

编写简单的示例程序来解决本书中的示例问题有助你从小处学习一门语言。编写现实世界中的程序则提出了比学习语言语法、或其设计原理，甚至是查找并使用语言库更高的要求。

实际编程要求你管理自己的代码：组织代码、了解其如何运作、让它在错误的意图和逻辑面前更为健壮、并以一种理性、清晰、可维护的方式完成以上目的。所幸的是，现代化的 Perl 为编写实际应用——从测试到组织源码——提供了许多工具和技巧。

测试

测试是编写并运行自动验证套件的过程，以保证软件整体或局部按预期的方式工作。从根本上来说，这是你已经无数次手动执行过程的自动化：编写一段代码，运行并检查是否正常。区别在与整个过程是否自动化。相比手动执行这些步骤并依靠人力保证每次都完美无缺，还是让计算机来处理这些重复部分。

Perl 5 提供了上佳的工具来帮助你编写良好且实用的自动化测试。

Test::More

Perl 测试始于核心模块 `Test::More` 及其 `ok()` 函数。`ok()` 接受两个参数，一个布尔值和一个描述测试目的的字符串：

```
ok( 1, 'the number one should be true' );
ok( 0, '... and the number zero should not' );
ok( '', 'the empty string should be false' );
ok( '!', '... and a non-empty string should not' );
```

最终，任何能够在程序中测试的条件将会变为一个布尔值。代码是否如期工作？一个复杂的程序也许有上千条独立的测试条件。通常，粒度越细越好。编写独立断言的目的是将一个个功能进行隔离以便了解哪些功能不正常以及做出进一步改动后哪些部分罢工了。

然而，上述代码片段并不是一个完整的测试脚本。`Test::More` 以及相关的模块要求写明测试计划，它代表了欲进行的独立测试个数：

```
use Test::More tests => 4;

ok( 1, 'the number one should be true' );
ok( 0, '... and the number zero should not' );
ok( '', 'the empty string should be false' );
ok( '!', '... and a non-empty string should not' );
```

`Test::More` 的 `tests` 参数为此程序设置测试计划。这向测试增加了一项额外的断言。如果实际执行的测试少于四项，表示有错误发生。如果多于四项，还是不对。在这种简单的情形下，该断言不那么有用，但它能够捕捉到代码中的简单到不太可能出错的那种缺陷作为一条规则，任何你吹嘘简单到不可能出错的代码会不幸地时候包含错误。。

你不必以 `import()` 参数的形式提供 `tests => ...`。你还可以在测试程序的结尾，调用 `done_testing()` 函数。虽然在程序开头包含固定的测试数目能保证只执行预期数量的测试，但有时候确认这个数量是非常痛苦的一件事。在这里情况下，`done_testing()` 将验证成功执行的测试数量——否则，你怎么可能会知道呢？

执行测试

结果就是一个功能齐全的 Perl 5 程序，它产生如下输出：

1..4

```
ok 1 - the number one should be true
not ok 2 - ... and the number zero should not
#   Failed test '... and the number zero should not'
#   at truth_values.t line 4.
not ok 3 - the empty string should be false
#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
ok 4 - ... and a non-empty string should not
# Looks like you failed 2 tests of 4.
```

此格式遵循名为 TAP, 即 Test Anything Protocol (<http://testanything.org/>) 的测试输出标准。作为此协议的一部分, 失败的测试输出诊断信息。这对于调试来说是莫大的帮助。

测试文件输出的各类断言(特别是多种失败断言)可能会很详细。在大多数情况下, 你希望了解测试是全部通过了或是其中 x、y、z 失败了。核心模块 `Test::Harness` 解析 TAP 并显示最贴切的信息。它同时提供了一个名为 `prove` 的程序, 它接手了所有这些繁重的工作:

```
$ B<prove truth_values.t>
truth_values.t .. 1/4
#   Failed test '... and the number zero should not'
#   at truth_values.t line 4.

#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
# Looks like you failed 2 tests of 4.
truth_values.t .. Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/4 subtests

Test Summary Report
-----
truth_values.t (Wstat: 512 Tests: 4 Failed: 2)
  Failed tests:  2-3
```

有很大部分显示的是一些很显然的内容: 第二、三两个测试因为零和空字符串求值得假而失败。进行双重否定布尔转换(`boolean_coercion`)即可很方便地修正这些错误:

```
ok(    B<!=> 0, '... and the number zero should not' );
ok(    B<!=> '', 'the empty string should be false'   );
```

有了这些修改, `prove` 现在显示:

```
$ B<prove truth_values.t>
truth_values.t .. ok
All tests successful.
```

更好的比较

即使所有自动测试归根究底只是一些“是真是假”的布尔条件, 将所有这些规约为一条条布尔条件仍显乏味且没有提供作进一步诊断的可能。`Test::More` 提供了若干方便函数来确保你的代码按你的意图行事。

`is()`

函数比较两个值。如果它们匹配，则测试通过。否则，测试失败并提供相关诊断信息：

```
is(      4, 2 + 2, 'addition should hold steady across the
universe' );
is( 'pancake',    100, 'pancakes should have a delicious numeric value'
);
```

按你预期的，第一项测试通过而第二项会失败：

```
t/is_tests.t .. 1/2
#   Failed test 'pancakes should have a delicious numeric value'
#   at t/is_tests.t line 8.
#       got: 'pancake'
#   expected: '100'
# Looks like you failed 1 test of 2.
```

`ok()` 只听过失败测试的行号，`is()` 显示未能匹配的值。

`is()` 对其值应用隐式的标量上下文。这意味着，例如，你可以不用明确地在标量上下文中对数组求值而检查其中元素的个数：

```
my @cousins = qw( Rick Kristen Alex Kaycee Eric Corey );
is( @cousins, 6, 'I should have only six cousins' );
```

.....虽然有些人考虑清晰度更倾向于编写 `scalar @cousins`。

`Test::More` 还提供了对应的 `isnt()` 函数，仅在所提供值不相等时通过测试。除此之外，它与 `is()` 的行为相同，也遵循标量上下文和比较类型。

`is()` 和 `isnt()` 都是通过 Perl 5 操作符 `eq` 及 `ne` 进行字符串比较。

这样几乎总是正确的，但是对于复杂的值，如，重载对象（overloading）或是双重变量（dualvars），你会更倾向使用明确的比较测试。`cmp_ok()` 函数允许你指定自己的比较操作符：

```
cmp_ok(    100, $cur_balance, '<=', 'I should have at least $100' );
cmp_ok( $monkey,      $ape, '==', 'Simian numifications should
agree' );
```

