1 NAME

perlopentut – Perl 中 open 的指南

2 DESCRIPTION

Perl 有两种简单的内建的方法打开文件：简便的 shell 方法和精确的 C 方法。 shell 的方法又分为两个参数的形式和三个参数的形式，这两种形式对于文件名 的处理有不同的语义。你可以自由选择。

3 Open à la shell

Perl 的 open 函数是模仿 shell 工作时的命令行重定向来设计的。这是 shell 中的一些基本例子：

```
$ myprogram file1 file2 file3
$ myprogram  <  inputfile
$ myprogram  >  outputfile
$ myprogram  >> outputfile
$ myprogram  |  otherprogram
$ otherprogram |  myprogram
```

这是一些高级的例子：

```
$ otherprogram     | myprogram f1 – f2
$ otherprogram 2>&1 | myprogram –
$ myprogram    <&3
$ myprogram    >&4
```

习惯构建上面这些的程序员可以很容易看到 Perl 使用几乎和 shell 相同的语 法支持这些类似的结构。

简单的 Open

open 函数使用两个参数：一个是文件句柄，另一个是描述怎样打开和要打 开什么的字符串。当成功时，open 返回值为真；当失败时，返回值为假，并 且设置特殊变量 $! 来反映系统错误。如果第一个参数即文件句柄已经打开 过，open 会先把它关掉。

例如：

```
open(INFO,    "datafile") || die("can't open datafile: $!");
open(INFO,  "<  datafile") || die("can't open datafile: $!");
open(RESULTS,">  runstats") || die("can't open runstats: $!");
open(LOG,   ">> logfile ") || die("can't open logfile:  $!");
```

如果你喜欢少用标点，可以这样写：

```
open INFO,  "<  datafile"  or die "can't open datafile: $!";
open RESULTS,">  runstats"  or die "can't open runstats: $!";
open LOG,   ">> logfile "  or die "can't open logfile:  $!";
```

有几点需要注意。一是，开头的小于号不是必须的，如果省略，Perl 假定你要 以可读的方式打开文件。

二是在第一个例子中使用 || 逻辑运算符，而第二个例子中使用优先级更低 的 or。如果在后一个例子中使用 || 则等效于：

```
open INFO, ( "<  datafile"  || die "can't open datafile: $!" );
```

这肯定不是你想要的。

另一个值得注意的是，和在 shell 中一样，文件名前面或者后面的空格都被忽略了。这很好，因为你不愿意这些会有不同吧：

```
open INFO,   "<datafile"
open INFO,   "< datafile"
open INFO,   "<  datafile"
```

当你从另一个文件中读取文件名，却在打开之前忘记去除两端的空格，忽略文件名两端的空格可以避免这种情况发生：

```
$filename = <INFO>;         # oops, \n still there
open(EXTRA, "< $filename") || die "can't open $filename: $!";
```

这不是一个 bug，而是一个特性。因为 open 模仿 shell 使用重定向箭头指定怎样打开文件的风格，所以也应当同样忽略文件名两端的空格。对于访问文件名不符合规定的文件，参见 §??。

也有三个参数版本的 open，这可以让你加入特殊的重定向字符到参数中：

```
open( INFO, ">", $datafile ) || die "Can't create $datafile: $!";
```

在这种情况下，打开的文件名是 $datafile 这个字符串。这样你不用担心 $datafile 中含有影响打开模式的字符，或者在两个参数版本中忽略文件名两端的空格。而且不进行不必要的字符串内插。

间接文件句柄

open 的第一个参数可以是一个文件句柄的引用。在 Perl 5.6.0 中，如果参数没有初始化，Perl 会自动创建一个文件句柄，然后存储在第一个参数中，例如：

```
open( my $in, $infile )   or die "Couldn't read $infile: $!";
while ( <$in> ) {
   # do something with $_
}
close $in;
```

间接文件句柄可以使名字空间的管理更简单。由于文件句柄对于当前包是全局的，两个函数试图同时打开 INFILE 会导致冲突。如果两个函数使用间接文件句柄比如 my $infile，则不会发生冲突，也不用担心以后会发生冲突。

还有一个方便之处是当运行到作用域外或者使用 undefine 时，间接文件句柄会自动关闭。

```
sub firstline {
    open( my $in, shift ) && return scalar <$in>;
    # no close() required
}
```

