

第二十九章，函数 (A-D)

- ↓ 第二十九章，函数 (A-D)
- ↓ 29.1 按类分的 Perl 函数
- ↓ 29.2 按照字母顺序排列的 Perl 函数

为了便于参考，本章以字母顺序（注：有时候，紧密联系的函数在系统手册页里组合在一起，因此我们在这里也将尊重那些分组。比如，要寻找 `endpwent` 的描述，你得先找 `getpwent`。）描述内建的 Perl 函数。每个函数描述以一个该函数语法的简短概要开头。象 **THIS** 这样的参数名字代表实际表达式的占位符，而跟在语法概要后面的文本将描述提供（或者省略）该实际参数的语意。

你可以把函数和文本以及变量想象成一个表达式里的项。或者你可以把它们想象成前缀操作符，处理它后面的参数。要知道我们有一半时间叫它们操作符。

这些操作符，哦，是函数，中有一些接受一个 **LIST** 作为一个参数。这个 **LIST** 的元素应该用逗号分隔（或者用 `=>`，它只是逗号的一种有趣的形式）。这个 **LIST** 的元素在列表环境中计算，所以每个元素都会返回一个标量或者一个列表值——取决于它对列表环境的敏感程度。每个返回的值，不管是标量还是列表，都会被解释成一个总体标量值序列的一部分。也就是说，把所有列表平面化成一个列表。从接收参数的函数的角度来看，全部参数 **LIST** 总是一个一维列表数值。（要把一个数组代换成一个元素，你必须明确地创建一个引用并用它代换该数组。）

预定义的 Perl 函数既可以带着圆括弧使用，也可以不带圆括弧使用；在本章中的语法概要里是省略圆括弧的。如果你确实使用圆括弧，其简单但有时有些怪异的规则如下：如果它看上去象函数，那它就是函数，所以优先级并不紧要。否则，它就是一个列表操作符或者一个单目操作符，而优先级就有关系了。这里要小心，因为就算你在关键字和它的左圆括弧之间放了空格，也不能不让它成为一个函数：

```
print 1+2*4      #打印 9。
print(1+2) * 4;  #打印 3!
print (1+2)*4;   #还是打印 3!
print +(1+2)*4;  # 打印12。
print ((1+2)*4)  # 打印12。
```

如果你带着 `-w` 开关运行 Perl，它会警告你这些问题。比如，上面的第二和第三行产生下面这样的信息：

```
print (...) interpreted as function at - line 2.
Useless use of integer multiplication in void context at - line 2.
```

给定一些函数的简单定义的前提下，你可以使用的传递参数的方法有相当可观的自由度。比如，使用 `chmod` 的最常见的方法就是把文件权限（模式）作为一个初始参数传递：

```
chmod 0644, @array
```

但是 `chmod` 的定义只是说：

```
chmod LIST
```

所以你也可以说：

```
unshift @array, 0644;
chmod @array;
```

如果列表的第一个参数不是一个有效的模式，`chmod` 调用将失败。但这是一种运行时语意问题，和

调用它的语法没有关系。如果该语意要求任何特殊的参数必须先传递，那么文本将描述这些限制。

与简单的 `LIST` 函数相反，其他函数强制附加的语法约束。比如，`push` 的语法概要看起来象下面这样：

```
push ARRAY, LIST
```

这意味着 `push` 要求一个正确的数组作为它的第一个参数，但是不在乎随后的参数。那就是最后的 `LIST` 的意思。（`LIST` 总是最后出现，因为它把所有剩下的数值都吃掉。）如果一个语法概要要在 `LIST` 之前包含任何参数，那么编译器会认为那些参数是有语法区别的，而不仅仅是在它稍后运行时解释器看到的语意区别。这样的参数从来不会放在列表环境里计算。它们可能会在标量环境里计算，或者它们也可能是象 `push` 里的数组一样的引用参数。（描述会告诉你某个参数是哪种类型。）

对于那些自身的操作直接基于 C 库的函数，我们不想复制你的系统的文档。如果一个 `function` 描述里说：参阅 `function(2)`，它的意思就是你应该查看对应的 C 版本的函数，学习更多其语意方面的内容。如果你安装了手册页，那么圆括弧里的数字指示你查阅手册页时会看到的系统程序员手册一节。（当然，如果你没装手册页那么你也看不到。）

这些手册页可能记载了一些系统相关的行为，象 `shadow` 口令文件，访问控制列表，等等。许多来自 Unix 里的 C 库的 Perl 函数现在甚至在非 Unix 平台上都有仿真。比如，也许你的操作系统可能不支持 `flock(2)` 或 `fork(2)` 系统调用，但 Perl 也会通过使用所有你的平台提供的设施尽可能地模拟这些调用。

有时候，你会发现那些记载了的 C 函数比对应的 Perl 函数的参数多。通常，那些缺少了的参数是 Perl 已经知道的东西，比如前面一个参数的长度，所以你用不着给 Perl 提供它们。任何其他的不同都是因为 Perl 和 C 在声明文件句柄和成功/失败值上的区别造成的。

通常，Perl 里用做系统调用封装的函数和该系统调用同名（比如 `chown(2)`，`fork(2)`，`closedir(2)`，等等），都是成功时返回真，而失败时返回 `undef`，在随后的描述里也有提到。这样的特性与这些操作的 C 库的接口是不同的，C 接口在失败时都返回 `-1`。这条规则的例外是 `wait`，`waitpid`，和 `syscall`。系统调用在失败的时候还设置特殊的 `$_`（`$OS_ERROR`）变量。除非故意，其他函数并不这样做。

对于那些在标量或者列表环境中都可以使用的函数，在标量环境中失败的时候通常是返回一个假值（通常是 `undef`）来表明而在列表环境中通常是返回空列表。成功地执行通常是用返回一个（在环境中）可以计算出真值的数值来表明的。

记住下面的规则：没有任何规则把一个函数在列表环境中的行为和标量环境中的行为相互联系起来，反之亦然。这两种情况可能是做两件完全不同的事情。

每个函数都知道它被调用的环境。同一个函数，如果它在列表环境中调用时返回一个列表，那么它在标量环境中调用将返回它认为最合适的任何类型的数值。有些函数返回它在列表环境中返回的列表的长度，有些操作符返回列表中的第一个数值。有些函数返回列表中最后的数值。有些函数返回“其他”数值——如果那些东西可以通过数字或者名字找出。有些函数返回成功操作的计数。通常，Perl 函数会干你想干的事情，除非你想要一致性。

最后一条注意：我们已经非常注意保持自己对“字节”和“字符”两个术语的使用的一致性了。因历史原因，这些术语已经是互相混淆在一起了（并且它们自身也含糊不清）。但是当我们说“字节”的时候，我们的意思总是一个八位字节，8位。而当我们说到“字符”的时候，我们的意思是一个抽象的字符，“通常”是一个 Unicode 字符，它在你的字符串里可能会用一个或多个字节代表。

