

PERL 编程 24

学时教程

目 录

第一部分 Perl 基础

第1学时 Perl入门3

1.1 安装Perl	3
1.1.1 等一等,也许你已经安装了Perl	4
1.1.2 在Windows 95/98/NT上安装Perl	5
1.1.3 在UNIX上安装Perl	6
1.1.4 在Macintosh系统上安装Perl	7
1.2 文档资料	7
1.2.1 某些特殊的文档资料举例	8
1.2.2 如果无法找到文档该怎么办	9
1.3 编写你的第一个Perl程序	9
1.3.1 键入程序	9
1.3.2 运行程序	9
1.3.3 程序正确将会发生什么情况	10
1.3.4 Perl程序的具体运行过程	10
1.3.5 必须知道的一些情况	11
1.4 课时小结	12
1.5 课外作业	12
1.5.1 专家答疑	12
1.5.2 思考题	12
1.5.3 解答	13
1.5.4 实习	13

第 2 学时 Perl 的基本构件：数字和字符串 14

2.1 直接量	14
2.1.1 数字	14
2.1.2 字符串	15
2.2 标量变量	16
2.3 表达式和运算符	18
2.3.1 基本运算符	18
2.3.2 数字运算符	19
2.3.3 字符串运算符	19
2.4 其他运算符	20
2.4.1 单参数运算符	20
2.4.2 递增和递减	21
2.4.3 尖括号运算符	21
2.4.4 其他赋值运算符	22
2.4.5 关于字符串和数字的一些说明	22
2.5 练习：利息计算程序	23
2.6 课时小结	24
2.7 课外作业	24
2.7.1 专家答疑	24
2.7.2 思考题	24
2.7.3 解答	25
2.7.4 实习	25

第 3 学时 控制程序流 26

3.1 语句块	26
3.2 if 语句	27
3.2.1 其他关系运算符	28
3.2.2 “真”对于 Perl 意味着什么	29
3.2.3 逻辑运算符	30
3.3 循环	32
3.3.1 用 while 进行循环	32
3.3.2 使用 for 循环	33
3.4 其他流控制工具	33
3.4.1 奇特的执行顺序	33
3.4.2 明细控制	34
3.4.3 标号	35
3.4.4 退出 Perl	35

3.5	练习：查找质数	35
3.6	课时小结	37
3.7	课外作业	37
3.7.1	专家答疑	37
3.7.2	思考题	37
3.7.3	解答	38
3.7.4	实习	38

第4学时 基本构件的堆栈：列表与数组 39

4.1	将数据放入列表和数组	39
4.2	从数组中取出元素	41
4.2.1	寻找结尾	42
4.2.2	关于上下文的详细说明	43
4.2.3	回顾以前的几个功能	44
4.3	对数组进行操作	45
4.3.1	遍历数组	46
4.3.2	在数组与标量之间进行转换	46
4.3.3	给数组重新排序	48
4.4	练习：做一个小游戏	49
4.5	课时小结	51
4.6	课外作业	51
4.6.1	专家答疑	51
4.6.2	思考题	51
4.6.3	解答	52
4.6.4	实习	52

第5学时 进行文件操作 53

5.1	打开文件	53
5.1.1	路径名	54
5.1.2	出色的防错措施	55
5.1.3	以适当的方式运行 die 函数	56
5.2	读取文件	56
5.3	写入文件	58
5.4	自由文件、测试文件和二进制数据	60
5.4.1	自由文件句柄	60
5.4.2	二进制文件	60
5.4.3	文件测试运算符	61
5.5	课时小结	62

5.6	课外作业	62
5.6.1	专家答疑	62
5.6.2	思考题	63
5.6.3	解答	63
5.6.4	实习	63

第 6 学时 模式匹配 64

6.1	简单的模式	64
6.2	元字符	66
6.2.1	一个简单的元字符	66
6.2.2	非输出字符	66
6.2.3	通配符	66
6.2.4	字符类	68
6.2.5	分组和选择	69
6.2.6	位置通配符	69
6.3	替换	70
6.4	练习：清除输入数据	70
6.5	关于模式匹配的其他问题	71
6.5.1	对其他变量进行操作	71
6.5.2	修饰符与多次匹配	72
6.5.3	反向引用	73
6.5.4	一个新函数：grep	73
6.6	课时小结	74
6.7	课外作业	74
6.7.1	专家答疑	74
6.7.2	思考题	75
6.7.3	解答	75
6.7.4	实习	75

第 7 学时 哈希结构 77

7.1	将数据填入哈希结构	77
7.2	从哈希结构中取出数据	78
7.3	列表与哈希结构	80
7.4	关于哈希结构的补充说明	81
7.4.1	测试哈希结构中的关键字	81
7.4.2	从哈希结构中删除关键字	81
7.5	用哈希结构进行的有用操作	81
7.5.1	确定频率分布	82

7.5.2 在数组中寻找惟一的元素	82
7.5.3 寻找两个数组之间的交汇部分和不同部分	83
7.5.4 对哈希结构进行排序	84
7.6 练习：用 Perl 创建一个简单的客户数据库	84
7.7 课时小结	86
7.8 课外作业	86
7.8.1 专家答疑	86
7.8.2 思考题	87
7.8.3 解答	87
7.8.4 实习	88

第 8 学时 函数 89

8.1 创建和调用子例程	89
8.1.1 返回子例程的值	90
8.1.2 参数	91
8.1.3 传递数组和哈希结构	91
8.2 作用域	92
8.3 练习：统计数字	94
8.4 函数的脚注	96
8.4.1 声明 local 变量	96
8.4.2 使 Perl 变得更加严格	97
8.4.3 递归函数	98
8.5 课时小结	99
8.6 课外作业	99
8.6.1 专家答疑	99
8.6.2 思考题	99
8.6.3 解答	100
8.6.4 实习	100

第二部分 高级特性

第 9 学时 其他函数和运算符 103

9.1 搜索标量	103
9.1.1 用 index 进行搜索	103
9.1.2 用 rindex 向后搜索	104
9.1.3 用 substr 分割标量	104

9.2	转换而不是替换	105
9.3	功能更强的 print 函数	106
9.4	练习：格式化报表	107
9.5	堆栈形式的列表	109
9.6	课时小结	110
9.7	课外作业	111
9.7.1	专家答疑	111
9.7.2	思考题	111
9.7.3	解答	112
9.7.4	实习	112

第 10 学时 文件与目录 113

10.1	获得目录列表	113
10.2	练习：UNIX 的 grep	116
10.3	目录	117
10.3.1	浏览目录	117
10.3.2	创建和删除目录	118
10.3.3	删除文件	119
10.3.4	给文件改名	119
10.4	UNIX 系统	120
10.5	你应该了解的关于文件的所有信息	121
10.6	练习：对整个文件改名	122
10.7	课时小结	123
10.8	课外作业	124
10.8.1	专家答疑	124
10.8.2	思考题	124
10.8.3	解答	124
10.8.4	实习	125

第 11 学时 系统之间的互操作性 126

11.1	system() 函数	126
11.2	捕获输出	128
11.3	管道	129
11.4	可移植性入门	131
11.5	课时小结	134
11.6	课外作业	134
11.6.1	专家答疑	134
11.6.2	思考题	135

11. 6. 3	解答	135
11. 6. 4	实习	136

第 12 学时 使用 Perl 的命令行工具 137

12. 1	什么是调试程序	137
12. 1. 1	启动调试程序	137
12. 1. 2	调试程序的基本命令	138
12. 1. 3	断点	139
12. 1. 4	其他调试程序命令	140
12. 2	练习：查找错误	141
12. 3	其他命令行特性	142
12. 3. 1	单命令行程序	142
12. 3. 2	其他开关	143
12. 3. 3	空的尖括号与更多的单命令行程序	144
12. 4	课时小结	145
12. 5	课外作业	145
12. 5. 1	专家答疑	145
12. 5. 2	思考题	146
12. 5. 3	解答	146

第 13 学时 引用与结构 147

13. 1	引用的基本概念	147
13. 1. 1	对数组的引用	149
13. 1. 2	对哈希结构的引用	149
13. 1. 3	作为参数的引用	150
13. 1. 4	创建各种结构	151
13. 2	结构的配置方法	152
13. 2. 1	一个例子：列表中的列表	152
13. 2. 2	其他结构	153
13. 2. 3	使用引用来调试程序	154
13. 3	练习：另一个游戏——迷宫	155
13. 4	课时小结	157
13. 5	课外作业	157
13. 5. 1	专家答疑	157
13. 5. 2	思考题	158
13. 5. 3	解答	158
13. 5. 4	实习	158

第 14 学时 使用模块 159

14.1 模块的概述	159
14.1.1 读取关于模块的文档	160
14.1.2 什么地方可能出错	161
14.2 已安装模块简介	162
14.2.1 文件和目录简介	162
14.2.2 拷贝文件	164
14.2.3 用于通信的 Perl 模块	164
14.2.4 使用 English 模块	165
14.2.5 diagnostics 模块	165
14.3 标准模块的完整列表	166
14.4 课时小结	167
14.5 课外作业	167
14.5.1 专家答疑	167
14.5.2 思考题	168
14.5.3 解答	168
14.5.4 实习	168

第 15 学时 了解程序的运行性能 169

15.1 DBM 文件	169
15.1.1 需要了解的重点	170
15.1.2 遍历与 DBM 文件相连接的哈希结构	170
15.2 练习：一种自由格式备忘录事板	171
15.3 将文本文件用作数据库	173
15.4 随机访问文件	175
15.4.1 打开文件进行读写操作	175
15.4.2 在读写文件中移动	176
15.5 锁定文件	176
15.5.1 锁定 UNIX 和 NT 下的文件	178
15.5.2 在加锁情况下进行读写操作	179
15.5.3 Windows 95 和 Windows 98 下的加锁问题	180
15.5.4 在其他地方使用文件锁的问题	181
15.6 课时小结	181
15.7 课外作业	181
15.7.1 专家答疑	181
15.7.2 思考题	182
15.7.3 解答	182

15.7.4 实习 182

第 16 学时 Perl 语言开发界 183

- 16.1 Perl 究竟是一种什么语言 183
 - 16.1.1 Perl 的简单发展历史 183
 - 16.1.2 开放源 184
 - 16.1.3 Perl 的开发 185
- 16.2 Perl 综合存档文件网 185
 - 16.2.1 什么是 CPAN 186
 - 16.2.2 为什么人们愿意提供自己的开发成果 186
- 16.3 下一步你要做的工作 187
 - 16.3.1 要做的第一步工作 187
 - 16.3.2 最有用的工具 187
 - 16.3.3 查找程序中的错误 188
 - 16.3.4 首先要靠自己来解决问题 188
 - 16.3.5 从别人的程序错误中吸取教训 189
 - 16.3.6 请求他人的帮助 190
- 16.4 其他资源 191
- 16.5 课时小结 192
- 16.6 课外作业 192
 - 16.6.1 专家答疑 192
 - 16.6.2 思考题 192
 - 16.6.3 解答 192

第三部分 将 Perl 用于 CGI

第 17 学时 CGI 概述 195

- 17.1 浏览 Web 195
 - 17.1.1 检索一个静态 Web 页 196
 - 17.1.2 动态 Web 页—使用 CGI 197
- 17.2 不要跳过这一节内容 198
- 17.3 编写你的第一个 CGI 程序 199
 - 17.3.1 在服务器上安装 CGI 程序 200
 - 17.3.2 运行你的 CGI 程序 201
- 17.4 CGI 程序无法运行时怎么办 201
 - 17.4.1 这是你的 CGI 程序吗 201

17.4.2	服务器存在的问题	202
17.4.3	排除服务器内部错误或 500 错误	203
17.5	课时小结	204
17.6	课外作业	204
17.6.1	专家答疑	204
17.6.2	思考题	205
17.6.3	解答	205
17.6.4	实习	206

第 18 学时 基本窗体 207

18.1	窗体是如何运行的	207
18.1.1	HTML 窗体元素概述	207
18.1.2	单击 submit 时出现的情况	208
18.2	将信息传递给你的 CGI 程序	209
18.3	Web 安全性	211
18.3.1	建立传输明码文本的连接	211
18.3.2	注意不安全数据	212
18.3.3	从事无法执行的操作	213
18.3.4	拒绝服务	213
18.4	宾客留言簿	214
18.5	课时小结	215
18.6	课外作业	215
18.6.1	专家答疑	215
18.6.2	思考题	216
18.6.3	解答	216
18.6.4	实习	216

第 19 学时 复杂窗体 217

19.1	复杂的多页窗体	217
19.2	隐藏域	217
19.3	多页调查窗体	219
19.4	课时小结	224
19.5	课外作业	224
19.5.1	专家答疑	224
19.5.2	思考题	225
19.5.3	解答	225
19.5.4	实习	225

第 20 学时 对 HTTP 和 CGI 进行操作 226

- 20.1 HTTP 通信概述 226
 - 20.1.1 举例：人工检索 Web 页 227
 - 20.1.2 举例：返回非文本信息 228
- 20.2 如何调用 CGI 程序的详细说明 230
 - 20.2.1 将参数传递给 CGI 程序 230
 - 20.2.2 特殊参数 231
- 20.3 服务器端的包含程序 232
- 20.4 部分环境函数简介 234
- 20.5 重定向 235
- 20.6 课时小结 237
- 20.7 课外作业 237
 - 20.7.1 专家答疑 237
 - 20.7.2 思考题 237
 - 20.7.3 解答 238

20.7.4 实习 238

第 21 学时 cookie 239

- 21.1 什么是 cookie 239
 - 21.1.1 如何创建 cookie 240
 - 21.1.2 举例：使用 cookie 241
 - 21.1.3 另一个例子：cookie 查看器 242
- 21.2 高级 cookie 特性 243
 - 21.2.1 设置 cookie 终止运行的时间 243
 - 21.2.2 cookie 的局限性 244
 - 21.2.3 将 cookie 发送到其他地方 244
 - 21.2.4 限制 cookie 返回到的位置 246
 - 21.2.5 带有安全性的 cookie 247
- 21.3 cookie 存在的问题 247
 - 21.3.1 cookie 的生存期很短 247
 - 21.3.2 并非所有浏览器都支持 cookie 247
 - 21.3.3 有些人不喜欢 cookie 247
- 21.4 课时小结 248
- 21.5 课外作业 248
 - 21.5.1 专家答疑 248
 - 21.5.2 思考题 249
 - 21.5.3 解答 250

21.5.4 实习 250

第 22 学时 使用 CGI 程序发送电子邮件 251

- 22.1 Internet 邮件入门 251
 - 22.1.1 发送电子邮件 252
 - 22.1.2 发送邮件时首先应该注意的问题 252
- 22.2 邮件发送函数 253
 - 22.2.1 用于 UNIX 系统的邮件函数 254
 - 22.2.2 用于非 UNIX 系统的邮件函数 255
- 22.3 从 Web 页发送邮件 257
- 22.4 课时小结 259
- 22.5 课外作业 259
 - 22.5.1 专家答疑 259
 - 22.5.2 思考题 260
 - 22.5.3 解答 260
 - 22.5.4 实习 260

第 23 学时 服务器推送和访问次数计数器 261

- 23.1 什么是服务器推送 261
 - 23.1.1 激活服务器推送特性 262
 - 23.1.2 一个小例子：更新 Web 页上的时钟 262
 - 23.1.3 另一个例子：动画 263
 - 23.1.4 客户机拖拉技术 264
- 23.2 访问次数计数器 264
 - 23.2.1 编写一个访问次数计数器程序 266
 - 23.2.2 图形访问次数计数器 267
- 23.3 课时小结 268
- 23.4 课外作业 269
 - 23.4.1 专家答疑 269
 - 23.4.2 思考题 269
 - 23.4.3 解答 269
 - 23.4.4 实习 270

第 24 学时 建立交互式 Web 站点 271

- 24.1 借用另一个站点的内容 271
 - 24.1.1 注意内容的版权问题 271
 - 24.1.2 举例：检索标题 272
- 24.2 调查窗体 275

24.2.1	调查窗体程序的第一部分：提出问题	276
24.2.2	调查窗体程序的第二部分：计算调查结果	277
24.3	课时小结	280
24.4	课外作业	280
24.4.1	专家答疑	280
24.4.2	思考题	281
24.4.3	解答	281
24.4.4	实习	281

第四部分 附录

附录 安装模块 285

China-pub.com

下载

第一部分

Perl 基础

第1学时 Perl 入门

第2学时 Perl 的基本构件：数字和字符串

第3学时 控制程序流

第4学时 基本构件的堆栈：列表与数组

第5学时 进行文件操作

第6学时 模式匹配

第7学时 哈希结构

第8学时 函数

China-pub.com

下载

第1学时 Perl入门

Perl是一种通用编程语言。凡是其他编程语言能够使用的地方，都有它的用武之地。在各行各业中，它已经被用于你能够想像到的各种各样的任务的处理。它已经用于股票市场、产品制造、产品设计、客户支持、质量控制、千年虫测试、系统编程、工资处理和库存管理等各个领域，当然还有Web。

Perl的用途之所以如此广泛，原因是 Perl被称为是一种“胶水语言”。所谓胶水语言，也就是说它是可以用来将许多元素连接在一起的语言。你可能不想用 Perl来编写一个文字处理程序（尽管可以编写这样的程序），因为现在已经有许多非常出色的文字处理程序了。用 Perl来编写数据库、电子表格、操作系统或者特性完善的 Web服务程序也不是聪明之举，不过编写这些程序是完全可以做到的。

Perl真正擅长的是将这些程序连接在一起。Perl能够利用你的数据库，将它转换成一个具有电子表格特性的文件，并且在你进行文件的处理时，根据需要对数据进行调整。Perl也能够利用文字处理文档，将它们转换成HTML文档，以便在Web上显示。

由于Perl是一种“胶水语言”，能够将许多元素连接在一起，因此它具有极强的适应性。它至少能够在二十几种操作系统下运行，甚至能够在更多的操作系统下运行。Perl的编程样式非常灵活，因此可以用许多不同的方法来做同一件事情。你编写的 Perl程序看上去可能与我的程序毫无共同之处，但是如果它们运行起来，却没有任何问题。必要的时候，Perl可以是一种非常严谨的语言，而对于编程新手来说，如果你愿意，它又可以是一种非常随意的语言，这完全可以根据你的需要来定。

下面让我们来澄清一些基本概念。这个编程语言的名字是 Perl。运行你的程序的程序（即解释程序）的名字是 perl。对你来说它们之间的差别通常并不十分重要，不过当你试图启动你的程序时，情况就不一样了，那时它的名字总是 perl。有时，你会看到 Perl被写成了PERL，这可能是因为Perl的名字是Practical Extraction and Report Language缩略而来的。现在已经没有人再说PERL，因为这个名字显得太一本正经了。Perl这个名字则比较随便。



Perl的许多特性是从其他语言中借来的。这种借用曾经在早期导致 Perl成为另一个用语Pathologically Eclectic Rubbish Lister的缩写。

本学时介绍的内容包括：

- 安装Perl。
- 访问Perl的内部文档。
- 编写你的第一个Perl脚本程序。

1.1 安装Perl

若要使用Perl，首先必须安装Perl。Perl的安装是非常容易的，并且不会出错。实际上，作为安装步骤的一部分，Perl应该进行自我测试，以确保它安装成功。安装的操作过程可以有

很大的差别，这要根据你运行的操作系统而定。因此，为了使安装操作能够进行下去，首先要确定你运行的是什么操作系统，然后再进行下一步工作。

1.1.1 等一等，也许你已经安装了Perl

当你着手在你的系统上安装 Perl之前，应该检查一下是否已经安装了 Perl。有些UNIX的供应商已经为操作系统配备了 Perl。Windows NT也将Perl作为Windows NT的资源工具包(Resource Kit)的组成部分提供给客户。若要查看你的操作系统上是否已经安装了 Perl，你需要获得一个命令提示符。

在UNIX系统下，只需要登录到该系统中。如果你拥有一个图形操作环境，需要打开终端窗口。当已经登录或者打开终端窗口后，你会看到下面这个提示符：

\$

这个提示符也可能是%，也可能是bash%，无论什么提示符，它都称为 shell 提示符或命令提示符。在本书的头几个学时中，你将需要使用这个提示符与 Perl 进行交互操作。

若要了解你的操作系统上是否已经安装了 Perl，请键入下面这行命令(不要键入\$提示符)：

\$ perl -v

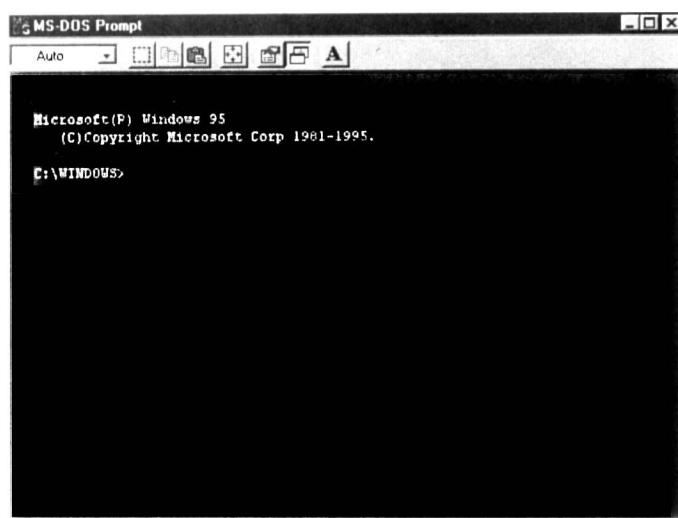
这时系统可能显示一个出错消息，如 command not found(命令没有找到)，也可能 Perl 作出响应，输出它的版本号。如果 Perl 输出它的版本号，那么就表示它已经安装好了，你就不需要重新进行安装。



报告的版本号至少应该是 5，也许是 5.004、5.005、5.6 等，不能小于这些数字。如果 Perl 的版本号是 4.x，那么你必须安装一个新拷贝。Perl 4 这个版本太老，错误很多，而且不再能够得到维护，本书中只有很少的示例程序能在 Perl 4 下运行。在撰写本书时，5.005 是 Perl 的当前版本，5.6 在 1999 年底推出。

如果你拥有一台运行 Windows 操作系统的计算机，要想查看是否安装了 Perl，必须显示图 1-1 所示的 MS-DOS 提示符。

图 1-1 可以在这个 DOS 提示符下查看 Perl 的版本



下载

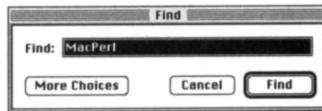
在这个提示符下，键入下面这个命令（不要重复键入提示符）：

```
C:\> perl -v
```

如果Perl已经安装，它就会显示版本号。正如上面的警告中所说的那样，它的版本号至少必须是5。如果DOS回答说Bad command or file name（命令或文件名不正确），那么你就应该安装Perl。

在Macintosh计算机上，你可以像图 1-2 所示的那样，运行 File Find 命令（Command-f），在 Find 的命令框中键入 ‘ MacPerl ’，来查看是否已经安装了 Perl。如果找到了该应用程序，那么将它打开，观察 Apple 菜单下面的 “ About MacPerl ” 选项。你至少应该拥有 Version 5.2.0 Patchlevel 5.004 这个版本号，否则就应该安装 MacPerl 的新版本。

图1-2 查看Macintosh计算机上的Perl



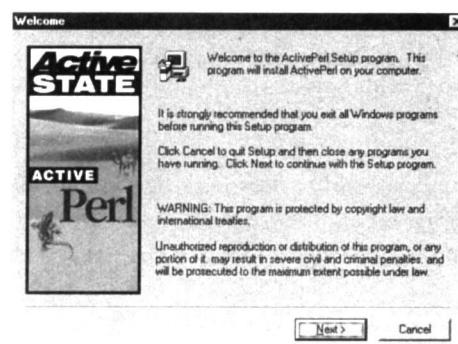
1.1.2 在Windows 95/98/NT上安装Perl

若要在 Windows 下安装 Perl，请记住，你可以像许多其他情况中那样，既可以使用捷径，也可以使用比较笨的办法。如果你对开发环境中需要的 C 编译器和各种工具（比如 Makefile、shell 等）非常熟悉，那么可以从头开始安装你自己的 Perl。可以随意查看、修改和改变 Perl 解释程序的源代码，使之适合你的需要。详细情况请参见第 16 学时（Perl 程序的开发界）的内容。想要在 Windows 下从头安装 Perl 并不容易，对于大多数人来说，这样做并不值得。

安装 Perl 时使用的捷径确实是非常容易的。 ActiveState Tool 公司提供了一个自行安装 Perl 的工具，安装工作就像安装其他任何 Windows 应用程序一样，如图 1-3 所示。这个 Perl 是在 ActiveState Community License （ ActiveState 团体许可证）下提供的，你应该阅读该许可证的有关说明。该公司的 URL 是 <http://www.ActiveState.com> 。

这个 Perl 与你自己建立的 Perl 是完全相同的东西。 ActiveState 公司只是为你做了最困难的那部分工作，并且用有关安装程序将它包装了起来。如果你需要的话， ActiveState 公司还为 Perl 提供了商业上的支持，并且提供了一些附加产品，如调试程序和其他开发工具及文件库。

图1-3 用ActiveState提供的工具在Windows下安装Perl



本书的光盘上包含了 ActiveState 公司的 Perl 产品的拷贝，你可以直接使用这个光盘来安装 Perl，也可以通过 ActiveState 公司的 Web 站点来获取最新版本的 Perl 。



1.1.3 在UNIX上安装Perl

若要在UNIX上安装Perl，需要具备两个条件。首先，需要一个Perl的源模块包的拷贝。你始终都可以从<http://www.perl.com>的Downloads区域下载它的最新版本。可以从那里找到多个版本，不过你需要的版本总是带有“Stable”或“Production”标号。还需要一个ANSI C编译器。如果不知道这个编译器的作用，也不必担心。Perl的配置程序能够选定一个，如果没有这个编译器，你可以安装一个预安装版本，这在本书的结尾处将要介绍。



如果你的UNIX配有一个用于安装预安装软件包的系统，你就能够安装一个预安装的Perl版本。Linux、Solaris、AIX和其他UNIX系统均配有已经捆绑的预安装Perl版本，它们的安装非常容易。请查看有关资料，以了解何处能够得到这些软件包。

当你拥有Perl的源模块包后（它的文件名类似 Stable.tar.gz），必须对它进行拆包，然后进行安装。若要进行操作，请输入下面的命令：

```
$ gunzip stable.tar.gz  
$ tar xf stable.tgz
```

这两个命令的运行需要花费一定的时间。如果没有 gunzip解压缩程序，可以从<http://www.gnu.org>下载一个拷贝。该程序包称为 gzip。当你完成所有的拆包操作后，就会看到一个提示符，然后键入下面的命令：

```
$ sh Configure
```

这时Configure程序就开始运行，并且问你一系列的问题。如果大部分问题你不知道如何回答，这没有关系，你只需要按 Enter键即可。默认的答案通常是最好的答案。Perl几乎能够在任何UNIX系统上安装而不会出现任何问题。当所有这些操作完成时，键入下面的命令：

```
$ make
```

Perl的安装需要花费相当长的时间，你可以乘此机会喝点儿咖啡。如果你的系统运行速度比较慢，你可以利用这个时间用午餐。当安装完成时，再键入下面这两个命令：

```
$ make test  
# make install
```

make test这个命令用于确保Perl的安装百分之百正确并且使之可以准备运行。若要运行make install命令，你必须以一个根用户的身份进行登录（正因为这个原因，所以使用提示符#，这是根用户的提示符），因为它需要将Perl安装到系统目录中去。

当make install运行正确时，你可以测试Perl的安装情况，方法是再次键入下面的命令：

```
$ perl -v
```

如果这个命令运行正确，那么祝贺你安装成功了！



在UNIX下安装Perl时使用的源模块拷贝位于本书所附的光盘上。可以直接从这里拷贝这个安装模块包，也可以从<http://www.perl.com>那里获得Perl的最新版本。

1.1.4 在Macintosh系统上安装Perl

Macintosh Perl的最新版本称为 MacPerl，可以从 CPAN端口目录下获得该版本。你必须访问<http://www.perl.com/CPAN/ports/mac>站点，从那里下载安装文件。你应该从该目录下下载 MacPerl appl.bin的最新版本，安装时，请使用 StuffIt Expander，从下载文件中取出 MacPerl安装程序，然后运行该安装程序。

当完成安装后，你可能想要为 Perl文档的阅读者安装一个帮助程序 Shuck，它是与 MacPerl一道安装的。 MacOS 8的用户可以通过 Internet控制面板进行安装，方法是打开 Advanced->File Mapping，给Shuck应用程序添加文件扩展名映像.pod。这样就可以更加容易地访问该文档了。还可以给MacPerl应用程序建立.ph、.pl、.plx、.pm、.cgi和_xs（这些都是Perl使用的扩展名）等文件映像，请务必将文件类型设置为‘ TEXT ’。

MacOS 7的用户必须使用 InternetConfig实用程序来进行类似的映射操作。在 InternetConfig中，选定Helpers，为pod添加新的帮助应用程序 shuck。另外，还要将前面提到的其他扩展名的帮助文件添加给 MacPerl应用程序。



本书所附的光盘上包含了一个 MacPerl的安装软件包的拷贝。你可以直接用光盘进行安装，也可以访问 <http://www.perl.com/CPAN/ports/mac>站点，从那里获得一个最新的拷贝。

1.2 文档资料

这个问题非常重要，因此你应该格外注意。每安装一个 Perl，你就会得到一份完整的 Perl语言和解释程序的当前文档资料的拷贝。

是的，安装软件包配用了Perl的整套文档资料，你可以免费获得这套资料。 Perl 5.005 版包含的资料超过 1700页。这些资料包括参考资料、培训资料、FAQ、历史资料，甚至是关于Perl内部情况的说明。

可以使用各种不同的方法来访问这些文档资料。在 Windows和UNIX系统上，与Perl一道安装了一个称为 perldoc的实用程序。你可以使用 perldoc程序来搜索这些文档资料，为手册提供格式化输出。若要运行 perldoc程序，你必须处在一个命令提示符下。下面这个例子使用的是UNIX提示符，不过在DOS命令提示符下也可以：

```
$ perldoc perl

PERL(1)          13/Oct/98 (perl 5.005, patch 02)          PERL(1)

NAME
    perl - Practical Extraction and Report Language

SYNOPSIS
    perl [ -sTuu ]      [ -hv ] [ -V[:configvar] ]
    [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
    [ -pna ] [ -Fpattern ] [ -l{octal} ] [ -0{octal} ]
    [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
    [ -P ]          [ -S ]      [ -x[dir] ]
    [ -i[extension] ]      [ -e 'command' ] [ --
    ] [ programfile ] [ argument ]...
```

For ease of access, the Perl manual has been split up into a number of sections:

手册的各个部分可以分成不同的节，其名字可以是 perlfunc (Perl函数)、perlop (Perl运算符) 和 perlfaq (Perl FAQ) 等。若要访问 perlfunc 手册页，可以输入命令 perldoc perlfunc。手册的所有部分的名字都在 perldoc perl 手册页中列出。

若要搜索手册，查找某个函数名，可以运行带有 -tf 开关的 perldoc 实用程序。下面这个例子用于查找 Perl 的 print 函数的手册页：

```
$ perldoc -tf print
```

FAQ 是指关于 Perl 的常见问题。这些问题是由学习 Perl 的人一再提出的问题。为了节省人们的时间，减少一些麻烦，这些问题集中起来放入称为 FAQ 的文件中。若要搜索 FAQ 文件，找出某个关键字，你应该使用 -q 开关，后随 FAQ 标题中可能出现的一个单词。例如，如果想要知道有关 Perl 的支持信息，可以使用下面这个命令：

```
$ perldoc -q support
```

这时，就会显示 FAQ 问题的条目“Who Supports Perl? Who develops it? Why is it free?”（谁支持 Perl？谁开发了 Perl？它为什么是免费的？）

1.2.1 某些特殊的文档资料举例

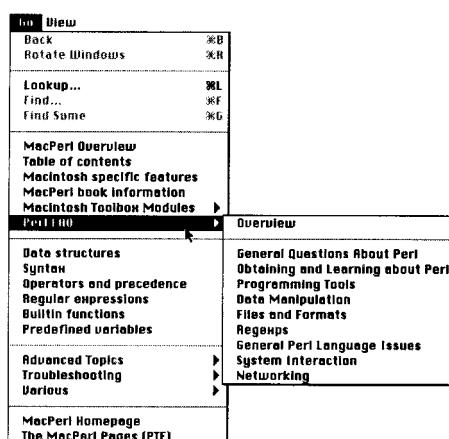
当在 UNIX 系统上安装 Perl 时，安装人员会看到一个选项提示，让他按传统的“man”格式来安装手册页。如果安装人员选择 yes (是)，那么标准 Perl 文档就转换成 man 格式，并且存放在一个相应的位置。若要访问 Perl 文档，既可以使用 perldoc 程序，也可以像通常在 UNIX 中那样使用 man 程序：

```
$ man perl
```

当在 Microsoft Windows 系统上安装 ActiveState 公司的 Perl 产品时，手册页被转换成 HTML 文档格式，并且可以用 Web 浏览器进行访问。如果想要阅读手册，请将你的 Web 浏览器指向本地目录 C:\Perl\html，假如你使用标准安装目录的话；如果你不使用标准目录，则使用你选定的目录。

如果是 Macintosh 系统，MacPerl 配有一个称为 Shuck 的实用程序，它位于 MacPerl 文件夹中。你可以用它来阅读和搜索 Perl 文档，如图 1-4 所示。

图 1-4 MacPerl 的 Shuck 文档阅读器



下载

1.2.2 如果无法找到文档该怎么办

如果无法找到你要的文档，那么可能在两个地方出现了错误。首先是没有查找正确的文档位置。也许 perldoc实用程序安装到的目录不在你的 shell搜索路径上，或者你的搜索路径具有很强的限制性。请反复搜寻 perldoc实用程序，将该目录添加到你的 shell的PATH环境变量中。

第二个原因是文档被删除了。也许是不小心删除的，也可能是恶意删除的。Perl的安装包括了文档的安装。你不能只安装 Perl而不安装文档。如果文档没有了，那么完全可以认为 Perl的安装不正确，或者安装以后遭到了破坏。也许你或者系统管理员应该考虑重新安装 Perl。文档是Perl开发环境的不可分割的组成部分，如果没有文档，那么 Perl的某些部分将无法运行。

如果其他方法都失败了，而你仍然无法得到文档的本地拷贝，那么你可以求助于 Web。在Perl的主要销售站点（<http://www.perl.com>）上，可以访问一组标准文档。拥有你的 Perl版本配备的实际文档更好，因为它是根据你的特定 Perl版本和安装情况裁剪的文档，不过这个在线文档只有在紧急情况下才能使用。

1.3 编写你的第一个Perl程序

若要编写你的Perl程序，需要一个称为文本编辑器的实用程序。使用文本编辑器，可以将不带任何格式的纯文本输入文件中。Microsoft Windows 的Notepad和DOS 的EDIT.EXE均属文本编辑器。在UNIX中，vi、emacs和pico等，都是文本编辑器。你的系统上至少存在其中的一个文本编辑器。在 Mac系统下，MacPerl应用程序包含一个基本的文本编辑器，若要打开一个新程序，请在File菜单下选定 New。

不应该使用文字处理程序来键入你的 Perl程序。文字处理程序，比如 Microsoft的Word、Wordpad和WordPerfect等，在保存文档时，会将格式化代码嵌入文档之中，即使文档并不包含任何格式。这些格式化代码会使 Perl感到莫名其妙，你的程序将无法正确运行。如果需要使用文字处理程序，那么务必将你的程序保存为纯文本文件。

1.3.1 键入程序

打开文本编辑器，正确无误地键入下面这个 Perl程序：

```
#!/usr/bin/perl  
  
print "Hello, World!\n";  
  
#! 这个程序行应该是文件中的第一行。
```

当将该程序键入你的文本编辑器后，将它保存在名字为 hello的文件中。不需要在这个文件名上使用扩展名，但是你加不加扩展名，Perl都无所谓。有些Windows和Macintosh实用程序使用扩展名来指明该文件是什么类型的文件。如果需要或者想要使用扩展名，那么常用的扩展名是.pl或.plx，因此你可以使用hello.pl这样的文件名。

1.3.2 运行程序

这时，若要运行该程序，必须进入到一个命令提示符下。在 UNIX中，请打开一个终端窗口，或者登录到系统中。在 Microsoft Windows计算机上，显示一个MS-DOS提示符。还应该

使用你的shell的cd命令，转到你存放hello程序的目录中。

当显示该提示符时，键入下面这个命令。（下面显示的是DOS提示符，UNIX提示符稍有不同。）

```
C:\PROGRAMS>perl hello
```

如果一切进行正常，Perl应该显示下面这行消息：

```
Hello, world!
```

如果你键入的这个命令运行正确，那么祝贺你的程序运行成功了！请记住如何运行这个程序，因为这也是你启动本书其余章节中的程序的方法。（也可以使用其他一些方法，下面很快就要介绍这些方法。）