打开管道

在 C 中，当你想用标准 I/O 库打开一个文件时，你要用 fopen 函数，当你 想打开一个管道时，你要用 popen 函数。但是在 shell 中，你只是用不同 的重定向字符。对于 Perl 也是如此。open 的调用方法一样，只是参数不同了。

如果以管道符号开头，open 启动一个新的命令，打开一个可写文件句柄通向 这个命令。你可以向这个句柄中写入内容，而你所写的将出现在这个命令的标准 输出里。例如：

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

如果末尾的字符是一个管道时，你启动一个新的命令，打开一个通向这个命令的 可读文件句柄。这使得这个命令写入标准输出的内容可以在你的句柄中读到的。

```
open(NET, "netstat -i -n |")    || die "can't fork netstat: $!";
while (<NET>) { }               # do something with input
close(NET)                      || die "can't close netstat: $!";
```

当你试图打开一个通向或者来自不存在的命令的管道时会发生什么呢？如果可 能，Perl 会检测到命令失败，然后照常设置 $!。但是如果命令中包含特殊 的 shell 字符，例如 > 或者 *，称为“metacharacters”，Perl 不 会直接执行命令。取而代之的是，Perl 运行 shell，它来试图运行命令。这意 味着是 shell 得到错误指示。在这种情况下，如果 Perl 根本不能运行 shell，open 只会指示一个错误。参考 How can I capture STDERR from an external command? in perlfaq8 了解如何应付这个问题。在 perlipc 中也有回答。

如果你想打开双向管道，IPC::Open2 可以处理。参考 Bidirectional Communication with Another Process in perlipc。

负号文件

和标准 shell 实用程序一样，Perl 的 open 函数对于只是一个负号的文件 名的处理方法是特殊的。如果你以可读方式打开负号，实际上是访问标准输入。 如果以可写方式打开，实际上是访问标准输出。

如果负号能用作默认的输入或者输出，当你打开一个通向或者来自负号的管道会 发生什么呢？它会运行什么默认的命令？仅仅是和你现在运行的脚本一样！事实 上在 open 背后秘密进行了一次 fork。更详细内容请参考 Safe Pipe Opens in perlipc。

交叉读写

同时指定读和写的访问是可以的。你只要在重定向符号前加上一个“+”号。但是同 在 shell 中一样，对一个文件使用 less-than 决不会创建一个新文件，它仅仅 是打开一个已经存在的文件。另一方面，使用 greater-than 总是破坏（截短长度为 0）已经存在的文件，或者如果没有旧文件的话，创建一个新文件。 Adding a "+" for read-write doesn't affect whether it only works on existing files or always clobbers existing ones.

```
open(WTMP, "+< /usr/adm/wtmp")
    || die "can't open /usr/adm/wtmp: $!";

open(SCREEN, "+> lkscreen")
    || die "can't open lkscreen: $!";

open(LOGFILE, "+>> /var/log/applog"
    || die "can't open /var/log/applog: $!";
```

第一个不会创建一个新文件，第二总是破坏旧文件。第三个在必要时总会创建一 个新文件而且不会破坏旧文件，并允许你读取文件任何中的位置，但是写入的内 容总是放到文件末尾。简短的说，第一个比第二个和第三个都要常用，而后者几 乎总是错的。（*不是太肯定*）（如果你了解 C，Perl 的 open 中加号是从 C 的 fopen(3S) 中演化而来的，它也是最终调用这个函数。）

事实上，当要更新文件时，除非是像上面 WTMP 的例子操作二进制文件，你很可 能不会用这种方法来更新。而是使用 Perl 的 -i 选项。下面这个命令将所 有 C、C++ 和 yacc 源文件或者头文件中的 foo 替换成 bar，并把旧版本文件 名加上 ".orig" 后缀：

```
$ perl -i.orig -pe 's/\bfoo\b/bar/g' *.[Cchy]
```

这是更新文本文件的最好方法。更详细内容，请参考 perlfaq5 中的第二个问题。 This is a short cut for some renaming games that are really the best way to update textfiles.

