条记录，直到到达文件结尾，这个时候后继的调用将返回 `undef`。在列表环境中，`readline` 读取记录直到文件结束，然后返回一个记录列表。这里说的“记录”，我们通常的意思是一行文本，但是通过改变 `$/`（`$INPUT_RECORD_SEPARATOR`）的值可以让这个操作符用不同的方法给文本“分段”。还有，有些输入纪律，比如 `:para`（段落模式）将以块的方式返回记录而不是行。设置 `:slurp` 纪律（或者解除 `$/` 的定义）可以令这样的一块就是整个文件。

在标量环境中读取文件的时候，如果你碰巧读进了一个空文件，`readline` 第一次返回 `""`，然后在后继的每次调用都返回 `undef`。如果从一个 `ARGV` 文件句柄中读取数据，那么每个文件都返回一个块（同样，空文件返回 `""`），当全部读取完成的时候再读取就返回 `undef`。

操作符在第二章的“输入操作符”中有更详细的讨论。

```
$line = ; $line = readline(STDIN); # 和上面一样 $line = readline(*STDIN); # 和上面一样
$line = readline(\*STDIN); # 和上面一样
```

```
open my $fh, "<=&STDIN" or die; bless $fh => 'AnyOldClass'; $line = readline($fh); #
和上面一样
```

`readlink` `readlink` `EXPR` `readlink`

这个函数返回一个符号链接指向的文件名。`EXPR` 应该计算出一个文件名，而且它的最后一个组成部

分应该是一个符号链接。如果它不是符号链接，或者如果符号链接没有在文件系统里实现，或者如果发生某种系统错误，则返回 **undef**，并且你应该检查在 **\$!** 里的错误代码。

请注意这个返回的符号链接可能是相对于你声明的位置。比如，你可能说：

```
readlink "/usr/local/src/express/yourself.h"
```

而 **readlink** 可能会返回：

```
../express.1.23/includes/yourself.h
```

除非你的当前目录碰巧是 **/usr/local/src/express**，否则这个东西可能不能直接当文件名使用。

readpipe readpipe scalar EXPR readpipe LIST（建议中）

它是实现 **qx//** 引起构造（也称之为反勾号操作符）的内部函数。偶尔它和很有用，比如说你需要一种特殊的声明你的 **EXPR** 的方法，而这种方法如果使用引起的形式的话并不方便。请注意的是我们以后可能改变它的接口以支持 **LIST** 参数，这样它就更象 **exec** 函数，所以，不要假设它会继续为 **EXPR** 提供标量环境。你应该自己给他 **scalar**，或者试着使用 **LIST** 形式。天知道到你阅读到这些内容的时候是不是就可以用了。

recv recv SOCKET, SCALAR, LEN, FLAGS

这个函数接收套接字上的一条信息。它试图从声明的 **SOCKET** 文件句柄中接收 **LENGTH** 个字节的数据到变量 **SCALAR** 中。该函数返回发送者的地址，或者如果有错误的话是 **undef**。**SCALAR** 将根据所读取的数据增长或者缩短到数据长度。该函数接受与 **recv(2)** 相同的标志。参阅第十六章的“套接字”一节。

redo redo LABEL redo

redo 操作符在不经重新计算条件的情况下重新开始一个循环块。如果存在任何 **continue** 块也不会执行它们。如果省略了 **LABEL**，那么该操作符指向最内层的闭合循环。该操作符通常用于那些希望欺骗自己输入的程序：

一个把那些用反斜杠续起来的行接起来的循环

```
while() { if (s/\\n$// && defined($nextline =)) { $_ .= $nextline; redo; } print; # 或者别的什么... }
```

redo 不能用于退出一个有返回值的块，比如 **eval {}**，**sub {}**，或者 **do {}**，并且不能用于退出一个 **grep** 或者 **map** 操作。在打开了警告的时候，如果你 **redo** 一个不在你的当前词法范围的循环，那么 **Perl** 会警告你。

一个块本身等效于一个只执行一次的循环。因此在这样的块里的 **redo** 将很有效地把它变成一个循环构造。参阅第四章里的“循环控制”。

ref ref EXPR ref

如果 **EXPR** 是一个引用，那么 **ref** 操作符返回一个真值，否则返回一个假。返回的值取决于该引用所引用的东西的类型。内建的类型包括：

SCALAR ARRAY HASH CODE GLOB REF LVALUE IO::Handle

如果这个引用的对象已经赐福到了一个包中，那么返回的就是该包的名字。你可以把 **ref** 当作一种“类型是”操作符。

```
if (ref($r) eq "HASH") { print "r is a reference to a hash.\n"; } elsif (ref($r) eq "Hump")
```

```
{ # 烦人——见下文 print "r is a reference to a Hump object.\n" } elsif (not ref $r) { print "r is not a reference at all.\n"; }
```

我们认为测试你的对象的类名字是否等于任意特定的类名字是非常糟糕的 OO 风格，因为一个派生类会有不同的名字，但是应该允许访问基类的方法。这种情况最好用下面的 UNIVERSAL 方法：

```
if ($r->isa("Hump")) { print "r is a reference to a Hump object, or subclass.\n" }
```

最好是根本别测试，因为除非 OO 机制在一开始就觉得合适，否则它不会给你的方法发送对象。参阅第八章和第十二章获取更多细节。又见第三十一章里的 use attributes 用法里的 reftype 函数。

rename rename OLDNAME, NEWNAME

这个函数修改一个文件的名字。成功时返回真，否则返回假。它（通常）不能跨文件系统运行，不过在 Unix 系统上 mv 通常可以做这件事情。如果一个名叫 NEWNAME 的文件已经存在，那么它将被删除。非 Unix 系统上可能还有其他限制。

参阅标准的 File::Copy 模块获取跨文件系统的重命名。

require require VERSION require EXPR require

这个函数断言在其参数上的某种依赖性。

如果该参数是一个字串，require 装载并执行那些在放在独立文件内的 Perl 代码，该文件的名字是该字串给出的。这种情况类似在一个文件上做 do，只不过是 require 会检查一下，看看该库文件是否已经装载了，而且，如果它碰到任何问题都会抛出一个例外。（因此它可以用于表达文件依赖性而不用担心重复编译。）和它的表亲 do 和 use 类似，require 知道如何搜索保存在 @INC 数组里的包含文件并且在成功的时候更新 %INC。参阅第二十八章。

该文件必须返回真作为最后的值以表示任何初始化代码的成功执行，因此我们习惯上用 1 结束这种文件；除非你确信它返回真。

如果 require 的参数是一个版本号，比如 5.6.2 这样；require 实际上就是要求当前正在运行的 Perl 版本必须至少是那个版本。（Perl 还接受一个浮点数，比如 5.005_03，这样就和老版本的 Perl 兼容，但是现在我们不鼓励使用那种形式，因为来自其他文化的家伙们不理解它。）因此，一个需要 Perl 5.6 的脚本可以把下面这个做其第一行：

```
require 5.6.0; # 或者 require v5.6.0
```

这样早期的 Perl 版本就会退出。不过，和所有 require 一样，这些事情是在运行时发生的。你可能更愿意说 use 5.6.0 以获得编译时间的检查。又见第二十八章里的 \$PERL_VERSION。

如果 require 的参数是一个光包的名字（参阅 package），require 假设它有一个自动的 .pm 后缀，这样就令它容易装载标准的模块。这个特性类似 use，只不过是发生在运行时间而不是编译时间，并且没有调用 import 模块。比如，要在不往当前包增加任何符号的情况下把 Socket.pm 拉进来，你可以说：

```
require Socket; # 而不是 "use Socket;"
```

不过，你可以用下面的代码获得相同效果，而且如果找不到 Socket.pm 的话还可以得到编译时警告：

```
use Socket ();
```

在一个光名字上使用 require 还把任何包名字里的 :: 替换成你的系统的目录分隔符，通常是 /。换句话说，如果你试验下面的代码：

```
require Foo::Bar; # 很好的空名
```

那么 `require` 函数在 `@INC` 数组里声明的目录里寻找 `Foo/Bar.pm` 文件。但是如果你试验这些：

```
$class = 'Foo::Bar'; require $class; # $class 不是一个光名
```

或者：

```
require "Foo::Bar"; # 引号文本不是光名
```

那么 `require` 函数将在 `@INC` 数组里寻找 `Foo::Bar` 文件并且回抱怨没有在那里找到 `Foo::Bar`。如果这样，你可以用：

```
eval "require $class";
```

又见 `do FILE`，`use` 命令，`use lib` 用法，以及标准的 [FindBin[?]](#) 模块。

`reset reset EXPR reset`

这个函数通用于（或者滥用于）循环的顶端或者在循环尾部的一个 `continue` 块里，用于清理全局变量或者重置 `??` 搜索，这样它们就又可以运转了。表达式解释成一个单字符的列表（连字符可以用做表示范围）。所有以这些字符之一开头的标量变量，数组，以及散列都恢复到它们最早的状态。如果省略了该表达式，那么匹配一次的搜索（`?PATTERN?`）被重置，重新进行匹配。该函数只为当前包重置变量或搜索。它总是返回真。

要重置所有“X”变量，你可以说：

```
reset 'X';
```

要重置所有小写变量，说：

```
reset 'a-z';
```

最后，只重置 `??` 搜索，说：

```
reset;
```

我们不推荐你在 `main` 包里重置“`A-Z`”，因为你会把你的所有 `ARGV`，`INC`，`ENV`，和 `SIG` 数组和散列都摧毁。

词法变量（由 `my` 创建）不受影响。`reset` 的使用已经模糊地废弃了，因为它很容易清空整个名字空间，而且 `??` 操作符本身也是模糊地废弃了。

又见来自标准 `Symbol` 的 `delete_package()` 函数，以及在第二十三章的“安全隔仓”里记录的所有安全隔仓的内容。

`return return EXPR return`

这个操作符令当前子过程（或者是 `eval` 或 `do FILE`）马上带着声明的数值返回。试图在这些地方之外使用 `return` 将抛出一个例外。还要注意 `eval` 不能代表调用它的子过程做 `return`。

`EXPR` 可能会在列表，标量，或者空环境中计算，具体是哪种环境由如何使用该返回值决定，而这个情况可能每次执行都会不一样。也就是说，你提供的表达式将在子过程调用的环境中计算。如果子过程是在标量环境中调用的，那么 `EXPR` 也在标量环境中计算。如果该子过程是在列表环境中调用的，那么 `EXPR` 也是在列表环境中计算并且返回一个数值列表。没有参数的 `return` 在标量环境中返回标量值 `undef`，在列表环境中返回一个空列表 `()`，而（自然）在空环境中什么也不返回。子过程调用的环境可以在子过程里用（错误命名的）`wantarray` 函数来判断。

reverse reverse LIST

在列表环境里，这个函数返回一个数值列表，该列表包含反序排列的 **LIST** 的元素。该函数可以用于创建递减序列：

```
for (reverse 1 .. 10) { ... }
```

因为如果散列作为 **LIST** 传递，那么它就会平面化为一个列表，所以 **reverse** 还可以用于反转一个散列，假设其值是唯一的：

```
%barfoo = reverse %foobar;
```

在标量环境里，该函数连接 **LIST** 的所有元素，然后返回这个连接出来的字串的一个字符一个字符的反序，

一条小提示：反转一个由用户定义函数排序的列表的时候，可能更容易通过先以相反方向对该列表排序来实现。

rewinddir rewinddir DIRHANDLE

这个函数为操作 **DIRHANDLE** 的 **readdir** 过程把当前位置设置到目录的开头。该函数可能无法在所有支持 **readdir** 的系统上使用，如果系统没有实现 **rewinddir**，那么它会退出。它成功的时候返回真，否则返回假。

rindex rindex STR, SUBSTR, POSITION rindex STR, SUBSTR

这个函数运行起来和 **index** 很相象，只不过它返回在 **STR** 里最后发生的 **SUBSTR** 的位置（反向的 **index**）。如果没有找到 **SUBSTR**，那么该函数返回 **\$[-1]**。因为 **\$[** 现在实际上总是 **0**，而该函数实际上总是返回 **-1**。如果声明了 **POSITION**，那么它就是可以返回的最右端的位置。要想从后向前扫描一遍你的字串，说：

```
$pos = length $string; while (($pos = rindex $string, $lookfor, $pos) >= 0) { print
"Found at $pos\n"; $pos--; }
```

rmdir rmdir FILENAME rmdir

如果 **FILENAME** 声明的目录是空的，那么此函数删除该目录。如果该函数成功，它返回真；否则，返回假。如果你想先删除该目录的内容，而又因为什么原因不想调用 **shell** 里的 **rm -r**，（比如说没有 **shell**，或者没有 **rm** 命令，因为你没办法获得 **PPT**。）那么可以看看 **File::Path** 模块。

```
s/// s///
```

替换操作符。参阅第五章里的“模式匹配操作符”。

scalar scalar EXPR

这个伪函数可以用于 **LIST** 里，当在列表环境中计算会生成一个不同的结果的时候，强迫 **EXPR** 在标量环境中计算。比如：

```
my ($nextvar) = scalar ;
```

避免 在做赋值之前从标准输入把所有的行都读了进来，因为给一个列表（甚至是一个 **my** 列表）赋值都会产生一个列表环境。（在这里例子里如果没有 **scalar**，那么来自 的第一行仍然会赋予 **\$nextvar**，但是随后的行将会被读取并抛弃，因为我们赋值的目标列表只能接受一个标量数值。）

当然，简单些而且没有那么混乱的方法是不使用圆括弧，这样就把标量环境改变成了列表环境：

```
my $nextvar = ;
```

因为 `print` 函数是一个 `LIST` 操作符，所以如果你想把 `@ARRAY` 的长度打印出来，那么你不得不说：

```
print "Length is ", scalar(@ARRAY), "\n";
```

Perl 里没有与 `scalar` 对应的 “list” 函数，因为实际上我们从来不需要强迫在列表环境里计算。这是因为任何需要 `LIST` 的操作已经给他的列表参数免费提供了一个列表环境。

因为 `scalar` 是单目操作符，如果你不小心给 `EXPR` 使用了圆括弧的列表，那么这个东西的行为就象一个标量逗号表达式一样，在空环境中计算除最后一个列表元素之外的所有其他元素，然后返回在标量环境中计算的最后一个元素。你很少想要这样的东西。下面的一个语句：

```
print uc(scalar(&foo, $bar)), $baz;
```

在道义上是等效于下面两个的：

```
&foo; print(uc($bar), $baz);
```

参阅第二章获取关于逗号操作符的更多细节。参阅第六章的“原型”获取关于单目操作符更多的信息。

`seek seek FILEHANDLE, OFFSET, WHENCE`

这个函数为 `FILEHANDLE` 定位文件指针，就好象用于标准 I/O 的 `fseek(3)` 调用一样。文件里的第一个位置是在偏移量 `0` 处，而不是 `1` 处。同样，偏移量指的是字节位置，而不是行数。通常，因为行的长度是变化的，所以我们不可能不检查到该点之间的所有文件内容就能访问某一行，除非你的所有行数都已知是特定的长度，或者你已经做了一个把行数转换成字节数的索引。（同样的限制也适用于有着变长字符编码的字符位置：操作系统不知道什么是字符，它们只知道字节。）