不过要注意我们说的“通常”。Perl 在 `use bytes` 声明的范围里故意混淆了字节和字符的概念，所以如果我们说到“字符”，那么在 `use bytes` 环境里时你应该把它看作一个字节，否则，看作一个 Unicode。换句话说，`use bytes` 只是把字符的定义封装回到了老版本的 Perl 里的概念。因此，如

果我们说一个标量 `reverse` 一个字符一个字符地反转一个字符串，你就别问我们这里真的意思是字符还是字节，因为答案就是，“没错，它就是干这事。”

29.1 按类分的 Perl 函数

下面是按照类别排列的 Perl 的函数和类函数关键字。有些函数在多个标题下出现。

- 标量操作
`chomp, chop, chr, crypt, hex, index, lc, lcfirst, length, oct, ord, pack, q//, qq//, reverse, rindex, sprintf, substr, tr///, uc, ucfirst, y///`
- 正则表达式和模式匹配
`m//, pos, qr//, quotemeta, s///, split, study`
- 数学函数
`m//, pos, qr//, quotemeta, s///, split, study`
- 数珠处理
`abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand`
- 列表处理
`pop, push, shift, splice, unshift`
- 散列处理
`delete, each, exists, keys, values`
- 输入和输出
`binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, readpipe, rewinddir, seek, seekdir, select (ready file descriptors), syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write`
- 定长数据和记录
`pack, read, syscall, sysread, sysseek, syswrite, unpack, vec`
- 文件句柄，文件，和目录
`chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, select (ready file descriptors), select (output filehandle), stat, symlink, sysopen, umask, unlink, utime`
- 程序流控制
`caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray`
- 范围（搜寻）
`caller, import, local, my, no, our, package, use`
- 杂项
`defined, dump, eval, formline, lock, prototype, reset, scalar, undef, wantarray`
- 进程和进程组
`alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid`
- 库模块
`do, import, no, package, require, use`
- 类和对象
`bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use`

- 低层套接字访问
accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair
- **System V** (系统五) 进程间通讯
msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite
- 抓取用户和组信息
endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent
- 抓取网络信息
endprotoent, endservent, gethostbyaddr, ethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent
- 时间
gmtime, localtime, time, times

29.2 按照字母顺序排列的 Perl 函数

后面的许多函数名字都带有许多注解。下面是它们的含义：

- `$_` 使用 `$_` (`$ARG`) 作为缺省变量
- `$!` 在系统错误时设置 `$!` (`$OS_ERROR`)。
- `$@` 抛出例外，用 `eval` 捕获 `$@` (`$EVAL_ERROR`)。
- `$?` 当子进程退出时设置 `$?` (`$CHILD_ERROR`)。
- `T` (实) 感染返回的数据
- `T` (虚) 在一些系统，区域，或者句柄设置下感染返回数据
- `XARG` 如果给出一个参数是不合适的类型，抛出一个错误
- `XRO` 如果修改一个只读对象，则抛出一个例外。
- `XT` 如果填充感染了的数据，则抛出一个例外。
- `XU` 如果在当前的平台上没有实现则抛出一个例外。

这里没有标记那些给它们填充了感染了的数据则返回感染数据的函数，因为大多数函数都是这样。尤其是如果你在 `%ENV` 或者 `@ARGV` 上使用任何函数，你都会拿到感染了的数据。

那些标记着 (`XARG`) 的函数如果在要求你给它一个特定类型的参数，（比如用于 `I/O` 操作的文件句柄啦，用于 `bless` 的引用啦等等）但是没有收到的时候，它们就会抛出一个例外。

那些标记着 (`XRO`) 的函数有时候需要修改它们的参数。如果因为该参数标记着只读标志而不能修改，那么它们就会抛出一个例外。只读变量的例子是那些包含在模式匹配过程中捕获的数据的特殊变量以及那些实际上是一个常量的别名的变量等。

标记着 (`XU`) 的函数可能没有在所有平台上实现。尽管许多这些函数都是用它们在 `Unix C` 库中的对应函数的名字命名的，但也不要因为你运行的不是 `Unix` 系统就认为你不能调用任何这些函数。它们中有许多都通过仿真实现了，甚至是那些你可能从来没想着要看到的——比如在 `Win32` 系统上的 `fork`，它在 `5.6` 版本的 `Perl` 里是可以用的。更多的有关系统相关的函数的移植性和行为的信息，请参阅 `perlport` 手册页，以及任何随着你的 `Perl` 移植版本一起发布的平台相关的文档。

那些抛出其他杂项例外的函数是用 (`$@`) 标记的，包括抛出范围错误的数学函数，比如 `sqrt(-1)`。

```
abs
abs VALUE
abs
```

这个函数返回它的参数的绝对值。

```
$diff = abs($first - $second);
```

注意：这里和随后的例子，好的编程风格（以及 **use strict** 用法）将要求你加一个 **my** 来声明一个新的词法范围的变量，比如：

```
my $diff = abs($first - $second);
```

不过，为了清晰起见，我们在大多数例子里省略 **my**。只是假设任何这样的变量都事先定义过了。

```
accept
    accept SOCKET, PROTOCKET
```

这个函数用于那些希望监听来自客户端套接字连接的服务器进程。**PROTOCKET** 必须是一个已经通过 **socket** 操作符打开，并且与服务器的一个网络地址或者 **INADDR_ANY** 绑定的文件句柄。它的执行会延迟到连接发生的时候，在该点 **SOCKET** 文件句柄被打开并且附着到新建立的连接上。最初的 **PROTOCKET** 保持不变；它（**PROTOCKET**）唯一的用途就是克隆成一个真正的套接字。如果调用成功，该函数返回连接上的地址，否则返回假。比如：

```
unless ($peer = accept(SOCK, PROTOCK)) {
    die "Can't accept a connection: $!\n";
}
```

在那些支持 **exec** 时关闭（**close-on-exec**）标志的系统上，将为新打开的文件描述符设置该标志，和 **\$^F (\$SYSTEM_FD_MAX)** 的值决定的一样。

参阅 **accept(2)**。又见第十六章，进程间通讯，里的“套接字”一节里的例子。

```
alarm
    alarm EXPR
    alarm
```

这个函数在 **EXPR** 秒后给当前进程发送一个 **SIGALRM** 信号。

每次只能有一个定时器处于活跃状态。每次调用都关闭前面一个定时器，并且如果你给的 **EXPR** 是 **0**，那么就能取消所有前面设置的定时器而又不会启动一个新的定时器。它的返回值是前面一个定时器里剩下的时间的数量。

```
print "Answer me within one minute, or die: ";
alarm(60);          # 一分钟终止程序
$answer = ;
$timeleft = alarm(0); # 清除警报
print "You had $timeleft seconds remaining\n";
```

