

编程和写好程序是两个相关但有所区分的技能。如果我们仅编写一次性程序并不再需要修改或维护它们，如果我们的程序毫无缺陷，如果我们的无需在耗内存和耗时间之间做出选择，并且如果我们永远不用和他人一起工作，我们就不必担心我们的程序写得有多差。要写好程序，你必须基于特定的时间、资源侧重点理解候选解决方案和进一步计划间的区别。

写好 *Perl* 程序意味着理解 *Perl* 是如何工作的。同时也意味这个培养良好的编程品味。要培养这种技能，你必须不断练习编写和维护代码，并阅读优秀的代码。此处并无捷径可走——但通过遵守如下指导，你可以提高练习效率。

## 编写可维护的 *Perl* 程序

程序越易理解和修改越好。这就是 可维护性。假设将你现在正编写的程序放一边，六个月以后回来修改缺陷或是添加功能。代码越一样维护，修改是遇到的人工复杂度 就越小。

要编写可维护的 *Perl* 程序，你必须：

- \* 去掉重复 *Perl* 提供了不少使用抽象消去重复的机会。函数、对象、角色和模块，举例来说，允许你定义程序和解决方案的模型。

程序中重复越多，做出必要修改时花的精力越多，并且很可能会忘记修改每一必要处。重复越少，说明你很可能找到了问题的有效解决。最佳设计让你在添加功能的同时减少整体代码量。

- \* 正确命名实体 系统中每一样由你命名的事物——函数、类、方法、变量、模块——可以有助也可以妨害代码的清晰程度。好的一面是，你可以通过命名这些实体来揭示你对问题的理解以及你设计的内聚力。你的设计就是在讲述一个故事，其中经斟酌的一词一字都有助于在日后维护代码时帮你记起故事的来龙去脉。

- \* 避免小聪明 新手有时候误将小聪明认作简明。简明的代码避免非必要的结构和复杂性。要小聪明的代码通常倾向于展示聪明而非简明。*Perl* 提供了许多解决相似问题的手段。通常其中之一更加可读。有时候某中形式的解更加快速或简单，通常某一解法的上下文特征更加明显。

你无法总是避开 *Perl*

中黑暗的角落，而且有部分问题需要高效解决的小聪明。仅有良好的代码品味和经验能帮助你估计小聪明的合适程度。按经验来看，如果你认为在你的同事面前解释你的解法更使你感到骄傲，你的代码更可能包含不必要的复杂性。

如果你 确实

需要编写小聪明代码，请将其封装在简单的接口之后并详尽地用文档记下你 的聪明才智。

- \* 拥抱简洁 给出两个解决相同问题的程序，简洁的那个几乎总是更易于维护。简洁并非让你避开高级 *Perl* 知识，或是避免使用库，或是扫清几百行过程式代码。

简洁意味着你高效地解决手边的问题而不用增加任何你不需要的东西。没有任何理由避开错误检查或验证数据或不注重安全性。相反，应该重点思考究竟什么是重要的。有时候你不需要框架、对象或复杂的数据结构。有时候你需要。简洁意味着你了解其中的区别。

## 编写惯用语化的 *Perl* 程序

*Perl* 从其他语言及编程以外的大千世界借鉴各式思想。*Perl* 倾向于使其 *Perl* 化来占有这些思想。要写好 *Perl* 程序，你必须了解有经验的 *Perl* 程序员是如何写程序的。

- \* 理解社区的智慧 *Perl* 社区通常就技巧进行辩论，有时非常激烈。甚至这些反对的声音也会给特点设计取舍和风格带来启示。你了解自身特定的需求，但 *CPAN* 作者，*CPAN* 开发人员，本地的 *Perl* 贩子小组以及其他程序员拥有解决类似问题的经验。和他们聊聊。阅读他们公开的代码。提问。并互相学习。

- \* 遵循社区标准 *Perl* 社区并不总是正确的，特别是在你的需求特别专一或独特时，但社区本身一直持续运作以尽可能广泛地解决各类问题。*Perl* 的测试和打包工具在代码符合 *CPAN* 发行