`FILEHANDLE` 可以是一个表达式，其值给出实际的文件句柄的名字或者是一个指向任何类似文件句柄对象的引用。该函数成功时返回真，失败时返回假。为了方便，该函数可以从各种文件位置计算偏移量。`WHENCE` 的值声明你的 `OFFSET` 使用文件的哪个偏移量做它的开始位置：`0`，文件开头；`1` 文件的当前位置；`2`，文件结尾。如果 `WHENCE` 的值是 `1` 或 `2`。那么 `OFFSET` 可以为负值。如果你喜欢用 `WHENCE` 的符号值，你可以用来自 `IO::Seekable` 或者 `POSIX` 模块的 `SEEK_SET`，`SEEK_CUR`，和 `SEEK_END`，或者在 Perl 5.6 里的 `Fcntl` 模块。

如果你想为 `sysread` 或者 `syswrite` 定位文件，那么不要使用 `seek`；标准 I/O 缓冲技术会令 `seek` 对文件在系统位置上的作用变得不可预料而且也不能移植。应该用 `sysseek`。

因为 `ANSI C` 的规则和严格，在一些系统上，如果你在在读取和写出之间做切换，那么你必须做一次搜寻。这样做的效果就好象调用标准 I/O 库的 `clearerr(3)` 函数。你可以用 `WHENCE` 为 `1`（`SEEK_CUR`）和 `OFFSET` 为 `0` 实现这个目的而又不会移动文件位置：

```
seek(TEST, 0, 1);
```

这个函数的一个有趣的用途是允许你跟随文件的增长，比如：

```
for (;;) { while () { grok($_); # 处理当前行 } sleep 15; seek LOG, 0, 1; # 重置 end-of-file 错误。 }
```

最后一个 `seek` 在不移动指针的情况下清理文件结束错误。因你的 `C` 库的标准 I/O 实现的标准程度的不同而异，你可能需要一些更象下面这样的东西：

```
for (;;) { for ($curpos = tell FILE; ; $curpos = tell FILE) { grok($_); # 处理当前行 }
```

```
sleep $for_a_while; seek FILE, $curpos, 0; # 重置 end-of-file 错误。 }
```

类似的策略可以用于在一个数组里记住 `seek` 每一行的地址。

seekdir seekdir DIRHANDLE, POS

这个函数为下一次 `readdir` 对 `DIRHANDLE` 的调用设置当前位置。POS 必须是 `telldir` 返回的值。这个函数与对应的系统库过程在可能的目录压缩问题上有相同的注意事项。该函数可能不是在所有实现了 `readdir` 的系统上都实现了。而且如果 `readdir` 没有实现，它也肯定没实现。

select (输出文件句柄) select FILEHANDLE select

由于历史原因，Perl 里有完全互不相关的两个 `select` 操作符。参阅下一节获取另外一个的描述。这个版本的 `select` 操作符返回当前选定的输出操作符，并且，如果你提供了 `FILEHANDLE`，那么把它设置为当前缺省输出操作符。这样做有两个效果：首先，一个没有文件句柄的 `write` 或者 `print` 将缺省输出到这个 `FILEHANDLE`。其次，与输出相关的特殊变量将指向这个文件句柄。比如，如果你想为多个输出文件句柄设置了相同的页顶格式，那么你可能需要做这些：

```
select REPORT1; $^ = 'MyTop'; select REPROT2; $^ = 'MyTop';
```

但请注意这样就把 `REPORT2` 当作当前选定的文件句柄了。这种做法可以说是反社会的做法，因为它可能会真的把一些其他过程的 `print` 或者 `write` 语句搞坏。写的好的库过程会在退出的时候把当前选定文件句柄设置为和进入过程时相同的那个。为了支持这个，`FILEHANLDE` 可以是一个表达式，该表达式的值给出实际文件句柄的名字。因此，你可以用下面的代码保存并恢复当前选项的文件句柄：

```
my $oldfh = select STDERR; $| = 1; select $oldfh;
```

或者使用惯用的但有点模糊的方法：

```
select((select(STDERR), $| = 1)[0])
```

这个例子是这样运转的：制作一个由 `select(STDERR)`（副作用是选定了 `STDERR`）的返回值和 `$|=1`（它总是 1）组成的列表，但同时，又作为副作用设置了现在选定的 `STDERR` 的自动冲刷。该列表的第一个元素（前面那个选定的文件句柄）现在用做外层 `select` 的一个参数。古怪吧？这些足够让你知道 `List` 比较危险了。

你还可以使用标准的 [SelectSaver²](#) 模块在退出范围的时候自动恢复前面一个 `select`。

不过，虽然我们给你解释了上面的全部东西，我们还是可以指出在当今的情况下，你很少需要使用这种形式的 `select`，因为你象设置的大多数特殊变量都有面向对象的封装方法帮你做这些事情。所以，你不用直接设置 `$|`，而是：

```
use IO::Handle; # 糟糕的是，这可不是个*小*模块。 STDOUT->autoflush(1);
```

而前面的格式化例子可以这样编码：

```
use IO::Handle; REPORT1->format_top_name("MyTop"); REPORT2->format_top_name("MyTop");
```

select (准备文件描述符) select RBITS, WBITS, EBITS, TIMEOUT

四个参数的 `select` 和前面那个 `select` 完全无关。这个操作符用于发现你的文件描述符中那个（如果有的话）已经准备好做输入或者输出了，或者报告一个例外条件。（这样就避免你做轮询。）它用你声明的位掩码调用 `select(2)` 系统调用，你可以用 `fileno` 和 `vec` 构造这个位掩码，象这样：

```
$rin = $win = $ein = ""; vec($rin, fileno(STDIN), 1) = 1; vec($win, fileno(STDIN), 1) = 1; $ein = $rin | $win;
```

如果你想在许多文件句柄上 `select`，你可能会写这样的子过程：

```
sub fhbits { my @fhlist = @_; my $bits; for (@fhlist) { vec($bits, fileno($_), 1) = 1; }
return $bits; } $rin = fhbits(qw(STDIN TTY MYSOCK));
```

如果你想重复使用同样的位掩码（这样做更高效），常用的惯用法是：

```
($nfound, $timeleft) = select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

或者阻塞住直到任意文件描述符准备好：

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

如你所见，在标量环境中调用 `select` 只返回 `$nfound`，找到的准备好的文件描述符数量。

可以用 `$wout=$win` 技巧是因为一个赋值语句的数值是它自身的左边，因此 `$wout` 先被赋值删除，然后又被 `select` 删除，而 `$win` 不会变化。

这些参数任何一个都可以是 `undef`，这个时候它们被忽略。如果 `TIMEOUT` 不是 `undef`，那么就是以秒计，而且可以是分数。（超时时间为 0 的效果就是轮询。）不是所有实现都能返回 `$timeleft`。如果不能返回 `$timeleft`，那么它总是返回等于你提供的 `$timeout` 的 `$timeleft`。

标准的 `IO::Select` 模块提供了 `select` 的更加友善的接口，主要是因为它为你做了所有你能用的位掩码。

`select` 的一个应用就是实现比 `sleep` 分辨率更好的睡眠。要实现这个目的，声明所有位映射为 `undef`。因此，如果要睡眠（至少）4.75 秒钟，用：

```
select undef, undef, undef, 4.75;
```

（在一些非 Unix 系统上，三个 `undef` 的形式可能不能用，你可能需要为一个有效的描述符至少伪装一个位掩码，而那个描述符可以是从来准备不好的。）

我们不应该把缓冲的 I/O（比如 `read` 或 `write`）和 `select` 混在一起用，除了 POSIX 允许的以外，而且就算 POSIX 允许，也只能在真正的 POSIX 系统上使用。这时应该用 `sysread`。

`semctl semctl ID, SEMNUM, CMD, ARG`

这个函数调用 System V IPC 函数 `semctl(2)`。你可能得先说 `use IPC::SysV` 以获取正确的常量定义。如果 `CMD` 是 `IPC_STAT` 或者 `GETALL`，那么 `ARG` 必须是一个它可以保存返回的 `semid_ds` 结构或信号灯数值数组的变量。和 `ioctl` 和 `fcntl` 一样，返回值用 `undef` 代表错误，“0 but true”代表零，其他情况则返回实际数值。

又见 `IPC::Semaphore` 模块。这个函数只有在那些支持 System V IPC 的机器上才能用。

`semget semget KEY, NSEMS, SIZE, FLAGS`

这个函数调用 System V IPC 系统调用 `semget(2)`。在调用之前，你应该 `use IPC::SysV` 以获取正确的常量定义。该函数返回信号灯 ID，或者如果有错误返回 `undef`。

又见 `IPC::Semaphore` 模块。这个函数只能在那些支持 System V IPC 的机器上用。

`semop semop KEY, OPSTRING`

这个函数调用 System V IPC 系统调用 `semop(2)` 以执行信号灯操作，比如发信号和等待等等。在调用之前，你应该使用 `use IPC::SysV` 以获取正确的常量定义。

OPSTRING 必须是一个 **semop** 结构的打包的数组。你可以通过说 `pack("s*", $semnum, $semop, $semflag)` 做每一个 **semop** 结构。信号灯操作的数量是由 **OPSTRING** 的长度隐含的。该函数在成功的时候返回真，在失败的时候返回假。

下面的代码在等待信号灯 **id** 为 **\$semid** 的信号灯 **\$semnum**：

```
$semop = pack "s*", $semnum, -1, 0; semop $semid, $semop or die "Semaphore
trouble: $!\n";
```

要给信号灯发出信号，只需要把 **-1** 换成 **1** 就可以了。

参阅第十六章的“**System V IPC**”一节。又见 **IPC::Semaphore** 模块。这个函数只有支持 **System V IPC** 的机器上可以用。

send send SOCKET, MSG, FLAGS, TO send SOCKET, MSG, FLAGS

这个函数在套接字上发送一条信息。它和同名系统调用接收相同的标志——参阅 **send(2)**。在未联接的套接字上，你必须声明一个要发送的目的 **TO**，这样就会令 Perl 的 **send** 象 **sendto(2)** 那样运行。C 的系统调用 **sendmsg(2)** 目前没有在标准的 Perl 里实现。**send** 函数在成功时返回发送的字节数，失败时返回 **undef**。

（有些非 **Unix** 系统错误地把套接字当作与普通文件描述符不同的东西对待，结果就是你必须总是在套接字上 **send** 和 **recv**，而不能使用方便的标准 **I/O** 操作符。）

我们中至少有一个人会常犯的错误就是把 Perl 的 **send** 和 C 的 **send** 和写混淆起来：

```
send SOCK, $buffer, length $buffer # 错
```

这行代码会莫名其妙地失败，具体情况取决于字符串长度和系统需要的 **FLAG** 位之间的关系。参阅第十六章中的“消息传递”一节。

setpgrp setpgrp PID, PGRP

这个函数为指定的 **PID**（对当前进程使用 **PID** 等于 **0**）设置当前进程组（**PGRP**）。如果在那些没有实现 **setpgrp(2)** 的系统上调用 **setpgrp** 将会抛出一个例外。注意：有些系统上会总是忽略你提供的参数并总是做 **setpgrp(0, \$\$)**。幸运的是，这些就是我们最常用的参数。如果省略了参数，它们缺省是 **0, 0**。**BSD 4.2** 版本的 **setpgrp** 不接受任何参数，但在 **BSD 4.4** 里，它是 **setpgid** 函数的同义词。如果需要更好的移植性（从某种角度来看），直接使用 **POSIX** 模块里的 **setpgid** 函数。如果你实际上想干的事是把你的脚本作成守护进程，那么请考虑使用 **POSIX::setsid()** 函数。请注意 **POSIX** 版本的 **getpgrp** 并不接受参数，所以只有 **setpgrp(0, 0)** 是真正可以移植的。

setpriority setpriority WHICH, WHO, PRIORITY

这个函数为 **WHICH** 和 **WHO** 里声明的一个进程，进程组，或者一个用户设置当前 **PRIORITY**，参阅 **setpriority(2)**。在那些没有实现 **setpriority(2)** 的机器上调用 **setpriority** 将抛出一个例外。要把你的程序“**nice**”下四个单位（和用 **nice(1)** 处理你的程序一样），用：

```
setpriority 0, 0, getpriority(0, 0) + 4;
```

一个给定的优先级的解释可能会因不同的系统而异。有些权限可能是那些非特权用户所不能使用的。

又见 **CPAN** 的 **BSD::Resource** 模块。

setsockopt setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL

这个函数设置你需要的套接字选项。出错时该函数返回 **undef**。**LEVEL** 表示你的调用瞄准的是哪一

个协议层。或者就是 `SOL_SOCKET`，指向在所有层之上的套接字本身。如果你不想传递参数，那么可以把 `OPTVAL` 声明为 `undef`。在套接字上一个常用的选项是 `SO_REUSEADDR`，这样才能绕开因为前一个在该端口的 `TCP` 联接仍然认为固执地认为它在关闭的时候，我们不能绑定特定的地址的问题。它看起来象这样：

```
use Socket; socket(SOCK, ...) or die "Can't make socket: $!\n"; setsocket(SOCK,
SOL_SOCKET, SO_REUSEADDR, 1) or warn "Can't do setdosockopt: $!\n";
```

参阅 `setsockopt(2)` 获取其他可能数值。

shift shift ARRAY shift

这个函数把数组的第一个值移出并且返回它，然后把数组长度减一并且把所有的东西都顺移。如果在数组中不再存在元素，它返回 `undef`。

如果省略了 `ARRAY`，那么该函数在子过程和格式的词法范围里移动 `@_`；它在文件范围（通常是主程序）或者在由 `eval STRING`，`BEGIN { }`，`CHECK { }`，`INIT { }`，和 `END { }` 这样的构造里面的词法范围里移动 `@ARGV`。

子过程通常以拷贝它们的参数到词法变量里开始，而 `shift` 可以用于这个目的：