把 **alarm** 和 **sleep** 混合在一起使用通常是一个错误，因为许多系统使用 **alarm(2)** 系统调用机制实现 **sleep(3)**。在老一些的系统上，流逝的时间可能比你声明的少上最多一秒钟，这是由于秒计数的方式造成的。另外，一台繁忙的系统可能无法立即运行你的进程。参阅第十六章获取关于信号处理的信息。

如果你需要比一秒的颗粒度更细的间隔，那么你可能要使用 **syscall** 函数访问 **setitimer(2)**，前提是你的系统支持这个系统调用。**CPAN** 模块，**Timer::HiRes**，也提供了用于这个目的的函数。

```
atan2
    atan2 Y, X
```

这个函数返回 **-pi** 和 **pi** 之间的 **Y/X** 的反正切值。一个获取 **pi** 的近似值的快速方法是：


```
$pi = atan2(1,1) * 4;
```

对于正切操作，你可能要用 `Math::Trig` 或者 `POSIX` 模块里的 `tan` 函数，或者只是使用类似的关系：

```
sub tan { sin($_[0]) / cos($_[0]) }
```

`bind`

```
bind SOCKET, NAME
```

这个函数把一个地址（一个名字）附着到一个由 `SOCKET` 文件句柄声明的已经打开的套接字上。如果它成功了，那么返回真，否则返回假。`NAME` 应该是该套接字使用的类型正确的打包地址。

```
use Socket;
$port_number = 80;          # 假装我们想成为一个 web 服务器
$sockaddr = sockaddr_in($port_number, INADDR_ANY);
bind SOCK, $sockaddr or die "Can't bind $port_number: $!\n";
```

参阅 `bind(2)`。又见第十六章的“套接字”一节里的例子。

`binmod`

```
binmod FILEHANDLE, DISCIPLINES
binmod FILEHANDLE
```

这个函数为 `FILEHANDLE` 安排具有 `DISCIPLINES` 参数声明的语意的属性。如果省略了 `DISCIPLINES`，那么给该文件句柄提供二进制（或者“`raw`”）语意。如果 `FILEHANDLE` 是一个表达式，那么将根据情况把其值当作文件句柄或者一个指向文件句柄的引用。

`binmode` 函数应该是在调用完 `open` 之后，但在对该文件句柄做任何 I/O 之前调用。重置一个文件句柄的该模式的唯一方法是重新打开该文件，因为各种不同的纪律（模式）会把一点一滴的数据各自存放在不同的缓冲区里。这个限制可能在将来会放松。

在古老的年代，`binmode` 主要是用于那些运行时间库区分文本和二进制文件的操作系统上。在那些系统上，`binmode` 的作用是关闭缺省的文本语意。不过，自从增加了 `Unicode` 的先进性之后，在所有系统上的所有程序都一定有一些可识别的区别，即使是在 `Unix` 和 `Mac` 系统上也如此。现在，只有一种二进制文件存在了（至少 `Perl` 是这么认为的），但是有许多种文本文件，但 `Perl` 也愿意用一种方法来处理这些文本文件。所以 `Perl` 对于 `Unicode` 文本只有一种内部格式，`UTF-8`。因为有许多不同类型的文本文件，所以文本文件在输入时常需要转换成 `UTF-8`，而在输出时常需要转换回某些传统的字符集，或者 `Unicode` 的一些其他表现形式。你可以用纪律（`DISCIPLINES`）告诉 `Perl` 如何精确地（或者不精确地）做这些转换。（注：更准确地说，你将能用纪律做这些事情，但是我们写本书的时候还在实现这些东西。）

比如，一个为 `":text"` 的纪律将告诉 `Perl` 做一般性的文本处理而不用告诉 `Perl` 是做具体的哪种文本处理。但象 `":utf8"` 和 `":latin1"` 这样的纪律就告诉 `Perl` 读和写哪种文本格式。另一方面，`":raw"` 纪律告诉 `Perl` 把它的肥爪从数据上拿开，不对数据做任何额外处理。有关纪律如何运转（或者将如何运转）的更多内容，请参考 `open` 函数。这个讨论的其余部分描述没有 `DISCIPLINES` 参数的时候，`binmode` 干的事情，也就是说，`binmode` 的历史含义，也就等效于：

```
binmode FILEHANDLE, ":raw";
```

除非你另外指明了，否则 `Perl` 将把你新打开的文件以文本模式读写。文本模式意味着 `\n`（换行/新行）将是你的内部的行终止符。所有系统都用 `\n` 做内部的行终止符，但是它真正代表的东西是因系统而异，因设备而异，甚至是因文件而异的，具体情况取决于你如何访问该文件。在这样的一种传统的系统里，（包括 `MS-DOS` 和 `VMS`），你的程序看做 `\n` 的东西可能不是物理上存储在磁盘上的东西。比如，该操作系统可能把在文本文件里存储成 `\cM\cJ` 序列，到你在程序里显示为 `\n` 这样的

东西，而当输出到文件中去的时候，又把你的程序里的 `\n` 转换回 `\cM\cJ` 序列。`binmode` 关闭在这些系统上的这种转换工作。

如果没有 `DISCIPLINES` 参数，`binmode` 在 Unix 或者 Mac OS 里没有什么作用，它们都用 `\n` 结束每一行，并且把它表现为单个字符。（不过，这个字符可能是不一样的：Unix 使用 `\xJ` 而老一些的 Mac 使用 `\c`。没关系。）

下面的例子演示了 Perl 脚本如何从一个 GIF 图象文件里读取数据并把它打印到标准输出上去。在有些系统上，如果不做 `binmode` 处理，那么它就会把显示数据篡改成与数据的物理存储形式不同的东西，所以你必须把下面程序里的两个句柄都准备成二进制模式。尽管你可以很容易地在 GIF 文件打开上使用 `":raw"` 纪律，想在一个象 `STDOUT` 这样的提前打开了的文件句柄上做这件事可不那么容易：

```
binmode STDOUT;
open(GIF, "vim-power.gif") or die "Can't open vim-power.gif: $!\n";
binmode GIF;
while (read(GIF, $buf, 1024)) {
    print STDOUT $buf;
}

bless
    bless REF, CLASSNAME
    bless REF
```

这个函数告诉 Perl 由引用 `REF` 指向的引用物现在是一个在 `CLASSNAME` 包里的对象了——如果没有声明 `CLASSNAME` 那么就是当前包。如果 `REF` 不是一个有效的引用，那么就会抛出一个例外。为方便起见，`bless` 返回该引用，因为它通常是一个构造器子过程里的最后一个函数。比如：

```
$pet = Beast->new(TYPE => "cougar", NAME => "Clyde");

# 然后在 Beast.pm:
sub new {
    my $class = shift;
    my %attrs = @_;
    my $self = { %attrs };
    return bless($self, $class);
}
```

