

第二十九章，函数（E-N）

↓ 第二十九章，函数（E-N）

↓ 29.2 按照字母顺序排列的 Perl 函数

- ↓ 29.2.30 each
- ↓ 29.2.31 eof
- ↓ 29.2.32 eval
- ↓ 29.2.33 exec
- ↓ 29.2.35 exists
- ↓ 29.2.35 exit
- ↓ 29.2.36 exp
- ↓ 29.2.37. fcntl
- ↓ 29.2.38 fileno
- ↓ 29.2.39 flock
- ↓ 29.2.40 fork
- ↓ 29.2.41 format
- ↓ 29.2.42 formline
- ↓ 29.2.43 getc
- ↓ 29.2.44 getgrent
- ↓ 29.2.45 getgrgid
- ↓ 29.2.46 getgrnam
- ↓ 29.2.47 gethostbyaddr
- ↓ 29.2.48 gethostbyname
- ↓ 29.2.49 gethostent
- ↓ 29.2.50. getlogin
- ↓ 29.2.51 getnetbyaddr
- ↓ 29.2.52 getnetbyname
- ↓ 29.2.53. getnetent
- ↓ 29.2.54. getpeername
- ↓ 29.2.55. getpgrp
- ↓ 29.2.56 getppid
- ↓ 29.2.57. getpriority
- ↓ 29.2.58 getprotobyname
- ↓ 29.2.59 getprotobynumber
- ↓ 29.2.60 getprotoent
- ↓ 29.2.61 getpwent
- ↓ 29.2.62 getpwnam
- ↓ 29.2.63. getpwuid
- ↓ 29.2.64. getservbyname
- ↓ 29.2.65. getservbyport
- ↓ 29.2.66. getservent
- ↓ 29.2.67. getsockname
- ↓ 29.2.68. getsockopt
- ↓ 29.2.69 glob
- ↓ 29.2.70. gmtime
- ↓ 29.2.71. goto
- ↓ 29.2.72. grep
- ↓ 29.2.73. hex
- ↓ 29.2.74 import
- ↓ 29.2.75. index
- ↓ 29.2.76. int
- ↓ 29.2.77. ioctl
- ↓ 29.2.78. join
- ↓ 29.2.79 keys
- ↓ 29.2.80. kill
- ↓ 29.2.81. last
- ↓ 29.2.82. lc
- ↓ 29.2.83 lcfirst
- ↓ 29.2.84. length
- ↓ 29.2.85. link
- ↓ 29.2.86. listen
- ↓ 29.2.87. local
- ↓

- ↓ [29.2.88. localtime](#)
- ↓ [29.2.89. lock](#)
- ↓ [29.2.90. log](#)
- ↓ [29.2.91. lstat](#)
- ↓ [29.2.92. m//](#)
- ↓ [29.2.93. map](#)
- ↓ [29.2.94. mkdir](#)
- ↓ [29.2.95. msgctl](#)
- ↓ [29.2.96. msgget](#)
- ↓ [29.2.97. msgrcv](#)
- ↓ [29.2.98. msgsnd](#)
- ↓ [29.2.99. my](#)
- ↓ [29.2.100. new](#)
- ↓ [29.2.101. next](#)
- ↓ [29.2.102. no](#)

29.2 按照字母顺序排列的 Perl 函数

29.2.30 each

- each HASH

这个以一次一个键字/数值对的方式遍历一个散列。如果在列表环境里调用它，**each** 返回一个两个元素的列表，该列表包含散列中下一个元素的键字和数值，这样你就可以逐一遍历它们。如果在标量环境里调用，**each** 只是返回散列中下一个元素的键字。如果散列已经全部读取完了，那么返回一个空列表，如果你给这个空列表赋值，那么在标量环境中会生成一个假值。下面是典型的用法，使用预定义的 **%ENV** 散列：

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

在散列内部，它以一种看上去是随机的顺序维护它自己的记录。**each** 可以遍历这个序列是因为每个散列都记得上一次返回的是哪条记录。这个序列的实际的顺序可能在将来的 **Perl** 版本里会改变，但有一点可以保证，就是 **keys**（或者 **values**）函数在同一个（未修改）的散列上生成的顺序是一样的。

每个散列都有一个遍历器，由在该程序里所有的 **each**，**keys**，和 **values** 函数调用共享；该遍历器可以通过从散列里读取所有元素来重置，或者通过计算 **keys %hash** 或 **values %hash** 来重置。如果你在遍历散列的过程中删除了元素，那么后果还没有很好的定义：记录可能会被忽略也可能被重复。

又见 **keys**，**values**，和 **sort**。

29.2.31 eof

- eof FILEHANDLE
- eof()
- eof

如果下一次对 **FILEHANDLE** 的读取返回文件结束（end-of-file）或者是 **FILEHANDLE** 没有打开，那么这个函数将返回真。**FILEHANDLE** 可以是一个表达式，其值给出真正的文件句柄，也可以是一个指向一个文件句柄对象或者类似的东西的引用。一个没有参数的 **eof** 为最后一次文件读动作返回文件结束状态。一个带空圆括弧对 **()** 的 **eof()** 测试 **ARGV** 文件句柄（最常见的就是 **<>** 里的空文件

句柄)。因此，在一个 `while (<>)` 循环里，一个带圆括弧的 `eof()` 只是检测一组文件中的最后一个文件的文件结束。用 `eof`（没有圆括弧）在 `while (<>)` 循环里检查每个文件的文件结束。比如，下面的代码在最后一个文件的最后一行前面插入一个划线：

```
while (<>) {
    if (eof()) {
        print "-" x 30, "\n";
    }
    print;
}
```

而下面这个脚本为每个输入文件重置行计数：

```
# 为每个输入文件重置行计数
while (<>) {
    next if /^\\s*#/;    # 忽略注释
    print "$\\.\\t$\\_";
} continue {
    close ARGV if eof;    # 不是 eof()!
}
```

类似 `sed` 程序里的“\$”，`eof` 会显示行数范围。下面是一个打印从 `/parrern/` 到每个输入文件结尾之间的行的脚本：

```
while (<>) {
    print if /pattern/ .. eof;
}
```

这里，触发器操作符 (`..`) 为每一行进行模式匹配。在模式匹配上之前，该操作符返回假。当它最终匹配上的时候，该操作符开始返回真，导致行的打印输出。当 `eof` 操作符最终返回真（在被检查的文件的结尾），该触发器操作符被重置，并且开始为 `@ARGV` 里的下一个文件返回假。

警告：`eof` 函数读取一个字节然后用 `ungetc(3)` 把它退回输入流中，所以在交互的环境中没有什么用。实际上，有经验的 Perl 程序员很少使用 `eof`，因为各种输入操作符的行为 `while` 循环里已经很礼貌了。参阅第四章里的 `foreach` 的讨论。

29.2.32 eval

- `eval BLOCK`
- `eval EXPR`
- `eval`

`eval` 关键字在 Perl 里起两种不同的但相关的作用。这些目的是用两种形式的语法来表现的，`eval BLOCK` 和 `eval EXPR`。第一种形式捕获那些致命的运行时例外（错误），类似于 C++ 或 Java 里的“try 块”。第二种形式在运行时实时地编译和执行一小段代码，并且也和第一种形式一样捕获任何例外。但是第二种形式比第一种形式运行的速度慢很多，因为它每次都要分析该字串。另外，它也更通用。不管你在那里使用，`eval` 都是一个在 Perl 里做全部例外处理的好地方。

两种形式的 `eval` 所返回的值都是它计算的最后一个表达式的值，这一点和子过程一样。类似的，你可以用 `return` 操作符从 `eval` 的中间返回一个数值。提供返回值的表达式是在空，标量，或者列表环境中计算的，具体哪种环境是由 `eval` 本身所处的环境决定的。参阅 `wantarray` 获取如何判断计算环境的信息。

如果有一个可捕获的错误存在（包括任何由 `die` 操作符生成的），`eval` 返回 `undef` 并且把错误信息放到 `$@` 里。如果没有错误，Perl 保证把 `$@` 设置为空字串，所以你稍后可以很可靠地做错误检

查。一个简单的布尔测试就足够了：

```
eval { ... };          # 捕获运行时错误
if ($?) { ... }       # 处理错误
```

eval BLOCK 形式是在编译时做语法检查的，所以它的效率相当高。（熟悉慢速的 **eval EXPR** 形式的人们可能会被这个问题搞糊涂。）因为在 **BLOCK** 里的代码是和周围的代码同时编译的，所以这种形式的 **eval** 不能捕获语法错误。

eval EXPR 形式可以捕获语法错误是因为它在运行时分析代码。（如果分析失败，它象平常一样在 **\$@** 里放分析错误。）另外，它把 **EXPR** 的值当作一小段 **Perl** 程序执行。这段代码是在当前 **Perl** 程序的环境中执行的，这就意味着它可以从包围的范围里看到任何词汇闭合域，并且在 **eval** 完成之后，任何非局部变量设置仍然有效，就象子过程调用或者格式定义一样。**eval** 的代码是当作一个块看待的，所以任何在 **eval** 里定义的局部范围的变量都只能持续到 **eval** 结束。（参阅 **my** 和 **local**。）和任何块里的代码一样，最后的分号不是必须的。

下面是一个简单的 **Perl shell**。它提示用户输入任意 **Perl** 代码字符串，编译并执行该字符串，并且打印出发生的任何错误：

```
print "\nEnter some Perl code: ";

while () {
    eval;
    print $?;
    print "\nEnter some more Perl code: ";
}
```

下面是用 **Perl** 表达式做大批文件改名的一个 **rename** 程序：

```
#!/usr/bin/perl
# rename - change filenames
$op = shift;
for (@ARGV) {
    $was = $_;
    eval $op;
    die if $?;
    # 下一行调用内建函数，而不是同名的脚本
    rename($was, $_) unless $was eq $_;
}
```

你要这样用这个程序：

```
$rename 's/\.orig$//'          *.orig
$rename 'y/A-Z/a-z/ unless /^Make/'  *
$rename '$_ .= ".bad"'          *.f
```

因为 **eval** 捕获那些致命错误，所以它可以用来判断某种特性（比如 **fork** 和 **symlink**）是否实现之类的东西。

因为 **eval BLOCK** 是在编译时做语法检查的，所以任何语法错误都提前报告。因此，如果你的代码不会变化，并且 **eval EXPR** 和 **eval BLOCK** 都完全符合你的要求，那么 **BLOCK** 形式好一些。比如：

```
# 零除零不是致命错误
eval { $answer = $a / $b; }; warn $? if $?;
```

```
# 一样的东西，但是如果多次运行就没有那么高效了
eval '$answer = $a / $b';          warn $@ if $@;

# 一个编译时语法错误（不捕获）
eval { $answer = };              # 错

# 一个运行时语法错
eval '$answer =';                # 设置 $@
```

这里，在 **BLOCK** 里的代码必须是合法的 Perl 代码，这样它才能通过编译阶段。在 **EXPR** 里的代码直到运行时才检查，所以要到运行时它才导致错误的发生。

eval BLOCK 的块并不是循环，所以 **next**，**last**，或 **redo** 这样的循环控制语句并不能用于离开或者重启该块。

29.2.33 exec

- o **exec** PATHNAME LIST
- o **exec** LIST

exec 函数结束当前程序的运行并且执行一条外部命令并且决不返回！！！如果你希望在该命令退出之后恢复控制，那么你应该使用 **system**。**exec** 函数只有在该命令不存在以及该命令是直接执行而没有通过你的系统的命令行 **shell**（下面讨论）执行的时候才失败并返回假。

