1 NAME

perlvar – Perl 预定义变量

2 DESCRIPTION

预定义名称

后面列出的名称对 Perl 来说具有特殊含义。大多数标点名称都有合理的助记方法或类似于在 shell 中的用法。然而，如果你就是想用长变量名，那只要在程序开头加上

```
use English;
```

即可。这样会为所有短名称在当前包中创建长名称别名。其中一些甚至还有中间名，一般是从 awk 借用过来的。一般来说，如果不需要 $PREMATCH，$MATCH 和 $POST-MATCH，那最好使用

```
use English '-no_match_vars';
```

调用方式，因为它能避免在用正则表达式时出现效率降低的情况。见 English。

依赖当前被选中文件句柄的变量可以通过在 IO::Handle 对象上调用合适的对象方法来设置，但是这要比使用普通的内建变量效率低一些。(下面的概括行里包含的单词 HANDLE 即指 IO::Handle 对象。) 首先你要声明

```
use IO::Handle;
```

然后就可以用

```
method HANDLE EXPR
```

或者更安全的形式，

```
HANDLE->method(EXPR)
```

每个方法都返回 IO::Handle 属性的旧值，同时接受一个可选的 EXPR。如果提供了该参数，则其指定了所涉及 IO::Handle 属性的新值。如果不提供该参数，大多数方法不改变当前值--除了 autoflush()，它会假定给定了参数 1，稍有不同。

载入 IO::Handle 类是一项代价高昂的操作，因此你该知道如何使用常规的内建变量。

这些变量中的少数几个是“只读的”。这意味着如果直接或者通过引用间接向该变量赋值，就会引起一个运行时异常。

在修改本文档中描述的大部分特殊变量的缺省值时都需要特别小心。多数情况下应该 在修改之前局部化这些变量，如果不这么做，就可能影响依赖于你所修改特殊变量缺 省值的其他模块。下面是一次性读入整个文件的一种正确方法：

```
open my $fh, "foo" or die $!;
local $/; # enable localized slurp mode
my $content = <$fh>;
close $fh;
```

但下面的代码就很糟糕：

```
open my $fh, "foo" or die $!;
undef $/; # enable slurp mode
my $content = <$fh>;
close $fh;
```

因为一些模块可能想以默认的“行模式”从文件中读取数据，而一旦我们刚才 列出的代码得到执行，在同一个 Perl 解释器内运行的所有其他代码读到的 $/ 全局值都会被改变。

通常，在局部化一个变量时总是想让影响限制在尽可能小的范围内，因此 应该自己建立一个 {} 块，除非你已经处于某些小的 {} 块内。例如：

```
my $content = '';
open my $fh, "foo" or die $!;
{
    local $/;
    $content = <$fh>;
}
close $fh;
```

下面是代码失控的一个例子：

```
for (1..5){
    nasty_break();
    print "$_ ";
}
sub nasty_break {
    $_ = 5;
    # do something with $_
}
```

你可能希望上述代码打印出：

1 2 3 4 5

但实际上得到的却是：

5 5 5 5 5

为什么？因为 nasty_break() 修改了 $_ 而没有事先将其局部化。 改正方法是增加 local()：

```
 local $_ = 5;
```

虽然在这样一个短小的例子里很容易发现问题，但在更复杂的代码中，如果不 对特殊变量进行局部化更改就是在自找麻烦。

下列内容按照先标量变量、后数组、最后散列的顺序排列。

$ARG

$_

默认的输入和模式搜索空间。下面的几对代码都是等同的：

```
while (<>) {...}    # equivalent only in while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/
```

```
tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chomp
chomp($_)
```

以下是几处即使没有写明 Perl 也会假定使用 $_ 的地方：

- 各种单目函数，包括像 ord() 和 int() 这样的函数以及除 -t 以外所有的文件 测试操作 (-f，-d)，-t 默认操作 STDIN。
- 各种列表函数，例如 print() 和 unlink()。
- 没有使用 =~ 运算符时的模式匹配操作 m//、s/// 和 tr///。
- 在没有给出其他变量时是 foreach 循环的默认迭代变量。
- grep() 和 map() 函数的隐含迭代变量。
- 当 while 仅有唯一条件，且该条件是对 <FH> 操作的结果进行测试时，$_ 就是存放输入记录的默认位置。除了 while 测试条件之外不会发生这种情况。

(助记：下划线在特定操作中是可以省略的。)

$a

$b

是使用 sort() 时的特殊包变量，参见 sort in perlfunc。 由于这一特殊性，$a 和 $b 即使在用了 strict 'vars' 指示符以后也不需要声明(用 use vars 或者 our())。 如果想要在 sort() 的比较块或者函数中使用它们，就不要用 my $a 或 my $b 将其词法化。

$<digits>

含有上次模式匹配中捕获括号集合所对应的子模式，不包括已经退出的嵌套 块中匹配的模式。(助记：类似 \digits。) 这些变量全都是只读的，对于 当前块来说具有动态作用域。

$MATCH

$&

含有上次成功的模式匹配所匹配到的字符串(不包括任何隐藏在块中的匹配或当前 块所包围的 eval())。(助记：同一些编辑器中的 & 类似。) 该变量是只读的， 对于当前块来说具有动态作用域。

在程序中任何地方使用该变量都会使所有正则表达式匹配产生可观的效率降低。 参见 BUGS。

$PREMATCH

$`

含有上次成功的模式匹配内容之前的字符串(不包括任何隐藏在块中的匹配或当前 块所包围的 eval)。(助记：` 常常出现在引起的字符串之前。) 该变量是只读的。

在程序中任何地方使用该变量都会使所有正则表达式匹配产生可观的效率降低。 参见 BUGS。

$POSTMATCH

$'

含有上次成功的模式匹配内容之后的字符串(不包括任何隐藏在块中的匹配或当前 块所包围的 eval())。(助记：' 常常跟在引起的字符串之后。) 例如：

```
local $_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";         # prints abc:def:ghi
```

该变量只读且对于当前块具有动态作用域。

在程序中任何地方使用该变量都会使所有正则表达式匹配产生可观的效率降低。 参见 BUGS。

$LAST_PAREN_MATCH

$+

含有上次成功的搜索模式中最后一个括号匹配的文本。在无法知道可选模式集中 到底哪一个匹配成功时，该变量是非常有用的。例如：

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(助记：积极一点儿向前看。)(译注：“积极”与“正号”是同一个单词。) 该变量只读且相对于当前块具有动态作用域。