规则时可最高效地工作。遵守编码、文档、打包、测试、代码发布的各项标准，利用好这些工具。

类似地，**CPAN**上的发行包如 `Perl::Critic`、`Perl::Tidy` 以及 `CPAN::Mini` 可让你的工作 更简单更轻松。

\* 阅读代码 加入诸如 `Perl Beginners` (<http://learn.perl.org/faq/beginners.html>) 之类的邮件列表，注册一个 `PerlMonks` (<http://perlmonks.org/>) 帐号，使自己沉浸在 `Perl` 社区 (<http://www.perl.org/community.html> 包括了丰富的链接)。你将会有非常多的机会见识他人是如何解决问题的（无论方法是好是坏）。学习优秀的方法（通常很明显），并从不那么好的方法中汲取教训。

就他人发帖提出的问题编写几行代码给出自己的解答，这是一种学习的好方法。

## 编写高效的 `Perl` 程序

了解 `Perl` 的语法和语义只是一个起步。你之能通过 鼓励 良好设计的习惯达成良好的设计。

\* 编写可测试的代码 也许确保你可以维护一段代码的最佳方法就是编写一个高效的测试套件。编写良好的测试代码和设计程序一样，都锻炼了设计技能；绝对不要忘记，测试代码仍是代码。即便如此，一个良好的测试套件会给你带来信心，让你知道你可以修改程序并不会打破你关心的程序行为。

\* 模块化 将你的代码分割为单独的模块强制推行封装和抽象边界。将此培养成一种习惯后你就能认出那些功能过于臃肿的代码单元。你也将识别出结合过于紧密的多个模块。

模块化同时强制你处理各个层面的抽象；你必须考虑系统中的各个实体如何协作。没有比将系统修改为高效抽象更能学到抽象的价值了。

\* 利用 `CPAN` 使任何 `Perl 5` 程序能力倍增的是这个唾手可得、令人惊叹的可重用代码库。数千开发人员已经编写了几万个模块，可以解决的问题超乎你的想象，`CPAN` 仍在继续成长。社区有关文档、打包、安装、测试的规范保证了代码质量，并且，以 `CPAN` 为中心的现代化 `Perl` 已经帮助 `Perl` 社区在知识、智慧和效能上发展壮大。

当可能时，请先搜索

`CPAN`——并询问你的社区伙伴——征询解决问题的建议。你甚至可以报告缺陷，或提交补丁，再或自己编写 `CPAN`

模块发行版。没有什么比帮助解决他人的问题更能展示你是一个高效的 `Perl` 程序员了。

\* 建立合理的编码标准 有效的指导为错误处理、安全性、封装、`API` 设计、项目布局以及其他可维护性考虑建立对策。出色的指导随着你和你的团队互相理解及项目跟进而革新。编程的目的是解决问题，建立编码标准的目的是帮助你清晰地表达意图。

## 文件

大部分程序都以某种方式与外界交互，其中交互又多于文件中发生：读、写、修改。`Perl` 早期作为系统管理和文本处理语言的历史，使其非常适用于文件修改。

## 输入和输出

与程序外界交互的主要机制是通过 文件句柄。文件句柄代表输入输出通道的某种状态，例如程序的标准输入输出、程序读取或写入的某个文件，给定文件的位置。每个 `Perl 5` 程序都有三个文件句柄可用，`STDIN`（程序的输入）、`STDOUT`（程序的输出），以及 `STDERR`（程序的错误输出）。

默认地，用 `print` 或 `say` 打印的内容将流向 `STDOUT`，同时，错误和警告和用 `warn()` 打印的内容将流向 `STDERR`。这种输出分离允许你将有用的输出及错误重定向到两个不同的地方——例如，输出文件和错误日志。

尚有其他可用的文件句柄；`DATA` 代表当前文件。当 `Perl` 完成对文件的编译，它留着

包全局文件句柄 `DATA` 不动，并在编译单元尾打开它。如果你在 `__DATA__` 或是 `__END__` 后放置字符串，你可以从 `DATA` 文件句柄处读取它们。对于小型自包含程序来说非常实用。`perldoc perldata` 对此特性进行了详细的描述。