```
sub marine { my $fathoms = shift; # 深度 my $fishies = shift; # 鱼的数量 my $o2 =
shift; # 氧气问题 # ... }
```

`shift` 还可以用于在你的程序前面处理参数：

...（略）

你还可以考虑使用 `Getopt::Std` 和 `Getopt::Long` 模块来处理程序参数。

又见 `unshift`，`push`，`pop`，和 `splice`。`shift` 和 `unshift` 函数在数组左边做的事情和 `pop` 和 `push` 在数组右边干的事情是一样的。

shmctl shmctl ID, CMD, ARG

这个函数调用 `System V IPC` 系统调用 `shmctl(2)`。在调用之前，你应该 `use IPC::SysV` 以获取正确的常量定义。

如果 `CMD` 是 `IPC_STAT`，那么 `ARG` 必须是一个将要保存返回的 `shmid_ds` 结构的变量。跟 `ioctl` 和 `fcntl` 一样，该函数错误时返回 `undef`，“0 but true”表示零，其他情况下返回实际返回值。

该函数只能在那些支持 `System V IPC` 的机器上用。

shmget shmget KEY, SIZE, FLAGS

这个函数调用 `System V IPC` 系统调用 `shmget(2)`。该函数返回共享内存段的 `ID`，如果有错误则返回 `undef`。在调用之前，先 `use SysV2::IPC`。

该函数只能在那些支持 `System V IPC` 的机器上用。

shmread shmread ID, VAR, POS, SIZE

这个函数从共享内存段 `ID` 的位置 `POS` 处开始读取 `SIZE` 大小的数据（方法是附着在该内存段上，拷贝出数据，然后与该内存段分离。）。`VAR` 必须是一个将保存读取出的数据的变量。如果成功，该函数返回真，如果失败返回假。

该函数只能在那些支持 **System V IPC** 的机器上用。

shmwrite shmwrite ID, STRING, POS, SIZE

这个函数向共享内存段 **ID** 的位置 **POS** 处开始写入 **SIZE** 大小的数据（方法是附着在该内存段上，拷贝入数据，然后与该内存段分离。）。如果 **STRING** 太长，那么只写入 **SIZE** 字节；如果 **STRING** 太短，那么在后面补空直到 **SIZE** 字节。如果成功，该函数返回真，如果有错误，返回假。

该函数只能在那些支持 **System V IPC** 的机器上用。（你可能都读烦了——我们已经写烦了。）

shutdown shutdown SOCKET, HOW

这个函数以 **HOW** 声明的方式关闭一个套接字联接。如果 **HOW** 为 **0**，那么不再允许进一步的接收。如果 **HOW** 为 **1**，那么不再允许进一步的发送。如果 **HOW** 为 **2**，那么任何事情都不允许。

```
shutdown(SOCK, 0); # 不许再读 shutdown(SOCK, 1); # 不许再写 shutdown(SOCK, 2); #
不许再 I/O
```

如果你想告诉对端你完成数据写出了，但还没有完成数据读取，或者反过来，在这些情况下它都非常有用。而且它还是一种更执着的关闭方式，因为同时还关闭任何这些文件描述符在派生出的进程中的的拷贝。

让我们想象有一个服务器想读取它的客户端的请求，直到文件结尾，然后发送一个回答。如果客户端调用 **close**，那么该套接字现在将不能用于 **I/O**，因此不会有回答能送回来。因此，客户端应该使用 **shutdown** 以半关闭这次联接：

```
print SERVER "my request\n"; # 发送一些数据 shutdown(SERVER, 1); # 发送完毕，没有更
多要发的东西了 $answer = ; # 但你还可以读
```

（如果你找到这里是为了找到关闭你的系统的办法，那么你就要执行一个外部的程序干这件事。参阅 **system**。）

sin sin EXPR sin

抱歉，这个操作符什么罪都没犯（译注：英文“**sin**”也有“罪恶”的含义）。它只是返回 **EXPR**（用弧度表示）的正弦。

如果需要正弦的逆操作，你可以使用 **Math::Trig** 或者 **POSIX** 模块的 **asin** 函数，或者用下面的关系：

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep sleep EXPR sleep

这个函数令脚本睡眠 **EXPR** 秒，如果没有 **EXPR** 则是永久睡眠，并且返回睡眠的秒数。你可以通过给该进程发送一个 **SIGALRM** 的方法来中断睡眠。在一些老式系统里，它可能比你要求的描述整整少睡一秒，具体情况取决于它是如何计算秒的。大多数现代的系统都是睡足秒数。不过，在这些系统上它们很有可能睡眠的时间要长一些，因为在一台繁忙的多任务系统上，你的系统可能无法马上得到调度。如果可能，**select**（等待文件描述符）调用可以给你更好的分辨率。你还可以用 **syscall** 调用一些 **Unix** 系统支持的 **getitimer(2)** 和 **setitimer(2)** 过程。你不应该混合 **alarm** 和 **sleep** 调用，因为 **sleep** 通常是用 **alarm** 实现的。

又见 **POSIXE** 模块的 **sigpause** 函数。

socket socket SOCKET, DOMAIN, TYPE, PROTOCOL

这个函数打开一个指定类型的套接字，并且把它附着在 **SOCKET** 文件句柄上。**DOMAIN**，**TYPE**，和 **PROTOCOL** 都是和 **socket(2)** 一样的声明。如果没有定义 **SOCKET**，那么 Perl 将自动激活它。在使用这个函数之前，你的程序应该包含下面这行：

```
use Socket;
```

它给你正确的常量。该函数成功时返回真，参阅在第十六章里的“套接字”节里的例子。

在那些支持对文件的 **exec** 时关闭 (**close-on-exec**) 的系统上，该标记将为新打开的文件描述符设置，就象 **\$^F** 判定的那样。参阅第二十八章里的 **\$^F (\$SYSTEM_FD_MAX)**。

socketpair socketpair SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL

这个函数在声明的域中创建一个指定类型的匿名套接字对。**DOMAIN**，**TYPE**，和 **PROTOCOL** 都和 **socketpair(2)** 里声明的一样。如果两个套接字参数都没有声明，那么它们自动激活。该函数成功时返回真，失败时返回假。在那些没有实现 **socketpair(2)** 的系统上，调用这个函数会抛出一个例外。

这个函数的通常用法是在 **fork** 之前使用。生成的进程中有一个关闭 **SOCKET1**，而另外一个关闭 **SOCKET2**。你可以双向使用这些套接字，而不象 **pipe** 函数创建的文件句柄那样是单向的。有些系统用 **socketpair** 的方式定义 **pipe**，这时候调用 **pipe(Rdr, Wtr)** 相当于：

```
use Socket; socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC); shutdown
(Rdr, 1); # 不允许读者写 shutdown(Wtr, 0); # 不允许写者读
```

在那些支持对文件的 **exec** 时关闭 (**close-on-exec**) 的系统上，该标记将为新打开的文件描述符设置，就象 **\$^F** 判定的那样。参阅第二十八章里的 **\$^F (\$SYSTEM_FD_MAX)**。又见在第十六章里的“双向通讯”一节尾部的例子。

sort sort USERSUB LIST sort BLOCK LIST sort LIST

这个函数对 **LIST** 进行排序并返回排好序的列表值。缺省时，它以标准字符串比较顺序排序（未定义数值排在已定义空字符串前面，而空字符串又在其他任何东西前面）。如果 **use locale** 用法起作用，那么 **sort LIST** 根据当前的区域集的数值对 **LIST** 排序。

如果给出了 **USERSUB**，那么它就是一个返回小于，等于，或者大于 0 的整数的子过程名字，具体返回什么取决于列表中的元素应该如何排序。（很便利的 **<=>** 和 **cmp** 操作符可以用于执行三向数字和字符串比较。）如果给出了 **USERSUB**，但该函数未定义，那么 **sort** 抛出一个例外。

为了提高效率，绕开了通常用于子过程的调用代码，这样就有了下面的结果：这个子过程不能是递归子过程（你也不能用一个循环控制操作符退出该块或者过程），并且将要接受比较的两个元素不是通过 **@_** 传递进子过程的，而是通过临时设置 **sort** 编译所在的包的全局变量 **\$a** 和 **\$b**（参阅后面的例子）。变量 **\$a** 和 **\$b** 是真实值的别名，所以不要在子过程中修改它们。

子过程要求的动作是比较。如果它返回的结果是不一致的（比如，有时候说 **\$x[1]** 小于 **\$x[2]**，而有时候说的正相反），那么结果就不会良好。（这也是你不能修改 **\$a** 和 **\$b** 的另外一个原因。）

USERSUB 可以是标量变量名字（未代换），这时，它的值要么是引用实际子过程的符号引用，要么是硬引用。（符号名更好些，即使用了 **use strict 'refs'** 用法也如此。）在 **USERSUB** 的位置，你可以提供一个 **BLOCK** 用做一个匿名内联排序子过程。

要做一次普通的数字排序，你可以说：

```
sub numerically { $a <=> $b } @sortedbynumber = sort numerically 53, 29,11,32, 7;
```

要以降序排序，你可以简单地在 **sort** 后面应用 **reverse**，或者你可以在排序过程里把 **\$a** 和 **\$b** 反

过来:

... (略)

要对字符串进行大小写不敏感的排序, 在比较之前用 `lc` 处理 `$a` 和 `$b`:

... (略)

(在 `Unicode` 里, 用 `lc` 做大小写规范化要比用 `uc` 好, 因为有些语言里抬头体和大写是不一样的。不过它对普通的 `ASCII` 排序没有什么影响, 并且如果你想让 `Unicode` 能正确排序, 那么你的规范化过程可能要比 `lc` 更别致一些。)

对散列按照数值排序是 `sort` 函数的常用法之一。比如, 如果 `%sales_amount` 散列记录部门销售情况, 那么在排序过程里做一次散列查找就可以让我们将散列键字根据它们的数值排序:

从销售额最高的部门到最低的部门

... (略)

你可以通过使用 `||` 或者 `or` 操作符级连多个比较的方法进行额外层次的排序。这种方法相当漂亮, 因为比较操作符通常在相等的时候返回 `0`, 这样就令它们能落到下一个比较。下面, 散列键字首先根据它们相关的销售额排序, 然后在根据键字本身进行排序 (以处理有两个或多个部门销售额相同的情况):

```
sub by_sales_then_dept { $sales_amount{$b} <=> $sales_amount{$a}
<
bgcolor="#FFFFCC">
$ a cmp $ b }
```

```
for $dept (sort by_sales_then_dept keys %sale_amount) { print "$dept =>
$sales_smount{$dept}\n"; }
```

假设 `@recs` 是一个散列引用的数组, 而这里每个散列包含象 `FIRSTNAME`, `LASTNAME`, `AGE`, `HEIGHT`, 和 `SALARY` 这样的域。下面的过程把那些记录中的人们按照下面的顺序排列: 先是财富, 然后是身高, 然后是年龄 (越小越靠前), 最后是名字的字母顺序:

... (略)

任何可以从 `$a` 和 `$b` 中得到的有用信息都可以在一个排序过程中比较的基础来用。比如, 如果多行文本要根据特定域来排序, 那么可以在排序过程中使用 `split` 以获取该域:

```
@sorted_lines = sort { @a_fields = split /:/, $a; # 冒号分隔的域 @b_fields = split /:/,
$b;
```

```
$a_fields[3] <=> $b_fields[3] # 对第四个域进行能够数字排序, 然后
<
bgcolor="#FFFFCC">
$a_fields[0] cmp $b_fields[0] # 对第一个域进行能够字符串排序, 然后
<
bgcolor="#FFFFCC">
$b_fields[2] <=> $a_fields[2] # 对第而个域进行行能够数字反向排序 ... # 等等 } @lines;
```

不过, 因为 `sort` 使用给 `$a` 和 `$b` 的不同的数值对多次运行排序过程, 所以前面的例子将会比对每一行都做多余的重新分裂。

为了避免发生象为了比较数据域导致的多次的行分裂带来的开销，我们可以在排序之前对每个值进行一次操作，然后把生成的信息保存起来。下面，我们创建了一个匿名数组以捕获每一行以及该行的分裂结果：

```
@temp = map { [$_, split /\:/] } @lines;
```

然后，我们对数组引用排序：

```
@temp = sort { @a_fields = @$_[1..$#$_]; @b_fields = @$_[1..$#$_];  
  
$a_fields[3] <=> $b_fields[3] # 对第四个域进行能够数字排序，然后  
<  
bgcolor="#FFFFCC">  
$a_fields[0] cmp $b_fields[0] # 对第一个域进行能够字符串排序，然后  
<  
bgcolor="#FFFFCC">  
$b_fields[2] <=> $a_fields[2] # 对第二个域进行能够数字反向排序 ... # 等等 } @temp;
```

在这个数组引用排完序之后，我们就可以从这个匿名数组里检索原始行了：

```
@sorted_lines = map { $_->[0] } @temp;
```

概而括之，这个 `map-sort-map` 技巧，就是我们通常称之为 **Schwartzian** 变换的东西，可以用一个语句实现：

```
@sorted_lines = map { $_->[0] } sort { @a_fields = @$_[1..$#$_]; @b_fields = @$_  
[1..$#$_];  
  
$a_fields[3] <=> $b_fields[3]  
<  
bgcolor="#FFFFCC">  
$a_fields[0] <=> $b_fields[0]  
<  
bgcolor="#FFFFCC">  
$b_fields[2] <=> $a_fields[2] ... } map { [$_, split /\:/] } @lines;
```

不要把 `$a` 和 `$b` 定义成词法变量（用 `my`）。它们都是包全局变量（如果它们可以免于 `use strict` 对普通全局变量的限制）。不过你的确需要保证你的排序过程是在同一个包里的，或者用调用者的包名字修饰 `$a` 和 `$b`。

我们已经说过，在 **Perl 5.6** 里你可以用标准的参数传递方法（以及不一样的是，用 **XS** 子过程做排序子过程）写排序子过程，前提是你用一个 `($$)` 的原型声明了这个排序子过程。并且如果你是这么用的，那么实际上你是可以把 `$a` 和 `$b` 声明为词法变量的：

```
sub numerically ($$) { my ($a, $b) = @_; $a <=> $b; }
```

将来，当完整的原型都实现了以后，你就可以只用说：

```
sub numerically ($a, $b) { $a <=> $b }
```

然后我们或多或少就能回到开始的地方。

```
splice splice ARRAY, OFFSET, LENGTH, LIST splice ARRAY, OFFSET, LENGTH splice  
ARRAY, OFFSET splice ARRAY
```

这个函数从一个 **ARRAY** 中删除 **OFFSET** 和 **LENGTH** 指明的元素，并且，如果给出了 **LIST**，则用 **LIST** 的元素替换它。如果 **OFFSET** 是负数，那么该函数从数组的后面数，但如果该值会伸到数组开头的前面，那么就会抛出一个例外。在列表环境中，**splice** 返回从该数组中删除的元素。在标量环境中，它返回最后删除的元素，而如果没有的话返回 **undef**。如果新元素的数量不等于旧元素的数量，那么该数组根据需要伸缩，并且元素的位置根据衔接后的情况进行改变。如果省略了 **LENGTH**，那么该函数从数组里删除从 **OFFSET** 开始的所有东西。如果省略了 **OFFSET**，那么该数组在读取的时候清空。下面的等式成立（假设 **\$[** 为 0）：

直接方法 | **splice** 等效

...（略）

splice 函数还可以方便地用于切开传递给子过程的参数列表。比如，假设列表长度在列表之前传递：