类和对象自身会以有趣的方式和测试互动。通过 `isa_ok()` 可以测试一个类或对象是否是其它类的扩展（inheritance）：

```
my $chimpzilla = RobotMonkey->new();
isa_ok( $chimpzilla, 'Robot' );
isa_ok( $chimpzilla, 'Monkey' );
```

`isa_ok()` 在失败时会提供自己的诊断信息。

`can_ok()` 验证一个类或对象是否能够执行所要求的（多个）方法：

```
can_ok( $chimpzilla, 'eat_banana' );
can_ok( $chimpzilla, 'transform', 'destroy_tokyo' );
```

`is_deeply()` 函数比较两个引用以保证它们的内容相同：

```
use Clone;
```

```

my $numbers    = [ 4, 8, 15, 16, 23, 42 ];
my $clonenums = Clone::clone( $numbers );

is_deeply( $numbers, $clonenums,
    'Clone::clone() should produce identical structures' );

```

如果比较失败，`Test::More` 将尽力做出合理的诊断指明结构间首处不等的位置。参见 `CPAN` 模块 `Test::Differences` 和 `Test::Deep`，了解更多有关可配置测试的信息。

`Test::More` 还有另外一些测试函数，但上面介绍的这些最为有用。

组织测试

`CPAN` 组织测试的标准方法是创建一个包含一个或多个以 `.t` 结尾程序的 `t/` 目录。所有的 `CPAN` 发行模块管理工具（包括 `CPAN` 基础设施自身）都能理解这套系统。默认地，当你使用 `Module::Build` 或 `ExtUtils::MakeMaker` 构建一个发行模块时，测试步骤将执行所有 `t/*.t` 文件，综合它们的输出，并按测试套件的总体结果决定测试通过 还是不过。

目前没有什么有关管理独立 `.t` 文件内容的建议，但有两种策略比较常见：

- * 每个 `.t` 文件对应一个 `.pm` 文件
- * 每个 `.t` 文件对应一个程序功能

由于大型文件较小文件难以维护，并且测试套件的粒度也是如此，对与测试文件组织方式的重要考虑之一便是可维护性。一种混合的管理方式较为灵活：由一个测试验证所有模块是否能够编译，其他测试确保每个模块都能如常工作。

通常只对当前在开发功能执行测试。如果你正向 `RobotMonkey` 添加喷火功能，那么你可能会希望执行 `t/breathe_fire.t` 测试文件。当你已经对此功能非常满意了，就可以运行全套测试以保证程序整体未受局部改动的影响。

其他测试模块

`Test::More` 依赖与名为 `Test::Builder` 的测试后端。后者管理测试计划并将测试结果 组织为 TAP。这种设计允许多个测试模块共享同一 `Test::Builder` 后端。因此，`CPAN` 有数以百计的测试模块可供使用——并且，它们可以在同一程序中协同工作。

- * `Test::Exception` 提供了保证你代码正确（不）抛出异常的函数。
- * `Test::MockObject` 和 `Test::MockModule` 允许你通过 模拟（mocking）（模仿但产出不同结果）测试难以测试的接口。
- * `Test::WWW::Mechanize` 允许你测试线上的 Web 应用。
- * `Test::Database` 提供测试对数据库使用及误用情况的函数。
- * `Test::Class` 另行提供组织测试套件的机制。它允许你按特定的测试方法组来创建类。你可以像一般对象继承那样从测试类继承。这是一种降低测试套件重复的好方法。参见由 Curtis Poe 编写的 `Test::Class` 系列，位于 <http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html>。
- * `Devel::Cover` 分析测试套件的执行情况并报告经由实际测试代码的数量。一般说来，覆盖率越高越好——虽然不是总能达到 100% 覆盖率，但 95% 要比 80% 好上不少。

Perl QA 项目 (<http://qa.perl.org/>) 是测试模块的主要源头，也是使 Perl 测试简单高效的智慧和实用经验的来源。

处理警告

Perl 5 会针对各类令人困惑、不甚清楚、带有歧义的情况产生可选的警告。虽然你应该在几乎任何时候无条件地启用警告，但某些情形使我们谨慎地做出禁用警告的决定——并且 Perl 支持这样做。

产生警告

可使用 `warn` 关键字发出警告：

```
warn 'Something went wrong!';
```

`warn` 将一个值列表打印至 `STDERR` 文件句柄 (filehandle)。Perl 会将文件名和 `warn` 调用发生的行号附加其后，除非列表最后一个元素以换行结尾。

核心模块 `Carp` 提供了其他产生警告的机制。其中的 `carp()` 函数把警告以调用方的角度汇报。就是说，你可以按如下方式检查一个函数是几元函数 (arity)：

```
use Carp;

sub only_two_arguments
{
    my ($lop, $rop) = @_;
    Carp::carp( 'Too many arguments provided' ) if @_ > 2;
    ...
}
```

..... 阅读错误消息的每一个人将得知 调用方 代码文件名和行号，而非 `only_two_arguments()`。类似地，`Carp` 中的 `cluck()` 输出到此调用为止的所有函数调用栈跟踪。

为在系统范围内全面跟进怪异的警告或异常，可以对整个程序启用 `Carp` 的详细模式：

```
$ perl -MCarp=verbose my_prog.pl
```

这样便将所有 `croak()` 调用改为 `confess()` 的行为而所有 `carp()` 调用改为 `cluck()` 的行为。

启用和禁用警告

对警告的词法封装如同对变量的词法封装一样重要。旧式代码可能会使用 `-w` 命令行参数对整个文件启用警告，即便其它代码没有明确压制警告产生的意图。它要么有要么没有。如果你有消除警告及潜在警告的资本，它很实用。

现代化的方法是使用 `warnings` 编译命令。在代码中对 `use warnings;` 或其等价 诸如 `use Modern::Perl;` 的使用意味着作者认为对此代码的常规操作不会引发警告。

`-w` 命令行参数单方面对整个程序启用警告，而不顾 `warnings` 编译命令的词法启、禁用。`-x` 参数对整个程序单方面 禁用 警告。没有哪个参数是常用的。

所有 `-w`、`-W` 和 `-X` 影响全局变量 `$^W` 的值。在 `warnings` 编译命令出现（2000 年春季，Perl 5.6.0）之前编写的代码，也许会 `local` 化 `$^W` 以在给定的作用域内压制某些警告。新编写的代码应使用编译命令。