$^N

含有上一次成功搜索模式中最近闭合的组(即最右边的右括号构成的组)所匹配 的文本。(助记：最近闭合的(可能)嵌套的括号。) (译注：嵌套的单词为 Nest。)

该变量主要用在 (?{...}) 块的内部，以便检查最近匹配到的文本。例如， 为了有效地用一个变量($1、$2 等等之外的变量)捕获文本，可以将 (...) 替换为

```
(?:(...)(?{ $var = $^N }))
```

像这样设置并使用 $var 就能把你从计算括号个数的烦恼中解放出来了。

该变量对于当前块具有动态作用域。

@LAST_MATCH_END

@+

该数组保存了当前活动的动态作用域中最近成功的子匹配结束处的偏移量。 $+[0] 为整个匹配在字符串中结束处的偏移量，这同用被匹配的变量调用 pos 函数得到的值一样。该数组的第 n 个元素保存了第 n 个子匹配 的偏移量，因此 $+[1] 就是紧接着 $1 结束处的偏移量，$+[2] 是 紧接着 $2 结束处的偏移量，以此类推。可以用 $#+ 得知最近成功的匹配 中有多少个组。参见为 @- 变量给出的例子。

$*

将其设为非零整数值就可以进行字符串内的多行匹配，设为 0(或未定义值) 相当于告诉 Perl 可以假定字符串都是单行的，从而能进行模式匹配的优化。当 $* 为 0 或未定义值时，对含有多个换行符的字符串进行模式匹配会产生很难 理解的结果。它默认为未定义值。(助记：* 匹配很多东西。) 该变量只影响对 ^ 和 $ 的解释。即使在 $* == 0 时也可以搜索一个字面的换行符。

在现在的 Perl 里不应使用 $*，在模式匹配中可以用 /s 和 /m 修饰符取代它的功能。

对 $* 赋非数值量会触发一个警告(并使 $* 表现为 $* == 0)，对 $* 赋数值量则会隐含对其应用 int。

HANDLE->input_line_number(EXPR)

$INPUT_LINE_NUMBER

$NR

$.

为最后访问的文件句柄对应的当前行号。

Perl 中每个文件句柄都记录从其中读出的行数。(Perl 中行的概念也许和你不一样，要看 $/ 的值是什么。) 当从某个文件句柄中读出一行(通过 readline() 或 <>)或对其调用 tell() 或 seek() 时，$. 即成为那个句柄的行计数器的别名。

你可以通过向 $. 赋值来调整计数器，但这并不会实际移动文件指针。局部化 $. 不会使对应文件句柄的行计数器局部化，而只会局部化 $. 和文件句柄的别名关系。

关闭文件句柄时会复位 $.，但在没有 close() 就重新打开一个已打开的文件句柄时不会这样。更多细节参见 IO Operators in perlop。<> 从不显式关闭文件，因此行号会在 ARGV 文件之间持续增长(不过请看看 eof in perlfunc 中的例子)。

你还可以用 HANDLE->input_line_number(EXPR) 访问一个给定文件句柄的行计数器，这样就无需担心最后访问的是哪个句柄了。

(助记：很多程序用“.”表示当前行号。)

IO::Handle->input_record_separator(EXPR)

$INPUT_RECORD_SEPARATOR

$RS

$/

为输入记录分隔符，默认为换行符。该变量会影响 Perl 对“行”这一概念的理解。其功能类似于 awk 中的 RS 变量，在被设置为空字符串时同样会将空白行作为终止标志。(空白行不能含有任何空格或制表符。) 你可以将其设置为含有多个字符的字符串，以匹配多字符的终止标志；也可以设为 undef 以便一直读到文件结束。当文件含有连续的空白行时，把它设为 "\n\n" 和设为 "" 有少许不同：设为 "" 会把两个或更多连续的空白行视为单个空白行；而设为 "\n\n" 则只是盲目地假定其后输入的字符属于下一段，即使这些字符是换行符也一样。(助记：在引用诗句时会用 / 作为行间的分隔。)

```
local $/;          # enable "slurp" mode
local $_ = <FH>;   # whole file now here
s/\n[ \t]+/ /g;
```

切记：$/ 的内容是一个字符串，而不是正则表达式。awk 得在某些方面改进一下了。:-)

将 $/ 设为整数、存有整数的标量或可转换成整数的标量这些值的引用时，Perl 会尝试读入记录而不是行，最大记录长度就是引用的那个整数。因此这段代码：

```
local $/ = \32768; # or \"32768", or \$var_containing_32768
open my $fh, $myfile or die $!;
local $_ = <$fh>;
```

会从 FILE 读取一条不长于 32768 字节的记录。如果你不是在读取一个面向记录 的文件(或者所用的 OS 没有面向记录的文件类型)，那很可能每次读取都得到一 整块的数据。若某条记录比你所设置的记录长度还大，就会把该记录拆成若干片 返回。

在 VMS 上，记录读取是用 `sysread` 的等价物完成的，因此最好不要在同一个 文件上混合使用记录和非记录读。(这不太可能成为问题，因为任何你想以记录模式 读取的文件也许都不能在行模式下用。) 非 VMS 系统用普通 I/O 进行读取，因此 在一个文件中混合记录和非记录读是安全的。

参见 Newlines in perlport 以及 `$.`。

HANDLE->autoflush(EXPR)

$OUTPUT_AUTOFLUSH

$|

若将该变量设为非零值，就会立刻强制进行刷新，并且当前选中的输出通道在每次 打印或写之后都会进行刷新。默认值为 0 (不管选中的通道实际上是否被系统所缓冲， `$|` 只是告诉你 Perl 是否在每次写完之后显式刷新)。典型情况下，若 STDOUT 的输出是终端则是行缓冲的，否则就是块缓冲。设置该变量在向管道或套接字输出 时很有用，比如你正在 rsh 下运行一个 Perl 程序并且想在输出时马上就能看到 输出内容。该变量不影响输入缓冲。关于输入缓冲请参见 getc in perlfunc。 (助记：when you want your pipes to be piping hot.)