```
sub list_eq { # 比较两个列表值 my @a = splice(@_, 0, shift); my @b = splice(@_, 0,
shift); return 0 unless @a == @b; # 长度相同? while(@a) { return 0 if pop(@a) ne pop
(@b); } return 1; } if (list_eq($len, @foo[1..$len], scalar(@bar), @bar)) { ... }
```

不过，拿数组引用来干这事更清晰一些。

split **split** /**PATTERN**/, **EXPR**, **LIMIT** **split** /**PATTERN**/, **EXPR** **split** /**PATTERN**/ **split**

这个函数扫描字符串中 **EXPR** 给出的分隔符，并且把该字符串劈成一个子字符串列表，在列表环境中返回生成的列表值，或者在标量环境中返回子字符串的数量。（注：标量环境同时还令 **split** 把它的结果写到 **@_**，不过这个用法现在废弃了。）分隔符是用重复的模式匹配进行判断的，用的是 **PATTERN** 里给出的正则表达式，因此分隔符可以是任意大小，并且不一定在每次匹配都是一样的字符串。（分隔符不象平常那样返回；我们在本节稍后讨论例外情况。）如果 **PATTERN** 完全不能匹配该字符串，那么 **split** 把原始字符串当作子字符串返回。如果它匹配了一次，那么你就得到两个子字符串，以此类推。你可以在 **PATTERN** 里使用正则表达式修饰词，比如 **/PATTERN/i**，**/PATTERN/x**，等等。如果你以模式 **/^/** 进行分裂，那么就假设是 **//m** 修饰词。

如果声明了 **LIMIT** 并且是正的，该函数分裂成不超过那么多的域（当然如果它用光了分隔符，那么是可以分裂成比较少的子字符串的）。如果 **LIMIT** 是负数，那就把它当作声明了任意大的 **LIMIT**。如果省略了 **LIMIT** 或者是零，那么将从结果中删除结尾的空域（那些潜在的 **pop** 用户应该好好记住）。如果省略了 **EXPR**，那么该函数就分裂 **\$_** 字符串。如果还省略了 **PATTERN** 或者它是一个文本空格，**" "**，那么该函数对空格进行操作，**/s+/**，但是忽略任何开头的空格。

可以分裂任意长度的字符串：

```
@chars = split //, $word; @fields = split /:/, $line; @words = split " ", $paragraph;
@lines = split /^/, $buffer;
```

一个可以匹配空串或者其他的一些比空串长的字符串的模式（比如，一个由任意一个字符加上 ***** 或者 **?** 修饰的模式）将把 **EXPR** 的值分裂成独立的字符，只要它匹配字符之间的空串；非空匹配会象通常的情况那样忽略匹配过的分隔符字符。（换句话说，一个模式不会在一个点匹配多过一次，即使它和一个零宽匹配也如此。）比如：

```
print join ':', split / */, 'hi there';
```

生成输出 **"h:i:t:h:e:r:e"**。空白消失的是因为它作为分隔符一部分匹配。举一个小例子，空模式 **//** 简单地分裂成独立的字符，而空格并不消失。（对于正常模式匹配而言，**//** 模式会在上一次成功匹配处重复，但是 **split** 的模式免受此过。）

LIMIT 参数只分裂字符串的一部分：


```
($login, $passwd, $remainder) = split /:/, $_, 3;
```

我们鼓励你把你的字符串分裂成这样的列表名字，这样你的代码就有了自文档的特性。（可以用于出错检查，请注意如果字符串比三个域少，那么 `$remainder` 将会是未定义。）当给一个列表赋值的时候，如果省略了 `LIMIT`，那么 Perl 提供一个 `LIMIT`，其数值比列表中的变量数量大一，以此避免不必要的工作。对于上面的分裂，`LIMIT` 缺省是 4，而 `$remainder` 将只收到第三个域，而不是所有剩下的域。在时间要求很严格的应用里，避免分裂成比我们需要的更多的域是一个好习惯。（强大的语言的问题就是，它给你强大的功能的是以花费在时间上的愚蠢为代价的。）

我们早先说过分隔符不会被返回，但是如果 `PATTERN` 包含圆括弧，那么每一对圆括弧匹配的子字符串都会包括在结果列表中，分散在那些平常返回的域之中。下面是一个简单的例子：

```
split /([-,)]/, "1-10,20";
```

生成列表：

```
(1, '-', 10, ',', 20)
```

如果有更多圆括弧，那么为每个圆括弧对返回一个域，即使有些圆括弧对没有匹配也如此，这种情况下，为那些位置返回未定义数值。因此，如果你说：

```
split /(-)|(|,)/, "1-10,20";
```

那么结果是：

```
(1, '-', undef, 10, undef, 20);
```

`/PATTERN` 参数的位置可以放这么一个表达式，该声明在运行时生成不同的模式。和普通模式一样，如果想只做一次运行时编译，那么用 `/ $variable /o`。

有一个特殊的情况，如果该表达式是一个空格（“ ”），那么该函数会象没有参数的 `split` 那样在空格上把字符串分裂开。因此 `split(" ")` 可以用于模拟 `awk` 的缺省行为。相反，`split(/ /)` 将给你和前导空格一样多的空的初始域。（除了这个特殊的例子以外，如果你提供的是一个字符串而不是一个正则表达式，那么它还是会被解释成一个正则表达式。）你可以用这个属性把开头和结尾的空白删除，并且把中间的空白都压缩成一个空白：

```
$string = join(' ', split(' ', $string));
```

下面的例子把一个 RFC 822 消息头分裂成一个包含 `$head{Date}`，`$head{Subject}`，等等的散列。它使用了给一个散列赋予一个配对列表的技巧，理由是域和分隔符交错。它利用圆括弧把每个分隔符的一部分当作返回列表值的一部分返回。因为 `split` 模式保证把返回的东西利用包含圆括弧的好处按照配对的形式返回，所以散列赋值就可以保证收到一个包含键字/数值对的列表，这里每个键字就是一个头域的名字。（糟糕的是，这个技巧会丢失有着相同域的多个行的信息，比如 `Received-By` 行。啊，哦...）

```
$header =~ s/\n\s+/ /g; # 融合连续行
%head = ('FRONTSTUFF', split /^(\S*?):\s*/m, $header);
```

下面的例子处理整个 `Unix passwd(5)` 文件。你可以忽略 `chomp`，这个时候 `$shell` 的结尾将有换行符。

```
open PASSWD, '/etc/passwd'; while () { chomp; # 删除结尾的换行符
($login, $passwd, $uid, $gid, $gcos, $home, $shell) = split /:/; ... }
```

下面是一个如何处理每个输入文件里的每一行中的每个词，创建一个单词频率散列的例子：

```
while (<>) { foreach $word (split) { $count{$word}++; } }
```

`split` 的逆操作由 `join` 执行（只不过 `join` 只能在所有域之间用同样的分隔符连接）。要用固定位置的域分解字串，请使用 `unpack`。

`sprintf` `sprintf` `FORMAT`, `LIST`

这个函数返回一个格式化字串，格式化习惯是 C 的库函数 `sprintf` 的是 `printf` 习惯。参阅你的系统的 `sprintf(3)` 或 `printf(3)` 获取一些通用原则的解释。`FORMAT` 包含一个带有嵌入的域指示符的文本，`LIST` 里的元素就是逐一替换到这些域中去的。

Perl 做自己的 `sprintf` 格式化——它模拟 C 函数 `sprintf`，但是它没有用 C 的 `sprintf`。（注：除了浮点数以外，并且就算是浮点数也只允许标准的修饰词。）结果是，任何你本地的 `sprintf(3)` 函数的扩展都不能在 Perl 里使用。

Perl 的 `sprintf` 允许全局使用的已知转化在 表29-4 中列出。

表29-4。 `sprintf` 的格式

域 | 含义

<code>%%</code>	一个百分号	<code>%c</code>	一个带有给定数字的字符	<code>%s</code>	一个字串	<code>%d</code>	一个有符号整数，十进制
<code>%u</code>	一个无符号整数，十进制	<code>%o</code>	一个无符号整数，八进制	<code>%x</code>	一个无符号整数，十六进制		
<code>%e</code>	一个浮点数，科学记数法表示	<code>%f</code>	一个浮点数，用固定的小数点表示	<code>%g</code>	一个浮点数，以 <code>%e</code> 或 <code>%f</code> 表示		

另外，Perl 允许下列广泛支持的转换：

域 | 含义

<code>%x</code>	类似 <code>%x</code> ，但使用大写字符	<code>%E</code>	类似 <code>%e</code> ，但使用大写的“E”	<code>%G</code>	类似 <code>%g</code> ，但是带一个大写的“E”（如果正确的话）
<code>%b</code>	一个无符号整数，二进制	<code>%p</code>	一个指针（输出十六进制的 Perl 值的地址）	<code>%n</code>	特殊：把到目前为止输出的字符数放到 参数列表中的下一个变量里

最后，为了向下兼容（我们的意思就是“向下”），Perl 允许下列不必要的但广泛支持的转换：

域 | 含义

<code>%i</code>	<code>%d</code> 的同义词	<code>%D</code>	<code>%ld</code> 的同义词	<code>%U</code>	<code>%lu</code> 的同义词	<code>%O</code>	<code>%lo</code> 的同义词	<code>%F</code>	<code>%f</code> 的同义词
-----------------	----------------------	-----------------	-----------------------	-----------------	-----------------------	-----------------	-----------------------	-----------------	----------------------

Perl 允许下列众所周知的标志出现在 `%` 和转换字符之间：

域 | 含义

<code>space</code>	用空格前缀正数	<code>+</code>	用加号前缀正数	<code>-</code>	在域内左对齐	<code>-</code>	用零而不是空格进行右对齐	<code>#</code>	给非零八进制前缀“0”，给非零十六进制前缀“0x”
<code>number</code>	最小域宽度	<code>.number</code>	“精度”：浮点数的小数点后面的位数	<code> </code>	字串最大长度。整数最小长度	<code>l</code>	把整数解释成 C 类型的 <code>long</code> 或者 <code>unsigned long</code>	<code>h</code>	把整数解释成 C 类型的 <code>short</code> 或者 <code>unsigned short</code> （如果没有 提供标志，那么把整数解释成 C 类型 <code>int</code> 或者 <code>unsigned</code>

还有两个 Perl 相关的标志

域 | 含义

V | 把整数解释成 Perl 标准的整数类型 **v** | 把字符串解释成一个整数向量，输出成用点 | 分隔的数字，或者是用任意参数列表里前面 | 带 * 的字符串分隔

如果你的 Perl 理解“四倍数”（64位整数），不管是该平台本机支持还是因为你指明 Perl 带着该功能编译，那么字符 `duoXBIDUO` 打印64位整数，并且它们前面可以选择前缀 `ll`，`L`，或则 `q`。比如，`%lld %16LX %qo`。

如果 Perl 理解“long double”（要求该平台支持 long double），那么你可以在 `efgEFG` 标志前面增加可选的 `ll` 或者 `L`。比如，`%llf %Lg`。

在标志里可以出现数字的位置，都可以用一个星号（“*”）代替，这时候 Perl 使用参数列表里的下一个项作为给出的数字（也就是说，当作域宽度或者精度）。如果通过“*”获取的域宽度是负数，那么它和“-”标志有一样的效果：左对齐。

v 标志可以用于显示任意字符串里的序数值：

```
sprintf "version is v%vd\n", $^V; # Perl 的版本
sprintf "address is %vd\n", %addr; # IPv4 地址
sprintf "address is %*vX\n", ":", $addr; # IPv6 地址
sprintf "bits are %*vb\n", " ", $bits; # 随机的位串
```

sqrt sqrt EXPR sqrt

这个函数返回 **EXPR** 的平方根。如果需要其他的根，比如立方根，你可以使用 ****** 操作符求那个数字的分数幂。不要试图在着两种方法里使用负数，因为它有一些稍微有些复杂的问题（并且抛出一个例外）。但是有一个模块可以处理这些事情：

```
use Main::Complex; print sqrt(-2); # 打印出 1.4142135623731i
```

srand srand EXPR srand

这个函数为 **rand** 操作符设置随机数种子。如果省略了 **EXPR**，那么它使用一个内核提供的半随机的数值（如果内核支持 `/dev/urandom` 设备）或者是一个基于当前时间和进程号以及一些其他东西的数值。通常我们完全没有必要调用 **srand**，因为如果你没有明确调用它，那么它也会在第一次调用 **rand** 操作符时隐含调用。不过，在早于 Perl 5.004 的版本里不是这样的，所以如果你的脚本需要在老 Perl 版本上运行，那么你就应该调用 **srand**。

那些经常被调用的程序（比如 CGI 脚本），如果只是简单地用 `time ^ $$` 做种子的话，那么很容易惨遭下面的数学性质的攻击，那就是：有三分之一的机会 `a^b == (a+1)^(b+1)`。所以不要这么干。应该用下面的代码：

```
srand( time() ^ ($$ + ($$ << 15)) );
```

如果用于加密目的，那么你需要用比缺省的种子生成更随机的算法。有些系统上有 `/dev/random` 设备就比较合适，否则，拿一个或多个会迅速改变操作系统状态的程序的输出，压缩以后进行校验和计算是常用的方法。比如：

