Perl 程序员应该知道的事

Andy Lester (著)　　Xiaodong Xu (译)

2013 年 11 月 1 日

1　前言

1.1　本书简介

*Perl 程序员应该知道的事*是一本以食谱形式来介绍 Perl 语言编程的书籍。通过本书，你不仅可以快速掌握 Perl 语言的基础知识，而且能够将所掌握的知识立即用到真实世界的 Perl 编码中。

我们认为，使用 Perl 编程是有趣的。因此，学习 Perl 编程也应当有趣才是。本书没有长篇大论式的枯燥说教，而是直击 Perl 程序员必需的每个知识点。既能让你在较短的时间内理解 Perl 语言，又能用于实际的 Perl 编程。

1.2　关于作者

Andy Lester，知名 Perl 社区贡献者，文件搜索工具 Ack 的作者，以及许多 CPAN 模块的维护者。

1.3　关于译者

Xiaodong Xu，GNU/Linux 及 Perl 爱好者，LinuxTOY 网站创始人，喜欢编程和爱好新奇事物。

1.4　可用版本

目前本书主要提供 HTML 格式版本。

1.5　关于

本书基本上是对 Andy Lester 所撰写的 Perl 101 的翻译。不过，我会根据自己的理解和认识对其进行扩充。同时，为了反映 Perl 当下的变化，我也会进行适当的改写。总之，希望这些内容能够对 Perl 新手提供帮助。

1.6 源代码

本书源代码位于 GitHub 上的 perl-things 仓库。无论是建议，还是批评，亦或贡献内容，一律欢迎。

1.7 致谢

感谢原著者 Andy Lester，没有他，今天的内容将无从谈起。同时，也要感谢你的关注，正是这种关注才让我有动力一直坚持在路上。

此外，感谢以下网友对此书的贡献：

- 依云：翻译修正
- yukirock：翻译修正

目录

1 前言 **1**

1.1 本书简介 1

1.2 关于作者 1

1.3 关于译者 1

1.4 可用版本 1

1.5 关于 1

1.6 源代码 2

1.7 致谢 2

2 如何获得 Perl **11**

2.1 你有 Perl 吗 12

2.2 GNU/Linux 12

2.3 Mac OS X 12

2.4 Windows 12

2.5 Perl 源代码 13
2.6 Perlbrew 和 Plenv 13
2.6.1 Perlbrew 13
2.6.2 Plenv 14

3 术语 **15**

3.1 Perl、perl 及 PERL 15
3.2 Perl 不是缩写词 15
3.3 There Is More Than One Way To Do It 15
3.4 Larry 15
3.5 Randal 15
3.6 Damian 15
3.7 懒惰、急躁和傲慢 16
3.8 DWIM 16
3.9 DTRT 16
3.10 脚本与程序 16

4 文档 **16**

4.1 perldoc 16
4.2 利用 `perldoc -f` 查阅 Perl 函数 17
4.3 利用 `perldoc -v` 查阅 Perl 变量 17
4.4 利用 `perldoc -q` 搜索 Perl FAQ 17
4.5 利用 `perldoc modulename` 查阅 Perl 模块文档 18
4.6 利用 `cpandoc` 查阅未安装 Perl 模块的文档 18
4.7 在线文档 18
4.8 编写自己的文档 19

5 字符串 **19**

5.1 利用内插将字符串嵌入到另一个字符串中 19

5.2 非内插字符串 19

5.3 在字符串中使用 Email 地址要小心 19

5.4 使用 `length()` 获得字符串的长度 20

5.5 使用 `substr()` 提取字符串 20

5.6 关于字符串 vs. 数字不必担心太多 20

5.7 利用 `++` 操作符自增非数字字符串 20

5.8 利用 `heredocs` 创建长字符串 21

6 数字 **21**

6.1 小心地测试浮点数的相等性 21

6.2 数字舍入 22

7 数组 **22**

7.1 简单创建单词数组 22

7.2 利用数字值访问数组 23

7.3 数组的长度是标量值 23

7.4 数组没有边界 24

7.5 数组会平展，但不会嵌套 24

7.6 列表能带扩展的逗号（,） 24

7.7 像队列和堆栈一样使用数组 25

7.8 利用数组分片提取数组的部分元素 25

7.9 利用数组分片来分配数组块 26

7.10 利用 `splice` 就地修改数组 26

7.11 利用 `map` 处理数组 27

7.12 利用 `grep` 从数组选择项目 27

8 哈希 **28**

8.1 哈希是键/值对 28
8.2 通过键/值对列表来创建哈希 29
8.3 利用花括号访问独立的哈希条目 29
8.4 获取哈希的键/值 30
8.5 哈希键自引用单词 30
8.6 哈希只能包含标量 30
8.7 哈希是无序的 30
8.8 你无法排序哈希 31
8.9 使用列表赋值合并哈希 31
8.10 何时使用哈希，何时使用数组 31
8.11 `defined` 与 `exists` 的差异 31

9 正则表达式 **32**

9.1 匹配和替换返回数量 32
9.2 不要在未检查匹配成功的情况下使用捕获变量 32
9.3 常用匹配选项 33
9.3.1 `/i`：不区分大小写 33
9.3.2 `/g`：匹配多次 33
9.3.3 `/m`：更改 `^` 和 `$` 的意义 33
9.3.4 `/s`：使 `.` 也匹配换行 34
9.4 捕获变量 `$1` 及之友 34
9.5 利用 `?:` 避免捕获 34
9.6 利用 `/x` 选项使正则表达式更易读 34
9.7 利用 `\Q` 和 `\E` 自动引起正则表达式 35
9.8 对 `s///` 使用 `/e` 选项来执行代码 35
9.9 了解何时使用 `study` 36
9.10 使用 `re => debug` 调试正则表达式 36

10 流程控制 **36**

10.1 四个假值 36

10.2 后缀控制 36

10.3 `for` 循环 37

10.4 `do` 块 37

10.5 Perl 没有 `switch` 或 `case` 38

10.6 `given ... when` 38

10.7 `next/last/continue/redo` 39

11 文件 **39**

11.1 利用 `open` 和 `<>` 操作符更易读取文件 39

11.2 利用 `chomp` 移除结尾的换行符 40

11.3 利用 `$/` 更改行分隔符 40

11.4 一次读取整个文件 40

11.5 利用 `glob()` 获取文件列表 41

11.6 使用 `unlink` 移除文件 42

11.7 在 Windows 下使用 Unix 风格的目录 42

12 子例程 **43**

12.1 使用 `shift` 检索子例程的参数 43

12.2 使用列表赋值来赋给子例程参数 43

12.3 通过访问 `@_` 直接处理子例程参数 43

12.4 传递的参数能被修改 43

12.5 子例程没有检查参数 44

12.6 Perl 有原型，忽略它们 44

12.7 利用 `BEGIN` 块在编译时做事 45

12.8 传递数组及哈希引用 45

13 POD 格式 **46**

13.1 内联文档及格式 46

13.2 使用 `=head1` 和 `=head2` 创建标题 46

13.3 创建无序列表 46

13.4 创建有序列表 47

13.5 使用内联标记样式 47

13.6 使用 `L<>` 超链接 48

13.7 段落模式 vs. 字面块 48

13.8 POD 不会使程序运行变慢 48

14 调试 **48**

14.1 开启 `strict` 和 `warnings` 48

14.2 检查每个 `open` 的返回值 50

14.3 利用 `diagnostics` 扩展警告 50

14.4 使用优化信号来获得栈跟踪信息 51

14.5 使用 Carp::Always 获得栈跟踪信息 52

14.6 使用 Devel::REPL 交互执行 Perl 代码 52

15 模块 **53**

15.1 利用 `use lib` 在非标准位置搜索模块 53

15.2 利用 Module::Starter 创建新模块 53

15.3 利用 `h2xs` 创建 XS 模块 54

15.4 利用 Dist::Zilla 创建、打包及发行模块 54

15.4.1 初始化 Dist::Zilla 配置 54

15.4.2 创建模块 54

15.4.3 打包模块 55

15.4.4 发布模块 55

15.4.5 其他功能 56

16 外部程序 **56**

16.1 `system()` 返回程序的退出状态 56

16.2 反引号（“）和 `qx()` 操作符返回程序的输出 57

17 CPAN **57**

17.1 在 search.cpan.org 搜索模块 57

17.2 在 metacpan.org 替代搜索 57

17.3 在 cpanratings.perl.org 查看关于模块的评论 57

17.4 在 rt.cpan.org 报告 Bug 57

17.5 在 annocpan.org 批注模块文档 57

17.6 在 backpan.perl.org 查找模块的旧版本 58

18 构造 **58**

18.1 C 风格的循环通常不必要 58

18.2 匿名哈希和数组 58

18.3 `q[qrwx]?//`、`m//`、`s///` 及 `y///` 59

18.4 `global`、`local`、`my` 及 `our` 59

19 引用 **59**

19.1 引用是引用其他变量的标量 59

19.2 用箭头符解引用更容易 60

19.3 ref vs. isa 60

19.4 引用匿名子例程 60

20 对象、模块及包 **61**

20.1 包基础 61

20.2 模块基础 61

20.3 对象基础 61

20.4 1; 61

20.5 @ISA 61

21 特殊变量 **62**

21.1 $_ 62

21.2 $0 62

21.3 @ARGV 63

21.4 @INC 63

21.5 %ENV 63

21.6 %SIG 63

21.7 <> 64

21.8 <DATA> 和 __DATA__ 64

21.9 $! 64

21.10 $@ 64

22 命令行选项 **64**

22.1 Shebang 行 64

22.2 perl -T 65

22.3 perl -c file.{pl,pm} 65

22.4 perl -e 'code' 65

22.5 -n、-p、-i 65

22.6 perl -M 66

22.7 明白模块是否已被安装 66

23 高级函数 **67**

23.1 上下文与 wantarray 67

23.2 .. 和 67

24 风格 **68**

24.1 camelCase 很糟 68

24.2 warnings 和 strict 68

24.3 使用 `perltidy` 格式化 Perl 源代码 69

25 性能 **69**

25.1 使用 `while` 而非 `for` 来迭代整个文件 70

25.2 避免不必要的引起和字串化 70

26 陷阱 **71**

26.1 while (<STDIN>) 71

27 如何做...？ **71**

27.1 如何验证 Email 地址是否有效 71

27.2 如何从数据库获得数据 72

27.3 如何从网页获得数据 72

27.4 如何做日期计算 72

27.5 如何处理程序的命令行参数 72

27.6 如何解析 HTML 72

27.7 如何来点颜色 72

27.8 如何读取键及不看到输入的密码 72

28 开发工具 **72**

28.1 剖析性能 72

28.2 分析代码质量 72

28.3 分析变量结构 73

29 出版物 **73**

29.1 Perl 语言编程，第四版 73

29.2 Perl 语言入门，第六版 73

29.3 Perl Cookbook，第二版 73

29.4 Object Oriented Perl 73

29.5 Perl 最佳实践 73

29.6 The Perl Review 73

30 社区 **73**

30.1 Perl 基金会 73

30.2 Perl Mongers 73

30.3 OSCON 74

30.4 YAPC 74

30.5 IRC 频道 74

30.6 邮件列表 74

30.7 PerlMonks 74

30.8 Blog 74

2 如何获得 Perl

既然你已经下定决心要学习 Perl 这门编程语言，那么摆在你面前的第一件事就是得到它。

2.1 你有 Perl 吗

试试从命令行执行 `perl -v`，如果你看到 Perl 的版本及版权等信息，那么说明你的系统已经具有 Perl。反之，如果你看到的是类似 `command not found` 这样的输出，那么你需要安装 Perl。

```
$ perl -v

This is perl 5, version 18, subversion 1 (v5.18.1) built for i486-linux-
gnu-thread-multi-64int
(with 46 registered patches, see perl -V for more detail)

Copyright 1987-2013, Larry Wall

Perl may be copied only under the terms of either the Artistic License
or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl". If you have access to
the
Internet, point your browser at http://www.perl.org/, the Perl Home Page
.
```

