

## 第十九章，命令行接口

- ↓ 第十九章，命令行接口
  - ↓ 19.1 命令行处理
    - ↓ 19.1.1 在非 Unix 系统上的 #! 和引号
    - ↓ 19.1.2 Perl 的位置
    - ↓ 19.1.3 开关
  - ↓ 19.2 环境变量

本章是帮你在用 Perl 开火之前先将它的方向校准。校准 Perl 的方法很多，但两个最基本的方法是通过命令行开关和通过环境变量。开关是校准某一特定命令的最快速和准确的方法。而环境变量常用于设置通用的策略。

### 19.1 命令行处理

很幸运的是 Perl 是在 Unix 世界里成长起来的，因为那就意味着它的调用语法在其他操作系统的命令行解释器里也能运行得相当好。大多数命令行解释器知道如何把一系列单词当作参数处理，而用不着关心某个参数是否以一个负号开头。当然，如果你从一个系统转到另外一个系统，也有一些乱七八糟的地方会把事情搞糟。比如，你不能在 MS-DOS 里象在 Unix 里那样使用单引号。而且对于象 VMS 这样的系统来说，有些封装代码必须经过一些处理才能模拟 Unix I/O 的重定向。而通配符就解释成通配符。一旦你搞定这些问题，那么 Perl 就能在任何操作系统上非常一致地处理它的开关和参数。

即使你自己并没有一个命令行解释器，你也很容易从用其它语言写的另外一个程序里执行 Perl 程序。调用程序不仅能够用常用的方法传递参数，而且它还可以通过环境变量传递信息，还有，如果你的操作系统支持的话，你还可以通过继承文件描述符来传递信息（参阅第十六章，进程间通讯，里的“传递文件句柄”一节。）甚至一些外来的参数传递机制都可以很容易地在一个模块里封装，然后通过简单的 use 指示引入的你的 Perl 程序里来。

Perl 以标准的风格传递命令行参数。（注：假设你认为 Unix 是标准风格。）也就是说，它预料在命令行上先出现开关（以负号开头的字）。在那之后通常出现脚本的名字，后面跟着任何附加的传递到脚本里的参数。有些附加的参数本身看起来就象开关，不过如果是这样的话，它们必须由脚本本身处理，因为 Perl 一旦看到一个非开关参数，或者特殊的"--"“开关（意思是说：“我是最后一个开关”），它就停止分析开关。

Perl 对你放源代码的地方提供了一些灵活性。对于短小的快速使用的工作，你可以把 Perl 程序全部放在命令行上。对于大型的，比较永久的工作，你可以把 Perl 脚本当作一个独立的文件使用。Perl 按照下面三种方式之一寻找一个脚本编译和运行：

- 通过在命令行上的 -e 开关一行一行地声明。比如：

```
%perl -e "print 'Hello, World.'"
Hello, World.
```

- 包含在命令行声明的第一个文件名的文件里面。系统支持可执行脚本第一行的 #! 符号为你调用解释器的用法。

1. 通过标准输入隐含地传递。这个方法只有在没有文件名参数的情况下才能用；要给一个标准输入脚本传递参数，你必须使用方法 2，为脚本名字明确地声明一个“-”。比如：

```
%echo "print qq(Hello, @ARGV.)" | perl - World
Hello, World.
```

对于方法 2 和 3，Perl 从文件开头开始分析——除非你声明了 `-x` 开关，那样它会寻找第一个以 `#!` 开头并且包含 `"perl"` 的行，然后从那里开始分析。这个开关对于在一个更大的消息里运行一段嵌入的脚本很有用。如果是这样，你可以用 `__END__` 记号指明脚本的结尾。

不管你是否使用了 `-x`，分析 `#!` 行的时候总是要检查是否有开关。这样，如果你所在的平台只允许在 `#!` 行里有一个参数，或者更惨，就根本不把 `#!` 行当作一个特殊的行，你仍然能够获得一致的开关特性，而不用管是如何调用 Perl 的，即使你是用 `-x` 来寻找脚本的开头。

警告：因为老版本的 Unix 不声不响地把内核对 `#!` 行分析时超过 32 个字符的部分截去，最后可能是有些开关原封不动地传给你的程序而其他的参数就没了；如果你不小心，你甚至有可能收到一个 `"-"` 而没有它的字母。你可能会想确保所有你的开关要么在 32 字符之前，要么在其后。大多数开关并不关心它们是否被多次处理，但如果拿到一个 `"-"` 而不是整个开关，就会导致 Perl 试图从标准输入读取它的源代码，而不是从你的脚本里。而且一个 `-I` 开关的片段也会导致很奇怪的结果。不过，有些开关的确关心它们是否被处理了两次，比如 `-l` 和 `-0` 的组合。你要么把所有开关放到 32 字符范围之后（如果可行），要么把 `-0DIGITS` 换成 `BEGIN{ $/ = "\0DIGITS"; }`。当然，如果你用的不是 Unix 平台，那么我们保证不会有这种问题发生。

对 `#!` 行的开关的分析从该行中首次出现 `"perl"` 的地方开始。为了 emacs 用户的习惯，`"-*"` 和 `"-"` 组成的序列特别被忽略掉，因此，如果你有意使用，你可以说：

```
#! /bin/sh -- # -*- perl -*- -p
eval 'exec perl -S $0 ${1+"$@"}'
if 0;
```