如果该命令不能运行，请进行下列检查：

- 如果看到出错消息Bad command or file name或者perl : command not found，那么表示perl程序不在你的执行路径上。必须确定perl程序究竟安装在什么地方，并将该目录添加给你的shell中的PATH变量。
- 如果看到出错消息Can't open perl script hello : A file or directory does not exist（无法打开Perl脚本hello：文件或目录不存在），那么你可能没有进入早些时候保存hello文件所在的这个目录，也许你将该文件保存到另一个目录中了。
- 如果看到syntax error这样的错误，那么Perl能够正常启动运行，但是无法确定hello文件中究竟有什么。也许键入的文件内容有错误，也可能使用了一个文字处理程序，它将格式应用到保存的文件中了。可以使用UNIX的cat命令，或者DOS的type命令，来核定文件中的内容。如果键入的内容有误，你应该对所有内容进行检查，引号和所有标点符号都很重要。

如果使用MacPerl，只需要从Script菜单中选定Run“hello”，运行你的第一个perl程序。如果没有使用MacPerl的内置编辑器来编写你的程序，那么请从File菜单中选定Open命令，打开MacPerl中你的程序，然后选定Run。

1.3.3 程序正确将会发生什么情况

当键入命令perl hello时，你的计算机上一个名叫perl的程序就启动运行。该程序称为perl解释程序。这个perl解释程序是Perl的核心和灵魂。它的作用是取出对它赋予的文件(这里是文件hello)，找出里面的程序，并运行该程序。

所谓“运行该程序”，是指首先要检查构成Perl程序的语句、函数、运算符、数学算法和其他所有元素，以确保句法正确，然后每次运行一个语句。

当perl解释程序完成了从磁盘中读取你的程序的全部操作后，它即开始运行该程序，并且继续运行，直到整个程序运行完成。当它完成程序的运行时，perl解释程序退出，将控制权重新交给你的操作系统。

现在让我们来看一看hello程序是如何“运行”的。

1.3.4 Perl程序的具体运行过程

Hello程序的第一行是：

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

对于Perl来说，代码行上的 # 符号后面的一切均被视为注释行。注释是指 Perl 将对它加以忽略的一些东西。在某些情况下，程序的第一行上的 # ! 是不同的。它后面的路径名 /usr/bin/perl 是到达 perl 解释程序的路径。如果 UNIX 程序有一行是以 # ! 开头，后随一个解释程序的路径，那么 UNIX 就知道这是个程序，并且可以按照名字来运行。关于如何运行程序的说明，请参见本学时结尾处的“专家答疑”。

有些能够运行 Perl 程序的 Web 服务器，比如 Apache，也对 # ! 这个程序行非常注意，并且能够在没有 perl 命令的显式说明下运行该程序。

现在我们正好是将 # ! 视为一个注释行。

下一个程序行是：

```
print "Hello,World!\n";
```

这里包含了许多内容。这一行构成了一个 Perl 语句，它为 Perl 标明了一项需要执行的操作。

首先，这一行包含一个函数，称为 print。这个 print 函数取出它后面的所有内容，并默认显示在屏幕上。print 函数的输出结果，是直到分号（；）前的那部分内容。

Perl 中的分号是个语句分隔符。你应该在 Perl 程序中的各个语句之间放一个分隔符，以便显示一个语句的结束和另一个语句的开始。

在这个例子中，print 函数显示了短语 Hello,World!。程序行结尾处的 \n 告诉 Perl 将一个新的空行插入到它输出的短语的后面。短语前后的引号和 \n 告诉 Perl，这是个文字串，不是另一个函数。下一个学时将要非常详细地介绍字符串的内容。

1.3.5 必须知道的一些情况

Perl 被人们称为是一种自由格式的编程语言。这意味着 Perl 语句在编写的时候并不是非常严格。可以将空格、制表符，甚至回车符（它们称为白空间）插入 Perl 语句中的任何位置，这实际上都没有关系。

但是有些位置不能随意插入白空间，这些位置是应该加以限制的位置。例如，不能在函数名的中间插入空格，print 是个无效函数。另外不能将空格插入数字中，比如 25 61 这个数字是不行的。像“Hello World!”这样的文字串中的白空间当然可以显示为白空间。几乎任何其他地方它都是有效的。你可以编写类似下面这样的 Perl 程序示例：

```
#!/usr/bin/perl

print
    "Hello, World!\n"
;
```

这个程序在功能上与原先的程序是相同的。Perl 语言的这种自由格式特性使你的 Perl 程序可以具有非常丰富的“样式”。在程序的格式上可以有很大的随意性。不过请记住，总有一天其他用户会查看你的程序的，因此你不能让他们看不懂。

本书中的程序所用的样式是相当保守的。有时为了清楚起见，或者为了节省空间，语句被分成了若干行，因为 Perl 的语句可能非常长。Perl 的文档资料甚至提供了一个建议性的样式指南，可以浏览该文档，以便了解有关的建议。可以搜索名字为 perlstyle 的文档。



Perl程序中的样式有时可能非常特殊。有的 Perl程序可以写成诗歌，甚至俳句，这些都是有效的。有些令人难忘的 Perl程序看上去就像是图画，不过它们仍然能够做一些有用的工作。《Perl Journal》(网址是 <http://www.tPj.com>) 每年都要举行一次编写样式特殊的 Perl程序的比赛，比赛的名字是Obfuscated Perl Contest。你不应该采取这些程序项目的样式。

1.4 课时小结

在本学时中，我们学习了一些关于 Perl和Perl是如何运行的知识。随着你阅读本书其他章节的内容，你会不断增加对它的了解。还学习了如何在你的系统上安装 Perl，并且如何来检验它是否运行正确以及它的所有文档资料是否已经安装到位。最后，键入并且运行了你的第一个Perl程序。接着分析了这个程序，并且进一步学习了 Perl如何运行的一些知识。

1.5 课外作业

1.5.1 专家答疑

问题：Perl运行的那些东西是称为Perl脚本还是Perl程序？

解答：名字实际上并不重要。传统上，程序被编译成机器语言并且按照机器语言的形式来存放，机器语言是可以多次运行的。而脚本可以被放入一个外部程序中，每当脚本运行时，外部程序就将脚本转换成一些操作。Perl的发明人Larry Wall曾经说过：“脚本是你为操作人员提供的东西，而程序则是你提供给用户的东西。”你可以随意称呼它们。在本书的其余章节中，将它们称为Perl程序，如果你学习成绩很好的话，那么你就可以称为一名Perl程序员。

问题：是否必须键入本书中的一些程序清单？有些程序清单非常长。

解答：本书中的所有程序清单和程序示例，以及这些程序需要的数据文件等，都在本书所附的光盘上。

问题：在“运行程序”这一节中，讲到在 UNIX下有一种比较简便的方法可以用来运行Perl程序。究竟如何运行呢？

解答：首先，必须确保程序的 #！行正确无误，同时路径名确实指向一个 perl解释程序；/usr/bin/perl是它的通常位置，在有些机器上它的位置是 /usr/local/bin/perl。接着，必须使用chmod命令，使该程序能够执行。如果是 hello程序，那么UNIX的shell命令是chmod 755 hello。这项操作完成后，你可以键入 hello或 ./hello，运行该Perl程序。应该说明的是，在 UNIX下，不要将你的程序命名为“ test ”。UNIX的shell有一个命令的名字是test，当错误地运行test命令时，就会带来很大的麻烦。关于不能使用的其他程序名，请参见你的 shell的文档资料。

另外，如果使用光盘上的程序清单，必须修改 #！行，使之与你的系统上的 Perl位置相一致，否则，你将必须从命令提示符下键入 perl programname运行你的程序。

1.5.2 思考题

1) Perl是编程语言的名字；perl则

- a. 也是该语言的名字。

- b. 是解释程序的名字。
 - c. 是一个DOS命令的名字。
- 2) 在何处总是可以找到Perl文档资料的拷贝？
- a. <http://www.microsoft.com>
 - b. <http://www.perl.com>
 - c. <http://www.perl.net>
- 3) 在哪个手册页中能够找到Perl句法的描述？
- a. perlsyn
 - b. perlop
 - c. perlfaq

1.5.3 解答

- 1) 答案是b。不过，Perl安装后，它也是DOS shell中的一个有效的命令。因此c也是对的。
- 2) 答案是b。它也可以安装在你的系统上。
- 3) 答案是a。除非你运行perldoc perl，否则无法明确知道能否在那里找到该描述。

1.5.4 实习

- 请浏览FAQ(常见问题)。即使你不能理解它里面的所有内容，也能够大致了解FAQ中可以得到哪些类型的信息。

如果你喜欢通过浏览器来阅读它的内容，请搜索 <http://www.perl.com>，并在那里阅读它的内容，不过应该仔细阅读才行。

China-pub.com

下载

China-pub.com

下载

第2学时 Perl的基本构件：数字和字符串

每种编程语言，以及人类的每种语言，都有一个相似的出发点，那就是必须要有谈话的要素。在Perl中，数字和字符串就是谈话的基本单位，这些基本单位称为标量。

标量是Perl的基本谈话单位。本书中的每个学时都要涉及到标量，对标量可以进行增加、减少、查询、测试、集中、清除、分隔、折叠、排序、保存、加载、输出和删除等操作。标量是Perl的单个名词，它们可以代表一个单词、一个记录、一个文档、一行文本或者一个字符。

Perl中的标量能够代表直接量数据，它在程序的生命期内是不变的。有些编程语言将这些值称为常量或直接量。直接量数据可以用于表示没有变化的值，比如 的值，物体落地的加速度和美国第15届总统的名字等。如果一个Perl程序需要这些值，那么在程序的某个位置上可以用一个标量直接量来代表它们。

Perl中还有另一些类型的标量是变化的，它们称为标量变量。变量可以在你对它进行操作时用来存放数据。可以改变变量的内容，因为它们只是作为它们代表的数据的句柄而存在的。变量要被赋予相应的名字，这些名字比较简单，而且很容易记住，它们可以帮助你引用你要操作的数据。

本学时还要介绍Perl的运算符。运算符是Perl语言中的一种动词，运算符取出Perl的名词，负责从事你在编写执行特定任务的程序时需要进行的实际操作。

在本学时中，将学习下列内容：

- 直接量数字和字符串。
- 标量变量。
- 运算符。

2.1 直接量

Perl拥有两种不同类型的标量常量，它们都称为直接量。一种是数字直接量，一种是字符串直接量。

2.1.1 数字

数字直接量都是一些数字，Perl可以接受若干种不同的数字写法。表 2-1显示的所有例子在Perl中都是有效的数字直接量。

表2-1 数字直接量示例

数 字	直接量的类型
6	整型数
12.5	浮点数
15.0	另一个浮点数
0.7320508	也是一个浮点数
1e10	科学记数法
6.67E-33	科学记数法（e或E均可以）
4_294_296	带有下划线而不是逗号的大数字

数字可以根据你设想的样子来加以表示。整数是一些连续的数字。至于浮点（十进制）数，可以按照通常的形式使用小数点。科学记数法用一个指数字母 e 和一个尾数（对数的十进制部分）来表示。至于大整数，可以在通常使用逗号的地方换上下划线，以便于阅读。当使用数字值的时候，Perl会删除这些下划线。



在数字前面不要使用前导 0，比如 010。对于 Perl 来说，这个数字代表一个八进制数字，它的基数是 8。Perl 还允许使用十六进制直接量数字（基数是 16）和二进制数字（基数是 2）。关于这些数字的详细信息请参见 perldata 部分的在线文档。

2.1.2 字符串

Perl 中的字符串直接量是指原义字符构成的串。它们能够包含你所想要的那么多数据。字符串的长度实际上是没有限制的，不过不能超出计算机中的虚拟内存的容量。字符串也可以包含任何种类的数据，比如简单的 ASCII 文本，最高位为 1 的 ASCII 码，甚至二进制数据。字符串也可以是空的。

应该用引号将字符串直接量括起来，不过在 Perl 中也有很少的一些例外。这个过程称为给字符串“加引号”。给字符串加引号有两种主要方法，一种是使用单引号（'），一种是使用双引号（"）。下面是字符串直接量的一些示例：

```
"foo!"  
'Fourscore and seven years ago'  
"One fish,\nTwo fish,\nRed fish,\nBlue fish\n"  
"  
"Frankly, my dear, I don't give a hoot.\n"
```

在双引号中，如果需要插入另一个引号，则必须使用反斜杠转义符。字符串直接量中的反斜杠（\）用于告诉 Perl，它后面的字符不应该按通常的情况来处理，在这种情况下，它应该被忽略。例如，下面这个字符串直接量对 Perl 来说就没有任何意义：

```
"Then I said to him, "Go ahead, make my day""
```

在这个字符串中，单词 Go 前面的引号与第一个引号是一对，从而将 Go ahead,make my day 这个短语留在引号的外面，因此这不是一个有效的 Perl。为了防止这种情况的发生，请在你想要让 Perl 不当成引号对待的引号的前面放一个反斜杠，如下所示：

```
"Then I said to him, \"Go ahead, make my day.\""
```

反斜杠使得 Perl 能够知道后面的引号与第一个引号不是匹配的一对，因此它应该不被作为引号处理。这条规则既适用于单引号，也适用于双引号，请看下面的例子：

```
'The doctors\'s stethoscope was cold.'
```

给字符串加双引号和单引号的主要差别是：使用单引号的字符串含义是非常直观的，单引号字符串中的每个字符就是表示它自己的含义。在双引号中的字符串中，Perl 要查看是否存在变量名或转义序列。转义序列是一些特殊字符串，你可以将难以键入和以后难以识别的字符嵌入字符串。表 2-2 显示了一个 Perl 的转义序列的简短列表。

表2-2 示例字符串的转义序列

序 列	含 义
\n	换行
\r	回车
\t	制表符
\b	退格
\u	将下一个字符改为大写
\l	将下一个字符改为小写
\\"	直接量反斜杠字符
\'	用单引号('')括起来的字符串中的直接量'
\"	用引号括起来的字符串中的直接量"



可以从在线手册中找到转义序列的完整列表。正如在第一学时中介绍的那样，可以使用Perl中包含的perldoc实用程序，找到整个Perl语言的文档资料。转义序列位于“perlop”手册中的“Quote and Quote-like Operators（引号与引号式的运算符）”这个标题下。

如果在字符串中包含许多个引号，那么当每个嵌入的引号必须转义时，键入字符串就会非常困难，而且很容易发生错误。请看下面这个例子：

```
'The doctors\'s stethoscope was cold.'
```

Perl还提供了另一个引号机制，即qq和q运算符。使用qq运算符，就可以用qq()而不是引号将字符串括起来：

```
qq(I said, "Go then," and he said "I'm gone")
```

qq取代了双引号。这个机制的作用几乎在所有方面都与双引号完全一样。

也可以用q运算符来代替单引号将文本括起来：

```
q(Tom's kite wedged in Sue's tree)
```

qq和q运算符可以使用任何非字母、非数字字符来标记字符串的开始和结束。这些标记称为界限符。在前面这个例子中使用了括号，不过也可以使用任何其他的非字母或非数字字符作为界限符：

```
q/Tom's kite wedged in Sue's tree/
```

```
q, Tom's kite wedged in Sue's tree,
```

你想用作界限符的字符必须出现在紧靠运算符qq或q的后面。使用任何一个成对的界限符，如()，<>，{}，[]，它们都能够正确地将字符串括起来。这就是说，如果在qq或q运算符中以偶数对的形式使用这些界限符，就不必使用反斜杠转义符：

```
q(Joe (Tom's dad) fell out of a (rather large) tree.);
```

但是，加上这些界限符后，Perl程序的可读性就会有所降低。通常来说，只选择字符串中不出现的界限符，读起来就比较容易。

```
q[Joe (Tom's dad) fell out of a (rather large) tree.];
```

2.2 标量变量

若要将标量数据存放在Perl中，必须使用标量变量。在Perl中，如果要写一个标量变量，

可以写一个美元符号，后跟变量的名字。下面是一些标量变量的例子：

```
$a  
$total  
$Date  
$serial_number  
$cat450
```

美元符号称为类型标识符，用于告诉Perl该变量包含标量数据。其他变量类型（哈希变量和数组）则使用不同的标识符，甚至根本没有标识符（文件句柄）。Perl中的变量名，比如哈希变量、数组、文件句柄和标量，必须符合下列规则：

- 变量名可以包含字母（a至z,A至Z）字符、数字或类型标识符后面的一个下划线字符（_）。不过，变量名的第一个字符不能是数字。
- 变量名是区分大小写字母的。这意味着变量名中的大写和小写字母都是有特定意义的，因此下面这些变量均代表不同的标量变量：

```
$value  
$VALUE  
$Value  
$valuE
```

Perl只使用单字符变量名，它们不是以字母字符或下划线开始的。以\$_、\$”、\$/、\$2和\$\$开始的变量均属于特殊变量，在Perl程序中不能用作普通变量。这些特殊变量的作用将在下面介绍。

Perl与某些其他编程语言不同，Perl中的标量变量在你使用它们之前，不必预先进行声明或初始化。若要创建一个标量变量，只要使用它就行了。当使用一个未经初始化的变量时，Perl将使用它的默认值。如果它被用作数字时（如数学运算中的数字），Perl将使用0（零）这个值；如果它被用作字符串（几乎其他任何地方都使用），那么Perl将使用“ ”这个值，即空字符串。



在用一个值对变量进行初始化之前就使用该变量，这被认为是一种不好的编程做法。当你在命令行上使用-w开关，或者在程序开头的#!行上使用-w来调用Perl程序时，Perl就会向你发出警告。如果你试图使用的变量值预先没有进行设定，那么当你的程序运行时以及试图使用该值时，Perl就会报一条出错消息Use of uninitialized value（使用未经初始化的值）。

特殊变量\$_

Perl拥有一个特殊变量，它的值可以用作“默认值”。对于许多运算符和函数来说，该变量称为\$变量。例如，如果你只是使用输出函数本身，而不设定要输出什么，那么它将输出\$的当前值：

```
$_="Dark Side of the Moon";  
print; # Prints the value of $_, "Dark side..."
```

像这样来使用\$_，肯定会造成某些混乱。因为它确实没有指明究竟输出的是什么，尤其是给\$_的赋值出现在程序中较高的位置上时更是如此。

有些运算符和函数实际上可以比较容易地用于\$_变量，尤其是第6学时中介绍的模式匹配运算符。不过在本书中将尽可能少用\$_，这样学习起来就比较容易一些。

2.3 表达式和运算符

既然你已经学习了什么是标量数据并且知道如何使用标量变量，那么现在就可以开始用Perl来执行一些有用的操作了。Perl程序实际上是一些表达式和语句的集合，它们从Perl程序的顶部向底部顺序执行，不过你也可以设定流控制语句的执行顺序，这将在第3学时“控制程序流”中介绍。程序清单2-1显示了一个有效的Perl程序。

程序清单2-1 一个简单的Perl程序

```

1:  #!/usr/bin/perl -w
2:
3:  $radius=50;
4:
5:  $area=3.14159*($radius ** 2);
6:  print $area;

```

第1行：这一行是到达Perl解释程序的路径，这在第1学时中已经做了介绍。开关-w告诉Perl，只要遇到警告就通知你。

第3行：这一行是个赋值行。数字标量数据50存放在标量变量\$radius中。

第5行：这一行是另一个赋值行。在赋值运算符的右边是个表达式，该表达式包含了标量变量\$radius、运算符(*和**，下面将介绍这两个运算符)和数字标量(2)。该表达式的值经过计算后被赋予\$area。

第6行：这一行负责输出存放在\$area中的计算结果。

表达式是具有值的简单元素。例如，2是个有效的表达式。 $54 * \$r$ 、“Java”、 $\sin(\$pi * 8)$ 和 $\$t = 6$ 等，也都是表达式。表达式的值是在程序运行时进行计算的。程序对表达式中的函数、运算符和标量常量进行计算，然后得出一个值。可以将这些表达式用于赋值，或者作为其他表达式的一个组成部分，也可以作为其他Perl语句的组成部分。

2.3.1 基本运算符

正如你在程序清单2-1中看到的那样，若要将标量数据赋予一个标量变量，可以使用赋值运算符=。这个赋值运算符取出位于右边的值，再将它放入左边的变量中：

```

$title="Gone With the Wind";
$pi=3.14159;

```

赋值运算符左边的操作数必须是能够被赋予变量的一个值。右边的操作数可以是任何种类的表达式。赋值的本身就是一个表达式，它的值是右边表达式的值。这就是说，在下面这个代码段中，\$a、\$b和\$c都被设置为42。

```
$a=$b=$c=42;
```

这个代码段中，\$c首先被设置为42。\$b被设置为表达式\$c=42(它就是42)的值，然后\$a被设置为表达式\$b=42的值。被赋值的变量甚至可以出现在赋值运算符的右边，如下所示：

```

$a=89*$a;
$count=$count+1;

```

赋值运算符的右边使用\$a或\$count的老的值进行计算，然后，它的结果作为新值被赋予左边。第二个例子在Perl中有一个特殊的名字，它称为递增。在后面我们还要更加详细地介绍递增。

2.3.2 数字运算符

Perl中有许多运算符可以用来对数字表达式进行操作。这些运算符中，有些你已经非常熟悉，有些是第一次遇到。已经知道的第一种运算符是数学运算符。表 2-3显示了这些运算符的一个列表。

表2-3 数学运算符

举 例	运 算 符 名	表达式的值
5 + \$t	加	5与\$t相加的和
\$y - \$x	减	\$y与\$x的差
\$e * \$pi	乘	\$e与\$pi的积
\$f / 6	除	\$f除以6得出的商
24 % 5	求余数	24除以5得出的余数(4)
4**2	取幂	取4的二次方

数学运算符是按照我们通常的规则进行计算的，先取幂，再乘除，然后求余数，最后进行加减。如果你不清楚表达式中各个元素的计算顺序，可以使用括号将元素括起来，以确保计算顺序的正确。嵌套的括号总是从里向外进行计算：

```
5*6+9;      # Evaluates to 39
5*(6+9);    # Evaluates to 75
5+(6*(4-3)); # Evaluates to 11
```

2.3.3 字符串运算符

在Perl中，数字值并不是可能受运算符影响的惟一值，字符串也可以进行相应的操作。第一个字符串运算符是并置运算符，用圆点(.)来代表。并置运算符取出左边的字符串和右边的字符串，返回一个将两个字符串合并在一起的字符串：

```
$a="Hello, World!";
$b=" Nice to meet you";
$c=$a . $b;
```

在这个例子中，\$a和\$b被赋予一些简单的字符串值。在最后一行上，\$a和\$b并置在一起，变成Hello,World! Nice to meet you，然后存放在\$c中。\$a与\$b没有被并置运算符修改。

并置式的操作特性可以用另一种方式来执行。在前面，你了解到 Perl在双引号字符串中查找变量。如果Perl在双引号字符串中找到了一个变量，那么它将被内插替换。这就是说，双引号字符串中的变量名将被它的实际值代替：

```
$name="John";
print "I went to the store with $name.,";
```

在这个例子中，Perl查看双引号字符串，发现\$name，并对字符串John进行替换。这个过程称为变量内插替换。为了防止变量查找的字符串被内插替换，可以使用单引号（它不进行任何形式的内插替换），也可以在变量标识符的前面加上一个反斜杠：

```
$name="Ringo";
print 'I used the variable $name';      # Will not print "Ringo"
print "I used the variable \$name";     # Neither will this.
```

上面这个例子中的两个print语句均用于输出I used the variable \$name，字符串\$name没有被内插替换。因此，若要在不使用并置运算符的情况下执行并置操作，只要使用双引号字符串即可，如下所示：

```
$fruit1="apples";
$fruit2="and oranges";
$bowl="$fruit1 $fruit2";
```

如果Perl不能清楚地指明变量名在何处结束和字符串的其余部分从何处开始，那么可以使用花括号将变量名括起来。使用这个句法，Perl就能够找到可能模糊的变量名：

```
$date="Thurs";
print "I went to the fair on ${date}day";
```

如果没有花括号，Perl将不知道是要对双引号中的\$date还是对变量\$dateday进行内插替换。加上花括号后，内插替换的对象就清楚了。

这里出现的另一个字符串运算符是重复运算符x。运算符x配有两个参数，一个是要重复的字符串，另一个是该字符串重复的次数，请看下面这个例子：

```
$line="-" x 70;
```

在上面这个例子中，字符-被运算符x重复70次。其结果存放在\$line中。

2.4 其他运算符

Perl的运算符非常多，由于本书的篇幅有限，因此无法一一列举。在本学时剩下的时间里，将要介绍Perl中最常用的一些运算符和函数。

2.4.1 单参数运算符

到现在为止，我们介绍的所有运算符都带有两个参数。除法(6/3)需要一个被除数(6)和一个除数(3)，乘法(5*2)需要一个被乘数(5)和一个乘数(2)，如此等等。另一种运算符只有一个参数。你可能已经熟悉这种运算符的一个例子，即一目减运算符(-)。一目减运算返回变成负数的参数值：

```
6;      # Six
-6;     # Negative six.
-(-5);  # Positive five, not negative five.
```

Perl的许多运算符实际上是带名字的运算符，也就是说，它们不是一个符号，比如一目减操作中的-符号，它们是一个单词。带名字的一目减运算符的操作数前后的括号是可有可无的，但是为了清楚起见，表2-4中都将括号显示了出来。由于Perl中带名字的运算符和函数看上去非常相似，因此带名字的运算符的操作数有时也称为变元，这是Perl函数用于它们的参数的一个术语。

表2-4显示了某些带名字的运算符的一个简单列表。

表2-4 一些带名字的运算符

运 算 符	用 法 举 例	结 果
int	int(5.6234)	返回它的参数的整数部分(5)
length	length("nose")	返回它的字符串参数的长度(4)
lc	lc("ME TOO")	返回它的转换成小写字母的参数("me too")
uc	uc("hal 9000")	返回与lc相反的参数值("HAL 9000")
cos	cos(50)	返回弧度50的余弦值(.964966)
rand	rand(5)	返回从0到小于该参数值之间的一个随机数字。如果该参数被省略，则返回0至1之间的一个数字

可以从在线手册中找到带名字的运算符的完整列表。我们在第 1 学时中讲过，可以使用 Perl 产品中包含的 perldoc 实用程序找到 Perl 语言的完整文档。运算符的完整列表位于“ perlop ” 和“ perlfunc ” 手册中。其他运算符将根据需要在后面的一些学时中介绍。

2.4.2 递增和递减

在“数字运算符”这一节中，我们讲到了一种特殊的赋值类型，称为递增，它类似下面的形式：

```
$counter=$counter+1;
```

递增通常用于计数，比如读取的记录数，或者生成序列号，比如给列表中的项目编号。在 Perl 中，它是个非常常用的术语，因此可以使用一个特殊运算符，称为自动递增运算符 (++)。自动递增运算符能够给它的操作数递增 1：

```
$counter++;
```

在执行这个代码后，\$counter 递增 1。

Perl 还提供了一种快捷运算符，用于递减一个变量的值，这个运算符称为自动递减运算符 (--)。自动递减运算符的使用方法与自动递增运算符的使用方法完全相同：

```
$countdown=10;  
$countdown--; # decrease to 9
```

关于自动递增运算符还有最后一个问题需要加以说明，那就是当你将它用于一个文本字符串，而该文本字符串是以字母字符开始，后随字母字符或数字，那么这个运算符就具有一种非常特殊的作用。字符串的最后一个（最右边的）字符被递增。如果它是个字母字符，它将成为序列中的下一个字母；如果它是个数字，那么该数字将递增 1。你可以像下面这样通过各个字母列和数字列：

```
$a="999";  
$a++;  
print $a; # prints 1000, as you'd expect  
$a="c9";  
$a++;  
print $a; # prints d0. 9+1=10, carry 1 to the c.  
$a="zzz";  
$a++;  
print $a; # prints "aaaa".
```

自动递减运算符并不像上面那样对字符串进行递减。

2.4.3 尖括号运算符

尖括号运算符 (<>)，有时也称为菱形运算符，主要用于读写文件。我们将在第 5 学时详细介绍这个运算符。不过现在要对它做一个简单的介绍，这会使我们的练习更加有趣。当你开始学习第 5 学时的内容时，就会对该运算符有所了解。

到那时，你将能够用它的最简单的形式，即 <STDIN> 来使用尖括号运算符。这种形式告诉 Perl，应该从标准输入设备（通常是键盘）读取一行输入信息。<STDIN> 表达式返回从键盘读取的这行信息：

```
print "What size is your shoe? ";  
$size=<STDIN>;  
print "Your shoe size is $size, Thank you";
```

上面这个代码在执行的时候（假设某人键入 9.5作为它的鞋子的尺寸），将类似下面的形式：

```
What is size is your shoe? 9.5
Your shoe size is 9.5
Thank you
```

<STDIN>表达式从键盘读取信息，直到用户按下 Enter键为止。整个输入行返回，并被存放在\$size中。由<STDIN>返回的文本行也包含用户键入的换行符（因为按下了 Enter键）。在大多数情况下，你不希望在字符串的结尾处出现换行符。若要删除换行符，可以像下面这样使用chomp运算符：

```
print "What size is your shoe?";
$size=<STDIN>;
chomp $size;
print "Your shoe size is $size, or so you say.\n";
```

chomp运算符能够删除它的参数结尾处的任何换行符。它返回被删除的字符数，这个数字通常是1，但是，有时如果没有字符需要删除，那么返回的是0。

2.4.4 其他赋值运算符

前面我们曾经讲过，可以使用赋值运算符（=），将一个值赋予一个标量变量。实际上 Perl 拥有一组完整的运算符，可以用于进行赋值操作。Perl的每个数学运算符和相当多的其他运算符可以组合起来同时进行赋值和运算操作。下面是建立一个带有运算符的赋值语句的一般规则：

变量 运算符=表达式

这个规则与下面这个规则相同：

变量=变量 运算符 表达式

使用赋值语句通常不会使程序更加容易阅读，但是它能够使程序更加简洁。例如，按照这个规则，语句\$a=\$a+3；可以简化为 \$a+=3。

下面是另一些赋值语句的例子：

```
$line.=", at the end";           # ", at the end" is appended to $line
$y**=$x                          # same as $y=$y*$x
$r%=$7;                           # Divide by 67, put remainder in $r
```

2.4.5 关于字符串和数字的一些说明

总的来说，Perl允许你对数字和字符串进行互换使用。它使用的表示法取决于在这种情况下Perl查找的是什么。

- 如果某个元素看上去是个数字，那么Perl在需要数字时可以将它用作数字：

```
$a=42;                      # A number
print $a+18;      # displays 60.
$b="50";
print $b-10;      # Displays 40.
```

- 如果某个元素看上去是个数字，那么当Perl需要一个字符串时，它可以使用数字的字符串表示法：

```
$a=42/3;
$a=$a . "Hello";  # Using a number like a string.
print $a            # displays "14Hello"
```

- 如果某个元素看上去不像一个数字，但是你将它用在需要数字的地方，那么 Perl 在它的位置上使用 0 这个值：

```
$a="Hello, World!";
print $a+6;      # displays the number 6
```

但是，如果你激活了警告特性，那么如果你这样操作的话，Perl 就会发出警告消息。

所有这些用法都与 Perl 的“尽可能随意自然”的原则是一致的。即使毫无意义，就像最后一个例子中的情况那样，Perl 也设法用它来做一些有意义的事情。如果在 Perl 程序中激活了警告特性，在 #! 行上加上了一个 -w 开关，或者运行带有 -w 选项的 Perl 程序，Perl 就会发出警告，说明你的操作毫无意义，并给出下面这条消息：Argument x isn't numeric (参数 x 不是个数字)。

2.5 练习：利息计算程序

这个练习是进行复利的计算。这个程序将根据利率、存款和时间等信息来计算储蓄帐户的利息。下面是你要使用的计算公式：

$$\text{accrued} = \text{payment} \left(\frac{(1 + \text{monthly interest})^{\text{number of deposits}} - 1}{\text{monthly interest}} \right)$$

使用文本编辑器，键入程序清单 2-2 中的程序，并将它保存为 Interest。不要键入行号。根据你在第 1 学时中学习到的方法，使该程序成为可执行程序。

程序清单 2-2 利息计算程序的完整源代码

```

1:  #!/usr/bin/perl -w
2:
3:  print "Monthly deposit amount? ";
4:  $pmt=<STDIN>;
5:  chomp $pmt;
6:
7:  print "Annual Interest rate? (ex. 7% is .07) ";
8:  $interest=<STDIN>;
9:  chomp $interest;
10:
11: print "Number of months to deposit? ";
12: $mons=<STDIN>;
13: chomp $mons;
14:
15: # Formula requires a monthly interest
16: $interest/=12;
17:
18: $total=$pmt * ( ( 1 + $interest ) ** $mons -1 )/ $interest;
19:
20: print "After $mons months, at $interest monthly you\n";
21: print "will have $total.\n";

```

第1行：这一行包含了到达解释程序的路径（可以对它进行修改，使它适合你的系统的需要）和-w开关。请始终使警告特性处于激活状态。

第3行：这一行提示用户输入一个金额。

第4行：从标准输入设备（键盘）读取 \$pmt。

第5行：从\$pmt的结尾处删除换行符。

第7至9行：从键盘读取\$interest，同时删除换行符。

第11至13行：从键盘读取\$mons，同时删除换行符。

第16行：\$interest除以12，并且重新存入\$interest。

第18行：执行利息计算，将结果存入\$total。

第20至21行：输出计算结果。

当你完成操作时，在一个命令行上键入下面的命令，设法运行该程序：

```
perl Interest
```

程序清单2-3显示了一个利息计算程序输出信息的例子。

程序清单2-3 利息计算程序的输出

```
1: Monthly deposit amount? 180
2: Annual Interest rate? (ex. 7% is .07) .07
3: Number of months to deposit? 120
4: After 120 months, at 0.005 annual you
5: will have 29498.2824251624.
```

2.6 课时小结

在本学时中，我们了解到Perl的最基本的数据类型是标量。标量几乎可以是任何类型的数据，并且可以存放在标量变量中。数字直接量可以用许多不同的格式来表示，比如整数、浮点数等。字符串直接量用加引号的字符串来表示，字符串的前后既可以使用双引号，也可以使用单引号。另外，Perl提供了许多运算符，用于进行字符串的操作和基本的算术运算。

2.7 课外作业

2.7.1 专家答疑

问题：利息计算程序的输出显得有些杂乱，我如何才能控制它显示几位小数点？

解答：控制小数点位数的最简单的方法是使用printf函数。我们将在第9学时中介绍这个函数。

问题：Perl是否有一个用于数字舍入的函数呢？

解答：当显示数字时，printf函数通常可以用来进行你想要做的舍入操作。如果你确实需要round函数，请查看POSIX模块，它包含了该函数和其他许多函数。

问题：Perl允许我操作的数字究竟可以有多大（或者多小）？

解答：答案取决于你使用的是什么操作系统。典型的Intel UNIX系统的双精度浮点数的指数幂可以超过300位。这就是说你操作的数字小数点的右边（或左边）可以有300个0。不过大数字的精度通常只有14位数。

2.7.2 思考题

1) 变量可以在qq引号中进行内插替换。

- a. 是
- b. 否

2) 运行下面这个代码后，什么值将存放在 \$c 中？

```
$a=6;  
$a++;  
$b=$a;  
$b--;  
$c=$b;
```

- a. 6
- b. 7
- c. 8

3) 并置操作只能使用并置运算符 (.) 进行。

- a. 是
- b. 否

2.7.3 解答

- 1) 答案是 a。qq 的作用与一对双引号完全相同。这意味着它能够对变量进行内插替换。
- 2) 答案是 a。\$a 被设置为 6，然后 \$a 递增为 7，并被赋值给 \$b。\$b 递减为 6，并被赋值给 \$c。
- 3) 答案是 b。在第 1 学时中我们讲过，在 Perl 中，一项操作可以使用多种方法来进行（这句英文可以缩写为 TIMTOWTDI）。并置操作可以在双引号字符串中包含两个（或多个）标量，如下所示：

```
qq( $a$b$c);
```

2.7.4 实习

- 请编写一个短程序，提示用户输入一个华氏温度值，并输出摄氏温度值。若要将华氏变成摄氏，你可以将华氏温度减去 32，然后乘以 5/9。例如，华氏 75 度等于摄氏 21.1 度。
- 修改程序清单 2-3 中的利息计算程序，输出的金额小数点不要超过两位。你可以不使用 printf 函数，而巧妙地使用 int 运算符、乘法和除法来进行计算。