禁用警告类

要在作用域内选择性禁用警告，使用 `no warnings;` 并提供一系列参数。忽略参数列表则在该作用域内禁用所有警告。

`perldoc perllexwarn` 列出了你的 Perl 5 版本在使用 `warnings` 编译命令时能够理解的所有类别的警告。它们中的大多数代表了你的程序可能会陷入的一些非常有意思的条件，并且可由 Perl 认出。另有一小部分在特定条件下不那么有用。举例来说，如果 Perl 检测到一个函数自行调用在一百次以上，则产生 `recursion` 警告。如果你对你编写递归函数的能力有信心，可以在递归的作用域内禁用该警告（另行参见 `tail_calls`。）

如果你正生成代码（`code_generation`）或局部地重定义符号，你或许想禁用 `redefine` 警告。

一些有经验的 Perl 黑客在从众多来源拼接值的字符串处理代码内禁用 `uninitialized` 值的警告。仔细地初始化变量可以免除禁用此警告的需要，但局部风格和简明的代码可使这种警告没有实际意义。

致命的警告

如果你的项目视警告如错误般繁重，你可以使其词法致命。让所有警告提升为异常：

```
use warnings FATAL => 'all';
```

你也许会想使特定类别的警告变得致命，例如对不推荐语法结构的使用：

```
use warnings FATAL => 'deprecated';
```

捕获警告

就像你可以捕获异常，你也可以捕获警告。`%SIG` 变量持有所有类型的信号处理器，这些信号可由 Perl 或你的操作系统抛出。它还包括两个专为 Perl 5 异常和警告准备的信号处理器槽。要捕获警告，将一个匿名函数安装到 `$SIG{__WARN__}`：

```
{
    my $warning;
    local $SIG{__WARN__} = sub { $warning .= shift };

    # 做一些冒险的事
    say "Caught warning:\n$warning" if $warning;
}
```

在警告处理器内，第一个参数是警告消息。不可否认，这个技巧不像在词法范围内禁用警告那么有用——但它在如来自 CPAN 的 `Test::Warnings` 等测试模块内得到良好的利用，这些时候重要的是警告消息的实际文本内容。

`perldoc perlvar` 更为详细地讨论了 `%SIG`。

注册自己的警告

对 `warnings::register` 编译命令的使用可以让你创建自己词法警告，使你代码的用户可以按合适启用、禁用词法警告。很容易就可以做到——在某模块内，用 `use` 引入 `warnings::register` 编译命令：

```
package Scary::Monkey;

B<use warnings::register;>

1;
```

这将创建一个按此包命名的新警告类别（此例中是 `Scary::Monkey`）。用户可以显式地用 `use warnings 'Scary::Monkey'` 启用或用 `no warnings 'Scary::Monkey'` 显式禁用它。要报告一条警告，结合 `warnings::enabled()` 使用 `warnings::warn()` 函数：

```
package Scary::Monkey;

use warnings::register;

B<sub import>
B<{>
    B<warnings::warn( __PACKAGE__ . ' used with empty import list' )>
    B<if @_ == 0 && warnings::enabled();>
B<}>

1;
```

如果 `warnings::enabled()` 为真，那么调用方词法作用域即启用此项警告。你也可以报告已存在类别的警告，诸如对不推荐语法结构的使用：

```
package Scary::Monkey;

use warnings::register;

B<sub import>
B<{>
    B<warnings::warnif( 'deprecated',
        'empty imports from ' . __PACKAGE__ . ' are now deprecated' )
        unless @_;>
B<}>

1;
```

`warnings::warnif()` 函数检查具名警告类别并在其活动时报告。

更多细节参见 `perldoc perllexwarn`。

模块

模块 就是一个包含于自身文件中、可用 `use` 或 `require` 加载的包。一个模块必须是合法的 Perl 5 代码。它必须以一个求值得真的表达式结束，使 Perl 5 语法分析器知道它已成功加载并编译了该模块。

除一些普遍使用的惯例外，没有其他要求。

包通常对应于磁盘上的文件，当你使用 `use` 或 `require` 的裸字形式加载一个模块时，Perl 根据双冒号 (`::`) 分割包名，并将包名的组成部分转换成路径。因此：

```
use StrangeMonkey;
```

.....使得 Perl 在 `@INC` 的每一个目录中依次搜索名为 `StrangeMonkey.pm` 的文件，直到找到该文件或完成对列表的遍历。同样地：

```
use StrangeMonkey::Persistence;
```

.....使得 Perl 在 `@INC` 中所有目录下存在的 `StrangeMonkey/` 子目录中查找名为 `Persistence.pm` 的文件，如此等等。最后：

```
use StrangeMonkey::UI::Mobile;
```

.....使得 Perl 在相对每个 `@INC` 中目录的 `StrangeMonkey/UI/Mobile.pm` 路径处寻找文件。换句话说，如果你想加载你的 `StrangeMonkey::Test::Stress` 模块，你必须拥有一个名为 `StrangeMonkey/Test/Stress.pm`，且它可以顺着 `@INC` 所列出的目录找到。

`perldoc -l Module::Name` 会打印出相关 `.pm` 文件的完整路径，并提供存在于 `.pm` 文件中该模块的文档。

技术上 不要求此位置下的文件必须包含 `package` 声明，更不用说匹配文件名的 `package` 声明了。然而，出于维护上的考虑，高度推荐此惯例。

使用（“`use`”）和导入（“`import`”）

当你用 `use` 关键字加载模块时，Perl 从磁盘上加载它，接着调用它的 `import()` 方法，将你提供的参数传递进去。这发生在编译期：

```
use strict;                # 调用 strict->import()
use CGI ':standard';       # 调用 CGI->import( ':standard' )
use feature qw( say switch ) # 调用 feature->import( qw( say switch )
)
```

你不必要提供一个 `import()` 方法，你也可以将其用于任何目的，但标准 API 期望它接受一个由符号组成的参数列表（通常是函数）使其在调用方名称空间内可用。这不是一个强制的要求，诸如 `strict` 等编译命令（`pragmas`）改变了它们的行为而非 导入符号。

`no` 关键字调用一个模块的 `unimport()` 方法，如果它存在，则传入参数。虽然可能移除已经导入的符号，但通常它用于禁用特定编译命令特性以及其他通过 `import()` 引入新特性的模块：

```
use strict;

# 禁用符号引用，要求变量声明，不允许裸字
...

{
    no strict 'refs';

    # 允许符号引用
    # 仍要求变量声明，禁止裸字
}
```