如果只有一个标量参数，那么 **exec** 检查该参数是否 **shell** 的元字符。如果找到元字符，那么它代表的所有参数都传递给系统的标准命令行解释器（在 **Unix** 里是 **/bin/sh**）。如果没有这样的元字符，那么该参数被分裂成单词然后直接执行，出于效率考虑，这样做绕开了所有 **shell** 处理的过荷。而且如果该程序没有退出，这样也给你更多错误恢复的控制。

如果在 **LIST** 里有多于一个参数，或者如果 **LIST** 是一个带有超过一个值的数组，那么就决不会使用系统的 **shell**。这样也绕开了 **shell** 对该命令的处理。在参数中是否出现元字符并不影响这个列表触发特性，这么做也是有安全考虑的程序的比较好的做法，因为它不会把自己暴露在潜在的 **shell** 逃逸之中。

下面的例子令当前运行的 Perl 程序用 **echo** 程序代替自身，然后它就打印出当前的参数列表：

```
exec 'echo', 'Your arguments are: ', @ARGV;
```

下面这个例子显示了你可以 **exec** 一个流水线，而不仅仅是一个程序：

```
exec "sort $outfile | uniq"
or die "Can't do sort/uniq: $!\n";
```

通常，**exec** 从不返回——就算它返回了，它也总是返回假，并且你应该检查 **\$_** 找出什么东西出错了。要注意的是，在老版本的 Perl 里，**exec**（和 **system**）并不刷新你的输出缓冲，所以你需要在一个或多个文件句柄上通过设置 **\$|** 打开命令缓冲功能以避免在 **exec** 的情况下丢失输出，或者在 **system** 的情况下打乱了输出顺序。在 Perl 5.6 里情况大致如此。

如果你让操作系统在一个现有的进程里运行一个新的程序（比如 Perl 的 **exec** 函数做的这样），你要告诉系统要执行的程序在哪里，但是你也告诉了这个新的程序（通过它的第一个参数）是什么程序执行了它。习惯上，你告诉它的名字只是该程序的位置的一个拷贝，但这么做不是必须的，因为在 C 语言的级别上，有两个独立的参数。如果这个名字不是拷贝，那么你就可能看到奇怪的结果：这个新程序认为自己是以一个和它所在的实际路径名完全不同的名字运行的。通常这样对那些满腹狐疑的程序来说没什么问题，但有些程序的确关心自己的名字，并且根据自己的名字的变化会有不同的性格。

比如，**vi** 编辑器会看看自己是作为“**vi**”还是作为“**view**”调用的。如果作为“**view**”，那么它就自动打开只读模式，就好像它是带着 **-R** 命令行选项调用的一样。

这个时候就是 **exec** 的可选 **PATHNAME** 参数起作用的地方了。从语意上来看，它放在间接对象的位置，就好像 **print** 和 **printf** 的文件句柄一样。因此，它并不需要在后面有一个对象，因为它实际上不是参数列表的一部分。（从某种意义上来说，**Perl** 与操作系统采取的方法正相反，它认为第一个参数是最重要的，并且如果它不同那么就让你修改路径名。）比如：

```
$editor = "/usr/bin/vi";
exec $editor "view", @files    # 触发只读模式
    or die "Couldn't execute $editor: $!\n";
```

和任何其他间接对象一样，你也可以用一个包含任意代码的块代替上面这个保存程序名的简单标量，这样就可以把前面这个例子简化为：

```
exec { "/usr/bin/vi" } "view" @files    # 触发只读模式
    or die "Couldn't execute $editor: $!\n";
```

如前所述，**exec** 把一个离散的参数列表当作一个它应该绕开 **shell** 处理的标志。不过，仍然有一个地方可能把你拌倒。**exec** 调用（以及 **system**）不能区别单个标量参数和一个只有一个元素的列表。

```
@args = ("echo surprise")    # 只有一个元素在列表里
exec @args                    # 仍然可能有 shell 逃逸
    or die "exec: $!";        # 因为 @args == 1
```

为了避免这种情况，你可以使用 **PATHNAME** 语法，明确地把第一个参数当路径名复制，这样就强制其他的参数解释成一个列表，即使实际上只有一个元素：

```
exec { $args[0] } @args    # 就算是只有一个元素的列表也安全了
    or die "can't exec @args: $!";
```

第一个没有花括弧的版本，运行 **echo** 程序，给它传递“**surprise**”做参数。第二个版本不是这样——它试图运行一个字面上叫 **echo surprise** 的程序，但找不到（我们希望如此），然后把 **!** 设置为一个非零值以表示失败。

因为 **exec** 函数通常是紧跟在 **fork** 之后调用的，所以它假定任何原先一个 **Perl** 进程终止的时候要发生的事情都被忽略。在 **exec** 的时候，**Perl** 不会调用你的 **END** 块，也不会调用与任何对象相关的 **DESTROY** 方法。否则，你的子进程结束的时候会做那些你准备在父进程里做的清理工作。（我们希望在现实生活中就是如此。）

因为把 **exec** 当作 **system** 用是一个非常普遍的错误，所以如果你带着流行的 **-w** 命令行开关运行，或者你用了 **use warnings qw(exec syntax)** 用法的时候，如果 **exec** 后面跟着的语句不是 **die**, **warn**, 或则 **exit**, 那么 **Perl** 就会警告你。如果你真的想在 **exec** 后面跟一些其他的语句，你可以使用下面两种风格之一以避免警告：

```
exec ('foo')    or print STDERR "couldn't exec foo: $!";
{ exec ('foo') };          print STDERR "couldn't exec foo: $!";
```

正如上面的第二行显示的那样，如果调用 **exec** 的时候是一个块里的最后一条语句，那么就可以免于警告。

又见 **system**。

29.2.35 exists

o exist EXPR

如果所声明的散列键字或者数组索引在它的散列或者数组中存在，那么这个函数返回真值。它不在乎对应的数值是真还是假，或者该值是否定义。

```
print "True\n"      if      $hash{$key};
print "Defined\n"   if defined $hash{$key};
print "Exists\n"    if exists $hash{$key};

print "True\n"      if      $array[$index];
print "Defined\n"   if defined $array[$index];
print "Exists\n"    if exists $array[$index];
```

一个元素只有定义后才为真，并且只有存在才能被定义，但反过来却不一定是真的。

EXPR 可以任意复杂，前提是它的最后的操作是一个散列键字或者索引查找：

```
if (exists $hash{A}{B}{$key} ) { ... }
```

尽管最后一个元素不会只是因为它的存在性已经经过测试而存在，中间的元素却会。因此 `$$hash{"A"}` 和 `$hash{"A"}->{"B"}` 都将真正存在。这个功能不是 `exists` 函数本身的；它发生在任何使用了箭头操作符的地方（明确地或隐含地）：

```
undef $ref;
if (exists $ref->{"Some key"}) { }
print $ref; # 打印 HASH(0x80d3d5c)
```

即使 "Some key" 元素没有突然存在，前面未定义的 `$ref` 变量也会突然变成持有一个匿名散列的变量。这是一个那种第一眼——甚至第二眼看上去都不是左值环境条件下的自动激活的一个有趣的例子。这种行为在将来的版本里可能会被修补。作为绕过的一种方法，你可以嵌套你的调用：

```
if ($ref and
    exists $ref->[$x] and
    exists $ref->[$x][$y] and
    exists $ref->[$x][$y] and
    exists $ref->[$x][$y]{$key} and
    exists $ref->[$x][$y]{$key}[2] ) { ... }
```

如果 EXPR 是子过程的名字，如果该子过程已经定义，那么 `exists` 函数将返回真，即使该子过程还没有定义也如此。下面的程序将打印 “Exists”：

```
sub flub;
print "Exists\n"      if exists &flub;
print "Defined\n"     if defined &flub;
```

在一个子过程名字上使用 `exists` 可以用于 AUTOLOAD 子过程，这个子过程可能需要知道某个包是否需要某个子过程的定义。该包可以通过声明一个想 `flub` 那样的 `sub` 存根来实现这个目的。

29.2.35 exit

- o exit EXPR
- o exit

这个函数把 EXPR 当作一个整数计算然后立即以该数值为最终的程序错误状态退出。如果省略了 EXPR，那么该函数以 0 状态退出（意思是“没有错误”）。下面是一个程序片段，它让用户通过敲入 `x` 或 `X` 退出程序：

```
$ans =
    exit if $ans =~ /^[Xx]/;
```

如果别人有任何机会可以捕获所发生的任何错误，那么你不应该用 `exit` 退出子过程。应该用 `die`，它可以用一个 `eval` 捕获。或者使用 `Carp` 模块的 `die` 的封装，比如 `croak` 或者 `confess`。

我们说 `exit` 函数立即退出，但这是一个赤裸裸的谎言。它尽可能快地退出，但是它首先调用任何已经定义了的 `END` 过程做退出时处理。这些过程无法退出 `exit`，尽管它们可以通过设置 `$?` 变量改变最终的退出值。同样，任何定义了 `DESTROY` 方法的类都将在程序真正退出前代表它的所有对象调用该方法。如果你确实需要忽略退出处理，那么你可以调用 `POSIX` 模块的 `_exit` 函数以避免所有 `END` 和析构器处理。而如果没有 `POSIX` 可用，你可以 `exec "/bin/false"` 或者类似的东西。

29.2.36 exp

- `exp EXPR`
 - `exp`

这个函数返回 `e` 的 `EXPR` 次幂。要获取 `e` 的值，用 `exp(1)` 好了。对于不同基数的通用指数运算，使用我们从 `FORTRAN` 偷来的 `**` 操作符：

```
use Math::Complex;
print -exp(1) ** (i * pi);    # 打印 1
```

29.2.37. fcntl

- `fcntl FILEHANDLE, FUNCTION, SCALAR`

这个函数调用你的操作系统的文件控制函数，就是那些 `fcntl(2)` 手册页里归档的东西。在你调用 `fcntl` 之前，你可能首先要说：

```
use Fcntl;
```

以装载正确的常量定义。

根据所用的不同的 `FUNCTION`，将对 `SCALAR` 进行读或者写。可以把一个指向 `SCALAR` 字符串值的指针作为真正 `fcntl` 调用的第三个参数传递。（如果 `SCALAR` 没有字符串值，但的确有一个数字值，那么该值将被直接传递，而不是传递一个指向字符串值的指针。）参阅 `Fcntl` 模块获取 `FUNCTION` 比较常见的可用数值描述。

如果在一个没有实现 `fcntl(2)` 的系统上使用 `fcntl` 函数，那么它会抛出一个错误。在那些实现了这个系统调用的系统上，你可以做诸如修改 `exec` 时关闭（`close-on-exec`）标志（如果你不想使用 `$^F`（`$SYSTEM_FD_MAX`）变量），修改非阻塞 I/O 标志，模拟 `lockf(3)` 函数，以及在 I/O 等待的时候安排接收 `SIGIO` 信号这样的事情。

下面是一个在系统级别上把一个叫 `REMOTE` 的文件句柄设置为非阻塞的例子。这样，如果从一个管道，套接字，或者串行线读取数据时，如果发现没有数据可读，就让任何输入操作马上返回，否则的话就会阻塞住。它还让那些通常会阻塞的写操作马上带着一个失败状态返回。（你也可以设置 `$!` 实现这些目的。）