除标准文件句柄之外，你可以用 `open` 关键字打开自己的文件句柄。为读取打开某文件：

```
open my $fh, '<', 'filename'
    or die "Cannot read '$filename': $!\n";
```

第一个操作数是存放文件句柄的词法变量。第二个操作数是 文件模式，它决定了文件句柄操作的类型。最后的操作数是文件名。如果 `open` 失败，`die` 分句将抛出异常，用 `$!` 的内容给出了打开失败的原因。

除了文件，你还可以在标量上打开文件句柄：

```
use autodie;

my $captured_output;
open my $fh, '>', \"captured_output;

do_something_awesome( $fh );
```

这类文件句柄支持所有现存的文件模式。

你也许会碰到使用双参数式 `open()` 的早期代码：

```
open my $fh, "> $some_file"
    or die "Cannot write to '$some_file': $!\n";
```

文件模式和文件名之间清晰界限的缺失，在将不受信任的输入内插入第二个操作数时，会使意外有机可乘

当你阅读这句时，训练自己进行如下思考：“这段代码是不是会导致安全问题？”  
你可以安全地将任何双参数式 `open` 替换为三参数式，不用担心有任何功能上的损失。

`perldoc perlopen` 提供了更多有关 `open` 奇形怪状用法的细节，包括它启动及控制其他进程的能力，同时也介绍了可以对输入输出进行更加精细控制的 `sysopen` 的用法。`perldoc perlfaq5` 包含处理许多常规 IO 任务的实用代码。

## 读取文件

给出为读打开的文件句柄，可用 `readline` 操作符从中读取内容，此操作符也写作 `<>`。最为常见的惯用语是用 `while()` 循环从文件中一次读取一行：

```
use autodie;
open my $fh, '<', 'some_file';

while (<$fh>)
{
    chomp;
    say "Read a line '$_'";
}
```

在标量上下文中，`readline` 遍历可以通过该文件句柄读取的每一行，直到它遇到文件结尾（`eof()`）。每次迭代都会返回下一行。在遇到文件结尾后，每次迭代都返回 `undef`。此 `while` 惯用语明确地检查迭代所用变量的是否定义，以便确保只在文件结尾处结束循环。

从 `readline` 读取的每一行包含标示行结束的一个或多个字符。大多数情况下，这是一个平台相关的序列，可由换行（`\n`），硬回车（`\r`），或者两者组合（`\r\n`）构成。使用 `chomp` 移除平台相关的换行序列。

综合上述，Perl 5 中读取文件最清晰的方式是：

```
use autodie;

open my $fh, '<', $filename;

while (my $line = <$fh>)
{
    chomp $line;
    ...
}
```

如果读取的不是 文本——而是 二进制 数据——在读写之前请在文件句柄上启用 `binmode`。这个关键字告诉 Perl 将来自该文件句柄的数据视作纯数据。出于对平台可移植性等考虑，Perl 不会以任何方式修改它。虽然在这种情况下类 Unix 平台不那么 需要使用 `binmode`，可移植的程序无论如何都应使用它。更多有关 `binmode` 的特性请参见 `unicode`。

## 写入文件

给定一个为写打开的文件句柄，你可以使用 `print` 或者 `say` 来写文件：

```
use autodie;
open my $out_fh, '>', 'output_file.txt';

print $out_fh "Here's a line of text\n";
say $out_fh "... and here's another";
```

注意文件句柄和后续操作数之间是没有逗号的。

由 Damian Conway 所著的 Perl 最佳实践 建议养成将文件句柄包裹在大括号中的习惯。这在对包含在集合变量内的文件句柄进行语法分析并解歧时是必须的，无论如何这是一个值得培养的好习惯。

你可以用 `print` 或 `say` 写入整列值，在这种情况下，Perl 5 使用特殊全局变量 `$`，作为列表值的分隔符。Perl 同时用 `$\` 的值作为 `print` 和 `say` 结尾的参数。