就像 `use` 和 `import()`, `no` 在编译期调用 `unimport()`。在效果上:

```
use Module::Name qw( list of arguments );
```

.....和下面的代码效果一样:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->import( qw( list of arguments ) );
}
```

类似的:

```
no Module::Name qw( list of arguments );
```

.....和下面的代码等效:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->unimport( qw( list of arguments ) );
}
```

.....包括对模块的 `require`。

你可以直接调用 `import()` 和 `unimport()`, 虽然在 `BEGIN` 块之外反导入 (“`unimport`”) 一个编译命令有些说不通, 通常它们对编译期另有影响。

如果 `import()` 或 `unimport()` 不存在于模块中, Perl 不会给出错误消息。它们事实上是可选的。

Perl 5 的 `use` 和 `require` 是大小写敏感的, 然而底层的文件系统不是。虽然 Perl 知道 `strict` 和 `Strict` 之间的区别, 你使用的操作系统和文件系统也许并不知道。如果你写的是 `use Strict;`, 一个大小写敏感的文件系统不会去查找 `strict.pm`。一个大写不敏感文件系统则将找到 `Strict.pm`。然而, 当 Perl 尝试在已加载的模块上调用 `Strict->import()` 时, 不会产生任何效果, 因为包名是 `strict`。

可移植的程序在虽然不必的情况下也会严格对待大小写。

导出 (“`export`”)

模块可以通过一个名为 `导出` 的过程使全局符号在其它包中可用。这是通过 `use` 语句 向 `import()` 传递参数的反面。

向其它模块导出函数或变量的标准方式是通过核心模块 `Exporter`。 `Exporter` 依赖于包全局变量——特别是 `@EXPORT_OK` 和 `@EXPORT`——它们包含了一个在请求时导出的符号 列表。

考虑一个提供若干全系统可用的独立函数的 `StrangeMonkey::Utilities` 模块:

```
package StrangeMonkey::Utilities;

use Exporter 'import';

our @EXPORT_OK = qw( round_number translate screech );
```

```
...
```

```
1;
```

任何人都可以使用这个模块，并且，可选地，导入任一或全部三个导出函数。你也可以导出变量：

```
push @EXPORT_OK, qw( $spider $saki $squirrel );
```

CPAN 模块 `Sub::Exporter`

为不使用包全局变量导出函数提供了一个更好的接口。它同时提供了更多强大的选项。然而，`Exporter` 可以导出变量，而 `Sub::Exporter` 只可以导出函数。

你可以通过将符号列在 `@EXPORT` 而非 `@EXPORT_OK` 中来默认地导出它们：

```
our @EXPORT = qw( monkey_dance monkey_sleep );
```

.....因此，任何 `use StrangeMonkey::Utilities;` 语句将导入两个函数。注意指定要导入的符号并不导入默认的符号。同时，你可以通过显式地提供一个空列表来加载一个模块而不导入任何符号：

```
# 是模块可用，但不用 import() 导入符号
use StrangeMonkey::Utilities ();
```

不理睬任何导入列表，你总是可以通过完全限定名称来调用其它包中的函数：

```
StrangeMonkey::Utilities::screech();
```

使用模块来组织代码

Perl 5 并不要求你使用模块，也不要求你使用包或是名称空间。你可以将所有代码放在单个 `.pl` 文件中，或多个 `.pl` 文件，随后你可以按需通过 `do` 或 `require` 加载。你拥有灵活性来按合适的方式管理代码，给出开发风格，控制项目的条框、风险和回报、增加经验，以及 Perl 5 部署的舒适程度。

还有一条经验之谈来自有经验的 Perl 5 程序员，就是一个上百行代码的项目可从创建模块中获得多重益处。

- * 模块有助于强制对系统中不同实体进行逻辑上的隔离；
- * 模块提供 API 边界，无论是过程式还是面向对象；
- * 模块使源代码自然组织；
- * Perl 5 生态系统有许多工具专门创建、维护、组织、部署模块和发行版；
- * 模块提供了一种代码重用机制。

即便你不采用面向对象的手法，为系统中不同实体或职责建立模块保持相关代码内聚、不相关代码隔离。

发行模块

发行模块（“distribution”）是一个或多个模块（modules）的集合，由此组成单个可重分发、可测试、可安装的单元。效果上即是模块和元数据的集合。

管理软件配置、构建、分发、测试和安装——甚至是在你的工作组织中——最为简便的方法就是创建和 CPAN 兼容的模块。CPAN 的惯例——如何打包发行、如何解决其依赖、将其安装至何处、如何验证它是否正常、如何显示文档、如何管理代码仓库——这些问题已经由维护万个项目的几千贡献者所共同提出。

特别是，由 CPAN 实现的测试、报告、依赖检查功能非常齐全，已大大超出同类语言社区所能提供的信息范围，品质也在其余之上。一个按 CPAN 标准构建的发行模块可以上传后几小时内若干 Perl 5 版本及不同硬件平台上被测试——所有这些无需人工干预。

你可以选择永远不像公共 CPAN 发行模块那样发布你的代码，但你可以在可能时重用现存的 CPAN 工具和设计。智能的默认设置和可定制性的组合多少都可以满足你特定的需求。

发行模块的属性

一个发行模块显然会包括一个或多个模块。它同时也包含其他的文件和目录：

- * `Build.PL` 或是 `Makefile.PL`，这些程序用于配置、构建、测试、捆绑及安装模块；
- * `MANIFEST`，发行模块中包含的所有文件的列表。这有助于打包工具生成完整的 `tar` 压缩包并帮助验证压缩包使用者得到所有必需的文件；
- * `META.yml` 和/或 `META.json`，一个包含有关发行模块及其依赖的元数据文件；
- * `README`，对发行模块的描述、意图、版权和许可信息；
- * `lib/`，含有 Perl 模块的目录；
- * `t/`，包含测试文件的目录；
- * `Changes`，模块变更的日志。

额外地，一个良好组织的发行模块必须包含唯一的名称和单个版本号（通常从其主要模块而来）。从公共 CPAN 上下载的组织良好的发行模块应遵循这些标准。（CPANTS 服务将评估所有 CPAN 发行模块的“`kwalitee`”质量难以启发式地衡量。`Kwalitee` 是“质量”的自动化亲属。并提出改进之处使它们更加易于安装和管理。）