2.2 GNU/Linux

Perl 支持许多平台，在 GNU/Linux 上基本都默认带有 Perl。但十有八九可能是旧版本。这种情况下，你可以通过所用 GNU/Linux 发行版的包管理器来更新 Perl。

2.3 Mac OS X

Mac OS X 系统本身也默认安装了 Perl，不过可能仍然存在版本过旧的问题。为此，你可以自己安装更新版。

2.4 Windows

Windows 系统默认没有 Perl。你可以选择下列 Perl 发行之一:

1. Strawberry：称为草莓 Perl，它专为 Windows 平台而生，其中打包了 CPAN 客户端、编译器、以及预装了大量模块。除非你有很特殊的需求，一般来说这就是你所需要的 Perl 发行。

2. ActiveState：Perl 针对 Windows 平台的发起者，至今仍然活跃参与社区。ActiveState 发布自己打包的 Perl，并且包含 PPM 模块安装系统。如果你嫌麻烦，不想自己管理 Perl 安装，那么它也许适合你。

2.5 Perl 源代码

Perl 源代码位于。如果你打算自行编译安装 Perl，需要准备编译器、Shell、以及某些系统库。如果你缺少某些东东，Perl 的 `Configure` 脚本将告诉你。通过以下指令可以从源代码编译并安装 Perl：

```
$ wget http://www.cpan.org/src/5.0/perl-5.18.1.tar.gz
$ tar -xzf perl-5.18.1.tar.gz
$ cd perl-5.18.1
$ ./Configure -des -Dprefix=$HOME/localperl
$ make
$ make test
$ make install
```

2.6 Perlbrew 和 Plenv

除了手动从源代码编译、安装 Perl 之外，你也可以选用时下比较流行的 Perl 多版本管理工具 Perlbrew 或 Plenv。

2.6.1 Perlbrew

要安装 Perlbrew，你可以在终端中执行：

```
$ curl -L http://install.perlbrew.pl | bash
```

然后，将下列内容添加到 `.bashrc` 或 `.zshrc` 文件中：

```
source ~/perl5/perlbrew/etc/bashrc
```

接着执行：

```
$ source ~/.bashrc
$ source ~/.zshrc
```

至此，你便能够使用 Perlbrew 来安装 Perl 的各种版本了。

先列出可用的 Perl 版本：

```
$ perlbrew available
```

安装具体的 Perl 版本：

```
$ perlbrew install 5.18.1
```

待安装完毕，你可以通过以下指令来切换到刚安装的 Perl 版本：

```
$ perlbrew switch perl-5.18.1
```

此外，Perlbrew 还有列出已安装的 Perl 版本、暂时关闭自身等功能，具体可以查看其帮助文档。

2.6.2 Plenv

Plenv 的功能与 Perlbrew 类似，其安装步骤为：

```
$ git clone git://github.com/tokuhirom/plenv.git ~/.plenv
$ echo 'export PATH="$HOME/.plenv/bin:$PATH"' >> ~/.bash_profile
$ echo 'eval "$(plenv init -)"' >> ~/.bash_profile
$ exec $SHELL -l
$ git clone git://github.com/tokuhirom/Perl-Build.git ~/.plenv/plugins/
perl-build/
```

注意：Zsh 用户需将上述指令中的 `.bash_profile` 替换为 `.zshrc`。另外，Ubuntu 用户需将其替换成 `.profile`。

现在，你可以使用 Plenv 来安装 Perl：

```
$ plenv install 5.18.1
```

安装完成后需要执行 `plenv rehash` 重建 shim 可执行文件。

Plenv 能够将某个 Perl 版本设置成局部、全局及 Shell 作用环境。其命令分别为：

```
$ plenv local 5.18.1 # 设置为局部作用环境，比全局作用环境具有更高的优先级
$ plenv global 5.18.1 # 设置成全局作用环境，将在所有 Shell 中使用
$ plenv shell 5.18.1 # 设置成 Shell 作用环境，具有最高的优先级
```

关于 Plenv 的更多用法，可以通过 `plenv help` 查阅。

3 术语

3.1 Perl、perl 及 PERL

Perl 指语言名称。

perl 为解释器，它用来执行 Perl 源文件。

PERL 则是错误的称法。

3.2 Perl 不是缩写词

回溯词 “Practical Extraction and Reporting Language” 和 “Pathologically Eclectic Rubbish Lister” 是在 Perl 这个名称被创造之后才出现的。

3.3 There Is More Than One Way To Do It

“办法不只一种”，这是 Perl 的座右铭，它拼为 *TMTOWTDI*，读作 “Tim Toady”。

3.4 Larry

Larry 指 Larry Wall，他是 Perl 的发明人及其领袖。当前，他虽然主要生活在 Perl 6 岛上，但仍然在对 Perl 5 作重要贡献。

3.5 Randal

Randal 指 Randal Schwartz，他是最大的 Perl 咨询及培训公司之一 Stonehenge 的所有者。他也花时间在 IRC 及其他论坛上帮助 Perl 新手。

Larry 和 Randal 也是 *Programming Perl* 的主要作者。

3.6 Damian

Damian 指 Damian Conway，一个偏心的邪恶天才及 Perl 6 大师。*Perl Best Practices* 一书是他的创新工作，激励人们改善代码写作质量。

3.7 懒惰、急躁和傲慢

程序员的三大美德是懒惰、急躁和傲慢。

懒惰的程序员将重用他们完成的成果。此美德鼓励通过仅写一次来让代码得到重用。

急躁的程序员将更快速的完成工作，并利用机器来完成他们不想手动去做的任务。

傲慢的程序员编写的代码更高效、更清晰、以及更具有可读性，他们因此而自豪。

Perl 正是建立在这些美德之上。

3.8 DWIM

即 Do What I Mean，我的意图紧密相关，但智能的机器并不存在。

3.9 DTRT

即 Do The Right Thing，Perl 试图做正确的事，这与 DWIM 尽可能接近。

3.10 脚本与程序

Perl 程序常被称为与 Windows 批处理文件和 Unix shell 脚本相似的“脚本”。事实上它们不是。Perl 程序就是真正的程序。

Perl 也被称为“脚本语言”，尤其是在 1990 年代当“Web 脚本”帮助推进 Web 时变得广泛流行。然而，Perl 每方面的能力都跟 Java、C++、Ruby、COBOL 或其他你能叫出的任何编程语言一样多。称它“脚本语言”否认了它的强大、灵活及优雅。

Perl、Python、PHP、Ruby 这类被归入次级名称“脚本语言”下的语言，应当称为“动态语言”更正确。学会认为 Perl 是一门编程语言将帮助你欣赏 Perl 的强大，并推动 Perl 得到应有的地位。

4 文档

4.1 perldoc

`perldoc` 是 Perl 完整的语言参考工具。

- `perldoc perldoc`: 用法介绍
- `perldoc perl`: 文档概述
- `perldoc Module`: 模块文档

4.2 利用 `perldoc -f` 查阅 Perl 函数

```
$ perldoc -f system

system LIST
system PROGRAM LIST
       Does exactly the same thing as "exec LIST", except that a fork
       is done first and the parent process waits for the child
...
```

4.3 利用 `perldoc -v` 查阅 Perl 变量

```
$ perldoc -v '$/'

IO::Handle->input_record_separator( EXPR )
$INPUT_RECORD_SEPARATOR
$RS
$/ The input record separator, newline by default. This
...
```

4.4 利用 `perldoc -q` 搜索 Perl FAQ

```
$ perldoc -q database

Found in /usr/share/perl/5.18/pod/perlfaq8.pod
How do I use an SQL database?

   The DBI module provides an abstract interface to most database
   servers
   and types, including Oracle, DB2, Sybase, mysql, Postgresql, ODBC,
   and
...
```

4.5 利用 perldoc modulename 查阅 Perl 模块文档

如果模块已经安装到你的系统中，那么你可以通过 perldoc 阅读该模块的文档。

```
$ perldoc WWW::Mechanize

NAME
    WWW::Mechanize - Handy web browsing in a Perl object

VERSION
    Version 1.73

SYNOPSIS
    "WWW::Mechanize", or Mech for short, is a Perl module for stateful
    programmatic web browsing, used for automating interaction with
    websites.
...
```

对于没有安装的模块，你将需要使用 http://search.cpan.org 或 cpandoc。

4.6 利用 cpandoc 查阅未安装 Perl 模块的文档

Pod::Cpandoc 模块提供了 cpandoc 工具，利用该工具，即便模块没有安装到系统上，但你仍然能够查阅该模块的文档。

```
$ cpandoc Web::Scraper

NAME
    Web::Scraper - Web Scraping Toolkit using HTML and CSS Selectors or
    XPath expressions

SYNOPSIS
    use URI;
    use Web::Scraper;
...
```

4.7 在线文档

一些网站维护有 Perl 的 HTML 文档，最大的两个站点是：

1. ：语言、函数及标准库
2. ：模块

4.8 编写自己的文档

Perl 具有浓烈的文档文化，我们鼓励你早日养成此习惯。你将使用一种叫做 POD 的格式来编写文档。

5 字符串

5.1 利用内插将字符串嵌入到另一个字符串中

双引号（“”）字符串能够内插其他变量。

```
my $name     = "Inigo Montoya";
my $relative = "father";

print "My name is $name, you killed my $relative";
```

5.2 非内插字符串

如果你不想要内插，那么使用单引号（”）。

```
print 'You may have won $1,000,000';
```

或者，你也可以转义特殊字符（印记）。

```
print "You may have won \$1,000,000";
```

5.3 在字符串中使用 Email 地址要小心

此 Email 地址并不是你想要的：

```
my $email = "andy@foo.com";
print $email;
# Prints "andy.com"
```