IO::Handle->output_field_separator EXPR

$OUTPUT_FIELD_SEPARATOR

$OFS

$,

为 print 的输出域分隔符。通常 print 不经任何修饰就输出它的参数，要 得到更像 awk 的行为，可以将该变量设置成和 awk 的 OFS 变量一样 ，以指定域之间打印什么。(助记：当 print 语句里有“,”时会打印的东西。)

IO::Handle->output_record_separator EXPR

$OUTPUT_RECORD_SEPARATOR

$ORS

$\

为 print 的输出记录分隔符。通常 print 简单地原样输出它的参数，不增加 任何结尾的换行符或其他表征记录结束的字符串。要得到更像 awk 的行为， 可以将该变量设为同 awk 的 ORS 变量一样，以指定在 print 的结尾输出 什么。(助记：设置 `$\` 而不是在 print 结尾加“\n”。另外，它长得和 `$/` 很像，但却是你从 Perl 那里拿“回”的东西。) (译注：“回”原文为 单词“back”，还指代反斜杠“backslash”，起一语双关作用。)

$LIST_SEPARATOR

$"

该变量同 $，类似，但应用于向双引号引起的字符串(或类似的内插字符串) 中内插数组和切片值的场合。默认为一个空格。(助记：我觉得显而易见。)

$SUBSCRIPT_SEPARATOR

$SUBSEP

$;

为模拟多维数组时的下标分隔符。如果你这样引用一个散列元素

```
$foo{$a,$b,$c}
```

实际上意思就是

```
$foo{join($;, $a, $b, $c)}
```

但是别这么写

```
@foo{$a,$b,$c}      # a slice--note the @
```

它的意思是

```
($foo{$a},$foo{$b},$foo{$c})
```

默认为“\034”，同 awk 的 SUBSEP 一样。如果你的散列键包含二进制数据，可能 $; 就没法包含任何可靠的值了。(助记：逗号(语法上的下标分隔符)是半个分号。是啊，我知道这完全没有说服力，但 $, 已经被用做更重要的用途了。)

请考虑像 perllol 里说明的那样使用“真正的”多维数组。

$#

为打印数值时的输出格式。该变量是 awk 中 OFMT 变量一个粗糙的模仿尝试。不过确实有段时间 awk 和 Perl 在什么可以看成数值的概念上有所分歧。它的初始值是“%.ng”，n 是你所用系统上 float.h 中 DBL_DIG 宏的值。这同 awk 的默认 OFMT 设置“%.6g”不一样，因此你需要显式设定 $# 以便得到和 awk 一样的结果。(助记：# 是数值标志。)

不建议使用 $# 变量。

HANDLE->format_page_number(EXPR)

$FORMAT_PAGE_NUMBER

$%

为当前选中的输出通道的当前页码。同格式配合使用。(助记：% 是 nroff 中的页码。)

HANDLE->format_lines_per_page(EXPR)

$FORMAT_LINES_PER_PAGE

$=

为当前选中的输出通道的当前页长度(可打印的行数)。默认为 60。 同格式配合使用。(助记：符号 = 由两个水平行组成。)

HANDLE->format_lines_left(EXPR)

$FORMAT_LINES_LEFT

$-

为当前选中的输出通道的页面残留行数。同格式配合使用。(助记：页面行数 - 已打印行数。)

@LAST_MATCH_START

@-

$-[0] 是最近一次成功匹配的起始偏移量。 $-[n] 是由第 n 个子模式所匹配的子字符串的起始偏移量， 若对应的子模式没有匹配则取为 undef。

因此对 $_ 进行匹配之后，$& 和 substr $_, $-[0], $+[0] - $-[0] 是一样的。类似地，若 $-[n] 已定义，则 $n 同 substr $_, $-[n], $+[n] - $-[n] 是一样的，$+ 也和 substr $_, $-[$#-], $+[$#-] - $-[$#-] 相同。可以用 $#- 找出最近一次成功匹配 中最后一个匹配的子组。与之相对的 $#+ 是那次匹配所用正则表达式中 子组的数目，对应的是 @+。

该数组存放当前活动的动态作用域中最近成功的子匹配开始处的偏移量。 $-[0] 是整个匹配在字符串中开始处的偏移量，数组中第 n 个元素 存有第 n 个子匹配的偏移量，因此 $-[1] 是 $1 开始处的位移， $-[2] 是 $2 开始处的位移，以此类推。

在对某个变量 $var 进行匹配后：

$`** 和 **substr($var, 0, $-[0]) 相同

$&** 和 **substr($var, $-[0], $+[0] - $-[0]) 相同

$'** 和 **substr($var, $+[0]) 相同

$1 和 **substr($var, $-[1], $+[1] - $-[1])** 相同

$2 和 **substr($var, $-[2], $+[2] - $-[2])** 相同

$3 和 **substr $var, $-[3], $+[3] - $-[3])** 相同

HANDLE->format_name(EXPR)

$FORMAT_NAME

$~

为当前被选中输出通道的当前报表格式名称。默认为文件句柄名。(助记：$^ 的兄弟。)

HANDLE->format_top_name(EXPR)

$FORMAT_TOP_NAME

$^

为当前被选中输出通道的当前页眉格式名称。默认为文件句柄名后加 _TOP。(助记：指向页眉。)

IO::Handle->format_line_break_characters EXPR

$FORMAT_LINE_BREAK_CHARACTERS

$:

是一组字符，字符串可以在这些字符后断行以填充格式中的连续域(以 ^ 开始)。默认为 " \n-"，使字符串能在空白或连字符处断开。(助记：诗中的“冒号”是一行的一部分。)

IO::Handle->format_formfeed EXPR

$FORMAT_FORMFEED

$^L

退纸时格式应输出的内容。默认为 \f。

$ACCUMULATOR

$^A

write() 对 format() 行的累加器的当前值。格式会调用 formline() 并将其结果放入 $^A。在调用对应的格式之后，write() 将 $^A 的内容打印出来并清空之。因此你永远不可能看到 $^A 的内容，除非自己直接调用 formline() 并查看该变量 。参见 perlform 和 formline() in perlfunc。

$CHILD_ERROR

$?