## 关闭文件

当你完成文件操作后，你可以用 `close` 明确地关闭它，或者让文件句柄自行超出作用域，在这种情况下 Perl 会替你关闭它。明确调用 `close` 的好处是，你可以检查特定的错误——同时也可以从其中恢复，例如，存储设备容量不足或是网络状况欠佳。

和往常一样，`autodie (autodie)` 会为你处理这类检查：

```

use autodie;

open my $fh, '>', $file;

...

close $fh;

```

## 特殊文件句柄变量

读取每一行，Perl 5 都会增加 `$.` 的值，它可以用作行计数器。

`readline` 将 `$/` 当前的内容用作行结束序列（详例参见 `dynamic_scope`）。此变量的值默认为当前平台上合适的文本文件行结束符序列。事实上，“行”一字用法不当。你可以将 `$/` 设置为任意字符序列 .....但别设置成正则表达式，因为 Perl 5 并不支持这样做。。这在一次读取高度结构化数据的一条记录时很有用。

默认地，Perl 使用 缓冲式输出，即仅在数据足够多且超出某一限制时才进行 IO 操作。这使得 Perl 可以批量处理昂贵的 IO 操作而不必每次只写非常少量的数据。有时你想尽快发送手边的数据而不想等待缓冲——特别是你在编写一些连接至其他程序或行式网络服务的命令行过滤器时。

当前活动的输出文件句柄缓冲由 `$|` 变量控制。当设置为非零值时，Perl 在每次对此文件句柄进行写入操作后都会冲洗输出。当设置为零值，Perl 仍将采用默认的缓冲策略。

作为对全局变量的代替，可以在词法文件句柄上调用 `autoflush()` 方法。请确认加载 `FileHandle` 在先，否则你将不能在词法文件句柄上调用此方法：

```

use autodie;
use FileHandle;

open my $fh, '>', 'pecan.log';
$fh->autoflush( 1 );

...

```

当加载完 `FileHandle`，你还可以使用其中的 `input_line_number()` 和 `input_record_separator()` 方法，作为和 `$.`、`$/` 对应的替代。参考 `perldoc FileHandle` 和 `perldoc IO::Handle` 以获取 更多信息。

如果你使用的是 Perl 5.12 或更新版本，`IO::File` 已经被 `FileHandle` 所替代。

## 目录和路径

你也可以用 Perl 5 修改目录和文件路径。操作文件夹和操作文件差不多，除了你不可以 写目录之外 作为代替，你可以保存、移动、重命名和删除其中的文件。。你可以用 `opendir` 打开目录句柄：

```

use autodie;

```

```
opendir my $dirh, '/home/monkeytamer/tasks/';
```

从文件夹中读取的语言关键字是 `readdir`。就像 `readline`，你可以一一遍历目录内容，或者将它们一举赋值给数组：

```
# 迭代
while (my $file = readdir $dirh )
{
    ...
}

# 展开为列表
my @files = readdir $otherdirh;
```

作为 5.12 新增特性，`while` 循环中 `readdir` 会设置 `$_`，正如 `while` 中的 `readline`：

```
use 5.012;
use autodie;

opendir my $dirh, 'tasks/circus/';

while (readdir $dirh)
{
    next if /^\.\/;
    say "Found a task $_!";
}
```

例子中可笑的正则表达式在 Unix 和类 Unix 系统上跳过所谓 隐藏文件，即前缀点号默认防止它们出现在文件列表中。它同时也跳过了每次 `readdir` 调用返回的最初两个文件，特别是 `.` 和 `..`。这两个文件代表了当前目录和上级目录。

注意由 `readdir` 返回的名称是 相对于 目录自身的。换句话说，如果 `tasks/` 目录包含三个名为 `eat`、`drink` 和 `be_monkey` 的文件，`readdir` 将返回 `eat`、`drink` 以及 `be_monkey` 而非 `tasks/eat`、`tasks/drink` 和 `task/be_monkey`。相反地，绝对路径是一个完全限定于文件系统的路径。