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(REMOTE, F_GETFL, 0)
    or die "Can't get flags for the socket: $!\n";

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
```



```
or die "Can't set flags for teh socket: $!\n";
```

fcntl（以及 ioctl）的返回值如下：

系统调用返回	Perl 返回
-1	undef
0	字串“0 but true”
其他任何东西	那个数字

因此 Perl 成功时返回真，而失败时返回假，但是你还是可以很容易地判断操作系统返回的实际值：

```
$retval = fcntl { ... } || -1;
printf "fcntl actually returned %d\n", $retval;
```

在这里，即使是字串“0 but true”也打印出 0，这是因为 %d 格式的作用。这个字串在布尔环境里为真，但在数字环境里为假。（它还很愉快地免于平时对数字转换的检查发出的警告。）

29.2.38 fileno

- fileno FILEHANDLE

这个函数返回在一个文件句柄下面的文件描述符。如果该文件句柄没有 open，那么 fileno 返回 undef。文件描述符是一个很小的，非负整数，比如 0 或 1，分别对应 STDIN 和 STDOUT，后者是符号。糟糕的是，操作系统可不认得你这些酷酷的符号。它只会用这样的小小的文件数字思维来打开文件，并且尽管 Perl 通常会自动为你做转换，但是偶尔你还是需要知道实际的文件描述符。

因此，举例来说，fileno 函数对于为 select 构造位图以及在实现了 syscall(2) 的情况下传递某些晦涩的系统调用来说是非常有用的。它还可以用于检查 open 函数给你的文件描述符是不是你想要的那个，以及判断两个文件句柄是否在使用同一个文件描述符。

```
if (fileno(THIS) == fileno(THAT) ) {
    print "THIS and THAT are dups\n";
}
```

如果 FILEHANDLE 是一个表达式，那么该值就会被当作一个间接的文件句柄，通常是它的名字或者一个指向某些构成一个文件句柄对象的引用。

一个警告：在程序的整个生命期里都不要依赖 Perl 文件句柄和数字文件描述符之间的关联关系。如果一个文件关闭以后重新打开，那么文件描述符可能改变。Perl 在保证某些文件描述符不会因为对它们的 open 失败而丢失的时候碰到了一些麻烦，它现在只能对那些不超过当前特殊变量 $\F （\$SYSTEM_FD_MAX）的当前值（缺省是 2）的文件描述符保证这一点。尽管文件句柄 STDIN，STDOUT，和 STDERR 从文件描述符 0，1，和 2 开始（Unix 标准传统），但如果你非常随意地打开和关闭它们的话，那么它们都有可能改变。只要你总是在关闭以后马上重新打开，那么你在 0，1，和 2 上不会碰到麻烦。在 Unix 系统上，基本规则是先用最小的描述符，而那个会是你刚刚关闭的那个。

29.2.39 flock

- flock FILEHANDLE OPERATION

flock 函数是 Perl 的可移植的文件锁定的接口，尽管它只是锁住整个文件，而不是记录。该函数会与 FILEHANDLE 关联的文件锁住，成功时返回真，失败时返回假。为了避免可能的数据丢失现象，Perl 在锁住或者解锁文件之前刷新 FILEHANDLE。Perl 实现它的 flock 的方法可能是 flock(2)，fcntl(2)，lockf(3)，或者其他的什么平台相关的锁机制，但是如果所有这些都没有，那么调用 flock

将会抛出一个例外。参阅第十六章的“文件锁定”一节。

OPERATION 是 LOCK_SH, LOCK_EX, 或者 LOCK_UN 之一，可能是与 LOCK_NB 或 (OR) 的。这些常量通常的值是 1, 2, 8, 和 4，但是如果你是从 Fcntl 里分别或者用 :flock 标签成组地输入它们的，那么你可以使用符号名字。

LOCK_SH 请求一个共享的锁，所以它常用于读取。LOCK_EX 请求一个排它的锁，所以它常用于写。LOCK_UN 释放前面一次请求的锁；关闭该文件同样也释放任何锁。如果 LOCK_NB 位和 LOCK_SH 或者 LOCK_EX 一起使用，那么 flock 会马上返回，而不是等待一个可用的锁。检查返回状态看看你是否获得了你请求的锁。如果你不使用 LOCK_NB，那么你就有可能永远等待系统授予你想要的锁。

flock 的另外一个不明显但很常用的方面是它的锁只是劝告性的。自由的锁更灵活，但是不能象命令性的锁那样有保证。这就意味着用 flock 锁住的文件可能被那些没有使用 flock 的程序修改。等待红灯的车相互之间可以和睦相处，但和不遵守红灯的车之间可就不能相容了。防卫性驾驶。

有些 flock 的实现不能透过网络锁住东西。尽管理论上你可以使用更加系统相关的 fcntl 来做这件事，但这么做是否（能够）可靠仍然是有怀疑的。

下面是一个用于 Unix 系统的邮箱附件箱，它使用 flock(2) 来锁住邮箱：

```
use Fcntl qw/:flock/;      # 输出 LOCK_* 常量
sub mylock {
    flock(MBOX, LOCK_EX)
        or die "can't lock mailbox: $!";
    # 预防在我们等待的时候有家伙附加
    # 以及我们的 stdio 缓冲区失准
    seek(MBOX, 0, 2)
        or die "can't seek to the end of mailbox: $!";
}

open(mbox, ">>/USR/SPool/MAIL/$ENV{'USER'}")
    or die "can't open mailbox: $!";

mylock();
print MBOX $msg, "\n\n";
close MBOX
    or die "can't close mailbox: $!";
```

在那些支持真正的 flock(2) 系统调用的系统上，锁是在跨 fork 继承的。其他实现则没那么走运，并且很可能在 fork 之间丢失锁。又见第三十二章的 DB_File 模块获取其他的 flock 的例子。

29.2.40 fork

- fork

这个函数通过调用 fork(2) 系统调用，从一个进程中创建两个进程。如果它成功，该函数给父进程返回新创建的子进程 ID，而给子进程返回 0。如果系统没有足够的资源分配一个新的进程，那么调用失败并返回 undef。文件描述符（以及有时候还有在那些描述符上的锁）是共享的，而所有其他的東西都是拷贝的——或者至少看起来是那样的。

在早于 5.6 版本的 Perl 里，未冲刷的缓冲区在两个进程里都是没有冲刷的，这就意味着你需要在程序的早些时候在一个或多个文件句柄上设置 \$| 以避免输出重复。

一个产生子进程然而有检查“cannot fork”错误的几乎没有毛病的方法是：

```

use Errno qw(EAGAIN);
FORK: {
    if ($pid = fork) {
        # 父进程在此
        # 在 $pid 里可以看到子进程的进程 id
    }
    elsif (defined $pid) {    # 如果定义了, $pid 在这里是 0
        # 子进程在此
        # 你可以用 getppid 在这里获取父进程的 pid
    }
    elsif ($! == EAGAIN) {
        # EAGAIN 是认为可以恢复的 fork 错误
        sleep 5;
        redo FORK;
    }
    else {
        # 奇怪的 fork 错误
        die "Can't fork: $!\n";
    }
}

```

这些预防措施在那些做隐含的 `fork(2)` 的操作上是不必要的, 比如 `system`, 反勾号, 或者把一个进程当作一个文件句柄打开, 因为 Perl 在为你做 `fork` 的时候碰到临时的失败会自动重新尝试 `fork`。要注意记得使用 `exit` 结束子进程的代码, 否则子进程会不小心地离开条件块并且开始执行原来只是想让父进程执行的代码。

如果你 `fork` 以后再也不等待你的子进程, 那么你就会积累僵死进程 (那些父进程还没等待它们的退出进程)。在一些系统上, 你可以通过设置 `$SIG{CHLD}` 为 “IGNORE” 来避免这些; 在大多数系统上, 你必须 `wait` 你的垂死的子进程。参阅 `wait` 函数获做这些的例子, 或则后参阅第十六章的 “信号” 一节获取更多关于 `SIGCHLD` 的信息。

如果一个派生出来的子进程继承了系统文件描述符, 象 `STDIN` 和 `STDOUT` 等, 它们又和一个远程的管道或者套接字连接, 那么你可能不得不在子进程里把他们重新打开到 `/dev/null`。这是因为即使父进程退出, 子进程仍将带着这些文件句柄的拷贝继续生存。而远端服务器 (比如说, 一个 CGI 脚本或者一个从远程 `shell` 发起的后台任务。) 就会挂起, 因为它仍然等待所有拷贝关闭。重新把系统文件句柄打开成别的什么东西可以修补这个问题。

在大多数支持 `fork(2)` 的系统上, 人们做了大量努力把它变得尽可能地高效 (比如, 数据页的写时拷贝 (`copy-on-write`) 技术), 而它也成了过去几十年来多任务领域的典范。但是 `fork` 函数可能并没有有效地 (甚至可能是根本没有) 在那些不象 Unix 的系统上实现。比如, Perl 5.6 甚至在 Microsoft 系统上都模拟了一个合适的 `fork`, 但是并不能保证可以达到很好的性能。可能用 `Win32::Process` 模块的时候, 你的运气会好一些。

29.2.41 format

```
=    format NAME ==    picture line==    value list==    ...=.
```

这个函数声明一个图形行的命名序列 (以及相关值) 用于 `write` 函数。如果省略了 `NAME`, 名字缺省是 `STDOUT`, 正好是 `STDOUT` 文件句柄的缺省格式名字。因此, 和 `sub` 声明类似, 这是一个编译时发生的包全局声明, 并且值列表里的变量应该在格式定义时是可见的。也就是说, 词法范围的变量必须在该文件的更早的位置定义, 而动态范围的变量只需要在调用 `write` 的时候设置就可以了。下面是一个例子 (它假设我们已经计算了 `$cost` 和 `$quantity`) :

```
my $str = "widget";    # 词法范围的变量
```

```

format Nice_Output =
Test: @<<<<<<< @| | | | @>>>>>
    $str,    $%,    '$'    . int($num)
.