于是 Perl 就只看见 `-p` 开关。奇妙的小发明 `"-*- perl -*-"` 告诉 emacs 以 Perl 模式启动；如果你不使用 emacs，那么你用不着它。我们稍后在描述 `-S` 的时候解释它那堆东西。

如果你有 `env(1)` 程序，也可以用类似的技巧：

```
#! /usr/bin/env perl
```

前面的例子使用 Perl 解释器的相对路径，把用户路径里出现的第一个 `perl` 拿过来。如果你想要特定版本的 Perl，比如，`perl5.6.1`，那么把它直接放进 `#!` 行的路径里，要么是和 `env` 程序一起，要么是 `-S` 那堆东西在一起，或者在普通的 `#!` 里。

如果 `#!` 行不包含单词 `"perl"`，那么在 `#!` 后面的程序代替 Perl 解释器执行。比如，假设你有一个普通的 Bourne shell 脚本，内容是：

```
#! /bin/sh
echo "I am a shell script"
```

如果你把这个文件给 Perl，那么 Perl 会为你运行 `/bin/sh`。这个举止可能有些怪异，但是它可以帮助那些不识别 `#!` 的机器的用户。因为这些用户可以通过设置 `SHELL` 环境变量告诉一个程序（比如一个邮件程序）说，它们的 shell 是 `/usr/bin/perl`，然后 Perl 就帮他们把该程序发配给正确的解释器，就算他们的内核傻得不会干这事也没关系。

不过还是让我们回到真正的 Perl 脚本里头来。在完成你的脚本的定位之后，Perl 把整个程序编译成一种内部格式（参阅第十八章，编译）。如果发生任何编译错误，脚本的执行甚至都不能开始。（这一点和典型的 shell 脚本或者命令文件不同，它们在发现一个语法错误之前可能先跑上一段。）如果脚本语法正确，那么就开始执行。如果脚本运行到最后也没有发现一个 `exit` 或者 `die` 操作符，那么

Perl 隐含地提供一个 `exit(0)`，为你的脚本的调用者标识一个成功的结束状态。（这一点和典型的 C 程序也不一样，在 C 里面，如果你的程序只是按照通常的方法结束，那么你的退出状态是随机的。）

### 19.1.1 在非 Unix 系统上的 #! 和引号

---

Unix 的 #! 技巧可以在其他系统上仿真：

- **\*Macintosh\***

在 Macintosh 上的 Perl 程序有合适的创建者和类型，所以双击它们就会 调用 Perl 应用。

- **\*MS-DOS\***

创建一个批处理文件运行你的程序，并且把它在 `ALTERNATIVE_SHEBANG` 里成文。 参阅 Perl 源程序发布的顶级目录里的 `dosish.h` 文件获取更多这方面的信息。

- **OS/2**

把下面这行：

```
extproc perl -S -your_siwtches
```

放在 \*.cmd 文件的第一行里（-S 绕开了一个在 `cmd.exe` 里的 “extproc” 处理的臭虫。）

- **VMS**

把下面几行：

```
% perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8'
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

放在你的程序的顶端，这里的 `-mysw` 是任何你想传递给 Perl 的命令行 开关。现在你可以直接通过键入 `perl program` 调用你的程序，或者说 `@program` 把它当作一个 DCL 过程调用，或者使用程序名通过隐含地 `DCL$PATH` 调用。这些方法记起来有点困难，不过如果你在 perl 里键入 `"-V:startperl"`，那么 Perl 会给你显示出来。如果你记不住这个用法—— 很好，那就是你买这本书的原因。

- **Win??**

如果在一些 Microsoft Windows 系列操作系统里（也就是 Win95，Win98，Win00（注：请原谅，我们只用两位数表示年代），WinNT，不过不包括 Win31。）使用 Perl 的 [ActiveState<sup>?</sup>](#) 版本。Perl 的安装过程修改了 Windows 的注册表，把 .pl 扩展名和 Perl 解释器关联起来。

如果你安装了另外一个移植的 Perl，包括那个在 Perl 版本里 Win32 目录 里的那个，那么你就必须 自己修改 Windows 注册表。

请注意如果你使用 .pl 扩展名就意味着你再也不能区分一个可执行 Perl 程序和一个 “perl 库” 文件了。 你可以用 .plx 做 Perl 程序的扩展名以 避免这个问题。现在这个问题已经不明显了，因为大多数 Perl 模块在 .pm 文件里。

在非 Unix 系统上的命令行解释器通常和 Unix shell 有不同的引号的用法。你必须了解你的命令行解释器里的特殊字符（\*，\，和 " 是比较常见的）以及如何保护通过 -e 开关运行的一行程序里的空白和这些特殊字符。如果 % 是你的 shell 的特殊字符，你还可以把单个 % 改成 %%，或者把它逃逸。

在一些系统上，你可能还要把单引号改成双引号。但是不要在 Unix 或者 Plan9 系统，或者任何运行 Unix 风格的 shell 上这么干，比如从 MKS 工具箱或者来自 Cygnus 的哥几个（现在在

Redhat) 的 Cygwin 包。呃, Microsoft 的叫 Interix 的新的 Unix 仿真器也开始看到了, 也要注意。

比如, 在 Unix 和 Mac OS X 里, 用:

```
%perl -e 'print "Hello world\n"'
```

在 Macintosh (Mac OS X 的前身) 里, 用:

```
print "Hello world\n"
```

然后运行 "Myscript" 或者 Shift-Command-R。

在 VMS 上, 使用:

```
$perl -e "print \"Hello world\n\""
```

或者再次使用 qq//:

```
$perl -e "print qq(Hello world\n)"
```

在 MS-DOS 等等里, 用:

```
A:> perl -e "print \"Hello world\n\""
```

或者用 qq// 使用自己的引号:

```
A:> perl -e "print qq(Hello world\n)"
```

问题是这些方法都是不可靠的: 它依赖于你使用的命令行解释器。如果 4DOS 是命令行 shell, 下面这样的命令可能跑得好些:

```
perl -e "print \"Hello world\n\""
```

Windows NT 上的 CMD.EXE 程序好象在没有人注意的情况下偷偷加入了许多 Unix shell 的功能, 但你需要看看它的文档, 查找它的引号规则。

在 Macintosh 上, (注: 至少在 Mac OS X 发布之前, 我们可以很开心地说它是源于 BSD 的系统。)所有这些取决于你用的是哪个环境。MacPerl<sup>?</sup> shell, 或者 MPW, 都很象 Unix shell, 支持几个引号变种, 只不过它把 Macintosh 的非 ASCII 字符自由地用做控制字符。

对这些问题我们没有通用的解决方法。它们就这么乱。如果你用的不是 Unix 系统, 但是想做命令行类的处理, 最好的解决方法是找一个比你的供应商提供的更好的命令行解释器, 这个不会太难。

或者把所有的东西都在 Perl 里写, 完全忘掉那些单行命令。

## 19.1.2 Perl 的位置

---

尽管这个问题非常明显, 但是 Perl 只有在用户很容易找到它的时候才有用。如果可能, 最好 /usr/bin/perl 和 /usr/local/bin/perl 都是指向真正二进制文件的符号链接。如果无法做到这一点, 我们强烈建议系统管理员把 Perl 和其相关工具都放到一个用户的标准 PATH 里面去, 或者其他的什么明显而又方便的位置。

在本书中, 我们在程序的第一行使用标准的 `#!/usr/bin/perl` 符号来表示在你的系统上能用的任何相应机制。如果你想运行特定版本的 Perl, 那么使用声明的位置:

```
#!/usr/local/bin/perl5.6.0
```

如果你只是想运行至少某个版本的 Perl, 而不在乎运行更高版本, 那么在你的程序顶端附近放下面

这样的语句：

```
use v5.6.0
```

（请注意：更早的 Perl 版本使用象 5.005 或者 5.004\_05 这样的数字。现在我们会把它们看作 5.5.0 和 5.4.5，不过比 5.6.0 早的 Perl 版本不能理解那些符号。）

### 19.1.3 开关

单字符并且没有自己的参数的命令行开关可以与其他跟在后面的开关组合（捆绑）在一起。

```
#!/usr/bin/perl -spi.bak # 和 -s -p -i.bak 一样
```

开关，有时候也叫选项或者标志。不管你叫他们什么，下面的是 Perl 识别的一些：

- **--**

结束开关处理，就算下一个参数以一个负号开头也要结束。它没有其他作用。

- **-0OCTNUM**

-0 把记录分隔符（\$/）声明为一个八进制数字。如果没有提供 OCTNUM，那么 NUL 字符（也就是 ASCII 字符 0，Perl 的 "\0"）就是分隔符。其他开关可以超越 或者遵循这个八进制数字。比如，如果你有一个可以打印文件名是空字符结尾的 文件的 find(1) 版本，那么你可以这么说：

```
% find . -name '*.bak' -print0 | perl -n0e unlink
```

特殊数值 00 令 Perl 以段落模式读取文件，等效于把 \$/ 变量设置为 ""。数值 0777 令 Perl 立即把整个文件都吃掉。这么做等效于解除 \$/ 变量的定义。我们使用 0777 是因为没有 ASCII 字符是那个数值。（不幸的是，有一个 Unicode 字符是那个数值，\N{LATIN SMALL LETTER O WITH STROKE AND ACUTE}，不过 有人说你不会用那个字符分隔你的记录。）

- **-a**

打开自动分割模式，不过只有在和 -n 或 -p 一起使用时才有效。在 -n 和 -p 开关生成的 while 循环里首先对 @F 数组进行一次隐含的 split 命令调用。因此：

```
% perl -ane 'print pop(@F), "\n";'
```

等效于：

```
LINE: while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

你可以通过给 split 的 -F 开关传递一个正则表达式声明另外一个域分隔符。 比如，下面两个调用是等效的：

```
% awk -F: '$7 && $7 !~ /\^\/bin/' /etc/passwd
% perl -F: -lane 'print if $F[6] && $F[6] !~ m(^/bin)' /etc/passwd
```

- **-c**

令 Perl 检查脚本的语法然后不执行刚编译的程序退出。从技术角度来讲，它比 那做得更多一些：它

会执行任何 **BEGIN** 或 **CHECK** 块以及任何 **use** 指令，因为 这些都是在执行你的程序之前要发生的事情。不过它不会再执行任何 **INIT** 或者 **END** 块了。你仍然通过在你的主脚本的末尾包括下面的行获得老一些的但很少 用到的性质：

```
BEGIN { $^C = 0; exit; }
```