由最近的管道关闭、反引号(``)命令、成功调用 wait() 和 waitpid() 或者 system() 操作符返回的状态信息。它就是由 wait() 系统调用返回的 16 位状态字 (或是由其他信息组合而成的类似值)。因此，子进程的退出值实际上是 ($? >> 8)，$? & 127 给出了导致进程结束的信号代码(如果存在)， 而 $? & 128 会报告是否产生了内核转储。(助记：同 sh 和 ksh 类似。)

另外如果 C 中支持 h_errno 变量，在任意 gethost*() 函数失败时，该变量 的值将会通过 $? 返回。

如果已经为 SIGCHLD 安装了一个信号处理器，那么在处理器之外 $? 的值通常 是有问题的。

在 END 子程序里，$? 含有即将交给 exit() 的值。可以在 END 子程序 中修改 $? 以达到改变程序退出状态的效果。例如：

```
END {
  $? = 1 if $? == 255;  # die would make it 255
}
```

在 VMS 系统下，指示符 use vmsish 'status' 使得 $? 反映实际的 VMS 退出 状态，而不是默认的对 POSIX 状态的模拟；细节见 $? in perlvms。

另见 错误指示器。

${^ENCODING}

为 Encode 对象的对象引用，用来将源代码转换成统一码。多亏有了该变量， 你的 Perl 脚本才不会被强制为 UTF-8 编码。默认为 undef。绝对不建议对 该变量进行直接操作。细节见 encoding。

$OS_ERROR

$ERRNO

$!

如果按数值使用该变量，就会得到 errno 变量的当前值；换句话说，如果 某个系统或者库函数调用失败了，就会设置该变量。这意味着 $! 的值仅当 紧接在一个失败之后时才有意义：

```
if (open(FH, $filename)) {
   # Here $! is meaningless.
   ...
} else {
   # ONLY here is $! meaningful.
   ...
   # Already here $! might be meaningless.
}
# Since here we might have either success or failure,
# here $! is meaningless.
```

上述的无意义代表任何东西：零、非零、undef。成功的系统或库函数 调用不会将该变量重置为零。

If used as a string, yields the corresponding system error string. You can assign a number to $! to set errno if, for instance, you want "$!" to return the string for error n, or you want to set the exit value for the die() operator. (Mnemonic: What just went bang?) 若作为字符串使用，则会产生对应的系统错误字符串。可以对 $! 赋一个数 来设置 errno，这样就能用 "$!" 得到错误 n 对应的字符串了，也可以 用来设置 die() 运算符所用的退出值。(助记：刚刚什么东西炸了？)

另见 错误指示器。

%!

当 $! 设置为某个值时 %! 的对应元素即为真值。例如，$!{ENOENT} 为真当且仅当 $! 的当前值为 ENOENT；也就是说，若最后的错误是 "No such file or directory"(或其他等价的东西：并不是所有操作系统都给出 完全一致的错误，当然也不可能都是同一种语言)。要检查某个特定的键在你的系统 上是否有意义，可以用 exists $!{the_key}；要取得合法键的列表，可以用 keys %!。更多信息见 Errno，另外从前面的描述也可以得出 $! 的有效 范围。

$EXTENDED_OS_ERROR

$^E

同当前操作系统相关的错误信息。目前该变量仅在 VMS、OS/2 和 Win32 (以及 MacPerl)下同 $! 有所不同。在所有其他平台上，$^E 总是和 $! 一样。

在 VMS 系统下，$^E 是最近一次系统错误的 VMS 状态值。这比 $! 提供的关于系统错误的信息更为详尽。当 $! 被设置成 EVMSERR 时 该变量尤为重要。

在 OS/2 系统下，$^E 被设置为最近一次通过 CRT 或直接通过 Perl 进行的 OS/2 API 调用所返回的错误代码。

在 Win32 下，$^E 总是返回由 Win32 调用 GetLastError() 所报告 的错误信息，它描述的是发生在 Win32 API 内部的最近一次错误。多数 特定于 Win32 的程序代码会通过

$^E 报告错误，而 ANSI C 及类 Unix 调用则会设置 errno，因此大部分可移植的 Perl 代码都通过 $! 报告错误。

在 $! 的描述中提到的警告一般也适用于 $^E。(助记：额外的错误解释。) (译注：英文中“额外”的单词为“Extra”。)

另见 错误指示器。

$EVAL_ERROR

$@

最近一个 eval() 运算符返回的 Perl 语法错误消息。若 $@ 是空字符串，则最近一次 eval() 进行了正确的解析和执行(但是你所进行的操作可能已经按照通常的形式失败了)。(助记：语法错误发生“在”哪里？) (译注：符号“@”为英文单词“at”简写，意为“在……”)

警告消息不会被收集在该变量中。但你可以像后面描述的那样，通过设置 $SIG{__WARN__} 自己建立一个例程来处理警告。

另见 错误指示器。

$PROCESS_ID

$PID

$$

运行本脚本的 Perl 的进程号。该变量应视为只读的，不过在 fork() 调用 时会被改变。(助记：和 shell 一样。)

Linux 用户注意：在 Linux 下，C 函数 getpid() 和 getppid() 对 不同的线程返回不同的值。为了可移植，该行为没有反映在 $$ 里，该变量 的值在线程间保持不变。如果你想调用底层的 getpid()，可以使用 CPAN 模块 Linux::Pid。

$REAL_USER_ID

$UID

$<

本进程的实际 uid。(助记：如果你用了 setuid，那么这是你原来的 uid。) 可以用 POSIX::setuid() 同时改变实际 uid 和有效 uid。由于改变 $< 需要 进行系统调用，在更改之后应检查 $! 以发现可能产生的错误。

$EFFECTIVE_USER_ID

$EUID

$>

本进程的有效 uid。例：

```
$< = $>;            # set real to effective uid
($<,$>) = ($>,$<);  # swap real and effective uid
```

可以用 POSIX::setuid() 同时改变有效 uid 和实际 uid。更改 $> 后需要 检查 $! 以发现可能产生的错误。

(助记：如果你用了 setuid，那么这是你要变成的 uid。) $< 和 $> 仅在支持 setreuid() 的机器上才能互换。

$REAL_GROUP_ID

$GID

$(

本进程的实际 gid。如果你使用的机器支持同时属于多个组，则该变量给出的是被空格隔开的所在组列表。第一个数值是由 getgid() 返回的结果，后续的内容是 getgroups() 返回的结果，其中可能有和第一个值相同的项。