可以通过越过作用域或用 `closedir` 关键字关闭一个目录句柄。

## 操作路径

Perl 5 用 Unix 式的视角看待世界，至少对于文件系统部分是这样的。即便不使用类 Unix 平台，Perl 仍将解析适合于你操作系统和文件系统的 Unix 式路径。换句话说，如果你使用 Microsoft Windows，你可以和使用 `C:\My Documents\Robots\Caprica Six\` 一般方便地使用 `C:/My Documents/Robots/Bender/` 这样的路径。

即便如此，以安全的、跨平台的行为操作文件路径意味着你必须避免字符串内插和拼接。核心模块 `File::Spec` 系列为以安全可以移植地操作文件路径提供了抽象。即便如此，理解并正确使用它们并不总是容易的。

CPAN 上的 `Path::Class` 模块为 `File::Spec` 提供了更好的接口。可以使用 `dir()` 函数创建代表目录的对象，以及使用 `file()` 函数创建代表文件的对象：

```
use Path::Class;

my $meals = dir( 'tasks', 'cooking' );
my $file  = file( 'tasks', 'health', 'exoskeleton_research.txt' );
```

.....并且你可以这样得到目录中的文件对象：

```
my $lunch = $meals->file( 'veggie_calzone.txt' );
```

.....反之亦然：

```
my $robots_dir = $robot_list->dir();
```

你甚至可以打开文件和目录的文件句柄：

```
my $dir_fh    = $dir->open();
my $robots_fh = $robot_list->open( 'r' ) or die "Open failed: $!";
```

更多信息请参见 `Path::Class::Dir` 和 `Path::Class::File` 文档。

## 文件操作

除了读写文件，你还可以像在命令行或是文件管理器中那样直接操作它们。`-x` 文件测试操作符可以提供有关系统上文件和目录的属性信息。例如，要测试某文件是否存在：

```
say 'Present!' if -e $filename;
```

`-e`

操作符只有一个操作数，即文件名或者文件、目录的文件句柄。如果文件存在，此表达式求值得真。`perldoc -f -x` 列出了所以其他的文件测试；最著名的有：

`-f`，如果操作数是普通文件，返回真值

`-d`，如果操作数是目录，返回真值

`-r`，如果操作数的文件属性对当前用户可读，返回真值

`-z`，如果操作数是非空文件，返回真值

在 Perl 5.10.1 之后，你可以用形如 `perldoc -f -r` 的方式察看这些文件测试操作的文档。

Perl 同时允许你改变当前目录。默认地，当前目录就是你启动程序的目录。核心模块 `Cwd` 允许你判断当前目录。`chdir`

关键字可以用于改变当前工作目录。这在用相对——而非绝对——路径操作文件时非常有用。

`rename` 关键字可以重命名或在目录间移动某个文件。它接受两个操作数，旧文件名和 新文件名：

```
use autodie;
```

```
rename 'death_star.txt', 'carbon_sink.txt';
```

复制文件没有对应的核心关键字，但是核心模块 `File::Copy` 提供了 `copy()` 和 `move()` 函数两者。可以使用 `unlink` 来删除一个或多个文件。这些函数和关键字在操作成功时候返回真 值，出错则设置 `$!`。

`Path::Class` 为检查特定文件属性和完整删除文件提供了便捷的方法，并且是跨平台的。

## 异常

如果一切都像预期的那样工作，编程就会变得简单些。不幸的是，要处理的文件可不一定。有时候你还会碰到磁盘空间不足、网络连接不稳、数据库拒绝写入数据等等。

异常总会发生，健壮的软件必须处理这些异常状况。如果可以从中回复，那太好了！如果不行，有时候能做的也只能是重试或至少为进一步调试将相关信息记入日志。Perl 5 以 异常的方式处理非正常状况：一个动态作用域的控制流程语法形式，让你可以在最合适的地方处理 错误。