China-pub.com

下载

China-pub.com

下载

第3学时 控制程序流

在第2学时中，我们学习了程序的语句、运算符和表达式等内容。该学时中的所有例子有一个共同点，那就是所有的语句都是从上向下执行的，并且只执行一次。

我们之所以使用计算机，原因之一是计算机非常擅长执行重复任务，一次又一次地重复，永不疲倦，也不会诱发手腕综合症。迄今为止，还没有任何办法来告诉 Perl，让它“执行该任务X次”，或者“重复执行该任务，直到任务完成为止”。在本学时中，我们要介绍Perl的控制结构。使用这些控制结构，就能够将语句组合成所谓的“语句块”，并且重复执行这些语句组，直到它们完成你想要完成的工作。

计算机擅长的另一项工作是能够迅速作出决策。如果计算机每次作出决策时都要询问你，那么你一定会感到非常讨厌，而且也说明计算机太笨了。检索和读取电子邮件，可使你的计算机作出上百万个决策，而这些决策并不是你想要处理的。比如，如何组合网络信息，什么颜色构成屏幕上的每个像素，你收到的电子邮件应该如何分类和显示，当鼠标光标稍稍移动了一点儿时应该怎么办，如此等等，这些都需要迅速作出决策。所有这些决策又是由其他决策构成的，而且其中的一些决策需要每秒钟作出数千次。在本学时中，我们将要介绍条件语句。使用这些语句，你就能够编写代码块，这些代码块是根据你的Perl程序中作出的决策来执行的。

在本学时中，我们将要介绍下列基本概念：

- 语句块。
- 运算符。
- 循环。
- 标号。
- 程序执行后退出Perl。

3.1 语句块

Perl中最简单的语句组合称为块。若要将语句组合在一个块中，只需要用一组匹配的花括号将语句括起来即可，如下所示：

```
{  
    statement_a;  
    statement_b;  
    statement_c;  
}
```

在语句块中，语句像前面已经介绍过的情况那样，从上向下执行。在语句块中，你可以拥有另一些语句块，请看下面的例子：

```
{  
    statement_a;  
    {  
        statement_x;  
        statement_y;  
    }  
}
```

语句块的格式与Perl的其他程序格式一样，是自由随意的格式。语句与花括号可以如下面所示放在同一行上，也可以放在若干不同行上，并且可以根据你的需要，采用任何一种对齐方式，只要使用匹配的一组花括号即可：

```
{ statement; { another_statement; }
  { last_statement; } }
```

虽然你可以按照你的喜好来安排语句块，但是，如果仅仅将各个语句凑合在一起，那么程序就很难阅读。采用好的缩进方式虽然并不是必须的，但是却能建立便于阅读的 Perl程序。

程序中出现的孤立语句块称为裸语句块。不过大多数情况下，你遇到的语句块是附属在其他Perl语句后面的。

3.2 if语句

若要根据Perl程序中的某个条件来控制语句是否执行，通常可以使用 if语句。下面显示了if语句的句法：

```
if (expression) BLOCK
```

该语句的工作原理是：如果表达式计算的结果是真（ TRUE），那么该代码块运行；如果该表达式的计算结果是假（ FALSE），那么代码块不运行。请记住，该代码块包含了花括号。请看下面这个例子：

```
if ( $r == 5 ) {
    print 'The value of $r is equal to 5.';
}
```

该代码中被测试的表达式是 \$r == 5。符号==是个等式运算符。如果等式两边的两个操作数（\$r和5）的数值是相等的，那么该表达式被视为真，同时 print语句执行。如果 \$r不等于5，那么print语句不执行。

如果一个条件是真，那么 if语句也能运行代码，否则，如果该条件不是真，则运行另一个代码。该结构称为 if-else语句，它的句法类似下面的样子：

```
if (expression)      # If expression is true...
  BLOCK             # ...this block of code is run.
else
  BLOCK           # Otherwise this block is run.
```

只有当表达式是真的时候，表达式后面的语句块才运行；如果表达式不是真，那么 else后面语句块运行。现在请看下面这个例子：

```
$r=<STDIN>; chomp $r;
if ($r == 10) {
    print '$r is 10';
} else {
    print '$r is something other than 10...';
    $r=10;
    print '$r has been set to 10';
}
```



请注意，在上面这个例子中，为了将一个值赋予 \$r，使用一个赋值运算符=。为了测试\$r的值，使用等式运算符==。这两个运算符有着很大的差别，其作用是不同的。在你的程序中不要混淆它们的用法，因为调试非常困难。请记住，运算符=用于赋值，而==则用于测试一个等式。

建立if语句的另一种方法是使用多个表达式，然后根据哪个表达式是真，来运行代码：

```
if (expression1)      # If expression1 is true ...
    BLOCK1           # ...run this block of code.
elsif (expression2)  # Otherwise, if expression2 is true...
    BLOCK2           # ...Run this block of code.
else
    BLOCK3          # If neither expression was true, run this.
```

可以像下面这样来读取上面这个语句块：如果标号为 expression1的表达式是真，那么语句块BLOCK1就运行。否则，控制权转给 elsif，对expression2进行测试，如果该表达式是真，则运行BLOCK2。如果expression1和expression2都不是真，那么BLOCK3运行。下面是一个实际的Perl代码的例子，用于显示这个句法的实际情况：

```
$r=10;
if ($r==10) {
    print '$r is 10!';
} elsif ($r == 20) {
    print '$r is 20!';
} else {
    print '$r is neither 10 nor 20';
}
```

3.2.1 其他关系运算符

到现在为止，我们都是使用等式运算符 == 来比较if语句中的数字的量值。实际上 Perl 还有一些运算符可以用来进行数字值的测试，表 3-1列出了这种运算符的大部分。

表3-1 数字关系运算符

运 算 符	举 例	说 明
==	\$x == \$y	如果\$x等于\$y，则为真
>	\$x > \$y	如果\$x大于\$y，则为真
<	\$x < \$y	如果\$x小于\$y，则为真
>=	\$x >= \$y	如果\$x大于或者等于\$y，则为真
<=	\$x <= \$y	如果\$x小于或者等于\$y，则为真
!=	\$x != \$y	如果\$x不等于\$y，则为真

若要使用这些运算符，只需要将它们放在你的程序中需要测试数字值之间的关系的任何位置，就像程序清单 3-1 中所示的if语句那样。

程序清单 3-1 一个小型猜数字游戏

```
1:  #!/usr/bin/perl -w
2:
3:  $im_thinking_of=int(rand 10);
4:  print "Pick a number:";
5:  $guess=<STDIN>;
6:  chomp $guess;  # Don't forget to remove the newline!
7:
8:  if ($guess>$im_thinking_of) {
9:      print "You guessed too high!\n";
10: } elsif ($guess < $im_thinking_of) {
11:     print "You guessed too low!\n";
12: } else {
13:     print "You got it right!\n";
14: }
```

第1行：这一行是Perl程序第一行的标准格式，它指明你想要运行的解释程序和用于激活警告特性的-w开关。与第1学时的内容比较，你会发现那里的第一行与这里的第1行稍有不同。

第3行：函数(rand 10)取出0至10之间的一个数字，int()对该数字范围进行舍位，这样只有0到9的整数被赋予\$im_thinking_of。

第4~6行：这一行让用户进行猜测，将它赋予\$guess，并删除结尾处的换行符。

第8~9行：如果\$guess大于\$im_thinking_of中的数字，那么这些行将输出一个相应的消息。

第10~11行：否则，如果\$guess小于\$im_thinking_of中的数字，那么这些行便输出该消息。

第12~13行：剩下的惟一选择是用户猜测该数字。

表3-1中的运算符只能用于测试数字值。使用这些运算符可以测试你可能不想要的运行特性中的非字母数据的结果。请看下面这个例子：

```
$first="Simon";
$last="simple";
if ($first == $last) {      # == is not what you want!
    print "The words are the same!\n";
}
```

\$first和\$last中存放的两个值实际上是要测试它们之间是否相等。对它们进行测试的原因在第1学时中已经做了说明。当Perl期望数字值的时候，如果使用了非数字字符串，那么这些字符串的计算结果将是0。因此，上面这个if表达式在Perl看来就像是：if(0 == 0)。这个表达式的计算结果是真，这可能不是你想要的结果。



如果程序中的警告特性被打开，那么当程序运行时，用==运算符来测试两个字母字符值（上面代码段中的simple和Simon），就会产生一个警告消息，以提醒你存在这个问题。

如果你想要测试非数字值，你可以使用另一组Perl运算符，表3-2列出了这些运算符。

表3-2 字母关系运算符

运 算 符	举 例	说 明
eq	\$s eq \$t	如果\$s等于\$t，则为真
gt	\$s gt \$t	如果\$s大于\$t，则为真
lt	\$s lt \$t	如果\$s小于\$t，则为真
ge	\$s ge \$t	如果\$s大于或者等于\$t，则为真
le	\$s le \$t	如果\$s小于或者等于\$t，则为真
ne	\$s ne \$t	如果\$s不等于\$t，则为真

这些运算符通过从左到右观察每个字符，然后按照ASCII的顺序对它们进行比较，来确定“大于”和“小于”。这意味着字符串按照升序进行排序，大多数标点符号放在最前面，然后是数字，接着是大写字母，最后是小写字母。例如，1506大于Happy，而Happy又大于happy。

3.2.2 “真”对于Perl意味着什么

到现在为止，我们已经介绍了“如果该表达式是真……”或者“……计算的结果为真……，”等情况，但是尚未介绍Perl认为的“真”的正式定义。关于什么是真，什么不是真，Perl有几个简短的规则，当你认真思考一下这些规则之后，就会认识到这些规则的意义。下面就是这些规则：

- 数字0为假。
- 空的字符串（“ ”）和字符串“0”为假。
- 未定义值`undef`为假。
- 其他东西均为真。

明白了吗？需要注意的另一个问题是：当你测试一个表达式，看它是真还是假时，该简化表达式，调用函数，使用运算符，数学算式简化为表达式，如此等等，然后转换成标量值，以便进行计算，确定它是真还是假。

请考虑这些规则，然后看一下表3-3。在查看答案之前，猜测一下该表达式是真还是假。

表3-3 真还是假的例子

表达式	真还是假
0	假。数字0为假
10	真。这是个非0数字，因此是真
9>8	真。关系运算符将根据你的期望返回真或假
-5+5	假。这个表达式计算后得出的结果是0，而0是假
0.00	假。这个数字是0的另一种表示法， <code>0x0</code> , <code>00</code> , <code>0b0</code> 和 <code>0e00</code> 也是一样
""	假。在上面介绍的规则中明确讲到这个表达式是假
""	真。引号中有一个空格，这意味着它们并不是完全空的
"0.00"	真。真奇怪！它已经是个字符串了，而不是“0”或“”。因此它是真
"00"	也是真，原因与“0.00”相同
"0.00" + 0	假。在这个表达式中，对 <code>0.00+0</code> 进行了计算，结果是0，因此这是假

到现在为止，我们只介绍了if语句的关系运算符。实际上你可以使用任何表达式，按照你想要使用的方法来计算它是真或假：

```
# The scalar variable $a is evaluated for true/false
if ($a) { ... }

# Checks the length of $b. If nonzero, the test is true.
if (length($b)) { .... }
```

`undef`这个值在Perl中是个特殊值。尚未设置的变量均拥有`undef`这个值，并且有些函数在运行失败时也返回`undef`。它不是0，也不是一个普通的标量值，它有几分特殊。当在一个if语句中测试为真时，`undef`总是计算为假。如果你试图使用数学表达式中的`undef`值时，它将被视为0。

使用尚未设置的变量，往往是程序出错的标志。如果你运行的程序激活了警告特性，那么在一个表达式中使用`undef`值或者将未定义的值传递给某个函数，就会使Perl发出一个警告，即Use of uninitialized value（使用了未经初始化的值）。

3.2.3 逻辑运算符

当你编写程序时，有时可能需要类似下面这样的某种代码，即如果`$x`是真，并且`$y`是真，那么执行这项操作，但是，如果`$z`是真，则不要执行该操作。可以将这个例子的代码编写成一系列的if语句，不过这并不是个出色的代码：

```
if ($x) {
    if ($y) {
        if ($z) {
            # do nothing
        } else {
            ...
        }
    }
}
```

下载

```
        print "All conditions met.\n";
    }
}
```

Perl拥有一套完整的运算符，可以将真和假的语句组合在一起，这些运算符称为逻辑运算符。表3-4显示了各个逻辑运算符。

表3-4 逻辑运算符一览表

运 算 符	替 代 名	举 例	分 析
&&	and	\$s && \$t	只有当\$s和\$t都是真时，才是真
		\$q and \$p	只有当\$q和\$p都是真时，才是真
	or	\$a \$b	如果\$a或\$b是真，则为真
		\$c or \$d	如果\$c或\$d是真，则为真
!	not	! \$m	如果\$m不是真，则为真
		not \$m	如果\$m不是真，则为真

使用表3-4中的运算符，可以将前面这个代码段改写得更加简洁，如下所示：

```
if ($x and $y and not $z ) {  
    print "All conditions met.\n";  
}
```

用逻辑运算符连接起来的表达式将自左向右进行计算，直到能够为整个表达式确定一个真或假的值。请看下面的代码段：

```
1: $a=0; $b=1; $c=2; $d="";
2:
3: if ($a and $b) { print '$a and $b are true'; }
4: if ($d or $b) { print 'either $d or $b is true'; }
5: if ($d or not $b or $c)
6:     { print '$d is true, or $b is false or $c is t
```

第1行：这一行代码为变量赋予一个默认值。

第3行：\$a首先被计算。它的结果是假，因此 and表达式不可能是真。\$b从来不计算，它也不必计算，因为表达式的真是在计算 \$a之后知道的。Print没有执行。

第4行：\$d首先被计算。它的计算结果是假。即使 \$d是假，该表达式仍然可能是真，因为它包含一个逻辑 or，因此下一步要观察 \$b。\$b的结果是真，因此该表达式是真，同时 print开始运行。

第5行：\$d首先被计算。它的结果是假。即使 \$d是假，该表达式可能仍然是真，正如第4行中的情况一样，因为它包含一个逻辑 or。接着，\$b的真（为1，因此是真）被求反，因此该表达式变成假。or语句的真尚不能确定，因此 \$c被计算。\$c的计算结果是真，因此整个表达式是真，print开始运行。

这个运行特性（即一旦确定表达式是真，立即停止逻辑表达式的计算）称为短路。整个特性可供Perl程序员用来借助逻辑运算符创建简单的流控制语句，而完全不使用 if语句：

```
$message="A and B are both true."  
($a and $b) or $message="A and B are not both true.;"
```

在上面这个例子中，如果 \$a或\$b中有一个是假，那么 or右边的表达式必须计算，并且消息被改变。如果 \$a和\$b都是真，那么 or必须是真，并且不必计算右边的表达式。整个表达式的真值根本不使用。这个例子使用 and和or运算符的短路副作用来操作\$message。



运算符 `||` 和 `or` 并不完全相同。它们的差别在于 `||` 的运行优先级要高于 `or`。这意味着在一个表达式中，`||` 要比 `or` 更早地进行计算。这与数学表达式中乘法的计算要先于加法的情况是一样的。这个规则也适用于 `&&/and` 和 `!/not`。如果你对计算顺序没有把握，可以使用括号来确保表达式的计算顺序的正确。

Perl的逻辑运算符有一个有趣的属性，那就是它们不仅仅返回真或假，它们实际上返回计算得出的最后值。例如，表达式 `5 && 7` 的计算结果并不只是返回真，而是返回 7。这样你就可以创建下面这个代码：

```
# Set $new to $old value if $old is true,
# otherwise use the string "default".
$new=$old || "default";
```

这比下面这个代码更加简洁：

```
$new=$old;
if (! $old) { # was $old empty (or false)?
    $new="default";
}
```

3.3 循环

在本学时的开头我们说过，仅仅根据条件来进行决策和运行代码是不够的。在许多情况下，需要一次又一次重复运行一段代码。程序清单 3-1 中显示的程序练习并没有太大的趣味，因为只是进行一次猜测，它是个毫无意义的游戏。如果你希望能够进行多次猜测，那么必须按照一定的条件重复执行一些代码段，这就是循环所要达到的目的。

3.3.1 用while进行循环

最简单的一种循环是 `while` 循环。只要表达式是真的，`while` 循环就会重复执行该代码段。`while` 循环的句法类似下面的形式：

```
while (expression) BLOCK
```

当Perl遇到 `while` 语句时，它就计算该条件。如果条件计算的结果是真，代码块就运行。当运行到代码块的结尾时，表达式被重新计算，如果结果仍然是真，代码块重复执行，如程序清单 3-2 所示：

程序清单 3-2 `while` 循环示例

```
1:  $counter=0;
2:  while ($counter < 10 ) {
3:      print "Still counting...$counter\n";
4:      $counter++;
5:  }
```

第1行：`$counter` 被初始化为 0。

第2行：表达式 `$counter < 10` 被计算。如果计算的结果是真，该语句块中的代码就运行。

第4行：`$counter` 的值递增 1。

第5行：花括号 } 给第2行上以 { 为开始的代码块做上结束标号。这时，Perl 返回到 `while` 循

环的顶部，并重新计算该条件表达式。

3.3.2 使用for循环

for语句是Perl循环结构中最复杂和最有用的语句。它的句法类似下面的形式：

```
for ( initialization; test; increment ) BLOCK
```

for语句分为3个部分，即initialization、test和increment，它们之间用分号隔开。当Perl遇到一个for循环时，便出现下面这个操作顺序：

- 初始化表达式被计算。
- 测试表达式被计算。如果它的计算结果的真，代码块就运行。
- 当该代码块执行结束后，便执行递增操作，并再次计算测试表达式。如果该测试表达式的计算结果仍然是真，那么代码块再次运行。这个进程将继续下去，直到测试表达式的计算结果变为假为止。

下面是for循环的一个例子：

```
for( $a=0; $a<10; $a=$a+2 ) {  
    print "a is now $a\n";  
}
```

在上面这个代码段中，\$a设置为0，执行测试表达式\$a<10，发现其结果为真。循环的本身输出了一条消息。然后递增语句\$a=\$a+2开始运行，它将\$a的值递增2。测试语句再次执行，循环重复运行。这个特殊的循环将重复运行，直到\$a的值达到10为止。这时测试语句变为假，for循环后面的程序将继续运行。

不必使用for语句来进行计数，它只是进行重复操作，直到测试表达式变为假为止。要记住，for语句的3个组成部分中的每一个都是可有可无的，但是两个分号是必不可少的。下面这个for语句漏掉了某些元素，不过它仍然是有效的：

```
$i=10;          # initialization  
for( ; $i>-1; ) {  
    print "$i..";  
    $i--;         # actually, a decrement.  
}  
print "Blast off!\n";
```

3.4 其他流控制工具

用循环和条件语句来控制你的程序运行方式是不错的，但是还需要其他的流控制语句，以便提高程序的可读性。例如，Perl有一些语句可以用来提前退出while循环，跳过for循环的某些部分，在代码块结束之前退出if语句，或者甚至在不到结束的时候就退出程序。使用本节中介绍的某些结构，就能够使你的Perl程序变得更加简洁和便于阅读。

3.4.1 奇特的执行顺序

if语句还可以使用另一种句法。如果在if语句块中只有一个表达式，那么该表达式实际上可以放在if语句的前面。因此不要写成下面这个语句：

```
if (test_expression) {  
    expression;  
}
```

可以写成：

```
expression if (test_expression);
```

下面是该语句变形的两个例子：

```
$correct=1 if ($guess == $question);
print "No pi for you!" if ( $ratio != 3.14159);
```

在Perl代码中使用该句法通常是为了清楚起见。有时，如果在条件之前看到它的作用，那么阅读代码就会更加容易。if前面的表达式必须是个单一表达式。if语句也必须后跟一个分号。

3.4.2 明细控制

除了使用语句块、for、while、if以及其他流控制语句来控制代码块以外，还可以使用Perl语句来控制语句块中的流程。

为你提供这种控制能力的最简单的语句是last。last语句能够使当前正在运行的最里面的循环块退出。请看下面这个例子：

```
while($i<15) {
    last if ($i==5);
    $i++;
}
```

last语句能够在\$i的值是5时使while循环退出，而不是在通常while测试的结果是假时退出。当你拥有多个嵌套的循环语句时，last将退出当前正在运行的循环。

程序清单3-3用于找出其乘积等于140的所有小于100的两个数，比如2与70、4与35等，不过查找的效率不太高。这里需要注意的是last语句。当找到一个乘积时，其结果被输出，里面的循环（在\$i上重复运行的循环）退出，外面的循环继续运行（通过递增\$i），并返回里面的循环。

程序清单3-3

```
1:   for($i=0; $i<100; $i++) {
2:     for($j=0; $j<100; $j++) {
3:       if ($i * $j == 140) {
4:         print "The product of $i and $j is 140\n";
5:         last;
6:       }
7:     }
8:   }
```

next语句使得控制权被重新传递给循环的顶部，同时下一个循环的重复运行则开始进行，如果该循环尚未结束的话：

```
for($i=0; $i<100; $i++) {
  next if (not $i % 2);
  print "An odd number=$i\n";
}
```

该循环将输出从0到98之间的所有偶数。如果\$i不是偶数，那么next语句将使该循环通过它的下一个迭代运行过程。表达式*\$i % 2*是*\$i*除以2的余数。在这个例子中，print语句被跳过了。编写这个循环时使用的一种更加有效的方法是按2这个值来递增*\$i*，但是这将无法展示next的作用，是不是？

3.4.3 标号

Perl允许你给语句块和某些循环语句（for、while）加上标号。也就是说，可以在语句块或语句的前面放置一个标识符：

```
MYBLOCK: {  
}
```

上面这个语句块的标号是MYBLOCK。标号名使用的约定与变量名基本相同，不过有一个很小的差别，那就是标号名不像变量，它不带%、\$和@之类的标识符。应该确保标号名与Perl的内置关键字不能冲突。就样式而言，如果标号名全部使用大写字母，那么这是最好的。不应该使它与目前和将来的Perl关键字的形式发生任何冲突。for和while语句也都可以带有标号。

```
OUTER: while($expr) {  
    INNER: while($expr) {  
        statement;  
    }  
}
```

last、redo和next语句都可以带有一个标号，作为参数。这样就可以退出一个特殊的语句块。在程序清单3-4中，使用一对嵌套的for循环，找到两个140的因子。如果想在找到一个因子后立即退出该循环，需要在两个循环之间对标志变量和if语句进行复杂的安排。问题是不能从内循环中退出外循环。

```
OUTER: for($i=0; $i<100; $i++) {  
    for($j=0; $j<100; $j++) {  
        if ($i * $j == 140) {  
            print "The product of $i and $j is 140\n";  
            last OUTER;  
        }  
    }  
}
```

现在，最后一个语句可以用来设定它想要退出哪个循环，在这个例子中，要退出的是OUTER循环。这个代码段只输出它找到的第一个因子。

3.4.4 退出Perl

exit语句是最后的一个流控制工具。当Perl遇到exit语句时，程序就停止执行，Perl将一个退出状态返回给操作系统。这个退出状态通常用来指明程序已经成功地完成运行。第11学时我们将要更加详细地介绍各种退出状态。现在，我们看到退出状态0意味着一切运行正常。下面是exit的一个例子：

```
if ($user_response eq 'quit') {  
    print "Good Bye!\n";  
    exit 0;          # Exit with a status of 0.  
}
```



exit语句具有某些对你的操作系统非常重要的副作用。当一个exit执行时，所有打开的文件均被关闭，文件锁被解开，Perl分配的内存被释放给系统，Perl解释程序执行清楚的关闭操作。

3.5 练习：查找质数

在这个练习中，我们将要观察一个小程序，它用来查找和输出质数。质数是只能被1和它

本身整除的数。例如，2是个质数，3也是个质数，而4不是质数（因为它可以被1、4和2整除），如此等等。质数的数量是无限的，它们需要占用大量的计算机功能来查找。

使用文本编辑器，键入程序清单 3-4 的程序，并将它保存为 Primes。不要键入行号。根据你在第 1 学时学习到的方法，使该程序成为可执行程序。

当你完成上面的操作时，键入下面这个命令行，设法使该程序启动运行：

Perl Primes

程序清单 3-4 用于查找质数的完整源代码

```

1:  #!/usr/bin/perl -w
2:
3:  $maxprimes=20;      # Stop when you've found this many
4:  $value=1;
5:  $count=0;
6:  while($count < $maxprimes) {
7:      $value++;
8:      $composite=0;
9:      OUTER: for ($i=2; $i<$value; $i++) {
10:         for($j=$i; $j<$value; $j++) {
11:             if (($j*$i)==$value) {
12:                 $composite=1;
13:                 last OUTER;
14:             }
15:         }
16:     }
17:     if (! $composite) {
18:         $count++;
19:         print "$value is prime\n";
20:     }
21: }
```

第1行：这一行包含到达解释程序的路径（可以修改该路径，使它适合你的系统的需要）和开关-w。请始终使警告特性处于激活状态。

第3行：\$maxprimes是你想要查找的质数的最大数量。

第4行：\$value是你将要测试其质数特性的值。

第5行：\$count是迄今为止找到的质数的数量。

第6行：只要程序没有找到足够数量的质数，while 循环就继续运行。

第7行：\$value被递增，因此，经过检查符合质数要求的第一个数是 2。

第8行：\$composite是for循环中使用的一个标志，用于指明找到的数是合数，不是质数。

第9~10行：for循环重复运行通过\$value的所有可能的因子。如果\$value是4，那么这些循环将产生2和2、2和3、3和3。

第11~14行：\$i与\$j的值相乘；如果乘积是\$value，那么\$value是个合数。\$composite标志被设置，同时，各个for循环均退出。

第17~20行：在for循环之后，检查\$composite标志。如果它是假，那么这数是质数。然后这些行输出一个消息，同时计数器的数字递增。



这里用来查找质数的算法其运行的速度并不特别快，效率也不高，但是它很好地展示了循环的运行情况。在更好的关于数值算法的著作中，你会找到更好的方法。

3.6 课时小结

在本学时中，我们介绍了 Perl 的许多流控制结构。有些结构，比如 if 和逻辑运算符，可以用于根据真或假的值来控制程序的各个部分是否运行。其他的结构，如 while、until 和 for，用于根据需要的次数循环运行代码段。我们还介绍了 Perl 的“真”究竟是什么概念，Perl 中的所有测试条件实际上都使用这个概念。

3.7 课外作业

3.7.1 专家答疑

问题：我熟悉另一种编程语言 C，它有一个 switch（或 case）语句。Perl 的 switch 语句在哪里？

解答：Perl 没有这个语句。Perl 提供了各种各样的测试方法，它确定 switch 语句的最佳句法是非常可怕的。下面是仿真 switch 语句的最简单的方法：

```
if ($variable_to_test == $value1) {  
    statement1;  
} elsif ($variable_to_test == $value2) {  
    statement2;  
} else {  
    default_statement;  
}
```

如果在命令行提示符后面键入 perldoc perlsyn，你会看到一个在线语法手册页，它包含了许多关于如何在 Perl 中仿真 switch 语句的出色方法，其中有些配有很像 switch 的句法。

问题：在 for（while、if）语句块中我能够嵌套多少个这样的语句块？

解答：如果你的系统的内存允许的话，嵌套的语句数目是不受限制的。但是，通常情况下，如果循环中嵌套的语句太多，那么就要使用不同的方法来处理这个问题。

问题：Perl 向我显示了一条消息，即 Unmatched right bracket 或 Missing right bracket（括号不匹配，或者括号遗漏）。报告的行号是文件的结尾。我该怎么办？

解答：在你的程序的某个位置，使用了左花括号（{），但是没有右括号，或者有了右括号，没有左括号。有时 Perl 能够猜到你的程序中的某个位置出现了键入错误，但是有时无法猜到这样的错误。由于控制结构可以人工嵌套许多层，因此，直到 Perl 到达文件的结尾但没有找到对应的括号时，它才知道你产生了键入错误。好的程序编辑器（如 vi、Emacs 或 UltraEdit）配有一个特性，可以帮助你查找不匹配的括号，你可以选择使用。

3.7.2 思考题

1) 只要条件为真，while 语句就始终循环运行。当条件是假时，什么语句将循环运行？

- a. if (not) {}
- b. while (! condition) {}

2) 下面的表达式是真还是假？

(0 and 5) || ("0" or 0 or "") and (6 and "Hello")) or 1

- a. 真
- b. 假

3) 下面这个循环运行之后，\$i的值是什么？

```
for ($i=0; $i<=10; $i++) { }
```

- a. 10
- b. 9
- c. 11

3.7.3 解答

1) 答案是b。while (! condition) {}语句将循环运行，直到条件变为假为止。

2) 答案是a。该表达式可以使用下面的步骤加以简化：

(假) || ((假) and (真)) or 真

假 || 真 or 真

真

3) 答案是c。测试表达式是 \$i<=10，因此当测试的条件最后变为假时，\$i必须是11。如果你在这个问题上出错，请不要担心。这是个非常常见的错误，在程序员中甚至有一个专门的名字来表示它，即篱笆桩错误或一步之差的错误。

3.7.4 实习

- 修改程序清单3-1，使游戏继续进行，直到做出一次正确的猜测。
- 程序清单3-4在查找质数时的效率实际上是非常低的。例如，它要分析2以上的所有偶数，而这些偶数不可能是质数。请对这种方法进行修改，使这个程序运行起来更加有效。

China-pub.com

下载

China-pub.com

下载

第4学时 基本构件的堆栈：列表与数组

标量是Perl的单数名词。它们可以代表任何一种元素，如单词、记录、文档、一行文本或者一个字符。但是，有时需要一些元素的集合，比如许多个单词、几个记录、两个文档、50行文本或者十几个字符等。

当需要谈论Perl中的许多东西时，可以使用列表数据。可以用3种方法来表示列表数据，它们是列表、数组和哈希结构。

列表是列表数据最简单的表示方法，它们只是一个标量的组合。有时它们使用一组括号将标量括起来，各个标量之间用逗号隔开。例如，(2, 5, \$a, "Bob")是两个数字，一个标量\$a和单词“Bob”的列表。列表中的每个项目称为列表元素。为了不违背自然随意的原则，Perl的列表可以根据你的需要包含任意多个元素。由于列表是标量的集合，并且标量也可以任意大，因此列表能够存放相当多的数据。

若要将一个列表存放在一个变量中，需要一个数组变量。在Perl中，数组变量用一个符号(@)后随一个有效的变量名(第1学时中的“数字与字符串”这一节做了介绍)来表示。例如，@FOO就是Perl中的一个有效的数组变量。数组变量可以与标量变量使用相同的名字，例如，\$names与@names可以指不同的东西，\$names指一个标量变量，而@names则指一个数组。这两个变量之间毫无关系。

数组中的各个项目称为数组元素。各个数组元素按它们在数组中的位置来引用，这个位置称为索引(比如说，数组@FOO的第三个元素，或者数组@names的第五个元素等等)。

另一种列表类型是哈希结构，它类似数组。哈希结构将在第7学时中详细介绍。

在本学时中，我们将要介绍：

- 如何填充和清空数组。
- 如果逐个元素查看数组。
- 如何对数组进行排序和输出。
- 如何将标量分割成数组，以及如何将数组重新合成为标量。

4.1 将数据放入列表和数组

将数据放入一个列表是非常容易的。正如你刚刚看到的那样，列表的语法是用一组括号将一些标量值括起来。下面就是列表的一个例子：

```
(5, 'apple', $x, 3.14159)
```

这个例子用于创建一个由4个元素组成的列表，它包含数字5、单词apple、标量变量\$x和值。如果列表只包含简单的字符串，而用单引号将每个字符串括起来对你来说又太麻烦，那么Perl提供了一个快捷方式，即qw运算符。下面是使用qw的一个例子：

```
qw (apples oranges 45.6$x)
```

这个例子创建了一个由4个元素组成的列表。列表的每个元素之间用一个白空间(空格、制表符或换行符)隔开。\$x是个直接量\$和x，它没有内插到它的值中去。如果有一些嵌入了

白空间的列表元素，那么就不能使用 qw运算符。在这种情况下，上面这个代码的作用就像编写的是下面这个代码一样：

```
('apples', 'oranges', '45.6' '$x')
```

请注意，\$x是用单引号括起来的。qw没有对看起来像变量的元素进行变量值内插，它们是作为常规的形式来处理的，因此'\$x'没有被转换成标量变量\$x的任何值，它只是留下了一个美元符号和字母x。

Perl有一个非常有用的能力对列表进行操作的运算符，称为范围运算符。范围运算符由一对圆点(..)来表示。下面是该运算符的用法的例子：

```
(1..10)
```

范围运算符用一个左边的操作数(1)和右边的操作数(10)构成了一个包含1到10(含1与10)之间的所有数的列表。如果需要在列表中使用若干个范围，那么只要使用多个范围运算符即可：

```
(1..10, 20..30);
```

上面这个例子创建了一个包含21个元素的列表，即包含1到10和20到30(含1、10、20和30)之间的数。如果范围运算符的右边的操作数小于左边的操作数，比如(10..1)，那么将产生一个空列表。范围运算符既可以用于字符串，也可以用于数字。范围(a..z)可以产生一个包含所有26个小写字母的列表。范围(aa..zz)可以生成一个大得多的列表，它由675个字母对组成，从aa、ab、ac、ad开始，到zx、zy、zz结束。

数组

直接量列表通常用于对某些其他结构进行初始化，比如数组或哈希结构。若要在Perl中创建一个数组，只需要将某些数据放入数组即可。Perl与其他编程语言不同，不必预先告诉Perl，你要创建一个数组，或者该数组将有多大。若要创建一个新数组，并用一个项目列表填入该数组，只要编写下面这个代码即可：

```
@boy=qw(Greg Peter Bobby);
```

这个例子称为数组赋值，它使用数组赋值运算符——等号，这与标量中的情况一样。当运行该代码后，数组@boys将包含3个元素，即Greg、Peter和Bobby。请注意，该代码也使用了qw运算符。使用该运算符后，你就不必键入6个引号和2个逗号。数组赋值也可以包含其他数组甚至空列表，如下面的例子所示：

```
@copy=@original;
@clean=();
```

在这里，@original数组的所有元素都被拷贝到新数组@copy中。如果@copy中原先已经拥有元素，那么这些元素就会丢失。这时@clean就变成空数组。将一个空列表(或者空数组)赋予一个数组变量，就会从该数组中删除所有的元素。

如果直接量列表中包含了其他列表、数组或哈希结构，那么这些列表将全部合并成一个大列表。请看下面这个代码段：

```
@girls=qw( Marcia Jan Cindy );
@kids=(@girls, @boys);
@family=(@kids, ('Mike', 'Carol'), 'Alice');
```

在将有关的值赋予 @kids 数组之前，列表（@girls，@boys）被 Perl 合并成一个由所有小孩名字（Greg，Peter 等等）组成的简单列表。在下一行上，数组 @kids 被合并，同时列表（‘Mike’，‘Carol’）被合并成一个长列表，然后该列表被赋予 @family。@boys、@girls、@kids 的原始结构和列表（‘Mike’，‘Carol’）本身则没有保留在 @family 中，保留的只是各个元素，即 Mike 和 Carol。

这意味着上面这个用于建立 @family 的代码段与下面这个赋值语句是等价的：

```
@family=qw(Greg peter Bobby Marcia Jan Cindy Mike Carol Alice);
```

如果赋值数组左边的列表只包含变量名，那么该列表可以用来对其元素进行初始化。请看下面这个例子：

```
($a, $b, $c)=qw (apples oranges bananas);
```

在这个例子中，\$a 被初始化为 ‘apples’，\$b 被初始化为 ‘oranges’ 而 \$c 则被初始化为 ‘bananas’。如果左边的列表包含一个数组，那么该数组就接受来自右边列表的剩余值。现在请看下面的例子：

```
($a, @fruit,$c) = qw (peaches mangoes grapes cherries);
```

在这个例子中，\$a 被设置为 ‘peaches’。右边列表中的其余水果被赋予左边的 @fruit。\$c 没有留下任何元素来接受一个值（因为赋值语句左边的数组吸收了来自右边的所有剩余值），因此\$c 被设置为 undef。

另一个值得注意的问题是：如果左边包含的变量比它拥有的元素多，那么多余的变量将接受 undef 这个值。如果右边的变量比左边的元素少，那么右边多余的元素将被忽略。下面让我们观察一个代码，以便理解这个概念：

```
($t, $u, $v) = qw (partridge robin cardinal quail);
($a, $b, $c, $d) = qw (squirrel woodchuck gopher);
```

在第一行中，\$t、\$u 和 \$v 均接受来自右边的值。右边多余的元素（‘quail’）将不用于该表达式。在第二行中，\$a、\$b 和 \$c 均接受来自右边的值。但是 \$d 没有从右边得到任何值（\$c 得到了最后一个值 ‘gopher’），因此\$c 被设置为 undef。

4.2 从数组中取出元素

到现在为止，我们在本学时中一直是在介绍整个数组和列表方面的内容。但是在许多情况下，需要获得数组中的各个元素，需要搜索数组，改变元素的值，或者将元素添加给数组或从数组中删除各个元素。