然而为了设置实际 gid，赋给 $(的必须是单个数值。因此从 $(得到的值在没有强制为数值(例如同零相加)的情况下不应再赋给 $(。

可以用 POSIX::setgid() 同时改变实际 gid 和有效 gid。更改 $(后需要检查 $! 以便发现可能出现的错误。

(助记：圆括号用来将事物分组。当使用 setgid 时，实际 gid 是你离开的那个组。)(译注：英文“离开”和“左”都是单词“left”。)

$EFFECTIVE_GROUP_ID

$EGID

$)

本进程的有效 gid。如果你使用的机器支持同时属于多个组，则该变量给出的是被空格隔开的所在组列表。第一个数值为 getegid() 的返回值，后续的值是 getgroups() 的返回值，其中可能有和第一个值相同的项。

类似地，赋给 $) 的值也必须是一个空格隔开的数值列表。第一个数设置有效 gid，其余部分(若存在)则传给 setgroups()。要达到向 setgroups() 传递空列表的效果，只需重复一遍新设置的有效 gid；例如，要将有效 gid 强制为 5 并向 setgroups() 传入空列表，就要这么写： $) = "5 5" 。

可以用 POSIX::setgid() 同时改变有效 gid 和实际 gid (只用单个数值参数)。更改 $) 后需要对 $! 进行检查以便发现可能出现的错误。

(助记：圆括号用来分组事物。当使用 setgid 时，有效 gid 就是你需要的那个组)(译注：原文为“that's right for you”，难以表达“right”的含义。)

$<、$>、$(和 $) 只能在支持对应的 set[re][ug]id() 例程的机器上进行设置。$(和 $) 只能在支持 setregid() 的机器上互换。

$PROGRAM_NAME

$0

包含当前运行程序名。

在一些(注意：不是全部)操作系统下，向 $0 赋值可以改变 ps 程序所看到的参数域。某些平台上你可能需要用特殊的 ps 选项或其他的 ps 才能看到这个改变。修改 $0 作为指示当前程序状态的一种方法，要比用来隐藏你在运行的程序更有用。(助记：同 sh 和 ksh 一样。)

注意 $0 的最大长度受相关平台的限制。多数极端情况下可能被限制在原始的 $0 所占据的空间之内。

在某些平台上可能会附加一些填充字符(例如空格)在 ps 显示出的修改名称之后。有些平台上这种填充会充满原参数域，并且不受你的控制(例如 Linux 2.2)。

BSD 用户注意：设置 $0 不会完全把“perl”从 ps(1) 的输出中抹去。例如，将 $0 设置为 "foobar" 可能产生 "perl: foobar (perl)"("perl: " 前缀和 " (perl)"后缀是否

显示出来依赖于确切的 BSD 种类及版本)。这是操作系统 的一个特性，Perl 没法改变它。

在多线程脚本中 Perl 对所有线程等同视之，因此任何线程都能更改自己的 $0 副本，并且这一更改对 ps(1) 也是可见的(假设操作系统肯合作)。注意其他线程 所看到的 $0 不会被改变，因为它们有自己的副本。

$[

数组第一个元素以及字符串中首个字符的索引号。默认为 0，但理论上你可以将其设为 1 ，以使 Perl 在处理下标或对 index() 和 substr() 函数求值时表现得更 像 awk(或 Fortran)。(助记：[代表下标开始。)

Perl 5 出现后，向 $[赋值被当作一个编译器指示符，因此不会影响任何其他文件 的行为。(这就是为什么你只能赋给它编译期常量的原因。) 强烈建议不要使用该变量。

注意，不像其他编译期指示符(例如 strict)，对 $[的赋值对于同一文件的外层词法 作用域来说是可见的。然而你可以对其使用 local()，从而将它的值严格限定在单个词法 块之内。

$]

Perl 解释器的版本 + 补丁级别 / 1000。该变量可用来判定一个脚本是否运行在恰当版本 范围的 Perl 解释器上。(助记：该版本的 Perl 是否是正确的类别？)(译注：英文“right bracket”可以表示正确的类别，字面意思则是右方括号。) 例：

```
warn "No checksumming!\n" if $] < 3.019;
```

在运行的 Perl 解释器太老时导致失败的一种简便方法可参见 use VERSION 和 require VERSION 的文档

为了避免浮点数的不精确性，在测试该变量时你可能更希望用不等测试 < 和 >，而不是包含有相等的测试：<=、== 和 >=。

浮点数表示法有时会导致不精确的数值比较结果。另一种允许对 Perl 版本进行精确的字符串 比较的现代化表示法请见 $^V。

$COMPILING

$^C

同开关 -c 相关联的标志的当前值。主要用于 -MO=...，以允许代码在编译时改变自身 行为，例如在编译期处理 AUTOLOAD 而不是像平时那样延迟载入。见 perlcc。设置 $^C = 1 同调用 B::minus_c 类似。

$DEBUGGING

$^D

调试标志的当前值。(助记：-D 开关的值。)可以读取或设置。同它的命令行等价物类似，你 可以使用数值或符号值，例如 $^D = 10 和 $^D = "st"。

$SYSTEM_FD_MAX

$^F

系统文件描述符最大数量，通常为 2。系统文件描述符会被传递到 exec() 出来的 进程里去，而数值高于它的文件描述符则不会。另外在 open() 时，若 open() 调用 失败系统文件描述符会保持不变。(普通文件描述符在尝试 open() 之前会被关闭。) 文件描述符的 exec 时关闭状态是根据对应的文件、管道或套接字打开时的 $^F 值来决定的，而不是 exec() 时的 $^F 值。

$^H

警告：该变量应严格限制在内部使用。其存在性、行为以及内容可能不经提醒就被改变。

该变量包含 Perl 解释器的编译期提示。在一个块编译结束时该变量会恢复到解释器开始 编译那个块时的值。

当 perl 开始解析任何提供了词法作用域的块构造时(例如 eval 体、require 的文件 、子程序体、循环体或条件块)，$^H 的已有值会被保存下来，但不会发生变化。当 该块的编译完成时，$^H 恢复到保存的值。在保存和恢复其值的两点之间，BEGIN 块中 执行的代码可以任意改变 $^H 的值。