## 抛出异常

考虑你需要为记日志打开文件这种情况。如果你不能打开该文件，就说明有错误发生。你 可以用 `die` 来抛出一个异常：

```
sub open_log_file
{
    my $name = shift;
    open my $fh, '>>', $name
        B<or die "Can't open logging file '$name': $!";>
    return $fh;
}
```

`die()` 设置全局变量 `$@` 为其参数并立即退出当前函数而 不返回任何值。如果函数调用方不明确地处理此异常，则此异常将向上传播至所有的调用者，直到有东西处理它，或者程序以一条错误信息退出。

异常抛出和处理的动态作用域与 `local` 符号 (`dynamic_scope`) 一致。

## 捕获异常

未捕获的异常最终将结束程序。有时候这是有用的；一个从 `cron` (一个 Unix 作业调度器) 运行的系统管理程序可以在填写错误日志后抛出异常；这样便可以将错误通知给系统管理员。而另一些异常并非致命的；优秀的程序可以从中恢复，或者至少保存状态并更为体面地退出。

要捕获一个异常，可以使用 `eval` 操作符的代码块形式：

```
# 也许无法打开日志文件
my $fh = eval { open_log_file( 'monkeytown.log' ) };
```

就像所有的代码块，`eval` 的代码块参数引入了新的作用域。如果文件打开成功，`$fh` 将包含此文件的文件句柄。如果失败，`$fh` 将维持未定义，且 Perl 将继续执行程序中的下一



行语句。如果 `open_log_file()` 调用了一个调用了其他函数的函数，如果其中某函数抛出了自身的异常，这条 `eval` 语句会捕获到，否则什么也不做。没什么要求异常处理器只处理你希望处理的异常。

要检查捕获的内容（或检查究竟有没有捕获到异常），检查 `$@` 的值：

```
# 也许无法打开日志文件
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# 捕获异常
B<if ($@) { ... }>
```

当然，`$@` 是一个 全局 变量。为了安全起见，你应该在意图捕获异常之前，用 `local` 本地化的它的值：

```
B<local $@;>

# 也许无法打开日志文件
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# 捕获异常
if ($@) { ... }
```

你可以按可能的异常逐条检查 `$@` 字符串值，看看是否能够处理或者是否应该重新抛出 这条异常：

```
if (my $exception = $@)
{
    die $exception unless $exception =~ /^Can't open logging file/;
    $fh = log_to_syslog();
}
```

将 `$@` 复制到 `$exception` 可以避免后续代码破坏全局变量 `$@` 值的可能。你绝对不会知道有什么在其他地方用 `eval` 代码块重置了 `$@` 的值。

你可以通过向 `die()` 传入 `$@` 再次抛出异常。

你会发现对 `$@` 的值使用正则表达式令人讨厌；你还可以对 `die` 使用 对象。诚然，这比较少见。`$@` 可能 包含任意引用，但是就实际来说差不多 95% 是字符串 5% 是对象。

作为自行编写异常处理机制的替代，参见 CPAN 发行模块 `Exception::Class`。

## 异常注意事项

正确使用 `$@` 很讲究技巧；全局变量的本质为其带来了不少微妙的缺陷：

- \* 动态作用域下使用未经 `local` 处理的变量值可能会导致其被重置
- \* 在异常捕获作用域内的对象析构可能调用 `eval` 并改变它的值
- \* 它可能包含一个覆盖自身布尔值的对象，使其返回假
- \* 一个信号处理器（特别是 `DIE` 信号处理器）可能在你不经意时修改它的值

编写非常安全和合理的异常处理器非常困难。来自 CPAN `Try::Tiny` 发行模块非常简短，易于安装，易于理解，同时也易于使用：

```
use Try::Tiny;
```

```
my $fh = try { open_log_file( 'monkeytown.log' ) }
            catch { ... };
```