若要获得整个数组的内容，最简单的方法是使用双引号中的数组：

```
print "@array";
```

这个例子用于输出 @array 的元素，每个元素之间用空格隔开。数组中的各个元素可以按索引来访问，如下面这个代码所示。数组元素的索引从数字 0 开始，每增加一个元素，索引便递增 1。数组的每个元素都有一个索引值，如下所示：

@trees		0	1	2	3
		oak	cedar	maple	apple

数组中元素的数量只受系统内存的限制。若要访问一个元素，可以使用句法 \$array[index]，其中 array 是数组的名字，index 是你想要的元素的索引。在引用各个元素之前，数组不一定存在。数组会魔术般地自动弹出来。下面是访问数组元素的一些例子：

```
@trees=qw(oak cedar maple apple);
print $trees[0];           # Prints "oak"
print $trees[3];           # Prints "apple".
$trees[4]='pine';
```

请注意，如果要指 @trees 中的某个元素，该代码可以使用一个 \$。你可能会问：“\$ 标记通常是保留供标量使用的，这里是怎么回事呢？”在 \$trees[3] 中的 \$ 是指 @trees 中的一个标量值。标量也可以用美元符号来表示，因为它们也是单数。你应该注意这里的一个模式。

在本学时的开头，我们讲过标量和数组可以拥有相同的变量名，但是它们互不相关。Perl 能够说明 \$trees 与 @trees[0] 之间的差别，因为 \$trees[0] 中有一个方括号。Perl 知道你指的是 @trees 的第一个元素，而根本不是指 \$trees。

还可以将数组划分成分组，称为片。若要使用数组的一个片，可以使用 @ 标号，以指明你说的是一组元素，也可以使用方括号，以指明你说的是数组的各个元素，如下所示：

```
@trees=qw(oak cedar maple apple cherry pine peach fir);
@trees[3,4,6];           # Just the fruit trees
@conifers=@trees[5,7];   # Just the conifers
```

4.2.1 寻找结尾

有时需要寻找数组的结尾，例如，要查看 @trees 数组中究竟有多少树状结构，或者从 @trees 数组中砍下一些树枝来。Perl 提供了两个机制，可以用来查找数组的结尾。第一个方法是个特殊变量，其形式是 \$#arrayname。它能够返回数组的最后一个有效索引的号码。请看下面这个例子：

```
@trees=qw(oak cedar maple apple cherry pine peach fir);
print $#trees;
```

这个例子包含 8 个元素，但是你必须记住，数组是从 0 开始编号的。因此，上面这个例子输出的号码是 7。如果修改 \$#trees 的值，就会改变数组的长度。如果要缩小数组，请在你设定的某个索引处截断数组；如果要扩大数组，那么就给它增加更多的元素。新增加的元素的值将全部设置为 undef。

寻找数组大小的另一种方法是在期望存在标量的位置上使用数组变量：

```
$size=@array;
```

这将把 @array 中的元素数量放入 \$size 中。它利用了 Perl 的一个概念，称为上下文(环境)，这将在下一节中介绍。



也可以为数组设定负索引。负索引号从数组的结尾开始计数，然后反向递增。例如，\$array[-1] 是 @array 的最后一个元素，\$array[-2] 是倒数第二个元素，依次类推。

4.2.2 关于上下文的详细说明

什么是上下文呢？上下文是指有关项目周围的一些事物，这些事物可以用来帮助定义这个项目的含义。例如，你看到一个人穿着外科消毒服，根据他所在的地方，这可能表示不同的含义。如果是在医院，那么穿着这样的衣服的人是一名医生。如果是在万圣节上，那么他只是一位化装了的宾客。

人类的语言使用上下文来帮助确定词汇的含义。例如，单词 level可以拥有若干不同的含义，这要根据如何使用这个单词以及在什么上下文中使用它：

- 木工使用水平仪（level）使安装的门能够达到水平。
- 调解人说话的语调平和（level）。
- 池塘里的水只有齐腰深（level）。

虽然每次使用的单词都是 level，但是它的含义在不同的场合却发生了变化。根据它在句子中的不同用法，它可以是个名词或形容词，也可以是另一种类型的名词。

Perl能够对上下文作出敏感的反应。在 Perl中，函数和运算符能够根据使用时所在的上下文而具有不同的运行特性。Perl中的两个最重要的上下文是列表上下文和标量上下文。

如你所见，可以将赋值运算符（等号）用于数组和标量。赋值运算符左边的表达式类型（列表或标量）用于决定右边的表达式计算时所在的上下文。现在请看下面这个代码段：

```
$a=$b;      # Scalar on the left, this is scalar context.
@foo=@bar;   # Array on the left, this is list context
($a)=@foo;   # List on the left, this is also list context.
$b=@bar;     # Scalar on the left, this is scalar context.
```

该代码段的最后一行很有意思，因为在标量上下文中计算时，数组将返回该数组中元素的数量。

观察下面这几行代码中的 \$a和\$b，它们执行的操作几乎相同：

```
@foo=qw( water pepsi coke lemonade );
$a=@foo;
$b=$#foo;
print "$a\n";
print "$b\n";
```

在该代码的结尾，\$a包含数字4，\$b包含数字3。为什么会出现这个差别呢？因为 @foo是在标量上下文中运行的，它返回 \$a的元素的数量。\$b设置为最后一个元素的索引号，而索引是从0开始计数的。由于标量上下文中的数组返回数组中的元素的数量，因此测试数组中是否包含元素就变得如此简单：

```
@mydata=qw( oats peas beans barley );
if (@mydata) {
    print "The array has elements!\n";
}
```

在这里，数组 @mydata被计算为一个标量，它返回元素的数量，在这个例子中，这个数量是4。在if语句中，数量4计算的结果是真，if语句块的本身就开始运行。



实际上，@mydata在这里是用在一个特殊的标量上下文中，称为布尔上下文，不过它的运行特性是相同的。当 Perl希望得到一个真或假的值时，就会出现布尔上下文，比如在一个 if语句的测试表达式中。另一个上下文称为无效(void)上下文，我们将在第9学时中对该上下文进行介绍。

4.2.3 回顾以前的几个功能

Perl的许多运算符和函数能够强制它们的参数成为标量上下文或列表上下文。有时这些运算符和函数的运行特性将根据它们所在的上下文而各不相同。你已经遇到的有些函数具备这些属性，但是以前这个情况却并不重要。

`Print`函数希望有一个列表作为其参数，不过该列表在什么上下文中计算却并不特别重要。因此，用类似这个 `print` 的函数来输出数组将会导致该数组在列表上下文中被计算，从而产生 `@foo` 的各个元素：

```
print @foo;
```

可以使用一个称为 `scalar` 的特殊伪函数来强制将某个东西放入标量上下文：

```
print scalar (@foo);
```

这个例子用于输出 `@foo` 中的元素的数量。`Scalar` 函数强制 `@foo` 在一个标量上下文中进行计算，因此 `@foo` 返回 `@foo` 中的元素的数量。然后 `print` 函数就输出返回的数量。

你在第2学时中了解到的 `chomp` 函数既能够将数组作为参数，也能够将标量作为其参数。如果 `chomp` 函数获得一个标量，那么它就从标量的结尾处删除记录分隔符。如果它获得一个数组，它将从数组中的每个标量的结尾处删除记录分隔符。

我们在第2学时中介绍 `chomp` 时，还讲述了如何使用 `<STDIN>` 读取键盘输入的一行数据。尖括号实际上是 Perl 中的一个运算符，它们的运行特性随着上下文的不同而各有差异。在标量上下文中，该运算符能够读取来自终端的一行输入。但是在列表上下文中，它能够读取来自终端的所有输入，直到读到文件的结尾，并将数据放入列表。请看下面这个代码：

```
$a=<STDIN>; # Scalar context, reads one line into $a.  
@whole=<STDIN>; # List context, reads all input into the array @whole.  
($a)=<STDIN>; # List context, reads all input into the assignable list.
```

在第三个例子中，`$a` 将收到什么呢？本学时的开头我们讲过，在列表赋值语句中，如果左边没有足够的变量来存放右边的全部元素，那么右边的多余元素将被放弃。这里，来自终端的所有输入均被读取，但是 `$a` 只接收第一行。



什么是文件结尾呢？当 Perl 读取来自终端的全部输入且你完成 Perl 数据的输入时，你必须发出通知。为此通常键入一个 End of File（文件结束）字符（EOF）。该字符随着你使用的操作系统的不同而各有差别。在 UNIX 下，该字符通常是在一行的开头使用 `Ctrl+D`。在 MS_DOS 或者 Windows 系统上，该字符是在输入的任何位置两次使用 `Ctrl+Z`。

我们在第1学时中介绍的重复运算符在列表上下文中有一个特殊的运行特性。如果左边的操作数放在括号中，那么运算符本身将用在列表上下文中，它返回一个重复的左边操作数的列表。下面的例子用于建立一个 100 个星号的数组：

```
@stars= ("*") x 100;
```

x 左边的操作数 “*” 放在括号中，如果将它赋予一个数组，那么就使它进入一个列表上下文中。该句法可以用于将数组的元素初始化为一个特定的值。

另一个运算符你一直在使用，但是可能不知道它是个运算符，这就是逗号 (,)。到现在为止，你一直使用逗号来分隔列表中的各个元素，比如下面这个例子：

```
@pets=('cat', 'dog', 'fish', 'canary', 'iguana');
```

上面这个代码段是在通常的列表上下文中对列表进行计算操作的。但是，在标量上下文中使用的逗号是个运算符，它用于从左至右对每个元素进行计算，再返回最右边的元素的值：

```
$last_pet=('cat', 'dog', 'fish', 'canary', 'iguana'); # Not what you think!
```

在这个代码段中，赋值运算符右边的那些宠物名字实际上并不是一个列表。它们只是一组字符串直接量，从左到右进行计算，因为表达式的右边是在标量上下文中计算的（因为等号的右边有一个\$last_pet）。结果是该\$last_pet被设置为等于‘iguana’。

另一个例子是localtime函数，根据它所在的上下文，可以用两种完全不同的方法来运行。在标量上下文中，localtime函数返回一个格式化很好的当前时间字符串。例如，print scalar(localtime)这个代码，它输出的结果将类似于 Thu Sep 16 23:00:06 1999。在列表上下文中，localtime将返回能够描述当前时间的一个元素列表：

```
($sec, $min, $hour, $mday, $mon, $year_off, $wday, $yday, $isdst)=localtime;
```

表4-1 显示了这些值代表的含义。

表4-1 列表上下文中localtime返回的值

字 段	值
\$sec	秒，0 ~ 59
\$min	分，0 ~ 59
\$hour	时，0 ~ 23
\$mday	月份中的日期，1 ~ 28、29、30或31
\$mon	年份中的月份，0 ~ 11（这里请特别要小心）
\$year_off	1900年以来的年份。将1900加上这个数字，得出正确的4位数年份
\$wday	星期几，0 ~ 6
\$yday	一年中的第几天，0 ~ 364或365
\$isdst	如果夏令时有效，则为真



不要将‘19附加给localtime返回的年份。它返回的年份是1900的偏移量。比如，在1999年，年份是‘99’；在2000年中，它是‘100’。将1999与该值相加，可以在2000年以后正确地产生年份。Perl不存在2000年问题，但是如果简单地将‘19（或‘20）附加给该年份，就会导致程序中产生2000年问题。

怎么能够知道函数或运算符使它的参数在什么上下文中进行计算呢？又怎么知道它在标量上下文或列表上下文中运行的情况呢？很简单，你无法知道，你也没有什么好办法来猜测这个问题的答案。如果你不清楚的话，在线文档列出了每个函数和运算符，并且说明了它们的各个因子。在本书的其他章节中，如果一个函数或运算符强制让它的参数在某个上下文中运行，或者它们在自己运行的上下文中出现了完全不同的运行特性，那么我们将在首次介绍该函数时指明这些情况。这种情况不会经常发生，不过当本书中发生这种情况时，一定有特别说明。

4.3 对数组进行操作

既然你已经了解到创建数组的基本规则，那么现在我们就可以介绍一些工具来帮助你对

这些数组进行操作，以便执行一些有用的任务。

4.3.1 遍历数组

在第3学时中，我们介绍了如何使用 while、for和其他结构进行循环的方法。你想使用数组执行的许多操作都涉及到查看数组的每个元素，这个进程称为数组的迭代。迭代的方法之一是使用for循环，如下所示：

```
@flavors=qw( chocolate vanilla strawberry mint sherbert );
for($index=0; $index<@flavors; $index++) {
    print "My favorite flavor is $flavors[$index] and..."
}
print "many others.\n";
```

第一行代码用于以冰淇淋的风味对该数组进行初始化，为了清楚起见，它使用了qw运算符。如果使用两个单词组成的冰淇淋风味，比如 Rocky Road，那么将需要一个普通的单引号列表。第二行进行了代码的大部分工作。\$index被初始化为0，并且按1进行递增，直到到达@flavors。请记住，@flavors是在标量上下文中计算的，计算的结果是5，即@flavors中的元素的数量。

上面这个例子似乎要对数组的迭代进行大量的工作。在Perl中，如果需要进行大量的工作，那么通常可以找到一种比较简单的方法来进行这项操作，这没有例外。Perl还有另一个循环语句，称为foreach语句，我们在第3学时没有介绍。foreach语句设置一个索引变量，称为迭代器，它相当于列表的每个元素。请看下面这个例子：

```
foreach $cone (@flavors) {
    print "I'd like a cone of $cone\n";
}
```

在这个代码中，变量\$cone设置为@flavors中的各个值。当\$cone被设置为@flavors中的各个值时，循环体就开始执行，为@flavors中的每个值输出消息。请注意，在一个foreach循环中，迭代器并不只是设置为列表中的每个元素的值，它实际上是对列表的元素的引用。因此，在上面这个foreach循环中，如果修改该循环中的\$cone，就能修改@flavors中的对应元素。下面让我们来观察一下这个例子：

```
foreach $flavor (@flavors) {
    $flavor="$flavor ice cream"; # This modifies @flavors!
    print "I'd like a bowl of $flavor, please.";
```

第2行修改了\$flavor，将ice cream附加给了结尾处，因此第3行负责输出I'd like a bowl of chocolate ice cream，然后继续输出vanilla，strawberry等等。当该循环最后运行结束时，@flavors改为在每个元素的结尾处包含ice cream。



在Perl中，foreach和for循环语句实际上是同义语句，它们可以互换使用。为了清楚起见，在本书的整个篇幅中，你将发现使用foreach()循环语句在数组上进行迭代运行，并将for()循环语句用于像第3学时中那样的普通for循环。请记住它们是可以互换的。

4.3.2 在数组与标量之间进行转换

一般来说，Perl并没有关于在标量与数组之间进行转换的规则。我们提供了许多函数和运

算符，以便进行它们之间的转换。将标量转换成数组的方法之一是使用 split函数。Split函数拥有一个模式和一个标量，并且使用该模式来分割该标量。第一个参数是该模式（这里用斜杠括起来），第二个参数是要分割的标量：

```
@words=split(/ /, "The quick brown fox");
```

当运行该代码时，@words包含了各个单词，即The、quick、brown和fox，单词之间没有空格。如果要设定一个字符串，请使用变量\$_。如果没有设定一个模式或字符串，那么使用白空间来分割变量\$_。一个特殊模式''（即空模式）用于将标量分割成各个字符，如下面所示：

```
while(<STDIN>) {
    ($firstchar)=split(//, $_);
    print "The first character was $firstchar\n";
}
```

第一行用于读取来自终端的数据，每次读一行，并将\$_设置为等于该行。第二行使用空模式来分割\$_。Split函数返回来自\$_中的这个行的每个字符的列表。该列表被赋予左边的列表，而该列表的第一个元素则被赋予\$firstchar，其余均放弃。



split使用的模式实际上是一些正则表达式。正则表达式是一种复杂的模式匹配语言，我们将在第6学时介绍这方面的内容。而现在，我们的例子将使用一些简单的模式，如空格、冒号、逗号等等。当你学习了正则表达式的内容之后，我们将举例说明如何使用比较复杂的模式来用split分割标量。

在Perl中使用这种方法来分割标量变量的列表是非常常见的。当分割一个标量，而标量中的每个元素都是与众不同的元素（比如记录中的域）时，就能够比较容易地确定哪个元素是什么。请看下面这个例子：

```
@Music=('White Album, Beatles',
         'Graceland, Paul Simon',
         'A Boy Named Sue, Goo Goo Dolls');
foreach $record (@Music) {
    ($record_name, $artist)=split(//, $record);
}
```

当你直接分割带有命名称量的列表时，能够清楚地看到哪个域代表什么。第一个域是记录名，第二个域是艺术家。如果将代码分割成一个数组，各个域之间的区别也许不会那样清楚。

若要用数组来创建一个标量，也就是进行split的反向操作，可以使用Perl的join函数。join函数取出一个字符串和一个列表，使用该字符串将列表的各个元素组合在一起，然后返回产生的字符串。请看下面这个例子：

```
$numbers=join(',', (1..10));
```

这个例子将字符串1,2,3,4,5,6,7,8,9,10赋予\$numbers。然后可以使用split和join将字符串分割后又将它们重新组合在一起。一个函数的输出（返回值）可以用作另一个函数的输入值，请看下面的代码：

```
$message="Elvis was here";
print "The string \"\$message\" consists of:",
      join('-', split(//, $message));
```

在这个例子中，\$message被split分割成一个列表。该列表被join函数使用，并用逗号重新组合在一起。产生的结果是下面这个消息：

```
The string "Elvis was here" consists of: E-l-v-i-s- -w-a-s- -h-e-r-e
```

4.3.3 给数组重新排序

当你创建一个数组时，常常想让它们用不同于创建时的顺序来显示。例如，如果你的Perl程序从文件中读取一个客户列表，那么用字母顺序来输出该客户列表是可行的。若要给数据排序，Perl提供了sort函数。Sort函数将一个列表作为它的参数，并且大体上按照字母顺序对列表进行排序，然后该函数返回一个排定顺序的新列表。原始数组保持不变，如下面这个例子所示：

```
@chiefs=qw(Clinton Bush Reagan Carter Ford Nixon);
print join(' ', sort @chiefs);
```

这个例子用于输出Bush Carter Clinton Ford Nixon Reagan。应该预先注意的是，它的默认排序次序是ASCII顺序。这意味着以大写字母开头的所有单词均排在以小写字母开头的单词的前面。用ASCII顺序对数字进行排序的方式与你期望的不一样，它们不按值来排序。例如，11排在100的前面。在这种情况下，必须按非默认顺序来进行排序。

使用sort函数，你可以使用代码块（或者子例程名）作为第二个参数，按照你想要的任何顺序进行排序。在代码块（或者子例程）中，两个变量\$a和\$b被设置为列表的两个元素。代码块的任务是：如果\$b小于、等于或大于\$a，则分别返回-1、0或1。下面是进行数字排序的硬办法，假设@numbers是包含了许多数字值的话：

```
@sorted=sort { return(1) if ($a>$b);
               return(0) if ($a==$b);
               return(-1) if ($a<$b); } @numbers;
```

上面这个例子当然按照数字顺序给@numbers进行排序。但是，该代码从事这样一个普通的排序任务看起来太复杂了。由于你可能认为这个方法太麻烦，所以Perl有一个捷径可供使用，你可以使用飞船运算符<=>。飞船运算符因为从侧面看它像一个飞行的碟子而得名。如果它左边的操作数小于右边的操作数，那么它返回-1，如果左边的操作数大于右边的操作数，则返回0：

```
@sorted=sort { $a<=>$b; } @numbers;
```

这个代码看上去比较清楚，并且更加直观。飞船运算符只应该用来比较数字值。

若要比较字母字符串，请使用cmp运算符，它的运行方式与飞船运算符完全相同。只需要编写一个更加复杂的排序例程，就可以将更加复杂的排序参数放在一起。如果你需要了解更多的情况，本学时第7节中的Perl常见问题给出了一些比较复杂的例子。

本学时要介绍的最后一个函数是个非常容易使用的函数，即reverse。当在标量上下文中被赋予一个标量值时，reverse函数能够对字符串的字符进行倒序操作，返回倒序后的字符串。例如，在标量上下文中调用reverse("Perl")将返回lrep。当在列表上下文中被赋予一个列表时，reverse函数能够返回倒序后的列表元素，如下面的例子所示：

```
@lines=qw(I do not like green eggs and ham);
print join(' ', reverse @lines);
```

这个代码段用于输出ham and eggs green like not do I。若要继续进行这个试验，充分展示该函数堆栈的功能，可以给混合列表添加更多的奇怪内容：

```
print join(' ', reverse $@tlines);
```

该代码首先运行 sort 函数，产生一个莫名其妙的列表 (I , and , do , eggs , green , ham , like , not)。该列表被倒序后再被传递给 join 函数，以便将这些元素连接起来，并且加上一个空格。结果是 not like ham green egg do and I ，真不敢恭维这样的句子。

4.4 练习：做一个小游戏

本学时充满了许多新鲜的内容，比如，你熟悉的运算符在不同的上下文中产生不同的运行特性，一些新的函数和运算符，还有一些需要记住的新的句法规则。为了使你不至于编写出难以运行的代码，所以增加了这个练习，让你做一个游戏，使你具备的关于数组和列表的知识能够得到很好的应用。

使用文本编辑器，将程序清单 4-1 中的程序键入编辑器，并将它保存为 Hangman。务必按照第 1 学时中的说明使该程序成为可执行程序。

当完成上面的操作后，键入下面这个命令，使该程序运行：

```
perl Hangman
```

程序清单 4-1 完整的 Hangman 程序清单

```
1:  #!/usr/bin/perl -w
2:
3:  @words=qw( internet answers printer program );
4:  @guesses=();
5:  $wrong=0;
6:
7:  $choice=$words[rand @words];
8:  $hangman="0-|---<";
9:
10: @letters=split(//, $choice);
11: @hangman=split(//, $hangman);
12: @blankword=(0) x scalar(@hangman);
13: OUTER:
14:     while ($wrong<@hangman) {
15:         foreach $i (0..$#letters) {
16:             if ($blankword[$i]) {
17:                 print $blankword[$i];
18:             } else {
19:                 print "-";
20:             }
21:         }
22:         print "\n";
23:         if ($wrong) {
24:             print @hangman[0..$wrong-1]
25:         }
26:         print "\n Your Guess: ";
27:         $guess=<STDIN>; chomp $guess;
28:         foreach(@guesses) {
29:             next OUTER if ($_ eq $guess);
30:         }
31:
32:         $right=0;
33:         for ($i=0; $i<@letters; $i++) {
34:             if ($letters[$i] eq $guess) {
35:                 $blankword[$i]=$guess;
36:                 $right=1;
```

```

37:           }
38:       }
39:       $wrong++ unless(not $right);
40:       if (join('', @blankword) eq $choice) {
41:           print "You got it right!\n";
42:           exit;
43:       }
44:   }
45:   print "$hangman\nSorry, the word was $choice.\n";

```

第1行：包含到达解释程序的路径（可以修改这个路径，使它适合你的系统的需要）和开关-w。请始终使警告特性处于激活状态！

第3行：数组@words使用游戏能够使用的单词列表进行初始化。

第4~5行：有些变量被初始化。@guesses用于存放游戏的玩主过去猜测的所有单词的列表。@wrong用于存放迄今为止猜错的单词的数量。

第7行：从数组@words中随机选择的一个单词，并将它赋予\$choice。rand()函数期望得到一个标量参数，由于@words被视为一个标量，因此它返回元素的数量（在这里返回4），rand函数返回一个0至3之间的一个数字，但是不包括0或4。结果，当将一个十进制数字用作数组索引时，小数部分将被放弃。

第8行：hangman被定义。他很难看，但是通过了。

第10行：\$choice中的mystery单词在@letters中被分割成各个字母。

第11行：hangman标量在@hangman中被分割成小块。头是\$hangman[0]，颈部是\$hangman[1]，等等。

第12行：数组@blankword用于标明游戏的玩主猜对了哪些字母。（0）x scalar(@hangman)创建了一个列表，其长度与@hangman中的元素的数量相同，它被存放在@blankword中。当猜字母时，这些0改变为第35行中的字母，这将标出猜对的字母的位置。

第13~14行：建立一个包含大部分程序的循环。它有一个标识符OUTER，这样，在循环的内部，就能够对它进行某些具体的控制。它不断进行循环，直到猜错的字母数量与hangman的长度相同为止。

第15~21行：这个foreach循环为猜测的每个字母在数组@blankword上迭代运行。如果@blankword不包含该特定元素中的一个字母，那么就输出一个破折号，否则，输出该字母。

第23~25行：\$wrong包含猜错的字母的数量。如果这个数量至少是1，那么第24行就使用一个程序片来输出hangman数组，从位置0开始，直到猜错的字母的数量（减1）。

第26~27行：这两行用于获得游戏玩主的猜测。chomp()删除结尾处的换行符。

第28~30行：这几行用于搜索@guesses，以了解玩主是否已经猜到该字母。如果他已经猜到，就可以重新启动第13行的循环。如果玩主多次猜错，他不会受到处罚。

第32~38行：这是该程序的核心！搜索包含猜字母游戏的@letters数组。如果在游戏中找到了猜测的字母，那么对应的@blankword元素就被设置为该字母。数组@blankword既包含猜对的字母，也包含任何特定元素中的undef。称为\$right的标志被设置为1，表示至少有一个字母已经被找到。

第39行：除非游戏的玩主猜到了一个字母，否则\$wrong被递增。

第40~43行：数组@blankword的各个元素被连接在一起，构成一个字符串，并与原始的要猜测的字符串相比较。如果它们完全相同，表示游戏的玩主猜到了所有的字母。

第45行：玩主没有能够猜对字母，解释程序放弃了从第 13行开始的循环。这一行输出一条安慰玩主的消息，然后退出游戏。

程序清单4-2显示了Hangman程序的输出的一个示例。

程序清单4-2 Hangman程序的输出示例

```
Your Guess: t
----t-
Your Guess: s
----t-
o
Your Guess: e
----te-
o
Your Guess:
```

这个游戏对本学时中介绍的大部分概念进行了操作练习，这些概念包括直接量列表、数组、split、join、上下文和foreach循环等。你可以使用各种各样的方法来实现这个小型Hangman游戏程序，不过希望你能够掌握数组具备的一些基本功能。

4.5 课时小结

数组和列表是Perl的集合变量。你可以使用它们来存放数量不受限制的标量，并且既可以让它们进行整体操作，也可以将它们作为单个元素来进行操作。Perl提供了许多使用非常方便的机制，可以用来对数组进行拷贝、排序、组合，以及在标量与数组的数据之间来回进行转换。另外，Perl的许多运算符和函数对它们所在的上下文是非常敏感的，它们的运行特性将根据它们是在标量上下文中还是列表上下文中而各有不同。

4.6 课外作业

4.6.1 专家答疑

问题：你能告诉我一种在数组元素中快速找到特定字符串的方法吗？

解答：迭代运行该数组，并查看每个元素，这是进行这项操作的常用方法。如果你经常发现你正在搜索一个数组，以了解某个元素是否在该数组中，那么不要首先将数据存储在一个数组中。随机访问元素的一种更加有效的结构是哈希结构，这将在第 7学时中介绍。

问题：如何消除数组中的重复元素？如何计算数组中的各个元素的数目？如何来了解两个数组是包含相同的还是不同的元素？

解答：这几个问题的答案都是一样的，那就是使用哈希结构。使用哈希结构，你就能够非常迅速而有效地对数组进行某些有意思的操作。所有这些问题将在第 7学时中回答。

4.6.2 思考题

1) 如果要将\$a和\$b这两个标量变量中包含的值进行交换，哪个方法最有效？

- a. \$a=\$b ;
- b. (\$a , \$b) = (\$b , \$a) ;
- c. \$c=\$a ; \$a=\$b ; \$b=\$c ;

2) 语句 \$a=scalar (@array) ; 将什么赋值给变量 \$a ?

- a. @array 中的元素的数量;
- b. @array 的最后一个元素的索引;
- c. 该语句无效。

4.6.3 解答

1) 答案是 b。第一个选择显然不能成立，因为 \$a 中包含的值将被撤消。c 这个选择虽然回答了这个问题，但是它要求数据交换时用第三个变量来存储数据。b 这个选择能够正确地进行数据的交换，它不使用额外的变量，并且是个非常清楚的代码。

2) 答案是 a。使用标量上下文中的数组能够返回数组中的元素数量。\$#array 将返回数组的最后一个索引。在这个例子中使用 scalar() 是不必要的，在赋值运算符的左边有一个标量，就足以将 @array 放入一个标量上下文中了。

4.6.4 实习

- 修改 Hangman 游戏程序，以便用竖向位置来输出 hangman。

China-pub.com

下载

China-pub.com

下载

第5学时 进行文件操作

到现在为止，我们介绍的 Perl 程序都是独立的程序。除了向用户提供消息和接收来自键盘的输入信息外，它们无法与外界进行通信。这种状况将要改变。

Perl 是一种能够进行文件输入和输出（文件 I/O）的非常出色的语言。Perl 的标量能够延伸，以便将尽可能长的记录存放在文件中，另外，Perl 的数组能够扩展，以便存放文件的全部内容，当然这必须是在内存允许的情况下才能做到。当数据包含在 Perl 的标量和数组中时，可以对该数据进行不受限制的操作，并且可以编写新的文件。

当你读取数据或者将数据写入文件时，Perl 总是尽量设法不妨碍你的操作。在某些地方，Perl 的内置语句甚至进行了优化，以便执行常用类型的 I/O 操作。

在本学时中，我们将要介绍 Perl 怎样使你能够访问文件中你可以使用的所有数据。

在本学时中，你将要学习：

- 如何打开和关闭文件。
- 如果将数据写入文件。
- 如何从文件中读取数据。
- 如何使你编写的 Perl 程序具备保护功能，从而使其更加强大。

5.1 打开文件

若要在 Perl 中读取文件或写文件，必须打开一个文件句柄。Perl 中的文件句柄实际上是另一种类型的变量，它们可以作为在你的程序与操作系统之间对某个特定文件使用的非常方便的一个引用（即句柄，如果你愿意这样说的话）。句柄包含了关于如何打开文件和你在文件中读（或写）到了什么位置等信息。它们还包含了用户定义的关于如何读写文件的属性。

在前面的课程中，你已经熟悉了一个句柄，即 STDIN。该句柄是在启动程序时 Perl 自动赋予你的，它通常与键盘设备相连接（后面还要更加详细地介绍 STDIN）。文件句柄名字的格式与第 2 学时介绍的变量名基本相同，不同之处是句柄的名字前面没有类型标识符（\$、@）。由于这个原因，句柄名字最好使用大写字母，这样就不会与 Perl 的当前和将来的保留字 foreach、else 和 if 等发生冲突。



也可以将字符串标量或能够返回字符串的任何东西（如函数）用作文件句柄名。这种类型的句柄名称为间接句柄。描述它们的用法会给 Perl 的初学者造成一些混乱。关于间接句柄的详细说明，请参见 perlfunc 手册页中关于 open 的在线文档。

每当需要访问磁盘上的文件时，必须创建一个新的文件句柄，并且打开该文件句柄，进行相应的准备。当然必须使用 open 函数来打开文件句柄。Open 的句法如下：

```
open (filehandle, pathname)
```

open 函数将文件句柄作为它的第一个参数，将路径名作为第二个参数。路径名用于指明

要打开哪个文件，因此，如果没有设定完整的路径名，比如 `c:/windows/system/`，那么 `open` 函数将设法打开当前目录中的文件。如果 `open` 函数运行成功，它将返回一个非 0 值。如果 `open` 函数运行失败，它返回 `undef`（假）：

```
if (open(MYFILE, "mydatafile")) {
    # Run this if the open succeeds
} else {
    print "Cannot open mydatafile!\n";
    exit 1;
}
```

在上面这个代码段中，如果 `open` 运行成功，它计算得出的值是真，而 `if` 代码块则用打开的文件句柄 `MYFILE` 来运行。否则，文件不能打开，代码的 `else` 部分开始运行，这表示出现了错误。在许多 Perl 程序中，这个“打开或失败”语句是使用 `die` 函数来编写的。`die` 函数用于停止 Perl 程序的执行，并且输出下面这个出错消息：

`Died at scriptname line xxx`

在这个消息中，`scriptname` 是 Perl 程序的名字，`xxx` 是遇到 `die` 的行号。`die` 和 `open` 这两个函数常常以下面的形式同时出现：

```
open(MYTEXT, "novel.txt") || die;
```

这一行代码可以读作“打开或撤销”，它有时表示你想要让程序如何处理没有打开的文件。如果 `open` 运行没有成功，也就是说它返回 `FALSE`，那么逻辑 OR (`||`) 必须计算右边的参数 (`die`)；如果 `open` 运行成功了，也就是说它返回 `TRUE`，那么就不要计算 `die` 的值。这个习惯用语也可以用逻辑 OR 的另一个符号 `or` 来书写。

当你完成文件句柄的操作后，将文件句柄关闭，这是个很好的编程习惯。关闭文件句柄的操作，将通知操作系统说，该文件句柄可以重复使用，同时，尚未为文件句柄写入的数据现在可以写入磁盘。另外，你的操作系统只允许打开规定数量的文件句柄，如果超过这个数量，你就不能打开更多的文件句柄，除非关闭某些句柄。若要关闭文件句柄，可以使用下面这个 `close` 函数：

```
close(MYTEXT);
```

如果文件句柄名字重复使用，即另一个文件用相同的文件句柄名字打开，那么原始文件句柄将先被关闭，然后重新打开。

5.1.1 路径名

到现在为止，我们只是用类似 `novel.txt` 的名字来打开文件。当试图打开没有设定目录名的文件名时，Perl 假定该文件是在当前目录中。若要打开位于另一个目录中的文件，必须使用路径名。路径名用于描述 Perl 为了打开系统中的文件而必须使用的路径。

若要设定路径名，可以使用你的操作系统期望的方式来设定，如下面这个例子所示：

```
open(MYFILE, "DISK5:[USER.PIERCE.NOVEL]") || die;      # VMS
open(MYFILE, "Drive:folder:file") || die;                # Macintosh
open(MYFILE, "/usr/pierce/novel") || die;                # Unix.
```

在 Windows 和 MS-DOS 系统下，设定 Perl 中的路径名时可以使用反斜杠作为路径名分隔符，比如 `\Windows\user\pierce\novel.txt`。需要注意的唯一的问题是：当在带有双引号字符串中使用反斜杠分隔符路径名时，反斜杠字符序列将被转换成一个特殊的字符。请看下面的例子：

```
open(MYFILE, "\Windows\users\pierce\novel.txt") || die; # WRONG
```

这个例子可能运行失败，因为双引号字符串中的 \n 是个换行符，而不是字母 n，而且其他的所有反斜杠将被 Perl 悄悄地删除。下面是打开文件的正确方法：

```
open(MYFILE, "C:\\Windows\\users\\pierce\\novel.txt") || die; # Right, but messy.
```

为了使这行代码看起来更好一些，也可以使用 Windows 和 MS-DOS 中的正斜杠 (/) 来分隔路径中的各个元素。Windows 和 DOS 能够正确地对它们进行转换，请看下面的代码：

```
open(MYFILE, "C:/Windows/users/pierce/novel.txt") || die; # Much nicer
```

你设定的路径名可以是绝对路径名，例如 UNIX 中的 /home/foo 或 Windows 中的 c:/windows/win.ini，也可以是相对路径名，如 UNIX 中的 .. /junkfile 或 Windows 中的 .. /bobdir/bobsfile.txt。open 函数也能够接受 Microsoft Windows 下的通用命名约定（UNC）路径名。UNC 路径名的格式如下：

```
\\\machinename\sharename
```

Perl 能够接受使用反斜杠或正斜杠的 UNC 路径名，如果你的操作系统的网络和文件共享特性的设置正确的话，它能打开远程系统上的文件，请看下面的例子：

```
open(REMOTE, "//fileserver/common/foofile") || die;
```

在 Macintosh 计算机上，路径名是按卷、文件夹，然后文件这样的顺序来设定的，各个元素之间用冒号分开，如表 5-1 所示。

表 5-1 MacPerl 路径名的说明符

Macintosh 路径	含 义
System:Utils:config	系统驱动器，文件夹实用程序，文件名 config
MyStuff:friends	从该文件夹向下到文件夹 MyStuff，文件名 friends
ShoppingList	该驱动器，该文件夹，文件名 ShoppingList

5.1.2 出色的防错措施

在计算机上进行编程肯定会产生一种自我陶醉的感觉。程序员会说：“这次程序一定能够运行，”或者说：“我已经找到了所有的错误。”你在工作中产生这种自豪感在某种意义上讲当然是很好的，对程序的改进往往是出于这样一种信念，即你能够做到似乎不可能的事情。不过，你在编程过程中要切忌盲目自信。