- **-C**

如果目标系统支持本机宽字符，则允许 **Perl** 在目标系统上使用本机宽字符 **API**（对于版本 **5.6.0** 而言，它只能用于 **Microsoft** 平台）。特殊变量 `${^WIDE_SYSTEM_CALLS}` 反映这个开关的状态。

- **-d**

在 **Perl** 调试器里运行脚本。参阅第二十章，**Perl** 调试器。

- **-dMODULE**

在调试和跟踪模块的控制下运行该脚本，该模块以 **Devel::MODULE** 形式安装在 **Perl** 库里。比如，**-d:Dprof** 使用 **Devel::Dprof** 调节器执行该脚本。参阅第 二十章的调试节。

- **-DLETTERS**

- **-DNUMBER**

设置调试标志。（这个开关只有在你的 **Perl** 版本里编译了调试特性（下面描述）之后才能用。）你可以声明一个 **NUMBER**，它是你想要的位的总和，或者一个 **LETTER** 的列表。比如，想看看 **Perl** 是如何执行你的脚本的，用 **-D14** 或者 **-DsIt**。另外一个有用的值是 **-D1024** 或 **-Dx**，它会列出你编译好的语法树。而 **-D512** 或 **-Dr** 显示编译好的正则表达式。数字值在内部可以作为特殊的变量 `$^D` 获得。表 19-1 列出了赋了值的位。

**表 19-1 -D 选项**

位	字母	含义
1	p	记号分解和分析
2	s	堆栈快照
4	l	标签堆栈处理
8	t	跟踪执行
16	o	方法和重载解析
32	c	字符串/数字转换
64	P	为 <b>-P</b> 打印预处理器命令
128	m	存储器分配
256	f	格式化处理
512	r	正则分析和执行
1024	x	语法树倾倒
2048	u	污染检查
4096	L	内存泄露（需要在编译 <b>Perl</b> 时使用 <b>-DLEAKTEST</b> ）
8192	H	哈希倾倒--侵占 <b>values()</b>
16384	X	便签簿分配
32768	D	清理



## 65536 S 线程同步

所有这些标志都需要 Perl 的可执行文件是为调试目的特殊制作的。不过，因为调试制作不是缺省，所以除非你的系统管理员制作了这个特殊的 Perl 的调试版本，否则你根本别想用 `-D` 开关。参阅 Perl 源文件目录里面的 `INSTALL` 文件获取细节，短一些的说法是你在编译 Perl 本身的时候需要给你的 C 编译器传递 `-DDEBUGGING` 编译选项。在 `Configure` 问你优化选项和调试选项的时候，如果你包括了 `-g` 选项，那么这个编译选项会自动加上。

如果你只是想在你的每行 Perl 代码执行的时候获取一个打印输出（象 `sh -x` 为 shell 脚本做的那样），那你就不能用 `-D` 开关。你应该用：

```
# Bourne shell 语法
$PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# csh 语法
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

见第二十章获取细节和变种。

### • -e PERLCODE

可以用于输入一行或多行脚本。如果使用了 `-e`，Perl 将不在参数列表中寻找程序的文件名。Perl 把 `PERLCODE` 参数当作有新行结尾看待，所以可以给出多个 `-e` 命令形成一个多行的程序。（一定要使用分号，就象你在文件里的程序一样。）`-e` 给每个参数提供新行并不意味着你必须使用多个 `-e` 开关；如果你的 shell 支持多行的引用，比如 `sh`，`ksh`，或 `bash`，你可以把多行脚本当作一个 `-e` 参数传递：

```
$perl -e 'print "Howdy, ";
        print "@ARGV!\n";' world

Howdy, world!
```

对于 `csh` 而言，可能最好还是使用多个 `-e` 开关：

```
%perl -e 'print "Howdy, ";' -e 'print "@ARGV!\n";' world
Howdy, world!
```

在行计数的时候，隐含的和明确给出的新行都算数，所以两个程序中的第二个 `print` 都是在 `-e` 脚本的第 2 行。

### • -F PATTERN

声明当通过 `-a` 开关（否则没有作用）自动分裂是要 `split` 的模式。该模式可以由斜杠（`/`），双引号（`"`），或者单引号（`'`）包围。否则，他会自动放到单引号里。请注意如果要通过 shell 传递引号，你必须把你的引号引起来，具体怎么做取决于你的系统。

### • -h

打印一个 Perl 命令行选项的概要

### • -i EXTENSION

`-i` 声明那些由 `<>` 构造处理的文件将被现场处理。Perl 是通过这样的办法实现的：先重命名输入文

件，然后用原文件名打开输出文件，并且把该输出文件选为调用 **print**, **printf**, 和 **write** 的缺省。  
(注：通常，这并不是真的“现场”。它是 相同的文件名，但是不同的物理文件。)

**EXTENSION** 用于修改旧文件名然后做一个备份的拷贝。如果没有提供 **EXTENSION**， 那么不做备份并且当前文件被覆盖。如果 **EXTENSION** 不包含一个 **\***，那么该字符串 被附加到当前文件名的后面。如果 **EXTENSION** 包含一个或多个 **\*** 字符，那么每个

- 都被当前正在处理的文件名替换。用 **Perl** 的话来说，你可以认为事情是这样的：