通常你应该把对象赐福到混合大小写的 `CLASSNAME` 里。名字空间里所有字母都小写的名字是 Perl 保留着在内部用做 Perl 用法（编译器指示器）的。内建的类型（比如“`SCALAR`”，“`ARRAY`”，“`HASH`”，等等，更不用说所有类的基类，“`UNIVERSAL`”）都是所有字母大写，因此你应该避免这样的包名字。

一定要确保 `CLASSNAME` 不是假值；把引用赐福到假包里目前还不支持，并且可能导致不可预见的后果。

Perl 里没有 `curse`（诅咒）操作符并不是臭虫。（不过我们有 `sin`（罪恶）操作符。）参阅第十二章，对象，看看更多有关给对象赐福的东西。

```
caller
    caller EXPR
    caller
```

这个函数返回关于当前子过程调用等方面的堆栈信息。如果没有参数，它返回包名，文件名，和调用当前子过程的程序的行号：

```
($package, $filename, $line) = caller;
```

下面是一个极为挑剔的函数的例子，它利用了第二章，集腋成裘，里描述特殊记号 **PACKAGE** 和 **__FILE__**：

```
sub careful {
    my ($package, $filename) = caller;
    unless ($package eq __PACKAGE__ && $filename eq __FILE__) {
        die "You weren't supposed to call me, $package!\n";
    }
    print "called me safely\n";
}

sub safecall {
    careful();
}
```

如果带参数调用，**caller** 把 **EXPR** 算做从当前的堆栈位置向回退的帧数。比如，参数 **0** 意思是当前堆栈帧，**1** 意思是该调用者，**2** 意思是调用者的调用者，以此类推。该函数还汇报下面显示的附加信息：

```
$i = 0;
while (($package, $filename, $line, $subroutine,
    $hasargs, $wantarray, $evaltext, %is_require,
    $hints, $bitmask) = caller($i++)) {
    ...
}
```

如果该堆栈帧是一个子过程调用，那么如果该子过程有自己的 **@_** 数组（不是那个它从调用者那里借过来的）那么 **\$hasargs** 就是真。否则，如果该帧不是一次子过程调用，而是一个 **eval**，那么 **\$subroutine** 可能是 **"(eval)"**。如果是这样，那么设置附加的元素 **\$evaltext** 和 **%is_require**：如果该帧是由 **require** 或者 **use** 语句创建的那么 **%is_require** 为真，而 **\$evaltext** 包含 **eval EXPR** 语句的文本。特别是，对于一个 **eval BLOCK** 语句，**\$filename** 是 **"(eval)"**，但 **\$evaltext** 是未定义。（还要注意每个 **use** 语句都在一个 **eval EXPR** 帧里创建一个 **require** 帧。）**\$hints** 和 **\$bitmask** 是内部值；除非你是仙境成员，否则不要动它们。

还有一点更深一层的东西：**caller** 还把数组 **@DB::args** 设置为传递到给定堆栈帧的参数——不过只有在它是被从 **DB** 包里面调用的才做这个工作。参阅第二十章，**Perl** 调试器。

```
chdir
    chdir EXPR
chdir
```

如果可能，这个函数改变当前进程的工作目录到 **EXPR**。如果省略 **EXPR**，则使用调用者的家目录。这个函数成功时返回真，否则返回假。

```
chdir "$prefix/lib" or die "Can't cd to $prefix/lib: $!\n";
```

又见 **Cwd** 模块，在第三十二章，标准模块，里描述，它可以自动跟踪你的当前目录。

```
chmod
    chmod LIST
```

这个函数改变一系列文件的权限。列表的第一个元素必须是一个数字模式，就象在 **chmod(2)** 系统调用里的一样。该函数返回成功改变了的文件的数目。比如：


```
$cnt = chmod 0755, 'file1', 'file2';
```

会把 `$cnt` 设置为 0, 1, 或 2, 具体是多少取决于改变的文件的数目。成功是通过没有错误来表示的, 而不是通过实际的修改的数目, 因为一个文件可能会拥有和操作之前相同的模式。一个错误可能意味着你缺乏修改文件模式的足够的权限, 你可能既不是文件的所有者也不是超级用户。检查 `$!` 看看失败的实际原因是什么。

下面是更多的一些典型用法:

```
chmod(0755, @executables) == @executables
or die "couldn't chmod some of @executables: $!";
```

如果你想知道是哪个文件不允许这样的修改, 使用象下面这样的代码:

```
@cannot = grep {not chmod 0755, $_} 'file1', 'file2', 'file3';
die "$0: could no chmod @cannot\n" if @cannot;
```

这个惯用法使用 `grep` 函数选择列表里那些 `chmod` 函数对之操作失败的元素。

如果使用非文本模式数据, 那么你可能需要用 `oct` 函数把一个八进制字符串转换成一个数字。这就是为什么 `Perl` 不会因为一个字串有一个前导 "0" 而就假定它包含一个八进制数字。

```
$DEF_MODE = 0644;          # 这里不能用引号!
PROMPT: {
    print "New mode? ";
    $strmode = ;
    exit unless defined $strmode;      # 测试 eof
    if ($strmode =~ /\^s*$/) {          # 测试空白行
        $mode = $DEF_MODE;
    }
    elsif ($strmode !~ /\^d+$/) {
        print "Want numeric mode, no $strmode\n";
        redo PROMPT;
    }
    else {
        $mode = oct($strmode);          # 把 "755" 转换成 0755
    }
    chmod $mode, @files;
}
```

这个函数与数字模式一起使用的时候很象 `Unix chmod(2)` 系统调用。如果你需要象 `chmod(1)` 命令提供的那样的符号接口, 你可以看看 `CPAN` 上的 `File::chmod` 模块。

你还可以从 `Fcntl` 模块里输入 `S_I*` 符号常量:

```
use Fcntl ':mode';
chmod S_IRWXU | S_IRGRP | S_IXGRP | S_IXOTH , @executalbes;
```

有些人认为上面这个比 `0755` 的可读性更好。你可以自己试试。

```
chomp
chomp VARIABLE
chomp LIST
chomp
```

这个函数通常把一个变量里包含的字串尾部的换行符删除。它使 `chop` 函数 (下面描述) 的一个略微安全些的版本, 因为它对没有换行符的字串没有影响。更准确地说, 它根据 `$/` 的当前值删除字串终

止符，而不只是最后一个字符。

和 `chop` 不同，`chomp` 返回删除的字符数量。如果 `$/` 是 `""`（处于段落模式下），`chomp` 从选出的字符串里删除所有结尾的换行符。你不能 `chomp` 一个文本常量，只能处理变量。

比如：

```
while () {
    chomp;          # 避免在最后一个字段里出现 \n
    @array = split /:/;
    ...
}
```