local $~ = "Nice_Output";    # 选择输出格式。
local $num = $cost * $quantity;    # 动态范围的变量。

write;

```

和文件句柄类似，格式名字是存在于一个符号表（包）里的标识符，而且是可以包名修饰成全名的。在一个符号表的记录的类型团里，格式存放在它们自己的名字空间里，它和文件句柄，目录句柄，标量，散列和子过程是不同的。不过，和其他六种类型一样，一个叫做 **Whatever** 的格式也会被一个对 ***Whatever** 类型团的 **local** 所影响。换句话说，格式只是包含在类型团里的另外一种小东西，与其他小东西相互独立。

第七章，格式，里的“格式变量”节里包含大量它们的细节和它们的使用的例子。第二十八章描写了内部的格式相关变量，而 **English** 和 **IO::Handle** 模块提供了一个对他们简化的访问的接口。

29.2.42 formline

- formline PICTURE, LIST

这是一个 **format** 用的内部函数，不过你还是可以自己调用它。它总是返回真。它根据 **PICTURE** 的内容格式化一系列数值，把输出放到格式化输出累加器，**\$^A**（或者是 **\$ACCUMULATOR**——如果你使用了 **English** 模块）。最后，当完成一个 **write** 的时候，**\$^A** 的内容写入某个文件句柄，但你也可以自己读取 **\$^A** 然后把 **\$^A** 设置为 **""**。一个格式通常每行表格做一个 **formline**，但是 **formline** 函数本身并不在意在 **PICTURE** 里嵌入了多少个新行。这意味着 **~** 和 **~~** 记号将把整个 **PICTURE** 当作一行看待。因此你可能需要用多个 **formline** 来实现一个记录格式，就好像格式化编译器在内部做的那样。

如果你在图形周围放双引号的事情要小心，因为一个 **@** 字符可能会被拿去表示一个数组名字的开头。参阅第六章“格式”获取使用的例子。

29.2.43 getc

- getc FILEHANDLE
- getc

这个函数从附着到 **FILEHANDLE** 上的输入文件返回下一个字节。在文件结尾的时候，或者碰到 **I/O** 错误的时候它返回 **undef**。如果省略了 **FILEHANDLE**，那么该函数从 **STDIN** 中读取。

这个函数有点慢，但是偶尔可以用于从键盘上读取一个字符输入（实际上是字节）——前提是你能让你的键盘输入不经过缓冲。这个函数需要从标准 **I/O** 库里来的未经缓冲的输入。糟糕的是，标准的 **I/O** 库还没有标准到能够提供一种可移植的方法，告诉下层操作系统供应无缓冲的键盘输入到标准 **I/O** 系统。要做着件事情，你必须稍微更聪明一点，并且采取操作系统相关的做法。在 **Unix** 里，你可以说：

```

if ($BSD_STYLE) {
    system "stty cbreak /dev/tty 2>&1"
} else {
    system "stty", "-icanon", "eol", ".....";
}

```

```

$key = getc;

if ($BSD_STYLE) {
    system "stty -cbreak /dev/tty 2>&1";
} else {
    system "stty", "icanon", "eol", "^@";    # ASCII NUL
}

print "\n";

```

上面的代码把在终端上敲入的下一个字符（字节）放到字符串 `$key` 里。如果你的 `stty` 程序有象 `cbreak` 这样的选项，那么你就需要 `$BSD_STYLE` 为真的地方的代码。否则你就需要它为假的地方的代码。判断 `stty(1)` 的选项就留给读者做练习吧。

POSIX 模块用 `POSIX::getattr` 函数提供了一个做这件事情的一个更具移植性的版本。又见来自离你最近的 CPAN 站点里的 `Term::ReadKey` 模块获取更具移植性和更灵活的方法。

29.2.44 getgrent

- `getgrent`
- `setgrent`
- `endgrent`

这些过程遍历你的 `/etc/group` 文件（或者是别人的 `/etc/group` 文件，如果这个文件来自一台服务器什么的地方的话）。在列表环境里，`getgrent` 的返回值是：

```
($name, $passwd, $gid, $members)
```

这里 `$members` 包含一个空格分隔的列表，该列表就是该组成员的登录名字。要设置一个散列把组名字转换成 GID，你可以用：

```

while (($name, $passwd, $gid) = getgrent) {
    $gid{$name} = $gid;
}

```

在标量环境里，`getgrent` 只返回组名字。标准的 `User::grent` 模块支持一个此函数通过名字访问的接口。参阅 `getgrent(3)`。

29.2.45 getgrgid

- `getgrgid GID`

这个函数通过组标识查找一条组文件记录。返回值在列表环境中是：

```
($name, $passwd, $gid, $members)
```

这里 `$members` 包含一个用空格分隔的列表，该列表就是该组成员的登录名字。如果你想重复做这件事情，考虑用 `getgrent` 把数据缓冲到一个散列里面。

在标量环境中，`getgrgid` 只返回组名字。`User::grent` 模块支持此函数的一个通过名字访问的接口。参阅 `getgrgid(3)`。

29.2.46 getgrnam

- `getgrnam NAME`

这个函数通过组名字查找一条组文件记录。返回值在列表环境中是：

```
($name, $passwd, $gid, $members)
```

这里 `$members` 包含一个用空格分隔的列表，该列表就是该组成员的登录名字。如果你想重复做这件事情，考虑用 `getgrent` 把数据缓冲到一个散列里面。

在标量环境中，`getgid` 只返回组 ID。`User::grent` 模块支持此函数的一个通过名字访问的接口。参阅 `getgrgid(3)`。

29.2.47 gethostbyaddr

- `gethostbyaddr ADDR, ADDRTYPE`

这个函数把地址转换成名字（和改变地址）。`ADDR` 应该是一个封包的二进制网络地址，而 `ADDRTYPE` 实际上通常应该是 `AF_INET`（来自 `Socket` 模块）。其返回值在列表环境里是：

```
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyaddr($packed_binary_address, $addrtype);
```

这里 `@addrs` 是一个封包的二进制地址。在互联网域里，每个地址都（因历史关系）是四个字节长，并且可以通过用下面这样的东西解包：

```
($a, $b, $c, $d) = unpack('C4', $addrs[0]);
```

另外，你可以给 `sprintf` 用 `v` 修饰词把它直接转换成点向量表示法：

```
$dots = sprintf "%vd", $addrs[0];
```

`Socket` 模块的 `inet_ntoa` 函数可以用于生成可打印的版本。这个方法在我们都准备切换到 `IPv6` 的时候会变得很重要。

```
use Socket;
$printable_address = inet_ntoa($addrs[0]);
```

在标量环境里，`gethostbyaddr` 只返回主机名字。

要从一个点向量中生成一个 `ADDR`，用：

```
use Socket;
$ipaddr = inet_aton("127.0.0.1");      # localhost
$claimed_hostname = gethostbyaddr($ipaddr, AF_INET);
```

有趣的是，在 `Perl 5.6` 里，你可以忽略 `inet_aton()` 并且使用新的用于版本号的 `v` 字符串表示法操作 `IP` 地址：

```
$ipaddr = v127.0.0.1;
```

参阅第十六章“套接字”一节获取更多的例子。`Net::hostent` 模块支持一个此函数的通过名字使用的接口。参阅 `gethostbyaddr(3)`。

29.2.48 gethostbyname

- `gethostbyname NAME`

这个函数把一个网络主机名翻译成它的对应地址（以及其他名字）。其返回值在列表环境里是：

```
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyname ($remote_hostname);
```


这里的 `@addrs` 是一个裸地址的列表。在互联网域，每个地址（因历史原因）是四个字节长，可以用下面方法解包的东西：

```
($a, $b, $c, $d) = unpack('C4', $addrs[0]);
```

你可以用带 `v` 修饰词的 `sprintf` 把它们直接转换成向量符号：

```
$dots = sprintf "%vd", $addrs[0];
```

在标量环境里，`gethostbyname` 只返回主机地址：

```
use Socket;
$ipaddr = gethostbyname($remote_host);
printf "%s has address %s\n",
    $remote_host, inet_ntoa($ipaddr);
```

参阅第十六章里的“套接字”一节获取另外一种方法。`Net::hostent` 模块提供了一个用名字访问这个函数的接口。又见 `gethostbyname(3)`。

29.2.49 gethostent

- `gethostent`
- `sethostent STAYOPEN`
- `endhostent`

这个函数遍历你的 `/etc/hosts` 文件并且每次返回一条记录。`gethostent` 的返回值是：

```
($name, $aliases, $addrtype, $length, @addrs)
```

这里 `@addrs` 是一个裸地址的列表。在互联网域，每个地址（因历史原因）是四个字节长，可以用下面方法解包的东西：

```
($a, $b, $c, $d) = unpack('C4', $addrs[0]);
```

使用 `gethostent` 的脚本不能认为是可移植的。如果一台机器使用一个名字服务器，它就不得不询问互联网以满足一个获取该星球上每一台机器地址的请求。所以 `gethostent` 没有在这样的机器上实现。参阅 `gethostent(3)` 获取其他细节。

`Net::hostent` 模块模块提供了一个用名字访问这个函数的接口。

29.2.50. getlogin

- `getlogin`

如果有的话，这个函数返回当前登录名。在 `Unix` 系统上，它是从 `utmp(5)` 文件里读取的。如果它返回假，那么用 `getpwuid` 取代。比如：

```
$login = getlogin() || (getpwuid($<)) [0] || "Intruder!!";
```

29.2.51 getnetbyaddr

- `getnetbyaddr ADDR, ADDRTYPE`

这个函数把一个网络地址转换成对应的网络名字。在列表环境中其返回值是：

```
use Socket;
($naem, $aliases, $addrtype, $net) = getnetbyaddr(127, AF_INET);
```

在标量环境中，`getnetbyaddr` 只返回网络名字。`Net::netent` 模块支持一个通过名字访问这个函数的接口。参阅 `getnetbyaddr(3)`。

29.2.52 getnetbyname

- `getnetbyname NAME`

这个函数把一个网络名字转换成它对应的网络地址。其返回值在列表环境里是：

```
($name, $aliases, $addrtype, $net) = getnetbynaem("loopback");
```

在标量环境里，`getnetbyname` 只返回网络地址。`Net::netent` 模块支持一个通过名字访问这个函数的接口。参阅 `getnetbyname(3)`。

29.2.53. getnetent

- `getnetent`
- `setnetent STAYOPEN`
- `endnetent`

这个函数遍历你的 `/etc/networks` 文件。其返回值在列表环境中是：

```
($name, $aliases, $addrtype, $net) = getnetent();
```

在标量环境里，`getnetent` 只返回网络名字。`Net::netent` 模块支持一个通过名字访问这个函数的接口。参阅 `getnetent(3)`。

现在，网络名字这个概念看上去相当奇怪；大多数 IP 地址是在无命名（而且也是无法命名的）子网里。

29.2.54. getpeername

- `getpeername SOCKET`

这个函数返回该 `SOCKET` 连接中对端的封包地址。比如：

```
use Socket;
$hersockaddr = getpeername SOCK;
($port, $heraddr) = sockaddr_in($hersockaddr);
$herhostname = gethostbyaddr($heraddr, AF_INET);
$herstraddr = inet_ntoa($heraddr);
```

29.2.55. getpgrp

- `getpgrp PID`

这个函数为声明的 `PID`（对当前进程用 `PID=0`）返回当前进程组。如果在那些没有实现 `getpgrp(2)` 的机器上使用，那么调用 `getpgrp` 将抛出一个例外。如果省略了 `PID`，该函数返回当前进程的进程组（与使用 `PID` 为 0 时一样）。在那些用 `POSIX getpgrp(2)` 系统调用实现这个操作符的系统上，必须省略 `PID` 或者，如果提供了，必须为 0。

29.2.56 getppid

- `getppid`

这个函数返回父进程的进程 ID。在典型的 Unix 系统上，如果你的父进程 ID 改为 1，那就意味着

你的父进程已经退出并且你已经被 `init(8)` 进程收养了。

29.2.57. `getpriority`

- `getpriority WHICH, WHO`

这个函数返回一个进程，一个进程组或者一个用户的当前优先级。参阅 `getpriority(2)`。如果在一台没有实现 `getpriority(2)` 的机器上调用 `getpriority` 将抛出一个例外。

CPAN 的 `BSD::Resource` 模块提供了一个更便利的接口，包括提供给 `WHICH` 的 `PRIO_PROCESS`，`PRIO_PGRP`，和 `PRIO_USER` 符号常量。尽管这几个常量通常是分别设置成 0，1，和 2，你实际上还是不知道在 C 的黑暗的 `#include` 文件的领土里发生了什么事情。

`WHO` 的值为 0 意思是当前进程，进程组，或者用户，因此要获得当前进程的优先级，用：