CPAN 发行模块管理工具

Perl 5 核心包含若干工具来管理发行模块——不仅仅是从 CPAN 安装它们，还有开发和管理属于自己的模块：

- * `CPAN.pm` 是官方的 CPAN 客户端。虽然它默认从公共 CPAN 安装模块，但你可以将其指向自己的 CPAN 仓库以代替或作为公共仓库的补充；
- * `CPANPLUS` 是有着不同设计手法的 CPAN 客户端替代品。很多方面它完成得比 `CPAN.pm` 更为出色，但是它们的大部分功能相同。可以按你的使用偏好进行选择。
- * `Module::Build` 是一个由纯 Perl 编写的工具套件，它可以配置、构建、安装和测试发行模块。它和早些提到的 `Build.PL` 文件配合工作。
- * `ExtUtils::MakeMaker` 是一个更老的遗留工具，它是 `Module::Build` 意图替代的目标。虽然它已经进入维护阶段、仅接纳最紧急的缺陷修复，但它仍在大规模地使用中。它和早些提到的 `Makefile.PL` 文件配套。
- * `Test::More` (`testing`) 是基本也是被广泛使用的测试模块，用于为 Perl 软件编写自动化测试；
- * `Test::Harness` 和 `prove` (`running_tests`) 是用于运行测试和解析、报告测试结果的工具。

作为附加，若干非核心 CPAN 模块是你的开发生涯更为轻松：

- * `App::cpanminus` 是一个新兴的实用工具，它对公共 CPAN 提供了几乎免配置的使用方式。它满足了你查找安装模块需求的 90%。
- * `CPAN::Mini` 和 `cpanmini` 命令运行你创建自己（私人的）公共 CPAN 镜像。你可以在此仓库中插入你自己的发行模块，并管理在你的组织内可用的公共模块版本。
- * `Dist::Zilla` 是通过自动化常规任务来管理发行模块工具集。很多情况下它可以替代对 `Module::Build` 或 `ExtUtils::MakeMaker` 的使用。
- * `Test::Reporter` 允许你报告对你安装的发行模块运行自动化测试套件的结果，向模块作者提供更详细的失败数据。

设计发行模块

设计发行模块的过程可以填满一整本书（参见 Sam Tregar 的著作 *Writing Perl Modules for CPAN*），但是有不少设计原理可以帮助你。来自 CPAN 的诸如 `Module::Starter` 或 `Dist::Zilla` 等实用工具开始。最初学习配置和规则的代价也许看上去像一笔不合理的投资，但是按部就班地配置所有事项（就 `Dist::Zilla` 一例来说，永不 过时）将你从冗长的记录中释放出来。

接着考虑若干规则。

- * 每一个发行模块应拥有单一、经良好定义的目的。该目的也许是处理某一类型的数据文件或是将若干相关发行模块集成为单一的可安装捆绑。将你的软件解耦为单个部分允许你正确地管理它们各自依赖并使其更好地封装。
- * 每一个发行模块需要单一的版本号。版本号必须递增。使用语义版本规则（“semantic version policy” (<http://semver.org/>) 是明智的，而且它兼容 Perl 5 的版本惯例。
- * 每一个发行模块必须包含经良好定义的 API。一个综合自动化测试套件可以验证你是否在版本间保持 API 一致。如果你使用本地 CPAN 镜像来安装你自己的发行模块，你可以重用 CPAN 基础设施来测试模块及其依赖。对可重用的组件进行持续测试是很方便的。
- * 自动化你的发行模块测试并使它们可重复进行且有价值。有效管理软件要求你明白其工作原理，和在它出错时知道为什么。
- * 提供一个有效的简洁的接口。避免使用全局符号和默认导出；允许人们按需使用并不对他们的名称空间进行污染。

来自 CPAN 的 `CPAN::Mini` 发行模块允许你创建你自己的本地 CPAN 镜像，你可以向其中插入你自己的发行模块。

UNIVERSAL 包

Perl 5 提高了一个特殊的包，就面向对象来说，它是所有包的先祖。UNIVERSAL 包为其它类和对象提供了若干可用的方法。

isa() 方法

`isa()` 方法接受包含类名或内置类型名称的字符串。你可以将其作为类方法调用或用作对象上的实例方法。如果类或对象从给出的类中衍生而来，或者对象本身是给定类型经 `bless` 的引用，则此方法返回真。

给出对象 `$pepper`，一个 `bless` 为 `Monkey` 类（继承 `Mammal` 类）的哈希引用：

```
say $pepper->isa( 'Monkey' ); # 打印 1
say $pepper->isa( 'Mammal' ); # 打印 1
say $pepper->isa( 'HASH' ); # 打印 1
say Monkey->isa( 'Mammal' ); # 打印 1
```

```
say $pepper->isa( 'Dolphin' ); # 打印 0
say $pepper->isa( 'ARRAY' ); # 打印 0
say Monkey->isa( 'HASH' ); # 打印 0
```

内置类型为 `SCALAR`、`ARRAY`、`HASH`、`Regexp`、`IO` 和 `CODE`。

你可以在你自己的类中覆盖 `isa()`。这在处理模拟对象（Mock Object）（示例参见 CPAN 上的 `Test::MockObject` 和 `Test::MockModule`）或不使用角色（roles）的代码时非常有用。

can() 方法

`can()` 方法接受包含方法名称的字符串（参见 `method_sub_equivalence` 中的免责声明）。它如果它存在，则返回指向实现该方法的函数引用。否则，返回假。你可以在类、对象或包名称上调用它。在后一种情况下，它返回函数引用，而非方法。

给出一个名为 `SpiderMonkey` 并带有名为 `screech` 方法的类，你可以这样得到方法的引用：

```
if (my $meth = SpiderMonkey->can( 'screech' )) { ... }

if (my $meth = $sm->can( 'screech' )
{
    $sm->$meth();
}
```

给出一个插件式结构，你可以用类似方法测试出一个包是否实现了特定的函数：

```
# 一个有用的 CPAN 模块
use UNIVERSAL::require;

die $@ unless $module->require();

if (my $register = $module->can( 'register' )
{
    $register->();
}
```

如果你使用了 `AUTOLOAD()`，可以（并且应该）在你自己的代码中覆盖 `can()`。篇幅更长的说明请参见 `autoload_drawbacks`。

已知在一种的情况下将 `UNIVERSAL::can()` 作为函数而非方法调用是错误的：决定某个类是否存在于 Perl 5 中。如果 `UNIVERSAL::can($classname, 'can')` 返回真，说明某人于某处定义了一个名为 `$classname` 的类。除此之外，`Moose` 的内省更强大也更易于使用。