```
($backup = $extension) =~ s/\*/$file_name/g;
```

这样就允许你把一个前缀用于备份文件，而不是--或者可以说是除了后缀以外：

```
%perl -pi'orig_*' -e 's/foo/bar/' xyx # 备份到 'orig_xyx'
```

你甚至可以可以把原来文件的备份放到另外一个目录里（只要该目录已经存在）：

```
%perl -pi'old/*.orig' -e 's/foo/bar/' xyx # 备份到 'old/xyx.orig'
```

这些一行程序对都是相等的：

```
%perl -pi -e 's/foo/bar/' xyx # 覆盖当前文件
%perl -pi'*' -e 's/foo/bar/' xyx # 覆盖当前文件

%perl -pi'.orig' -e 's/foo/bar/' xyx # 备份到 'xyx.orig'
%perl -pi'*.orig' -e 's/foo/bar/' xyx # 备份到 'xyx.orig'
```

从 **shell** 上，你说：

```
%perl -p -i.oirg -e "s/foo/bar/;"
```

等效于使用下面的程序：

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

而上面的又是下面的程序的便利缩写：

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while(<>){
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        unless (rename($ARGV, $bckup)) {
            warn "cannot rename $ARGV to $backup: $! \n";
            close ARGV;
            next;
        }
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
}
```



```

        s/foo/bar/;
    }
    continue {
        print;      # 这一步打印到原来的文件名
    }
    select(STDOUT);

```

这一段长代码实际上相当于那条简单的单行带 `-i` 开关的命令，只不过 `-i` 的形式不需要拿 `$ARGV` 和 `$oldargv` 进行比较以判断文件名是否改变。不过，它的确使用 `ARGVOUT` 作为选出的文件句柄并且在循环结束以后把原来的 `STDOUT` 恢复为缺省文件句柄。象上面的代码那样，Perl 创建备份文件时并不考虑任何输出是否真的被修改了。如果你想附加到每个文件背后，或者重置行号，那么请参阅 `eof` 函数的描述，获取关于如何使用不带圆括号的 `eof` 定位每个输入文件的结尾的例子。

如果对于某个文件来说，Perl 不能创建象 `EXTENSION` 里声明的那样的备份文件，它会为之发出一个警告然后继续处理列出的其他文件。

你不能用 `-i` 创建目录或者剥离文件的扩展名。你也不能用一个 `~` 来表示家目录 -- 因为有些家伙喜欢使用这个字符来表示他们的备份文件：

```
%perl -pi~ -e 's/foo/bar' file1 file2 file3...
```

最后，如果命令行上没有给出文件名，那么 `-i` 开关不会停止 Perl 的运行。如果发生这种事情，则不会做任何备份，因为不能判断原始文件，而可能会发生从 `STDIN` 到 `STDOUT` 的处理。

## • -IDIRECTORY

`-I` 声明的目录比 `@INC` 优先考虑，它包含搜索模块的目录。`-I` 还告诉 C 预处理器到那里寻找包含文件。C 预处理器是用 `-P` 调用的；缺省时它搜索 `/usr/include` 和 `/usr/lib/perl`。除非你打算使用 C 预处理器（而实际上几乎没人再干这事了），否则你最好在你的脚本里使用 `use lib` 指示器。不过，`-I` 和 `use lib` 类似，它隐含地增加平台相关的目录。参阅第三十一章，实用模块，里的 `use lib` 获取细节。

## • -IOCTNUM

`-l` 打开自动行结束处理。它有两个效果：首先，如果它和 `-n` 或者 `-p` 一起使用，它自动 `chomp` 行终止符，其次，它把 `$\` 设置为 `OCTNUM` 的数值，这样任何打印语句将有一个值为 `OCTNUM` 的 ASCII 字符追加在结尾代替行终止符。如果省略了 `OCTNUM`，`-l` 把 `$\` 设置为 `$/` 的当前值，通常是新行。因此要把行截断为 80 列，你这么说：

```
%perl -lpe 'substr($_, 80) = ""'
```

请注意在处理这个开关的时候完成了 `$\ = $/` 的赋值，因此如果 `-l` 开关后面跟着 `-0` 开关，那么这个输入记录分隔符可以与输出记录分隔符不同：

```
%gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

这条命令把 `$\` 设置为新行而稍后把 `$/` 设置为空字符。（请注意如果 `0` 直接跟在 `-l` 后面，它将被当作 `-l` 开关的一部分。这就是为什么我们在它们之间绑上了 `-n` 开关。）

## • -m 和 -M

这些开关装载一个 `MODULE`，就象你执行了一个 `use` 那样，如果你声明的是 `-MODULE`，而不是 `MODULE`，那么它将调用 `no`。比如，`-Mstrict` 类似 `use strict`，而 `-M-strict` 类似 `-no strict`。

- **-mMODULE**

在执行你的脚本之前执行 `use MODULE()`。

- **-MMODULE**

- **-M'MODULE ...'**

在执行你的脚本之前执行 `use MODULE`。这条命令是通过简单的解析 `-M` 后面 剩下的参数形成的，因此你可以用引号在该模块名后面加额外的代码，比如，

```
-M'MODULE qw(foo bar)'。  
-MMODULE=arg1,arg2...
```

一块小小的语法糖，意思是你还可以把 `-Mmodule=foo,bar` 当作 `-M'module qw(foo bar)'` 的一个缩写来用。这样当输入符号的时候就避免了 引号的使用。`-Mmodule=foo,bar` 生成的实际的代码是：