```
$curprio = getpriority(0, 0);
```

29.2.58 `getprotobyname`

- `getprotobyname NAME`

这个函数把一个协议名字转换成它对应的数字。在列表环境里的返回值是：

```
($name, $aliases, $protocol_number) = getprotobyname("tcp");
```

如果在标量环境里调用，`getprotobyname` 只返回协议号。`Net::proto` 模块提供一个通过名字访问这个函数的接口。参阅 `getprotobyname(3)`。

29.2.59 `getprotobynumber`

- `getprotobynumber NUMBER`

这个函数把一个协议数字转换成它对应的名字。在列表环境里的返回值是：

```
($name, $aliases, $protocol_number) = getprotobynumber(6);
```

如果在标量环境里调用，`getprotobynumber` 只返回协议名字。`Net::proto` 模块提供一个通过名字访问这个函数的接口。参阅 `getprotobynumber(3)`。

29.2.60 `getprotoent`

- `getprotoent`
- `setprotoent STAYOPEN`
- `endprotoent`

这些函数遍历 `/etc/protocols` 文件。在列表环境里，`getprotoent` 的返回值是：

```
($name, $aliases, $protocol_number) = getprotoent();
```

如果在标量环境里调用，`getprotoent` 只返回协议名字。`Net::proto` 模块提供一个通过名字访问这个函数的接口。参阅 `getprotoent(3)`。

29.2.61 `getpwent`

- `getpwent`
- `setpwent`
- `endpwent`

这些函数概念上是遍历你的 `/etc/passwd` 文件，但是如果你是超级用户并且使用了影子文件，或者用了 `NIS` 或 `NIS+` 两者之一，那么它可能涉及到 `/etc/shadow` 文件。在列表环境中的返回值是：

```
($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwent()
```

有些机器可能使用份额（`quota`）和注释域做名字用途，但其他的域都是一样的。如果想设置一个散列把登录名字转换成 `UID`，用：

```
while (($name, $passwd, $uid) = getpwent()) {
    $uid{$name} = $uid;
}
```

在标量环境里，`getpwent` 只返回用户名。`User::pwent` 模块支持一个通过名字访问这个函数的接口。参阅 `getpwent(3)`。

29.2.62 getpwnam

- `getpwnam NAME`

这个函数把一个用户名翻译成对应的 `/etc/passwd` 文件的记录。其返回值在列表环境里是：

```
($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwnam($name)
```

在支持影子口令的系统上，要想获取真正的口令，你就必须是超级用户。你的 `C` 函数库会注意你是否合适的权限并且打开 `/etc/shadow`（或者那些保存影子口令的文件）。至少，这就是它的工作方法。如果你的 `C` 库太蠢而不会注意这些，那么 `Perl` 会试图做这些事情。

如果需要重复查找，请考虑把数据用 `getpwent` 缓存到一个散列里。

在标量环境里，`getpwnam` 只返回数字用户 `ID`。`User::pwent` 模块支持一个通过名字访问这个函数的接口。参阅 `getpwnam(3)` 和 `passwd(5)`。

29.2.63. getpwuid

- `getpwuid UID`

这个函数把一个数字用户 `ID` 转换成对应的 `/etc/passwd` 文件记录。其返回值在列表环境中是：

```
($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwuid($uid)
```

如果需要重复查找，请考虑把数据用 `getpwent` 缓存到一个散列里。

在标量环境里，`getpwuid` 返回用户名。`User::pwent` 模块支持支持一个通过名字访问这个函数的接口。参阅 `getpwnam(3)` 和 `passwd(5)`。

29.2.64. getservbyname

- `getservbyname NAME, PROTO`

这个函数把一个服务（端口）名翻译成它对应的端口号，`PROTO` 是象“`tcp`”这样的协议名。其返回值在列表环境里是：

```
($name, $aliases, $port_number, $protocol_name) = getservbyname($name, $proto)
```

在标量环境里，`getservbyname` 只返回该服务的端口号。`Net::Servent` 模块支持支持一个通过名字访问这个函数的接口。参阅 `getservbyname(3)`。

29.2.65. getservbyport

- getservbyport PROT, PROTO

这个函数把一个服务（端口）号翻译成它对应的名字。PROTO 是一个象“tcp”这样的协议名字。其返回值在列表环境中是：

```
($name, $aliases, $port_number, $protocol_name) = getservbyport(80, "tcp");
```

在标量环境中，getservbyport 只返回服务名。Net::Servent 模块支持支持一个通过名字访问这个函数的接口。参阅 getservbyport(3)。

29.2.66. getservent

- getservent
- setservent STAYOPEN
- endservent

这个函数遍历 /etc/services 文件或者该文件的等效文件。其返回值在列表环境中是：

```
($name, $aliases, $port_number, $protocol_name) = getservent();
```

在标量环境中，getservent 只返回服务端口名。Net::Servent 模块支持支持一个通过名字访问这个函数的接口。参阅 getservent(3)。

29.2.67. getsockname

- getsockname SOCKET

这个函数返回该 SOCKET 连接的本地端的封包套接字地址。（为什么你还不知道自己的地址？也许是因为你在 accept 之前绑定了一个指向服务器套接字的包含通配符的地址，而现在你想知道别人是用什么接口和你连接的。或者你的套接字是你的父进程传递过来的——比如说，inetd。）

```
user Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
$myname = gethostbyaddr($myaddr, AF_INET);
printf "I am %s [%vd]\n", $myname, $myaddr;
```

29.2.68. getsockopt

- getsockopt SOCKET, LEVEL, OPTNAME

这个函数返回你请求的套接字选项，如果有错误则返回 undef。参阅 setsockopt 获取更多信息。

29.2.69 glob

- glob EXPR
- glob

这个函数把 EXPR 的值带着 shell 那样的文件名扩展返回。它是实现 <*> 操作符的内部函数。

由于历史原因，这个算法和 csh(1) 的扩展风格相匹配，而不是 Bourne shell 的。早于 5.6 版本的 Perl 使用了一个外部的处理，但 5.6 及以后的版本在内部进行聚团的工作。那些第一个字符是点（“.”）的文件被忽略，除非这个字符是明确匹配的。一个星号（“*”）匹配任意字符的任意序列（包括空）。一个问号（“?”）匹配任意一个字符。一个方括弧序列（“[...]”）声明一个简单的

字符表，比如 “[chy0-9]”。字符表可以用音调符号取反，象 “*.[^oa]”，它匹配任意非点文件，这些文件的文件名博爱喊一个点，后面跟着一个字符在文件名尾部，但这个字符既不能是 “a” 也不能是 “o”。一个波浪号 (“~”) 扩展成一个家目录，象 “~/.rc” 是指当前用户的所有 “rc” 文件，或者 “~jane/Mail/*” 似乎所有 Jane 的邮件文件。花括弧可以用于候补，象在 “!/{mail,ex,csh,twm,}rc” 里面的的是获取那些特定的 rc 文件。

如果你想聚集那些可能包含空白的文件名，你坑内需要直接使用 `File::Glob` 模块，因为老祖父 `glob` 把空白用于分隔多个模式，比如 `<*.c *.h>`。更多细节，请参阅第三十二章的 `File::Glob`。调用 `glob`（或者 `<*>` 操作符）自动 `use` 该模块，因此如果该模块莫名其妙地从你的库里消失了，那么就会抛出一个例外。

当你调用 `open` 的时候，Perl 并不扩展通配符，包括波浪号。你需要先把结果 `glob` 起来。

```
open(MAILRC, "~/.mailrc")      # 错：波浪号是一个 shell 的东西
or die "can't open ~/.mailrc: $!";

open(MAILRC, (glob("~/.mailrc"))[0]) # 先扩展波浪号
or die "can't open ~/.mailrc: $!";
```

`glob` 函数和 Perl 的类型团的概念没有任何关系，只不过它们都用 `*` 代表多个项。

又见第二章里的“文件名聚集操作符”。

29.2.70. gmtime

- gmtime EXPR
- gmtime

这个函数把 `time` 这样的函数返回的时间转换成对应的格林威治时间（也叫做 GMT，或者 UTC，或者在某些文化里甚至是 Zulu，不过让人奇怪的是在 Zulu 文化里不这么叫。）的一个九个元素的列表。它的典型用法如下：

```
# 0      1      2      3      4      5      6      7      8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime;
```

如果象这种情况一样，省略了 `EXPR`，那么它做 `gmtime(time())`。Perl 库模块 `Time::Local` 包含一个子过程，`timegm`，它可以把该列表转换回一个时间值。

所有列表元素都是数字并且是从 `struct tm`（这是一个 C 编程结构——别害怕）里直接拿出来的。因此这就意味着 `$mon` 的范围是 0 .. 11，而一月是月份 0，而 `$wday` 的范围是 0 .. 6，星期日是 0。你很容易记住哪些是零为基的，因为那些都是你在包含月份和日期名字的零为基的数组里用做的脚标的东西。

比如，要获取伦敦的当前月份，你可以说：

```
$london_month = (qw(Jan Feb Mar Apr May Jun
                    Jul Aug Sep Oct Nov Dec))[(gmtime)[4]];
```

`$year` 是自 1900 年以来的年数；也就是说，在 2023 年，`$year` 是 123，而不只是 23。要获取四位数年，只需要用：`$year + 1900`。要获取两位数年（比如2001里的“01”），用 `sprintf` (“%2d”, `$year % 100`)。

在标量环境，`gmtime` 返回一个基于 GMT 时间值的 `ctime(3)` 风格的字串。`Time::gmtime` 模块支持一个通过名字访问这个函数的接口。参阅 `POSIX::strftime()` 获取更多更好地格式化时间的方法。

这个标量值是与区域设置无关的，而是一个 Perl 的内建。同样参阅 `Time::Local` 模块和通过 POSIX 可以获取的 `strftime(3)` 和 `mktime(3)` 函数。如果需要获取类似的，但又是与区域设置相关的时间字符串，那么你要正确设置你的区域环境变量（请参考 `perllocale` 手册页），并尝试：

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

这里的 `%a` 和 `%b` 逃逸是表示星期几和第几月的缩写形式，可能不是在所有区域设置中都是三个字符宽。

29.2.71. goto

- `goto LABEL`
- `goto EXPR`
- `goto &NAME`

`goto LABEL` 先找到标记着 `LABEL` 的语句，然后从那里重新开始执行。如果没有找到 `LABEL`，则抛出一个例外。它不能用于进入任何需要初始化的构造中去，比如一个子过程或者一个 `foreach` 循环。它也不能用于进入那些优化过的构造中。你可以用它进入在一个动态范围里的（注：这就意味着如果它在当前过程中没有找到标签 `LABEL`，那么它就往回到调用当前过程的那个过程里找标签，因此可以把你的程序变得几乎不能维护。）几乎任何地方，包括子过程外边，如果要退出子过程的话，最好还是使用其他的构造，比如 `last` 或者 `die`。Perl 的作者自己从来没有觉得要这么用 `goto`（也就是说在 Perl 里，C 则是另外一个问题了。）

在矛盾的更高层次（或者说愚蠢的更深层次），Perl 允许 `goto EXPR`，这里的 `EXPR` 可以得出一个标签名字，而该标签位置肯定是到运行时才确定的，因为当编译该语句的时候该标签还是未知的。这样就可以象 FORTRAN 那样计算 `goto`，不过如果你为可维护性做优化的话，我们并不推荐你这么 做：（注：“待定”的东西总是很有趣，所以我们在这里还是做个实验。）