在版本 5.6 里，`chomp` 的含义略微改变了一些，我们可以用输入纪律覆盖 `$/` 变量的值，并且把字符串打上它们应该如何砍断的标记。这样做的优点是输入的纪律可以识别多于一种的行终止符（比如 Unicode 段落和行分隔符），而且还能安全的 `chomp` 掉终止当前行的东西。

```
chop
    chop VARIABLE
    chop LIST
    chop
```

这个函数把一个字符串变量的最后一个字符砍掉，并且返回砍掉的字符。`chop` 主要用于从一条输入记录的尾部删除换行符，并且比使用一个子过程更高效。如果这就是你在做的事情，那么用 `chomp` 更安全一些，因为 `chop` 不管字符串里的是什么都会剪短它，而 `chomp` 则更有选择性一些。

你不能 `chop` 文本常量，只能 `chop` 一个变量。

如果你 `chop` 一列 `LIST` 变量，那么列表中的每个字符串都被剪短：

```
@lines = `cat myfile`;
chop @lines;
```

你可以 `chop` 任何是左值的东西，包括一个赋值：

```
chop($cwd = `pwd`);
chop($answer = );
```

上面的和下面这句是不同的：

```
$answer = chop($tmp = ); # 错误
```

它把一个新行放到了 `$answer` 里，因为 `chop` 返回砍掉的字符，而不是剩下的字符串（在 `$tmp` 里）。获取我们这里想要的结果的一个方法是用 `substr`：

```
$answer = substr , 0, -1;
```

不过我们更经常的是写：

```
chop($answer = );
```

在最常见的情况下，`chop` 可以用 `substr` 来表示：

```
$last_char = chop($var);
$last_char = substr($var, -1, 1, "" ); # 一样的东西
```

一旦你理解了这个等效原理，你就可以用它做更大的砍削。要砍掉多于一个字符，那么把 `substr` 当作左值使用，赋予一个空字符串。下面的代码删除 `$caravan` 的最后五个字符：

```
substr($caravan, -5) = " ";
```

这里的负数脚本令 **substr** 从字符串的尾部而不是头部开始计算。如果你想保存这样删除的字符，你可以使用四个参数形式的 **substr**，创建一个五倍的砍削：

```
$tail = substr($caravan, -5, 5, "");
```

chown

```
chown LIST
```

这个函数修改一列文件的所有者和组。列表的头两个元素必须是数字 **UID** 和 **GID**，顺序如前。其中任何一个位置的 **-1** 的值在大多数系统上解释为把该值保留不变。该函数返回成功改变的文件的数目。比如：

```
chown($uidnum, $gidnum, 'file1', 'file2') == 2
or die "can't chown file1 or file2: $!";
```

会把 **\$cnt** 设置为 **0**，**1**，或 **2**，具体是何值取决于究竟有几个文件被修改（是以操作成功为准，而不是以修改完成以后所有者是否不同为准）。下面是一个更典型的应用：

```
chown($uidnum, $gidnum, @filename) == @filenames
or die "can't chown @filenames: $!";
```

下面是一个接受一个用户名，为你找出该用户和组 **ID**，然后做 **chown** 的子过程：

```
sub chown_by_name {
    my($user, @files) = @_;
    chown((getpwnam($user))[2,3], @files) == @files
        or die "can't chown @files: $!";
}

chown_by_name("fred", glob("*.c"));
```

不过，你可能不想象前一个函数那样修改组，因为 **/etc/passwd** 文件把每个用户和一个组关联起来，即使根据 **/etc/group** 而言，该用户可以是许多从属组的成员也如此。一个变化的方法是传递 **-1** 做为 **GID**，它的含义是不改变该文件的组。如果你把 **-1** 当作 **UID** 同时传递一个有效的 **GID**，那么你可以设置组而不修改所有者。

大多数系统上不会允许你修改文件的所有权，除非你是超级用户，尽管你可以把组改成你所在的任何从属组。在不安全的系统里，这样的限制可以放松，但是这么做不是一个可移植的假设。在 **POSIX** 系统上，你可以用类似下面的方法检测应用的是哪种规则：

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
# 只是测试我们是超级用户还是一个打开许可的系统
if ($? == 0 || !sysconf(_PC_CHOWN_RESTRICTED) ) {
    chown($uidnum, -1, $filename)
        or die "can't chown $filename to $uidnum: $!";
}
```

chr

```
chr NUMBER
chr
```

这个函数返回 **NUMBER** 在字符集中代表的字符。比如，**chr(65)** 在 **ASCII** 或 **Unicode** 中都是“A”，而 **chr(0x263a)** 是一个 **Unicode** 的笑脸。如果想反向的 **chr** 功能，用 **ord**。

如果你喜欢用名字来声明你的字符，而不是用数字（比如，"`\N{WHITE SMILING FACE}`" 就是 Unicode 笑脸），参阅第三十一章，实用模块的 `charnings`。

```
chroot
    chroot FILENAME
    chroot
```

如果成功，`FILENAME` 成为当前进程的新的根目录——用 “/” 开头的路径名的起点。这个目录是跨 `exec` 调用继承的，以及被所有 `chroot` 调用后 `fork` 出来的子进程继承。我们没有办法撤消一次 `chroot`。出于安全原因，只有超级用户可以使用这个函数。下面是一些许多 FTP 服务器的做法的近似模拟：

```
chroot((getpwnam('ftp'))[7])
    or die "Can't do anonymous ftp: $!\n";
```

这个函数在非 Unix 系统里可能不能运行。参阅 `chroot(2)`。

```
close
    close FILEHANDLE
    close
```

这个函数关闭与 `FILEHANDLE` 关联的文件，套接字，或者管道。（如果省略参数，那么它关闭当前选定的文件句柄。）如果关闭成功它返回真，否则返回假。如果你准备马上就对 `FILEHANDLE` 做另外一次 `open`，那么你用不着关闭它，因为下一次 `open` 会替你关闭它。（参阅 `open`）不过，对输入文件的明确的关闭重置行计数器（`$.`），而 `open` 做的隐含关闭不会做这件事情。

`FILEHANDLE` 可以是一个表达式，它的值可以用做一个间接的文件句柄（要么是一个真实文件句柄名字，要么是一个指向任何可以解释为一个文件句柄对象的引用。）

如果该文件句柄来自一个管道打开，如果任何下层系统调用失败或者在管道另一端的程序退出值非零，那么 `close` 将返回假。对于后面一种情况，`close` 强制 `$(OS_ERROR)` 为零。所以如果一个在管道上的 `close` 返回了一个非零值，那么可以检查 `$(` 的值判断问题来自管道本身（非零值）还是来自管道对端的程序（零值）。不管哪种情况，`$(CHILD_ERROR)` 都包含与管道另一端的命令相关联的等待状态值（参阅在 `system` 里它的解释）。比如：