不仅是语法比 Perl 5 默认的更加友好，而且该模块在你不知情的情况下处理了所有边边角角的情况。

## 内置异常

Perl 5 有部分异常状况可以通过 `eval` 块捕获。`perldoc perldiag` 把它们列为 “trappable fatal errors”。

大多数是在编译期抛出的语法错误。其他一些是运行时错误。有些异常也许值得捕获；语法错误则不值得。最有趣或者很可能因如下原因发生异常：

- \* 在上锁的哈希里使用不允许的键 (`locked_hashes`)
- \* 对非引用进行 `bless` (`blessed_references`)
- \* 在非法的调用物上调用方法 (`moose`)
- \* 在调用物上无法找到对应名称的方法
- \* 以不安全的方式使用污点 (`taint`) 值 (`taint`)
- \* 修改只读值
- \* 在类型错误的引用上使用错误的操作 (`references`)

如果你启用了致命词法警告 (`registering_warnings`)，你可以捕获因它们而起的异常。对来自 `autodie` 的异常同样适用 (`autodie`)。

## 编译命令

Perl 5 的扩展机制是模块 (`modules`)。大部分模块提供了函数可供调用的函数，或定义了一些类 (`moose`)，但有部分模块则不同，它们对语言自身的行为产生了影响。

影响编译器行为的模块称为一条 编译命令 (`pragma`)。按照惯例，编译命令的名称为小写，以示与其他模块的区别。你已经听说过的一些编译命令有：`strict` 和 `warnings`。

## 编译命令和作用域

编译命令的工作方式是向闭合的静态作用域导出特定行为或信息。编译命令的作用域与词法变量相同。在某种程度上你可以将词法变量声明看做是一种带有可笑语法的编译命令。用例子说明编译命令的作用域会更加清楚：

```
{
    # $lexical B<不> 可见; strict B<未> 生效
    {
        use strict;
        my $lexical = 'available here';
        # $lexical B<可见>; strict B<有效>
        ...
    }
    # $lexical 再次 B<不> 可见; strict B<不> 起作用
}
```

一个潜能被充分激发的 Perl 大师可以实现一个无视作用域的行为不良的编译命令，但这太不友善了。

如同词法声明影响内层作用域那样，编译命令也会如此：

```
# 整个文件范围
use strict;

{
    # 内层作用域；但是仍在 strict 生效的范围内
    my $inner = 'another lexical';
    ...
}
```

## 使用编译命令

编译命令与模块的使用机制相同。和使用模块时相同，你可以指定所需编译命令的版本，也可以向编译命令传递参数列表以便更好地控制其行为：

```
# 要求变量声明；防止裸函数名称
use strict qw( subs vars );
```

在作用域内你可以用 `no` 关键字禁用全部或部分编译命令：

```
use strict;

{
    # 准备修改符号表
    no strict 'refs';
    ...
}
```

## 常用核心编译命令

Perl 5 包含了一些有用的核心编译命令：

- \* `strict` 编译命令启用编译器对符号引用、裸字、变量声明的检查
- \* `warnings` 编译命令为不推荐的、出人意料的、古怪的行为启用可选的警告。它们不一定是错误，但可能产生多余的行为
- \* `utf8` 编译命令对源代码启用 UTF-8 编码
- \* `autodie` 编译命令（5.10.1 新增）对系统调用和关键字启用自动错误检查，减少手动检查之需
- \* `constant` 编译命令允许你创建编译期常量（也请参见来自 CPAN 的 `Readonly` 替代）
- \* `vars` 编译命令允许你声明包全局变量，诸如 `$VERSION`、导出变量（exporting）以及来自 Perl 文档的 `OO`（`blessed_references`）

另有一些来自 CPAN 的实用编译命令。两个值得细看的是 `autobox`，它为 Perl 5 核心类型（标量、引用、数组和哈希）启用了类对象行为；以及 `perl5i`，它组合并启用了许多实验性的语言扩展，使之成为一个有机的整体。这两个编译命令不经大量测试和慎重考虑可能不会出现在你的产品代码中，但它们展示了编译命令的实用和强大。

Perl 5.10.0 新增了用纯 Perl 代码编写你自己的词法编译命令的能力。`perldoc perlpragma` 阐述了如何去做，同时，`perldoc perlvar` 内对 `$^H` 的解释说明了此特性的工作原理。