这里的问题是 *@foo* 作为数组被内插了。如果你打开了 `use warnings`，那么此类问题就很明显了：

```
$ perl foo.pl

Possible unintended interpolation of @foo in string at foo line 1.
andy.com
```

解决办法是，要么使用非内插的引号：

```
my $email = 'andy@foo.com';
my $email = q{andy@foo.com};
```

要么转义 @：

```
my $email = "andy\@foo.com";
```

好的着色代码编辑器将帮助你防止此类问题。

5.4 使用 length() 获得字符串的长度

```
my $str = "Chicago Perl Mongers";
print length( $str ); # 20
```

5.5 使用 substr() 提取字符串

substr() 能够做各种字符串提取：

```
my $x = "Chicago Perl Mongers";

print substr( $x, 0, 4 );   # Chic
print substr( $x, 13 );     # Mongers
print substr( $x, -4 );     # gers
```

5.6 关于字符串 vs. 数字不必担心太多

不像其他语言，Perl 不知道字符串来自于数字。它将做最好的 DTRT。

```
my $phone    = "312-588-2300";

my $exchange = substr( $phone, 4, 3 ); # 588
print sqrt( $exchange ); # 24.2487113059643
```

5.7 利用 ++ 操作符自增非数字字符串

你能够利用 ++ 来自增字符串。字符串 abc 自增后变成 abd。

```
$ cat foo.pl

$a = 'abc'; $a = $a + 1;
$b = 'abc'; $b += 1;
$c = 'abc'; $c++;

print join ", ", ( $a, $b, $c );

$ perl -l foo.pl

1, 1, abd
```

注意：你必须使用 ++ 操作符。在上述示例中，字符串 abc 被转换成 *0*，然后再自增。

5.8 利用 heredocs 创建长字符串

Heredocs 允许连续文本，直到遇到下一个标记。使用内插，除非标记在单引号内。

```
my $page = <<HERE;
<html>
    <head><title>$title</title></head>
    <body>This is a page.</body>
</html>
HERE
```

6 数字

6.1 小心地测试浮点数的相等性

在计算中过度相信 IEEE 浮点数是一种错误。例如：

```
print "---\n";
print "A: ", 2.4, "\n";
print "B: ", 0.2*12, "\n";
if ( 0.2*12 == 2.4 ) {
    print "These are equal.\n";
}
else {
```

```
    print "These are not equal.\n";
}

A: 2.4
B: 2.4
These are not equal.
```

这样的结果是由于这个事实：*0.2* (1/5) 在 IEEE 空间无法被表示为二进制分数。

因此，如果你想要检查浮点数的相等性，那么可以使用 `sprintf` 或类似的东东。

参阅 http://perldoc.perl.org/perlfaq4.html 了解细节。

6.2 数字舍入

不要使用 `int()` 来做数字舍入，它只会返回整数部分。`sprintf()` 或 `printf()` 一般够用了。

```
printf("%.3f", 3.1415926535);   # prints 3.142
```

如果你需要向下或向上舍入，那么可以使用 POSIX 模块所提供的 `ceil()` 和 `floor()` 函数。其中，`ceil()` 是向上舍入，而 `floor()` 是向下舍入。

```
use POSIX;
my $ceil  = ceil(3.5);   # 4
my $floor = floor(3.5);  # 3
```

7 数组

7.1 简单创建单词数组

`qw` 操作符使创建数组更容易。它意为将引用空白分成列表。

```
# Perl 5
my @stooges = qw( Larry Curly Moe Iggy );
# or
my @stooges = qw(
    Larry
    Curly
```

```
    Moe
    Iggy
);
```

在 Perl 6 中，qw 更简洁：

```
my @stooges = < Larry Curly Moe Iggy >;
# or
my @stooges = <
    Larry
    Curly
    Moe
    Iggy
>;
```

元素不做内插：

```
my @array = qw( $100,000 ); # Perl 5
my @array = < $100,000 >;   # Perl 6
```

@array 的单一元素是 “$100,000”。

7.2 利用数字值访问数组

要获得数组的单个标量，使用 [] 和 $ 印记。在 Perl 中所有数组都从 0 开始。

```
$stooges[1] # Curly
```

数组也能使用负偏移从尾端访问。

```
$stooges[-1] # Iggy
```

读取不存在的元素将得到 undef。

```
$stooges[47] # undef
```

7.3 数组的长度是标量值

将数组放到标量环境能得到其长度。一些人也喜欢直接使用 scalar。

```perl
my $stooge_count = scalar @stooges; # 4

my $stooge_count = @stooges;        # 4
```

不要使用 `length` 来获得数组的长度。那只会得到字符串的长度。

```perl
my $moe_length = length $stooges\citep{stooges/2};
                 length $stooges[2];
                 length 'Moe';
                 3;
```

7.4 数组没有边界

数组没有任何有限的大小，且不必预先声明。数组按需更改其大小。

数组也不稀疏。下列代码创建 10,000 个元素的数组。

```perl
my @array     = ();
$array[10000] = 'x';
```

@array 现在具有 10,001 个元素（0-10,000）。它只填入了一个，其他 10,000 个都是 `undef`。

7.5 数组会平展，但不会嵌套

不像 PHP，当数组合并时会平展开为一个大列表。

```perl
my @sandwich       = ( 'PB', 'J' );
my @other_sandwich = ( 'B', 'L', 'T' );
my @ingredients    = ( @other_sandwich, @sandwich );
# ( 'B', 'L', 'T', 'PB', 'J' )
```

这意味着你不能将一个数组包含到另一个数组或哈希中。那样的话，你将需要使用引用。

7.6 列表能带扩展的逗号（,）

Perl 最好的特性之一是在列表的尾端能带一个扩展的逗号。例如：

```
my @array = (
    'This thing',
    'That thing',
);
```

当你编辑代码时，这使添加或者删除项目十分容易，因为你无需将最后一个项目作为特殊情况处理。

```
my @array = (
    'This thing',
    'That thing',
    'The other thing',
);
```

7.7 像队列和堆栈一样使用数组

`shift` 从数组开头移除元素。

```
my $next_customer = shift @customers;
```

`unshift` 将元素添加到数组的开头。

```
unshift @customers, $line_jumper;
```

`push` 将元素添加到数组的结尾。

```
push @customers, $dio; # The last in line
```

`pop` 从数组的结尾移除元素。

```
my $went_home = pop @customers;
```

7.8 利用数组分片提取数组的部分元素

数组分片只是使用多个索引访问数组。

```
my @a = 'a'..'z'; # 26 letters

# a, e, i, o, u...
my @vowels = @a[0,4,8,14,20];
```

```
# And sometimes "y"
push( @vowels, $a[-2] ) if rand > .5;
```

注意：当访问数组分片时，印记是 `@`，不是 `$`。因为你返回的是数组，而不是标量。新手常范的错误是使用 `@` 印记而不是 `$` 访问一个数组元素。那将返回分片，实则是列表。

```
# WRONG: Returns a 1-element list, or 1 in scalar context
my $z = @a[-1];

# RIGHT: Returns a single scalar element
my $z = $a[-1];
```

7.9 利用数组分片来分配数组块

你能够将数组分片作为左值（*lvalues*），即能赋值给等号左边的值。

```
# Replace vowels with uppercase versions
@a[0,4,8,14,20] = qw( A E I O U );

# Swap first and last elements
@a[0,-1] = @a[-1,0];
```

注意：分片的左边和右边大小必须相同。在等号右边缺少的值将使用 `undef` 换掉。

7.10 利用 `splice` 就地修改数组

`splice` 让你拼接数组为另一个数组。让我们来看几个常见的错误做法，那应该能说明它的有用性。

```
my @a = qw(Steve Stu Stan);
$a[1] = ['Stewart', 'Zane'];
# @a  = ('Steve', ARRAY(0x841214c), 'Stan')
# Memory address to an array reference

my @a = qw(Steve Stu Stan);
my @b = qw(Stewart Zane);
$a[1] = @b;
```

```
# @a  = ('Steve', 2, 'Stan')
# Returns a scalar reference, the length of @b
```

现在使用 splice：

```
@a = qw(Steve Stu Stan);
splice @a, 1, 1, 'Stewart', 'Zane';
# @a = ('Steve', 'Stewart', 'Zane', 'Stan')
# This is just what we wanted
```

让我们分解 splice 的参数列表：首先，我们命名要操作的数组（@a）；其次，我们定义偏移（离我们想拼接的列表开始有多远）；第三，我们指定拼接的长度；最后，我们列出想要插入的项目。

如果遇到错误，那么通过 perldoc -f splice 来了解详情。

7.11 利用 map 处理数组

map 本质上是返回列表的 foreach 循环。

你可以使用它将数组转换成哈希：

```
my @array = ( 1, 2, 3, 4, 5 );
my %hash  = map { $_ => $_ * 9 } @array;
# %hash   = ( 1 => 9, 2 => 18, 3 => 27, 4 => 36, 5 => 45 )
```

或者变成列表：

```
my @array       = ( 'ReD', 'bLue', 'GrEEN' );
my @fixed_array = map(ucfirst, map(lc, @array)); # note the nested 'map' functi
# @fixed_array  = ( 'Red', 'Blue', 'Green' )
```

注意：如果你修改 $_，源数据也会被修改。这样，上例可以修改成：

```
my @array = ( 'ReD', 'bLue', 'GrEEN' );
map { $_ = ucfirst lc $_ } @array;
# @array = ( 'Red', 'Blue', 'Green' )
```

7.12 利用 grep 从数组选择项目

grep 本质上也是返回列表的 foreach 循环。但它不像 map，它只返回导致条件为真的元素。

```
my @array     = ( 0, 1, 2, 3, 4, 5 );
my @new_array = grep { $_ * 9 } @array;
# @new_array  = ( 1, 2, 3, 4, 5 );
```

它也将如 map 一样修改源数据：

```
my @array     = ( 0, 1, 2, 3, 4, 5 );
my @new_array = grep { $_ *= 9 } @array;
# @array      = ( 0, 9, 18, 27, 36, 45 );
# @new_array  = ( 9, 18, 27, 36, 45 );
```

我们也可以传递正则表达式给 grep。在本例中，我们只想把包含 *Doe* 的人放到新数组：

```
my @people = (
    'John Doe',
    'Jane Doe',
    'Joe Sixpack',
    'John Q. Public',
);

my @does = grep { $_ =~ /\bDoe$/ } @people;
# @does  = ('John Doe', 'Jane Doe');
```