```
open(OUTPUT, '| sort -rn | lpr -p')    # 输出给 sort 和 lpr
    or die "Can't start sortlpr pipe: $!";
print OUTPUT @lines;                  # 把内容打印到输出
close OUTPUT                          # 等待 sort 结束
    or warn $! ? "Syserr closing sortlpr pipe: $!"
      : "Wait status $? from sortlpr pipe";
```

用 `dup(2)`（复制）管道的方法制作的文件句柄被当作一个普通的文件句柄，所以在那个文件句柄上的 `close` 不会等待子进程。不过你关闭最初的文件句柄的时候就必须等待子进程。比如：

```
open(NETSTAT, "netstat -rn |")
    or die "can't run netstat: $!";
open(STDIN, "<&NETSTAT")
    or die "can't dup to stdin: $!";
```

如果你关闭上面的 `STDIN`，那么不会有等待发生，但是如果你关闭 `NETSTAT`，那么就有。

如果你自己打理了一个已退出的管道子进程，那么关闭就会失败。如果你有一个自己的 `$(SIG{CHLD})` 句柄，在管道子进程退出的时候触发，或者你故意在从 `open` 调用返回的进程 ID 上调用 `waitpid`，那么就有可能发生关闭失败的情况。

```
closedir
    closedir DIRHANDLE
```

该函数关闭一个 **opendir** 打开的目录并且返回该操作的成功。参阅 **opendir** 里的例子。
DIRHANDLE 可以是一个表达式，它的值可以用做一个间接目录句柄，通常是真实目录句柄的名字。

```
connect
    connect SOCKET, NAME
```

这个函数初始化一个与另外一个进程的连接，该进程正在一个 **accept** 上等待。如果成功该函数返回真，否则返回假。**NAME** 应该是一个类型正确的该套接字打包了的网络地址。比如，假设 **SOCK** 是一个前面已经创建了的套接字：

```
use Socket;

my ($remote, $port) = ("www.perl.com", 80);
my $destaddr = sockaddr_in($port, inet_aton($remote));
connect SOCK, $destaddr
    or die "Can't connect to $remote at port $port: $!";
```

为了断开一个套接字，可以使用 **close** 或者 **shutdown** 之一。参阅在第十六章里的“套接字”一节里的例子，参阅 **connect(2)**。

```
cos
    cos EXPR
    cos
```

这个函数返回 **EXPR** 的余弦值（用弧度表示）。比如，下面的脚本将打印一个用角度计量的余弦表：

#这是个实现角度到弧度的偷懒的方法

```
$pi = atan2(1,1) * 4;
$piover180 = $pi/180;

# 打印表格。
for ($deg = 0; $deg <= 90; $deg++) {
    printf "%3d %7.5f\n", $deg, cos($deg * $piover180);
}
```

如果需要反余弦操作，你可以使用来自 **Math::Trig** 或者 **POSIX** 模块的 **acos()** 函数，或者使用下面的关系：

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

```
crypt
    crypt PLAINTEXT, SALT
```

这个函数用和 **crypt(3)** 完全一样的方法计算一个字串的单向散列。这个功能对于检查口令文件中是否有极烂的口令很有用，（注：只有那些具有诚实的意图的人才可以做这些事情。）尽管你真正想干的事情是防止人们在一开始就使用糟糕的口令。

crypt 是有意做成一个单向函数的，和打破鸡蛋炸煎鸡蛋一个道理。目前没有（已知）的办法解密一个加了密的口，只能用穷举法的强行猜测方法。

当验证一个现有的加密字串的时候，你应该用该加密的文本做 **SALT**（象 **crypt(\$plain, \$crypteq \$crypteq)**）。这样就令你的代码和标准 **crypt** 兼容，并且还可以有更多奇特的实现。

当你选择一个新的 **SALT** 的时候，你最少需要创建一个两字符长的随机字符串，它的字符来自集合 `[/0-9A-Za-z]`（类似 `join ' ', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]`）。更早的 **crypt** 的实现只需要 **SALT** 的头两个字符，不过我们现在认为那些只给出头两个字符的代码是不能移植的。参阅你本地的 **crypt(3)** 手册页获取更多有趣的细节。

下面是一个例子，它可以证实哪个运行此程序的人知道他们自己的口令：

```
$pwd = (getpwuid ($<))[1];    # 假设我们在 Unix 上。

system "stty -echo";          # 或者看看 CPAN 上的 Term::ReadKey
print "Password: ";
chomp($word = );
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Sorry...\n";
} else {
    print "ok\n";
}
```

当然，向任何问你索要口令的东西键入你的口令都是不明智的举动。

影子口令文件比传统的口令文件稍微更安全些，并且如果你想访问它们的话那么你就必须是超级用户。因为很少的几个程序才能运行在这样强大的权限下，所以你可能需要让程序维护它自己独立的认证系统，方法是把 **crypt** 字符串保存在与 `/etc/passwd` 和 `/etc/shadow` 不同的地方。

crypt 函数不适合做大量数据的加密，更重要的是因为你不能取回数据。查看你喜爱的 CPAN 的 `by-module/Crypt` 和 `by-module/PGP` 目录寻找一些可能有用的模块。

```
dbmclose
    dbmclose HASH
```

这个函数打破在一个 **DBM**（数据库管理）文件和一个散列之间的绑定。**dbmclose** 实际上只是一个带有合适参数的 **untie** 的调用，提供它只是为了保持与古老的 Perl 版本的向下兼容。

```
dbmopen
    dbmopen HASH, DBNAME, MODE
```

这个函数把一个 **DBM** 文件和一个散列（也就是说，一个关联性数组）捆绑在一起。（**DBM** 的意思是数据库管理，由一个 C 库过程组成，它允许你通过散列算法随机地访问记录。）**HASH** 是散列的名字（包括 `%`）。**DBNAME** 是数据库的名字（没有 `.dir` 或者 `.pag` 扩展）。如果该数据库并不存在并且声明了一个有效的 **MODE**，那么该数据库就带着声明的 **MODE** 的保护（模式）创建，就象 **umask** 修改的那样。为了避免数据库不存在的时候的创建动作，你可以声明 **MODE** 为 **undef**，并且如果该函数无法找到一个现有数据库，那么将返回假。在 **dbmopen** 之前赋予散列的值是不能访问的。

dbmopen 函数实际上只是一个带有合适参数的 **tie** 的调用，提供它只是为了保持与早期的 Perl 版本的向下兼容。你可以通过直接使用 **tie** 接口或者通过在调用 **dbmopen** 之前装载合适的模块来控制自己使用哪个 **DBM** 库。下面是一个例子，它可以在一些 **DB_File** 的版本和你的 Netscape 浏览器版本类似的系统上运行：

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.dat", undef_
    or die "Can't open netscape history file: $!";
```

```

while (($url, $when) = each %NS_Hist) {
    next unless defined($when);
    chop($url, $when);      # 删除尾部的空字节
    printf "Visited %s at %s.\n", $url,
        scalar(localtime(unpack("V", $when)));
}