```
goto +("FOO", "BAR", "GLARCH")[$i];
```

不相关的 `goto &NAME` 是非常神奇的东西，它把当前运行着的子过程替换为一个命名子过程的调用。这种构造可以在不麻烦 `AUTOLOAD` 子过程（它们希望装载其他子过程）的情况下假装这个新的子过程（不是原来那个）是先调用的（只不过任何原来那个子过程对 `@_` 的修改都传播到这个替换子过程中）。在 `goto` 之后，甚至 `caller` 都不能确定是否最初的 `AUTOLOAD` 过程是先调用的。

29.2.72. grep

- `grep EXPR, LIST`
- `grep BLOCK LIST`

这个函数在布尔环境里为 `LIST` 里的每个元素计算 `EXPR` 或者 `BLOCK`，临时地把 `$_` 轮流设置为每个元素，非常类似 `foreach` 构造。在列表环境中，它返回一个对该表达式为真的元素的列表。（该操作符是按照大家喜爱的同名 Unix 程序命名的，那个程序从一个文件中抽取出匹配特定模式的行。在 Perl 里，该表达式通常是一个模式，但并不必须是模式。）在标量环境里，`grep` 返回该表达式为真的次数。

如果 `@all_lines` 包含代码的行，下面的例子删除注释行：

```
@code_lines = grep !/^#\s*/, @all_lines;
```

因为 `$_` 是每个列表值的隐含别名，修改 `$_` 就会修改原始列表的元素。尽管这样做是允许的并且也有用，但如果你没有心理准备，它还是可能会导致非常奇怪的结果。比如：

```
@list = qw(barney fred dino wilma);
```

```
@greplist = grep { s/^[bfd]// } @list;
```

@greplist 现在是 “arney”，“red”，“ino”，但 @list 现在是 “arney”，“red”，“ino”，“wilma”！因此，小心从事。

又见 map。下面两个语句的作用是相同的：

```
@out = grep {EXPR} @in;
@out = map { EXPR ? $_ : () } @in
```

29.2.73. hex

- hex EXPR
- hex

这个函数把 EXPR 当作一个十六进制字符串并且返回相等的十进制值。如果前面有 “ox”，那么被忽略。要转换可能以 0，0b，或者 0x 开头的字符串，请参考 oct。下面的代码把 \$number 设置为 4,294,906,560：

```
$number = hex(ffff12c0");
```

要实现相反的功能，用 sprintf：

```
sprintf "%lx", $number;      #（那里是 L 的小写，不是阿拉伯数字一）
```

Hex 字符串只能表示整数。那些可能导致整数溢出的字符串发出一个警告。

29.2.74 import

- import CLASSNAME LIST
- import CLASSNAME

实际上没有内建的 import 函数。它只是一个普通的类方法，那些模块定义（或者继承）它用来通过 use 操作符把名字输出到另外一个模块。参阅 use 获取细节。

29.2.75. index

- index STR, SUBSTR, OFFSET
- index STR, SUBSTR

这个函数在一个字符串里寻找另外一个字符串。它返回在 STR 里找到的第一个 SUBSTR 的位置。如果声明了 OFFSET，表示在开始搜索之前，忽略从开头开始的多少个字符。位置是以零为基的（或者你设置的脚标基变量 \$[的值——不过最好别干那事）。如果没有找到字符串，该函数返回脚标基减一，通常是 -1。要对你的字符串扫描一遍，你可能会说：

```
$pos = -1;
while (($pos = index($string, $lookfor, $pos)) > -1) {
    print "Found at $pos\n";
    $pos++;
}
```

29.2.76. int

- int EXPR
- int

这个函数返回 EXPR 的整数部分。如果你是 C 程序员，那么你很容易忘记和除法一起使用 int，因

为除法在 Perl 里是浮点数操作：

```
$average_age = 939/16;      # 结果是 58.6875 (在 C 里是 58)
$average_age = int 939/16;   # 结果是 58
```

你不能把这个函数用做通用的圆整方法，因为它向 0 截断，而且还因为浮点数的机器表现形式有时候可能生成不那么直观的结果。比如，`int(-6.725/0.025)` 结果是 -268 而不是正确的 -269；原因是该值的样子可能更象 -268.99999999999994315658。通常，`sprintf`，`printf`，或者 `POSIX::floor` 和 `POSIX::ceil` 函数可能比 `int` 的效果更好些：

```
$n = sprintf("%0.f", $f);    # 圆整到（不是截断）最接近的整数
```

29.2.77. ioctl

- `ioctl` FILEHANDLE, FUNCTION, SCALAR

这个函数实现 `ioctl(2)` 系统调用，这个系统调用控制 I/O。要获取正确的函数定义，可能你首先得说：

```
require "sys/ioctl.ph";      # 可能是 /usr/local/lib/perl/sys/ioctl.ph
```

如果 `sys/ioctl.ph` 不存在或者没有正确的定义。你就不得不基于你的 C 头文件（比如 `sys/ioctl.h`）自己玩了。（Perl 发布中包括一个叫 `h2ph` 的脚本可以帮你做这些事情，但是运行它不那么有用。）根据 `FUNCTION` 的不同，可能会读写 `SCALAR`——一个指向 `SCALAR` 的字符串值的指针将被当作实际 `ioctl(2)` 调用的第三个参数传递。（如果 `SCALAR` 没有字符串值，而是有一个数字值，那么该值将被直接传递，而不是传递指向字符串值的指针。）`pack` 和 `unpack` 函数可以用于操作 `ioctl` 的结构值。下面的例子判断用 `FIONREAD` `ioctl` 的时候还有多少字节可读：

```
require 'sys/ioctl.ph';

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)
    or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

如果没有安装 `h2ph` 或者它没什么用，你可以手工 `grep` 包含文件或者写一个小 C 程序打印出该值。

`ioctl`（和 `fcntl`）的返回值如下：

系统调用返回	Perl 返回
-1	undef
0	字符串 “0 but true”
其他任何东西	那个数字

因此 Perl 在成功时返回真，而失败时返回假，但是你还是可以很容易地判断操作系统返回的实际值：

```
$retval = ioctl(...) || -1;
printf "ioctl actually returned %d\n", $retval;
```

特殊的字符串 “0 but true” 可以免于 `-w` 关于不当整数转换的警告。

对 `ioctl` 的调用不能认为是可移植的。如果说你只是想为整个脚本关闭回现一次，更可移植的方法是：

```
system "stty -echo";    # 在大多数 Unix 系统上都能用
```

不要因为你能够在 Perl 干某事就认为你就应该这么做。引用 Apostle Paul 的话来说，“任何事情都是允许的——不过并非任何事情都是有益的。”

如果需要更好的移植性，你应该看看 CPAN 里的 `Term::ReadKey` 模块。

29.2.78. join

- join EXPR, LIST

这个函数把 LIST 里分离的字串连接成一个字串，里面的域用 EXPR 的值分隔，并且返回该字串。比如：

```
$rec = join ':', $login, $passwd, $uid, $gid, $gcos, $home, $shell;
```

如果要反向功能，参阅 `split`。要把东西按照固定位置的域连接起来，参阅 `pack`。把许多东西连接起来最高效的办法就是用空字串把他们 join 起来：

```
$string = join "", @array;
```

和 `split` 不同，`join` 不能接受模式作为它的第一个参数，如果你这么干它会生成一个警告。

29.2.79 keys

- keys HASH

这个函数返回指定的 HASH 里的所有键字组成的一个列表。返回的键字在外观上是随机顺序的，但是这个顺序和 `values` 或者 `each` 函数产生的顺序是一样的（假设该散列在不同调用之间没有被修改），它的一个副作用是重置 HASH 的遍历符。下面是一个（相当呆的）打印你的环境变量的方法：

```
@keys = keys %ENV;      # 以同样顺序的键字
@values = values %ENV;   # 数值
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

你可能更愿意看到键字排序的环境：

```
foreach $key (sort keys %ENV) {
    print $key, '=', $ENV{$key}, "\n";
}
```

你可以直接对散列的数值排序，但是如果你不把这些数值映射回键字，这么做好象没什么用。要按照数值对散列排序，通常你需要通过提供一个基于键字访问数值的比较函数来对 `keys` 排序。下面是一个对散列的数值的降序排序：

```
foreach $key (sort {$hash{$b} <=> $hash{$a}} keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

在一个与某个相当大的 DBM 文件捆绑的散列上使用 `keys` 会生成一个相当大的列表，导致你产生一个相当大的进程。这时候你可能更愿意使用 `each` 函数，它是一个一个地遍历散列，而不会把它们一下子都吞到一个粒度非常大的列表里。

在标量环境里，`keys` 返回散列中元素的数量（并且重置 `each` 遍历器）。不过，要从一个捆绑的散

列里获取这样的信息，包括 DBM 文件，Perl 就必须遍历整个散列，所以这样做并不高效。在空环境里调用 `keys` 会好一些。

如果用做左值，`keys` 就增大为给定散列分配的散列桶的数目。（类似于通过给 `$#array` 赋予更大的数字预扩展一个数组。）如果你知道某个散列要增长，而且你知道它要增长到多大，那么预扩展你的散列可以获得效率上的提高。如果你说：

```
keys %hash = 1000;
```

那么 `%hash` 就会至少有分配给它的 1000 个桶（实际上，你得到 1024 个桶，因为它是圆整为最近的二的指数）。你 cannot 通过这种方法用 `keys` 缩小分配的桶的数量（不过如果你不小心这么做了也不用担心，因为这样的尝试是没有作用的）。甚至你 `%hash = ()`，这些桶也会照样存在。如果你想在 `%hash` 仍然在范围里的时候释放它的存储器，那么用 `undef %hash`。

又见 `each`, `values`, 和 `sort`。

29.2.80. kill

- `kill SIGNAL, LIST`

这个函数向一个列进程发送一个信号。对于 `SIGNAL` 而言，你既可以用一个整数也可以用信号名字（前面没有“SIG”）。如果试图使用一个系统不识别的 `SIGNAL` 名字将会抛出一个例外。该函数返回成功发送信号的进程数。如果 `SIGNAL` 是负数，该函数杀死进程组，而不是进程。（在 [SysV²](#) 上，一个负数的进程号也可以杀死进程组，但那是不可移植的。）`PID` 为零的时候向与发送者同组的所有进程发送该信号。比如：

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
kill 'STOP', getppid      # 这样就可以延缓我的登录 shell ...
    unless getppid == 1;   # (但不要戏弄 init(8)。)
```

一个为 0 的 `SIGNAL` 测试一个进程是否仍然存活以及你是否仍有权限给它发信号。这里并不发信号。因此你可以用这个方法测试一个进程是否仍然存活以及是否没有改变 `UID`。

```
use Errno qw(ESRCH EPERM);
if (kill 0 => $minion) {
    print "$minion is alive!\n";
} elsif ( $! == EPERM) {          # UID 改变了
    print "$minion has escaped my control!\n";
} elsif ( $! == ESRCH) {
    print "$minion is deceased.\n"; # 或者是僵尸
} else {
    warn "Odd; I couldn't check on the status of $minion: $!\n";
}
```

参阅第十六章的“信号”节。

29.2.81. last

- `last LABEL`
- `last`

`last` 马上退出有问题的循环，就好象 C 或者 Java 里的 `break`（也是在循环里使用）语句一样。如果省略了 `LABEL`，那么该操作指的是最内层的闭合循环。如果有任何 `continue` 块，那么将不执行。