过滤器

open 的一个最普遍的用途可能是你根本没有注意到的。当你使用 <ARGV> 处理 ARGV 文件句柄时，Perl 实际上是隐式的对 @ARGV 使用 open。 所以像这样调用一个程序：

```
$ myprogram file1 file2 file3
```

能够打开所有的文件，通过这样一个简单的结构一个一个的处理：

```
while (<>) {
   # do something with $_
}
```

如果在循环开始时 @ARGV 还是空列表，Perl 假定你打开一个负号，也就是标准 输入。事实上，在处理 <ARGV> 时作为当前文件的 $ARGV 在这种情况下 就是设置为 "-"。

你最好在开始循环之前对 @ARGV 进行预处理来确保这是你所想要的。一个理由 是去掉以负号开头的命令行选项。尽管手工操作也很简单，但是最好使用 Getopts 模块：

```
use Getopt::Std;

# -v, -D, -o ARG, sets $opt_v, $opt_D, $opt_o
getopts("vDo:");

# -v, -D, -o ARG, sets $args{v}, $args{D}, $args{o}
getopts("vDo:", \%args);
```

或者使用标准模块中的 Getopt::Long 来允许命名参数：

```
use Getopt::Long;
GetOptions( "verbose"  => \$verbose,        # --verbose
            "Debug"    => \$debug,          # --Debug
            "output=s" => \$output );
       # --output=somestring or --output somestring
```

另一个预处理参数的原因是使空的参数列表变成默认为全部文件：

```
@ARGV = glob("*") unless @ARGV;
```

你甚至可以过滤后只剩下纯文本文件。This is a bit silent, of course, and you might prefer to mention them on the way.

```
@ARGV = grep { -f && -T } @ARGV;
```

如果你使用 -n 或者 -p 命令行选项，你应当在 BEGIN{} 块中对 @ARGV 进行修改。

记住一个普通的 open 有特殊的性质，它可能调用 fopen(3S) 也可能调用 popen(3S)，取决于调用的参数。这也是人们称它为“神奇的 open”的原因。这里有一个例子：

```
$pwdinfo = `domainname` =~ /^(\(none\))?$/
            ? '< /etc/passwd'
            : 'ypcat passwd |';

open(PWD, $pwdinfo)
        or die "can't open $pwdinfo: $!";
```

这使得 open 也充当了过滤器的角色。因为 <ARGV> 的处理包含了通常的 shell 风格的 Perl open，它包含了我们已经说到的所有特殊的情况：This sort of thing also comes into play in filter processing. Because <ARGV> processing employs the normal, shell-style Perl open, it respects all the special things we've already seen:

```
$ myprogram f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

这个程序将从文件 f1，进程 cmd1，标准输入（在这个例子中是 tmpfile），文件 f2，命令 cmd2，最后是文件 f3 中读入。

对，这意味着如果你的目录下有一个名为“-”（或者其它类似名字）的文件，open 不会把它们当做合法的文件处理。你需要用“./-”，就像使用 rm 程序一样，或者你可以使用下面将提到的 sysopen。

一个更有趣的应用是将文件或者某个名字转换成管道。例如，用 gzip 自动处理 gzipped 或者压缩的文件：

```
@ARGV = map { /^\.(gz|Z)$/ ? "gzip -dc $_ |" : $_ } @ARGV;
```

或者，如果你在安装 LWP 之后有 GET 程序，你可以在通过 URL 获取之后进行处理：

```
@ARGV = map { m#^\w+://# ? "GET $_ |" : $_ } @ARGV;
```

所以称为魔术 <ARGV> 不是平白无故的，很贴切，不是吗？

4 Open à la C

如果你想象 shell 那样简便，Perl 的 open 无疑是一种方法。另一方面，如果你想要比 C 中简单的 fopen(3S) 更精确，那么应该看看 Perl 的 sysopen。这是会直接进行系统调用 open(2)。确实意味着牵涉了更多，这是精确性的代价。That does mean it's a bit more involved, but that's the price of precision.

sysopen 有 3（或者 4）个参数：

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

HANDLE 参数是像 open 中一样文件句柄。PATH 是一个合法的路径，这是不管 greater-than 或者 less-thans 或者管道或者负号，也不忽略空格。如果参数中有这些符号，将作为路径的一部分。FLAG 参数包含一个或者多个从 Fcntl 模块中衍生出来的值，它们用位运算符“|”连接起来。最后一个参数 MASK 是可选的，如果使用的话，它和用户当前的 umask 进行与操作来决定创建的文件的模式。你应当通常都忽略这一项。

虽然传统上只读、只写和读写的值分别为 0、1 和 2，但是在有些系统上不是这样。所以最好从 Fcntl 模块中导入相应的常数。它提供了下面的标准 flag：