```
srand (time ^ $$ ^ unpack "%32L*", `ps wwaxl | gzip`);
```

如果你特别关心这些问题，那么请参阅 CPAN 上的 `Math::TrulyRandom` 模块。

不要在你的程序里多次调用 **srand**，除非你知道你在干什么并且知道为什么这么做。这个函数的目的是给 **rand** 函数种子，这样 **rand** 就可以在你每次运行你的程序的时候生成不同的序列。只需要在你

的程序开始做一次，否则你就不能从 `rand` 中获得随机的数值！

stat stat FILEHANDLE stat EXPR stat

在标量环境里，这个函数返回一个布尔值，该值表示调用是否成功。在列表环境里，它返回一个 13 个元素的列表，给出一个文件的统计信息，该文件要么是通过 `FILEHANDLE` 打开，要么是用 `EXPR` 命名。它的典型应用如下：

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

不是在所有文件系统类型上都支持这些域；不支持的域返回 0。表 29-5 列出了各个域的含义。

表 29-5. `stat` 返回的域

索引 | 域 | 含义

0		<code>\$dev</code>		文件系统的设备号	1		<code>\$ino</code>		i 节点号码	2		<code>\$mode</code>		文件模式（类型和权限）	3		<code>\$nlink</code>		指向该文件的（硬）链接数量	4		<code>\$uid</code>		文件所有者的数字用户 ID	5		<code>\$gid</code>		文件所属组的数字组 ID	6		<code>\$rdev</code>		设备标识符（只用于特殊文件）	7		<code>\$size</code>		文件的总尺寸，以字节计	8		<code>\$atime</code>		自纪元以来以秒计的上一次访问时间	9		<code>\$mtime</code>		自纪元以来以秒计的上一次修改时间	10		<code>\$ctime</code>		自纪元以来以秒计的上一次i节点改变的时间（不是创建时间！）	11		<code>\$blksize</code>		选定的用于文件系统 I/O 的块尺寸	12		<code>\$blocks</code>		实际分配的块数量
---	--	--------------------	--	----------	---	--	--------------------	--	--------	---	--	---------------------	--	-------------	---	--	----------------------	--	---------------	---	--	--------------------	--	---------------	---	--	--------------------	--	--------------	---	--	---------------------	--	----------------	---	--	---------------------	--	-------------	---	--	----------------------	--	------------------	---	--	----------------------	--	------------------	----	--	----------------------	--	-------------------------------	----	--	------------------------	--	--------------------	----	--	-----------------------	--	----------

`$dev` 和 `$ino` 放在一起，在同一个系统里唯一地标识一个文件。`$blksize` 和 `$blocks` 很可能只在 BSD 衍生出的文件系统里有。如果有 `$block` 域，那么它是以 512 字节的块汇报的。

`$blocks*512` 的值可能和 `$size` 差距相当大，因为有些文件包含未分配的块，或者说“洞”，它们没有在 `$blocks` 中计算。

如果传递给 `stat` 一个特殊的文件句柄，该句柄里包含下划线，那么就不会做实际的 `stat(2)` 调用，而是返回上一次 `stat`，`lstat`，或者基于 `stat` 的文件测试操作符（比如 `-r`，`-w`，和 `-x`）的 `stat` 结构。

因为模式包含文件类型及其权限，所以如果你想看真正的权限，那么你应该屏蔽掉文件类型部分，并且在 `printf` 或者 `sprintf` 里用 “%o”：

```
$mode = (stat($filehane))[2]; printf "Permissions are %04o\n", $mode & 07777;
```

`File::stat` 模块提供一个方便的通过名字访问的机制：

```
use File::stat; $sb = stat($filename); printf "File is %s, size is %s, perm %04o, mtime %s\n", $filename, $sb->size, $sb->mode & 07777, scalar localtime $sb->mtime;
```

你还可以从 `Fcntl` 模块里输入各种不同模式位的符号定义。参阅联机文档获取更多细节。

提示：如果你只需要文件尺寸，那么可以用 `-s` 文件测试操作符，它直接返回以字节计的文件大小。另外还有返回以天计的文件年龄的文件测试。

study study SCALAR study

这个函数花了一些额外的时间用在研究 `SCALAR` 上，预测在进行下一次修改之前要做的匹配次数。这么做也许能但也许不能节约时间，具体情况取决于你正在搜索的模式的天性和数量，以及待搜索字符串中字符出现频率的分布——你可能需要比较有它和没它的运行时间来看看哪个运行得快一些。那些扫描许多常量字符串（包括在更复杂的模式里的常量部分）的循环会从 `study` 中得到最大好处。如果你的所有模式匹配都是前面有锚符号的常量字符串，那么 `study` 一点忙都帮不上，因为那里没有扫描。你

每次只能拥有一个 **study**——如果你研究另外一个标量的话，那么前面那个就“没有研究”了。

study 运转的方法是：做一个包含待搜索字符串中的每个字符的链表，因此，打个比方，我们就知道了所有的“k”字符的位置。从每个搜索字符串里选出最少见的字符，这是基于从一些 C 程序和英文文本构造出来的频率统计表的。只对那些包含这个最少见字符的位置进行检查。

比如，下面是一个循环，它在每个包含特定模式的行前面插入一个生成索引的记录：

```
while(<>) { study; print ".IX foo\n" if /\bfoo\b/; print ".IX bar\n" if /\bbar\b/; print ".IX
blurfl\n" if /\bblurfl\b/; ... print; }
```

为了搜索 `/\bfoo\b/`，只查看 `$_` 里的那些包含“f”的位置，因为“f”比“o”少见。除了在病态的情况下，这样做是很有优势的。唯一的问题是它是否能节约比先制作链表花的时间更多的时间。

如果你必须搜索那些你直到运行时才知道的字符串，那么你可以把整个循环作成一个字串然后 **eval** 它以避免每次都要重新编译你的模式。再通过设置 `$/` 把整个文件输入成一条记录，把这些组合起来可以非常快，通常比那些专业程序，象 **fgrep(1)** 什么的都快。下面的程序扫描一个文件列表（`@files`），搜索一个单词列表（`@words`），并且把那些包含大小写无关匹配的文件名字打印出来：

```
$search = 'while (<>) { study;'; foreach $word (@words) { $search .= "++\$seen
{\$ARGV} if /\b$word\b/i;\n"; } $search .= "}; @ARGV = @files; undef $/; # 吃进整个文件
eval $search; # 这里运行程序 die $_ if $_; # 处理 eval 失败 $/ = "\n"; # 恢复正常的输入终止符
foreach $file (sort keys(%seen)) { print "$file\n"; }
```

既然我们有 `qr//` 操作符，那么上面的编译完的运行时 **eval** 看上去没有必要。下面的做相同的事情：

```
@pats = (); foreach $word (@words) { push @pats, qr/\b${word}\b/i; } @ARGV =
@files; undef $/; # 吃进每个完整的文件 while (<>) { for $pat (@pats) { $seen{$ARGV}
++ if /$pat/; } } $/ = "\n"; # 恢复正常的输入终止符 foreach $file (sort keys(%seen))
{ print "$file\n"; }
```

sub

命名声明：

```
sub NAME PROTO ATTRS sub NAME ATTRS sub NAME PROTO sub NAME
```

命名定义：

```
sub NAME PROTO ATTRS BLOCK sub NAME ATTRS BLOCK sub NAME PROTO BLOCK sub
NAME BLOCK
```

未命名定义：

```
sub PROTO ATTRS BLOCK sub ATTRS BLOCK sub PROTO BLOCK sub BLOCK
```

子过程声明和定义的语法看起来挺复杂的，但是在实践上实际相当简单。所有东西都是基于下面语法的：

```
sub NAME PROTO ATTRS BLOCK
```

所有的四个域都是可选的；唯一的限制就是如果这些域的确存在的话那么它们必须以这些顺序出现，并且你必须至少使用 **NAME** 或者 **BLOCK** 之一。目前，我们会忽略 **PROTO** 和 **ATTRS**；它们只是基本语法的修饰词。**NAME** 和 **BLOCK** 都是保证正确重要部分：

* 如果你只有 **NAME** 而没有 **BLOCK**，它就是一个该名字的声明（并且如果你想调用该子过程，那么你就必须稍后用 **NAME** 和 **BLOCK** 提供一个定义。）命名的声明是非常有用的，因为如果编译器知道它是一个用户定义子过程，那么它会对该名字另眼相看。你可以把这样的子过程当作一个函数或者当作一个操作符来调用，就象内建函数一样。有时候我们把这样的东西叫做提前声明。

* 如果你同时提供了 **NAME** 和 **BLOCK**，那么它就是一个标准的命名子过程定义（如果你在 前面没有声明，那么它还是声明）。命名定义也很重要，因为 **BLOCK** 把一个实际的含义（子过程体）和声明关联起来。这就是我们所谓的定义和声明的区别。不过，定义和声明也有类似的地方，那就是子过程代码看不到它，并且它不返回你可以用之来引用子过程的 内联的值。

* 如果你只有 **BLOCK** 而没有 **NAME**，那么它就是一个匿名的定义，也就是一个匿名子过程。因为它没有名字，所以它根本就不是声明，而是一个真正的操作符，在运行时返回一个指向匿名子过程体的引用。这个东西在把代码当作数据对待的时候极为有用。它允许你传递一段奇怪的代码用做回调函数，并且甚至还可以当作闭合块用——如果该 **sub** 定义操作符提到了任何在其自身以外的词法变量的话。这就意味着对同样 **sub** 操作符的不同调用都将进行记录工作，以保证在闭合块的生命期里，每个这样的词法变量的正确的“版本”都是该闭合块可见的，甚至于该词法变量所属的最初的范围被摧毁了也如此。

在上面三种情况中的任何一种里，**PROTO** 和 **ATTRS** 之一或者全部都可以在 **NAME** 之后和/或 **BLOCK** 之前出现。原型是一个放在圆括弧里的字符列表，它告诉分析器如何对待该函数的参数。属性是用一个冒号引入的，它告诉分析器有关这个函数的额外的信息。下面是一个包含四个域的典型的定义：

```
sub numstrcmp ($$) : locked { my ($a, %b) = @_ ; return $a <=> $b || $a cmp %b; }
```

有关属性列表和它们的操作的细节，请参阅第三十一章里的 **attributes** 用法。又见第六章和第八章的“匿名子过程”。

**substr substr EXPR, OFFSET, LENGTH, REPLACEMENT substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET**

这个函数从 **EXPR** 给出的字串中抽取一个子字串然后返回它。这个子字串是从字串前面 **OFFSET** 个字符的位置开始抽取的。（注意：如果你曾经修改了 **\$[**，那么字串的开头就不再是 **0** 了，不过因为你没有修改过 **\$[**，所以它的开头还是 **0**。）如果 **OFFSET** 是负数，那么子字串是从字串后面数这么多偏移量位置开始的。如果省略了 **LENGTH**，那么把从该位置到字串结尾的东西都抽取出来。如果 **LENGTH** 是负数，那么该长度是当作在字串尾部剩余那么多字符来理解的。否则，**LENGTH** 表示要抽取的子字串的长度，通常就是你想要的东西。

你可以把 **substr** 当作一个左值（可以给之赋值的東西）来用，这个时候 **EXPR** 也必须是一个合法的左值。如果你给你的子字串赋予比它短的东西，那么该字串将收缩，而如果你给它赋予比它长的东西，那么它会变长。要想保持该字串长度一致，你可能需要用 **sprintf** 或者 **x** 操作符填充或者截断你的数值。如果你试图给一个跨过该字串尾部的未分配区域赋值，那么 **substr** 就会抛出一个例外。

在 **\$_** 的当前值前面增加字串“Larry”，用：

```
substr($var, 0, 0) = "Larry";
```

替换 **\$_** 的第一个字符为“Moe”，用：

```
substr($var, 0, 1) = "Moe";
```

最后，把 **\$var** 的最后一个字符换成“Curly”，用：

```
substr($var, -1) = "Curly";
```

把 **substr** 当作左值使用的另外一个方面就是声明 **REPLACEMENT** 字符串作为其第四个参数。这样就允许你替换 **EXPR** 的某部分并且返回在一次操作之前的东西，就好像你用 **splice** 实现的功能那样。下面一个例子也是把 **\$var** 的最后一个字符替换成“Curly”，并且把那个被替换的字符放到 **\$oldstr** 里：

```
$oldstr = substr($var, -1, 1, "Curly");
```

你不一定只是在赋值语句中使用 **substr** 作为左值。下面的代码把任何空格替换成句点，但是只替换字符串中的最后十个字符：

```
substr($var, -10) =~ s/ ./g;
```

symlink symlink OLDNAME, NEWNAME

这个函数创建一个新的文件，该文件是指向一个旧文件的符号链接。此函数成功时返回真，否则返回假。在那些不支持符号链接的系统上，它在运行时抛出一个例外。要想检查这个例外，你可以用 **eval** 捕获所有可能的错误：

```
$can_symlink = eval { symlink("", ""); 1 };
```

或者使用 **Config** 模块。要注意的是如果你提供了一个相对符号链接，那么它会被解释成相对于该符号链接本身的路径，而不是相对于你的当前工作目录。

又见本章早些时候的 **link** 和 **readlink**。

syscall syscall LIST

这个函数调用列表的第一个元素声明的系统调用（意思就是一次系统调用，而不是一个 **shell** 命令），同时把列表中其他元素作为参数传递给系统调用。（现在，许多这些调用可以通过 **POSIX** 模块更容易地使用。）如果 **syscall(2)** 未实现，那么该函数抛出一个例外。

这些参数按照下面的方式解释：如果一个给定的参数是数字，那么该参数就作为一个 **C** 整数传递。如果不是，那么就传递一个指向该字符串值的指针。你有责任确保这个字符串足够长，以便能够接收任何可能写到它里面去的结果；否则，你就等着核心倾倒吧。你不能拿一个字符串文本（或者其他只读的字符串）当作给 **syscall** 的参数，因为 **Perl** 已经假定任何字符串指针都是可写的。如果你的整数参数不是文本并且在数字环境里从不会被解释，那么你可能需要给它们加 **0** 以强迫它们看起来象数字。

syscall 返回被调用的系统调用返回的任何数值。在 **C** 传统里，如果你那个系统调用失败，那么 **syscall** 返回 **-1** 并且设置 **\$_** (**errno**)。有些系统调用在成功的时候合理地返回 **-1**。操作这样的调用的正确方法是在调用之前赋值给 **\$_=0**；然后如果 **syscall** 返回 **-1** 的话检查 **\$_** 的值。

不是所有的系统调用可以用这个方法访问。比如，**Perl** 支持最多给你的系统调用传递 **14** 个参数。通常这么多已经足够了，但是，对于那些返回多个数值的系统调用就有问题了。比如 **syscall (\$SYS_pipe)**：它返回创建的管道的读端文件号码。我们没有办法检查另外一端的文件号码。你可以用 **pipe** 避免这个例子的问题。要解决一般性的问题，写直接访问系统调用的 **XSUB**（外部过程模块，一种 **C** 写的程序）。然后把你的新模块放到 **CPAN**，使之流行开来。

下面的子过程以浮点数返回当前时间，而不是象 **time** 那样返回整数秒。（它只能在那些支持 **gettimeofday(2)** 系统调用的机器上用。）

```
sub finetime() { package main; # 用于下一个 require require 'syscall.ph'; # 预先把缓冲区
  设置为 32 位长... my $tv = pack("LL", ()); syscall(&SYS)gettimeofday, $tv, undef) >= 0
  or die "gettimeofday: $_!"; my($seconds, $microseconds) = unpack("LL", $tv); return
  $seconds + ($microseconds/ 1_000_000); }
```


假设 Perl 不支持 `setgroups(2)` 系统调用，（注：尽管它可以通过 `$()` 支持），但是你的核心支持。那么你就可以用下面的方法获得它的支持：