自从计算机最初使用以来，人们就产生了这种看法。Fredrick P.Brooks 曾经在他的经典著作“ Mythical Man-Month ”中说：“所有程序员都是乐观主义者。但是，由于我们的思路常常出错，因此在编程的时候总是会犯错误，所以我们的乐观主义是没有理由的。”

到现在为止，你看到的所有代码段和程序练习都是用来处理内部数据（对数字进行因子分解，对数据进行排序等等）的，用户输入的信息很简单。当你进行文件的处理时，你的程序就要与外部信息源进行交流，而这些程序是无法控制外部信息源的。当你与不是在你的计算机上的数据源（比如网络上的数据）进行通信时，情况也是如此。请记住，如果某个地方可能出现错误，那么它就可能出错，因此你应该根据情况来编写程序。用这种方法来编写程序称为防错性编程，如果你进行防错编程，那么从长远来说你会感到更加乐观。

每当程序要与外界进行交互操作时，比如打开一个文件句柄，始终必须确保操作成功之后才能进行下一步操作。有人调试了上百个程序，在这些程序中，程序员要求操作系统执行某项操作，但是却不检查结果，因此就导致程序的错误。甚至当你的程序只是个“程序示例”，或者是个“简易程序”时，也应该进行程序检查，以确保程序能够产生你期望的结果。

5.1.3 以适当的方式运行die函数

在Perl中，die函数可以用来在出现错误的时候停止解释程序的运行，并输出一条有意义的出错消息。正如你在前面已经看到的那样，只要调用 die函数，就能够输出类似下面的消息：

```
Died at scriptname line xxx
```

die函数也可以带有一系列的参数，这些参数将取代默认消息而被输出。如果消息的后面没有换行符，那么消息的结尾就附有 at scriptname line xxx字样：

```
die "Cannot open";      # prints "Cannot open at scriptname line xxx"
die "Cannot open\n";    # prints "Cannot open"
```

Perl中有一个特殊的变量\$!，它总是设置为系统需要的最后一个操作（比如磁盘输入或输出）的出错消息。当\$!用于数字上下文时，它返回一个错误号，这个号可能对任何人都没有什么用处。在字符串上下文中，\$!返回来自你的操作系统的相应的出错消息：

```
open(MYFILE, "myfile") || die "Cannot open myfile: $!\n";
```

如果由于文件不存在，上面这个代码段中的代码运行失败，那么输出的消息将类似 cannot open myfile : a file or directory in the path does not exist，这个出错消息很好。在你的程序中，好的出错消息应该能够指明什么错了，为什么出错，你想怎么办。如果程序的某个方面出错了，那么好的诊断程序能够帮助你找出问题的所在。



不要使用\$!的值来检查系统函数的运行是失败还是成功。只有当系统执行一项操作（比如文件输入或输出）之后，\$!才有意义，并且只有在该操作运行失败后，\$!才被设置。在其他时间中，\$!的值几乎可以是任何东西，并且是毫无意义的。

不过有时并不想使程序停止运行，只是想要发出一个警告。若要创建这样的警告，Perl有一个warn函数可供使用。warn的运行方式与die完全一样，你可以从下面这个代码中看出来，不过差别是它的程序将保持运行状态：

```
if (! open(MYFILE, "output")) {
    warn "cannot read output: $!";
} else {
    :    # Reading output...
}
```

5.2 读取文件

可以使用两种不同的方法读取Perl的文件句柄。最常用的方法是使用文件输入运算符，也叫做尖括号运算符(<>)。若要读取文件句柄，只需要将文件句柄放入尖括号运算符中，并将该值赋予一个变量：

```
open(MYFILE, "myfile") || die "Can't open myfile: $!";
$line=<MYFILE>;           # Reading the filehandle
```

下载

标量变量中的尖括号运算符能够读取来自文件的一行输入。当该文件被读完时，尖括号返回值`undef`。



“一行输入”通常是指发现第一个行尾序列之前的文本流。在 UNIX中，行尾是一个换行符（ASCII 10）；在DOS和Windows中，它是回车符与换行符的序列（ASCII 13、10）。这个默认行尾值可以被Perl进行操作，产生某些有趣的结果。这个问题将在第12学时中介绍。

若要读取和输出整个文件，那么如果`MYFILE`是个开放的文件句柄，你可以使用下面的代码：

```
while (defined ($a=<MYFILE>)) {  
    print $a;  
}
```

结果表明，读取文件句柄的快捷方式是使用`while`循环。如果尖括号运算符是`while`循环的条件表达式中的唯一元素，那么Perl将自动把输入行赋予该特殊变量`$_`，并重复运行该循环，直到输入耗尽为止：

```
while (<MYFILE>) {  
    print $_;  
}
```

`while`循环将负责把输入行赋予`$_`，并确保文件句柄尚未耗尽（称为文件结束）。这个奇特的运行特性只有在`while`循环中才能发生，并且只有在尖括号运算符是条件表达式中的唯一表达式时才能发生。



请记住，在Perl中用文件句柄读取的所有数据中，除了包含文件行中的文本外，还包含行尾字符。如果只需要文本，请在输入行上使用`chomp`，以便舍弃行尾字符。

在列表上下文中，尖括号运算符能够读取整个文件，并将它赋予该列表。文件的每一行被赋予列表或数组的每个元素，如下所示：

```
open(MYFILE, "novel.txt") || die "$!";  
@contents=<MYFILE>;  
close(MYFILE);
```

在上面这个代码段中，文件句柄`MYFILE`中的剩余数据也被读取并赋予`@contents`。文件`novel.txt`的第一行被赋予`@contents:$contents[0]`的第一个元素。第二行被赋予`$content[1]`，如此等等。

在大多数情况下，将整个文件读入一个数组（如果文件不是太大），是处理文件数据的一种非常容易的方法。你可以来回通过该数组，对数组元素进行操作，并可使用数组和标量的所有运算符来处理该数组的内容，而不必担心会出现什么问题，因为你实际上只是对数组中的一个文件拷贝进行操作。程序清单5-1显示了对内存中的文件可能进行的某些操作。

程序清单5-1 对文件进行倒序

```
1:  #!/usr/bin/perl -w  
2:  
3:  open(MYFILE, "testfile") || die "opening testfile: $!";  
4:  @stuff=<MYFILE>;
```

```

5:    close(MYFILE);
6:    # Actually, any manipulation can be done now.
7:    foreach(reverse(@stuff)) {
8:        print scalar(reverse($_));
9:    }

```

如果该文件的测试文件包含文本 I am the very model of a modern major-general, 那么程序清单5-1中的程序将产生下面的输出：

```
.lareneg-rojam nredom a
fo ledom yrev eht ma I
```

第1行：这一行包含到达解释程序的路径（可以修改它，使之适合你的系统的需要）和开关-w。请始终使警告特性处于激活状态！

第3行：使用文件句柄 FH 打开文件的测试文件。如果该文件没有正确地打开， die 函数便开始运行，并产生一个出错消息。

第4行：测试文件的整个内容被读入数组 @stuff。

第7行：数组 @stuff 被倒序，第1行变成最后一行，依次类推。 foreach 语句遍历产生的列表。倒序后的列表的每一行被赋予 \$_，同时 foreach 循环体被执行。

第8行：列表的每一行（现在位于 \$_ 中）本身也进行倒序，由从左到右，变为从右到左，并且输出倒序后的每一行。它需要使用 scalar 函数，因为 print 期望一个列表。另外，在列表上下文中， reverse 用于对列表进行倒序，这样， \$_ 就不会发生任何问题。 scalar 函数将强制 reverse 进入标量上下文中，同时它按逐个字符对 \$_ 进行倒序。

只有对小型文件，才能将整个文件读入数组变量，以便进行操作处理。如果要将非常大的文件读入内存，虽然是允许的，但是可能导致 Perl 占用系统中的全部可用内存。

如果将太大的文件读入内存，或者进行其他的一些操作，从而使你占用的内存超出了 Perl 能够使用的内存，Perl 就会显示下面这条出错消息：

```
Out of memory!
```

这时你的程序就会终止运行。如果当一次性地将整个文件读入内存时出现了这种情况，你应该考虑每次读入文件的一行。

5.3 写入文件

若要将数据写入文件，首先必须有一个打开的文件句柄，以便进行写入操作。打开一个文件以便进行写入操作的句法如下：

```
open(filehandle, ">pathname")
open(filehandle, ">> pathname")
```

第一个语句行你应该是熟悉的，不过在路径名的前面有一个 >。> 符号用于告诉 Perl， pathname 设定的文件应该用新数据改写，而现有的任何数据都应该删除，同时 filehandle 是打开的，可以用于写入。在第二个例子中， >> 告诉 Perl 打开该文件，以便进行写入操作，但是，如果文件存在，那么将数据附加到该文件的结尾处。现在请看下面这些例子：

```
# Overwrite existing data, if any
open(NEWFH, ">output.txt") || die "Opening output.txt: $!";
# Simply append to whatever data may be there.
open(APPFH, ">>logfile.txt") || die "Opening logfile.txt: $!";
```



到现在为止，你的 Perl 程序几乎还不可能对任何东西造成损害。现在，你已经知道如何将数据写入文件，那么必须非常注意只能对你想要修改的文件进行写入操作。在操作系统文件非常容易受到损坏的系统（Windows 95/98、Mac）上，如果不小心将数据写入文件，那么就会损坏你的操作系统。你应该非常清楚要将数据写入什么文件。要想对不小心用 > 打开的文件中的数据进行还原，那几乎是不可能的。想要清除你不小心用 >> 打开的文件中的数据，那也是困难的，因此你要格外小心。

当完成了对打开的文件句柄的写入操作后，关闭该文件句柄是特别重要的。当你对文件进行写入操作时，操作系统并不将数据存放到磁盘，它只是将数据放入缓存，然后随时进行写入操作。close 函数通知操作系统说，你已经完成写入操作，数据应该移到磁盘上的永久性存储器中：

```
close(NEWFH);
close(APPFH);
```

当你打开了用于写入操作的文件句柄时，将数据写入文件实际上是非常容易的，因为你已经熟悉 print 函数。到现在为止，你一直使用 print 函数将数据显示在屏幕上。print 函数实际上可以用于将数据写入任何文件句柄。将数据写入文件句柄的句法如下：

```
print filehandle LIST
```

filehandle 是要将数据写入到的文件句柄，LIST 是要写入的数据的列表。

在 print 语句中，请注意文件句柄名与列表之间没有逗号，这一点很重要。在列表中，逗号用于分隔各个项目，迄今为止一直是这样做的。如果在文件句柄与列表之间没有逗号，那么 Perl 就会知道 print 后面的标记是个文件句柄，而不是列表中的文件元素。如果忘记使用这个逗号，但是打开了 Perl 的警告特性，那么 Perl 就会向你发出下面这条警告消息：No comma allowed after filehandle (文件句柄后面不允许有逗号)。

现在请看下面这个代码：

```
open(LOGF, ">>logfile") || die "$!";
if (! print LOGF "This entry was written at", scalar(localtime), "\n" ) {
    warn "Unable to write to the log file: $!";
}
close(LOGF);
```

在上面这个代码段中，名字为 logfile 的文件被打开，以便进行附加操作。print 语句将一条消息写入 LOGF 文件句柄。print 的返回值被检查核实，如果记录项无法输出，便发出警告消息。然后文件句柄被关闭。

可以同时打开多个文件句柄，以便进行读取和写入操作，正如下面这个代码段显示的那样：

```
open(SOURCE, "sourcefile") || die "$!";
open(DEST, ">destination") || die "$!";
@contents=<SOURCE>;                      # Slurp in the source file.
print DEST @contents;                     # Write it out to the destination
close(DEST);
close(SOURCE);
```

上面这个代码段实现了一个简单的文件拷贝。实际上同时进行读取和写入操作可以将例程缩短一些：

```
print DEST <SOURCE>;
```

由于print函数希望有一个列表作为其参数，因此 <SOURCE>是在列表上下文中计算的。当尖括号运算符在列表上下文中进行计算时，整个文件将被读取，然后输出到文件句柄 DEST。

5.4 自由文件、测试文件和二进制数据

文件和文件系统不一定只包含你写入文件的数据。有时文件句柄代表的不仅仅是个简单文件。例如，文件句柄可以被附加给键盘设备、网络套接字和大容量存储设备，如磁带驱动器。

另外，文件系统也可以包含所谓的关于你的文件的元数据。Perl可以用来查询文件系统，以便确定你的文件究竟有多大，上次修改文件的时间，谁修改了这个文件，以及文件中究竟有些什么信息等。在有些操作系统中，文件的元数据甚至能够确定文件是作为文本文件还是二进制文件来处理。

5.4.1 自由文件句柄

最初，Perl是个UNIX实用程序，有时，甚至在非UNIX平台上也会出现它的许多蛛丝马迹。当你的Perl程序启动运行时，它会得到3个自动打开的文件句柄。它们是STDOUT（标准输出）、STDIN（标准输入）和STDERR（标准错误）。按照默认设置，它们均与你的终端相连接。

当你键入数据时，Perl能够读取来自STDIN文件句柄的输入：

```
$guess=<STDIN>;
```

当想要显示输出信息时，可以使用 print函数。按照默认设置，print使用STDOUT文件句柄：

```
print "Hello, World!\n";                      # is the same as...
print STDOUT "Hello, World!\n";
```

在第12学时中，我们将要介绍如何改变print的默认文件句柄。

STDERR通常也可以设置为你的终端，但是它用于显示出错消息。在 UNIX中，出错消息和正常的输出可以发送到不同的显示设备，并且，将出错消息写入 STDERR文件句柄是一种传统的做法。die和warn函数都能够将它们的消息写入 STDERR。如果你的操作系统没有单独的报告错误的文件句柄，比如像 Windows或DOS那样，那么 STDERR输出将被送往 STDOUT设备。



在UNIX中将出错消息和输出信息送往其他设备的问题不在本书讲解的范围之内，而且这方面的技术将根据你使用的 shell程序而各有差别。凡是介绍UNIX的好的著作都会全面介绍这方面的内容。

5.4.2 二进制文件

有些操作系统、比如VMS、Atari ST，尤其是Windows和DOS，都对二进制文件（原始文件）与文本文档进行了区分。这种区分产生了许多的问题，因为Perl无法知道它们之间的差别，并且你也不希望它进行这样的区分。文本文档只不过是以行尾字符（称为记录分隔符）结束的记

下载

录。二进制文件是指需要进行内部转换的许多信息位的集合，如映像、程序和数据文件等。

当你将数据写入一个文本文件时，Perl将\n字符序列转换成你的操作系统使用的记录分隔符。在UNIX中，\n变成一个ASCII 10 (LF)；在Macintosh上，\n转换成ASCII 13 (CR)；在DOS和Windows系统上，它变成序列 ASCII 13和ASCII 10 (CRLF)。当你写入文本时，这个运行特性是合适的。

当写入二进制数据即GIF文件、EXE文件或MS word文档时，这种转换是你所不想要的。每当你真的想要写入二进制数据并且不希望 Perl或操作系统对它进行转换时，你必须使用binmode函数，将文件句柄标记为二进制文件。应该在文件句柄打开之后和对它进行输入输出之前使用binmode：

```
open(FH, "camel.gif") || die "$!";
binmode(FH);           # The filehandle is now binary.
# Start of a valid GIF file...
print FH "GIF87a\056\001\045\015\000";
close(FH);
```

只能对文件句柄使用一次binmode函数，除非将它关闭后再重新打开。如果在不能区分二进制文件和文本文件的系统（UNIX或Macintosh）上使用binmode函数，不会造成任何危害。

5.4.3 文件测试运算符

在打开文件之前，有时应该了解一下该文件是否存在，或者该文件是否是个目录，或者打开文件是否会生成一个permission denied（不允许访问）的错误。对于这些情况，Perl提供了文件测试运算符。这个文件测试运算符的句法如下：

```
-X filehandle
-X pathname
```

这里的x是指你想进行的特定测试，而filehandle是你想要测试的文件句柄。也可以在不打开文件句柄的情况下测试一个pathname。表5-2列出了一些运算符。

表5-2 部分文件测试运算符一览表

运 算 符	举 例	结 果
-r	-r 'file'	如果可以读取‘file’，则返回真
-w	-w \$a	如果\$a中包含的文件名是可以写入的文件名，则返回真
-e	-e 'myfile'	如果‘myfile’存在，则返回真
-z	-z 'data'	如果‘data’存在，但是它是空的，则返回真
-s	-s 'data'	如果‘data’存在，则返回‘data’的大小，以字节为计 量单位
-f	-f 'novel.txt'	如果‘novel.txt’是个普通文件，则返回真
-d	-d '/tmp'	如果‘/tmp’是个目录，则返回真
-T	-T 'unknown'	如果‘unknown’显示为一个文本文件，则返回真
-B	-B 'unknown'	如果‘unknown’显示为一个二进制文件，则返回真
-M	-M 'foo'	返回程序启动运行以来‘foo’文件被修改后经过的时间 (以天数计算)

可以通过在线文档来查看文件测试运算符的完整列表。在命令提示符处键入 perldoc perlfunc，查看“Alphabetical List of Perl Functions”(按字母顺序排列的Perl函数列表)这一节的内容。

文件测试运算符可以像下面这样在改写文件之前用于检查该文件是否存在，或者用于核

实用户的输入信息，也可以用于确定需要的目录是否存在以及是否可以进行写入操作：

```
print "Save data to what file?";
$filename=<STDIN>;
chomp $filename;
if (-s $filename) {
    warn "$file contents will be overwritten!\n";
    warn "$file was last updated ",
-M $filename, "days ago.\n";
}
```

5.5 课时小结

本学时我们介绍了如何在 perl 中打开和关闭文件句柄。可以使用 open 函数来打开文件，使用 close 函数来关闭文件。当文件句柄打开时，它们可以使用 <> 或 read 函数进行读取，使用 print 进行写入操作。另外，我们还介绍了你的操作系统处理文件时使用的一些特殊方法，以及如何使用 binmode 来处理文件。

此外，希望你也了解到了一些关于防错编程的方法。

5.6 课外作业

5.6.1 专家答疑

问题：我的 open 语句经常出错，我不知道究竟问题何在。请问问题究竟在哪里？

解答：首先，请检查 open 语句的句法是否正确。应该确保打开的是正确的文件名。如果希望更加有把握的话，请在 open 的前面放上文件名。如果打算写入文件，请务必在文件名的前面加上一个 >，你必须这样做。最重要的是，你有没有使用 open() || die “\$!”；语句来检查 open 的退出状态呢？die 消息在帮助你查找错误方面是非常重要的。

问题：我对文件进行写入操作，可是什么也写不进去。我的输出数据到哪里去了呢？

解答：你的文件句柄正确打开了吗？如果使用了错误的文件名，你的数据就会送往别的文件。常见的错误是在路径名中使用反斜杠并且用双引号将路径名括起来，以便打开文件进行写入操作，如下所示：

```
open(FH, ">c:\temp\notes.txt") || die "$!"; #WRONG!
```

这行代码创建了一个文件，称为 c: (tab) emp(newline)otes.txt，也许这不是你想要创建的文件。另外，你应该确保 open 函数运行成功。除非你激活了 Perl 的警告特性，否则，如果你将数据写入一个尚未正确打开的文件，Perl 就会悄悄地删除输出数据。

问题：当我试图打开一个文件时，open 运行失败了，perl 报告说 permission denied (不允许访问)。为什么？

解答：Perl 要遵守你的操作系统的文件安全性规则。如果你不拥有对文件、目录或者文件所在驱动器的访问权，那么 Perl 也无权访问它们。

问题：我如何才能一次读取一个字符？

解答：Perl 的函数 getc 能够从文件读取单字符输入。从键盘读取单字符就比较困难，因为一次输入一个字符要取决于你的操作系统。当你学习了第 15 学时中关于模块的内容后，并且读到第 16 学时中的专家答疑时，请查看里面的有关说明。在那里的专家答疑中，详细介绍了如何从各种不同的平台读取单个字符的方法，并且配有许多代码举例。大多数代码不属于本

书讲解的范围。

问题：如何才能使其他程序能够同时将数据写入同一个文件？

解答：你所说的问题称为文件锁定。文件锁定将在第 15 学时中介绍。应该说明的是，这项操作并不太容易，也不是太难。

5.6.2 思考题

1) 为了打开一个名叫 data 的文件，以便写入数据，你应该使用下面的哪个代码：

- a. open (FH , “ data ” , write) ;
 - b. open (FH , “ data ”) ; 并且只是输出到 FH
 - c. open (FH , “ >data ”) || die “ Cannot open data: \$! ” ;
- 2) (-M \$file > 1 and -s \$file) 是真，如果
- a. \$file 已经在几天以前被修改，并且里面有数据。
 - b. 该表达式不可能是真。
 - c. \$file 可以写入，并且里面没有数据。

5.6.3 解答

1) 答案是 c。选择 a 是不能成立的，因为这不是 open 如何打开一个文件以便进行写入操作的问题；选择 b 也是不行的，因为它打开的文件句柄只能读取。c 是正确的，因为它才符合你的要求，并且使用正确的形式来进行错误检查。

2) 答案是 a。-M 返回文件已经存在的天数（>1 是指一天以上），如果文件里面有数据，则 -s 返回真。

5.6.4 实习

- 修改第 4 学时中的 Hangman 程序，以便从数据文件中取出可能的单词的列表。

China-pub.com

下载

China-pub.com

下载

第6学时 模式匹配

在上个学时中，我们介绍了如何从文件中读取数据的方法。懂得这个方法后，再加上标量、数组和运算符方面的知识，就可以准备对该数据进行操作，以便做你想要做的任何事情。有时，文件中的数据没有采用便于使用的格式进行格式化，它不能使用简单的 split函数对数据进行分割，或者有的文件行包含了你不感兴趣的数据，你想通过编辑将它删除。

你必须具备一种能力，来识别输入数据流中的模式。根据这些模式来选择数据，并且对数据进行编辑，使之变成比较容易使用的格式。Perl的工具中有一个工具可以用来执行这项任务，即正则表达式。在本学时的课文中，正则表达式与模式几乎可以互换使用。

正则表达式本身几乎是一种语言，它是用于描述要匹配模式的正式方法。在本学时中，我们将要介绍一些关于这种模式匹配的语言。



在线文档对 Perl使用的完整的正则表达式语言进行了更加深入的（但是比较扼要的）描述。你可以查看 Perl中包含的 perlre 文档。这个问题涉及的内容非常广泛，因此有人出版了整整一本书来介绍正则表达式。Perl界大力推荐的这本书名叫“Mastering Regular Expressions”，它是由 Jeffery E.F. Friedl 撰写的，1997年出版。它全面介绍了正则表达式，并且对 Perl给予了很大的关注。

正则表达式也可以用于其他编程语言，如 TCL、JavaScript、Pathon 和 C 语言。UNIX操作系统的许多实用程序也使用正则表达式。Perl正好配有一组非常丰富的正则表达式，这与其他操作系统的情况非常相似，学习这些正则表达式不仅有助于你在 Perl中的应用，而且可以用于其他语言。在本学时中，你将学习：

- 如何创建简单的正则表达式。
- 如何使用正则表达式进行模式匹配。
- 如何使用正则表达式来编辑字符串。

6.1 简单的模式

在Perl中，模式被括在模式匹配运算符中间，有时该运算符采用 m//的形式。下面就是一种简单的模式：

```
m/simon/
```

上面这个模式依次与字母 S-i-m-o-n 相匹配。但是究竟在什么地方才能找到 Simon 呢？以前我们讲过，Perl 变量 \$_ 常常用于 Perl 需要默认值的时候。模式匹配是根据 \$_ 来进行的，除非你告诉 Perl 用别的方式来进行匹配（后面我们将要介绍）。因此前面的模式可以在标量变量 \$_ 中寻找 S-i-m-o-n。

如果由 m// 规定的模式可以在变量 \$_ 中的任何地方找到，那么匹配运算符返回真。这样，

下载

能够看到匹配模式的正常位置是在条件表达式中，如下所示：

```
if ($m/Piglet/) {
    : # the pattern "Piglet" is in $_
}
```

在这个模式中，除非字符是个元字符，否则每个字符均与自己相匹配。大多数“标准”字符均与自己相匹配，这些字符包括 A至Z、a至z和数字。元字符是指改变了模式匹配运行特性的那些字符。下面是元字符的列表：

`^ $ () \ | @ [{ ? . + *`

下面我们很快就要介绍元字符能够做些什么。在你的模式中，如果想要匹配元字符的原义值，只需要在元字符的前面加上一个反斜杠即可，如下所示：

```
m/I won \$10 at the fair/;      # The $ is treated as a literal dollar-sign.
```

前面我们已经讲过模式匹配运算符通常用 `m//` 来表示。实际上，可以用你想要的任何其他字符来代替斜杠，如下面这个例子所示：

```
if ($m,Waldo,) { print "Found Waldo.\n"; }
```

在许多情况下，当模式中包含斜杠（`/`）时且模式的结尾则可能与模式内的斜杠相混淆，可用另一个字符来代替它，因此括号里面的斜杠的前面必须加上反斜杠，如下所示：

```
if ($m/\usr\local\bin\hangman/) { print "Found the hangman game!" }
```

可以编写下面这个代码，使上面的代码更加容易阅读：

```
if ($m:/usr/local/bin/hangman:) { print "Found the hangman game!" }
```

如果将模式括起来的字符（称为界限符）是斜杠，那么编写模式匹配代码时也可以不带 `m`。因此，也可以将 `m/Cheetos` 写成 `Cheetos/`。通常情况下，除非需要使用不是斜杠（`//`）的其他界限符，否则，可以只使用斜杠而不使用 `m` 来编写模式匹配代码。

变量也可以用在正则表达式中。如果在正则表达式中看到一个标量变量，Perl首先计算该标量，然后查看正则表达式。这个功能使你能够动态地创建正则表达式。下面这个 `if` 语句中的正则表达式是根据用户输入创建的：

```
$pat=<STDIN>; chomp $pat;
$_="The phrase that pays";
if ($/pat/) {      #Look for the user's pattern
    print "\"$_\\" contains the pattern $pat\n";
}
```



联机手册页和其他文档中的正则表达式有时称为 RE或regexp。为了清楚起见，在本书中将继续将它们称为正则表达式。

匹配的规则

当你开始在Perl中编写正则表达式时，应该知道它必须遵循几条规则。不过，规则并不多，大多数规则在你理解它们之后才具有更大的意义。这些规则是：

- 通常情况下，模式匹配从目标字符串的左边开始，然后逐步向右边进行匹配。
- 如果并且只有当整个模式能够用于与目标字符串相匹配时，模式匹配才返回真（在任何

上下文中均如此)。

- 目标字符串中第一个能够匹配的字符串首先进行匹配。正则表达式不会漏掉某一个能够匹配的字符串，而去寻找另一个更远的字符串来进行匹配。
- 进行第一次最大字符数量的匹配。你的正则表达式可能迅速找到一个匹配的模式，然后设法尽可能延伸能够匹配的字符范围。正则表达式是“贪婪的”，也就是说，它会尽可能多地寻找能够匹配的字符。

6.2 元字符

在下面的所有例子中，与模式相匹配的文本部分用下划线标出。请记住，即使目标字符串中只有一部分与正则表达式相匹配，整个目标字符串也可以说是匹配的。下划线标记用于帮助说明究竟哪些部分是匹配的。

阅读下列各节内容是非常重要的，但是，如果有些内容你无法立即理解，请不要担心。你很快就会理解的。这些元字符的应用将会有所明确。

6.2.1 一个简单的元字符

第一个元字符是圆点(.)。在正则表达式中，圆点用于匹配除了换行符外的任何单个字符。例如，在模式/p.t/中，‘.’用于匹配任何单个字符。这个模式用于匹配pot、pat、pit、carpet、python和pup_tent。‘.’要求存在一个字符，但是不能有更多的字符。因此，该模式不能与apt相匹配(p与t之间没有任何字符)，也不能与expect相匹配(pt之间的字符太多)。

6.2.2 非输出字符

前面我们讲过，若要将元字符纳入正则表达式，应该在字符前面加上一个反斜杠，使它失去“元”的含义，如下所示：

```
/^\$/; # A literal caret and dollar sign
```

当普通字符的前面加上反斜杠后，它就变成了元字符。正如你在第2学时中看到的字符串那样，有些字符的前面加上反斜杠后，它在字符串中就具备了特殊的含义。在正则表达式中，所有这些字符几乎代表相同的值，如表6-1所示：

表6-1 特殊字符

字 符	匹配的字符
\n	换行符
\r	回车符
\t	制表符
\f	换页符

6.2.3 通配符

到现在为止，模式中的所有字符与它们要匹配的目标字符串之间存在一种一对一的关系。例如，在/Simon/中，s与S匹配，i与i匹配，m与m匹配，等等。通配符是一种元字符，它告诉正则表达式有多少字符需要匹配。通配符可以放在任何单个字符或一组字符的后面(后面我们将要详细介绍这方面的内容)。

最简单的通配符是+元字符。+用于使前面的字符与后面的字符至少匹配一次，也可以任意次地进行匹配，并且仍然拥有匹配的表达式。因此，/do+g/将能够与下面的字符串匹配：

下载hounddoghotdogdoogie howserdoooooogdoog

但是不能与下面的字符串匹配：

badge (因为没有o)

doofus (因为没有g)

Doogie (因为D与d不同)

pagoda (因为d、o和g的顺序不对)

与元字符+的作用类似的是*。元字符*使得前面的字符可以进行0次或多次匹配。换句话说，模式/t*/可以进行任意次的匹配，但是，如果没有匹配的字符存在，这也没有问题。因此，/car*t/将能够与下面的字符匹配：

cartedcatcarrrt

但是不能与下面的字符匹配：

carrot (多了一个字符o)

carl (模式中的t是可有可无的)

caart (多出来的a不能匹配)

下一个元字符是?。元字符?用于使前面的字符进行0次或一次匹配(但是不能超过一次)。因此，模式/c?ola/用于对c进行匹配，如果c存在的话。然后对o、l和a进行匹配。实际上，该模式可以对带有ola在内的任何字符串进行匹配。如果ola的前面是一个c，那么该字符串也是匹配的。

元字符?与*之间的区别是：模式/c?ola/可以匹配cola和ola，但是不能与ccola匹配。多出来的c需要进行两次匹配。模式/c*ola/可以匹配cola、ola和ccola，因为c可以根据需要重复匹配任意次，而不只是0次或一次。

如果对一个模式进行0次、一次或许多次匹配不能满足你的需要，那么Perl允许根据你需要的具体次数为你进行匹配，方法是使用花括号{}。花括号的格式如下：

pat{n, m}

这里的n是匹配的最小次数，m是匹配的最大次数，pat是你试图量化匹配的字符或字符组。可以省略n，也可以省略m，但是不能同时省略n和m。请看下面这些例子：

/x{5, 10}/ x至少出现5次，但是不超过10次。

/x{9, }/ x至少出现9次，也可能出现更多次。

/x{0, 4}/ x最多出现4次，也可能根本不出现。

/x{8}/ x必须正好出现8次，不能多，也不能少。

正则表达式中常用的一个通配符是.*。可以用它来匹配任何东西，通常是你感兴趣的其他两样东西之间的任何东西。例如/first.*last/。这个模式设法匹配单词first，再匹配它后面的任何东西，然后匹配单词last。请观察/first.*last/是如何匹配下面的字符串的：

first then last

The good players get picked first, the bad last.

The first shall be last, and the last shall be first.

请仔细观察上面第3行中的匹配过程。这个匹配过程首先从单词 `first` 开始。接着对单词 `last` 进行匹配，然后继续进行匹配，直到第二次出现单词 `last`。这时，通配符`*`遵循“匹配规则”这一节中列出的第4条规则，即它要匹配数量最大的字符串，并且仍然要完成该匹配过程。许多情况下，匹配数量最大的字符串并不是你想要进行的操作，因此 Perl 提供了另一个解决方案，称为最小数量匹配，这在 perlre 手册页中有详细的描述。

6.2.4 字符类

正则表达式中的另一个常用做法是要求匹配“这些字符中的任何字符”。如果你想要匹配数字，那么能够编写一个匹配“0~9的任何数字”的模式将是非常好的。或者，如果你要搜索一个名字列表，想要匹配 `Von Beethoven` 与 `von Beethoven`，那么使用能够匹配“v或者V”的模式，将对你是有帮助的。

Perl 的正则表达式拥有这样一个工具，它称为字符类。若要编写一个字符类，可以用方括号`[]`将这些字符括起来。进行匹配时，字符类中的所有字符被视为单个字符。在一个字符类中，可以设定字符的范围（在范围有意义的时候），方法是在上限与下限之间加一个连字符。下面是一些例子：

字 符 类	说 明
<code>[abcde]</code>	用于匹配 a、b、c、d 或 e 中的任何一个字符
<code>[a-e]</code>	与上面相同。用于匹配 a、b、c、d 或 e 中的任何一个字符
<code>G</code>	用于匹配大写字母 G 或小写字母 g
<code>[0-9]</code>	用于匹配一个数字
<code>[0-9]+</code>	用于顺序匹配一个或多个数字
<code>[A-Za-z]{5}</code>	用于匹配任何一组 5 个字母字符
<code>[* ! @ # \$ % & ()]</code>	用于匹配这些符号中的任何一个

最后一个例子非常有意思，因为在字符类中，大多数通配符会失去它们的“通配符性质”，换句话说，它们的运行特性将类似其他任何一个普通字符。因此，`*` 实际上代表一个普通的`*`字符。

如果插入记号`(^)`作为字符类中的第一个字符，该字符类将变为无效。也就是说，该字符类可以匹配不在该字符类中的任何单个字符。如下面的例子所示：

```
/[^A-Z]/;      # Matches non-uppercase-alphabetic characters.
```

由于`]`、`^` 和`-` 等字符都是字符类中的特殊字符，因此，如果按照字符的原义来匹配字符类中的这些字符，就要使用某些规则。若要匹配字符类中的原义字符`^`，必须确保它不出现在字符类中的第一个字符的位置上；若要匹配字符`]`，你既必须将它放在字符类中的第一个字符位置上，也可以在它的前面加上一个反斜杠（例如`/[abc\]/`）；若要将一个原义连字符`(-)` 放在字符类中，你只需要将它放在字符类中的第一个字符位置上，或者在它的前面放一个反斜杠。

Perl 包含了某些常用字符类的快捷方式。它们用反斜杠和通配符来表示，如表 6-2 所示。

下面是一些例子：

```
/\d{5}/;      # Matches 5 digits
/\s\w+\s/;    # Matches a group of word-characters surrounded by white space
```

表6-2 特殊字符类

模 式	用 于 匹 配
\w	一个单词字符，与[a-zA-Z0-9_]相同
\W	一个非单词字符（与\w相反）
\d	一个数字，与[0-9]相同
\D	一个非数字
\s	一个白空间字符，与[\t\f\r\n]相同
\S	一个非白空间字符

不过请注意，上面的最后一个例子不一定匹配一个单词，它也可以匹配一个前后是空格的下划线。同时，并不是所有单词都可以被最后一个模式来匹配的，它们的前后必须加上白空间，并且“don’t”之类的单词不能匹配，因为它包含一个省字号。本学时的后面部分的内容还要进一步介绍用于单词匹配的更好的模式。

6.2.5 分组和选择

有时在正则表达式中，你可能想要知道是否找到了一组模式中的任意一个模式。例如，这个字符串包含dogs还是cats？正则表达式对这个问题的解决办法称为选择。当可能的匹配项之间用一个|字符隔开时，正则表达式中就出现了选择，如下例所示：

```
if (/dogs|cats/) {
    print "\$_ contains a pet\n";
}
```

选择可能是非常有趣的，但是，当想要匹配许多类似的东西时，它也可能很麻烦。例如，如果你想要匹配frog、bog、log、flog或clog等单词时，可以试用/frog|bog|log|flog|clog/这个表达式，不过它包含许多重复的选择操作。而你真正想要进行的操作只是比较字符串的第一部分，如下所示：

```
/fr|b|l|f1|clog/; # Doesn't QUITE work.
```

上面这个例子不一定能够运行，因为Perl没有办法知道选择是你要进行匹配的一个对象，而og是要匹配的另一个对象。为了解决这个问题，可以使用Perl的正则表达式，用括号将模式的各个部分组合起来，如下所示：

```
/(fr|b|f1|c1)og/;
```

可以对括号进行嵌套，使一个组中包含另一个组。例如，也可以将上面的表达式写成/(fr|b|(f|c)|) og/。

在列表上下文中，匹配运算符返回括号中匹配的表达式的各个部分的一个列表。每个加括号的值都是列表的返回值，如果模式不包含括号，则返回1。请看下面这个例子：

```
$_= "apple is red";
($fruit, $color)=/(.*)\sis\s(.*)/;
```

在上面这个代码段中，该模式先对任意对象（作为一个组）进行匹配，然后对白空间进行匹配，再对单词is进行匹配，然后匹配更多的白空间，再对任意对象（也作为一个组）进行匹配。这两个分组的表达式返回左边的列表，并赋予\$fruit和\$color。