或更短的：

```
my @does = grep { /\bDoe$/ } @people;
```

8 哈希

8.1 哈希是键/值对

如果你想按名字查询，那么需要哈希。哈希的键必须唯一，但值可以是任意标量。

有时候你仍然会看到人们称它为 “关联数组”，但不要想当然的把它作为数组。

8.2 通过键/值对列表来创建哈希

使用键/值对列表创建哈希：

```
my %stooges = (
    'Moe', 'Howard',
    'Larry', 'Fine',
    'Curly', 'Howard',
    'Iggy', 'Pop',
);
```

=> 称为胖逗号，它与逗号相同，前面的单词加上引号：

```
my %stooges = (
    Moe   => 'Howard',
    Larry => 'Fine',
    Curly => 'Howard',
    Iggy  => 'Pop',
);
```

哈希在列表环境中变成键/值对列表。

```
my @hash_as_an_array = %stooges;
# Contains ( 'Curly', 'Howard', 'Larry', 'Fine', etc... )
```

正如哈希的键和值顺序事实上随机的一样，平展开的哈希键/值顺序也是随机的。

8.3 利用花括号访问独立的哈希条目

利用花括号代替中括号来获取哈希的值。

```
print $stooges{'Iggy'};
# Prints "Pop"
```

用同样的方式设置值：

```
$stooges{'Shemp'} = 'Howard';
```

覆盖现有的值：

```
$stooges{'Iggy'} = 'Ignatowski';
```

从哈希中删除条目：

```
delete $stooges{'Curly'};
```

注意：delete 不会删除文件。unlink 才会。

```
unlink $stooges{'Moe'};
# Deletes a file called 'Howard';
```

8.4 获取哈希的键/值

使用 keys 和 values 关键字：

```
my @stooge_first_names = keys %stooges;

my @stooge_last_names  = values %stooges;
```

这会保证键和值的顺序相匹配。

8.5 哈希键自引用单词

如果你的哈希键是单个词，那么你不需要引用它。

```
$stooges{Curly} = 'Howard';
```

8.6 哈希只能包含标量

在 Perl 哈希中的值只能是标量。它不能包含数组或数组中的列表。

```
$hash{comedians} = @stooges;
# Assigns the length of @stooges to the value
```

如果你想要在哈希中存储数组，你将需要使用引用。

8.7 哈希是无序的

keys %hash 和 values %hash 的顺序实事上是随机的。每次执行程序都将不同。它也与添加时的顺序不相关。

如果你想保留哈希元素添加时的顺序，那么可以使用 Tie::IxHash 模块。

8.8 你无法排序哈希

在 Perl 中排序哈希的想法不存在，因为哈希是无序的。你可以排序哈希的键，或哈希的值，它们只是列表而已。

8.9 使用列表赋值合并哈希

要合并两个哈希，将它们看作列表，并赋给哈希。

```
my %new_hash = (%hash1, %hash2);
```

等号右边是来自两个哈希的键/值对长列表。然后将该列表赋给 %new_hash。如果在 %hash2 中的任意键与 %hash1 中的键重复，那么 %hash2 中的键/值对具有更高的优先级，因为它们赋值更晚。

8.10 何时使用哈希，何时使用数组

如果你做线性、有序的序列，那么使用数组。

- 要读取的文件列表
- 队列中的人列表

如果你做想要查询的无序的事，那么使用哈希。

- 姓索引，通过名字查询
- 显示文件大小的索引，按名称查询

8.11 defined 与 exists 的差异

使用 defined 来看哈希元素是否有 undef 之外的值。如果哈希元素有任意 undef 之外的值，甚至求值为假的 0 和 “”(空字符串)，都将返回真。

使用 exists 来看哈希元素是否已被初始化，即便它没有被定义（如，它有值 undef)。

```
my %h;
$h{'foo'} = undef;

defined $h{'foo'} ? print 1 : print 0;
# $h{'foo'} is not defined, so it prints 0
exists $h{'foo'} ? print 1 : print 0;
```

```
# but it has been initialized nonetheless, and so this line prints 1
```

哈希元素仅被定义后才为真。它仅在存在后才能被定义。

然而，哈希元素未被定义仍能存在。这意味着即便它存在，也不会返回真。

```
if ( $h{'foo'} ) {
    print 'true';
}
else {
    print 'false';
}
# prints 'false'; since $h{'foo'} is not defined, it cannot be true
```

9 正则表达式

正则表达式比此处所介绍的主题要大得多，确信你理解了这些概念。对于教程，可以参阅 perlrequick 或 perlretut。而权威的文档，则能够参阅 perlre。

9.1 匹配和替换返回数量

`m//` 和 `s///` 操作符分别返回匹配或替换的数目。你既可以直接使用该数目，也可以检查其真值。

```
if ( $str =~ /Diggle|Shelley/ ) {
    print "We found Pete or Steve!\n";
}

if ( my $n = ($str =~ s/this/that/g) ) {
    print qq{Replaced $n occurrence(s) of "this"\n};
}
```

9.2 不要在未检查匹配成功的情况下使用捕获变量

除非匹配成功，捕获变量 `$1` 等是无效的，并且它们不会被清理。

```
# BAD: Not checked, but at least it "works".
my $str = 'Perl 101 rocks.';
$str =~ /(\d+)/;
print "Number: $1"; # Prints "Number: 101";

# WORSE: Not checked, and the result is not what you'd expect
$str =~ /(Python|Ruby)/;
print "Language: $1"; # Prints "Language: 101";
```

你必须检查匹配的返回值：

```
# GOOD: Check the results
my $str = 'Perl 101 rocks.';
if ( $str =~ /(\d+)/ ) {
    print "Number: $1"; # Prints "Number: 101";
}

if ( $str =~ /(Python|Ruby)/ ) {
    print "Language: $1"; # Never gets here
}
```

9.3 常用匹配选项

9.3.1 /i：不区分大小写

9.3.2 /g：匹配多次

```
$var = "match match match";

while ($var =~ /match/g) { $a++; }
print "$a\n"; # prints 3

$a = 0;
$a++ foreach ($var =~ /match/g);
print "$a\n"; # prints 3
```

9.3.3 /m：更改 ^ 和 $ 的意义

正常情况下，^ 意为字符串的开头，而 $ 为字符串的结尾。/m 使它们分别意为行首和行尾。

```
$str = "one\ntwo\nthree";
@a = $str =~ /^\w+/g;  # @a = ("one");
@b = $str =~ /^\w+/gm; # @b = ("one","two","three")
```

不管是否有 /m，使用 \A 和 \z 来匹配字符串的开头和结尾。\Z 除了会忽略最后的换行之外，与 \z 相同，

9.3.4 /s：使 . 也匹配换行

```
$str = "one\ntwo\nthree\n";
$str =~ /^(.{8})/s;
print $1; # prints "one\ntwo\n"
```

9.4 捕获变量 $1 及之友

捕获括号对的内容被存储到数字变量中。括号从左到右分配：

```
my $str = "abc";
$str =~ /(((a)(b))(c))/;
print "1: $1 2: $2 3: $3 4: $4 5: $5\n";
# prints: 1: abc 2: ab 3: a 4: b 5: c
```

捕获括号及变量的数目没有上限。

9.5 利用 ?: 避免捕获

如果括号后紧接着 ?:，那么该分组不会被捕获。在你不想保存匹配的内容时会有用：

```
my $str = "abc";
$str =~ /(?:a(b)c)/;
print "$1\n"; # prints "b"
```

9.6 利用 /x 选项使正则表达式更易读

如果你在使用正则表达式时玩了些花样，那么为它写注释。你可以使用 /x 选项达到目的。

丑陋的庞然大物：

```
my ($num) = $ARGV[0] =~ m/^\+?((?:(?<!\+)-)?(?:\d*.)?\d+)$/x;
```

使用 /x 允许的空白和注释更可读:

```
my ($num) =
    $ARGV[0] =~ m/^ \+?         # An optional plus sign, to be discarded
                  (             # Capture...
                  (?:(?<!\+)-)? # a negative sign, if there's no plus behind it,
                  (?:\d*.)?     # an optional number, followed by a point if a d
                  \d+           # then any number of numbers.
                  )$/x;
```

除非被转义，空白和注释将被去除。

9.7 利用 \Q 和 \E 自动引起正则表达式

这会自动转义正则表达式的元字符。不会转义 $ 符号。

```
my $num = '3.1415';
print "ok 1\n" if $num =~ /\Q3.14\E/;
$num = '3X1415';
print "ok 2\n" if $num =~ /\Q3.14\E/;
print "ok 3\n" if $num =~ /3.14/;
```

输出:

```
ok 1
ok 3
```

9.8 对 s/// 使用 /e 选项来执行代码

这将允许任意代码替换正则表达式中的字符串。

```
my $str = "AbCdE\n";
$str =~ s/(\w)/lc $1/eg;
print $str; # prints "abcde"
```

必要时使用 $1 及之友。

9.9 了解何时使用 study

study 在多数情况下都无用。它所做的是创建一个每个单字节字符首次出现在字符串中的位置的表。这意味着如果你有 1,000 个字符长的字符串，你要寻找由一个常量字符开头的各种字符串，匹配器可以立即跳转到正确的位置。例如：

```
"This is a very long [... 900 characters skipped...] string that I have
here,
ending at position 1000"
```

现在，如果你要匹配正则表达式 /Icky/，匹配器将试图寻找第一个匹配的字母 I。在找到它之前得扫描前面的 900+ 个字符。但 study 创建了一个 256 个字节第一次出现的地方的表。所以扫描器能够立即跳转到那个位置来开始匹配。

译注：这里没有考虑到多字节字符。

9.10 使用 re => debug 调试正则表达式

```
-Mre=debug
```

10 流程控制

10.1 四个假值

在 Perl 中有 4 种假值：

```
my  $false = undef;
    $false = "";
    $false = 0;
    $false = "0";
```

最后一个为假值是因为 "0" 在数字上下文中将变成 0，根据第三条规则，它是假值。

10.2 后缀控制

简单的 if 或 unless 块可能看起来像这样：

```
if ($is_frobnitz) {
    print "FROBNITZ DETECTED!\n";
}
```

在这些情况下，`if` 或 `unless` 能够追加到简单语句的尾部。

```
print "FROBNITZ DETECTED!\n" if $is_frobnitz;
die "BAILING ON FROBNITZ!\n" unless $deal_with_frobnitz;
```

`while` 和 `for` 也可以这样用。

```
print $i++ . "\n" while $i < 10;
```

10.3 `for` 循环

`for` 循环有三种风格。

```
my @array;

# Old style C for loops
for (my $i = 0; $i < 10; $i++) {
    $array[$i] = $i;
}

# Iterating loops
for my $i (@array) {
    print "$i\n";
}

# Postfix for loops
print "$_\n" for @array;
```

你也许会看到 `foreach` 用于 `for` 的位置。它们两个可以互换。在上述后两种循环风格中多数人使用 `foreach`。

10.4 `do` 块

`do` 允许 Perl 在期待语句的位置使用块。

```
open( my $file, '<', $filename ) or die "Can't open $filename: $!"
```