VERSION() 方法

`VERSION()` 方法对所有包、类和对象都是可用的。它返回合适的包或类中 `$VERSION` 变量值。它接受一个版本号作为可选参数。如果你提供了版本号，此方法将会在目标 `$VERSION` 大于等于所提供参数时抛出异常。

给出 1.23 版的 `HowlerMonkey` 模块：

```
say HowlerMonkey->VERSION();      # 打印 1.23
say $hm->VERSION();                # 打印 1.23
say $hm->VERSION( 0.0 );           # 打印 1.23
say $hm->VERSION( 1.23 );          # 打印 1.23
say $hm->VERSION( 2.0 );           # 抛出异常
```

你可以在代码中覆盖 `VERSION()`，但这样做并没有什么很好的理由。

DOES() 方法

`DOES()` 是 Perl 5.10.0 新加的。它的存在支持了程序中对角色（roles）的使用。向其传递调用物和角色名称，此方法会在合适的类饰演此角色时返回真。（类也可以通过继承、委托、合成、角色应用或其他机制饰演此角色。）

DOES() 的默认实现仍回到 isa() 上, 因为继承是类饰演某角色的一种机制。给出 一个名为 Cappuchin 的类:

```
say Cappuchin->DOES( 'Monkey'      ); # 打印 1
say $cappy->DOES(      'Monkey'      ); # 打印 1
say Cappuchin->DOES( 'Invertebrate' ); # 打印 0
```

如果你手动提供了角色或其他同质异晶行为, 你可以 (也应该) 在自己的代码中覆盖 DOES()。除此之外, 可以使用 Moose 并无需关心细节。

扩展 UNIVERSAL

在 UNIVERSAL 存储另外方法以使其在 Perl 5 中的其余类和对象中可用是一种诱惑。但请拒绝这种诱惑; 这个全局行为可因其自身不受约束而引发隐晦的副作用。

话虽如此, 出于 调试 目的或修复不正确的默认行为而偶尔滥用 UNIVERSAL 尚可以宽恕。例如, Joshua ben Jore 的 UNIVERSAL::ref 发行模块使几乎无用的 ref() 操作符可用。UNIVERSAL::can 和 UNIVERSAL::isa 发行模块能够帮助你找到并除去阻止使用多态的陈旧惯用语 (但请参见 Perl::Critic 以了解其他带有其它优势的另一些手段)。

UNIVERSAL::require

模块增加了有助于在运行时加载模块和类的有用行为——虽然使用 如 Module::Pluggable 的模块更安全也更不带入侵性。

在小心控制之下的代码和非常特殊非常现实的情况之外, 确实没有什么直接向 UNIVERSAL 中添加代码的理由。几乎总是有其他更好的替代设计可以选择。

代码生成

程序员的进步需要你去找寻更好的抽象。越少代码要写越好。解决方案越通用越好。当你可以删代码加功能的时候, 你已经达成了某种完美的目标。

新手程序员常会写出多于要求的代码, 其原因部分基于对语言、库、惯用语的不熟悉, 同时也归咎于无法熟练地创建和维护良好的抽象。他们以编写长篇的过程式代码起步, 接着发现函数, 再是参数, 然后是对象, 还有——可能的话——高阶函数和闭包。

元编程 (或 代码生成) ——编写编写程序的程序——是另一种抽象技巧。它可以如发掘高阶函数能力般清晰, 也可能如鼠洞一般让你身陷其中, 困惑而恐惧。然而, 这种技巧强大、实用——其中一些还是 Moose (moose) 这类强大工具的基础。

处理缺少函数和方法的 AUTOLOAD (autoload) 技巧展示了此技巧勉强的一面; Perl 5 的函数和方法分派系统允许你定制常规查找失败后的行为。

eval

生成代码最简单的 至少是 概念上.... 技巧莫过于创建一个包含合法 Perl 代码片段的字符串并通过 eval 字符串操作符编译。不像捕获异常的 eval 代码块操作符, eval 字符串在当前作用域内编译其中内容, 包括当前包和词法绑定。

此技巧的常用于提供后备, 如果你不能 (或不想) 加载某个可选的依赖:

```
eval { require Monkey::Tracer }
or eval 'sub Monkey::Tracer::log {}';
```

如果 Monkey::Tracer 不可用, 其中的 log() 函数仍将存在, 只是不做任何事。

这不一定是处理这种特性的 最佳 途径, 空对象 (Null Object) 模式通常提供更好的封装, 但这是完成任务的 一种 方法。

这个简单的例子可能有点靠不住。为在 `eval` 代码中包含变量，你必须处理引号问题。这增加了内插的复杂度：

```
sub generate_accessors
{
    my ($methname, $attrname) = @_;

    eval <<"END_ACCESSOR";
    sub get_$methname
    {
        my \ $self = shift;

        return \ $self->{$attrname};
    }

    sub set_$methname
    {
        my (\ $self, \ $value) = \@_;

        \ $self->{$attrname} = \ $value;
    }
    END_ACCESSOR
}
```

对忘记加反斜杠的你表示悲哀！祝你调教语法高亮器好运！更糟糕的是，每次对 `eval` 字符串的调用都将创建一个代表整段代码的全新数据结构。编译代码也不是免费的——也许，比IO操作便宜些，但并非免费。

即便如此，此技巧简单合理、易于理解。

参数闭包

虽然使用 `eval` 构建访问器和增变器时很直接，但闭包 (closures) 允许你在编译期向已生成的代码添加参数而无需进行额外的求值：

```
sub generate_accessors
{
    my $attrname = shift;

    my $getter = sub
    {
        my $self = shift;
        return $self->{$attrname};
    };

    my $setter = sub
    {
        my ($self, $value) = \@_;

        $self->{$attrname} = $value;
    };

    return $getter, $setter;
}
```

这段代码避免了不愉快的引号问题。由于只有一道编译过程，性能也更好，无论有多少要创建的访问器。通过重用 相同的 已编译代码作为两个函数的主体，它甚至使用更少的内存。所有的区别来自对词法变量 `$attrname` 的绑定。对于长期运行的进程或是包含大量访问器的程序中，此技巧非常有用。

向符号表安装比较容易，但很丑陋：

```
{
    my ($getter, $setter) = generate_accessors( 'homecourt' );

    no strict 'refs';
    *{ 'get_homecourt' } = $getter;
    *{ 'set_homecourt' } = $setter;
}
```