6.2.6 位置通配符

最后两个通配符（相信你可能认为通配符是没有止境的）是位置通配符。可以使用位置

通配符来告诉正则表达式，你要查找的模式的准确位置是在字符串的开头，还是在字符串的结尾。

第一个位置通配符是插入记号（^）。正则表达式开头的插入记号告诉正则表达式只匹配一行开头的字符。例如，/^video/只匹配单词video，如果它出现在一行的开头的话。

与它相对应的通配符是美元符号（\$）。正则表达式结尾处的美元符号能够使模式只匹配一行结尾的字符。例如/earth\$/用于匹配earth，不过它只能位于行尾。

模 式	作 用
/^Help/	只匹配以 Help开头的行
/^Frankly.*darn\$/	用于匹配以 Frankly开头和以 darn结尾的行。它们中间的所有字符也进行匹配
/^hysteria\$/	用于匹配只包含单词 hysteria的行
/^\$/	用于匹配一行的开头，紧接着匹配该行的结尾。它只用于匹配空行
/^/	用于匹配带有开头字符的行（所有行）。/\$的作用也相同

6.3 替换

仅仅查找字符串中的模式和输入的信息行是不够的，有时也需要修改数据。方法之一（当然不是惟一的方法）是使用替换运算符s///。它的句法如下：

```
s/searchpattern/replacement/;
```

替换运算符用于默认搜索\$_，找出searchpattern，并且用replacement来替换整个匹配的正则表达式。该运算符返回匹配的数量或进行替换的数量，如果没有进行任何匹配，则返回0。下面是一个例子：

```
$_= "Our house is in the middle of our street".
s/middle/end/;           # Is now: Our house is in the end of our street
s/in/at/;                # Is now: Our house is at the end of our street.
if (s/apartment/condo/) {
    # This code isn't reached, see note.
}
```

在这个例子中，替换按照你希望的那样进行。单词middle替换成end，in替换成at。但是if语句运行失败了，因为单词apartment没有出现在\$_中，因此无法替换。

替换运算符也可以使用非斜杠（/）的界限符，使用的方法与匹配运算符相同。只需要直接在s的后面加上你想要的界限符即可，如下所示：

```
s#street#avenue#;
```

6.4 练习：清除输入数据

当你试图更改数据的时候，常常会进行上面这个例子中的“盲目”替换，即替换是进行了，但是不检查退出状态。更改数据是从用户或文件中取出没有完全按照你的要求进行格式化的数据，然后对数据重新进行格式化。程序清单6-1显示了一个例程，用于将你在地球上的体重转换成月球上的体重，这可以展示数据操作时的情况。

使用文本编辑器，键入程序清单6-2中的程序，并将它保存为Moon。务必按照第1学时中的说明使该程序成为可执行程序。

当完成上述操作后，键入下面这个命令行，设法运行该程序：

```
perl Moon
```

程序清单 6-1 显示了某些输出示例。

程序清单 6-1 月球体重转换程序的输出示例

```

1: $ perl Moon
2: Your weight: 150lbs
3: Your weight on the moon: 25.00005 lbs
4: $ perl Moon
5: Your weight: 90 kg
6: Your weight on the moon: 90.9090 lbs

```

程序清单 6-2 你的月球体重转换程序

```

1: #!/usr/bin/perl -w
2:
3: print "Your weight:";
4: $_=<STDIN>; chomp;
5: s/^\\s+//; # Remove leading spaces, if any.
6:
7: if (m/(lbs|kg|kilograms|pounds?)/i) {
8:     if (s/\\s*(kg|kilograms?).*/{$_
9:         $_*=2.2;
10:    } else {
11:        s/\\s*(lbs|pounds?).*/{$_
12:    }
13: }
14: print "Your weight on the moon: ", $_*.16667, " lbs\n";

```

第1行：这一行包含到达解释程序的路径（可以修改该路径，使之适合你的系统的需要）和开关-w。请始终使警告特性处于激活状态。

第3~4行：这几行用于提示用户输入他的体重，将输入的值赋予 \$_，并且用chomp命令删除换行符。请记住，如果没有设定其他变量，那么 chomp 将改成 \$_。

第5行：模式 /\\s+/对该行的开头的白空间进行匹配。它没有列出任何替换字符串，因此，匹配该模式的 \$_ 部分被删除了。

第7行：如果在用户的输入中发现了计量单位，那么该 if 代码块便删除该计量单位，并在适当的情况下对它进行转换。

第8~9行：模式 / \\s*(kgs?|kilograms?) / i 对白空间进行匹配，然后对 kg 或 kilograms（每个元素的结尾都有一个选项 s）进行匹配。这意味着如果输入中包含 kg 或 kg（没有空格），那么它就被删除。如果该模式被发现和删除了，\$_ 中留下的数据将与 2.2 相乘，或者转换成磅。

第11行：否则，从 \$_ 中删除 lbs 或 pounds（并删除前导白空间）。

第14行：\$_ 中的重量（已经转换成磅）乘以 1/6，然后输出。

6.5 关于模式匹配的其他问题

现在你已经能够对 \$_ 进行模式匹配，并且懂得替换的基本概念，因此可以学习更多的功能。为了使正则表达式的运行做到真正有效，必须对 \$_ 之外的变量进行匹配，并且进行更加复杂的替换，还要使用适用于（但不是只能用于）正则表达式的 Perl 函数。

6.5.1 对其他变量进行操作

在程序清单 6-2 中，用户输入的重量存放在 \$_ 中，并用替换运算符和匹配运算符进行操作。

不过该程序清单存在一个问题，那就是 `$_` 并不是用来存放“重量”的最佳变量名。对于初学者来说，`$_`并不十分直观，在不经意中，`$_`也许被改变了。



一般来说，将数据长期存放在 `$_` 中是非常危险的，最终你会感到非常恼火。Perl的许多运算符都使用 `$_`作为默认参数，其中有些运算符也会修改 `$_`。`$_`是Perl的通用变量，如果试图将一个值长时间存放在 `$_` 中（尤其是当你学习了第8学时的内容后），最终将会导致某些错误。

在程序清单 6-2 中使用变量 `$weight` 就比较好。如果要对非 `$_` 的变量使用匹配运算符和替换运算符，则必须将它们与该变量连接起来。为此可以使用连接运算符 `=~`，如下所示：

```
$weight="185 lbs";
$weight=~s/ lbs//;      # Do substitution against $weight
```

`=~` 运算符并不进行赋值，它只是取出右边的运算符，并使它对左边的变量进行操作。整个表达式拥有的值与使用 `$_` 时所拥有的值是相同的，正如你在下面这个例子中看到的那样：

```
$poem="One fish, two fish, red fish";
$n=$poem=~m/fish/;      # $n is true, if $poem has fish7
```

6.5.2 修饰符与多次匹配

到现在为止，你看到的所有正则表达式都是区分大小写字母的。也就是说，在模式匹配中，大写字符与小写字符是不一样的。如果在匹配单词时不考虑它们是大写字母还是小写字母，那么需要使用下面的代码：

```
/[Mm][Aa][Cc][Bb][Ee][Tt][Hh]/;
```

这个例子看上去不仅很笨，而且很容易出错，因为很容易在键入大写和小写字母对时发生错误。替换运算符 (`s//`) 和匹配运算符 (`m//`) 能够在匹配正则表达式时不考虑大小写字母，如果匹配项的后面跟一个字母 `i` 的话。

```
/macbeth/i;
```

上面这个例子可以对 `Macbeth` 进行匹配，无论它是使用大写字母、小写字母还是大小写混合字母 (`MaCbEtH`)。

用于匹配和替换的另一个修饰符是全局匹配修饰符 `g`。正则表达式（或替换）的匹配操作不是一次完成的，它要重复通过整个字符串，第一次匹配后，立即进行下一次匹配（或替换）。

在列表上下文中，全局匹配修饰符可使匹配代码返回一个放在括号中的正则表达式的各个部分的列表：

```
$_= "One fish, two frog, red fred, blue foul";
@F=m/\W(f\w\w\w)/g;
```

该模式首先匹配一个非单词字符，然后匹配字母 `f`，接着匹配4个单词字符。字母 `f` 和4个单词用括号分组。该表达式被计算后，变量 `@F` 将包含4个元素，即 `fish`、`frog`、`fred` 和 `foul`。

在标量上下文中，修饰符 `g` 使得匹配操作迭代通过整个字符串，为每个匹配操作返回真，当不再进行更多的匹配操作时，返回假。现在请看下面这个代码：

```
$letters=0;
$phrase="What's my line?";
while($phrase=~/\w/g) {
    $letters++;
}
```

下载

上面这个代码段使用匹配运算符 (//)，它带有标量上下文中的修饰符 g。while循环这个条件提供了标量上下文。该模式用于匹配一个单词字符。While循环将继续运行(并且letters 被递增)，直到匹配代码返回假为止。当该代码段运行完成后，\$letters的结果将是11。



在第9学时中，我们将要展示更加有效的对字符进行计数的方法。

6.5.3 反向引用

当将括号用于Perl的正则表达式中时，由每个带括号的表达式进行匹配的目标字符串的这个部分将被记住。Perl将把这个匹配的文本记录在一些特殊的变量中，这些变量的名字是 \$1 (用于第一组括号) \$2 (用于第二组括号) \$3、和\$4等等。现在请看下面这个例子：

```
/(\d{3})-(\d{3})-(\d{4})/
   ^       ^       ^
   $1     $2     $3
```

上面这个模式用于匹配格式很好的美国 /加拿大电话号码，比如 800-555-1212，同时将电话号码的每个部分记录在\$1、\$2和\$3中。这些变量可以用在下面的表达式的后面：

```
if (/(\d{3})-(\d{3})-(\d{4})/) {
    print "The area code is $1";
}
```

它们也可以用作替换操作中的替换文本的组成部分，如下所示：

```
s/(\d{3})-(\d{3})-(\d{4})/Area code $1 Phone $2-$3/;
```

不过应该注意，模式匹配运行成功时，变量 \$1、\$2和\$3的值将被清除（不管它是否使用括号），如果并且只有当模式匹配运行成功时，这些变量才被设置。基于这个情况，请看下面这个例子：

```
m/(\d{3})-(\d{3})-(\d{4})/;
print "The area code is $1"; # Bad idea. Assumes the match succeeded:
```

在上面这个例子中，使用 \$1时根本没有确定模式匹配是否可行。如果模式匹配运行失败，将会带来麻烦。

6.5.4 一个新函数：grep

Perl中的一个常见操作是搜索数组，寻找某些模式。例如，如果将一个文件读入一个数组，然后你想要知道哪一行包含某个单词。Perl有一个特殊的函数，可以用来进行这项操作，这个函数称为grep。grep函数的句法如下：

```
grep expression, list
grep block list
```

grep函数迭代运行通过 list中的每个元素，然后执行 expression或block。在expression或block中，\$_被设置为要计算的列表中的每个元素。如果该表达式返回真， grep就返回该元素。请看下面这个例子：

```
@dogs=qw(greyhound bloodhound terrier mutt chihuahua);
@hounds=grep /hound/, @dogs;
```

在上面这个例子中，@dogs的每个元素被依次赋予 \$_。然后根据\$_对表达式/hound/进行测试。返回真的每个元素被 grep返回，并存放在@hounds中。

这里你必须记住两点。首先，在表达式中，`$_`是对列表中的实际值的引用。如果修改`$_`，就会改变列表中的原始元素：

```
@hounds=grep s/hound/hounds/, @dogs;
```

当运行这个代码后，`@hounds`将包含`greyhounds`和`bloodhounds`（请注意它们结尾处的字母`s`）。通过修改`$_`，原始数组`@dogs`也被修改了，同时，它现在包含了`greyhounds`、`bloodhounds`、`terrier`、`mutt`和`chihuahua`。

需要记住的另一点（Perl程序员有时忘记了这一点）是：`grep`不一定必须与模式匹配或替换运算符一道使用，它可以与任何运算符一道使用。下面这个例子用于检索长度超过8个字符的犬名：

```
@longdogs=grep length($_)>8, @dogs;
```



`grep`函数与UNIX的一个命令同名，该命令用于搜索文件中的模式。UNIX的`grep`命令在UNIX中的用处是如此之大（因此在Perl中用处也很大），以至于它已经变成了一个动词，即“to grep”（进行模式搜索）。如果我们说`to grep through a book`，那么这句话的意思是翻阅每一页，寻找某个模式。

一个相关函数`map`的句法与`grep`基本相同，不过它的表达式（或语句块）返回的值是从`map`返回的，而不是`$_`的值。可以使用`map`函数，根据第一个数组来产生第二个数组。下面是该函数的一个例子：

```
@words= map {split ' ', $_} @ input;
```

在这个例子中，数组`@input`的每个元素（作为`$_`传递给语句块）均用空格隔开。这意味着`@input`的每个元素均产生一个单词列表。该列表存放在`@words`中。`@input`的每个相邻行均被分隔开来，并在`@words`中进行累加。

6.6 课时小结

在本学时中，我们介绍了什么是正则表达式，它们采用什么样的结构，以及如何在Perl中使用正则表达式。正则表达式是由标准字符和元字符构成的。标准字符通常是指字符本身的意义，而元字符则用于改变标准字符的含义。正则表达式可以用于测试是否存在某些模式，或者用于替换模式。

6.7 课外作业

6.7.1 专家答疑

问题：模式`\w (\w) +W`似乎不能与文本行上的所有单词相匹配，它只能与中间的几个单词进行匹配。为什么？

解答：你查找的是用非单词字符括起来的单词字符。文本行的第一个单词的前面没有非单词字符。它的前面根本就没有任何字符。

问题：`m//`与`//`之间有什么差别。我不明白。

解答：它们之间几乎没有差别。当你决定设置一个不是 / 的模式界限符时，两者之间就会显示出惟一的一个差别。如果你在该模式的前面放置一个 m，比如m!pattern!，那么你才能不使用 / 界限符。

问题：我想要检查用户键入的一个数字，但是 /\d*/似乎不起作用。它总是返回真！

解答：它返回真，因为仅仅使用通配符 * 的模式总是能够运行成功的。它既能够对出现 0 次的\d进行匹配，也能够对出现 2 次、100 次或 1000 次的\d进行匹配。使用 /\d+/，能够确保你至少对一个数字进行匹配。

如果你已经开始学习正则表达式的模式，请设法做一下下面这些思考题，以了解学到了哪些知识。

6.7.2 思考题

1) 如果你拥有一些格式化为“x=y”的代码行，那么使用什么表达式可以将表达式的左边与右边相交换？

- a. s/ (.+) = (.+) /\$2=\$1/ ;
- b. s/ (*) = (*) /\$2=\$1/ ;
- c. s/ (.*) = (.*) /\$2\$1/ ;

2) 下面这个代码运行后，\$2中的值是什么？

```
$foo= "Star Wars: The Phantom Menace
$foo=~/Star\s((Wars):The Phantom Menace)/
```

- a. 模式匹配后\$2没有被设置，因为匹配失败。
- b. Wars
- c. Wars:The Phantom Menace

3) 模式m/^[-+]?[0-9]+(\.[0-9]*)?\$/匹配的结果是什么？

- a. 日期，格式为04-03-1969
- b. 格式很好的数字，如45，15.3，-0.61
- c. 类似加法的模式，如4+12或89+2

6.7.3 解答

1) 答案是a。如果选择c，那么在替代字符串中将不包括符号=，它在\$1或\$2中将不能被捕获，因为=出现在括号的外面。选择b是无效的，一个字符必须出现在*的前面。选择a，就能够正确地执行该操作。

2) 答案是a。匹配失败是因为star没有使用大写，同时，匹配代码不包含区分大小写字母的修饰符i。由于这个原因，你始终都应该在使用\$1、\$2等之前测试匹配代码运行是否成功。（如果模式匹配使用了i修饰符，或者star已经变成大写字母，那么选择b就可以得到正确的结果。）

3) 答案是b。在匹配行的开头，该模式应该是一个选项+或-，接着是一个或多个数字，然后（可选项）在行的结尾是一个小数点并且可能是多个数字。该模式可以匹配简单的、格式很好的数字。

6.7.4 实习

- 看一看你是否能够创建一个用于匹配标准时间格式的模式。下面的所有格式应该都是可

行的：12:00am、5:00pm、8:30AM。下面这些格式是不行的：3:00、2:60am、99:00am、3:0pm。

- 编写一个短程序，使它能够执行下列操作：

- 1) 打开一个文件。
- 2) 将所有文件行读入一个数组。
- 3) 从每个行中取出所有单词。
- 4) 找出至少拥有4个连续辅音或非元音字母的所有单词（比如“thoughts”或“yardstick”这样的单词）。

China-pub.com

下载

China-pub.com

下载

第7学时 哈希结构

哈希是Perl中的第三种基本数据类型。你学习的第一种数据类型是标量，它是一种简单 的数据类型，用于存放一个数据（任何一个大小任意的数据，但是只能存放一个数据）。第二种 数据类型是数组，它是标量的集合。数组可以根据你的需要存放任意多个标量，但是，如果 要在数组中搜索你需要的标量，通常必须顺序访问该数组，直到找到你需要的标量。

哈希是另一种集合型数据类型。与数组一样，哈希包含了许多个标量。数组与哈希的差 别是：哈希是按照名字来访问它们的标量的，而不是像数组那样使用数字标号进行访问。哈 希元素包含两个部分，即一个关键字和一个值。关键字用于标识哈希的每个元素，而值则是 与该关键字相关联的数据。这种关系称为关键字值对。

许多应用程序非常适合于这种类型的数据结构。例如，如果想存放某个州的持照驾驶员 的信息，那么就可以使用这些驾驶员的执照号码作为关键字来存放执照信息。这些号码是独 一无二的（每个驾驶员只有一个号码）。与每个号码相关的数据就是驾驶员的信息（执照类型、 地址和年龄等）。每个驾驶员的执照代表哈希结构中的一个元素，其号码和信息就构成了关 键字值对的关系。具备哈希性质的其他数据结构有库存零件号、医院病历、电话付款记录、磁 盘文件系统、音乐光盘收藏、Rolodex信息、国会图书馆、ISBN号码和其他许多数据结构。

Perl中的哈希结构可以根据你的需要包含任意多个元素，至少可以包含系统内存允许存 放的最大数量的元素。当将元素添加到该哈希结构或者从哈希结构中删除元素时，哈希结构就 会改变其大小。访问哈希结构中的各个元素是非常快的，并且不会因为哈希结构变大而大幅 度降低访问速度。这意味着不管哈希结构拥有10个元素还是10万个元素，Perl都能够得心应手 并迅速地处理哈希结构。哈希结构的关键字的长度可以根据需要而定（它们只是标量而已）， 而哈希结构的数据部分的长度也可以根据需要来确定。

传统上，在Perl和其他语言中，哈希结构称为关联性数组。这是个冗长的术语，用于说明关 键字是与值相关的。由于Perl程序员不喜欢冗长的词语，因此关联性数组现在简称为哈希结构。

在Perl中，哈希变量是以百分比符号（%）来标识的，它们与数组和标量不使用相同的名 字。例如，你可以拥有一个名字叫%a的哈希变量，也可以有一个名字叫@a的数组，还可以有 一个名字叫\$a的标量。这些名字指的是3个互不相关的变量。

在本学时中，你将要学习如何进行下面的操作：

- 创建哈希结构。
- 将元素插入哈希结构和从哈希结构中删除元素。
- 使用哈希结构对数组进行操作。

7.1 将数据填入哈希结构

若要创建哈希元素，只需要将值赋予这些元素即可，这与创建数组的元素很相似。例如， 可以使用类似下面的代码来创建各个哈希元素：

```
$Authors{'Dune'}='Frank Herbert';
```

在这个例子中，将 %Authors 赋予哈希结构。该元素的关键字是单词 Dune，数据是名字 Frank Herbert。这个赋值操作在哈希中创建了 Dune 与 Frank Herbert 之间的一种关系。\$Authors{ 'Dune' } 可以像任何其他标量那样来进行处理，它可以被传递给函数，被操作员修改，可以输出，也可以重新赋值。当修改一个哈希元素时，请始终记住，你是修改存放在哈希元素中的值，不是修改哈希本身的价值。

为什么这个例子使用的是 \$Authors{}，而不是 %Authors{} 呢？与数组一样，当哈希结构作为一个整体来展示时，它们的变量名的前面有它自己的标记（%）。当你访问哈希结构的单个元素，即一个标量值时，要在变量名的前面加上一个美元符号（\$），表示它引用的是单个值，同时使用花括号来指明该值。对于 Perl 来说，\$Authors{ 'Dune' } 代表单个标量值，在这个例子中，代表 Frank Herbert。

只有一个关键字的哈希结构并不特别有用。若要将若干个值放入一个哈希结构，可以使用一系列的赋值语句，如下面的代码所示：

```
$food{'apple'}='fruit';
$food{'pear'}='fruit';
$food{'carrot'}='vegetable';
```

若要使这个代码变得短一些，可以用一个列表对该哈希结构进行初始化。该列表应该包含成对的关键字与值，如下所示：

```
%food=('apple', 'fruit', 'pear', 'fruit', 'carrot', 'vegetable');
```

这个例子看上去与第 4 学时介绍的数组初始化的例子有些相似。实际上，当你学到本学时后面部分的内容时，就会知道哈希结构可以在许多上下文中作为一种特殊类型的数组来处理。

当你对哈希结构进行初始化时，要想跟踪大型列表中的哪些项目是关键字，哪些项目是值，这是很容易搞混的。Perl 有一个特殊的运算符，称为逗号箭头运算符，即 =>。使用 => 运算符，同时利用 Perl 忽略白空间的特性，就能够编写下面这样的哈希结构的初始化代码：

```
%food=( 'apple' => 'fruit',
       'pear'    => 'fruit',
       'carrot'   => 'vegetable',
     );
```

可以使用两个辅助的快捷方式来进行哈希结构的初始化。=> 运算符的左边将是个简单的字符串，不需要用引号括起来。另外，花括号中的单个单词的哈希关键字会自动加上引号。因此，前面显示的初始化代码将变成下面的形式：

```
$Books{Dune}='Frank Herbert';
%food=( apple => 'fruit', pear => 'fruit', carrot => 'vegetable' );
```



逗号箭头运算符之所以叫做这个名字，原因是它的作用类似于逗号（当它用于分隔列表的项目时），并且它看上去像个箭头。

7.2 从哈希结构中取出数据

若要从哈希结构中取出单个元素，只需要使用一个 \$、哈希结构的名字和你想要检索的关键字。请看下面这个例子：

```
%Movies=( 'The Shining' => 'Kubrick', 'Ten Commandments' => 'DeMille',
          Goonies=> 'Spielberg');
print $Movies{'The Shining'};
```

这些代码行用于输出哈希结构 %Movies 中的元素 The Shining。这个例子将输出 Kubrick。

有时查看哈希结构中的所有元素是非常有用的。如果哈希结构中的所有关键字都是已知的，那么你可以像上面显示的那样按照关键字来访问它们。但是，大多数情况下，按照名字来访问每个关键字很不方便。你可能不一定知道所有关键字的名字，也可能关键字的数量太大，无法一一列举。

可以使用 keys 函数来检索作为列表返回的哈希结构的所有关键字，然后可以查看该列表，找出哈希结构的所有元素。在哈希结构的内部，它的关键字并不按照特定的顺序进行存放，keys 函数返回的关键字也不使用特定的顺序。若要输出该哈希结构中的所有电影名字，可以使用下面的代码：

```
foreach $film (keys %Movies) {  
    print "$film\n";  
}
```

这里，\$film 使用 keys %Movies 返回的列表的每个元素的值。如果除了电影的名字外，还想输出所有导演的名字，那么可以输入下面的代码：

```
foreach $film (keys %Movies) {  
    print "$film was directed by $Movies{$film}.\n";  
}
```

这个代码段输出的结果如下：

```
Ten Commandments was directed by DeMille.  
The Shining was directed by Kubrick.  
Goonies was directed by Spielberg.
```

由于 \$film 包含一个哈希关键字的值，因此 \$Movies{\$film} 将检索该关键字代表的哈希结构的元素值。可以将它们同时输出，以便观察哈希结构中的关键字与值之间的关系。（请记住，你的输出可能以不同的顺序出现，因为 keys 函数返回的关键字不是按照特定的顺序排列的。）

Perl 还提供了另一个函数 values，用于检索哈希结构中存放的所有值。如果仅仅检索值，通常是没有用处的，因为你不知道哪个关键字与哪个值相关联。返回的哈希结构的值的顺序与 keys 函数返回的关键字的顺序是相同的。现在请观察下面这个例子：

```
@Directors=values %Movies;  
@Films=keys %Movies;
```

在这个例子中，@Directors 和 @Films 的每个下标都包含了一个对来自 %Movies 的相同关键字值对的引用，包含在 \$Directors[0] 中的导演名字对应于存放在 \$Films[0] 中的电影名字，等等。

有时，需要按值而不是按关键字从哈希结构中检索各个元素。按值来检索元素的最好方法是对哈希结构进行切换，也就是说，所有关键字变成值，所有值变成新哈希结构的关键字。下面就是一个例子：

```
%Movies=( 'The Shining' => 'Kubrick', 'Ten Commandments' => 'DeMille',  
         Goonies=> 'Spielberg');  
%ByDirector=reverse %Movies;
```

这是什么呢？当你对哈希结构使用 reverse 函数时，Perl 就将哈希结构转换成一个简单的列表，也许类似于下面这个列表：

```
('The Shining', 'Kubrick', 'Ten Commandments', 'DeMille', 'Goonies',  
'Spielberg')
```

然后 Perl 对该列表中的元素顺序进行倒序，得到下面这个输出：

```
('Spielberg', 'Goonies', 'DeMille', 'Ten Commandments', 'Kubrick', 'The Shining')
```

请注意，现在所有的关键字值对的顺序都倒了过来（值放在了前面）。当你将这个列表赋予%ByDirector时，产生的哈希结构将与原始哈希结构相同，只不过现在所有的关键字变成了值，而所有的值则变成了关键字。不过你应该知道，如果由于某个原因你的哈希结构拥有相重复的值，如果该值（将要变成关键字）不是唯一的，那么你得到的哈希结构拥有的元素将比原先要少。由于在新的哈希结构中，重复的值会发生冲突，因此老的关键字将被新的关键字代替。

7.3 列表与哈希结构

当我们介绍如何对哈希结构进行初始化时，曾经提到哈希结构与数组之间有着一定的相关性。每当哈希结构用于列表环境中时，Perl会将哈希结构重新变为由关键字和值组成的普通列表。该列表可以被赋予数组，这与其他任何列表的情况是一样的，如下所示：

```
%Movies=( 'The Shining' => 'Kubrick', 'Ten Commandments' => 'DeMille',
          Goonies=> 'Spielberg');
@Data=@Movies;
```

这时，@Data是个包含6个元素的数组，偶数元素（包含0的元素）是导演的名字，奇数元素是电影名字。可以对@Data进行任何通常的数组操作，然后将数组赋予%Movies，如下所示：

```
%Movies=@Data;
```



Perl显然是以随机顺序来存放哈希关键字的，这个随机顺序只对Perl有用。Perl并不设法记住放入哈希结构中的关键字的顺序，当检索关键字时，也不按照任何特定的顺序来放置它们。如果要按照某个顺序来显示它们，那么就必须对它们进行排序（参见本学时后面部分中的“用哈希结构进行的有用操作”这一节的内容），否则你应该记住它们插入时的顺序（参见本学时结尾处的“专家答疑”的内容）。

就其他方面来说，数组与哈希结构是相似的。若要拷贝一个哈希结构，只需要像下面这样将这个哈希结构赋予另一个哈希结构即可：

```
%New_Hash=%Old_Hash;
```

当你将%Old_Hash置于哈希初始化代码的右边时（Perl通常希望右边是个列表或数组），Perl便将哈希结构转换成一个列表。然后该列表用于对%New_Hash进行初始化。同样，可以像处理列表那样，将几个哈希结构组合起来并对它进行操作，如你下面看到的那样：

```
%Both=(%First, %Second);
%Additional=(%Both, key1=> 'value1', key2 => 'value2');
```

上面代码中的第一行将两个哈希结构%First和%Second组合成第三个哈希结构%Both。对于这个例子，你应该记住的是，如果%First的有些关键字也出现在%Second中，那么第二次出现的关键字值对就取代%Both中的第一个关键字值对。在第二行代码中，%Both显示为一个放在括号中的关键字值对的列表，另外两个关键字值对也放在括号中。然后整个列表用于对%Additional进行初始化。

7.4 关于哈希结构的补充说明

如果你刚刚开始学习 Perl，那么对哈希结构进行某些操作时可能会遇到困难。由于哈希结构的特殊性质，有两个常用操作需要一些专门的函数，而这些函数对于标量和数组来说是不必要的。

7.4.1 测试哈希结构中的关键字

若要测试哈希结构中是否存在某个关键字，需要使用下面的代码句法：

```
if ( $Hash{keyval} ) {          # WRONG, in this case  
:  
}
```

由于几方面的原因，这个代码是不能满足需要的。首先，这个代码段并不是用来测定 keyval 是否是哈希结构中的一个关键字，它实际上是测试哈希结构中的 keyval 关键字的值。如果像下面这样定义了该关键字，那么它能否达到上面的测试要求呢？

```
if ( defined $Hash{keyval} ) {    # WRONG, again in this case  
:  
}
```

同样，这个代码也是不行的。这个代码段仍然只是测试与关键字 keyval 相关的数据，而不是测试是否存在该关键字。`undef` 是个完全有效的值，用于与哈希关键字相关联，如下面的代码所示：

```
$Hash{keyval}=undef;
```

这个已经定义的测试将返回假，因为它们并没有测试哈希结构中是否存在某个关键字，它测试的是与关键字相关的数据。那么正确的方法应该是什么呢？Perl 有一个专门用于这个目的的函数，称为 `exists`。下面显示的 `exists` 函数可以用于测试哈希结构中是否存在哈希关键字，如果存在，便返回真，否则返回假：

```
if ( exists $Hash{keyval} ) {    # RIGHT!  
:  
}
```

7.4.2 从哈希结构中删除关键字

你可以进行的另一个操作是从哈希结构中删除关键字。正如你在前面看到的那样，仅仅将哈希元素设置为 `undef` 是不行的。若要删除单个哈希关键字，可以使用 `delete` 函数，如下所示：

```
delete $Hash{keyval};
```

若要从哈希结构中删除所有关键字和值，只需要将哈希结构初始化为一个空的列表即可，如下所示：

```
%Hash=();
```

7.5 用哈希结构进行的有用操作

由于许多方面的原因，在 Perl 中使用哈希结构，其目的不仅仅是为了按关键字来存储记录，供以后检索。使用哈希结构的优点是可以迅速访问各个关键字，并且哈希结构中的所有关键字都是唯一的。由于具备这些特性，因此哈希结构对于数据操作是非常有用的。毫不奇怪，由于数组和哈希结构非常相似，因此你用哈希结构进行的许多有趣的操作属于数组操作。

7.5.1 确定频率分布

在第6学时中，你学习了如何取出一行文本，然后将它分割成单词。请观察下面这个代码段：

```
while(<>) {
    while ( /(\w[\w-]*)/g ) { # Iterate over words, setting $1 to each.
        $Words{$1]++;
    }
}
```

第一行代码每次读取一行标准输入，为每一行设置 `$_`。

然后，第二行的 `while()` 循环对 `$_` 中的每个单词进行迭代操作。让我们回顾一下第 6 学时介绍的内容，在标量上下文中使用带有修饰符 `g` 的模式匹配运算符 (`//`)，将返回匹配的每个模式，直到不再剩下匹配的模式为止。寻找的模式是个单词字符 `\w`，后随 0 个或多个单词字符或者连字符 `[\w-]*`。在这个例子中使用了括号，以便记住特殊变量 `$1` 中匹配的字符串。

下一行虽然很短，但是它是该代码段中令人感兴趣的部分。`$1` 依次设置为第二行上的模式匹配的每个单词。该单词用作哈希结构 `%Words` 的关键字，该关键字值对的值开始时没有定义。通过对它进行递增，在第一次看到该单词时，Perl 将该值设置为 1。第二次看到一个单词时，哈希结构 `%Words` 中已经存在关键字（该单词），同时它的值从 1 递增为 2。这个过程将继续进行下去，直到不再有遗漏的输入为止。

当你完成操作后，哈希结构 `%Words` 将包含读入的单词的频率分布情况。若要查看该频率分布，可以使用下面这个代码：

```
foreach( keys %Words ) {
    print "$_ $Words{$_}\n";
}
```

7.5.2 在数组中寻找惟一的元素

上面这个代码中展示的方法也可以用来寻找数组中只出现一次的元素。假设已经将输入的全部单词放入一个数组而不是哈希结构，同时没有专门采取措施来保证在将一个单词放入列表之前，该列表中还没有这个单词。在这种情况下，列表中可能存在许多重复的单词。

如果输入的文本的开始行是 One Fish , Two Fish , 那么该列表看上去如下所示：

```
@fishwords=('one', 'fish', 'two', 'fish', 'red', 'fish', 'blue', 'fish');
```

如果你被赋予这个单词列表（在 `@fishwords` 中），同时你只需要该列表的独一无二的元素，那么使用哈希结构就非常适合你的需要，如程序清单 7-1 所示：

程序清单 7-1 寻找数组中的惟一的元素

```
1:  %seen=();
2:  foreach(@fishwords) {
3:      $seen{$_}=1;
4:  }
5:  @uniquewords=keys %seen;
```

第1行：用于对临时哈希结构 `%seen` 进行初始化，该哈希结构用于存放你的所有单词。

第2行：对单词列表进行迭代操作，依次将 `$_` 设置为每个单词。

第3行：用于创建哈希结构 `%seen` 中的关键字，以 `$_` 中的该单词作为关键字，并为该数据创建一个名义上的值。

第5行：只是从哈希结构中取出所有关键字，并将它们存放在 @uniquewords中。哈希结构中的任何重复单词（例如fish）将互相改写，然后只以一个关键字出现。

7.5.3 寻找两个数组之间的交汇部分和不同部分

对数组经常要进行的一项操作是寻找两个数组之间的交汇部分（即它们的重叠部分）和两个数组之间的不同部分（它们不重叠的部分）。在这个例子中，你有两个列表，一个是包含电影明星的列表，另一个是包含政治家的列表。你的任务是找出是电影明星的所有政治家。下面是你的两个（非常不完整的）数组：

```
@stars=('R. Reagan', 'C. Eastwood', 'M. Jackson', 'Cher', 'S. Bono');
@polis = ('N. Gingrich', 'S. Thurmon', 'R. Reagan',
          'S. Bono', 'C. Eastwood', 'M. Thatcher');
```

程序清单7-2显示了寻找交汇部分的代码。

程序清单7-2 寻找两个数组的交汇部分

```
1:  %seen=();
2:  foreach (@stars) {
3:    $seen{$_}=1;
4:  }
5:  @intersection=grep($seen{$_}, @polis);
```

第1行：用于对哈希结构 %seen进行初始化。这个临时哈希结构用于存放所有电影明星的名字。

第2行：对电影明星的列表进行迭代操作，依次将 \$_设置为每个名字。

第3行：用电影明星的名字填入哈希结构 %seen的各个关键字，并将值设置为1，这个值可以是你想要的任何真值。

第5行：这一行看起来比它实际上更复杂一些。 @polis中的Grep函数对政治家的列表进行迭代操作，依次将 \$_设置给每个政治家。然后，在哈希结构 %seen中寻找该名字。如果该名字返回真，那么它就位于哈希结构中，表达式 \$seen{\$_}计算的结果为真。如果该表达式计算的结果是真，那么 grep返回\$_的值，然后该值被放入 @intersection中。这个过程将重复进行，直到@polis被grep全部查看完毕。当该代码段运行结束时， @intersection便包含既是 @stars又是 @polis的所有成员的名字。

用于寻找两个数组之间的不同部分（即在一个数组中存在，而在另一个数组中不存在的那些元素）的代码与上面这个代码几乎是相同的，可以使用程序清单 7-3来查找不是电影明星的所有政治家。

程序清单7-3 寻找两个数组之间的不同部分

```
1:  %seen=();
2:  foreach (@stars) {
3:    $seen{$_}=1;
4:  }
5:  @difference=grep(! $seen{$_}, @polis);
```

惟一有变化的一行是第5行。它仍然用于查找哈希结构 %seen中的每个政治家的名字，但是，现在如果它找到了政治家的名字，则返回假。相反，如果没有找到政治家的名字，则返回真。出现在哈希结构 %seen中的所有政治家的名字并不返回给 @difference。如果你想要找到不是政治家的所有电影明星，使用的代码几乎完全一样，但是必须将 @stars切换成 @polis。