该行为提供了词法作用域语义，并被应用在像 use strict 指示符这样的地方。

其值应为一个整数；不同的位代表不同的指示标志。例如：

```
sub add_100 { $^H |= 0x100 }

sub foo {
  BEGIN { add_100() }
  bar->baz($boon);
}
```

考虑在 BEGIN 块执行过程中发生了什么。此时 BEGIN 块已经通过编译，而 foo() 函数体尚未完成编译。因此 $^H 的新值仅在 foo() 函数体编译时可见。

将上面的 BEGIN 块替换为：

```
BEGIN { require strict; strict->import('vars') }
```

即演示了 use strict 'vars' 是如何实现的。下面是同一词法指示符的一个条件化 版本：

```
BEGIN { require strict; strict->import('vars') if $condition }
```

%^H

警告：该变量应严格限定在内部使用。其存在性、行为以及内容可能不经提醒就被改变。

%^H 散列表提供与 $^H 相同的作用域语法。这使其在实现词法作用域内的指示符时非常 有用。

$INPLACE_EDIT

$^I

原地编辑扩展的当前值。设置为 undef 时关闭原地编辑。(助记：-i 开关的值。)

$^M

默认情况下，内存耗尽是一个不可捕捉的致命错误。然而在适当编译的版本中，Perl 可以将 $^M 的内容用作 die() 后的紧急内存池。假设你的 Perl 编译时使用 了 -DPERL_EMERGENCY_SBRK 选项以及 Perl 自己的 malloc。那么

```
$^M = 'a' x (1 << 16);
```

将会分配一个 64K 的缓冲区以备紧急情况时使用。如何打开该选项的相关信息请参见 Perl 发行版中的 INSTALL 文件。为了防止无意中使用这一高级特性，该变量没有 对应的 English 长名。

$OSNAME

$^O

为当前 Perl 副本编译时所处的操作系统名称，在配置过程中即确定。其值同 `$Config{'osname'}` 相同。另见 Config 及 perlrun 中说明的 -V 命令行开关。

在 Windows 平台下，$^O 并不十分有用：因为它总是 `MSWin32`，而无法表示出 95/98/ME/NT/2000/XP/CE/.NET 之间的区别。请用 Win32::GetOSName() 或 Win32::GetOSVersion() (见 Win32 及 perlport) 区分这些变种。

${^OPEN}

由 PerlIO 使用的一个内部变量。是由 \0 字节分开的两部分组成的一个字符串， 前一部分描述输入层，后一部分描述输出层。

$PERLDB

$^P

供调试支持用的内部变量。其不同位所代表的意义很可能改变，但目前表示的是：

1. x01
   进入/退出调试子程序。
2. x02
   逐行调试。
3. x04
   关闭优化。
4. x08
   为后续的交互检查预留更多数据。
5. x10
   保留子程序定义时所处源代码行的信息。
6. x20
   启动时打开单步调试。
7. x40
   报告时用子程序地址而不是名称。
8. x80
   还要报告 `goto &subroutine`。
9. x100
   根据编译的位置为 eval 提供表现内容较多的“文件”名。
10. x200
    根据编译的位置为匿名子程序提供表现内容较多的名称。
11. x400
    进入/退出调试断言子程序。

某些位仅同编译期相关，还有一些是运行期相关的。这是一种新机制，以后可能 会修改其细节。

$LAST_REGEXP_CODE_RESULT

$^R

上一次成功的 (?{ code }) 正则表达式断言(见 perlre)的求值结果。 可以被改写。

$EXCEPTIONS_BEING_CAUGHT

$^S

解释器当前状态。

$^S	状态
undef	解析模块/eval
true (1)	正在执行一个 eval
false (0)	其他

第一个状态也可能在 $SIG{__DIE__} 和 $SIG{__WARN__} 处理程序内产生。

$BASETIME

$^T

程序开始运行的时间，是从格林威治标准时刻(1970 年初)开始的秒数。由 -M 、-A 和 -C 文件测试所返回的结果是基于该值的。

${^TAINT}

反映污染模式是否打开。1 表示打开(程序用 -T 运行)，0 表示关闭，-1 表示仅 打开了污染警告(即使用了 -t 或 -TU)。

${^UNICODE}

反映了 Perl 的确切 Unicode 设置。关于其可能值的更多信息请见 perlrun 文档 中对 -C 开关的描述。该变量在 Perl 启动时被设置，然后就是只读的了。

$PERL_VERSION

$^V

Perl 解释器的修订号、版本号以及子版本号，由具有这些序数的字符组成的字符串表示 。因此 Perl v5.6.0 中该变量等于 chr(5) . chr(6) . chr(0) 且 $^V eq v5.6.0 会返回真值。注意该字符串值中的字符可能处于 Unicode 范围内。

该变量可用来确定某个脚本是否在正确版本范围内的 Perl 解释器上运行。(助记：用 ^V 进行版本控制。) 例如：

```
warn "No \"our\" declarations!\n" if $^V and $^V lt v5.6.0;
```

可以用 sprintf() 的 "%vd" 转换将 $^V 变成等价的字符串表达形式：

```
printf "version is v%vd\n", $^V;  # Perl's version
```

关于在运行的 Perl 解释器过于古老时产生失败的方便方法，请参见 use VERSION 和 require VERSION 的文档。

较老的 Perl 版本表示法请见 $]。

$WARNING

$^W

警告开关的当前值，当使用 -w 时初始化为真，否则为假，不过可以直接修改。(助记：同 -w 开关相关。) 另见 warnings。

${^WARNING_BITS}

当前由 use warnings 指示符打开的警告检查集合。更多细节参见 warnings 的 文档。

$EXECUTABLE_NAME

$^X

是用于执行当前 Perl 副本的名称，来自 C 的 argv[0]。

根据主机操作系统，$^X 的值可能是 perl 程序文件的一个相对或绝对路径名、或者是用于调用 perl 而不是 perl 程序文件路径名的字符串。另外，多数 操作系统允许调用不在 PATH 环境变量中的程序，因此并不保证 $^X 的值一定 在 PATH 中。对 VMS 来说，该值可能包含一个版本号。