```
use module split(/,/, q{foo,bar})
```

请注意 `=` 的形式删除了 `-m` 和 `-M` 之间的区别，但是最好还是使用大写的 形式以避免混淆。

你可能只会在真正的 Perl 命令行调用的时候使用 `-M` 和 `-m` 开关，而不会在 `#!` 封装的选项行上用。（如果你准备把它放在文件里，为什么不用一个等效的 `use` 或者 `no` 代替呢？）

- **-n**

令 Perl 认为在你的脚本周围围绕着下面的循环，这样就让你的脚本遍历文件名 参数，就象 `sed -n` 或者 `awk` 做的那样：

```
LINE:
while(<>) {
    ...      # 你的脚本在这里
}
```

你可以在你的脚本里把 `LINE` 当作一个循环标记来用，即使你在你的文件里看 不到实际的标记也如此。

请注意，那些行缺省的时候并不打印。参阅 `-p` 选项看看如何打印。下面是一个 删除旧于一周的文件的有效方法：

```
find . -mtime +7 -print | perl -nle unlink
```

这样做比用 `find(1)` 的 `-exec` 开关要快，因为你不需要对每个找到的文件启动 一个进程。一个很有趣的一致性，你可以用 `BEGIN` 和 `END` 块捕获这个隐含的 循环之前或之后的控制，就象 `awk` 一样。

- **-p**

令 Perl 认为在你的脚本周围围绕着下面的循环，这样就让你的脚本遍历文件名参数，象 `sed` 那样：

```
LINE:
while(<>) {
    ...      # 你的脚本在这里
}
```

```

    }
    continue {
        print or die "-p destination: ${!}\n";
    }
}

```

你可以在你的脚本里把 **LINE** 当作一个循环标记来用，即使你在你的文件里看不到实际的标记也如此。

如果由于某种原因一个参数命名的文件无法打开，Perl 会警告你，然后继续下一个文件。请注意这些行都自动打印出来。在打印的过程中如果发生错误则认为是致命错误。同样也是一个很有趣的一致性，你可以用 **BEGIN** 和 **END** 块捕获这个隐含的循环之前或之后的控制，就象 **awk** 一样。

## • -P

令你的脚本先由 **C** 预处理器运行，然后才由 Perl 编译。（因为注释和 **cpp(1)** 指示都是由 **#** 字符开头，所以你应该避免注释任何 **C** 预处理器可以识别的词，比如 **"if"**，**"else"**，或者 **"define"**。）不过你用不用 **-P** 开关，Perl 都会注意 **#line** 指示以控制行号和文件名，这样任何预处理器都可以通知 Perl 这些事情。参阅第二十四章，普通实践，里面的“在其他语言里生成 Perl”。

## • -S

打开命令行上脚本名之后，但在任何文件名或者一个 **--** 开关处理终止符之前的基本的开关分析。找到的任何开关都从 **@ARGV** 里删除，并且在 Perl 里设置一个同名的开关变量。这里不允许开关捆绑，因为这里允许使用多字符开关。

下面的脚本只有在你带着 **-foo** 开关调用脚本的时候才打印 **"true"**。

```

#!/usr/bin/perl -s
if ($foo) {print "true\n"}

```

如果该开关形如 **-xxx=yyy**，那么 **\$xxx** 变量的值设置为跟在这个参数的等号后面的值（本例中是 **"yyy"**）。下面的脚本只有在你带着 **-foo=bar** 开关调用的时候才打印 **"true"**。

```

#!/usr/bin/perl -s
if ($foo eq 'bar') { print "true\n" }

```

## • -S

让 Perl 使用 **PATH** 环境变量搜索该脚本（除非脚本的名字包含目录分隔符）。

通常，这个开关用于帮助在那些不支持 **#!** 的平台上仿真 **#!**。在许多有兼容 **Bourne** 或者 **C shell** 的平台上，你可以用下面这些：

```

#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 ${*}"
if $running_under_some_shell;

```

系统忽略第一行然后把脚本交给 **/bin/sh**，然后它继续运行然后试图把 Perl 脚本当作一个 **shell** 脚本运行。该 **shell** 把第二行当作一个普通的 **shell** 命令执行，因此启动 Perl 解释器。在一些系统上，**\$0** 并不总是包含路径全名，因此 **-S** 告诉 Perl 在必要的时候搜索该脚本。在 Perl 找到该脚本以后，它分析该行并且忽略它们，因为变量 **\$running\_under\_some\_shell** 总是为假。一个更好的构造是 **\*\$** 应该是 **\${1+"\$@"}**，它可以处理文件名中嵌入的空白这样的东西，但如果该脚本被 **csh** 解释将不能运行，为了用 **sh** 代替 **csh** 启动，有些系统必须用一个只有一个冒号的行替代 **#!** 行，Perl 会礼貌地忽略那样的行。其他不能支持这些的系统必须用一种完全迂回的构造，这

种构造可以在任何 **csh**, **sh**或者**perl** 里运行, 这种构造是这样的:

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'  
& eval 'exec /usr/bin/perl -S $0 $argv:q'  
if -;
```