但如果你需要做别的事：

```
open( my $file, '<', $filename ) or do {
    close_open_data_source();
    die "Aborting: Can't open $filename: $!\n";
};
```

下列代码也是等价的：

```
if ($condition) { action(); }
do { action(); } if $condition;
```

作为特殊情况，`while` 至少执行块一次。

```
do { action(); } while action_needed;
```

10.5 Perl 没有 `switch` 或 `case`

如果你从其他语言而来，你可能用过 `case` 语句。Perl 没有它们。

最接近的我们有 `elsif`：

```
if ($condition_one) {
    action_one();
}
elsif ($condition_two) {
    action_two();
}
...
else {
    action_n();
}
```

没有办法像 `case` 那样清晰。

10.6 `given ... when`

从 Perl 5.10.1 开始，你可以使用下列代码来打开实验性的 `switch` 特性：

```
use feature "switch";

given ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

10.7 next/last/continue/redo

考虑以下循环:

```
$i = 0;
while ( 1 ) {
    last if $i > 3;
    $i++;
    next if $i == 1;
    redo if $i == 2;
}
continue {
    print "$i\n";
}
```

输出:

```
1
3
4
```

`next` 跳到块尾并继续或重新开始。

`redo` 立即跳回到循环的开头。

`last` 跳到块尾并从再次执行时停止循环。

`continue` 在块尾执行。

11 文件

11.1 利用 open 和 <> 操作符更易读取文件

使用 Perl 打开并读取文件很简单。下面的示例代码演示如何打开文件，一行一行地读取，检查匹配正则表达式的文本，以及输出匹配的行。

```
open( my $fh, '<', $filename ) or die "Can't open $filename: $!";
while ( my $line = <$fh> ) {
    if ( $line =~ /wanted text/ ) {
        print $line;
    }
}
close $fh;
```

总是检查 open 的返回码是否为真。如果为假，其结果在 $! 中。

11.2 利用 chomp 移除结尾的换行符

从文件读取行时会包含结尾的换行符。假如你有一个文本文件，其第一行是：

```
Aaron
```

Aaron 实际上是 6 个字符 Aaron\n。此代码将失败：

```
my $line = <$fh>;
if ( $line eq 'Aaron' ) {
    # won't reach here, because it's really "Aaron\n";
}
```

要移除 \n 及结尾的其他任意空白，调用 chomp。

```
my $line = <$fh>;
chomp $line;
```

现在 $line 为 5 个字符长。

11.3 利用 $/ 更改行分隔符

可以更改输入记录分隔符 $/，其默认设置为 \n。

设置 $/ 一次读取一段。设置 $/ 为 undef 将一次读取整个文件。参阅 perlvar 了解细节。

11.4 一次读取整个文件

你将注意到新手在读取文件时会使用下述两种方法之一：

```
open (FILE,$filename) || die "Cannot open '$filename': $!";
undef $/;
my $file_as_string = <FILE>;
```

或：

```
open (FILE,$filename) || die "Cannot open '$filename': $!";
my $file_as_string = join '', <FILE>;
```

选择两种中的前者。第二种读取所有行到数组，然后组合成一个大字符串。第一种仅读取到字符串，不会间接创建行列表。

然而最佳的方式是像这样：

```
my $file_as_string = do {
    open( my $fh, $filename ) or die "Can't open $filename: $!";
    local $/ = undef;
    <$fh>;
};
```

do 块返回块中最后求解的值。此方法将 $/ 设置为局部作用域，所以超出块范围会设置为默认值。如果没有局部化 $/，那么它将保留设置的值，其他代码段可能并不期望它被设置为 undef。

下面是另一种方法：

```
use File::Slurp qw( read_file );
my $file_as_string = read_file( $filename );
```

File::Slurp 是一次性读取和写入的有用模块，它将在背后做魔术般的快速处理。

11.5 利用 glob() 获取文件列表

使用标准的 Shell 展开模式来获取文件列表。

```
my @files = glob( "*" );
```

将它们传递给 grep 来做快速过滤。例如，要获取文件而非目录：

```
my @files = grep { -f } glob( "*" );
```

11.6 使用 unlink 移除文件

Perl 内置函数 delete 用来删除哈希的元素，而非文件系统中的文件。

```
my %stats;

$stats{filename} = 'foo.txt';

unlink $stats{filename}; # RIGHT: Removes "foo.txt" from the filesystem

delete $stats{filename}; # WRONG: Removes the "filename" element from %stats
```

术语 unlink 来自于 Unix 从目录节点移除文件链接的想法。

11.7 在 Windows 下使用 Unix 风格的目录

即使 Unix 使用 /usr/local/bin，而 Windows 使用 C:\foo\bar\bat 这样的路径，你仍然能够在文件名中使用斜杠。

```
my $filename = 'C:/foo/bar/bat';
open( my $fh, '<', $filename ) or die "Can't open $filename: $!";
```

在这种情况下，Perl 在打开文件前魔术化地将 C:/foo/bar/bat 更改为 C:\foo\bar\bat。这也会防止文件名包含未引起的反斜杠所带来的问题。

```
my $filename = "C:\tmp";
```

$filename 包含 5 个字符：C、:、tab 字符、m、及 p。实际上，它应该写成：

```
my $filename = 'C:\tmp';
my $filename = "C:\\tmp";
```

或者你让 Perl 来照料它：

```
my $filename = 'C:/tmp';
```

12 子例程

12.1 使用 shift 检索子例程的参数

子例程的参数来自于特殊的 @_ 数组。不带参数的 shift 默认使用 @_。

```
sub volume {
    my $height = shift;
    my $width  = shift;
    my $depth  = shift;

    return $height * $width * $depth;
}
```

12.2 使用列表赋值来赋给子例程参数

你也可以使用列表赋值赋给子例程参数:

```
sub volume {
    my ($height, $width, $depth) = @_;

    return $height * $width * $depth;
}
```

12.3 通过访问 @_ 直接处理子例程参数

在某些情况下，但我们希望很少，你能够通过 @_ 数组直接访问参数。

```
sub volume {
    return $_[0] * $_[1] * $_[2];
}
```

12.4 传递的参数能被修改

传递给子例程的参数是实际参数的别名。

```
my $foo = 3;
print incr1($foo) . "\n"; # prints 4
print "$foo\n"; # prints 3

sub incr1 {
```

```
    return $_[0]+1;
}
```

如果你想要这种效果的话，这样更好：

```
sub incr2 {
    return ++$_[0];
}
```

12.5 子例程没有检查参数

如果你喜欢，你能够将任意东东传递给子例程。

```
sub square {
    my $number = shift;

    return $number * $number;
}

my $n = square( 'Dog food', 14.5, 'Blah blah blah' );
```

该函数只会使用第一个参数。因为这个关系，你可以使用任意数目的参数，甚至没有参数来调用函数。

```
my $n = square();
```

Perl 不会对此抱怨。

Params::Validate 模块解决了许多验证问题。

12.6 Perl 有原型，忽略它们

在演进的道路上加入了原型，因此你可以像这样干：

```
sub square($) {
    ...
}

my $n = square( 1, 2, 3 ); # run-time error
```

无论如何都不要使用它们。它们不会作用于对象，它们需要在调用子例程前先予以声明。它们是好想法，但只是不实用。

12.7 利用 BEGIN 块在编译时做事

BEGIN 是一种特殊的代码块类型。它允许程序员在 Perl 的编译阶段执行代码，这样可以执行初始化及做其他事情。

Perl 使用 BEGIN 在任意时导入模块。下列两个语句是等效的：

```
use WWW::Mechanize;

BEGIN {
    require WWW::Mechanize;
    import WWW::Mechanize;
}
```

12.8 传递数组及哈希引用

记住传给子例程的参数是作为一个大数组传递的。如果你像下面这样干：

```
my @stooges    = qw( Moe Larry Curly );
my @sandwiches = qw( tuna ham-n-cheese PBJ );

lunch( @stooges, @sandwiches );
```

那么传给 lunch 的是列表：

```
( "Moe", "Larry", "Curly", "tuna", "ham-n-cheese", "PBJ" );
```

在 lunch 中，你如何能告诉 stooges 结束及 sandwiches 开始的位置？你不能。如果你尝试这样：

```
sub lunch {
    my (@stooges, @sandwiches) = @_;
```

那么所有 6 个元素都会跑到 @stooges 中，而 @sandwiches 什么都不会得到。

答案是使用引用，正如：

```
lunch( \@stooges, \@sandwiches );

sub lunch {
    my $stoogeref   = shift;
    my $sandwichref = shift;
```

```
    my @stooges     = @{$stoogeref};
    my @sandwichref = @{$sandwichref};
    ...
}
```

13 POD 格式

13.1 内联文档及格式

POD 允许你在 Perl 代码中使用标记来创建文档。如果你见过 Javadoc 或 PHPdoc，它跟它们很相像。

POD 是语言的一部分，并非额外的规范。

13.2 使用 =head1 和 =head2 创建标题

```
=head1 MOST IMPORTANT

Blah blah

=head2 Subheading

blah blah

=head2 Subhading

blah blah
```

13.3 创建无序列表

要创建这样的列表：

- Wango
- Tango
- Fandango

使用：

```
=over

=item * Wango

=item * Tango

=item * Fandango

=back
```

13.4 创建有序列表

要创建下面的列表:

1. Visit perl101.org
2. ???
3. Profit!

使用:

```
=over

=item 1 Visit perl101.org

=item 2 ???

=item 3 Profit!

=back
```

13.5 使用内联标记样式

POD 用 `B<>`、`I<>` 及 `C<>` 分别表示粗体、斜体、代码。

```
B<This is bolded>

I<This is italics>
```

```
C<This is code>
```

标记格式能够嵌套。

13.6 使用 L<> 超链接

L<> 既可以链接文档中的关键字，如 L<Methods>；也可以链接 URL，如 L<http://search.cpan.org>。

13.7 段落模式 vs. 字面块

段落会自动换行，并由至少一个空行分隔。

```
A literal block is indented at least one space
and does not
get
wrapped.
```

这来自于：

```
Everything in a paragraph word-wraps automatically.  A paragraph
is separated by at least one blank line.