```
O_RDONLY        Read only
O_WRONLY        Write only
O_RDWR          Read and write
O_CREAT         Create the file if it doesn't exist
O_EXCL          Fail if the file already exists
O_APPEND        Append to the file
O_TRUNC         Truncate the file
O_NONBLOCK      Non-blocking access
```

有时在某些操作系统中有一些不常用的 flag，包括 O_BINARY、O_TEXT、 O_SHLOCK、O_EXLOCK、O_DEFER、O_SYNC、O_ASYNC、O_DSYNC、O_RSYNC、O_NOCTTY、O_NDELAY 和 O_LARGEFILE。详细情况请查看你的 open(2) 手册页或者相应文档（注意：从 Perl 5.6 开始如果允许的话，O_LARGEFILE flag 自动加入到 sysopen() 的 flag 中，因为默认是大文件）。

这是怎样用 sysopen 来模拟前面简单的 open 调用。为了简洁，我们省略了 || die $! 检验，但是在真正用时你要总是检查返回值。这不是完全相同的，因为 open 会去掉两端的空格，但是你应该明白它的意思。

为了打开一个只读的文件：

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

打开一个只写的文件，如果有需要的话，创建一个新文件，否则清空旧文件：

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

打开一个文件，往文件内追加内容，如果有需要的话，创建一个新文件：

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

打开一个文件更新内容，这个文件必须已经存在：

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

这还有一些不能用常规的 open 而可以由 sysopen 来完成的事情。你将看到这是通过控制第三个参数 flag 实现。 As you'll see, it's just a matter of controlling the flags in the third argument.

打开一个只写文件，创建一个新文件而且这个文件先前一定不能存在：

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

打开一个文件进行追加内容，这个文件先前一定是已经存在的：

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

打开一个文件进行更新，如果有必要创建一个新文件：

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

打开一个文件进行更新，这个文件先前一定是已经存在的：

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

以无阻塞方式打开文件，如果在必要创建一个新文件：

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

Permissions à la mode

如果你忽略了 sysopen 的 MASK 参数，Perl 使用八进制数值 0666。对于可 执行程序和目录，通常 MASK 值是 0777，其它是 0666。

为什么这样宽松呢？事实上不是这样的。MASK 将用你当前进程的 umask 修 饰。umask 是一个代表 disabled 的权限位的数字，也就是由创建的文件中 不允许打开的权限位组成的。

例如，如果你的 umask 是 027，020 部分将使用户组不能写，007 部分将使 其它用户不能读、写和执行。在这些条件下，传递 0666 给 sysopen 将创建 一个模式为 0640 的文件，因为 0666 & -027 是 0640。

你应该尽量不用 sysopen 的 MASK 参数。因为这剥夺了你的用户选择新文件 权限的自由了。但是对于敏感或者隐私的数据是一个例外，例如邮件目录、 cookie 文件和内部的临时文件。

5 Obscure Open Tricks

Re-Opening Files (dups)

有时当你有一个打开的文件句柄要，你想使另一个句柄是前一个的拷贝。在 shell 中，我们在文件描述符前加一个与符号（‘&’）来重定向。例如， 2>&1 使描述符 2（在 Perl 中是标准错误输出）重定向到描述符 1（在 Perl 中通常是标准输出）。这在 Perl 中也是相同的，以与符号开头的文件名，如 果是数字将当作文件描述符，如果是字符串将当作文件句柄。

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

这意味着，如果一个函数需要一个文件名，但是由于你已经打开了这个文件，不 想传递一个文件名给它，你只要传递一个以与符号开头的文件句柄。最好用全称 形式的句柄，防止这个函数是在不同的包中：

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

像这样，如果 somefunction() 将要 open 它的参数，它会使用已经打开的句柄。 这与传递一个句柄是不同的，因为你如果传递一个句柄，你根本不 open 这个文 件，而这里你确实给 open 传递了参数。

如果你有一个 I/O 对象（这是 C++ 程序员们强烈要求的，诡异新奇的），这就 不能正常工作了，因为在对于原来 Perl 看来这不是一个文件句柄。你要使用 fileno() 来得到它的描述符，假如可以：