```

LINE: while () {
    last LINE if /^$/;      # 如果完成头处理以后则退出
    # 循环其他部分
}

```

`last` 不能用于退出一个返回一个值的块，比如 `eval {}`，`sub {}`，或者 `do {}`，并且也不能用于退出一个 `grep` 或者 `map` 操作，如果打开了警告，那么如果你 `last` 出了一个不在你的当前词法范围的循环，比如说一个在调用你的子过程里的循环，那么 Perl 会警告你。

一个块本身从语意上来说是等效于一个执行一次的循环的。因此 `last` 可以用于实现在这样的一个块中提前退出。

又见第四章获取一个 `last`，`next`，`redo`，和 `continue` 如何运行的例子。

29.2.82. lc

- `lc EXPR`
- `lc`

这个函数返回 `EXPR` 的小写形式。它是实现双引号字符串里 `\L` 逃逸的内部函数。如果 `use locale` 起作用，那么将会考虑你当前的 `LC_CTYPE` 区域设置，不过区域设置和 `Unicode` 之间如何相互作用仍然是一个正在进行的研究。参阅 `perllocale` 手册页获取更多最近的结果。

29.2.83. lcfirst

- `lcfirst EXPR`
- `lcfirst`

这个函数返回 `EXPR` 的第一个字符小写的版本。它是实现双引号字符串里的 `\l` 逃逸的内部函数。如果 `use locale` 起作用，并且我们知道如何处理区域和 `Unicode` 的关系，那么将会考虑你当前的 `LC_CTYPE` 区域设置。

29.2.84. length

- `length EXPR`
- `length`

这个函数返回标量值 `EXPR` 的以字符记的长度。如果省略 `EXPR`，它返回 `$_` 的长度。（但是要小心不要让下一个东西看着象一个 `EXPR` 的开头，否则 Perl 的词法器将被你搞糊涂。比如，`length < 10` 是编译不了的。如果有疑问，请使用圆括弧。）

不要试图使用 `length` 寻找一个数组或者散列的大小。用 `scalar @array` 获取数组的尺寸，用 `scalar keys %hash` 获取散列中键字/数值对的数量。（通常如果 `scalar` 多余则会把它省略。）

要查看一个字串的按字符计的长度，而不是按字节计，那么你可以说：

```
$blen = do { use bytes; length $string; };
```

或者：

```
$blen = bytes::length($string);      # 必须先用 bytes
```

29.2.85. link

- `link OLDFILE, NEWFILE`

这个函数创建一个链接到旧文件名的新文件名，该函数成功时返回真，失败时返回假。又见本章中的 `symlink`。这个函数在那些非 **Unix** 风格的文件系统上很可能没有实现。

29.2.86. listen

- `listen SOCKET, QUEUESIZE`

这个函数告诉系统说你准备在这个 **SOCKET** 上接受联接，并且系统可以把等待的联接排成最长 **QUEUESIZE** 的队列。就好象你的电话有一些呼叫在等待，最多可以让 **17** 个呼叫排队等待。（够吓人的！）如果成功该函数返回真，否则返回假。

```
use Socket;
listen(PROTOSOCK, SOMAXCONN)
    or die "cannot set listen queue on PROTOSOCK: $!";
```

参阅 `accpet`，又见第十六章的“套接字”一节。参阅 `listen(2)`。

29.2.87. local

- `local EXPR`

这个操作符并不创建一个局部变量；用 **my** 创建局部变量。它的作用是局部化一个现有变量；也就是说，它令一个或多个全局变量在最内层的闭合块，**eval**，或者文件里拥有局部范围的数值。如果列出的变量多于一个，那么该列表必须放在圆括弧里，因为该操作符比逗号绑定得更紧密。所有列出的变量都必须是合法的左值，也就是说，那些你可以赋值的東西；它可以包含数组或者散列中独立的元素。

这个操作符的工作方法是把声明的变量的当前值保存在一个隐藏的变量里并且在退出该块，子过程，**eval**，或者文件的时候恢复它们。在执行 **local** 之后但推出该范围之前，任何子过程和执行的格式看到的都是这个局部的，内层的数值，而不是以前的那个，外层的数值，因为该变量尽管有一个局部化的数值，仍然是一个全局变量。这种做法的技术术语是“动态范围”。参阅第四章的“范围声明”一节。

如果有必要，你可以给 **EXPR** 赋值，这样就让你可以在局部化变量的同时对它们进行初始化。如果没有给出初始值，那么所有标量都初始化为 **undef**，所有数组和散列都初始化为 **()**。对于普通赋值来说，如果你用圆括弧包围左边的变量（或者如果该变量是一个数组或散列），那么右边的表达式就将在列表环境中进行计算。否则，右边的表达式在标量环境中计算。

在任何情况下，在右边的表达式都是在局部化之前进行计算的，但是初始化是发生在局部化之后的，所以你可以用局部化变量的非局部化值对它进行初始化。比如，下面的代码演示了如何给一个全局数组做一次临时性的修改：

```
if ($sw eq '-v') {
    # 用全局数组初始化局部数组
    local @ARGV = @ARGV;
    unshift @ARGV, 'echo';
    system @ARGV;
}
# @ARGV 在这里恢复原值
```

你还可以临时修改全局散列：

```
# 临时向 %digits 散列增加一对记录
if ($base12) {
    # （注意：我们可没说这么做是高效的！）
```

```

    local(%digits) = (%digits, T => 10, E => 11);
    parse_num();
}

```

你可以使用 **local** 给数组或散列的独立变量赋予临时值，甚至连词法范围的都可以这么干：

```

if ($protected) {
    local $SIG{INT} = 'IGNORE';
    precious();    # 在这个函数期间没有中断
}                # 恢复原来的句柄（如果有的话）

```

你还可以在类型团上使用 **local** 来创建文件句柄而不用装载一大堆对象模块：

```

local *MOTD;          # 保护任何全局的 MOTD 句柄
my $fh = do { local *FH };    # 创建新的间接文件句柄

```

（对于 Perl 5.6 来说，简单的 **my \$fh**；就足够好了，因为如果你在一个需要文件句柄的地方（就象 **open** 或者 **socket** 的第一个参数）给出一个未定义的变量，现在 Perl 将自动为你激活一个新的文件句柄。）

但是一般而言，你通常会希望使用 **my** 而不是 **local**，因为 **local** 不是人们普遍认为的“局部”的意思，参阅 **my**。

29.2.88. localtime

- localtime EXPR
- localtime

这个函数把 **time** 函数返回的值转化为一个九元素的列表，该列表包含的时间对应本地时区时间。它的典型用法如下：

```

# 0      1      2      3      4      5      6      7      8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime;

```

如果象上例一样省略了 **EXPR**，那么它的实际效果是 **localtime(time())**。

所有列表元素都是数字，并且是直接来自 **struct tm** 中得来。（那东西是 C 里的，用不着想它。）因此这就意味着 **\$mon** 的范围是 0 .. 11，而一月是月份 0，而 **\$wday** 的范围是 0 .. 6，星期日是 0。你很容易记住哪些是零为基的，因为那些都是你在包含月份和日期名字的零为基的数组里用做的脚标的东西。

比如，要获取当前时间是星期几：

```

$thisday = (Sun, Mon, Tue, Wed, Thu, Fri, Sat)[(localtime)[6]];

```

\$year 是自 1900 年以来的年数；也就是说，在 2023 年，**\$year** 是 123，而不只是 23。要获取四位数年，只需要用：**\$year + 1900**。要获取两位数年（比如2001里的“01”），用 **sprintf("%2d", \$year % 100)**。

Perl 的库模块 **Time::Local** 包含一个子过程，**timelocal**，它可以做反向的转换。

在标量环境里，**localtime** 返回一个类 **ctime(3)** 的字串。比如，**date(1)** 命令（几乎）（注：**date (1)** 打印时区，但是标量 **localtime** 不能打印时区）可以用下面的命令模拟：

```

perl -le 'print scalar localtime'

```

又见标准 **POSIX** 模块的 **strftime** 函数，该函数有格式化时间更细致的方法。**Time::localtime** 模块提供一个通过名字访问此函数的接口。

29.2.89 lock

- lock THING

`lock` 函数在一个变量，子过程或者 `THING` 引用的对象上加一把锁，直到该锁超出范围。为了保证向下兼容，如果你的 Perl 版本打开了线程编译，并且你说过 `use Threads`，那么该函数只是内建的。否则，Perl 就会假设它是一个用户定义的函数。参阅第十七章，线程。

29.2.90. log

- log EXPR
- log

这个函数返回 `EXPR` 的自然对数（也就是，以 `e` 为底）。如果 `EXPR` 是负数，那么它抛出一个例外。如果要使用其他底数的对数，那么你可以利用基本代数知识：一个数的底数为 `N` 的对数等于该数的自然对数除以 `N` 的自然对数。比如：

```
sub log10 {  
    my $n = shift;  
    return log($n)/log(10);  
}
```

`log` 的逆运算，参阅 `exp`。

29.2.91. lstat

- lstat EXPR
- lstat

这个函数和 Perl 的 `stat` 函数干的事情是一样的（包括设置特殊的 `_` 文件句柄），但如果该文件名字的最后部件是一个符号链接，那么它就 `stat` 这个符号链接本身，而不是 `stat` 哪个符号链接指向的文件。（如果你的系统里没有实现符号链接，那么做的就是一次普通的 `stat` 操作。）

29.2.92. m//

- /PATTERN/
- m/PATTERN/

这是匹配操作符，它把 `PATTERN` 当作一个正则表达式。该操作符是当作一个双引号引起的字符串分析，而不是当作一个函数。参阅第五章，模式匹配。

29.2.93. map

- map BLOCK LIST
- map EXPR, LIST

这个函数为 `LIST` 里的每一个元素计算 `BLOCK` 或者 `EXPR`（在本地把 `$_` 设置为每个元素）并且返回由每次计算的结果组成的列表。它在列表环境里计算 `BLOCK` 或者 `EXPR`，因此 `LIST` 里的每个元素都可以映射成零个，一个，或者更多个返回值里的元素。这些东西都填充到一个列表中去了。比如：

```
@words = map { split ' ' } @lines;
```

把一个行组成的列表分裂成一个单词列表。但是通常在输入值和输出值之间是一一映射的：