```
require 'syscall.ph'; syscall(&SYS_setgroups, scalar @newgids, pack("i*", @newgids))
or die "setgroups: $!";
```

你可能需要按照 Perl 安装指导里描述的那样运行 `h2ph`，检查 `syscall.ph` 是否存在。有些系统可能要求使用 "II" 模板。更头疼的是，`syscall` 假设 C 类型 `int`，`long`，和 `char*` 的尺寸是相等的。希望你不要把 `syscall` 当作移植性的体现。

参阅 CPAN 里的 `Time::HiRes` 模块获取一个有着更好的分辨率的时间装置的更严格的方法。

`sysopen` `sysopen FILEHANDLE, FILENAME, MODE, MASK` `sysopen FILEHANDLE, FILENAME, MODE`

`sysopen` 函数打开 `FILENAME` 给出文件名的文件，并且把它和 `FILEHANDLE` 关联起来。如果 `FILEHANDLE` 是一个表达式，那么它的值用做该文件句柄的名字或者引用。如果 `FILEHANDLE` 是一个未定义值的变量，那么 Perl 将会为你创建一个值。如果调用成功，那么返回值是真，否则是假。

这个函数是你的系统 `open(2)` 系统调用后面跟着一个 `fdopen(2)` 库调用的接口。因此，在这儿你需要略微把自己想象成一个 C 程序员。`MODE` 参数的可能数值和标志位可以通过 `Fcntl` 模块获得。因为不同的系统支持不同的标志位，所以不要指望你的系统里能够用上所有这些标志位。参阅你的 `open(2)` 手册页或者它本地的等价物获取细节。当然，下面的标志在那些带有合理 C 库的系统里是存在的：

标志 | 含义

`O_RDONLY` | 只读 `O_WRONLY` | 只写 `O_RDWR` | 读和写 `O_CREAT` | 如果文件不存在则创建之 `O_EXCL` | 如果文件已经存在则失败 `O_APPEND` | 附加到文件上 `O_TRUNC` | 截断该文件 `O_NONBLOCK` | 非阻塞访问

不过，还有许多其他选项。下面是一些不那么常见的标志：

标志 | 含义

`O_NDELAY` | `O_NONBLOCK` 的老式同义词 `O_SYNC` | 写块时直到数据已经从物理上写入下层的硬件以后（才返回） | 可能还会看到 `O_ASYNC`，`O_DSYNC` 和 `O_RSYNC`。 `O_EXLOCK` | 带着 `LOCK_EX` 的 `flock`（只是劝告性的） `O_SHLOCK` | 带着 `LOCK_SH` 的 `flock`（只是劝告性的） `O_DIRECTORY` | 如果该文件不是目录则失败 `O_NOFOLLOW` | 如果路径中最后一个部件是符号链接则失败 `O_BINARY` | 为 Microsoft 系统 `binmode` 该句柄。有时候还会有一个 `O_TEXT` | 存在以获取相反的行为 `O_LARGEFILE` | 有些系统需要这个标志来标识超过 2 GB 的文件 `O_NOCTTY` | 如果你还没有控制终端，那么打开一个终端文件不会令该终端成为这个进程的 | 控制终端，通常不再需要了。

`O_EXCL` 标志不是用于锁定的：在这里，排它意味着如果该文件已经存在，那么 `sysopen` 失败。

如果 `FILENAME` 命名的文件还不存在，并且 `MODE` 包括 `O_CREAT` 标志，那么 `sysopen` 将在被你的当前 `umask` 修改后的参数 `MASK` 决定的权限范围内（或者如果省略这个参数时缺省是 0666）创建该文件。这样的缺省是有道理的，参阅 `unmask` 里的记录获得一个解释。

用 `open` 和 `sysopen` 打开的文件句柄可以交互地使用。你不必因为碰巧用 `sysopen` 打开了文件而使用 `sysread` 和它的朋友们来操作文件，而如果你用 `open` 打开它也不意味着你就不能用 `sysread`

等函数。`open` 和 `sysopen` 都可以做一些对方做不了的事情。普通的 `open` 可以打开管道，派生进程，设置纪律，复制文件句柄，以及把一个文件描述符号码转换成一个文件句柄。它还忽略文件名开头和结尾的空白，并且把“-”当作一个特殊的文件名。但是如果你要打开的是一个真正的文件，那么 `sysopen` 就可以做 `open` 能做的任何事情。

下面的例子显示了对两个函数的等效调用。我们为了清晰起见省略了 `or die $!` 检查，不过你自己的程序里可是一定要检查这些值的呦。我们将把我们限制于只检查那些实际上在所有操作系统里都可以用的标志。这个活只是用位操作符 `|` 把传递给 `MODE` 参数的数值 `OR`（或）在一起而已。

* 打开文件读取：

```
open(FH, "<", $path); sysopen(FH, $path, O_RDONLY);
```

* 打开文件写，如果必要，创建该文件，或者把原来的文件截除：

```
open(FH, ">", $path); sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

* 打开一个文件用于附加，必要时创建一个：

```
open(FH, ">>", $path); sysopen(FH, $path, O_RDWR);
```

* 打开一个文件用于更新，该文件必须已经存在：

```
open(FH, "+<", $path); sysopen(FH, $path, O_RDWR);
```

而下面的事情是你可以用 `sysopen` 干的，但是却不能用普通的 `open` 干：

* 打开并创建文件用于写，这个文件必须是尚未存在的：

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

* 打开一个文件用于附加，该文件必须已经存在：

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

* 打开一个文件用于更新，必要时创建该文件：

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

* 打开一个文件用于更新，该文件必须尚未存在：

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

* 打开一个非阻塞的只写文件，但如果该文件不存在则不创建它：

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK);
```

在第三十二章描述的 [FileHandle²](#) 模块提供了一套打开文件的面向对象的同义词（以及一点点新的功能）。我们很欢迎你在任何用 `open`, `sysopen`, `pipe`, `socket`, 或者 `accept` 创建的句柄上调用 [FileHandle²](#) 方法（注：实际上是 `IO::File` 或者 `IO::Handle` 方法），就算你不用该模块初始化这些句柄也可以。

```
sysread sysread FILEHANDLE, SCALAR, LENGTH, OFFSET sysread FILEHANDLE,
SCALAR, LENGTH
```

这个函数试图使用低层系统调用 `read(2)` 从你声明的 `FILEHANDLE` 里读取 `LENGTH` 字节到变量 `SCALAR` 中。该函数返回读取的字节数量，或者在 `EOF` 时返回 `0`。（注：在 `Perl` 里没有 `syseof` 函数，但这样是对的，因为 `eof` 在设备文件（比如说终端）上运转的并不怎么正确。用 `sysread` 并

且检查返回值是否为 0 来判断你是否读完了。) 出错时, `sysread` 函数返回 `undef`。 `SCALAR` 将会根据实际读取的长度伸缩。如果声明了 `OFFSET`, 那么它指明应该从字符串里的哪个位置开始读取字节, 这样你就可以在一个用做缓冲区的字符串中间读取。要获取使用 `OFFSET` 的例子, 请参阅 `syswrite`。如果 `LENGTH` 为负数或者 `OFFSET` 指向了该字符串的外边, 那么就会抛出一个例外。

你应该准备处理那些标准 I/O 通常会为你处理的问题 (比如中断了的系统调用)。因为它绕开了标准的 I/O, 所以不要把 `sysread` 和其他类型的读取, `print`, `printf`, `write`, `seek`, `tell`, 或者 `eof` 在同一个文件句柄上混合使用, 除非你准备承受极其稀奇古怪 (和/或痛苦) 的东西。同样, 请注意, 如果你从一个包含 `Unicode` 或者任何其他多字节编码的文件里读取数据, 那么缓冲区的边界有可能落在一个字符的中间。

`sysseek sysseek FILEHANDLE, POSITION, WHENCE`

这个函数使用系统调用 `lseek(2)` 设置 `FILEHANDLE` 的系统位置。它绕开了标准 I/O, 因此把它和读 (除了 `sysread` 以外), `print`, `printf`, `write`, `seek`, `tell`, 或者 `eof` 混合起来使用将会导致混乱。 `FILEHANDLE` 可以是一个表达式, 该表达式的值给出文件句柄的名字。 `WHENCE` 的值为 0 时设置句柄新位置为 `POSITION`, 1 时设置为当前位置加 `POSITION`, 2 时设置为 EOF 加 `POSITION` (通常为负数)。你可以用来自标准 `IO::Seekable` 和 `POSIX` 模块或者——Perl 5.6 里的 `Fcntl` 模块里面的 `SEEK_SET`, `SEEK_CUR` 和 `SEEK_END` 作为 `WHENCE` 的值, 而 `Fcntl` 模块可能更容易移植和更方便一些。

成功时返回新位置, 失败时返回 `undef`。位置零是以特殊字符串 “0 but true” 返回的, 该字符串可以直接当数字使用而不会导致警告。

`system system PATHNAME LIST system LIST`

这个函数为你执行任何系统里的程序并返回该程序的退出状态——而不是它的输出。要捕获命令行上的输出, 你应该用反勾号或者 `qx//`。 `system` 函数的运转非常类似 `exec`, 只不过 `system` 先做一个 `fork`, 然后在 `exec` 之后等待执行的程序的结束。也就是说它为你运行这个程序并且在它完成之后返回, 而 `exec` 用新的程序代替你运行的程序, 所以如果替换成功的话它从不返回。

参数的处理因参数的数目的不同而不同, 就象在 `exec` 里描述的那样, 包括判断是否调用 `shell` 以及你是否用声明另外一个 `PATHNAME` 的方法使用了该函数其他的名称。

因为 `system` 和反勾号阻塞 `SIGINT` 和 `SIGQUIT`, 所以向那些正在这样运行的程序发送这些信号之一 (比如通过一个 `Control-C`) 时并不会中断你的主程序。但是你运行的另外一个程序的确收到这个信号。请检查 `system` 的返回值, 判断你运行的程序是否正常退出。

```
@args = ("command", "arg1", "arg2"); system(@args) == 0 or die "system @args
failed: $?"
```

返回值是和该函数通过 `wait(2)` 系统调用返回的一样的退出状态。在传统的语意里, 要获取实际的退出值, 要除以 256 或者右移 8 位。这是因为低 8 位里有一些其他的东西。(实际上是其他的两东西。) 最低七位标识杀死该进程的信号号码 (如果有的话), 而第八位标识该进程是否倾倒了核心。你可以通过 `$? ($CHILD_ERROR)` 来检查所有失效可能性, 包括信号和核心倾倒:

```
$exit_value = $? >> 8; $exit_value = $? & 127; # 或者 0x7f, 0177, 0b0111_1111
$dumped_core = $? & 128; # 或者 0x80, 0200, 0b1000_0000
```

如果该程序是通过系统 `shell` (注: 定义为 `/bin/sh` 或者任何在你的平台上有意义的东西, 但不是那些用户碰巧在某个时候用到的 `shell`。) 运行的, 这可能是因为只有一个参数而且该参数里面有 `shell` 元字符, 那么通常返回码受那个 `shell` 的怪癖和功能的影响。换句话说, 在这种情况下, 你可能无法获取我们前面描述了详细信息。

syswrite **syswrite** FILEHANDLE, SCALAR, LENGTH, OFFSET **syswrite** FILEHANDLE, SCALAR, LENGTH **syswrite** FILEHANDLE, SCALAR

这个函数试图用 **write(2)** 系统调用向你声明的 **FILEHANDLE** 里写入从变量 **SCALAR** 里获取的 **LENGTH** 字节的数据。该函数返回实际写入的字节数，或者是出错时返回 **undef**。如果声明了 **OFFSET**，那么它指明从字符串里的哪个位置开始写。（比如，你可能用一个字符串做一个缓冲区，这时你就需要这个功能了，或者你需要从一个部分写中恢复过来。）负数 **OFFSET** 表示写应该从该字符串的后面数这么多字节。如果 **SCALAR** 是空的，那么唯一允许的 **OFFSET** 是 0。如果 **LENGTH** 为负数或者 **OFFSET** 指向了字符串的外面，那么就会抛出一个例外。

要从文件句柄 **FROM** 中拷贝数据到文件句柄 **TO**，你可以用下面这样的东西：

```
use Errno qw/EINTR/; $blksize = (stat FROM)[11] || 16384; # 选定的块大小？ while
($len = sysread FROM, $buf, $blksize) { if (defined $len) { next if $! == EINTR; die
"System read error: $!\n" } $offset = 0; while ($len) { # 处理部分写问题 $written =
syswrite TO, $buf, $len, $offset; die "System write error: $!\n" unless defined $written;
$offset += $written; $len -= $written; } }
```

你必须准备处理标准 I/O 通常会为你处理的问题，比如部分写。因为 **syswrite** 绕开了 C 标准 I/O 库，所以不要它的调用和读（除了 **sysread** 以外），写（象 **print**, **printf**, 或者 **write**），或者其他 **stdio** 函数，比如 **seek**, **tell**, 或者 **eof** 混合在一起用，除非你想自找麻烦。

tell **tell** FILEHANDLE **tell**

这个函数返回 **FILEHANDLE** 的当前文件位置（以零为基的字节数）。该值通常可以在程序中稍后的时候传递给 **seek** 函数以找回当前位置。**FILEHANDLE** 可以是一个给出实际文件句柄的表达式，或者一个指向文件对象的引用。如果省略 **FILEHANDLE**，那么该函数返回最后一个读取的文件的位置。只有普通文件的文件位置才有意义。设备，管道，和套接字都没有文件位置。

没有 **sysseek** 函数，你可以用 **sysseek(FH, 0, 1)** 来实现同样的功能。参阅 **seek** 获取一个如何使用 **tell** 的例子。

tell**dir** **tell****dir** DIRHANDLE

这个函数返回在 **DIRHANDLE** 上的 **readdir** 的当前位置。而这个返回值可以给 **seekdir** 用于访问一个目录里的某个特定的位置。该函数和对应的系统库过程在关于可能的目录压缩问题上有这同样的注意事项。该函数可能不是在所有实现了 **readdir** 的地方有实现了，即使该平台实现了它，你也不能计算它的返回值。因为该返回值只是一个晦涩的数值，只对 **seekdir** 有意义。

tie **tie** VARIABLE, CLASSNAME, LIST

此函数把一个变量和一个类绑定在一起，而该类提供了该变量的实现。**VARIABLE** 是要绑定的变量（标量，数组，或者散列）或者类型团（代表一个文件句柄）。**CLASSNAME** 是实现一个正确类型的类名字。

任何额外的参数都传递给该类的合适的构造方法，可能是 **TIESCALAR**, **TIEARRAY**, **TIEHASH** 或者 **TIEHANDLE** 之一。（如果没有找到合适的方法，则抛出一个例外。）通常，那些都是可能被传递给象 **dbm_open(2)** 这样的 C 函数的参数，但是它们的含义是和包相关的。构造器返回的对象随后被 **tie** 函数返回，而如果你想在 **CLASSNAME** 里访问其他方法，那么这个对象就很有用了。（该对象也可以通过 **tied** 函数访问。）因此，一个把散列与一个 **ISAM** 绑定的实现可以提供一些额外的方法用于顺序地跨过一个键字的集合（**ISAM** 里的“S”就是 **sequentially**, 顺序的意思），因为你典型的 **DBM** 实现是不能做这些事情的。

象 **keys** 和 **values** 这样的函数在和 **DBM** 这样的大对象一起使用的时候可能返回非常巨大的数值列

表。你可能会更愿意使用 `each` 函数来遍历这样的列表。比如：