```
use IO::Socket;
$handle = IO::Socket::INET->new("www.perl.com:80");
$fd = $handle->fileno;
somefunction("&$fd");  # not an indirect function call
```

如果就用真正的文件句柄会更简单一些（而且显然会更快）：

```
use IO::Socket;
local *REMOTE = IO::Socket::INET->new("www.perl.com:80");
die "can't connect" unless defined(fileno(REMOTE));
somefunction("&main::REMOTE");
```

如果在文件句柄或者描述符之前不是“&”而是“&=”，Perl 将不会对这个相同的文 件句柄用系统调用 dup(2) 创建一个新的描述符，而是使用 fdopen(3S) 库调用 产生一个对已经存在的文件句柄的别名。这会减少一些系统资源，尽管现在比较 少关注这个方面。这是一个例子：

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

如果你使用魔术 <ARGV>，你也可以在命令行参数中传递类似于 "<&=$MHCONTEXTFD 的参数给 @ARGV，但是从来没有见过有人这样做。

Dispelling the Dweomer

Perl 比 Java 等其它语言更像一个 DWIMmer，DWIM 是“按我的意思去做（do what I mean）”的缩写。而且比你想象的还要有魔力。Perl 是具有强大的魔法（dweomer，这是一个生僻词，意思是魔法）。有时，Perl 的 DWIMmer 是像 dweomer 一样令人感到舒服。 Perl is more of a DWIMmer language than something like Java--where DWIM is an acronym for "do what I mean". But this principle sometimes leads to more hidden magic than one knows what to do with. In this way, Perl is also filled with dweomer, an obscure word meaning an enchantment. Sometimes, Perl's DWIMmer is just too much like dweomer for comfort.

如果魔术 open 对你来说太神奇了，你也不必因此使用 sysopen。要打开一个文件名中有任意奇怪字符的文件，有必要保护好开头和结尾的空格。开头的空格可以通过在文件名前插入 "./"，末尾的空格可以通过在字符串末尾添加一个 ASCII NUL 字节。

```
$file =~ s#^(\s)#./$1#;
open(FH, "< $file\0")  || die "can't open $file: $!";
```

当然，这里假定你的系统点是当前工作目录，斜线是目录分隔符，而且在合法的文件名中不允许出现 ASCII NUL。许多系统是遵守这些约定的，包括所有的 POSIX 系统，也包括个人的 Microsoft 系统。唯一对这种方式结果不确定的流行系统是“Classic” Macintosh 系统，它使用冒号（colon）而不是斜线。 所以 sysopen 不是一个坏主意。

如果你把 <ARGV> 处理成非魔术的形式，你可以这样： If you want to use <ARGV> processing in a totally boring and non-magical way, you could do this first:

```
#  "Sam sat on the ground and put his head in his hands.
#  'I wish I had never come here, and I don't want to see
#  no more magic,' he said, and fell silent."
for (@ARGV) {
   s#^([^./])#./$1#;
   $_ .= "\0";
}
while (<>) {
   # now process $_
}
```

这样转换之后，用户就不能再用“-”来表示标准输入了，其它标准惯例也不能用了。

用路径名打开

你可能已经注意到 Perl 的 warn 和 die 函数能产生这样的消息：

```
Some warning at scriptname line 29, <FH> line 7.
```

这是因为你打开了一个文件句柄 FH，并从中读取了 7 个记录。但是怎样显示文件名，而不是文件句柄呢？

如果你不是使用 strict refs，或者暂时关闭，你只要像这样做：

```
open($path, "< $path") || die "can't open $path: $!";
while (<$path>) {
    # whatever
}
```

因为你是用文件的路径名作为文件句柄，你将得到这样的警告消息：

```
Some warning at scriptname line 29, </etc/motd> line 7.
```

一个参数的 open

还记得曾经说过 Perl 的 open 作用两个参数吗？那样是不贴切的。你看，它也 能只使用一个参数。当且仅当这个变量是一个全局变量，而不是词法变量，你可 以只传递给 open 一个参数——文件句柄。open 将从有相同名字的全局标量 中获得路径名。