```

如果你没有 **DBM** 文件的写权限，那么你就只能读取散列变量，而不能设置它们。如果你想知道自己是否能写，你可以用文件测试，比如 `-w $file`，或者尝试在一个 `eval{}` 里面设置一个假散列记录，而 `eval{}` 会捕获例外。

如果在很大的 **DBM** 文件上使用，**keys** 和 **values** 这样的函数可能会返回巨大的列表值。你可能会愿意使用 **each** 函数叙述特大 **DBM** 文件，这样就不会一次把所有东西都装到内存里。

与 **DBM** 文件绑定的散列和你使用的 **DBM** 包有同样的局限性，包括你可以放到一个桶里的数据数量限制。如果你坚持使用短键字和数值，那么这一点很难成为问题。又见第三十二章的 **DB_File** 模块。

你应该记在心里的另外一件事情是许多现有的 **DBM** 数据库包含空结尾的键字和数值，因为它们是以 **C** 程序的概念考虑问题的。**Netscape** 历史文件和老的 **sendmail** 别名文件就是例子。只需要在取出数据的时候用 `"$key\0"`，然后从数值里删除空。

```

$alias = $aliases{"postmaster\0"};
chop $alias;    # 删除空

```

目前没有内建的方法锁住一个一般性的 **DBM** 文件。有些人认为这是一只臭虫。**GDBM_File** 模块的确实试图以整个文件为粒度提供锁定的机制。如果你有问题，那么最好还是使用一个独立的锁文件。

```

defined
    defined EXPR
defined

```

这个函数返回一个布尔值，表明 **EXPR** 是否是一个已经定义的数值。你处理的大多数数据都是已经定义的，但是一个标量如果不包含有效的字串，数字，或者引用值，那么就说它是未定义的数值，或者简称 **undef**。把一个标量变量初始化为一个特定的值将定义它，并且将一直保持已定义状态直到你给它赋一个未定义值或者在该变量上明确调用 **undef** 函数。

许多操作在异常条件下都返回 **undef**，比如文件末尾，使用一个未初始化的变量的数值，一个操作系统错误等等。因为 **undef** 只是假值的一种，所以简单布尔测试并不能分别 **undef**，数字零，和一个单字符串字串，“0”——它们都等于假。当你正在进行的操作可能返回一个真正的空字符串时，**defined** 函数可以让你可以区分未定义的空字符串和定义了的空字符串。

下面是一个测试一个来自某散列的标量值的片段：

```

print if defined $switch{D};

```

当象上面这样在散列元素上使用的时候，**defined** 只告诉你该数值是否定义，而不会告诉你该键字在散列里是否有记录。有可能出现这种情况：有一个键字，其值是未定义的；但键字本身仍然存在。使用 **exists** 判断某个散列键字是否存在。

在下面一个例子里，我们利用了有些操作在用光你的数据之后会返回未定义数值的传统：

```

print "$val\n" while defined($val = pop(@ary));

```

而在下面这个例子里，我们使用用于检索系统用户信息的 **getpwent** 函数获取同样的信息。

```

setpwent();
while(defined($name = getpwent())) {
    print "<<$naem>>\n";
}
endpwent();

```

下面的例子从那些可能返回合法假值的系统调用中获取错误返回:

```

die "Can't readlink $sym: $!"
unless defined($value = readlink $sym);

```

你还可以用 **defined** 检查一个子过程是否已经被定义。这样就有可能避免把一个不存在的子过程给破坏掉（或者是已经声明但尚未定义的子过程。）:

```

indir("funcname", @arglist);
sub indir {
    my $subname = shift;
    no restrict 'refs';          #这样我们就可以使用间接地使用 subname
    if (defined &$subname) {
        &$subname(@_);          # 或者 $subname->(@_);
    }
    else {
        warn "Ignoring call to invalid function $subname";
    }
}

```

在聚集（散列和数组）上使用 **defined** 是不允许的。（以前这么做是报告该聚集的内存是否已经分配。）你可以使用简单的布尔测试来检查该数组或散列是否含有元素:

```

if (@an_array) { print "has array elements\n" }
if (%a_hash) { print "has hash members\n" }

```

又见 **undef** 和 **exists**。

```

delete
    delete EXPR

```

这个函数从指定散列或者数组删除一个元素（或者一段元素）。（如果你想删除一个文件请参阅 **unlink**。）被删除的元素会按照声明它们的顺序返回，不过对捆绑的变量（比如 **DBM** 文件）不能保证这一点。在删除操作之后，对任何已经删除的键字或者索引调用 **exists** 函数都会返回假值。（相比之下，在 **undef** 函数之后，**exists** 函数仍然返回真值，因为 **undef** 函数只是撤消该元素的数值的定义，并没有删除该元素本身。）

从 **%ENV** 散列删除则修改环境。从一个与（可写的）**DBM** 文件绑定的散列删除记录将导致从 **DBM** 文件把该散列删除掉。

在过去的版本里，你只能从散列里删除记录，但是到了 **Perl 5.6**，你也可以从一个数组里删除东西。从一个数组删除记录导致在声明位置的元素完全回复到未初始化的状态，但是这样的删除并不弥合所造成的裂缝，因为那样做会导致所有随后的记录的位置的变化。你可以使用 **splice** 实现这样的目的。（不过，如果你删除了数组的最后一个元素，数组的尺寸将减一（或者更多，取决于倒数第二位的元素（如果存在的话）的位置。））

EXPR 可以是任意复杂，前提是其最终操作是一个散列或者一个数组查找:

```

# 设置一个散列数组的数组
$dungeon[$x][$y] = \%properties;

```

```
# 从散列中删除一个属性
delete $dungeon[$x][$y]{"OCCUPIED"};

# 一次从散列中删除三个属性
delete @{$dungeon[$x][$y]}{"OCCUPIED", "DAMP", "LIGHTED"};

# 从数组中把 %properties 的引用删除
delete $dungeon[$x][$y];
```

下面的本机例子低效地删除所有 %hash 中的值:

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

下面这个一样:

```
delete @hash{keys %hash};
```

但上面两个都比给散列赋予一个空列表或者解除它的定义来得慢:

```
%hash = ();      # 彻底清空 %hash
undef %hash;      # 忘记 %hash 曾经存在过。
```

类似的是数组的删除:

```
foreach $index (0 .. $#array) {
    delete $array[$index];
}
```

和:

```
delete @array[0 .. $#array];
```

都比下面的低效:

```
@array = ();      # 彻底清空 @array
undef @array;      # 忘记 @array 曾经存在过
```

```
die
    die LIST
    die
```

在 `eval` 之外, 这个函数把 `LIST` 里的数值连接起来打印到 `STDERR` 并且带着当前的 `$!` (C 库的 `errno` 变量) 的值退出。如果 `$!` 为 0, 它就带着 `$?>>8` 值 (这个值是从 `system`, `wait`, 管道 `close`, 或者 ``commend`` 退出的最后一个子进程的状态) 退出。如果 `$?>>8` 为 0, 那么该函数带着 255 退出。