的确, 这个东西很难看, 不过那些系统也一样难看(注: 我们故意用这个词)。

在一些平台上, **-S** 开关同样也令 **Perl** 在搜索的时候给文件名附加后缀。比如, 在 **Win32** 平台上, 如果对最初的文件名查找失败而且这个文件名原来没有用 **.bat** 或者 **.com**, 则会给该文件名后缀上 **.bat** 或者 **.com** 进行查找。如果你的 **Perl** 是打开调试编译的, 那么你可以使用 **Perl** 的 **-Dp** 开关来观察搜索过程。

如果你提供的文件名包含目录分隔符(即使只是一个相对路径名, 而不是绝对路径名), 而且如果没有找到该文件, 那么在那些会隐含附加文件扩展名的平台上(不是 **Unix**)就会做这件事, 然后一个接一个的找那些带这些扩展名的文件。

在类似 **DOS** 的平台上, 如果脚本不包含目录分隔符, 它首先会在当前目录搜索, 然后再寻找 **PATH**。在 **Unix** 平台上, 出于安全性考虑, 为了避免未经明确请求, 偶然执行了当前工作目录里面的东西, 将严格地在 **PATH** 里搜索,

#### • **-T**

强制打开“感染”检查, 这样你就可以检查它们了。通常, 这些检查只有在运行 **setuid** 或者 **setgid** 的时候才进行。把它们明确地打开, 让程序的作者自己控制 是一个不错的主意, 比如在 **CGI** 程序上。参阅第二十三章, 安全。

#### • **-u**

令 **Perl** 在编译完你的脚本以后倾倒核心。然后从理论上讲你可以用 **undump** 程序(未提供)把它转成一个可执行文件。这样就以一定的磁盘空间为代价(你可以通过删除可执行文件来最小化这个代价)换来了速度的提升。如果你想在输出之前执行一部分你的脚本, 那么使用 **Perl** 的 **dump** 操作符。注意: **undump** 的可用性是平台相关的; 可能在某些 **Perl** 的移植版本里不能用。它已经被新的 **Perl** 到 **C** 的代码生成器替换掉了, 由这个代码生成器生成的东西更具移植性(不过仍然处于实验期)。

#### • **-U**

允许 **Perl** 进行不安全的操作。目前, 唯一的“不安全”的操作是以超级用户身份运行是删除目录, 以及在把致命污染检查转换成警告的情况下运行 **setuid** 程序。请注意如果要真的生成污染检查警告, 你必须打开警告。

#### • **-v**

打印你的 **Perl** 的版本和补丁级别, 以及另外一些信息。

#### • **-V**

打印 **Perl** 的主要配置值的概要以及 **@INC** 的当前值。

#### • **-V:NAME**

向 **STDOUT** 打印命名配置变量的值。**NAME** 可以包括正则字符, 比如用 **“.”** 匹配任何字符, 或

者 `".*"` 匹配任何可选的字符序列。

```
%perl -V:man.dir
man1dir='/usr/local/man/man1'
man3dir='/usr/local/man/man3'

%perl -V:'.*threads'
d_oldpthreads='undef'
use5005threads='define'
useithreads='undef'
usethreads='define'
```

如果你要求的环境变量不存在，它的值将输出为 `"UNKNOWN"`。在程序里可以使用 `Config` 模块获取配置信息，不过在哈希脚标里不支持模式：

```
%perl -MConfig -le 'print $Config{man1dir}'
/usr/local/man/man1
```

参阅第三十二章，标准模块，里的 `Config` 模块。

#### • **-w**

打印关于只提到一次的变量警告，以及在设置之前就使用了的标量的警告。同时 还警告子过程重定义，以及未定义的文件句柄的引用或者文件句柄是只读方式打开 的而你却试图写它们这样的信息。如果你用的数值看上去不象数字，而你却把它们 当作数字来使用；如果你把一个数组当作标量使用；如果你的子过程递归深度超过 **100** 层；以及无数其他的东西时也会警告。参阅第三十三章，诊断信息，里的每条 标记着 `“(W)”` 的记录。

这个开关只是设置全局的 `$^W` 变量。它对词法范围的警告没有作用--那方面的请 参阅 `-W` 和 `-X` 开关。你可以通过使用 `use warning` 用法打开或者关闭特定的 警告，这些在第三十一章描述。

#### • **-W**

无条件地永久打开程序中的所有警告，即使你用 `no warnings` 或者 `$^W = 0` 局部关闭了警告也没用。它的作用包括所有通过 `use`, `require`, 或者 `do` 包括 进来的文件。把它当作 Perl 的等效 `lint` (1) 命令看待。

#### • **-XDIRECTORY**

##### ○ **-x**

告诉 Perl 抽取一个嵌入在一条信息里面的脚本。前导的垃圾会被丢弃，直到以 `#!` 开头并包括字符串 `“perl”` 的第一行出现。任何在该行的单词 `“perl”` 之后的 有意义的开关都会被 Perl 使用。如果声明了一个目录名，Perl 在运行该脚本 之前将切换到该目录。`-x` 开关只控制前导的垃圾，而不关心尾随的垃圾。如果 该脚本有需要忽略的尾随的垃圾，那它就必须以 **END** 或者 **DATA** 结束，（如果需要，该脚本可以通过 `DATA` 文件句柄处理任何部分或者全部尾随的垃圾。 它在理论上甚至可以 `seek` 到文件的开头并且处理前导的垃圾。）