```
$FILE = "/etc/motd";
open FILE or die "can't open $FILE: $!";
while (<FILE>) {
    # whatever
}
```

为什么会这样呢？某些人为了迎合那些 hysterical porpoise。那是 Perl 刚开 始的事了，或者说是以前的事了。

Playing with STDIN and STDOUT

聪明的做法是在程序结束时明确的关闭标准输出：

```
END { close(STDOUT) || die "can't close stdout: $!" }
```

如果你不这样做，并且你的程序由于命令行的重定向使磁盘填满，它在错误退出 时不会向你报告错误。

你也不必一定要用缺省的标准输入和标准输出。如果愿意的话，你可以重定向它 们：

```
open(STDIN, "< datafile")
    || die "can't open datafile: $!";

open(STDOUT, "> output")
    || die "can't open output: $!";
```

然后这些可以直接访问或者传递给子进程。这就使程序看上去像在命令行中使用 重定向来调用一样。

把这些连接到管道中可能更有趣。例如：

```
$pager = $ENV{PAGER} || "(less || more)";
open(STDOUT, "| $pager")
    || die "can't fork a pager: $!";
```

这使得你的程序看上去是以标准输出通过管道指向页显示程序的方式来调用。你 也可以使用这个方法连接到一个隐式的 fork。如果你对你的程序进行后处理的 话，你可以在不同的进程中这样做：

```
head(100);
while (<>) {
    print;
}
```

```
sub head {
    my $lines = shift || 20;
    return if $pid = open(STDOUT, "|-");       # return if parent
    die "cannot fork: $!" unless defined $pid;
    while (<STDIN>) {
        last if --$lines < 0;
        print;
    }
    exit;
}
```

这个技术可以在输出流上放置任意数量的过滤器。

6 Other I/O Issues

这些内容和 open 和 sysopen 没有关系，但是它们影响你对打开文件的 操作。

打开非文件的文件

什么时候一个文件不是一个文件呢？你可以说这个文件存在，但是不是一个 plain 文件。只是以防万一，我们先要检查它是不是一个符号链接：

```
if (-l $file || ! -f _) {
    print "$file is not a plain file\n";
}
```

还有什么类型的文件呢？目录、符号链接、命名管道、Unix-domain 套接字、 block 和 character device。这些都是文件，只不过不是 plain 文件。 plain 文件和文本文件是相同的概念。不是所有的文本文件都是 plain 文件， 不是所有的 plain 文件都是文本文件。这也是为什么有 -f 和 -T 文件 测试。

为了打开目录，你应当用 opendir 函数，然后用 readdir 来处理，如果 有必要的话，恢复目录名：

```
opendir(DIR, $dirname) or die "can't opendir $dirname: $!";
while (defined($file = readdir(DIR))) {
    # do something with "$dirname/$file"
}
closedir(DIR);
```

如果你要递归的处理目录，最好使用 File::Find 模块。例如，这是递归地输出 所有文件，如果文件是一个目录的话，输出一个斜线：

```
@ARGV = qw(.) unless @ARGV;
use File::Find;
find sub { print $File::Find::name, -d && '/', "\n" }, @ARGV;
```

这是找出在一个特定目录下所有的错误符号链接：

```
find sub { print "$File::Find::name\n" if -l && !-e }, $dir;
```

正如你所看到的那样，如果是一个符号链接，你可以把它当作它指向的文件。或 者，如果你想知道它指向什么，可以调用 readlink：

```
if (-l $file) {
    if (defined($whither = readlink($file))) {
        print "$file points to $whither\n";
    } else {
        print "$file points nowhere: $!\n";
    }
}
```

打开命名管道

命名管道是另一种类型。你可以把它当作普通文件，但是只有同时有读方（reader）和写方（writer）存在时，它才不会阻塞。更多内容参见 Named Pipes in perlipc。Unix-domain 套接字也相当不同，在 Unix-Domain TCP Clients and Servers in perlipc 中有描述。

当你要打开设备时，有容易的方法，也有窍门。假定你要打开一个块设备，你知道你在做什么。字符设备更有趣。它们通常用于调制解调器、鼠标和某些打字机。参见 How do I read and write the serial port? in perlfaq8 。打开它们时要特别小心：

```
sysopen(TTYIN, "/dev/ttyS1", O_RDWR | O_NDELAY | O_NOCTTY)
        # (O_NOCTTY no longer needed on POSIX systems)
    or die "can't open /dev/ttyS1: $!";
open(TTYOUT, "+>&TTYIN")
    or die "can't dup TTYIN: $!";