7.5.4 对哈希结构进行排序

许多情况下，仅仅按照默认顺序来检索哈希结构中的关键字是不够的（默认顺序是非常随机的）。这是许多情况中的一种。可以用两种方法来输出你创建的频率分布，一种是按单词的字母顺序，另一种是按频率的顺序。由于 keys函数能够返回一个简单的列表，因此可以像下面这样使用sort函数对该列表进行排序：

```
foreach( sort keys %Words ) {
    print "$_ $Words{$_}\n";
}
```

按频率给列表排序并没有太大的差别。第4学时中我们讲过，按照默认设置，sort函数只是用ASCII顺序给既定的列表排序。但是，如果需要进行比较复杂的排序，可以在代码块中调用sort函数，以便设定排序顺序。下面这个代码显示了按照值该哈希结构进行排序的情况：

```
foreach ( sort { $Words{$a} <=> $Words{$b} } keys %Words ) {
    print "$_ $Words{$_}\n";
}
```

也许你还记得，与sort一道使用的BLOCK被sort函数反复调用，\$a和\$b被设置为sort函数需要进行排序的每一对值。在这种情况下，\$a和\$b被设置为哈希结构%words中的各个不同关键字。该代码不是直接比较\$a与\$b，而是查看哈希结构%Words中的这些关键字的值，然后对它们进行比较。

7.6 练习：用Perl创建一个简单的客户数据库

当你打电话给客户服务中心，并且最后接通对方的电话时，对方问你的第一个问题是你的电话号码是什么。确实，每次几乎都是这样。有时，客户服务代表想要你的客户号码，甚至你的社会保险号。对方需要的是计算机能够用来识别你的一个唯一标志。这些号码可以作为在数据库中检索关于你的信息时使用的关键字，这就像Perl的哈希结构，是不是？

在这个练习中，你将要搜索一个客户数据库。这个程序假设数据库已经存在，并且尚无更新数据库的任何手段。在这个数据库中，你将允许用户搜索一个或两个不同的域。

在开始做这个练习时，需要一些数据。请打开文本编辑器，键入文本（或类似的某些数据），并将它保存为customers.txt。不必担心列与列之间的空格数量，也不必考虑使它们对齐，你只需要在每一列之间留一个空格。

```
Smith,John (248)-555-9430 jsmith@aol.com
Hunter,Apryl (810)-555-3029 april@showers.org
Stewart,Pat (405)-555-8710 pats@starfleet.co.uk
Ching,Iris (305)-555-0919 iching@zen.org
Doe,John (212)-555-0912 jdoe@morgue.com
Jones,Tom (312)-555-3321 tj2342@aol.com
Smith,John (607)-555-0023 smith@pocahontas.com
Crosby,Dave (405)-555-1516 cros@csny.org
Johns,Pam (313)-555-6790 pj@sleepy.com
Jeter,Linda (810)-555-8761 netless@earthlink.net
Garland,Judy (305)-555-1231 ozgal@rainbow.com
```

在同一个目录中，键入程序清单7-5中的短程序，并将它保存为customer。务必按照第1学时中的说明使该程序成为可执行程序。

当完成上述操作后，键入下面这个命令行，设法运行该程序。

```
perl Customer
```

程序清单 7-4 显示了 Customer 程序的输出。

程序清单 7-4 Customer 程序的示例输出

```
1: Type 'q' to exit
2: Number? <return>
3: E-Mail? cros@csny.org
4: Customer: Crosby, Dave (405)-555-1516 cros@csny.org
5:
6: Number? (305)-555-0919
7: Customer: Ching,Iris (305)-555-0919 iching@zen.org
8:
9: Number? q
10:
11: All done.
```

程序清单 7-5 Customer 程序的完整清单

```
1: #!/usr/bin/perl -w
2:
3: open(PH, "customers.txt") or die "Cannot open customers.txt: $!\n";
4: while(<PH>) {
5:     chomp;
6:     ($number, $email)=(split(/\s+/, $_))[1,2];
7:     $Phone{$number}=$_;
8:     $Email{$email}=$_;
9: }
10: close(PH);
11:
12: print "Type 'q' to exit\n";
13: while(1) {
14:     print "\nNumber? ";
15:     $number=<STDIN>; chomp($number);
16:     $address="";
17:     if (! $number) {
18:         print "E-Mail? ";
19:         $address=<STDIN>; chomp($address);
20:     }
21:
22:     next if (! $number and ! $address);
23:     last if ($number eq 'q' or $address eq 'q');
24:
25:     if ( $number and exists $Phone{$number} ) {
26:         print "Customer: $Phone{$number}\n";
27:         next;
28:     }
29:
30:     if ( $address and exists $Email{$address} ) {
31:         print "Customer: $Email{$address}\n";
32:         next;
33:     }
34:     print "Customer record not found.\n";
35:     next;
36: }
36: print "\nAll done.\n";
```

第1行：这一行包含到达解释程序的路径（可以修改这个路径，使之适合系统的需要）和开关-w。请始终使警告特性处于激活状态。

第3行：文件句柄PH上的customers.txt文件被打开。当然，对它的错误进行了检查，并且作了报告。

第4~5行：文件句柄PH被读取，每一行均被赋予`$_`。`$_`则使用`chomp`命令删除结尾处的换行符。

第6行：`$_`中的这一行在白空间处（`\s+`）被分割。`split`语句的前后加上一组括号，后面是方括号。由于你只对每一行上的电话号码和电子邮件地址感兴趣，所以从分割的语句中取出一部分返回值。这两个值被赋予`$number`和`$email`。

第7~8行：`%Email`用于存放客户记录，关键字是电子邮件地址。`%Phone`用于存放与客户相关的电子邮件地址。

第10行：这一行用于关闭文件句柄。

第13行：该`while`循环包含了需要重复运行的这部分代码。语句`while (1)`是个Perl的习惯用语，意思是“永远循环”。若要退出该循环，最终需要使用`last`语句。

第14~15行：电话号码被读取，换行符被删除。

第17~20行：如果没有电话号码，那么这些代码行提示你输入一个电子邮件地址。

第22~23行：如果没有输入任何信息，这一行便重复运行该循环。如果输入一个`g`，则该循环退出。

第25~28行：如果输入了一个号码，并且是个有效的号码，那么第26行将输出该客户记录。控制权重新传递给顶部带有`next`语句的代码块。

第30~33行：输入了一个地址，并且是个有效的地址，那么输出客户记录。控制权重新传递给顶部带有`next`的代码块。

第34~35行：如果输入了一个地址或电话号码，但是发现它是个无效的地址或号码，那么这两行便输出一条消息，并重复运行带有`next`的代码块。

这个例子展示了Perl的几个特性。哈希结构可以用于根据关键字来迅速查找数据。由于Perl能够非常有效地实现哈希结构，即使该程序拥有哈希结构中的成千上万的记录，查询的响应时间也不会变得很长。另外，这个程序也展示了使用简单BLOCK的程序控制流，而不是其他的控制结构（如`while`、`do`和`until`等）。

7.7 课时小结

哈希结构为程序员提供了许多非常有用的工具。除了简单的记录存储和检索工具外，哈希还提供了对数据进行转换和分析的有用机制。数组操作、记录存储和检索的公式将会给你带来许多好处。在后面的几个学时中，哈希结构将为你提供一个进一步学习DBM文件处理、复杂数据结构的操作，以及与你的系统环境打交道等内容的途径。

7.8 课外作业

7.8.1 专家答疑

问题：如果需要用一个关键字来存储若干个数据（一个列表），我能否在哈希结构中存储多个数据呢？

解答：可以。这需要使用两个基本方法。第一个方法（也是最笨的方法）是将哈希元素

中的值的部分格式化为可以识别的东西，比如用一个用逗号分隔的列表。每当你存储哈希元素时，可以使用join函数，将该列表组合到一个标量中，每当你从哈希结构中检索一个值时，可以使用split函数，将标量重新分割成列表。这种方法比较麻烦，也容易出错。

另一种方法是使用一个引用项。使用引用项后，你就能够创建数组的哈希结构，哈希结构中的哈希结构和其他复杂的数据类型。当你掌握了它的诀窍后，使用引用项来创建复杂数据结构是非常容易的。关于这方面的内容，我们将在第13学时中介绍。

问题：我应该如何按照将关键字赋予哈希结构时的顺序来保持它的顺序？

解答：同样你可以使用两种方法来保持它们的顺序。第一种方法比较难，你要跟踪你的插入顺序。方法是使用一个对哈希结构进行镜像的数组。当将新元素放入哈希结构时，可以使用push将相同的关键字放入一个数组。当需要查看插入顺序时，只需要使用该数组而不是keys函数。这种方法比较复杂，并且容易出错。

更好的方法是使用模块Tie::IxHash。这个模块可以根据你的需要，使keys函数按照插入顺序返回哈希关键字。在第14学时中，我们将要介绍如何使用这个模块的方法。

问题：你能介绍一种将哈希结构写入文件时可以使用的简便方法吗？

解答：当然可以。Data::Dumper或Storable之类的模块能够将哈希和数组等数据类型重新格式化为便于存放的标量值，这些标量值可以写入文本文件。这些模块还拥有一些函数，它们能够利用那些格式化的标量并重新创建你存储的原始结构。

在第15学时中，我们将要介绍一种将哈希写入文件的非常方便的方法，这就是使用DBM文件。使用DBM文件，你可以将你的哈希结构与一个磁盘文件关联起来。当你改变哈希结构时，磁盘文件也会相应地变更。该磁盘文件能够使你的哈希结构一直保留着，只要文件保持不变。

7.8.2 思考题

1) 对于本学时中你编写的Customer程序来说，尤其是如果客户列表非常长的话，为什么name是个不合适的搜索关键字？

- a. 姓和名字的组合超出了Perl的哈希结构允许的长度。
- b. 人的名字不是独一无二的关键字。
- c. 没有人想要按照名字来搜索客户数据库。

2) 相关性数组与哈希结构之间的区别是什么？

- a. 没有区别。
- b. 相关性数组用于更加正式的数据集，比如帐单记录。
- c. 在Perl中，哈希不是真正的相关性数组，因此它们拥有不同的名字。

3) 哪些种类的数据最适合哈希结构？

- a. 简单的项目列表。
- b. 普通常用数据。
- c. 关键字值对的列表。

7.8.3 解答

1) 答案是b。Perl的哈希结构的大小实际上是不受限制的，用户必然要求按名字进行搜索。

但是按照人的名字来搜索是不合适的，因为名字不是独一无二的。电话簿中有许多的重复名字，比如John Smith和Robert Jones等。

2) 答案是a。哈希结构与相关性数组是同一个东西。惟一的区别是哈希说起来顺口，拼写容易。

3) 答案是c。选择c是正确的，不过组织得很好的普通数据哈希结构也是可以的。

7.8.4 实习

- 修改Customer程序，使得它可以按照名字进行搜索。由于你不能将name用作哈希关键字，因此必须通过哈希中的值来进行搜索。
- 修改Customer程序，使得它可以按照关键字的某个部分（比如电话号码的一部分或者电子邮件地址的一部分）来进行搜索。可以使用正则表达式来搜索模式。请记住应该找到多个结果，并且返回每个结果。

China-pub.com

下载

China-pub.com

下载

第8学时 函数

几乎所有的计算机语言都支持函数。函数是一组代码，可以按名字对它进行调用，以便执行某项工作，然后返回某个值。在本书中，你要使用许多函数，比如，你已经使用了 print、reverse、sort、open、close和split等函数。它们都是Perl的内置函数。

Perl还允许你编写自己的函数。在Perl中，用户定义的函数称为子例程。与Perl的内置函数一样，用户定义的函数也可以拥有参数，并且可以将值返回给调用者。

Perl还支持作用域的概念。作用域用于确定某个时间内程序能够看到的一组变量。由于有了Perl的作用域，你就能够编写运行时不受你的程序的其余部分影响的函数。编写得非常出色的函数可以在其他程序中重复使用。

在本学时中，你将要学习：

- 如何定义你自己的函数和如何调用这些函数。
- 如何将值传递给函数，然后返回值。
- 如何使用use strict来编写程序，以便强制使用某种结构。

8.1 创建和调用子例程

可以使用下面的代码来创建用户定义的子例程：

```
sub subroutine_name {  
    statement1;  
    :  
    statementx;  
}
```

Perl中的子例程名与第2学时中介绍的标量、数组和哈希结构的命名约定是相同的。子例程与现有的变量可以使用相同的名字。但是，你应该避免创建名字与Perl的内置函数和运算符相同的子例程。如果在Perl中创建了名字相同的两个子例程，那么在报警特性激活的情况下，Perl就会发出一条警告消息，否则第二个定义的名字会使第一个名字被忘记。

当子例程被调用时，子例程的代码启动运行，并且任何返回值均被重新传递到子例程被调用时的位置。（调用子例程和返回值的内容将在后面介绍。）例如，下面这个短子例程将提示用户输入一个答案：

```
sub yesno {  
    print "Are you sure (Y/N)?";  
    $answer=<STDIN>  
}
```

若要调用一个子例程，可以使用下面两个语句行中的一个：

```
&Yesno();
```

或者

```
Yesno();
```

如果代码中已经声明了子例程，那么可以使用第二个语句（不带&）；&yesno()语句是任何位置上都能使用的。在本书中，我们将使用不带&的语句形式，虽然两种语句形式都可以使用。

当子例程被调用时，Perl能够记住它是在什么位置被调用的，并执行子例程的代码，然后，当子例程运行完成时，返回它记住的程序中的位置，如下面这个例子所示：

```
sub countdown {
    for($i=10; $i>=0; $i--) {
        print "$i -";
    }
}
print "T-minus: ";
countdown();
print "Blastoff";
```

Perl的子例程可以在程序中的任何位置进行调用，包括在其他子例程中进行调用，如下所示：

```
sub world {
    print "World!";
}
sub hello {
    print "Hello, ";
    world();
}
hello();
```

8.1.1 返回子例程的值

子例程并不只是用于按照一个便于使用的名字将代码组合在一起。子例程与 Perl的函数、运算符和表达式一样，它也有一个值。这个值称为子例程的返回值。子例程的返回值是子例程中计算的最后一个表达式的值，或者是 return语句显式返回的值。子例程的值是在子例程被调用时计算的，然后该返回值将用于调用的任何子例程中。现在请看下面这个代码：

```
sub two_by_four {      # A silly subroutine
    2*4;
}
print 8*two_by_four();
```

在上面这个代码段中，若要使 Perl计算表达式`8*two_by_four()`的值，那么子例程`two_by_four()`便开始运行，并返回值8。然后计算表达式`8*8`，并输出64。

值也可以由子例程的return语句显式返回。当你的程序需要在子例程结束之前返回，或者当你想要明确知道返回的是什么值，而不是“堕入”子例程的结尾并使用最后的表达式的值时，就需要使用return语句。下面这个代码段同时使用了两种方法：

```
sub x_greaterthan100 {
    # Relies on the value of $x being set elsewhere
    return(1) if ($x>100);
    0;
}
$x=70;
if (x_greaterthan100()) {
    print "$x is greater than 100\n";
}
```

子例程能够返回数组和哈希结构，也能返回标量，如下所示：

```
sub shift_to_uppercase {
    @words=qw( cia fbi un nato unicef );
    foreach(@words) {
        $_=uc($_);
    }
    return(@words);
}
@acronyms=shift_to_uppercase();
```

8.1.2 参数

上面的所有子例程举例都有一个共同点，那就是它们都对硬编码的数据（`2*4`）或者变量进行操作，而这些变量里边恰好拥有正确的数据（`x_greaterthan100()`的`$x`）。这个限制条件产生了一个问题，因为如果函数依赖硬编码的数据，或者希望得到函数之外的值的数据，那么这样的函数并不是真的能够移植的函数。当你调用函数时说：“取出这个数据并且用它进行某些操作”，然后在以后又调用它并且说：“取出另一些数据并且用它进行某些操作。”这样，函数的运行特性就可以根据传递给它的值来改变。

为了改变函数的运行特性而赋予函数的这些值称为参数，在本书中你都需要使用这些参数。Perl的内置函数（`grep`、`sort`、`reverse`和`print`等）都拥有一些参数，并且现在你的函数也可以拥有参数。若要传递子例程的参数，可以使用下面任何一个语句：

```
subname(arg1, arg2, arg3);
subname arg1, arg2, arg3;
&subname(arg1, arg2, arg3);
```

只有当Perl已经遇到子例程的定义时，才能使用上面不带括号的第二种参数形式。

在子例程中，被传递的参数可以通过Perl的特殊变量`@_`来访问。下面这个代码段显示了为函数传递参数（3个字符串直接量）和输出参数的情况：

```
sub printargs {
    print join(',', @_);
}
printargs('market', 'home', 'roast beef');
```

若要像下面这个例子中那样，访问传递过来的各个参数，可以使用数组`@_`上的下标，就像你对任何其他数组操作时那样。请记住，`$_[0]`（`@_`的一个下标）与标量变量`$_`毫不相干：

```
sub print_third_argument {
    print $_[2];
}
```

对`$_[3]`这样的变量名进行操作并不是一种“明确的”编程风格。拥有多个参数的函数常常为这些参数赋予一个名字，这样，就能够清楚地知道它们能够做些什么。为理解这些话的含义，请看下面这个例子：

```
sub display_box_score {
    ($hits, $at_bats)=@_;
    print "For $at_bats trips to the plate, ";
    print "he's hitting ", $hits/$at_bats, "\n";
}
display_box_score(50, 210);
```

在上面这个子例程中，数组`@_`被拷贝到列表`($hits, $at_bats)`中。`@_`的第一个元素`$_[0]`变成了`$hits`，第二个元素变成了`$at_bats`。这里使用的变量名只是为了增强可读性。



变量`@_`实际上包含了传递给子例程的原始参数的别名。如果修改了`@_`（或者修改了`@_`的任何元素），就会修改参数列表中的元素变量。如果突然进行这样的修改，将被视为一种不好的做法，你的函数不应该干扰来自函数调用者的参数，除非函数的使用者要求这样做。

8.1.3 传递数组和哈希结构

传递给子例程的参数不一定是标量。你可以将数组和哈希结构传递给子例程，但是这样

做需要三思而后行。将数组或哈希结构传递给子例程时使用的方法与传递标量的方法一样：

```
@sorted_items=sort_numerically(@items);
```

在该子例程中，整个数组 @items 通过 @_ 进行引用：

```
sub sort_numerically {
    print "Sorting...";
    return(sort { $a <=> $b } @_);
}
```

当将数组和哈希结构传递给子例程时，会遇到一个小小的困难。将两个或多个哈希结构（或数组）传递给子例程，通常并不执行你想要做的操作。请看下面这个代码段：

```
sub display_arrays {
    (@a, @b)=@_;
    print "The first array: @a\n";
    print "The second array: @b\n";
}
display_arrays(@first, @second);
```

@first 和 @second 两个数组一道被放入一个列表，各个元素则在调用子例程时放入 @_ 中。@first 的各个元素的结尾与 @_ 中的 @second 的元素的开始是无法区分的，它只是一个大型平面列表。在子例程中，赋值语句 (@a, @b) = @_ 取出 @_ 中的所有元素，并将它们赋予 @a。数组 @b 没有得到任何元素。（其原因已经在第 4 学时中做了介绍。）

标量的混合体可以用单个数组或哈希结进行传递，只要标量在参数列表中被首先传递，并且知道有多少个标量。这样，哈希结构或数组就包含了最后一个标量以外的所有值，正如下面这个例子中的情况一样：

```
sub lots_of_args {
    ($first, $second, $third, %hash)=@_;
    # rest of subroutine...
}
lots_of_args($foo, $bar, $baz, %myhash);
```

如果必须将多个数组和哈希结构传递到一个子例程中（并且能够在以后对它们进行区分），则必须使用“引用”。我们将在第 13 学时中介绍如何传递引用的方法。

8.2 作用域

在本学时的开头，我们介绍了子例程被用来取出一些代码片，并将它们捆绑在一起，再给它们赋予一个名字。然后就可以使用这个名字在需要的时候执行该代码。子例程还允许你取出子例程中的代码，并使它独自运行。这就是说，你可以让它只使用它的参数、该语言的内置函数和表达式来运行，以产生一个返回值。然后你可以在其他程序中重复使用该函数，因为该函数不再依赖它被调用时所在的上下文，它只是取出它的参数，即内部数据，然后产生一个返回值。该函数将变成一个黑匣子，数据可以进去，也可以出来，你不必从外面关心发生了什么事情。这称为纯函数。

现在请看下面两个代码段：

```
# One fairly good way to write this function
sub moonweight {
    ($weight)=@_;
    return($weight/6);
}
print moonweight(150);
```

```
# A poor way to write this function.
sub moonweight {
    return($weight/6);
}
$weight=150;
print moonweight;
```

从长远来看，上面显示的第一个代码段在某种程度上说要好一些。它不要求设置任何外部变量，即函数外边的变量。它使用它的参数（它拷贝到 \$weight 的参数），然后进行计算。第二个实现代码不容易在另一个程序中重复使用，必须确保 \$weight 已经正确地进行设置，并且没有用于某些其他值。如果它被用于其他某个值，你就必须编辑 moonweight() 函数，以便使用一个不同的变量。这样做并不非常有效。

因此，上面的第一个例子是个比较好的函数，但是它仍然忽略了某些东西。变量 \$weight 可能与程序中的其他某个位置上的名字为 \$weight 的变量发生冲突。

Perl 允许你在大型程序中为了不同的目的一次又一次地重复使用变量名。按照默认设置，在你的程序的主体中和子例程中，Perl 的变量是可视的，这些类型的变量称为全局变量。

你要做的工作是使变量成为函数的专用变量。为此，必须使用 my 操作符：

```
sub moonweight {
    my $weight;
    ($weight)=@_;
    return($weight/1.66667);
}
```

在 moonweight() 中，\$weight 现在是个专用变量。程序中的其他函数都不能访问 \$weight 的值。带有 \$weight 名字的任何其他变量均与 moonweight() 函数的 \$weight 完全隔开。这个子例程现在完全是个独立的子例程。

可视变量的这部分程序称为变量的作用域。

可以使用 my 操作符来声明标量、数组和哈希结构的变量是子例程的专用变量。例如，文件句柄、子例程和 Perl 的特殊变量 \$!、\$_ 和 @_ 都不能标记为子例程的专用变量。如果将括号用于 my 操作符，你就能声明多个专用变量：

```
my($larry, @curly, %moe);
```

子例程的专用变量与全局变量的存储方式是完全不同的。全局变量和专用变量可以拥有相同的名字，但是它们互相之间毫不相干，如下所示：

```
sub myfunc {
    my $x;
    $x=20;      # This is a private $x
    print "$x\n";
}
$x=10;          # This is a global $x
print "$x\n";
myfunc();
print "$x\n";
```

上面这个代码段将输出 10、20，然后输出 10。myfunc() 子例程中的 \$x 与该子例程外面的 \$x 是完全不同的。子例程有可能同时使用它的专用 \$x 和全局 \$x 吗？答案是肯定的，不过答案有些复杂，这个问题不在 Perl 入门书籍要讲解的范围之内。

大部分时候，Perl 子例程首先要将 @_ 赋予一个变量名的列表，然后声明该列表是子例程的专用列表：

```
sub player_stats {
```

```
my($at_bats, $hits, $walks)=@_;
# Rest of function...
}
```

这种方法能够创建一个与程序员友好的函数，它的变量都是函数的专用变量，因此它们不会影响其他的函数，或者受其他函数的影响（包括程序的主体）。当子例程运行结束时，所有专用变量均被撤消。

my操作符的其他用法

你为变量声明的作用域也可以小于这个子例程。My操作符声明的变量的作用域实际上可以包含一个属于子例程块的代码块。例如，在下面这个代码段中，专用变量 \$y（用my声明的这个变量）只有在该代码块中才能看到：

```
$y=20;
{
    my $y=500;
    print "The value of \$y is $y\n";    # Will print 500
}
print "$y\n";                      # Will print 20.
```

这个声明甚至可以在控制结构 for、foreach、while或if中出现。实际上，在你拥有代码块的任何地方，都可以为变量设定作用域，这样，它就只能在该代码块中才能看到，如下面这个例子中那样：

```
while($testval) {
    my $stuff;      # Only visible within the while() loop.
    :
}
foreach(@t) {
    my %hash;      # Only visible within the foreach loop.
}
```

在上面这个代码段中，每次通过这个循环时，便会创建 my的新变量\$stuff和%hash。

Perl的5.004和更新的版本允许将 for和foreach循环中的迭代器以及 while和if中的测试条件声明为代码块的专用迭代器和测试条件：

```
foreach my $element (@array) {
    # $element is only visible in the foreach()
}

while(my $line=<STDIN>) {
    # $line is visible only in the while()
}
```

同样，当包含的代码块运行结束时，代码块的任何专用变量及其值均被撤消。

8.3 练习：统计数字

既然你已经懂得了函数的基本概念，那么应该开始进一步了解将代码封装在独立函数中的好处。函数提供了可以很容易重复使用的代码。在下面这个练习中，有 3个函数能够对几个数字组进行分析。

让我们回顾一下在学校中学过的一些知识。一组数字的平均值（也称为算术平均值）只是所有数字的平均值。中项值是你给一组数字排序时位于中间的这个数字的值。如果元素的数量是偶数，那么中项值是位于中间的两个数字的平均值。标准偏差的概念是指平均值附

近的数字“聚合”有多密。大标准偏差意味着数字分布得很宽，而小标准偏差则意味着数字在平均值附近聚合得很紧密。平均值加上或减去标准偏差用于代表大约 68% 的数字集，而平均值加上或减去两个标准偏差则代表 95% 的数字集。

使用文本编辑器，键入程序清单 8-1 的程序，将它保存为 stats。务必根据第 1 学时中的说明使该程序成为可执行程序。

当完成上述操作后，键入下面的命令，以运行该程序：

```
perl stats
```

程序清单 8-1 Stats 程序的完整清单

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  sub mean {
5:      my(@data)=@_;
6:      my $sum;
7:      foreach(@data) {
8:          $sum+=$_;
9:      }
10:     return($sum/@data);
11: }
12: sub median {
13:     my(@data)=sort { $a <= $b } @_;
14:     if (scalar(@data)%2) {
15:         return($data[@data/2]);
16:     } else {
17:         my($upper, $lower);
18:         $lower=$data[@data/2];
19:         $upper=$data[@data/2 - 1];
20:         return(mean($lower, $upper));
21:     }
22: }
23: sub std_dev {
24:     my(@data)=@_;
25:     my($sq_dev_sum, $avg)=(0,0);
26:
27:     $avg=mean(@data);
28:     foreach my $elem (@data) {
29:         $sq_dev_sum+=($avg-$elem)**2;
30:     }
31:     return(sqrt($sq_dev_sum/@data-1));
32: }
33: my($data, @dataset);
34: print "Please enter data, separated by commas: ";
35: $data=<STDIN>; chomp $data;
36: @dataset=split(/[\s,]+/, $data);
37:
38: print "Median: ", median(@dataset), "\n";
39: print "Mean: ", mean(@dataset), "\n";
40: print "Standard Dev.: ", std_dev(@dataset), "\n";

```

第 1 行：这一行包含了到达解释程序的路径（可以改变这个路径，使之适合系统的需要）和开关-w。请使警告特性始终处于激活状态。

第 3 行：命令 use strict 意味着所有变量必须用 my 来声明，裸单词必须用引号括起来。

第 4~11 行：使用 foreach 循环，使 mean() 函数运行，将 \$sum 中的所有数字相加，然后除

以数字的数量。

第12~21行：median（）函数以两种方式运行。如果元素是个奇数，它只是选择中间的元素，方法是取出数组的长度，再除以2，然后使用它的中间部分。如果元素的数量是偶数，它进行相同的操作，但是它取出两个中间的数字，然后使用mean（）函数计算这两个在\$upper和\$lower中的数字的平均数，并将它作为中间值返回。

第23~32行：std_dev（）函数非常简单，不过它主要是个数学运算函数。简单说来，从平均数中减去@data中的每个元素值，再求它的平方值，然后将产生的结果在\$sq_dev_sum中累加。若要求得标准偏差，则用平方差的合计值除以元素的数量减1，然后求它的平方根。

第33~35行：程序的主体中需要的变量被声明为词法单位（使用my进行声明），并提示用户输入\$data。然后使用模式/[\s,]+/将变量\$data分割成数组@dataset。该模式按照逗号和空格对该行进行分割。额外的空格和逗号被忽略。

第38~40行：产生输出。请注意，这并不是调用函数mean（）、median（）和std_dev（）的惟一位置。这些函数也可以互相调用，std_dev（）与median（）同时使用mean（），这是重复使用代码的很好的例子。

程序清单8-2给出了一个Stats程序输出的示例。

程序清单8-2 Stats程序的输出示例

```
Please enter data, separated by commas: 14.5,6,8,9,10,34
Median: 9.5
Mean: 13.5833333333333
Standard Dev.: 10.3943093405318
```

8.4 函数的脚注

现在你已经懂得了作用域的概念，有些操作只能用作用域才能有效地进行。可用函数之一是递归函数，另一个是Perl语句use strict，它能够激活更严格的Perl，使你能够避免在编程中出现错误。

8.4.1 声明local变量

Perl的第4版并不配有真正意义上的“专用”变量。相反，Perl 4的变量只能称为“准”专用变量。这个“准”专用变量的概念在Perl 5中仍然存在。若要声明这些变量，可以像下面这个例子中那样，使用local操作符：

```
sub myfunc {
    local($foo)=56;
    # rest of function...
}
```

在上面这个代码段中，\$foo被声明为myfunc（）子例程的局部变量。用local声明的变量的作用与使用my声明的变量几乎相同，它的作用域可以局限于一个子例程，一个代码块，或者eval，它的值将在退出子例程或代码块时被撤消。

它们之间的差别是：声明为局部变量的那些变量，可以在它的作用域范围内的代码块中看到，也可以在从该代码块中调用的任何子例程中看到。表8-1显示了这两种变量之间的逐项比较。

表中显示的两个代码段基本相同，差别在于myfunc（）中对\$foo的声明不一样。在左边，它用my进行声明，而在右边，它用local来声明。

表8-1 my变量与local变量的比较

```

sub mess_with_foo {
    $foo=0;
}

sub myfunc {
    my $foo=20;
    mess_with_foo();
    print $foo;
}
myfunc();
```

```

sub mess_with_foo {
    $foo=0;
}

sub myfunc {
    local $foo=20;
    mess_with_foo();
    print $foo;
}
myfunc();
```

当左边的代码运行时，创建的\$foo是myfunc()的专用变量。当mess_with_foo()被调用时，在mess_with_foo()中被修改的\$foo是全局变量\$foo。当控制返回给myfunc()时，输出的值是20，因为myfunc()中的\$foo从来没有改变。

当右边的代码运行时，\$foo被创建并声明为myfunc()的局部变量。当mess_with_foo()被调用时，\$foo被设置为0。该\$foo与myfunc()返回的\$foo相同，“专用性”传递到了被调用的子例程。当返回到myfunc()时，输出0这个值。



如果你对术语非常讲究，局部变量在技术上可以称为动态作用域变量，因为它们的作用域可以随着被调用的子例程而变化。用my声明的变量可以称为词法作用域变量，因为它们的作用域只需要通过读取代码并指明它们声明时所在的代码块来决定，该作用域是不变的。

每当你的程序需要子例程专用的变量时，你几乎总是想要一个用my声明的变量。

8.4.2 使Perl变得更加严格

Perl是一种比较随意的编程语言。它并不试图限制你的编程操作，它允许你在完成工作时不会过多地抱怨代码的外在形式。你也可以使Perl对你的代码更加严格一些。例如，如果你在命令行或者#!行上使用警告开关，那么Perl能够帮助你避免犯一些愚蠢的错误。当你使用未定义的变量，或者仅仅使用一次变量名时，Perl就会向你发出警告。

在较大的软件项目中，或者在你的程序变得越来越大的时候，就需要让Perl帮助你对程序有所约束。除了使用-w开关外，也可以在编译时告诉Perl解释程序打开更多的警告消息。可以使用use strict来进行这项操作：

```

strict;

use strict;
sub mysub {
    my $x;
    :
}
mysub();
```

use strict语句实际上可以称为编译器命令。它能够告诉Perl，给下列情况做上指向当前代码块或文件中的运行期错误的标志：

- 试图使用不是用 my 声明的变量名（不是特殊变量的名字）。
- 当函数定义尚未设置时，试图将裸单词用作函数名。
- 其他潜在的错误。

现在，use strict命令能够帮助你避免产生前面两个问题。让Perl给没有使用my声明的变量做上标志，就可以在你实际上打算使用专用变量时，避免使用全局变量。这是帮助你编写更具独立性的代码并且不依赖全局变量的一种方法。

use strict解决的最后一个问题是裸关键字的问题。请看下面这个代码：

```
$var=value;
```

在这个例子中，你打算将 value解释为一个函数调用还是一个字符串呢（但是你忘记了引号）？Perl的use strict命令会指出这个代码是个含糊的代码，并且不允许使用这种句法，除非在到达该语句之前已经对子例程的值进行了声明。

从现在起，将把use strict命令纳入本书中的所有的练习和较长的程序清单中。

8.4.3 递归函数

你早晚都会遇到一个特殊的子例程类别。这些子例程实际上通过调用它们自己来执行它们的操作。这些子例程称为递归子例程。

每当一些任务能够分割成较小的任务和较小的相同任务的时候，就可以使用递归子例程。比如，一个递归子例程正在搜索一个目录树，以便寻找一个文件。当搜索了最上面的目录后，找到了子目录，因此又必须搜索这些子目录。在这些子目录中，如果又找到了下一个层子目录，那么又必须搜索下一个层子目录。在这里我们可以看到一种模式。

另一种递归任务是计算阶乘，它常用于统计学。字母 ABCDEF排列方法的数量是6的阶乘。阶乘是一个数与所有较小的数（最小为1）的乘积。因此，6的阶乘是 $6 \times 5 \times 4 \times 3 \times 2 \times 1$ ，即720。若要计算6的阶乘，你必须计算5的阶乘，再将它乘以6。若要计算5的阶乘，你必须计算4的阶乘，再将它乘以5，等等。程序清单8-3显示了计算阶乘时使用的递归函数。

程序清单8-3 计算阶乘的递归函数

```

1:  sub factorial {
2:      my ($num)=@_;
3:      return(1) if ($num <= 1);
4:      return($num*factorial($num-1));
5:  }
6:  print factorial(6);

```

第2行：factorial（）子例程的参数被拷贝到\$num，它声明为该子例程的专用变量。

第3行：每个递归函数都需要有一个终止条件。也就是说，需要设定一个位置，在这个位置上，该函数不再能够调用自己以便获得一个答案。对于 factorial（）子例程来说，终止条件是1的阶乘（或0的阶乘）。这两个阶乘的值都是1。当使用1或0（\$num<=1）调用factorial（）子例程时，它就执行return（1）。

第4行：否则，如果参数不是0或1，那么必须计算下一个较小的数列的阶乘。如果\$num分别是：6、5、4、3、2、1，则第4行就分别计算下面的值：返回（ $6 \times$ 阶乘（5））返回（ $5 \times$ 阶乘（4））返回（ $4 \times$ 阶乘（3））返回（ $3 \times$ 阶乘（2））返回（ $2 \times$ 阶乘（1））不能到达第4行。factorial（1）返回1。

当下一个较小的阶乘计算后（一直计算到 1），该函数开始返回上面各个函数调用序列的值，直到最后能够计算 6 的阶乘为止。

递归函数并不是个常用的函数。大的递归函数创建起来非常复杂，而且很难调试。凡是可以通过迭代（使用 for、while 和 foreach）来执行的任务都可以使用递归函数来进行操作，同时，任何递归任务也可以通过迭代来完成。递归函数通常保留用于执行少数任务，目的是使它们执行起来容易一些。

8.5 课时小结

Perl 支持用户定义的函数，这种函数称为子例程，子例程的运行特性与内置函数相同，它们可以带有参数，执行操作，然后在必要时将各个值返回给调用者。Perl 的函数能够调用其他函数，甚至能够调用它们自己。Perl 还允许声明对一个函数（或任何代码块）来说专用的变量，并能创建可以重复使用的独立的代码块。

8.6 课外作业

8.6.1 专家答疑

问题：在调用函数时，使用和不使用 & 有没有真正的区别？

解答：现在你还没有必要考虑这个区别。当使用函数的原型或者当你调用的函数没有括号时，在\$foo 与 foo 之间存在一个很小的差别。这些问题不在本书讲解的范围之内，但是为了满足用户的好奇心，在 perlsub 手册页中，对这个问题做了介绍。

问题：当我在程序中使用 my (\$var) 时，Perl 报了一条消息：syntax error, next 2 tokens my(，这是什么意思？

解答：可能出现了键入错误，也可能安装了 Perl 4 这个版本。请在命令提示符后面键入 perl -v。如果 Perl 报出的版本是第 4 版，那么你应该立即进行版本升级。

问题：应该如何将函数、文件句柄、多个数组或哈希结构传递（或返回）给子例程？

解答：若要传递函数、多个数组和哈希结构，必须使用引用，这个问题将在第 13 学时中介绍。若要将文件句柄传递给子例程，或者从子例程那里接收文件句柄，必须使用称为 typeglob 的工具，或者使用 IO::Handle 模块。这两个内容都不属于本书讲解的范围。

问题：我使用的一个函数返回了许多值，但是我只对其中的一个值有兴趣，我应该如何跳过其他的返回值？

解答：方法之一是用函数建立一个列表，方法是将整个函数调用放在括号中。当它是个列表时，你就可以使用普通的列表块来获得该列表的各个部分的信息。下面这个代码只是从内置函数 localtime（它实际上有 9 个返回值）中取出年份（当前年份减去 1900）：