    A literal block is indented at least one space
    and does not
    get
    wrapped.
```

13.8 POD 不会使程序运行变慢

它将在编译时被去除。

14 调试

14.1 开启 strict 和 warnings

无论何时调试代码，都确信已开启了 strict 和 warnings 编译指令。

将下面两行：

```
use strict;
use warnings;
```

放到你试图调试或将来可能想调试的程序的顶部。

strict 编译指令强制你使用那些允许 Perl 在编译时找出错误的许多特性。首先最重要的是，在 `strict` 下，变量必须在使用前声明。多数情况下，这意味着使用 `my`：

```
use strict;
my $foo = 7;                # OK, normal variable
print "foo is $fooo\n";     # Perl complains and aborts compilation
```

没有 `strict`，Perl 仍然会高兴地执行上面的程序。但你可能会感到杯具，想不明白 `$foo` 为何没有值。启用 `strict` 也会减少许多令人头痛的输入错误。

另外，`strict` 不允许使用多数裸字。

```
no strict;
$foo = Lorem;
print "$foo\n";          # Prints "Lorem"

use strict;
my $foo = ipsum;         # Complains about bareword
$foo = (
    Lorem   => 'ipsum'   # OK, barewords allowed on left of =>
);

$SIG{PIPE} = handler;    # Complains
$SIG{PIPE} = \&handler;  # OK
$SIG{PIPE} = "handler"; # Also, OK, but above is preferred
```

最后，如果你使用符号引用，启用 `strict` 将抛出运行时错误。

```
no strict;
$name = "foo";
$$name = "bar";              # Sets the variable $foo to 1
print "$name $$name\n";      # Prints "foo bar"
```

```
use strict;
my $name = "foo";
$$name = "bar";                 # Complains: can't use "foo" as ref
```

warnings 编译指令将使 Perl 吐出许多有用的抱怨，以便让你知道程序中的某个东东并非你所想要的：

```
use warnings;
my $foo = ;
$foo += 3;
my $foo = 1;                # Compains: redeclaration of variable

my $bar = '12fred34';
my $baz = $bar + 1;         # Complains: Argument "12fred34" isn't numeric
                            # Complains: Name "main::baz" used only once
```

参阅 strict 及 warnings 的文档了解其他信息。关于不用 strict 所带来的恐怖故事，可以看看 PerlMonks 上面的帖子。

14.2 检查每个 open 的返回值

你将经常看到人们抱怨下面的代码无法执行：

```
open( my $file, $filename );
while ( <$file> ) {
    ...
}
```

接着抱怨 while 循环也被破坏了。这儿的常见问题是文件 $filename 实际上并不存在，因此 open 将失败。如果没有检查，那么你将从来不会知道。使用以下代码替换它：

```
open( my $file, '<', $filename ) or die "Can't open $filename: $!";
```

14.3 利用 diagnostics 扩展警告

有时候警告消息并没有解释你想要的那么多。例如，为何你会获得此警告？

```
Use of uninitialized value in string eq at /Library/Perl/5.8.6/WWW/
Mechanize.pm line 695.
```

试试将下列内容放到程序顶部并重新执行代码：

```
use diagnostics;
```

现在警告看起来像这样：

```
Use of uninitialized value in string eq at /Library/Perl/5.8.6/WWW/
Mechanize.pm
   line 695 (#1)
(W uninitialized) An undefined value was used as if it were already
defined. It was interpreted as a "" or a 0, but maybe it was a mistake.
To suppress this warning assign a defined value to your variables.

To help you figure out what was undefined, perl tells you what operation
you used the undefined value in. Note, however, that perl optimizes your
program and the operation displayed in the warning may not necessarily
appear literally in your program. For example, "that $foo" is
usually optimized into "that " . $foo, and the warning will refer to
the concatenation (.) operator, even though there is no . in your
program.
```

更多的信息将帮助你找出程序的问题。

记住你也可以从命令行指定模块和编译指令，因此你甚至不必编辑源代码来使用 `diagnostics`。使用 `-M` 命令行选项再次执行它：

```
perl -Mdiagnostics mycode.pl
```

14.4 使用优化信号来获得栈跟踪信息

有时候你将获得警告，但你并不明白是如何得到的。比如：

```
Use of uninitialized value in string eq at /Library/Perl/5.8.6/WWW/
Mechanize.pm line 695.
```

你可以看到模块的 695 行代码干了什么，但你无法看到你的代码在此时干了什么。你将需要看到子例程被调用的跟踪信息。

当 Perl 调用 `die` 或 `warn` 时，它将分别指定 `$SIG{__DIE__}` 和 `$SIG{__WARN__}` 来通过子例程。如果你修改它们，让其成为比 `CORE::die` 和 `CORE::warn` 更

有用的话，你就得到了一个有用的调试工具。这种情况，可以使用 Carp 模块的 confess 函数。

在你程序的顶部，添加这些行：

```
use Carp qw( confess );
$SIG{__DIE__} =  \&confess;
$SIG{__WARN__} = \&confess;
```

现在，当代码调用 warn 或 die 时，Carp::confess 函数将处理它。confess 打印原始警告，跟着栈跟踪信息，然后停止执行程序。

```
Use of uninitialized value in string eq at /Library/Perl/5.8.6/WWW/Mechanize.pm
    at /Library/Perl/5.8.6/WWW/Mechanize.pm line 695
        WWW::Mechanize::find_link('WWW::Mechanize=HASH(0x180e5bc)', 'n', 'undef
        main::go_find_link('http://www.cnn.com') called at foo.pl line 8
```

现在我们有更多信息来调试代码，包括精确的调用函数及传递的参数。从这儿，我们能够容易地看到 find_link 的第三个参数是 undef，那将是一个开始调查的好地方。

14.5 使用 Carp::Always 获得栈跟踪信息

如果你不想覆盖信号处理器，那么可以安装 CPAN 模块 Carp::Always。

在安装之后，添加下行到你的代码中：

```
use Carp::Always;
```

或者使用 -MCarp::Always 从命令行调用你的程序，这将总是会得到栈跟踪信息。

14.6 使用 Devel::REPL 交互执行 Perl 代码

Devel::REPL 模块提供一个交互式的 Shell。通过该 Shell，你不用创建临时的源代码文件就可以做快速的原型开发及测试代码。

在安装 Devel::REPL 之后，你可以执行以下命令启动 Shell：

```
$ re.pl
```

15 模块

15.1 利用 use lib 在非标准位置搜索模块

要搜索没有安装到 @INC 所指定路径的模块，使用 lib 编译指令：

```
use lib '/home/andy/private-lib/';
use Magic::Foo;
```

注意：use lib 必须置于试图使用 Magic::Foo 之前。

15.2 利用 Module::Starter 创建新模块

Module::Starter 及其命令行工具 module-starter 创建模块发行套件的基本框架，以便发布到 CPAN 上。它包含基本的代码布局、POD 指令、文档片断、基本测试、Makefile.PL 和 MANIFEST 文件、以及 README 和 Changes 记录的开头。

```
$ module-starter --module=Magic::Foo --module=Magic::Foo::Internals \
    --author="Andy Lester" --email="andy@perl.org" --verbose
Created Magic-Foo
Created Magic-Foo/lib/Magic
Created Magic-Foo/lib/Magic/Foo.pm
Created Magic-Foo/lib/Magic/Foo
Created Magic-Foo/lib/Magic/Foo/Internals.pm
Created Magic-Foo/t
Created Magic-Foo/t/pod-coverage.t
Created Magic-Foo/t/pod.t
Created Magic-Foo/t/boilerplate.t
Created Magic-Foo/t/00-load.t
Created Magic-Foo/.cvsignore
Created Magic-Foo/Makefile.PL
Created Magic-Foo/Changes
Created Magic-Foo/README
Created Magic-Foo/MANIFEST
Created starter directories and files
```

15.3 利用 h2xs 创建 XS 模块

如果你想创建 XS 模块，即 Perl 代码与外部 C 代码的接口，那么你将需要使用原始的模块开始工具 `h2xs`。`h2xs` 已包含到每个 Perl 发行中，但它可能并没有 `Module::Starter` 那么新。除非你需要 XS 代码，否则使用 `Module::Starter`。

15.4 利用 Dist::Zilla 创建、打包及发行模块

Dist::Zilla 是一个相当好用的 Perl 模块，它使创建、打包、以及发行模块的过程变得十分容易。如果你打算将编写的模块发布到 CPAN 上，那么使用 Dist::Zilla 将为你节省许多时间。

15.4.1 初始化 Dist::Zilla 配置

安装 Dist::Zilla 之后的第一件事就是初始化其配置：

```
$ dzil setup
```

根据向导提供你的姓名、Email、选择版权许可、以及 PAUSE 帐号（发布模块到 CPAN 时需要）即可。

Dist::Zilla 默认将配置文件保存在 `~/.dzil/config.ini` 文件中，所以后续你也可以通过修改此文件来变更相应信息。

15.4.2 创建模块

执行以下命令可以创建一个新的模块，如 Foo::Bar：

```
$ dzil new Foo::Bar

[DZ] making target dir /home/xiaodong/code/Foo-Bar
[DZ] writing files to /home/xiaodong/code/Foo-Bar
[DZ] dist minted in ./Foo-Bar
```

这将创建如下目录结构及文件：

```
Foo-Bar
├── dist.ini
└── lib
    └── Foo
        └── Bar.pm
```

其中，dist.ini 为该模块的配置文件，Bar.pm 为模块源文件。

15.4.3 打包模块

待模块编写完毕，你就可以将模块打包了：

```
$ dzil build

[DZ] beginning to build WebService-TaobaoIP
[DZ] guessing dist's main_module is lib/WebService/TaobaoIP.pm
[DZ] extracting distribution abstract from lib/WebService/TaobaoIP.pm
[DZ] writing WebService-TaobaoIP in WebService-TaobaoIP-0.03
defined(@array) is deprecated at /usr/share/perl5/Log/Log4perl/Config.pm
 line
864.
    (Maybe you should just omit the defined()?)
[DZ] building archive with Archive::Tar::Wrapper
[DZ] writing archive to WebService-TaobaoIP-0.03.tar.gz
```

执行该命令后，模块就会被打包成 .tar.gz 格式。

15.4.4 发布模块

如果你要将模块发布到 CPAN 上，只需执行：

```
$ dzil release

[DZ] beginning to build WebService-TaobaoIP
[DZ] guessing dist's main_module is lib/WebService/TaobaoIP.pm
[DZ] extracting distribution abstract from lib/WebService/TaobaoIP.pm
[DZ] writing WebService-TaobaoIP in WebService-TaobaoIP-0.03
defined(@array) is deprecated at /usr/share/perl5/Log/Log4perl/Config.pm
 line
864.
    (Maybe you should just omit the defined()?)
[DZ] building archive with Archive::Tar::Wrapper
[DZ] writing archive to WebService-TaobaoIP-0.03.tar.gz
\citep{Basic/TestRelease} Extracting
/home/xiaodong/code/WebService-TaobaoIP/WebService-TaobaoIP-0.03.tar.gz
to .build/x8WYcGBWoY
```

```
Checking if your kit is complete...
Looks good
Writing Makefile for WebService::TaobaoIP
Writing MYMETA.yml and MYMETA.json
cp lib/WebService/TaobaoIP.pm blib/lib/WebService/TaobaoIP.pm
Manifying blib/man3/WebService::TaobaoIP.3pm
No tests defined for WebService::TaobaoIP extension.
\citep{Basic/TestRelease} all's well; removing .build/x8WYcGBWoY
*** Preparing to release WebService-TaobaoIP-0.03.tar.gz with
@Basic/UploadToCPAN ***