```
@chars = map chr, @nums;
```

把一个数字列表转换成对应的字符。而下面是一个一对二映射的例子：

```
%hash = map { genkey($_) => $_ } @array;
```

它只是下面程序的一种有趣的写法：

```
%hash = ();
foreach $_ (@array) {
    $hash{genkey($_)} = $_;
}
```

因为 `$_` 是一个到该列表数值的别名（隐含引用），所以这个变量可以用于修改数组的元素。这样做是允许并且是有用的，不过，如果 `LIST` 不是一个命名数组，那么它可能导致非常奇怪的后果。这种情况下使用一个普通的 `foreach` 循环可能更清晰一些。又见 `grep`；`map` 和 `grep` 的不同在于：`map` 返回一个由所有成功计算 `EXPR` 后得到的结果组成的列表，而 `grep` 返回一个由所有对 `EXPR` 计算为真的 `LIST` 的值组成的列表。

29.2.94. mkdir

- `mkdir FILENAME, MASK`
- `mkdir FILENAME`

这个函数创建 `FILENAME` 声明的目录，赋予它的权限是数字 `MASK` 被当前的 `umask` 修改后得到的数字。如果该操作成功，返回真；否则返回假。

如果省略 `MASK`，那么就假定掩码为 `0777`，几乎就是大多数情况下你需要的掩码。通常，用一个权限比较宽松的 `MASK`（比如 `0777`）创建目录然后让用户用它们的 `umask` 修改该值要比使用一个权限严格的 `MASK` 而不给用户放松权限的机会要好。如果文件或者目录会保存私人的文件（比如邮件文件）时是个例外。

如果 `mkdir(2)` 系统调用不是你的 C 库内建的东西，那么 Perl 通过为每个目录调用 `mkdir(1)` 程序模拟它。如果你在这样的系统上创建一长串目录，那么你自己调用 `mkdir` 程序创建这些目录要比启动无数子进程要高效得多。

29.2.95. msgctl

- `msgctl ID, CMD, ARG`

这个函数调用 System V IPC `msgctl(2)` 系统调用；参阅 `msgctl(2)` 获取更多细节。你可能需要先 `use IPC::SysV` 以获得正确的常量定义。如果 `CMD` 是 `IPC_STAT`，那么 `ARG` 必须是一个变量，它将保存返回的 `msqid_ds` C 结构。返回值类似 `ioctl` 和 `fcntl`：undef 是出错，“0 but true” 是零，否则就是实际返回的数值。

这个函数只有在那些支持 System V IPC 的系统上才能用，好象要远比支持套接字的系统少。

29.2.96. msgget

- `msgget KEY, FLAGS`

这个函数调用 System V IPC `msgget(2)` 系统调用。参阅 `msgget(2)` 获取细节。该函数返回消息队列 ID，如果有错误则返回 `undef`。在调用它之前，你应该 `use IPC::SysV`。

这个函数只有在那些支持 System V IPC 的系统上才有。

29.2.97. msgrcv

- `msgrcv` ID, VAR, SIZE, TYPE, FLAGS

这个函数调用 `msgrcv(2)` 系统调用从消息队列 ID 接收消息到 VAR 变量，最大消息尺寸是 SIZE。参阅 `msgrcv(2)` 获取细节。如果收到一条消息，其消息类型将会是 VAR 里的第一个东西，而 VAR 的最大长度是 SIZE 加上消息类型的尺寸。该函数在成功时返回真，如果有错误则返回假。在调用前，你应该 use IPC:SysV。

这个函数只有在那些支持 System V IPC 的系统上才有。

29.2.98. `msgsnd`

- `msgsnd` ID, MSG, FLAGS

这个函数调用 `msgsnd(2)` 系统调用向消息队列 ID 发送消息 MSG。参阅 `msgsnd(2)` 获取细节。MSG 必须带有长整型消息类型。你可以用下面方法创建一条消息：

```
$msg = pack "L a*", $type, $text_of_message;
```

此函数成功时返回真，如果有错误返回假。在调用前，use IPC::SysV。

这个函数只有在那些支持 System V IPC 的系统上才有。

29.2.99. `my`

- `my` TYPE EXPR : ATTRIBUTES
- `my` EXPR : ATTRIBUTES
- `my` TYPE EXPR
- `my` EXPR

这个操作符声明一个或多个私有的变量，这些变量只存在于最内层的闭合块，子过程，`eval`，或者文件里。如果列出了多于一个变量，那么该列表必须放在圆括弧里，因为该操作符比逗号的捆绑力更强。只有简单标量或者完整的数组和散列可以这样声明。

变量名不能用包名字修饰，因为包变量都是可以通过它们的对应符号表进行全局访问的，而词法变量与任何符号表都无关。和 `local` 不同的是，这个操作符与全局变量没有任何关系，它只是在它自己的范围里（也就是私有变量存在的范围里）隐藏任何其他同名变量，使那些变量不可见。不过，全局变量总是可以通过加了包修饰的名字，或者通过一个符号引用进行访问。

一个私有变量的范围直到它的定义之后才开始。然后该变量的范围就扩展到从那往后的闭合块，直到该变量自己的范围结束。

不过，这就意味着你从一个私有变量的范围里调用的子过程无法看到这个私有变量，除非定义子过程的块本身也是原文包括在那个变量的范围之内。听起来有点复杂，但是只要你碰到一回就不再复杂了。这种情况的技术术语是词法范围，所以我们常把这些叫做词法变量。在 C 文化中，它们有时候叫做“自动”变量，因为它们在范围的入口和出口自动分配和删除。

如果需要，你可以给 EXPR 赋值，这样你就可以初始化你的词法变量。（如果没有给出初始化代码，所有标量都初始化成未定义值而所有数组和散列初始化为空列表。）和普通赋值一样，如果你在左边使用圆括弧（或者如果该变量是一个数组或者散列），那么在右边的表达式就会在列表环境中计算。否则，在右边的表达式则在标量环境中计算。比如，你可以用一个列表赋值给你的正式的子过程参数命名，象这样：

```
my ($friends, $romans, $countrymen) = @_;
```

但是要小心不要省略那个标识列表赋值的圆括弧，象这样：


```
my $countrymen = @_;      # 对还是错？
```

这个赋值把数组的长度（也就是子过程的参数个数）赋予了变量，因为该数组是在标量环境中计算的。不过，只要你使用 **shift** 操作符，你还是可以从使用标量赋值给正式参数赋值中获益的。实际上，因为对象方法把对象当做第一个参数传递，许多法子过程是通过“偷取”第一个参数开始的：

```
sub simple_as {
    my $self = shift;      # 标量赋值
    my ($a, $b, $c) = @_;  # 列表赋值
    ...
}
```

如果你试图用 **my sub** 声明一个词法范围的子过程，那么 **Perl** 会带着一一条说它还没有实现这个特性的信息退出。（当然，除非这个特性已经实现了。）

TYPE 和 **ATTRIBUTES** 都是可选的，同时它们也被认为上实验性的特性。下面是使用它们的声明可能的样子：

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

如果声明了 **TYPE**，那么它表明在 **EXPR** 里声明的是什么类型的标量，用的方法可能是直接的一个或多个标量变量，或者间接地通过一个数组或者散列。如果 **TYPE** 是该类的名字，那么这些标量就会被认为包含指向该类型对象的引用，或者是指向与该类型兼容的对象的引用。要说明的是，派生的类被认为是兼容的。也就是说，假设 **Collie** 是从 **Dog** 派生出来的，你可能声明：

```
my Dog $lassie = new Collie;
```

你的声明说的是，你将把 **\$lassie** 对象一致地用做一个 **Dog** 对象。虽然它实际上是 **Collie** 对象，但是只要你只拿它做 **Dog** 的事情，那么就没什么问题。通过虚拟方法的作用，那些 **Dog** 方法的实现会好端端地在 **Collie** 类里，但是上面的声明只是谈谈接口，而不是实现。至少理论上如此。

有趣的是，直到 **5.6.0** 为止，**Perl** 唯一注意 **TYPE** 声明的地方就是在对应的类里有用 **use fields** 用法声明的域。这些声明在一起就允许一个类的伪散列实现可以“展现”给类外部的代码，因此该散列查找可以由编译器优化成数组的查找。从某种意义上来说，伪散列是这种类型的接口，因此如果允许稍微扁一点，仍然没有触动我们的理论。有关伪散列的更多信息，参阅第八章，引用，中的“伪散列”。

将来，其他类可能会把 **TYPE** 解释成不同的东西。我们应该把 **TYPE** 声明当作一种通用的类接口，有朝一日它可以根据类的不同以各种形式出现。实际上，**TYPE** 甚至可以不是一个正式的类型名字。我们为 **Perl** 保留了小写的类型名字，是因为我们想出来的扩展类型接口的一个办法是允许可选的低层类型声明，比如 **int**，**num**，**str**，和 **ref**。这些声明不是为了把 **Perl** 变成强类型语言；而是为了给编译器一些优化提示，告诉它可以假设该变量的存储在大多数时候都是声明的类型。而标量的语意很大程度上仍然是原来的语意——你仍然可以拿两个 **str** 标量相加，或者打印一个 **int** 标量，就和你熟悉的多态的标量是一样的。但是如果有 **int** 声明，**Perl** 可能会决定只存储整数值而不再捕获结果串当作当前值。使用 **int** 的循环变量的循环可能运行得更快些，尤其是在那些编译成 **C** 的代码里。而且，数字数组可以更紧凑地存储。不过，它也有其有缺陷的一面，如果我们可以写象下面这样的声明的时候，内建的 **vec** 函数甚至可能都会过时：

```
my bit @bitstring;
```

ATTRIBUTES 声明甚至更是实验性的。我们除了保留该语法和做内部接口的原形以外，还没干什么事情；参阅第三十一章里的 **use attributes** 用法获取更多信息。我们将实现的第一个属性可能是 **constant**：

```
my num $PI : constant = atan2(1,1) * 4;
```


不过还有许多其他的可能性，比如为数组和散列建立缺省值，或者令变量在相互合作的解释器之间共享。和类型接口类似，属性接口也应该被认为是一个通用接口，一种可以发明新的语法和语意的工作台。我们不知道 Perl 在下一个十年会怎样进化。我们只知道我们可以通过预先的规划让它变得对我们更简单些。

又见 `local`，`our`，和第四章里的“范围声明”。

29.2.100. new

- `new CLASSNAME LIST`
- `new CLASSNAME`

实际上没有内建的 `new` 函数。它只是一个普通的构造器方法（也就是说，一个用户定义的子过程），由 `CLASSNAME` 类（也就是它的包）定义或者继承过来，好让你构造类型为 `CLASSNAME` 的对象。许多构造器是用“`new`”命名的，但只是习惯，这样才好诱导 C++ 程序员明白它们在干什么事情。要记住阅读有问题的类的文档，这样你就知道如何调用它的构造器；比如，在 Tk 窗口集中构造列表窗口的构造器就是调用 `Listbox()`。参阅第十二章。

29.2.101. next

- `next LABEL`
- `next`

`next` 操作符类似 C 里的 `continue` 语句：它启动 LABEL 指明的下一圈循环：

```
LINE: while () {
    next LINE if /^#/;      # 抛弃注释
    ...
}
```

如果有一个此例中有一个 `continue` 块，那么它会马上在调用 `next` 之后执行。如果省略了 LABEL，该操作符指向闭合循环的最内层。

一个块本身在语意上等效于一个只执行一次的循环。因此，`next` 将提前退出这样的块（通过 `continue` 块，如果有的话）。

`next` 不能用于退出有返回值的块，比如 `eval {}`，`sub {}`，或者 `do {}`，并且不能用于退出一个 `grep` 或者 `map` 操作。如果打开了警告，Perl 将警告你。如果你 `next` 出一个不在你当前的词法范围的循环外边，比如一个调用你的子过程的循环什么的。参阅第四章里的“循环语句”一节。

29.2.102. no

- `no MODULE LIST`

参阅 `use` 操作符，它是 `no` 的反向操作符。大多数标准模块不会逆输入，把 `no` 当作一个空操作，不过这正是它想要的。这个时候用法模块更具强制性。如果找不到 MODULE，那么抛出一个例外。

Revision: r1.1 - 14 Jan 2006 - 14:32 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [Reference](#) > [PerReference_Functions1](#)

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](#)