```
use NDBM_File; tie %ALIASES, "NDBM_File", "/etc/aliases", 1, 0 or die "Can't open
aliases: $!\n"; while (($key, $val) = each %ALIASES) { print $key, ' = ', $val, "\n"; }
untie %ALIASES;
```

一个实现散列的类应该提供下列方法：

...（略）。

一个实现普通数组的类应该提供下列方法：

...（略）。

一个实现标量的类应该提供下列方法：

...（略）

一个实现文件句柄的类应该提供下列方法：

...（略）

并不是上面提到的所有方法都需要实现：`Tie::Hash`，`Tie::Array`，`Tie::Scalar`，和 `Tie::Handle` 模块提供了有着合理的缺省的基类。参阅第十四章，捆绑变量，获取所有这些方法的详细描述。和 `dbmopen` 不同，`tie` 函数将不会为你 `use` 或者 `require` 一个模块——你必须自己明确地做这件事情。参阅 `DB_File` 和 `Config` 方法获取有趣的 `tie` 实现。

`tied teid VARIABLE`

这个函数返回一个引用，该引用指向包含在 `VARIABLE` 里的标量，数组，散列或者类型团的下层对象。（`VARIABLE` 是最初用 `tie` 调用把该变量和一个包绑定在一起的同个值。）如果 `VARIABLE` 没有和一个包绑定，它返回未定义的数值。因此，比如，你可以用：

```
ref tied %hash
```

找出你的散列与哪个包捆绑。（假设你忘记了。）

`time time`

这个函数返回自“纪元”以来的没有润秒的秒数，纪元通常是 1970年1月1日 00:00:00 UTC。

（注：不要和创造 Unix 的“历史”相混淆。（其他操作系统可能有不同的纪元，更别说历史了。））返回值可以传递给 `gmtime` 和 `localtime`，可以用于比较 `stat` 返回的文件修改以及访问的时间，还有就是传递给 `utime`。

```
$start = time(); system("some slow command"); $end = time(); if ($end - $start > 1)
{ print "Program started: ", scalar localtime($start), "\n"; print "Program ended: ",
  scalar localtime($end), "\n"; }
```

`times times`

在这个环境里，这个函数返回一个四元素的列表，该列表给出这个进程和它已结束的子进程以秒计（可能是分数）的用户和系统 CPU 时间。

```
($user, $system, $cuser, $csystem) = times(); printf "This pid and its kids have
consumed %.3f seconds\n", $user + $system + $cuser + $csystem;
```

在标量环境里，只返回用户时间。比如，要计算一段 Perl 代码的执行速度：

```
$stat = times(); ... $end = times(); printf "that took %.2f CPU seconds of user time\n",
$end - $start;
```

```
tr/// tr/// y///
```

这是转换（也称之为翻译）操作符，它和 Unix `sed` 程序里的 `y///` 操作符类似，但不论从任何人的角度来看都更好些。参阅第五章。

truncate truncate FILEHANDLE, LENGTH truncate EXPR, LENGTH

这个函数截断在 `FILEHANDLE` 上打开的或者 `EXPR` 命名的文件到指定的长度。如果在你的系统上，`ftruncate(2)` 或者等效的东西没有实现，那么该函数抛出一个例外。（如果你有磁盘空间的话，你总是可以通过拷贝文件的开头来截断它。）该函数在成功的时候返回真，否则返回 `undef`。

uc uc EXPR uc

这个函数返回 `EXPR` 的大写的版本。它是实现双引号字符串里的 `\U` 逃逸的内部函数。Perl 将试图在考虑你的区域设置的前提下做正确的事情，不过我们仍在努力让这个功能也能用于 **Unicode**。参阅 `perllocale` 手册页获取最新的进展。在任何情况下，如果 Perl 使用 **Unicode** 表，`uc` 都会转换成大写字符而不是标题字符。参阅 `ucfirst` 获取转换成标题字符的信息。

ucfirst ucfirst EXPR ucfirst

这个函数返回将 `EXPR` 第一个字符标题化（“**Unicode**”里的标题字符）的版本。而其他字符则不加触动。它是实现双引号字符串里的 `\u` 逃逸的内部函数。如果你 `use locale` 并且你的数据看上去不象 **Unicode**，那么 Perl 会考虑你当前的 `LC_CTYPE` 区域设置，但是我们现在不能做任何保证。

要强制字符串里第一个字符是标题字符而其他的都是小写字符，你可以用：

```
ucfirst lc $word
```

它等效于 `"\u\L$word"`。

umask umask EXPR umask

这个函数用 `umask(2)` 系统调用为该进程设置 `umask` 并返回原来的那个。你的 `umask` 告诉操作系统在创建新文件的时候，哪个权限位是不允许的，包括那些正好是目录的文件。如果省略了 `EXPR`，那么该函数只是返回当前 `umask`。比如，为了确保“**user**”位是允许，而“**other**”位是不允许的，你可以用下面的代码：

```
umask((umask() & 077) | 7); # 不改变组的权限位
```

请记住 `umask` 是一个数字，通常是以八进制形式给出的；它不是八进制位的字符串。如果你拿到的是一个字符串，又见 `oct`，还要记住这个 `umask` 位是普通权限位的补。

Unix 权限位 `rwxr-x---` 是用三个三位集，或者三个八进制位来表示的：**0750**（前面的 **0** 表示它是八进制而不是其中一位）。因为 `umask` 的位是翻转的，所以它代表关闭了的权限位。你提供给你的 `mkdir` 或者 `sysopen` 的权限值（或者“模式”）都会被你的 `umask` 修改，所以就算你告诉 `sysopen` 创建一个权限为 **0777** 的文件，如果你的 `umask` 是 **0022**，那么创建出来的文件的权限也是 **0755**。如果你的 `umask` 是 **0027**（组不能写，其他不能读，写，和执行），那么给 `sysopen` 传递一个 `MASK` 为 **0666** 的值将创建一个模式为 **0640** 的文件（因为 **0666 & ~0027** 是 **0640**）。

这里是一些建议：使用模式 **0666** 创建普通文件（在 `sysopen` 里）以及 **0777** 给目录（用 `mkdir`）和可执行文件。这样就给予用户自由的选择：如果它们想保护文件，那么它们选择进程

`umask 022, 027`，或者甚至特别反社会的掩码 `077`。程序最好让用户自己做策略决策。这条规则的例外是那些需要写私人文件的程序：邮件文件，网络浏览器的 `cookie`，`.rhost` 文件，等等。

如果你的系统里没有实现 `umask(2)` 并且你试图限制你自己的权限（也就是说，如果 `(EXPR & 0700) > 0`），那么你就会触发一个运行时例外。如果你的平台没有实现 `umask(2)` 并且你不准备限制自己的权限，那么这个函数简单地返回 `undef`。

undef undef EXPR undef

undef 是我们所谓的“未定义值”的缩写。同时，它还是一个永远返回未定义值的函数的名字。我们很高兴能混淆它们俩。

相同的是，如果你给 **undef** 函数提供一条记录作为它的参数，那么它还可以明确的解除该记录的定义。如果声明了 **EXPR** 参数，那么它必须是一个左值。因此你可能只能在一个标量数值，整个散列或者数组，一个子过程名字（用 `&` 前缀），或者一个类型团上这么用，任何和该对象关联的存储空间都将被恢复用于重复使用（不过在大多数操作系统上都不是返回给系统）。**undef** 函数对大多数特殊变量可能都不会做你想像的处理。在象 `$1` 这样的只读变量上使用将抛出一个例外。

undef 函数是一个单目操作符，不是列表操作符，因此你只能每次解除一个东西的定义。下面是一些 **undef** 作为单目操作符的用法：

```
undef $foo; undef $bar{'blurfl'}; # 和 delete $bar{'blurfl'} 不同
undef @ary; undef %hash; undef &mysub; undef *xyz; # 删除 $xyz, @xyz, %xyz, &xyz 等等。
```

如果没有参数，**undef** 只是用做数值：

```
select(undef, undef, undef, $naptime);
```

```
return (wantarray ? () : undef) if $they_blew_it; return if $they_blew_it; # 一样的东西
```

你可以把 **undef** 用做一个列表赋值中左边的一个占位符，这个时候右边的对应的数值只是简单地抛弃。除此之外，你不能在其他地方拿 **undef** 做左值：

```
($a, $b, undef, $c) = &foo; # 忽略返回的第三个数值
```

同样，不要拿任何东西和 **undef** 做比较——那样不会按照你想像的方式处理的。它所做的事情只是与 `0` 或者空字符串比较。使用 **defined** 函数判断一个数值是否定义。

unlink unlink LIST unlink

这个函数删除一系列文件。（注：实际上，在一个 **POSIX** 文件系统里，它删除指向真实文件目录记录（文件名）。因为一个文件可以从一个或多个目录里引用（链接），该文件不会被删除，直到指向它的最后一个引用被删除。）此函数返回被成功删除的文件名的个数。一些简单的例子：

```
$count = unlink 'a', 'b', 'c'; unlink @goners; unlink glob("*.orig");
```

除非你是超级用户或者给 **Perl** 使用了 `-U` 命令行选项，否则 **unlink** 函数不会删除目录。即使符合这些条件，你也要注意删除一个目录可能造成对你的文件系统的彻底损坏，应该用 **rmdir** 代替。

下面是一个带有非常简单的错误检查的 **rm** 命令：

#!/usr/bin/perl

```
@cannot = grep {no unlink} @ARGV; die "$0: could not unlink @cannot\n" if @cannot;
```

unpakc unpack TEMPLATE, EXPR

这个函数是 **pack** 的逆操作：它根据 **TEMPLATE** 把一个表示一个数据结构的字串（**EXPR**）扩展成一系列数值并返回那些数值。在标量环境里，它可以用于解包一个数值。这里的 **TEMPLATE** 有着和 **pack** 函数里的大多数格式——它声明要解包的数值的顺序和类型。参阅 **pack** 函数获取有关 **TEMPLATE** 的详细描述。如果 **TEMPLATE** 里有非法元素，或者试图跨过 **x**, **X**, 或者 **@** 格式字串的外面，都会抛出例外。

该字串会分解成 **TEMPLATE** 里描述的片段。每个片段都独立地转化成一个数值。通常，该字串的字节要么是 **pack** 的结果，要么代表某种类型的 **C** 结构。

如果一个域的重复计数比输入字串剩余部分允许的尺寸大，那么重复计数就会被不声不响地缩小。

（不过，你通常会在这个地方放一个 ***** 做重复计数。）如果输入字串比 **TEMPLATE** 描述的长，那么字串剩余的部分被忽略。

unpack 函数对纯文本数据也很有用，而不仅仅是对二进制数据管用。设想你有一个数据文件，它的内容看起来象下面这样：

...（略）

你不能用 **split** 来分析出各个域，因为这里没有明显的分隔符。这里的域是由它们的字节偏移量来决定的。因此就算这是一个普通的文本记录，但因为它是固定格式的，所以你就可以用 **unpack** 把它们分解成数据域：

```
while (<>) { ($year, $title, $author) = unpack("A4 x A23 A*", $_); print "$author won
${year}'s Hugo for $title.\n"; }
```

（我们在这里写成 **\${year}'s** 的原因是 Perl 会把 **\$year's** 当作 **\$year::s** 看待。）

下面是一个完整的 **undecode** 程序：

! /usr/bin/perl

```
$_ = <> until ($mode, $file) = /^begin\s*(\d*)\s*(\S*)/; open(OUT, "> $file") if $file
ne ""; while (<>) { last if /^end/; next if /[a-z]/; next unless int((((ord() - 32) & 077)
+ 2) / 3) == int (length() / 4); print OUT unpack "u", $_; } chmod oct($mode), $file;
```

除了那些 **pack** 里允许的数据域以外，除了各个项自身以外，你还可能在一个数据域前面前缀一个 **%number** 作成所有项的简单的 **number** 位附加校验和。该校验和是通过类加扩展数值的数字值来计算的（对于字串域，求 **ord(\$char)** 的和，而对于位域，计算零和一的和）。比如，下面的代码计算和 [SysV²](#) **sum(1)** 相同的数字：

```
undef $/; $checksum = unpack ("%32C*", <>) % 65535;
```

下面的代码高效的计算一个位流里的设置上的位的个数：

```
$setbits = unpack "%32b*", $selectmask;
```

下面是一个简单的 **BASE64** 解码器：

```
while (<>) { tr#A-Za-z0-9+/#cd; # 删除非 base64 字符 tr#A-Za-z0-9+/# -_#; # 转
换成 uuencode 格式 $len = pack("c", 32 + 0.75*length); # 计算长度字节 print unpack
("u", $len . $_); # uudecode 并打印 }
```

unshift **unshift** **ARRAY**, **LIST**

这个函数做 **shift** 的逆操作。（或者是 **push** 的逆操作，取决于你怎么看它。）它在数组前面增加 **LIST**，并返回在数组里的新的元素个数：


```
unshift @ARGV, '-e', $cmd unless ARGV[0] =~ /^-/;
```

请注意 `LIST` 是整个放到前面，而不是每次一个元素，因此放到前面的元素保持相同顺序。用 `reverse` 实现这些元素的翻转。

`untie untie VARIABLE`

打破 `VARIABLE` 里包含的变量或者或者类型团和与它捆绑的包之间的绑定。参阅 `tie`，以及第十四章的全部，尤其是“一个精细的松绑陷阱”节。

```
use use MODULE VERSION LIST use MODULE VERSION () use MODULE VERSION use
MODULE LIST use MODULE () use MODULE use VERSION
```

`use` 声明装载一个模块（如果它还没有被装载），并且把子过程和变量从这个命名模块输入到当前包。（从技术上来讲，它从那个命名模块向当前包输入一些语意，通常是通过把一些子过程或者变量名作成你的包里的别名的方法。）大多数 `use` 的声明看起来象：

```
use MODULE LIST;
```

这样和下面是完全一样的：

```
BEGIN { require MODULE; import MODULE LIST; }
```

`BEGIN` 迫使 `require` 和 `import` 在编译时发生。`require` 确保该模块在还没有装载的时候装入内存。`import` 不是内建的函数——它只是一个普通的类方法，调用名字叫 `MODULE` 的包，告诉该模块把列表里的特性拖到当前包里来。模块可以用自己喜欢的任何方法实现它的输入方法，尽管大多数只是通过从 `Exporter` 类中继承 `import` 方法。`Exporter` 类在 `Exporter` 模块中定义。参阅第十一章，模块，以及 `Exporter` 模块获取更多信息。如果找不到 `import` 方法，那么调用将不声不响地忽略。

如果你不希望你的名字空间被修改，那么明确地提供一个空列表：