在一个 `eval` 里面, 该函数把 `$@` 变量设置为将要生成的错误信息, 然后退出 `eval`, 而 `eval` 返回 `undef`。因此 `die` 函数可以用于抛出命名例外, 而这个命名例外可以在程序的更高层捕获。参阅本章稍后的 `eval`。

如果 `LIST` 是一个单对象引用, 那么就假设该对象是一个例外对象, 并且象在 `$@` 里的例外那样毫不修改地返回。

如果 `LIST` 为空并且 `$@` 已经包含一个字串值 (通常是来自前面一次 `eval`), 那么就在该字串后面附加 `"\t ...propagated"` 再次使用。这样做有益于传播 (抛出) 例外:

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

如果 **LIST** 为空并且 **\$@** 已经包含一个例外对象，则调用 **\$@->PROPAGATE** 方法来判断该例外应该如何传播。

如果 **LIST** 为空并且 **\$@** 也为空，那么使用字符串 “Died”。

如果 **LIST** 的最后一个值不是以换行符结尾（而且你不是在传递一个例外对象），则在该信息后面附加当前脚本文件名，行号，以及输入行数（如果有），然后再附加一个换行符。提示：有时候在你的信息上附加 **", stopped"** 会令它更有意义，因为后面会附加 **"at scriptname line 123"** 这样的东西。假设你正在运行一个名叫 **canasta** 的脚本；看看下面两种退出方法的区别：

```
die "/usr/games is no good";
die "/usr/games is no good, stopped";
```

它们分别输出：

```
/usr/games is no good at canasta line 123.
/usr/games is no good, stopped at canasta line 123.
```

如果你希望自己的错误信息报告文件名和行数，使用特殊标记 **_ FILE _** 和 **_ LINE _**：

```
die '"', __FILE__, '"', line ' ', __LINE__, '"', phooey on you!\n";
```

这句代码生成下面这样的东西：

```
"canasta", line 38, phooey on you!
```

再说另外一种风格——看看下面两个相等的例子：

```
die "Can't cd to spool: $!\n"    unless chdir '/usr/spool/news';

chdir '/usr/spool/news'    or die "Can't cd to spool: $!\n";
```

因为重要的部分是 **chdir**，所以通常会用第二种形式。

又见 **exit**，**warn**，**%SIG**，和 **Cary** 模块。

```
do (block)
    do BLOCK
```

do BLOCK 形式执行在 **BLOCK** 里的语句序列并且返回在该块里计算的最后一个表达式的值。如果用一个 **while** 或者 **until** 语句修饰词修饰，那么 **Perl** 在测试循环条件之前执行一次 **BLOCK**。（在其他语句上，循环修饰词先测试循环条件。）**do BLOCK** 本身并不算一个循环，所以循环控制语句 **next**，**last**，或 **redo** 不能用于离开或者重新运行该块。参阅第四章，语句和声明，里的 “光块” 一节获取绕开的办法。

```
do(file)
    do FILE
```

do FILE 形式把 **FILE** 的值当作一个文件名使用，并且把该文件的内容当作一个 **Perl** 脚本运行。它的主要用途是（或者最好是）从一个 **Perl** 子过程库中包含子过程，所以：

```
do 'stat.pl';
```

更象：


```
scalar eval `cat stat.pl`; # 在 Windows 里是 `type stat.pl`
```

只不过是 **do** 更高效更紧凑一些，它还跟踪当前文件名用于错误信息，扩展所有在 **@INC** 数组里的目录，并且如果找到该文件还会更新 **%INC**。（参阅第二十八章，特殊名字。）另外一个区别是用 **do FILE** 执行的代码看不到在闭合范围里的词法，而在 **eval FILE** 里的代码却可以看到。不过，它们都是在每次调用该文件的时候重新分析它——因为你可能不想在一个循环里用这个语句，除非文件名本身每循环一圈都改变一次。

如果 **do** 无法读取该文件，那么它返回 **undef** 并且把 **\$!** 设置为错误。如果 **do** 可以读取该文件但无法编译它，那么它返回 **undef** 并且在 **\$@** 里设置一个错误信息。如果该文件成功编译，那么 **do** 返回所计算的最后一个表达式的值。

库模块的包含（它们都有强制的 **.pm** 后缀）最好是用 **use** 和 **require** 操作符来干，它们也做错误检查，并且如果有问题地话它们抛出一个例外。它们还有其他好处：它们可以避免重复装载，有助于面向对象的编程，并且给编译器提供函数原形的提示。

但是 **do FILE** 在一些方面还是很有优势的，比如从程序配置文件中读取数据。可以用下面的方法做手工的错误检查：

```
# 读进 config 文件：先系统，后用户
for $file ("/usr/share/proggie/defaults.rc",
           "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@ if $@";
        warn "couldn't do $file: $!" unless defined $return;
        warn "couldn't run $file" unless $return;
    }
}
```

一个长时间运行的守护进程可以周期性地检查它的配置文件的时间戳记，如果该文件自上次读取之后改变过，那么该守护进程可以用 **do** 重新装载该文件。这种任务用 **do** 要比用 **require** 或者 **use** 更加干净。

```
do (subroutine)
do SUBROUTINE(LIST)
```

do SUBROUTINE(LIST) 是一个已经废弃了的子过程调用的方法。如果 **SUBROUTINE** 没有定义那么抛出一个例外。参阅第六章，子过程。

```
dump
dump LABEL
dump
```

这个函数导致一次立即的核心倾倒。它的主要用途是你可以用 **undump** 程序（还没提供）在你的程序开头刚刚完成所有变量初始化之后，把你倾倒的核心转换成一个可执行文件。而当这个新的二进制文件执行的时候，它将以执行一个 **goto LABEL**（受所有 **goto** 自己的限制）开始。把它想象成一个在重生和核心倾倒之间的一个 **goto**。如果省略了 **LABEL**，那么程序从顶部重新开始。警告：任何在倾倒时已经打开的文件在该程序重生时将不再打开，这有可能导致 **Perl** 这边的混乱。又见第十九章，命令行接口，里的 **-u** 命令行选项。

现在，这个函数基本上已经过时了，部分原因是想在一般情况下把核心文件转换成可执行文件是非常困难的，还有是因为各种生成可移植的字节码和可编译的 **C** 代码的编译器后端逐渐超越了它。

如果你试图通过使用 **dump** 来加快你的程序的速度，那么请先检查一下在第二十四章，普通实践，

里的关于效率的讨论，以及第十八章，编译，里的 **Perl** 本机代码生成器的讨论。你可能还应该考虑自动装载和自装载，它们至少能让你的程序看起来快一些。

Revision: r1.3 - 10 Jan 2006 - 03:39 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [PerlReference_Functions](#)

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](#)