通常可以用 $^X 的值再次产生和当前运行的 perl 相同的一个独立副本，例如，

```
@first_run = `$^X -le "print int rand 100 for 1..100"`;
```

但考虑到并不是所有的操作系统都支持分叉或捕获命令的输出，这条复杂的语句 也许不可移植。

将 $^X 的值用作一个文件的路径名并不安全，因为某些为可执行文件使用强制后缀 的操作系统在调用一个命令时并不需要使用该后缀。要将 $^X 的值还原为路径名， 可以用如下语句：

```
# Build up a set of file names (not command names).
use Config;
$this_perl = $^X;
if ($^O ne 'VMS')
  {$this_perl .= $Config{_exe}
      unless $this_perl =~ m/$Config{_exe}$/i;}
```

由于许多操作系统允许任何对 Perl 程序文件具有读权限的用户复制该文件、对其打 补丁并执行之，对安全敏感的 Perl 程序员应注意调用 perl 的安装副本而不是 $^X 引用的副本。下面的语句可以完成该目的，产生一个可以作为命令或当作文件引用的 路径名。

```
use Config;
$secure_perl_path = $Config{perlpath};
if ($^O ne 'VMS')
  {$secure_perl_path .= $Config{_exe}
      unless $secure_perl_path =~ m/$Config{_exe}$/i;}
```

ARGV

对 @ARGV 里的命令行文件名进行迭代的特殊文件句柄。通常写为角操作符 <> 中的空文件句柄。注意目前 ARGV 仅在 <> 操作符内具有这一 特殊效果；在其他位置上它只是一个对应于 <> 打开的最后一个文件的普通 文件句柄。特别地，将 *ARGV 作为参数传递给期望一个文件句柄的函数时， 函数不一定能自动读取 @ARGV 中所有文件的内容。

$ARGV

当读取 <> 时包含当前文件名。

@ARGV

数组 @ARGV 含有脚本的命令行参数。$#ARGV 通常是参数数量减 1，因为 $ARGV[0] 是第一个参数，而不是程序本身的命令名称。命令名称请见 $0。

ARGVOUT

用 -i 进行原地编辑处理时，这是指向当前打开的输出文件的特殊文件句柄。当你需要进行 大量插入操作而不想一直修改 $_ 时，该句柄相当有用。关于 -i 开关请参考 perlrun。

@F

当打开自动分割模式时，数组 @F 包含读入的一行所产生的域内容。关于 -a 开关请参考 perlrun。该数组是包内的特殊变量，当运行在 strict 'vars' 模式下且不处于 main 包 内时，必须对其进行声明或给出完整的包名。

@INC

数组 @INC 包含一个路径列表，do EXPR、require 和 use 结构都会在该列表中查找自己 所需的库文件。它的初始值由所有 -I 命令行开关的参数、默认 Perl 库目录(如 /usr/local/lib/perl) 和代表当前目录的 "." 依次组合而成。(当使用 -T 或 -t 开启 污染检查开启时则不会把 "." 附加在后面。) 若需要在运行时修改该变量，你应该使用 use lib 指示符，以便能正确载入平台相关的库：

```
use lib '/mypath/libdir/';
use SomeMod;
```

你还可以直接在 @INC 中放入 Perl 代码，以达到在文件包含系统中插入拦截点的目的。这些拦截点 可以是函数引用、数组引用或 bless 过的对象。细节请参考 require in perlfunc

@_

在某个函数内，数组 @_ 包含传递给该函数的所有参数。参见 perlsub。

%INC

散列表 %INC 含有若干项，每一项都代表由 do、require 和 use 运算符包含进来的一个 文件。散列键是包含处给定的文件名(模块名已转换为路径名)，散列值为找到该文件的位置。 require 运算符用这个散列表判断某个特定文件是否已经被包含过。

若某个文件是由拦截点(例如一个函数引用，这些拦截点的说明参见 require in perlfunc) 载入的， 那么默认插入 %INC 的是这个拦截点而不是文件名。但要注意的是，拦截点可能已经自行修改了 %INC 中的对应项以提供某些特别信息。

%ENV

$ENV{expr}

散列表 %ENV 含有当前的环境。对 ENV 设置值会改变后续 fork() 出来的所有子进程的环境。

%SIG

$SIG{expr}

散列表 %SIG 包含信号对应的处理器。例如：

```
sub handler {      # 第一个参数是信号名称
   my($sig) = @_;
   print "Caught a SIG$sig--shutting down\n";
   close(LOG);
   exit(0);
}

$SIG{'INT'}  = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'}  = 'DEFAULT';  # 恢复默认行为
$SIG{'QUIT'} = 'IGNORE';   # 忽略 SIGQUIT
```

设置为 'IGNORE' 值通常具有忽略该信号的效果，除了 CHLD 信号以外。对于这一特殊情况 的详细说明请见 perlipc。

下面是其他一些例子：

```
$SIG{"PIPE"} = "Plumber";  # 假定为 main::Plumber (不推荐)
$SIG{"PIPE"} = \&Plumber;  # 挺好；假定为当前的 Plumber
$SIG{"PIPE"} = *Plumber;   # 有点儿深奥
$SIG{"PIPE"} = Plumber();  # 啊！ Plumber() 返回的是什么？
```

请确保没有使用裸字作为一个信号处理器的名字，免得产生无意中的调用。

如果系统有 sigaction() 函数，就用它来安装信号处理器。这意味着你能得到可靠的信号处理方式。

默认的信号投递策略在 Perl 5.8.0 中从立即发送(也即“不安全”)改为了延迟发送，即“安全信号” 。更多信息请见 perlipc。

用 %SIG 散列表也可以设置特定的内部拦截点。在即将打印一条警告信息时，由 $SIG{__WARN__} 指定的例程会被调用。警告信息作为第一个参数被传递给该例程。__WARN__ 拦截点的存在会消除通常 要打印到 STDERR 上的警告。你可以利用这一点将警告保存到变量里，或者像这样将警告转变为致命错误：