Do you want to continue the release process? [y/N]: N
```

根据提示，回答 `y` 将发布模块，`N` 将终止发布过程。

15.4.5 其他功能

Dist::Zilla 不愧为一站式工具，除上述基本功能之外，还包括添加模块到现有发行、执行测试、列出模块依赖等特性。

有关 Dist::Zilla 的更多用法，可参考其官方文档。

16 外部程序

在 Perl 中有三种方式来调用外部程序。

16.1 system() 返回程序的退出状态

```perl
my $rc = system("/bin/cp $file1 $file2"); # returns exit status values
die "system() failed with status $rc" unless $rc == 0;
```

如果可能，用列表传递你的参数，而不是用单个的字符串。

```perl
my $rc = system("/bin/cp", $file1, $file2 );
```

如果 `$file1` 或 `$file2` 有空格或其他特殊字符，这将确保不会在 Shell 中出错。

`system()` 的输出不会被捕获。

16.2 反引号（“）和 qx() 操作符返回程序的输出

当你想要输出时，使用：

```
my $output = `myprogram foo bar`;
```

你将需要检查 $! 中的错误代码。

如果你使用反引号或 qx()，首选 *IPC::Open2* 或 *IPC::Open3* 代替，因为它们将给你相同的参数控制，并允许你捕获输出。

IPC::Open3 是在 Perl 中不使用 Shell 命令来捕获 STDERR 的仅有方法。

17 CPAN

CPAN 是所有其他语言羡慕嫉妒恨的魔弹。它是人们贡献的巨大模块仓库。它意为 *Comprehensive Perl Archive Network*。

17.1 在 search.cpan.org 搜索模块

http://search.cpan.org 是 CPAN 搜索的标准界面。它也包括链向其他站点的链接。

17.2 在 metacpan.org 替代搜索

CPAN 也有一个不同的界面。metacpan.org 具有更多的特性及用于浏览 CPAN 和发行的链接。

17.3 在 cpanratings.perl.org 查看关于模块的评论

http://cpanratings.perl.org 让你对模块进行打分及评论，在你试用之前，不妨看看别人的看法。

17.4 在 rt.cpan.org 报告 Bug

从 http://rt.cpan.org 提交你的 Bug 到 RT。

17.5 在 annocpan.org 批注模块文档

http://annocpan.org 让你对模块文档进行批注，有望让作者在将来合并更改。

17.6 在 backpan.perl.org 查找模块的旧版本

http://backpan.perl.org 具有 CPAN 上的每个模块，甚至包含过时的旧版本。

18 构造

18.1 C 风格的循环通常不必要

你可以写 C 风格的循环，但常常不需要它们。

不要在 `foreach` 的位置使用它们：

```
for (my $i = 0; $i <= $#foo; $i++) { # BAD

foreach (@foo) { # BETTER
```

不要在 `while` 的位置使用它们：

```
for (my $i = <STDIN>; $i; $i = <STDIN>) { # BAD

while (my $i = <STDIN>) { # BETTER
```

想想你编写的代码，并找找感觉。

18.2 匿名哈希和数组

创建一个匿名数组引用，并给它赋值：

```
my $array = [ 'one', 'two', 'three' ];
```

匿名是因为我们不必创建数组。

哈希有相似的构造器：

```
my $hash = { one => 1, two => 2, three => 3 };
```

看作你应认为的而非引用。

18.3 q[qrwx]?//、m//、s/// 及 y///

Perl 让你自行指定定界符：

- 单引号：'text' => q/text/
- 双引号："text" => qq/text/
- 正则表达式：qr/text/。除此之外，在 Perl 匹配及替换操作符外没有别的方式指定正则表达式匹配。
- 单词：("text", "text") => qw(text text);
- 反引号：`text` => qx/text/
- 正则匹配（m//）、正则替换（s///）、及转换 (tr///、y///) 工作方式相同

你可以使用除空白之外的任意字符。但要注意平衡括号或花括号：

```
qq//
qq#A decent <html> delimiter </html> #
qq( man perl(1) for details ) # valid!
```

18.4 global、local、my 及 our

- 使用 use vars 声明全局变量
- 使用 my 声明词法变量
- local 并非你所认为的，除非你知道为何使用 local，否则使用 my 代替
- 仅当你的包需要全局变量时使用 our

19 引用

19.1 引用是引用其他变量的标量

引用像 C 中引用其他变量的指针。使用 \ 操作符创建引用。

```
my $sref = \$scalar;
my $aref = \@array;
my $href = \%hash;
```

```
my $cref = \&subroutine;
```

引用指向的事物即其所指。

使用合适的印记解引用，首选使用花括号。

```
my $other_scalar = ${$sref};
my @other_array  = @{$aref};
my %other_hash   = %{$href};
&{$cref} # Call the referent.
```

19.2 用箭头符解引用更容易

要访问数组和哈希引用，使用 -> 操作符。

```
my $stooge = $aref->[1];
my $stooge = $href->{Curly};
```

19.3 ref vs. isa

- 一个引用属于一个类
- 你可以使用 ref 查检类
- 一个对象引用能从其他类继承
- 你可以使用 isa 来询问一个对象是否继承自一个类
- 没有好理由不要用 ref
- isa 是 *UNIVERSAL* 包的一部分，因此你可以在对象上调用它

```
my $mech = WWW::Mechanize->new;
print "ok\n" if $mech->isa('LWP::UserAgent');
```

19.4 引用匿名子例程

子例程能被赋给变量，并被调用，以允许代码引用被传递及使用。这将十分有用，比如编写需要执行所提供代码的子例程。

```
my $casefix = sub { return ucfirst lc $_[0] };

my $color = $casefix->("rED");
print "Color: $color\n"; # prints Red
```

20 对象、模块及包

20.1 包基础

- 包是方法的集合
- 具有自己的命名空间
- 包方法能被导出或直接调用

```
Foo::bar()
Foo->bar()
bar() (如果 Foo 已导出它)
```

20.2 模块基础

- 模块是包含一个或多个包的文件
- 多数人交替使用模块和包

20.3 对象基础

- 对象是被 bless 的哈希引用（不必是哈希引用，但它最常见）
- bless 将单个类赋给对象
- 对象可被重新 bless

20.4 1;

- 模块必须以真值结束
- 不必是 1
- 包没有相同的限制

20.5 @ISA

Perl 的对象继承方法使用 `@ISA` 来决定模块继承自什么类。多年前，通过直接修改 `@ISA` 声明继承。现在，多数程序使用 `base` 编译指令声明继承。

下列代码是等效的：

```
package Foo;
require Wango;
@ISA = ( "Wango" );

package Foo;
use base "Wango";
```

21 特殊变量

21.1 $_

$_ 是默认变量。它常用于内置函数的默认参数。

```
while ( <> ) {  # Read a line into $_
    print lc;   # print lc($_)
}
```

这与下列代码相同：

```
while ( $it = <> ) {
    print lc($it);
}
```

21.2 $0

$0 包含执行程序的名称，正如给 Shell 的一样。如果程序直接通过 Perl 解释器执行，那么 $0 包含文件名称。

```
$ cat file.pl
#!/usr/bin/perl
print $0, "\n";
$ ./file.pl
file.pl
$ perl file.pl
file.pl
$ perl ./file.pl
./file.pl
$ cat file.pl | perl
-
```

$0 是 C 程序员期望从 argv 数组找到的第一个元素。

21.3 @ARGV

`@ARGV` 包含给程序的参数，顺序与 Shell 中一样。

```
$ perl -e 'print join( ", ", @ARGV), "\n"' 1 2 3
1, 2, 3
$ perl -e 'print join( ", ", @ARGV), "\n"' 1 "2 3" 4
1, 2 3, 4
```

C 程序员可能会搞混，因为 `$ARGV[0]` 是他们的 `argv[1]`。不要犯这样的错。

21.4 @INC

`@INC` 包含 Perl 搜索模块的所有路径。

Perl 程序员通过后置或前置到 `@INC` 添加库路径。眼下，使用 `use lib` 代替。下面的代码等效：

```
BEGIN { unshift @INC, "local/lib" };

use lib "local/lib";
```

21.5 %ENV

`%ENV` 包含当前环境的拷贝。该环境由 Perl 创建的子 Shell 所给予。

这对 `taint` 模式很重要，`%ENV` 具有能修改 Shell 行为的内容。正因如此，perlsec 推荐在 `taint` 模式执行命令时使用下列代码：

```
$ENV{'PATH'} = '/bin:/usr/bin'; # change to your real path
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
```

21.6 %SIG

Perl 具有丰富的信号处理能力。使用 `%SIG` 变量，你能够在当信号发送给运行的进程时执行任意子例程。

如果你有耗时进程，这将特别有用。通过发送信号（通常是 `SIGHUP`）来重载配置，你不必启动和停止进程。

通过分别赋值 `$SIG{__DIE__}` 和 `$SIG{__WARN__}`，你也可以更改 `die` 和 `warn` 的行为。

21.7 <>

钻石操作符 <> 用于程序期望的输入时，而不用关心它如何到达。

如果程序收到任何参数，它们将分成文件名及其内容发送给 <>。否则，使用标准输入（STDIN）。

<> 对于过滤程序特别有用。

21.8 <DATA> 和 ___DATA___

如果程序包含自身为一行的魔法标记 __DATA__，那么它下面的任何东东均可通过魔法 <DATA> 句柄为程序所用。

如果你想在程序中包含数据，但又想与主程序逻辑分开，那么这将特别有用。

21.9 $!

当使用 system 执行命令时，如果命令返回非真状态，那么 $! 将为真。否则，可能未被执行。$! 将包含出错消息。

21.10 $@

如果使用 eval，那么 $@ 将包含 eval 所抛出的语法错误。

22 命令行选项

22.1 Shebang 行

几乎每个 Perl 程序都如此开始：

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

这是 UNIX 结构，它告诉 Shell 直接执行余下的输入程序文件。

你可以在此行添加 Perl 的任何命令行选项，它们将成为选项之后命令行的一部分。如果你有一个程序包含：

```
#!/usr/bin/perl -T
```

然后执行：

```
perl -l program.pl
```

-l 和 -T 两个选项都会使用，但 -l 将先用。在 *perlrun* 文档中介绍了 Perl 的命令行选项。此处只介绍最有用的内容。

22.2 perl -T

Perl 允许你在 `taint` 模式执行。在此模式中，变量在使用前需要“消毒”，以应对不安全的操作。

何为不安全？

- 执行程序
- 写入文件
- 创建目录
- 基本上，修改系统的任何事情

如果你没有“去污”数据，那么这些操作将是程序中的严重错误。

如何去污？使用正则表达式匹配有效的值，然后将匹配赋给变量。