$ofh = select(TTYOUT); $| = 1; select($ofh);

print TTYOUT "+++at\015";
$answer = <TTYIN>;
```

对于还没有用 c<sysopen> 打开的描述符，例如套接字，你可以用 `fcntl` 将它们设置成非阻塞的：

```
use Fcntl;
my $old_flags = fcntl($handle, F_GETFL, 0)
    or die "can't get flags: $!";
fcntl($handle, F_SETFL, $old_flags | O_NONBLOCK)
    or die "can't set non blocking: $!";
```

如果你打算操作 tty，为了避免在曲折的 `ioctl` 及其它不相同的东西中迷路，如果有 stty 程序最好调用这个程序，否则使用 POSIX 接口。要弄清楚这些，你需要阅读 termios(3) 手册页，它描述了 tty 的 POSIX 界面，还有 POSIX，它描述了 Perl 的 POSIX 接口。另外还有一些高层的 CPAN 模块能够帮助你。可以试试 Term::ReadKey 和 Term::ReadLine。

打开套接字

还有什么可以打开呢？使用套接字打开一个连接，你不会使用 Perl 的两个 open 函数。参考 Sockets: Client/Server Communication in perlipc。这里有一个例子。有了这个，你就可以把 FH 当作双向文件句柄了：

```
use IO::Socket;
local *FH = IO::Socket::INET->new("www.perl.com:80");
```

如果要打开一个 URL，CPAN 的 LWP 模块是首选。它没有文件句柄接口，但是仍然可以很方便的得到文档的内容：

```
use LWP::Simple;
$doc = get('http://www.linpro.no/lwp/');
```

二进制文件

在一些古董级的系统中，它们有，说得好听的，被称为极端费解的（有些人会说是坏掉的）I/O 模型。一个文件不是一个文件——至少不符合 C 标准 I/O 库。在这些老式系统中，库（但不是内核）会区分文本和二进制流。为了使文件显得正常，你花很大力气去避免讨厌的问题。在这些倒霉的系统中，套接字和管道已经用二进制模式打开，而且现在没有办法把它们关掉。但是对于文件，你还是有更多选择。

另一种选择是在进行常规 I/O 操作之前对相应的句柄使用 `binmode` 函数：

```
binmode(STDIN);
binmode(STDOUT);
while (<STDIN>) { print }
```

只要系统支持，向 `sysopen` 传递非标准的 flag 选项也可以用二进制模式打开文件。这等效于以正常方式打开文件之后，对文件句柄使用 `binmode`：

```
sysopen(BINDAT, "records.data", O_RDWR | O_BINARY)
    || die "can't open records.data: $!";
```

现在你可以对文件句柄用 `read` 和 `print`，而不用担心在非标准系统中 I/O 库破坏你的数据。这虽然不漂亮，但是在老式系统上很少是漂亮的。CP/M 将与我们同在，直到世界末日，并将延续下去。

在一些有特殊的 I/O 体系的系统，你将惊奇的发现，即使是使用 `sysread` 和 `syswrite` 进行不缓冲的 I/O 也可能背着你偷偷的破坏数据：

```
while (sysread(WHENCE, $buf, 1024)) {
    syswrite(WHITHER, $buf, length($buf));
}
```

取决于你使用的系统，即使是这些调用也需要先进行 `binmode` 或者 `O_BINARY`。已知的没有这些问题的系统包括 Unix、Mac OS、Plan 9 和 Inferno。

文件锁定

在多任务环境中，你要小心不要和其它对你当前工作文件进行 I/O 操作的进程发生冲突。你需要在读写文件时分别对文件设置共享或者排他锁定。你可以认为只有排他锁定。

永远不要用文件的存在 `-e $file` 作为锁定文件的指示，在测试文件存在和创建文件过程之间存在竞态条件（race condition）。很可能在你测试文件存在和试图创建文件的一瞬间另一个进程创建了这个文件。原子操作是关键的。Atomicity is critical.

Perl 最容易移植的锁定接口是通过 `flock` 函数。在 SysV 或者 Windows 这些不直接支持的系统 Perl 提供了模拟。这意味着会影响它的工作方式，所以你应当了解 Perl 是怎样在你的系统中实现 `flock` 的。The underlying semantics may affect how it all works, so you should learn how `flock` is implemented on your system's port of Perl.

文件锁定不会阻止其它进程对文件的 I/O 操作。文件的锁定只是阻止其它进程对文件加锁，而不是进行 I/O 操作。因为锁定是建议性的，如果一个进程使用锁定，而另一个不用，规定就被打破了（all bets are off）。

默认情况下，在允许锁定之前 `flock` 调用将被阻塞。如果没有排他的锁定要求共享锁定是将被允许的。如果没有任何类型的锁定，要求排他锁定是将被允许的。锁定是对文件描述符，而不是文件名。你只有打开一个文件后才能锁定文件。一旦关闭文件之后，也就不维持锁定状态了。

这是怎样对文件进行阻塞共享锁定，一般用于文件的读取：

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename")  or die "can't open filename: $!";
flock(FH, LOCK_SH)      or die "can't lock filename: $!";
# now read from FH
```