这一古怪的、哈希那样的语法指向当前 符号表 中的一个符号，它是当前名称空间内存放诸如包全局变量、函数、方法等全局可见符号的地方。将引用赋值给符号表某项将安装或替换对应的条目。要将一个匿名函数提升为方法，可把函数引用赋值到符号表中的对应条目。

这个操作是一个符号引用，因此应该禁用 `strict` 对此操作的引用检查。许多程序在类似的代码中有不少隐晦的缺陷，它们在单个步骤内进行赋值和生成：

```
{
    no strict 'refs';

    *{ $methname } = sub {
        # 隐晦的缺陷：strict refs
        # 在此处也被禁用
    };
}
```

这个例子在外部块、内部块和函数体中都禁用严格检查。只有赋值违反了严格的引用检查，因此只要对该操作禁用即可。

如果你编写的代码中，方法名称是一个字符串面值，而非变量的内容，你可以不用通过符号引用而直接向相关符号赋值：

```
{
    no warnings 'once';
    (*get_homecourt, *set_homecourt) = generate_accessors(
'homecourt' );
}
```

这没有违反严格检查，但是会引发一条 “used only once” 警告，除非你已经在作用域内部显式地抑制它的产生。

编译期操控

不同于显式编写的代码，通过 `eval` 字符串生成的代码于运行时生成。虽然你预计一个常规函数在你程序的生命周期内都是可用的，但（运行时）生成的函数也许直到你要求时才是可用的。

在编译期强制 Perl 运行代码———生成其他代码———的方法是将其包装于 `BEGIN` 块内。当 Perl 5 语法分析器遇到标有 `BEGIN` 的代码块时，它将对整个代码块进行语法分析。证实其不含任何语法错误后，代码块将立即执行。执行完毕，语法分析过程就好像未曾中断一般继续。

实际点说，编写：

```
sub get_age    { ... }
sub set_age    { ... }

sub get_name   { ... }
sub set_name   { ... }

sub get_weight { ... }
sub set_weight { ... }
```

.....和：

```
sub make_accessors { ... }

BEGIN
{
    for my $accessor (qw( age name weight ))
    {
        my ($get, $set) = make_accessors( $accessor );

        no strict 'refs';
        *{ 'get_' . $accessor } = $get;
        *{ 'set_' . $accessor } = $set;
    }
}
```

.....之间的区别主要是可维护性。

由于 Perl 隐式地将 `require` 和 `import` (importing) 用 `BEGIN` 包装起来，在模块内，任何函数外部的代码都会在你 `use` 它时执行。任何处于函数外、模块内的代码会在 `import()` 调用发生之前执行。如果你 `require` 该模块，则不含隐式的 `BEGIN` 代码块。函数外部代码的执行将放在语法分析的结尾。

同时也请注意词法声明（名称和作用域间的联系）和词法赋值之间的交互。前者发生于编译期，而后者发生于执行点处。如下代码隐含一处缺陷：

```
use UNIVERSAL::require;

# 有缺陷；不要使用
my $wanted_package = 'Monkey::Jetpack';

BEGIN
{
    $wanted_package->require();
    $wanted_package->import();
}
```

.....因为 `BEGIN` 块在对 `$wanted_package` 的字符串值赋值前执行。结果将是意图在未定义值上调用 `require()` 方法而引发的异常。

`Class::MOP`

不像安装函数引用来填充名称空间及创建方法，目前没有简易的内置途径在 Perl 5 中创建类。所幸的是，一个成熟且强大的 CPAN 发行模块恰好可以完成此项工作。Class::MOP 是 Moose (moose) 的支柱库。它提供了元对象协议 (Meta Object Protocol) —— 一种用于对象系统创建操控自身的机制。

相比自行编写脆弱的 eval 字符串代码或是尝试手动干涉符号表，你可以通过对象和方法操控程序中的实体和抽象。

要创建一个类：

```
use Class::MOP;

my $class = Class::MOP::Class->create( 'Monkey::Wrench' );
```

在你创建它时，你可以添加属性和方法到该类中：

```
use Class::MOP;

my $class = Class::MOP::Class->create(
    'Monkey::Wrench' =>
    (
        attributes =>
        [
            Class::MOP::Attribute->new( '$material' ),
            Class::MOP::Attribute->new( '$color' ),
        ]
        methods =>
        {
            tighten => sub { ... },
            loosen  => sub { ... },
        }
    ),
);
```

.....或在创建后，把它们添加到元类 (Metaclass) (代表类的对象) 中：

```
$class->add_attribute( experience => Class::MOP::Attribute->new(
'$xp' ) );
$class->add_method(    bash_zombie => sub { ... } );
```

.....你可以对元类进行内省：

```
my @attrs = $class->get_all_attributes();
my @meths = $class->get_all_methods();
```

使用 Class::MOP::Attribute 和 Class::MOP::Method，你可以类似地创建、操作并内省属性和方法。此元对象协议及其带来的灵活性是 Moose (moose) 强大的根源。

重载

Perl 5 不是一个彻头彻尾面向对象的语言。其核心数据类型 (标量、数组和哈希) 也非有方法让你重载的对象。即便如此，你还是能够控制你编写的类和对象的行为，特别是当它们在各类上下文中强制类型转换或求值。这称为重载 (Overloading)。

重载隐晦而强大。一个有趣的例子就是重载对象在布尔上下文中的行为，特别是在你使用如空对象 (Null Object) 模式 (<http://www.c2.com/cgi/wiki?NullObject>) 时。在布尔上

下文中，对象为真.....但仅在你不对布尔化操作进行重载的情况下。
你可以重载几乎所有的对象操作：字符串化、数值化、布尔化、迭代、调用、数组访问、哈希访问、算术操作、比较操作、智能匹配、按位操作甚至赋值。

重载常见操作

最为有用的通常也是最为常见的：字符串化、数值化以及布尔化。`overload` 编译命令允许你将函数和可重载操作关联起来。下面就是一个重载布尔求值的类：

```
package Null;

use overload 'bool' => sub { 0 };
```

在所有布尔上下文中，此类所有实例求值得假。

`overload` 编译命令的参数是一个键值对，键描述了重载的类型而值则是替代 Perl 默认行为的函数引用。

添加字符串化也是很容易的：

```
package Null;

use overload
    'bool' => sub { 0 },
    B<< '""' => sub { '(null)' }; >>
```

重载数值化操作则更为复杂，因为算术操作符倾向于执行二元操作（arity）。给出两个重载了加法的操作数，如何确定优先级？答案应是一致的、易于解释、便于未阅读实现源码的人理解的。