```
my ($ok_filename) = $filename =~ /^(\w+\.log)$/;
```

你应当达到程序 `taint` 安全的目的。

22.3 perl -c file.{pl,pm}

此命令行选项允许检查给定文件的语法错误。它也会执行 `BEGIN` 块中的任意代码，并检查程序中已使用的模块。

你应当使用 `-c` 在每次更改后检查代码的语法。

22.4 perl -e ‘code’

该选项允许你从命令行执行代码，以代替将程序写入文件来执行。

```
$ perl -e 'print "1\n"'
1
```

这对小程序、快速计算、以及与其他选项组合使用非常有用。

22.5 -n、-p、-i

Perl 的 `-n` 选项允许你针对标准输入的每行重复执行代码（通常使用 `-e` 指定）。这些是等效的：

```
$ cat /etc/passwd | perl -e 'while (<>) { if (/^(\w+):/) { print "$1\n";
 } }'
root
...
$ cat /etc/passwd | perl -n -e 'if (/^(\w+):/) { print "$1\n" }'
root
...
```

`-p` 选项与 `-n` 相同，除了它在每行后打印 `$_`。

如果你组合 `-i` 选项，Perl 将就地编辑你的文件。因此，要将一堆文件从 DOS 转换成 UNIX 换行，你可以这样干：

```
$ perl -p -i -e 's/\r\n/\n/' file1 file2 file3
```

22.6 perl -M

Perl 的 `-M` 选项使你可以从命令行使用模块。有好些模块首选此方式运行（如 *CPAN* 和 *Devel::Cover*）。如果你需要使用 `-e` 包含模块，它也是习惯的简写。

```
$ perl -e 'use Data::Dumper; print Dumper( 1 );'
$VAR1 = 1;

$ perl -MData::Dumper -e 'print Dumper( 1 );'
$VAR1 = 1;
```

22.7 明白模块是否已被安装

试试从命令行加载模块。`-e1` 只是一个立即退出的空程序。如果你获得错误，那么该模块未被安装：

```
$ perl -MWWW::Mechanize::JavaScript -e 1
Can't locate WWW/Mechanize/JavaScript.pm in @INC...
BEGIN failed--compilation aborted.
$
```

返回没有错误则意味着该模块已安装。

```
$ perl -MWWW::Mechanize -e 1
$
```

当它存在时，检查版本：

```
$ perl -MWWW::Mechanize -e'print $WWW::Mechanize::VERSION'
```

并非所有模块都有 `$VERSION` 变量，因此这可能不总是工作。

23 高级函数

23.1 上下文与 wantarray

Perl 有三种上下文：空、标量、以及列表。

```
          func(); # void
my $ret   = func(); # scalar
my ($ret) = func(); # list
my @ret   = func(); # list
```

如果你在子例程或 `eval` 块中，你能够使用 `wantarray` 来决定想要的上下文。

以下是处理正则表达式返回值的上下文例子：

```
my $str = 'Perl 101 Perl Context Demo';
my @ret = $str =~ /Perl/g; # @ret = ('Perl','Perl');
my $ret = $str =~ /Perl/g; # $ret is true
```

23.2 .. 和 ...

这些叫区间操作符，它们能够帮助代码处理整数或字符区间。

在下例中，`@array` 是手动填充的。这些是等价的：

```
my @array = ( 0, 1, 2, 3, 4, 5 );
my @array = 0..5;
```

当用于此种方式时，`..` 和 `...` 是等效的。

区间操作符只能增加。这会产生一个空列表：

```
my @array = 10..1; # @array is empty
```

如果你想要逆向，要求它。

```
my @array = reverse 1..10; # @array descends from 10 to 1
```

你也可以在标量上下文中使用区间操作符，但那超出了本节的范围。参阅手册页了解细节。

24 风格

24.1 camelCase 很糟

你曾维护过别人的代码吗？你维护过像这样的代码吗？

```
my $variableThatContainsData =
    someSubroutineThatMucksWithData( $someAwfulVariable );
```

混合大小写单词在 Perl 世界被称为 *camelCase*，通常它的令人不悦之处是使阅读代码更难。

甚至具有糟糕名称的代码使用下划线也能变得更可读：

```
my $variable_that_contains_data =
    some_subroutine_that_mucks_with_data( $some_awful_variable );
```

24.2 warnings 和 strict

对于你希望维护、重用、及发布的任何程序，都应当具有下列代码行：

```
#!/usr/bin/perl

use strict;
use warnings;
```

启用 `strict` 使 Perl 抱怨不确定的代码结构，比如：未声明的变量、裸字、以及软引用等。这些警告将导致 Perl 执行失败。

```
#!/usr/bin/perl
use strict;

$foo     = 4;      # undeclared variable error
$foo     = Bar;    # bareword error
my $bat = "foo";
print $$bat;       # reference error
```

启用 `warnings` 使 Perl 甚至抱怨更多东东。但不像 `strict`，这些抱怨在一般条件下并不严重。

```
#!/usr/bin/perl
use warnings;
$a + 0;                          # void context warning
                                 # name used once warning
                                 # undef warning
print "program continued\n"; # prints
```

如果你想要 `warnings` 变得严重，告诉它：

```
use warnings FATAL => 'all';
$a + 0;                          # void warning and then exits
print "program continued\n"; # doesn't print
```

24.3 使用 perltidy 格式化 Perl 源代码

选择何种代码风格是仁者见仁，智者见智的事情。但重要的是保持风格的一致性。为了使格式化 Perl 源代码更容易，你可以使用 Perl::Tidy 模块随付的 `perltidy` 工具。

例如，使用 *Perl 最佳实践*一书所推荐的风格来格式化源代码：

```
$ perltidy -pbp myprogram.pl -o myprogram.formated.pl
```

25 性能

Perl 让你干想干的事，包括很慢或内存消耗这样的事。此处将告诉你如何避免。

25.1 使用 while 而非 for 来迭代整个文件

代替读取文件的所有行并使用 for 处理数组，使用 while 一次仅读取一行。

这两个循环的功能相同：

```
for ( <> ) {
    # do something
}

while ( <> ) {
    # do something
}
```

差异是 for 将整个文件读到临时数组，而 while 一次读一行。对于小文件，这可能不重要。但对于更大的文件，它可能吃掉你的所有可用内存。

在这种情况下使用 while 是一种好实践。

25.2 避免不必要的引起和字串化

除非绝对必要，不要引起大字符串：

```
my $copy = "$large_string";
```

这将创建两个 $large_string 的拷贝（一个是 $copy，而另一个是引起）。然而：

```
my $copy = $large_string;
```

仅创建一个拷贝。

对于字串化大的数组也是一样：

```
{
    local $, = "\n";
    print @big_array;
}
```

这比下面的代码更内存高效：

```
print join "\n", @big_array;
```

或：

```
{
    local $" = "\n";
    print "@big_array";
}
```

26 陷阱

26.1 while (<STDIN>)

一定要小心这点。如果你不知怎么回事地得到了假值（如：空行），你的文件可能停止处理了。假如你在处理文件读取（除非修改了 $/），这种事一般不会发生，但却可能发生。

你更喜欢这样运行：

```
while (readdir(DIR)) {
```

假设你有文件名为 0 的话，那么程序将停止，且不会继续处理文件。

更合适的 while 循环看起来像这样：

```
while ( defined( my $line = <STDIN> ) ) {
while ( defined( my $file = readdir(DIR) ) ) {
```

27 如何做...？

27.1 如何验证 Email 地址是否有效

一般来说，你不能。有一些看起来合理的方法可以使用，但却没有办法检测地址是否实际可以投递，如果没有实际尝试投递的话。

使用正则表达式：

```
# Match basically blah@blah.blah
if ( $addr =~ /^\S+\@\S+\.\S+$/ ) {
    print "Looks OK";
}
```

如果你干真活的话，可能希望看看 CPAN 上可用的模块，比如：*Email::Address*、*Email::Valid*。

27.2 如何从数据库获得数据

DBI 及其 *DBD* 子模块，如 *DBD::SQLite*。

27.3 如何从网页获得数据

LWP 意为 *libwww-perl*，它是与网页交谈的标准方式。

WWW::Mechanize 是使 HTML 处理更容易的 *LWP* 的超集。

27.4 如何做日期计算

使用 *Date::Manip*、*Date::Calc*、或 *DateTime*。全部都有不同的样式和不同的能力。

27.5 如何处理程序的命令行参数

使用 *Getopt::Long*。

27.6 如何解析 HTML

无论你做什么，都不要使用正则表达式。使用 *HTML::Parse* 或别的类似东东。如果你解析 HTML 是为了从网页提取链接或图像，不妨使用 *WWW::Mechanize*。

27.7 如何来点颜色

使用 *Term::ANSIColor*。

27.8 如何读取键及不看到输入的密码

使用 *Term::ReadKey*。

28 开发工具

28.1 剖析性能

使用 *Devel::NYTProf*，或 *Devel::DProf*。

28.2 分析代码质量

使用 *Perl::Critic*，它基本上是针对 Perl 的 lint。

28.3 分析变量结构

使用 *Data::Dumper*。

29 出版物

29.1 Perl 语言编程，第四版

称为大骆驼书，真正的 Perl 圣经。

29.2 Perl 语言入门，第六版

称为小骆驼书，Perl 标准教程。

29.3 Perl Cookbook，第二版

一本如何利用 Perl 干活的问题导向书籍。

29.4 Object Oriented Perl

关于 Perl 5 面向对象编程的最佳解释。

29.5 Perl 最佳实践

展示如何写优良、可维护的 Perl 代码。

29.6 The Perl Review

在芝加哥出版，致力于 Perl 的杂志。

http://theperlreview.com

30 社区

30.1 Perl 基金会

Perl 基金会致力于推进 Perl 编程语言的开放讨论、协作、设计及编码。Perl 基金会是非盈利、501(c)(3) 组织。

http://perlfoundation.org http://news.perlfoundation.org

30.2 Perl Mongers

世界各地的 Perl 用户组。

http://www.pm.org

30.3 OSCON

奥莱理的开源大会。起初为 Perl 大会，后被扩展。

http://conferences.oreilly.com

30.4 YAPC

又一个 Perl 大会。便宜、TPF 驱动以替代 OSCON。

http://www.yapc.org

30.5 IRC 频道

在线聊天频道。

Freenode (irc.freenode.net) 至少不让人讨厌。

MAGnet (irc.perl.org) 有最铁杆的 Perl 家伙，但最令人讨厌。

30.6 邮件列表

在 http://lists.perl.org 有一打邮件列表。

30.7 PerlMonks

一半是论坛，一半是问答。主要对新手有用。

http://perlmonks.org

30.8 Blog

Perl 用户 Blog。

http://blogs.perl.org