你可以用 LOCK_NB 进行非阻塞的锁定：

```
flock(FH, LOCK_SH | LOCK_NB)
   or die "can't lock filename: $!";
```

在进行锁定之前，先通过警告来提高用户友好性： This can be useful for producing more user-friendly behaviour by warning if you're going to be blocking:

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename")  or die "can't open filename: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
   $| = 1;
   print "Waiting for lock...";
   flock(FH, LOCK_SH)  or die "can't lock filename: $!";
   print "got it.\n"
}
# now read from FH
```

在进行排他锁定时，一般用于文件写入，你要小心。我们用 sysopen 打开文 件，这样在清空之前进行锁定。你可以用 LOCK_EX | LOCK_NB 进行非阻塞的 锁定。

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
   or die "can't open filename: $!";
flock(FH, LOCK_EX)
   or die "can't lock filename: $!";
truncate(FH, 0)
   or die "can't truncate filename: $!";
# now write to FH
```

最后，由于不能阻止对无用空设备循环调用中增加计数，这是怎样安全的递增一 个文件中的数字。 Finally, due to the uncounted millions who cannot be dissuaded from wasting cycles on useless vanity devices called hit counters, here's how to increment a number in a file safely:

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "numfile", O_RDWR | O_CREAT)
   or die "can't open numfile: $!";
# autoflush FH
$ofh = select(FH); $| = 1; select ($ofh);
flock(FH, LOCK_EX)
   or die "can't write-lock numfile: $!";

$num = <FH> || 0;
seek(FH, 0, 0)
   or die "can't rewind numfile : $!";
print FH $num+1, "\n"
   or die "can't write numfile: $!";
```

```
truncate(FH, tell(FH))
    or die "can't truncate numfile: $!";
close(FH)
    or die "can't close numfile: $!";
```

IO 层

在 Perl 5.8.0 引入了称为“PerlIO”的新 I/O 框架。在 Perl 中的 I/O 使用了 新的“管道（plumbing）”。对于大多数情况仍然能够用以前的方式工作，PerlIO 也引入了一些新的特性，比如从“层”的观点来看 I/O。一个 I/O 层可能不仅仅 是移动数据，还进行数据的转换。这种转换包括压缩和解压缩，加密和解密，在 不同字符编码之间转换。

对于 PerlIO 的特性进行详尽讨论超出本指南的范围，这里是怎样应用 PerlIO 层的例子：

- 可以使用三个（或者更多）参数形式的 open，其中第二个参数除了常用的 '<'、'>'、'>>'、'|' 及其变体加入其它东 西，例如：

  ```
  open(my $fh, "<:utf8", $fn);
  ```

- 可以使用两个参数形式的 binmode，例如：

  ```
  binmode($fh, ":encoding(utf16)");
  ```

对于 PerlIO 的更详细讨论请参考 PerlIO； 对于 Unicode 和 I/O 的更详细讨论请参考 perluniintro。

7 SEE ALSO

perlfunc(1) 中的 open 和 sysopen 函数；manpage 中系统 open(2)，dup(2)，fopen(3) 和 fdopen(3)；POSIX 文档。

8 AUTHOR and COPYRIGHT

9 HISTORY

第一次发布: Sat Jan 9 08:09:11 MST 1999

10 TRANSLATORS

YE Wenbin redcandle (redcandle51@chinaren.com)