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

当即将抛出一个致命异常时，由 $SIG{__DIE__} 指定的例程会被调用。错误信息作为第一个参数被 传递给该例程。当 __DIE__ 拦截例程返回后，异常处理会像拦截点不存在一样继续进行，除非拦截例程 本身通过 goto、循环退出或 die() 的方式结束。__DIE__

处理器在调用过程中被显式关闭，因此 你可以在 __DIE__ 处理器中继续 die。__WARN__ 也具有类似行为。

由于一个实现的小问题，$SIG{__DIE__} 拦截点即使在 eval() 内也会被调用。请不要利用这一点复盖 $@ 中挂起的异常或莫名其妙地重载 CORE::GLOBAL::die()。这一奇特行为在将来应该会被修正为只在 程序即将退出时调用 $SIG{__DIE__}，这也是最初的目的。不应采用任何其他形式的用法。

__DIE__/__WARN__ 处理器在一种情况下是很特殊的： 它们可能会在汇报解析器发现的(可能)错误时被调用。在这种情况下解析器可能处于不一致的状态，任何从 这类处理器中 eval Perl 代码的尝试都可能导致段错误。这意味着处理解析 Perl 时产生的警告或错误时应 极度小心，像这样：

```
require Carp if defined $^S;
Carp::confess("Something wrong") if defined &Carp::confess;
die "Something wrong, but could not load Carp to give backtrace...
    To see backtrace try starting Perl with -MCarp switch";
```

这里的第一行只有在调用处理器的对象不是解析器时才会执行载入操作。第二行只有在 Carp 可用时才打印 出回溯信息并退出程序。第三行则仅在 Carp 不可用时才会运行。

额外信息请见 die in perlfunc、warn in perlfunc、eval in perlfunc 和 warnings。

错误指示器

变量 $@、$!、$^E 和 $? 含有关于不同类型错误条件的信息，这些错误可能在执行一个 Perl 程序时产生。这些变量按照到 Perl 进程和错误报告子系统的“距离”远近排列顺序，它们分别对应由 Perl 解释器、C 库、操作系统和外部程序检测到的错误。

为了展示这些变量之间的区别，请考虑以下这个使用了单引号引起字符串的 Perl 表达式：

```
eval q{
    open my $pipe, "/cdrom/install |" or die $!;
    my @res = <$pipe>;
    close $pipe or die "bad pipe: $?, $!";
};
```

当这条语句执行之后，4 个变量都有可能被设置。

在需要 eval 的字符串没有通过编译(若 open 或 close 导入的原型错误则可能发生)或者 Perl 代码在执行过程中 die() 掉，则 $@ 变量会被设置。这些情况下 $@ 的值是编译错误信息或 die 的 参数(其中会内插 $! 和 $?)。(另见 Fatal。)

上面的 eval() 表达式执行后，open()、<PIPE> 和 close 被翻译成对 C 运行库的调用，继而 进入操作系统内核。若其中某个调用失败，则 $! 会设置为 C 库的 errno 值。

在少数操作系统下，$^E 可能含有更详细的错误指示，例如“CDROM 仓门没有关闭”。不支持扩展错误 信息的系统只是将 $^E 设置为和 $! 一样的值。

最后，$? 在外部程序 /cdrom/install 失败时设置为非 0 值。高 8 位反映出该程序遇到的特定错误 条件(程序的 exit() 值)，低 8 位反映失败方式，例如信号致死或核心转储，细节参见 wait(2)。对比 仅在检测到错误条件时才设置的 $! 和 $^E，变量 $? 在每个 wait 或管道 close 时都会 设置并冲掉旧值。这一行为更接近 $@，后者在每次 eval() 后总是在失败时设置并在成功时清除。

更多细节请分别参见 $@、$!、$^E 和 $? 各自的说明。

关于变量名语法的技术事项

Perl 中的变量名可以有多种格式。通常，它们要以下划线或字母开头，这种情况下变量名可以任意长(但有 251 个字符的内部限制)且可包含字母、数字、下划线或特殊序列 :: 和 '。最后一个 :: 或 ' 之前的 部分被当作包限定符；参见 perlmod。

Perl 变量名也可以是一串数字、单个标点或控制符。这些名字都被 Perl 保留用作特殊用途；例如，全数字的 名字用来在正则表达式匹配之后保存反向引用捕获的数据。Perl 对单个控制符的名字具有一套特殊语法：它将 ^X (脱字符后跟 X) 理解为 Ctrl-X 字符。例如，记法 $^W (美元符 脱字符 W) 是一个名字 为单个字符 Ctrl-W 的标量变量。这要比在程序里键入一个字面上的 Ctrl-W 好一些。

最后，在 Perl 5.6 中引入了一个新特性，Perl 变量名可以是以控制符(更进一步，也可以是脱字符)开头的字母 数字字符串。这些变量必须写成 ${^Foo} 的形式；大括号是必需的。${^Foo} 代表了一个名字是 Ctrl-F 后跟两个 o 的标量。这些变量被 Perl 保留作将来的特殊用途，但以 ^_ (Ctrl-下划线或 脱字符-下划线)开头的那些除外。以 ^_ 开头的名字将不会在 Perl 的未来版本中产生任何特殊含义；因而 可以在程序中安全使用这类名字。但 $^_ 本身是保留的。

以数字、控制符或标点字符开头的 Perl 标识符不受 package 声明的影响，它们始终被强制在 main 包里； 另外它们也不受 strict 'vars' 错误的约束。其他一小部分名字也具有同样的豁免权：

```
ENV          STDIN
INC          STDOUT
ARGV           STDERR
ARGVOUT        _
SIG
```

特别地，不管当前作用域中的 package 声明如何，新形式的特殊变量 ${^_XYZ} 总是被放置在 main 包里。

3 BUGS

由于 Perl 实现时一个不幸的事故，use English 会使程序中所有正则表达式匹配产生显著的效率降低，不管它们 是否出现在 use English 的作用域里。因此，强烈不推荐在库里使用 use English。更多信息请参见 CPAN 上的 Devel::SawAmpersand 模块文档 (http://www.cpan.org/modules/by-module/Devel/)。

在异常处理器中不应该考虑 $^S。$SIG{__DIE__} 的当前实现令人伤心，很难用它跟踪错误。请避免使用它并 用 END{} 或 CORE::GLOBAL::die 重载来代替之。

4 TRANSLATORS

ChaosLawful