```
use MODULE ();
```

它和下面的代码完全一样：

```
BEGIN { require MODULE; }
```

如果给 `use` 的第一个参数是象 `5.6.2` 这样的版本号，那么当前执行着的 `Perl` 版本必须至少和声明的版本一样新。如果当前的版本比 `VERSION` 小，那么就会打印出一条错误信息然后 `Perl` 马上退出。这样就可以有效地在装载需要更新的版本的库模块之前检查当前 `Perl` 的版本，因为有时候我们必须“破坏”老版本的错误特性。（我们总是试图尽可能不破坏任何东西。而实际上我们总是试图少破坏东西。）

谈到不破坏其他东西，`Perl` 仍然接受下面形式的老版本号：

```
use 5.005_03;
```

不过，为了和工业标准更好的看齐，`Perl 5.6` 现在接受（并且也更愿意使用）下面的三段式：

```
use 5.6.0; # 它是版本 5，子版本 6，补丁级 0。
```

如果 `VERSION` 参数在 `MODULE` 后面出现，那么 `use` 将在类 `MODULE` 里调用 `VERSION` 方法，同时把给出的 `VERSION` 当作参数给他。请注意在 `VERSION` 后面没有逗号！缺省的 `VERSION` 方法（通常是从 `UNIVERSAL` 类里继承过来的。）会在给出的版本大于变量 `$Module::VERSION` 的值的情况下发表意见。

参阅第三十二章获取一个标准模块的列表。

因为 `use` 提供了一个非常开放的接口，所以用法（编译器指示器）也是通过模块来实现的。当前实现了的用法包括：

```
use autouse 'Carp' => qw(carp croak); use bytes; use constant PI => 4 * atan2(1,1);
use diagnostics; use integer; use lib '/opt/projects/spectre/lib'; use locale; use sigtrap
qw(die INT QUIT); use strict qw(subs vars refs); use warnings "deprecated";
```

许多这些用法模块向当前词法范围输入语意。（它和普通模块不同，普通模块只是向当前包里输入符号，而除了该词法范围是在带有该包的情况下编译的以外，那些符号和当前词法范围没有什么关系。也就是说，哦，看看第十一章吧。）

还有一个对应的声明，`no`，它“戒除”任何原来用 `use` 输入的东西，让它们变得不再重要：

```
no integer; no strice 'refs'; no utf8; no warnings "unsafe";
```

参阅第三十一章获取一个标准用法的列表。

utime utime LIST

该函数改变一系列文件里的每一个文件的访问和修改时间。列表的头两个元素必须是数字化的访问和修改时间，顺序是访问在前修改在后。该函数返回成功改变的文件的数目。每个文件的 `inode` 修改时间设置成当前时间。下面是一个 `touch` 命令的例子，它设置该文件的修改日期（假设你是所有者）为近一个月后：

! /usr/bin/perl montouch - post-date files now + 1 month

```
$day = 24 * 60 * 60; # 24 小时的秒数 $later = time() + 30 * $day; # 30 天接近一个月
utime $later, $later, @ARGV;
```

values values HASH

这个函数返回一个包含指定散列 `HASH` 里的所有值的列表。这些值是以看似随机的顺序返回的，但是这个顺序和 `keys` 或 `each` 函数在同一个散列上生成的顺序相同。怪异的是，如果要通过一个散列的数值对它进行排序，那么你通常需要使用 `keys` 函数，所以看看 `keys` 函数里的例子找找灵感。

你可以用这个函数修改一个散列的数值，因为它返回的列表包含数值的别名，而不是拷贝。（在早期的版本里，你需要用散列的片段来实现这个功能。）

```
for (@hash{keys %hash}) { s/foo/bar/g } # 老办法 for (values %hash) { s/foo/bar/g }
# 新手段
```

在一个和某个巨大的 `DBM` 文件捆绑的散列上使用 `values` 也肯定会生成一个巨大的列表，导致你拥有一个巨大的进程。你可能应该使用 `each` 函数，它会一个一个地遍历散列记录，而不会把它们的所有东西都吞进一个庞大的，哦，应该是巨大的列表里。

vec vec EXPR, OFFSET, BITS

`vec` 函数制造一个存储紧凑的无符号整数的列表。这些整数在一个普通的 `Perl` 字串里尽可能紧密的绑在一起。`EXPR` 里的字串被当作一个位串对待，该串是用若干元素组成的，而元素的数目取决于字串的长度。

`OFFSET` 声明你关心的特定元素的索引。读和写元素的语法是一样的，因为 `vec` 根据你是在左值环

境还是右值环境里来存储和恢复元素值。

BITS 声明每个元素以位计算的时候宽度是多少，它必须是二的幂：**1, 2, 4, 8, 16, 或 32**（有些平台上还有 **64**）。（如果声明了任何其他数值，那么就会抛出一个例外。）因此每个元素都可以包含一个范围在 **0 .. (2**BITS)-1** 的整数。对于小一些的尺寸，那么每个字节里都会尽可能多的打包进去元素。如果 **BITS** 是 **4**，那么每个字节里有两个元素（通常它们被称为半字节（**nybble**））。等等。大于一个字节的整数是以大头在前的字节序存储的。

一个无符号整数列表可以存储在一个标量变量里，方法是把它们分别赋予 **vec** 函数。（如果 **EXPR** 不是一个合法的左值，那么抛出一个错误。）在随后的例子里，那些元素是 **4** 位宽：

```
$bitstring = ""; $offset = 0;
```

```
foreach $num (0, 5, 5, 6, 2, 7, 12, 6) { vec($bitstring, $offset++, 4) = $num; }
```

如果一个元素超出了它要写的字串的结尾，那么 **Perl** 先用足够的零内容字节扩展该字串。

存储在标量变量里的向量然后就可以通过声明正确的 **OFFSET** 来检索：

```
$num_elements = length($bitstring)*2; # 每个字节 2 元素
```

```
foreach $offset (0 .. $num_elements-1) { print vec($bitstring, $offset, 4), "\n"; }
```

如果选择的元素超出了字串的结尾，那么返回 **0**。

用 **vec** 创建的字串还可以用逻辑操作符 **|**, **&**, **^**, 和 **~** 操作。如果两个操作数都是字串，那么这些操作符将假定需要进行的是位串操作。参阅第三章，单目和双目操作符，“位操作符”一节里的例子。

如果 **BITS == 1**，那么就可以创建一个所有位都存储在一个标量里的位序列。顺序是这样的，**vec (\$bitstring, 0,1)** 保证进入字串里的第一个字节的最低位。

```
@bits = (0,0,1,0, 1,0,1,0, 1,1,0,0, 0,0,1,0);
```

```
$bitstring = ""; $offset = 0;
```

```
foreach $bit (@bits) { vec($bitstring, $offset++, 1) = $bit; }
```

```
print "$bitstring\n"; # "TC", 也就是 '0x54', '0x43'
```

一个位串可以通过声明一个“**b***”模板给 **pack** 或者 **unpack** 从一串 **1** 和 **0** 转换过来，或者是转换成这些 **1** 和 **0** 的字串。另外，**pack** 可以和“**b***”模板一起使用从一个 **1** 和 **0** 的字串创建一个位串。该顺序和 **vec** 需要的顺序是兼容的。

```
$bitstring = pack "b*", join(", ", @bits); print "$bitstring\n"; # "TC", 和前面例子一样
```

unpack 可以用于从该位串中抽取这个 **0** 和 **1** 的列表：

```
@bits = split(/,/, unpack("b*", $bitstring)); print "@bits\n"; # 0 0 1 0 1 0 1 0 1 1 0 0 0  
0 1 0
```

如果你知道位串的位的确切长度，那么这个长度可以用于“*****”的位置。

参阅 **select** 获取使用 **vec** 生成的位图的其他例子。参阅 **pack** 和 **unpack** 获取操作二进制数据的更高级别的方法。

wait wait

这个函数等待一个子进程退出并返回消失了的进程的 **PID**，或者如果没有子进程了就返回 **-1**（或者在一些系统里，子进程自动被收割也如此。）它在 **\$?** 里返回的状态和在 **system** 里描述的一样。如果你有僵死子进程，那么你就应该调用这个函数，或者 **waitpid**。

如果你在等待一个子进程，但是用 **wait** 没有找到它，那么你可能就是调用了 **system**，关闭了一个管道，或者在 **fork** 和 **wait** 之间使用了反勾号。这些构造也做 **wait(2)** 并且因此可能收割你的子进程。使用 **waitpid** 避免这样的情况。

waitpid waitpid PID, FLAGS

这个函数等待特定的子进程结束并在该进程死亡之后返回其 **PID**，如果没有其他的子进程时返回 **-1**，或者 **FLAGS** 里的标志声明的是非阻塞状态而该进程尚未退出，则返回 **0**。死亡的进程返回的状态存储在 **\$?**，并且和 **system** 里描述的一样。要获取有效的标志值，那么你必须输入

“**:sys_wait_h**”从 **POSIX** 里输入标签组。下面是一个等待所有挂起的僵死进程的非阻塞的例子：

```
use POSIX ":sys_wait_h"; do { $kid = waitpid(-1, &WNOHANG); } until $kid == -1;
```

在那些既没有实现 **waitpid(2)** 也没有实现 **wait4(2)** 系统调用的平台上，你可以声明的 **FLAGS** 只有 **0**。换句话说，你在那里可以等待一个特定的 **PID**，但是你不能在非阻塞模式里做这些事情。

在一些系统里，返回值为 **-1** 意味着该子进程被自动收割了，因为你设置了 **\$SIG{CHLD} = 'IGNORE'**。

wantarray wantarray

如果当前正在执行的子过程在寻找列表数值，那么此函数返回真，否则返回假。如果调用环境需要的是一个标量，那么该函数返回一个定义了的假（**""**），而如果调用环境并不需要任何东西，（也就是说，空环境）那么返回一个未定义的真值（**undef**）；

下面是它的典型用法的例子：

```
return unless defined wantarray; # 不需要干什么事情 my @a = complex_calculation();
return wantarray ? @a : \@a;
```

又见 **caller**。这个函数真是应该命名为“**wantlist**”，但是我们是在列表环境还叫数组环境的时候命名它的。（译注：数组英文词是“**array**”，列表英文词是“**list**”。）

warn warn LIST warn 这个函数生成一条错误信息，象 **die** 那样把 **LIST** 打印到 **STDERR**，但是并不试图退出或者抛出一个例外。比如：

```
warn "Debug enabled" if $debug;
```

如果 **LIST** 为空并且 **\$@** 已经包含一个数值（通常是前面的 **eval** 留下来的），那么字符串 **"\t... caught"** 在 **STDERR** 上附加在 **\$@** 后面。（这样做类似 **die** 传播错误的方法，只不过 **warn** 并不传播（抛出）该例外。）如果你提供的字符串上是空的，那么使用 **"Warning: something's wrong"**。

和 **die** 一样，如果你提供的字符串并不是以换行符结尾，那么自动附加文件和行号信息。**warn** 函数和 **Perl** 的 **-w** 命令行选项没有关系，但是可以和它一起使用，比如在你想模拟内建函数的时候：

```
warn "Something wicked\n" if $^W;
```

如果安装了 **\$SIG{__WARN__}** 句柄，那么不会打印任何信息。这个句柄是负责对它看到的信息进行适当处理用的。你想这么做的一个原因可能是把简单的警告转化成一个例外：

```
local $SIG{__WARN__} = sub { my $msg = shift; die $msg if $msg =~ /isn't
```

```
numeric/; };
```

因此大多数句柄都必须对那些它们原先没有准备处理的警告安排显示处理的工作，方法是在句柄里调用 **warn**。这么做非常安全，它不会产生无限的循环，因为 **WARN** 挂钩不会在 **WARN** 里面被调用。这个行为和 **\$SIG{__DIE__}** 的行为（它不会消除错误文本，但是可以再次调用 **die** 来改变它）略有区别。

使用 **WARN** 句柄给我们提供了一个强有力的抑制所有警告的方法，甚至连那些强制性的警告也给抑制住了。有时候你需要把这个东西封装在 **BEGIN{}** 块里，这样它就可以在编译时起做用：

扫荡掉所有编译时间警告

```
BEGIN { $SIG{__WARN__} = sub { warn $_[0] if $DOWARN } } my $foo = 10; my
$foo = 20; # 不要警告我说重复了 my $foo, # 不过，这可是你说的！
```

在这以前没有编译时和运行时的警告

```
$DOWARN = 1; # 不是一个内建的变量
```

在这以后打开运行时的警告

```
warn "\$foo is alive an $foo!"; # 做显示
```

参阅 **use warnings** 用法获取警告的词法范围里的控制。参阅 **Carp** 模块里的 **carp** 和 **cluck** 函数获取其他制造警告信息的方法。

```
write write FILEHANDLE write
```

这个函数写一条格式化了了的记录（可能是多行）到声明的文件句柄，使用与该文件句柄相关联的格式——参阅第七章里的“格式变量”一节。缺省时与文件句柄相关联的格式是和文件句柄同名的那个。不过，一个文件句柄的格式可以在你 **select** 了该句柄以后修改 **\$~** 变量来修改：

```
$old_fh = select(HANDLE); $~ = "NEWNAME"; select($old_fh);
```

或者说：

```
use IO::Handle HANDLE->format_name("NEWNAME");
```

因为格式是放到一个包名字空间里的，所以如果该 **format** 是在另外一个包里声明的，那么你可能不得不用该格式的全称：

```
$~ = "OtherPack::NEWNAME";
```

表单顶部（**Top-of-form**）的处理是自动操作的：如果当前页里面没有足够的空间容纳格式化的记录，那么通过写一个进纸符来续该页，这时候在新页上使用一个特殊的表单顶部格式，然后写该记录。在当前页里余下的行数放在变量 **\$-** 里，你可以把它设置为 **0** 以强迫在下次 **write** 的时候使用新的一页。（你可能先得 **select** 该文件句柄。）缺省时，页顶格式的名字就是文件句柄后面加上 **"_TOP"**，不过，一个文件句柄的格式可以在你 **select** 了该句柄以后修改 **\$~** 变量来修改，或者说：

```
use IO::Handle; HANDLE->format_top_name("NEWNAME_TOP");
```

如果没有声明 **FILEHANDLE**，那么输出就会跑到当前缺省的输出文件句柄，这个文件句柄初始时是 **STDOUT**，但是可以用单参数形式的 **select** 操作符修改。如果 **FILEHANDLE** 是一个表达式，那么在运行时计算该表达式以决定实际的 **FILEHANDLE**。

如果声明的 **format** 或者当前的页顶 **format** 并不存在，那么抛出一个例外。

糟糕的是，**write** 函数不是 **read** 的逆操作。用 **print** 做简单的字符串输出。如果你找到这个函数的原

因是因为你想绕开标准 I/O，参阅 `syswrite`。

`y// y///`

转换操作符（因历史原因，也叫做翻译操作符），也叫做 `tr///`。参阅第五章。

Revision: r1.1 - 14 Jan 2006 - 15:02 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [Reference](#) > [PerlReference_Functions2](#)

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.
向为这里贡献想法, 文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)