`perldoc overload` 意图在标有 `Calling Conventions for Binary Operations` 和 `MAGIC AUTOGENERATION` 的两个小节中解释这一切，但最简单的解决方法是重载数值化操作并告诉 `overload` 在可能时将提供的重载操作用作后备：

```
package Null;

use overload
    'bool' => sub { 0 },
    '""' => sub { '(null)' },
    B<< '0+' => sub { 0 }, >>
    B<< fallback => 1; >>
```

将 `fallback` 设为真值使 Perl 在可能的情况下使用其他已定义的重载操作来合成所要求的操作。如果不行，Perl 将表现得好像未经任何重载那样。这通常是你想要的。

没有 `fallback`，Perl 将仅使用由你提供的特定重载操作。如果某人尝试进行未经你重载的操作，Perl 将会抛出异常。

重载和继承

子类继承祖先重载的操作。它们可以以两种方法中的一种覆盖这个行为。如果父类如所示使用重载操作，函数引用直接提供，则子类 必须 直接通过 `overload` 覆盖父类的 重载行为。

父类可通过指定执行重载操作所调用的方法 名称 来允许其子代更为灵活，而非将函数 硬性编码：

```
package Null;

use overload
```

```
'bool'    => 'get_bool',
''        => 'get_string',
'0+'      => 'get_num',
fallback => 1;
```

子类在不直接用 `overload` 时只能覆盖指定的方法。

对方法名的使用可以产生更加灵活的代码，但是开发人员对代码引用的使用更加频繁。在这种情况下，请使用开发小组定下的编码规范。

重载的用途

重载也许看上去像是一种富有诱惑力的工具，用于产生新操作的符号快捷方式。IO::All CPAN 发行模块极尽此特性之能事，为简明、可复合的代码提供聪明的点子。还有各类通过合理利用重载而精炼的出色 API，使众多混乱凝固下来。有时最好的代码会出于简单直接的设计而避开灵巧。

因为加法、乘法还有拼接操作记法已经到处都是，针对 `Matrix`（矩阵）类重载这些操作还可以说得通。一个全新的问题领域则无需刻意重载这些操作——你没必要因此将现存 Perl 操作符弄得一词二义。

Damian Conway 的著作 `Perl 最佳实践`（Perl Best Practices）建议另行将重载用于防止对对象的意外滥用。举例来说，将无法用单一数值表达的对象上的数值化操作重载为 `croak()` 可以帮助你实际程序中找到真正的缺陷。Perl 5 中重载还是比较少见的，但是这条建议可以提升程序的可靠性和安全性。

Taint

Perl 给予你编写安全代码的工具。这些工具并非认真思考和计划的替代品，但它们对谨慎和深入理解做出回报，帮助你避免隐晦的失误。

使用 Taint 模式

一个名为 `taint`（“污点”）模式或 `taint` 的特性向所有来自程序之外的数据添加一小部分元数据。任何衍生自污点数据的数据同样带有污点，你可以在你的程序内使用污点数据，但如果你用其影响外部世界——如果你不安全地使用它——Perl 将抛出一条致命异常。

`perldoc perlsec` 中对污点模式进行翔实地解释，并带有其他的安全指导。

要启用污点模式，可以用 `-T` 参数启动你的程序。你可以在将文件设置为可执行且不使用 `perl` 启动的情况下于 `#!` 行中使用此参数；如果你像 `perl mytaintedappl.pl` 这样运行它并忽略 `-T` 参数，Perl 会以异常退出。当 Perl 遇到位于 `#!` 行的参数时，它已经错过了污染 `%ENV` 中环境数据的时机，此即一例。

Sources of Taint

污点数据来自两个位置：文件输入和程序的操作环境。前者是你从文件中读取或从网页或网络编程收集的用户数据。后者则更为隐晦。它包括命令行参数、环境变量以及来自系统调用的数据。即便是如读取目录句柄（用 `opendir()` 打开）的操作产出污点数据。

来自核心模块 `Scalar::Util` 的 `tainted()` 函数在其参数带污点时返回真：

```
die "Oh no!" if Scalar::Util::tainted( $some_suspicious_value );
```

从数据中除去污点

要除去污点，你必须用正则表达式捕获操作从数据中提取已知良好的部分。捕获的数据将是不带污点的。如果你的用户输入由美国电话号码组成，你可以向这样去污：

```
die "Number still tainted!"
unless $tainted_number =~ /(\\d{3}\\) \\d{3}-\\d{4})/;

my $safe_number = $1;
```

提供的模式对你所需部分描述得越具体，你的程序就越安全。相反，拒绝 特定条目或形式的手段则有可能过度看重有害数据。就安全来说，Perl 更希望你禁止不需要的安全数据而非允许有害但看上去安全的数据。即便如此，没有什么阻止你编写捕获变量所有内容的模式——既然如此，为什么还要使用污点模式呢？

从环境中除去污点

一个污点数据的来源就是超级全局变量 `%ENV`，它代表了系统中的环境变量。这部分数据带有污点，因为来自程序以外的力量可以操控这些值。任何修改 Perl 或 Shell 查找文件和目录的环境变量都是攻击的目标。一个污点敏感的程序应该从 `%ENV` 中删除若干键并将 `$ENV{PATH}` 设置为具体且相当安全的路径：

```
delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
$ENV{PATH} = '/path/to/app/binaries/';
```

如果你不恰当地设置 `$ENV{PATH}`，你将收到有关其不安全性的消息。

如果这一环境变量包含当前工作目录，或者它包含相对路径，再或者指定的目录有着全局可写的属性，一个聪明的攻击者可以通过劫持系统调用执行不安全的操作。

基于相似的理由，污点模式下的 `@INC` 并不包含当前工作目录。Perl 也将忽略 `PERL5LIB` 以及 `PERLLIB` 环境变量。如果你需要添加库目录，可以使用 `lib` 编译命令或是 perl 的 `-I` 参数。

Taint 陷阱

污点模式要么起作用要么不起。它只有开和关两种状态。有时候这会导致程序员使用宽松的模式除去数据污点，并留下安全的错觉。请仔细审查去污代码。

不幸的是，并非所有模块都能正确处理污点数据。这是 CPAN 作者应该严肃对待的缺陷。如果你必须使遗留代码污点安全（`taint-safe`），请考虑使用 `-t` 参数，它启用污点模式但把违反条件从异常改为警告。这不是完全污点模式的替代，但允许你在不使用“`-T` 独大”的情况下让现有程序更加安全。