#### • **-X**

无条件及永久地关闭所有警告，做的正好和 `-W` 完全相反。

## 19.2 环境变量

除了各种明确修改 Perl 行为的开关以外，你还可以设置各种环境变量以影响各种潜在的性质。怎样设置环境变量是系统相关的，不过，如果你用 **sh**，**ksh** 或者 **bash**，有一个技巧就是你可以为单条命令临时地设置一个环境变量，就好象它是一个有趣的开关一样。你必须在命令前面设置它：

```
$PATH='/bin:/usr/bin' perl myproggie
```

你可以在 **csh** 或者 **tcsh** 里用一个子 **shell** 干类似的事：

```
%(setenv PATH "/bin:/usr/bin"; perl myproggie)
```

否则，你通常就要在你的家目录里的一些名字象 **.chsrc** 或者 **.profile** 这样的文件里设置环境变量。在 **csh** 或者 **tcsh** 里，你说：

```
%setenv PATH '/bin:/usr/bin'
```

而在 **sh**，**ksh**，和**bash** 里，你说：

```
$PATH='/bin:/usr/bin'; export PATH
```

其他系统里有其他的半永久的设置方法。下面是 Perl 会注意的环境变量：

- **HOME** 如果不带参数调用 **chdir** 时要用到。
- **LC\_ALL, LC\_CTYPE, LC\_COLLATE, LC\_NUMERIC, PERL\_BADLAND**

控制 Perl 操作某种自然语言的方法的环境变量。参阅 **perllocale** 的联机 文档。

- **LOGDIR**

如果没有给 **chdir** 参数，而且没有设置 **HOME**。

- **PATH** 用于执行子进程，以及使用了 **-S** 开关的时候寻找程序。
- **PERL5LIB**

一个用冒号分隔的目录的列表，用于指明在搜索标准库和当前目录之前搜索 Perl 库文件的目录。如果存在有任何声明的路径下的体系相关的目录，则 都回自动包括进来。如果没有定义 **PERL5LIB**，则测试 **PERLLIB**，以保持与 老版本的向下兼容。如果运行了污染检查（要么是因为该程序正在运行 **setuid** 或者 **setgid**，要么是因为使用了 **-T** 开关。），则两个库变量都 不使用。这样的程序必须用 **use lib** 用法来实现这些目的。

- **PERL5OPT**

缺省命令行开关。在这个变量里的开关会赋予每个 Perl 命令行。只允许 **-[DIMUdmw]**。如果你运行污染检查（因为该程序正在运行 **setuid** 或者 **setgid**，或者使用了 **-T** 开关），则忽略这个变量。如果 **PERL5OPT** 是以 **-T** 开头，那么将打开污染检查，导致任何后继选项都被忽略。

- **PERL5DB**

用于装载调试器代码的命令。缺省的是：

```
BEGIN { require 'perl5db.pl' }
```



参阅第二十章获取这个变量的更多用法。

### • PERLSHELL(仅用于 Microsoft 移植)

可以设置一个候选 shell，这个 shell 是 Perl 在通过反勾号或者 system 执行命令的时候必须的。缺省时在 WinNT<sup>2</sup> 上是 cmd.exe /x/c 以及在 Win95 上是 command.com /c。Perl 认为该值是空白分隔的。在 任何需要保护的字符（比如空白和反斜杠）之前用反斜杠保护。

请注意 Perl 并不将 COMSPEC 用于这个目的，因为 COMSPEC 在用户中有 更高的可变性，容易导致移植问题。另外，Perl 可以使用一个不适合交互 使用的 shell，而把 COMSPEC 设置为这样的 shell 可能会干涉其他程序的 正常功能（那些程序通常检查 COMSPEC 以寻找一个适于交互使用的 shell）。

### • PERLLIB

一个冒号分隔的目录列表，在到标准库和当前目录查找库文件之前先到这些 目录搜索。如果定义了 PERLSLIB，那么就不会使用 PERLLIB。

### • PERL\_DEBUG\_MSTATS

只有在编译时带上了与 Perl 发布带在一起的 malloc 函数时才有效（也 就是说，如果 perl -V:d\_mymalloc 生成“define”）。如果设置，这就会 导致在执行之后显示存储器统计信息。如果设置为一个大于一的整数，那么也 会导致在编译以后的存储器统计的显示。

### • PERL\_DESTRUCTI\_LEVEL

只有在 Perl 的可执行文件是打开了调试编译的时候才相关，它控制对对象 和其他引用的全局删除的行为。

除了这些以外，Perl 本身不再使用其他环境变量，只是让它执行的程序以及任何该程序运行的子程序可以使用他们。有些模块，标准的或者非标准的，可能会关心其他的环境变量。比如，use re 用法使用 PERL\_RE\_TC 和 PERL\_RE\_COLORS，Cwd 模块使用 PWD，而 CGI 模块使用许多你的 HTTP 守护进程（也就是你的web 服务器）设置的环境变量向 CGI 脚本传递信息。

运行 setuid 的程序在做任何事情之前执行下面几行将会运行得更好，它只是保持人们的诚实：

```
$ENV{PATH} = '/bin:/usr/bin';      # 或者任何你需要的
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

参阅第二十三章获取细节。

---

Revision: r1.3 - 18 Nov 2005 - 01:10 - [TingYu](#)

Perl > [PerlProgramming3](#) > PerlTheCommandLineInterface

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](#)