```
print "It is now ", 1900 + (localtime [5])
```

另一种方法是将来自函数的返回值赋予一个列表，并且将不需要的值赋予 undef 或哑变量：

```
(undef, undef, undef, undef, undef, $year_offset) = localtime;
```

8.6.2 思考题

观察下面这个代码段：

```

sub bar {
    ($a,$b)=@_;
    $b=100;
    $a=$a+1;
}
sub foo {
    my($a)=67;
    local($b)=@_;
    bar($a, $b);
}
foo(5,10)

```

1) 当你运行 `bar($a,$b)` 后，\$b 中的值是什么？

- a. 5
- b. 100
- c. 68

2) `foo()` 的返回值是什么？

- a. 67
- b. 68
- c. undef

3) 在 `foo()` 中，\$b 是什么？

- a. 词法规定的作用域
- b. 动态作用域
- c. 全局作用域

8.6.3 解答

- 1) 答案是 b。\$b 在 `foo()` 中声明为 `local`，因此每次调用的子例程均共享同一个 \$b 值（除非它们后来再次用 `local` 或 `my` 声明了 \$b）。调用 `bar()` 后，在 \$b 被修改的地方，\$b 被设置为 100。
- 2) 答案是 b。`foo()` 中的最后一个语句是 `bar($a, $b)`。`bar()` 返回 68。因为 \$a 的值被传递给 `bar`，并且它被递增了。`foo()` 返回最后一个表达式的值，它是 68。
- 3) 答案是 b。用 `local` 声明的变量是动态调用的作用域变量。

8.6.4 实习

- 使用本学时的统计练习中的函数和第 7 学时中的单词统计代码来观察文档中单词的长度。计算它们的平均值、中间值和标准偏差。
- 编写一个函数，输出斐波纳奇数列的一部分。这个数列的开始部分是 0, 1, 1, 2, 3, 5, 8，它可以永远延续下去。斐波纳奇数列是数学和自然界中的一种递归模式。后面这个数是前两个数的和（0 和 1 是例外）。这些数可以用迭代方式和递归方式进行计算。

China-pub.com

下载

China-pub.com

下载

第二部分 高级特性

第9学时 其他函数和运算符

第10学时 文件与目录

第11学时 系统之间的互操作性

第12学时 使用Perl的命令行工具

第13学时 引用与结构

第14学时 使用模块

第15学时 了解程序的运行性能

第16学时 Perl 语言开发界

China-pub.com

下载

第9学时 其他函数和运算符

Perl遵循的传统原则是“一件事情可以使用许多方法来完成”。在本学时中，我们将要更加深入地掌握这个原则。我们将要学习丰富多彩的新函数和运算符。

为了进行标量搜索和操作，到现在为止我们一直使用正则表达式。不过我们可以使用多种方法来完成这项任务，Perl提供了各种各样的函数，以便对标量进行搜索和编辑。在本学时中，我们将要介绍其他的几种方法。

另外，我们介绍了作为项目的线性列表的数组，你可以使用 foreach迭代通过这些列表，或者使用join将它们组合起来，构成标量。在本学时中，我们将要介绍一种观察数组的全新方法。

最后，我们要重新介绍一下常用的 print函数，并且给它增加一点特性。使用新的改进后的print函数，你就能够编写格式优美、适合向他人展示的报表。

在本学时中，你将要学习：

- 如何对标量进行简单的字符串搜索。
- 如何进行字符替换。
- 如何使用print函数。
- 如何将数组用作堆栈和队列。

9.1 搜索标量

正则表达式非常适合对标量进行搜索，以便找出你要的模式，但是有时使用正则表达式来搜索标量有点像杀鸡用牛刀的味道。在 perl中，对模式进行组装，然后在标量中搜索该模式，需要花费一定的开销，不过这个开销并不大。另外，当你编写正则表达式时，很容易出错。为此，perl提供了若干个函数，用于对标量进行搜索，或者从标量中取出简单的信息。

9.1.1 用index进行搜索

如果你只想在另一个标量中搜索单个字符串，Perl提供了index函数。index函数的句法如下：

```
index string, substring
index string, substring, start_position
```

index函数从string的左边开始运行，并搜索substring。index返回找到substring时所在的位置，0是指最左边的字符。如果没有找到substring，index便返回-1。被搜索的字符串可以是字符串直接量，可以是标量，也可以是能够返回字符串值的任何表达式。substring不是一个正则表达式，它只是另一个标量。

请记住，你编写的Perl函数和运算符可以带有包含参数的括号，也可以不带。下面是一些例子：

```
index "Ring around the rosy", "around";      # Returns 5
index("Pocket full of posies", "ket");        # Returns 3
$a="Ashes, ashes, we all fall down";
index($a, "she");                            # Returns 1
index $a, "they";                           # Returns -1 (not found)
@a=qw(oats      peas      beans);
index join(" ", @a), "peas";                 # Returns 5
```

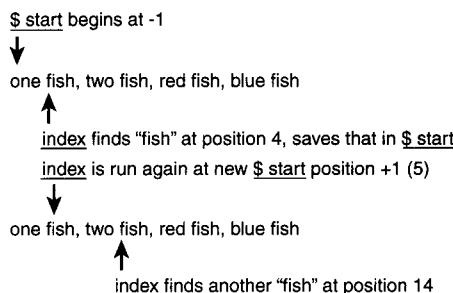
根据情况，可以给 index 函数规定一个字符串中开始进行搜索的起始位置，如下面的例子显示的那样。若要从左边开始搜索，使用的起始位置是 0：

```
$reindeer="dasher dancer prancer vixen";
index($reindeer, "da");           # Returns 0
index($reindeer, "da", 1);        # Returns 7
```

也可以使用带有起始位置的 index 函数，以便“遍历”一个字符串，找到出现一个较短字符串的所有位置，如下所示：

```
$source="One fish, two fish, red fish, blue fish.";
$start=-1;
# Use an increasing beginning index, $start, to find all fish
while(($start=index($source, "fish", $start)) != -1) {
    print "Found a fish at $start\n";
    $start++;
}
```

上面这个代码滑动通过 \$source，如下所示：



9.1.2 用rindex向后搜索

函数 rindex 的作用与 index 基本相同，不过它是从右向左进行搜索。它的句法如下所示：

```
rindex string, substring
rindex string, substring, start_position
```

当搜索到结尾时，rindex 返回 -1。下面是一些例子：

```
$a="She loves you yeah, yeah, yeah.";
rindex($a, "yeah");      # Returns 26.
rindex($a, "yeah", 25);  # Returns 20
```

用于 index 的遍历循环与使用 rindex 进行向后搜索的循环略有不同。rindex 的起点必须从字符的结尾开始，或者从结尾的后面开始，(在下例中，从 length(\$source) 开始)，但是，当返回 -1 时，它仍然应该结束运行。当找到每个字符串后，\$start 必须递减 1，而不是像 index 那样递增 1。

```
$source="One fish, two fish, red fish, blue fish.";
$start=length($source);
while(($start=rindex($source, "fish", $start)) != -1) {
    print "Found a fish at $start\n";
    $start--;
}
```

9.1.3 用substr分割标量

substr 是个常常被忽略和很容易被遗忘的函数，不过它提供了一种从标量中取出信息并对标量进行编辑的通用方法。substr 的句法如下：

```
substr string, offset
substr string, offset, length
```

substr函数取出 string，从位置offset开始运行，并返回从 offset到结尾的字符串的剩余部分。如果设定了length，那么取出length指明的字符，或者直到找出字符串的结尾，以先到者为准，如下例所示：

```
#Character positions in $a
#   0      10     20      30
$a="I do not like green eggs and ham.";
print substr($a, 25);      # prints "and ham."
print substr($a, 14, 5);   # prints "green"
```

如果offset设定为负值，substr函数将从右边开始计数。例如，substr (\$a,-5) 返回 \$a 的最后5个字符。如果length设定为负值，则 substr 返回从它的起点到字符串结尾的值，少于 length 指明的字符，如下例所示：

```
print substr($a, 5, -10);    # prints "not like green egg"
```

在上面这个代码段中，substr从位置5开始运行，返回字符串的剩余部分，但不包含最后10个字符。

你也可以使用赋值表达式左边的 substr 函数。当用在左边时，substr 用于指明标量中的什么字符将被替换。当用在赋值表达式的左边时，substr 的第一个参数必须是个可以赋值的值，比如标量变量，而不应该是个字符串直接量。下面是使用 substr 对字符串进行编辑的一个例子：

```
$a="countrymen, lend me your wallets";
# Replace first character of $a with "Romans, C"
substr($a, 0, 1)="Romans, C";

# Insert "Friends" at the beginning of $a
substr($a, 0, 0)="Friends, ";

substr($a, -7, 7)="ears.";           # Replace last 7 characters.
```

9.2 转换而不是替换

下一个运算符是转换运算符（有时称为翻译运算符），它使我们想起正则表达式中的替换的操作方式。替换操作符的形式是 s/pattern/replacement/，在第6学时中我们已经作了介绍。除非你用连接运算符 = ~ 设定了另一个标量，否则该操作符将对 \$ _ 变量进行操作。转换操作符的作用与它有些类似，不过它并不使用正则表达式，而且它的运行方式完全不同。转换操作符的句法如下所示：

```
tr/searchlist/replacementlist/
```

转换操作符 tr/// 用于搜索一个字符串，找出 searchlist 中的各个元素，并用 replacementlist 中的对应元素对它们进行替换。按照默认设置，转换操作符用于对变量 \$ _ 进行搜索和修改。若要搜索和修改其他变量，你可以像使用正则表达式进行匹配操作那样，使用连接运算符，如下所示：

```
tr/ABC/XYZ/;          # In $_, replaces all A's with X's, B's with Y's, etc..
$r=~tr/ABC/XYZ/;     # Does the same, but with $r
```

字符的逻辑分组之间可以使用连字符。例如 A - Z 代表大写字母 A 到 Z，这样你就不必把它们全部写出来，请看下例：

```
tr/A-Z/a-z/;          # Change all uppercase to lowercase
tr/A-Za-z/a-zA-Z/;    # Invert upper and lowercase
```

如果 replacementlist 是空的，或者与 searchlist 相同，那么 tr/// 将计算并返回匹配的字符。目

标字符串并不被修改，如下例所示：

```
$eyes=$potato=~tr/i//;      # Count the i's in $potato, return to $eyes
$numbs=tr/0-9//;           # Count digits in $_, return to $nums
```

最后要说明的是，由于历史的原因，tr///也可以写成y///，其结果相同，因为y与tr同义。tr///运算符（和y///）也允许你为searchlist和replacementlist设定另一组界限符。这些界限符可以是任何一组自然配对的字符，如括号或任何其他字符，请看下面的例子：

```
tr(a-z)(n-zA-m);          # Rotate all characters 13 to the left in $_
y[,-_-][;:=|];            # Switch around some punctuation
```



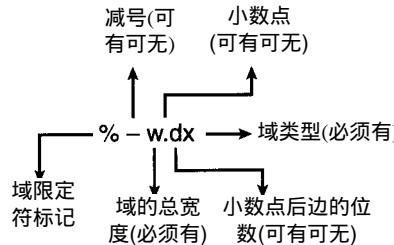
tr///运算符实际上还具备另外一些功能，不过用得不多。若要了解tr///能够执行的所有其他任务，请查看perlop节中的在线文档。

9.3 功能更强的print函数

print函数是个非常简单的输出函数，它几乎不具备任何格式化功能。为了更具体地控制输出操作，如左对齐和右对齐，十进制精度，以及固定宽度的输出，你可以使用Perl的printf函数。printf函数是从C编程语言那里借用的（几乎是原原本本的借用），不过其他编程语言也配有类似的函数，如BASIC的print using函数。printf函数的句法如下：

```
printf formatstring, list
printf filehandle formatstring, list
```

formatstring是一个描述输出格式的字符串，下面我们很快就要对它进行介绍。list是一个你想让printf显示的值的列表，它类似print语句中的list。通常而言，printf将它的输出显示给STDOUT文件句柄，但与print一样，如果你设定了一个文件句柄，那么printf就使用该文件句柄。请注意，filehandle名与formatstring之间不使用逗号。



通常情况下formatstring是个字符串直接量，它也可以是一个用来描述输出格式的标量，formatstring中的每个字符均按其原义输出，但是以%开头的字符则属例外。%表示这是一个域说明符的开始。域说明符的格式是% -w.dx，其中w是域需要的总宽度，d是小数点左边的位数（对于数字来说）和字符串域允许的总宽度，x表示输出的是数据类型。x说明符前面的连字符表示该域在w字符中左对齐，否则它进行右对齐。只有%和x是不可少的。表9-1列出了一些不同类型的域说明符。

表9-1 Printf函数的部分域说明符列表

域类型	含义
c	字符
s	字符串
d	十进制整数；截尾的小数
f	浮点数

完整的域说明符列表请参见在线手册。你可以在命令提示符后面键入 `perldoc -f printf`，以查看该列表。

下面是使用 `printf` 的一些例子：

```
printf("%20s", "Jack");           # Right-justify "Jack" in 20-characters
printf("%-20s", "Jill");          # Left-justify "Jill" in 20 characters
$amt=7.12;
printf("%6.2f", $amt);            # prints "    7.12"
$amt=7.127;
printf("%6.2f", $amt);            # prints "    7.13", extra digits rounded
printf("%c", 65);                # prints ASCII character for 65, "A"
$amt=9.4;
printf("%6.2f", $amt);            # prints "    9.40"
printf("%6d", $amt);              # prints "      9"
```

每个格式说明符均使用列表中的一个项目，如上所示。对于每个项目来说，都应该有一个格式说明符；对于每个格式说明符来说，都有一个列表元素：

```
printf("Totals: %6.2f %15s %7.2f %6d", $a $b $c $d);
```

若要输出数字中的前导 0，只需要在格式说明符中的宽度的前面设置 1 个 0，如下所示：

```
printf("%06.2f", $amt); # prints "009.40"
```

`sprintf` 函数与 `printf` 几乎相同，不过它不是输出值，而是输出 `sprintf` 返回的格式化输出，你可以将它赋予一个标量，或者用于另一个表达式，如下所示：

```
$weight=85;
# Format result nicely to 2-decimals
$moonweight=sprintf("%.2f", $weight*.17);
print "You weigh $moonweight on the moon."
```

请记住，带有 `%f` 格式说明符的 `printf` 和 `sprintf` 函数能够将计算结果圆整为你指定的小数点位数。

9.4 练习：格式化报表

当你使用计算机时，必然要处理的一项任务是将原始数据格式化为一个报表。计算机程序能够按照不同于人类阅读的格式对数据进行交换，常见的任务是使用该数据，并将它格式化为人能够阅读的报表。

为了进行这个练习，我们为你提供了一组员工记录，它包含关于某些虚构员工的信息，包括每小时的工资、工作的小时数、名字和员工号码。这个练习使用这些数据将它们重新格式化为一个很好的报表。

你可以很容易地修改这种类型的程序，以便输出其他类的报表。这个练习的数据包含在一个在程序开始初始化的数组中。在真实的报表中，数据可能来自磁盘上的一个文件。后面我们还要修改这个练习，以便使用外部的文件。

使用文本编辑器，键入程序清单 9-2 中的程序，并将它保存为 `Employee`，不要键入行号。按照第 1 学时中的说明，使该程序成为可执行程序。

当完成上述操作后，在命令行提示符后面键入下面的命令，设法运行该程序。

```
Perl Employee
```

程序清单 9-1 显示了 `Employee` 程序的输出举例。

程序清单 9-1 `Employee` 程序的输出

2:	132912 Alice Franklin	10.15	35	355.25
3:	141512 Wendy Ng	9.50	40	380.00
4:	123101 Bob Smith	9.35	40	374.00
5:	198131 Ted Wojohowicz	6.50	39	253.50

程序清单 9-2 Employee 程序的完整清单

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:
5:  my @employees=(
6:      'Smith,Bob,123101,9.35,40',
7:      'Franklin,Alice,132912,10.15,35',
8:      'Wojohowicz,Ted,198131,6.50,39',
9:      'Ng,Wendy,141512,9.50,40',
10:     'Cliburn,Stan,131211,11.25,40',
11: );
12:
13: sub print_emp {
14:     my($last,$first,$emp,$hourly,$time)=
15:         split(',',$_[0]);
16:     my $fullname;
17:     $fullname=sprintf("%s %s", $first, $last);
18:     printf("%6d %-20s %6.2f %3d %7.2f\n",
19:             $emp, $fullname, $hourly, $time,
20:             ($hourly * $time)+.005 );
21: }
22:
23: @employees=sort {
24:     my ($L1, $F1)=split(',',$a);
25:     my ($L2, $F2)=split(',',$b);
26:     return( $L1 cmp $L2 # Compare last names
27:            || # If they're the same...
28:            $F1 cmp $F2 # Compare first
29:        );
30:     } @employees;
31:
32: foreach(@employees) {
33:     print_emp($_);
34: }
```

第1行：这一行包含到达解释程序的路径（可以更改这个路径，使之适合系统的需要）和开关-w。请始终使警告特性处于激活状态。

第3行：use strict命令意味着所有变量都必须用my进行声明，裸单词必须用引号括起来。

第5~11行：员工列表被赋予@employees。数组中的每个元素均包含名字、姓、员工号、计时工资和工作的小时数。

第23~30行：@employees数组按姓和名字排序。

第24行：被排序的第一个元素（\$a）分割为各个域。姓被赋予\$L1，名字被赋予\$F1。两者都用my声明为排序块的专用变量。

第25行：对另一个元素\$b进行与上面相同的操作。姓名被赋予\$L2和\$L1。

第26~29行：使用类似第4学时中的程序清单4-1介绍的顺序，按字母对各个名字进行比较。

第32~34行：@employees中的已排序列表被传递给print-emp()，每次传递一个元素。

第13~21行：print-emp()函数输出格式化很好的员工记录。

第14~15行：传递来的记录\$_[0]被分割成各个域，并被赋于变量\$last，\$first等，它们都是该子例程的专用变量。

第17行：名字和姓被合并为单个域，这样，两个域就可以放入某个宽度，并一道对齐。

第18~20行：记录被输出。\$hours与\$time相乘，得出合计金额。金额.005与合计相加，这样，当乘积被截尾而成为两位数时，它能正确地圆整。

9.5 堆栈形式的列表

到现在为止，列表（和数组）一直是作为线性数据数组来展示的，其索引用于指明每个元素。

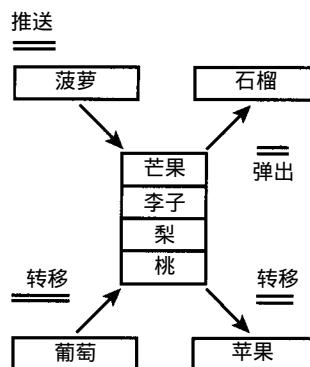
0	1	2	3	4	5
苹果	桃	梨	李子	芒果	石榴

请使用你的想象力，将各个组数元素设想为一个纵向堆栈。

在计算机术语中，这种列表称为堆栈。堆栈可用于按顺序来处理的累计操作。Klondike Solitaire游戏就是一个很好的例子。7堆牌中的每一堆牌分别代表一个堆栈；开始时，这些牌面向下放入堆栈。当需要这些牌时，它们被翻过来，并从堆栈中取走，再将其他牌放在新翻过来的牌的上面。

Perl中的堆栈通常用数组来实现。若要将各个项目放在堆栈的顶部，可以使用push函将项目压入堆栈；若要将项目从堆栈的顶部取出，可以使用pop函数。

石榴
芒果
李子
梨
桃
苹果



另外，堆栈可以从底部进行修改，可以将它视为从一叠纸牌的底部对它进行处理一样。

Shift函数用于将元素添加到堆栈的底部，unshift用于从底部取出元素。每个函数的句法如下：

```
pop target_array
```

```
shift target_array
```

```
unshift target_array, new_list
```

```
push target_array, new_list
```

pop与shift函数分别从target_array中删除一个元素。如果target_array没有设定，那么元素可以从 @_ 中删除，也可以从 @ARGV中删除。pop和shift函数返回被删除的元素，如果数组是

空的，则返回`undef`。数组的大小将相应地缩小。



在子例程中，如果没有设定其他数组，那么`pop`、`shift`、`unshift`和`push`函数将修改`@_`。在子例程外面，你的程序主体中，如果没有设定其他数组，那么这些函数将修改数组`@ARGV`。

`push`和`unshift`函数将`new_list`的元素添加给`target_array`，数组的大小将增大，以适应放置新元素的需要。被放入`targer_array`的项目，或者从`target_array`取出的项目既可以是一个列表，也可以是一个数组，如下例所示：

```
#band=qw(trombone);
push @band, qw( ukulele clarinet );
# @band now contains trombone, ukulele, clarinet

$brass=shift @band;    # $brass now has "trombone",
$wind=pop @band;      # $wind now has "clarinet"

# @band now contains only "ukulele"
unshift @band, "harmonica";
# @band now contains harmonica, ukulele
```



当你将元素添加给数组时，将元素推送（或移动）到数组中要比用手工将元素添加到数组的结尾处更加有效。比如，`push(@list, @newitems)`比`@list=(@list, @newitems)`更加有效。Perl的`push`、`shift`、`unshift`和`pup`函数都进行了优化，以适应这些操作的需要。

“堆栈”中的数组元素仍然属于标准的数组元素，可以用索引进行编址。堆栈的“底部”是元素`0`，堆栈的“顶部”是数组中的最后一个元素。

拼接数组

迄今为止，我们介绍了数组可以按元素进行寻址、分行、移动、弹出、取消移动和推送。数组操作的最后一个工具是`splice`。`splice`函数的句法如下：

```
splice array, offset
splice array, offset, length
splice array, offset, length, list
```

`splice`函数用于删除数组中从`offset`位置开始的元素，同时返回被删除的数组元素。如果`offset`的值是负值，则从数组的结尾处开始计数。如果设定了`length`，那么只删除`length`指定的元素。如果设定了`list`，则删除`length`指定的元素，并用`list`的元素取代之。通过这个处理过程，数组就会根据需要扩大或缩小，如下面所示的那样：

```
@veg=qw( carrots corn );
splice(@veg, 0, 1);                      # @veg is corn
splice(@veg, 0, 0, qw(peas));            # @veg is peas, corn
splice(@veg, -1, 1, qw(barley, turnip)); # @veg is peas, barley, turnip
splice(@veg, 1, 1);                      # @veg is peas, turnip
```

9.6 课时小结

在本学时中，我们介绍了搜索其他字符串中的字符串时不需要使用正则表达式，你可以

使用index和rindex进行简单的搜索，也可以使用tr//运算符进行简单的替换。substr函数既可以用来从字符串中检索数据，也可以对它们进行编辑。你可以使用printf和sprintf语句，用Perl创建格式很好的输出。另外，我们介绍了用作项目堆栈而不是平面列表的数组，还学习了如何对这些堆栈进行操作。

9.7 课外作业

9.7.1 专家答疑

问题：substr、index和rindex等函数真的有必要使用吗？当正则表达式可以用来执行它们的大多数操作时，为什么还要这几种函数？

解答：首先，用于进行简单的字符串搜索的正则表达式的运行速度比index和rindex慢。第二，为字符位置固定的正则表达式编写替换表达式会产生很大的混乱，而有时substr则是比较出色的解决方案。第三，Perl是一种丰富多彩的语言，你可以使用你喜欢的解决方案，你可以有多种多样的选择。

问题：如果我设定的索引位于标量的结尾之外，使用substr（或index或rindex）将会出现什么情况？

解答：计算机的好处之一是：它具有很强的一致性，而且它有很强的耐心。对于“如果...将会出现什么情况”之类的问题，有时你只要试一试它的最容易实现的方法即可！什么是可能出现的最坏情况呢？

在这种情况下，如果你激活了警告特性，那么访问并不存在的标量的某个部分，就会产生一个“use of undefined value”（使用了未定义的值）的错误。例如，如果你使用\$ a=“Foo”；substr（\$ a,5）；那么substr函数将返回undef。

9.7.2 思考题

1) 假如有下面这个代码，那么在@A中将留下什么？

```
@A=qw(oats peas beans);
shift @A;
push @A, "barley";
pop;

a. oats peas beans
b. deans barley
c. peas beans barley
```

2) printf（“%18.3f”，\$ a）这个代码能够进行什么操作？

- a. 它输出一个浮点数，长度为18个字符，小数点左边是15个字符，小数点右边是3个字符。
- b. 它输出一个浮点数，小数点左边是18个字符，小数点右边是3个字符。
- c. 它输出一个浮点数，长度为18个字符，小数点左边是14个字符，右边3个字符。

3) 如果对一个字符串运行tr/a-z/A-Z，tr/A-Z/a-z能否使字符串恢复其原始形式？

- a. 是，当然能够。
- b. 也许做不到。

9.7.3 解答

- 1) 答案是c。shift删除了cats，而push则将barley添加到结尾处。最后的pop是个假像，它并没有设定任何数组，因此它从array @_中弹出某些数据，但是它没有给@A带来任何变化。
- 2) 答案是c。如果你猜测的答案是a，那么你没有将小数点计算在内，它在总数（ $18 = 14 + 1 + 3$ ）中占有一个位置。
- 3) 答案是b。tr/a-z/A-Z/转换的“rosebud”变成了“ROSEBUD”。如果试图用tr/A-Z/a-z/将它变回原样，那么产生的是“rosebud”，而不是原始字符串。

9.7.4 实习

- 使用标量而不是数组，重新编写第4学时中的Hangman游戏。你可以使用substr，对标量中的各个字符进行操作。
- 修改程序清单9-2，从文件中读取数据，而不是从数组中获取数据。打开文件，将数据读入一个数组，然后按正常情况继续操作。当然你必须在磁盘上创建该文件。

China-pub.com

下载

China-pub.com

下载

第10学时 文件与目录

操作系统中的文件为数据提供了一个非常方便的存储方式。操作系统为数据提供了一个名字（即文件名）和一个组织结构，这样你就可以在以后找到你要的数据。这个组织结构称为文件系统。然后你的文件系统再将文件分成各个组，称为目录，有时也称为文件夹。这些目录能够存放文件或其他目录。

在目录中嵌套目录的方法给计算机中的文件系统提供了一个树状结构。每个文件都是一个目录的组成部分，每个目录又是父目录的组成部分。除为你的文件提供一个组织结构外，操作系统还存放了关于文件的各种数据，比如上次读取文件是在什么时候，上次修改文件是在什么时候，谁创建了文件，当前文件有多大等等。所有的现代计算机操作系统几乎都采用这种组织结构。

在Macintosh系统中，仍然采用这种结构，不过它的高层目录称为卷，子目录称为文件夹。

Perl允许你访问这个组织结构，修改它的组织方法，并可查看关于文件的各种信息。Perl用于这些操作的函数全部源自 unix操作系统，但是在Perl运行的任何操作系统下，这些函数都能够很好地运行。Perl的文件系统的操作函数是可以移植的，也就是说，如果你使用 Perl的函数对你的文件进行操作并查询你的文件，那么在Perl支持的任何操作系统下，运行你的代码都是没有问题的，只要目录结构相类似。

在本学时中，你将要学习：

- 如何获得目录列表。
- 如何创建和删除文件。
- 如何创建和删除目录。
- 如何获取关于文件的信息。

10.1 获得目录列表

从系统中获取目录信息的第一步是创建一个目录句柄。目录句柄与文件句柄相类似，不同之处是：不是通过读取文件句柄来获得文件的内容，而是使用目录句柄来读取目录的内容。若要打开目录句柄，可以使用 opendir函数：

```
opendir dirhandle, directory
```

在这个语句中，dirhandle是要打开的目录句柄，directory是要读取的目录的名字。要是目录句柄不能打开，你就无权读取该目录的内容，或者该目录根本不存在。opendir函数将返回假。目录句柄的结构应该与文件句柄相类似，它使用第2学时介绍的变量名的创建规则，目录句柄应该全部使用大写字母，以避免与Perl的关键字发生冲突。下面是目录句柄的一个例子：

```
opendir(TEMPDIR, '/tmp') || die "Cannot open /tmp: $!"
```

本学时中介绍的所有例子都使用 UNIX样式中的正斜杠，因为与反斜杠相比，它不易产生混乱，并且它可以同时用于UNIX和Windows操作系统。

目录句柄打开后，可以使用 readdir函数来读取它的内容：

```
readdir dirhandle;
```

在标量上下文中，readdir函数返回目录中的下一项，如果目录中没有剩下任何项目，则返回undef。在列表上下文中，readdir返回所有的（剩余的）目录项。readdir返回的名字包括文件、目录的名字，而对于UNIX来说，则返回特殊文件的名字。它们返回时没有特定的次序。readdir返回目录项.和..。readdir返回的目录项不包含作为目录名的组成部分的路径名。

当完成目录句柄的操作后，应该使用 closedir函数将它关闭：

```
closedir dirhandle;
```

下面这个例子说明如何读取一个目录：

```
opendir(TEMP, '/tmp') || die "Cannot open /tmp: $!";
@FILES=readdir TEMP;
closedir(TEMP);
```

在上面这个代码段中，整个目录被读入 @FILES中。不过，在大多数时候，你对.和..文件是不感兴趣的。若要读取文件句柄并清除这些文件，可以输入下面的代码：

```
@FILES=grep(/!^\.\.?$/ , readdir TEMP);
```

正则表达式 (/^\.\.?\$/) 用于匹配也位于行尾的一个前导原义圆点（或两个圆点），而grep则用于清除它们。若要获得带有特定扩展名的全部文件，可以使用下面的代码：

```
@FILES=grep(/\.\txt$/i, readdir TEMP);
```

readdir返回的文件名并不包含 opendir使用的路径名。因此，下面的例子可能无法运行：

```
opendir(TD, "/tmp") || die "Cannot open /tmp: $!";
while($file=readdir TD) {
    # The following is WRONG
    open(FILEH, $file) || die "Cannot open $file: $!\n";
    :
}
closedir(TD);
```

除非你在运行代码时恰好在 /tmp 目录中工作，否则 open(FILEH, \$file)语句的运行将会失败。例如，如果 /tmp 中存在文件 myfile.txt，那么 readdir便返回 myfile.txt。当你打开 myfile.txt 时，实际上必须使用全路径名来打开 /tmp/myfile.txt。正确的代码如下所示：

```
opendir(TD, "/tmp") || die "Cannot open /tmp: $!";
while($file=readdir TD) {
    # Right!
    open(FILEH, "/tmp/$file") || die "Cannot open $file: $!\n";
    :
}
closedir(TD);
```

Globbing

读取目录中的文件名时使用的另一种方法称为 globbing。如果你熟悉DOS中的命令提示符，那么一定知道命令 dir*.txt 可用于输出以.txt结尾的所有文件的目录列表。在 UNIX中，globbing 是由 shell 来完成的，但是 ls*.txt 几乎能产生相同的结果，即列出以.txt结尾的所有文件。

Perl有一个操作符，能够进行这项操作，它称为 glob。Glob的句法是：

```
glob pattern
```

这里的pattern是你要匹配的文件名模式。pattern可以包含目录名和文件名的各个部分。此外，pattern可以包含表 10-1 列出的任何一个特殊字符。在列表上下文中，glob返回与模式匹配的所有文件（和目录）。在标量上下文中，每查询一次 glob，便返回一个文件。

下载



glob的模式与正则表达式的模式不同。

表10-1 globbing的模式

字 符	匹配的模式	举 例
?	单个字符	f ? d用于匹配fud、fid和fdd等。
*	任何数目的字符	f*d用于匹配fd、fdd、food和filled等
[chars]	用于匹配任何一个chars; MacPerl不支持这个特性	f[ou]d用于匹配fod和fud，但不能匹配fad
{a , b , ...}	既可以匹配字符串a， 也可以匹配字符串b， MacPerl不支持这个特性	f*{txt , doc}用于匹配以f开头并且以.txt或.doc结尾的文件



对于UNIX爱好者来说，Perl的glob操作符使用C语言的shell样式的文件globbing，而不是Bourne（或Korn）shell的文件globbing。它适用于安装了Perl的任何UNIX系统，而不管你个人使用的是何种 shell。Bourne shell globbing和Korn shell globbing不同于C语言的shell globbing。它们在某些方面很相似，比如 *与?的运行特性相同，但是在其他方面差别很大，请注意。

下面请看几个globbing的例子：

```
# All of the .h files in /usr/include
my @hfiles=glob('/usr/include/*.h');
# Text or document files that contain 1999
my @curfiles=glob('*1999*.{txt,doc}')
# Printing a numbered list of filenames
$count=1;
while($name=glob('*')) {
    print "$count. $name\n";
    $count++;
}
```

下面是使用glob与opendir/readdir/closedir之间的某些差别：

- glob只能返回有限数量的文件。对于较大的目录，glob可能会报告“太多的文件”，但是却不能返回任何文件。这是因为glob当前是使用外部程序即shell来实现的，它只能返回有限数量的文件。opendir/readdir/closedir函数则不存在这个问题。
- glob返回模式中使用的路径名，而opendir/readdir/closedir函数则不能。例如，glob(' /u s r /i n c l u d e) 返回“/u s r /i n c l 作为任何匹配项的组成部分，而readdir则不能返回它。
- glob的运行速度通常比opendir/readdir/closedir慢。同样，它之所以运行速度比较慢，是因为Perl必须启动一个外部程序来为它进行globbing，而该程序将在对文件名排序后再返回这些文件名。

那么你究竟应该使用哪一个函数呢？这完全取决于你。不过，使用opendir/readdir/closedir函数往往是个复杂得多的解决方案，在本书中的大多数程序例子中，我们都使用这个函数。

为了完整起见，Perl提供了另一种方法，用于编写模式 glob。只需将模式放入尖括号运算符 (<>) 中，就可以使尖括号运算符像 glob 函数那样来运行：

```
@cfiles=<*.c>; # All files ending in .c
```

用于globbing的尖括号运算符的句法比较老，并且可能引起混乱。在本书中，为了清楚起见，将继续使用 glob 函数。

10.2 练习：UNIX的grep

当你进一步阅读本书的内容时，一些练习将为你展示更多的非常有用的工具。下面这个练习展示了一个 UNIX 的 grep 实用程序的简化版本。UNIX 的 grep（不要与 Perl 的 grep 相混淆）用于搜索文件中的模式。这个练习展示了一个实用程序，它提示你输入一个目录名和一个模式。目录中的每个文件均被搜索，以便寻找该模式，与该模式相匹配的文件行将被输出。

使用文本编辑器，键入程序清单 10-1 的程序，并将它保存为 mygrep。务必按第 1 学时中介绍的方法使该程序成为可执行程序。另外，一定不要将该文件改名为 UNIX 系统上的 grep，它可能被混淆为实际的 grep 实用程序。

完成上述操作后，键入下面的命令行，设法运行该程序：

```
perl mygrep
```

程序清单 10-1 mygrep 的完整清单

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:
5:  print "Directory to search: ";
6:  my $dir=<STDIN>; chomp $dir;
7:  print "Pattern to look for: ";
8:  my $pat=<STDIN>; chomp $pat;
9:
10: my($file);
11:
12: opendir(DH, $dir) || die "Cannot open $dir: $!";
13: while ($file=readdir DH) {
14:     next if (-d "$dir/$file");
15:     if (! open(F, "$dir/$file")) {
16:         warn "Cannot search $file: $!";
17:         next;
18:     }
19:     while(<F>) {
20:         if (/^$pat/) {
21:             print "$file: $_";
22:         }
23:     }
24:     close(F);
25: }
26: closedir(DH);

```

第1行：这一行包含到达解释程序的路径（可以修改它，使之适合系统的需要）和开关 -w。请始终使警告特性处于激活状态。

第3行：use strict 命令意味着所有变量都必须用 my 来声明，并且裸单词必须加上引号。

第5~8行：\$dir（要搜索的目录）和 \$pat（要搜索的模式）是从 STDIN 中检索而来的。每

行结尾处的换行符均被删除。

第10行：\$file被声明为符合use strict的专用变量。\$file用在本程序的后面。

第12行：目录\$dir被打开，如果这项操作没有成功，则输出一条出错消息。

第13行：从目录中检索各个条目，每次检索1条，然后存入\$file。

第14行：确实是目录本身（-d）的任何目录条目均被拒绝。请注意，被核实的路径名是\$dir/\$file。核实该路径是必要的，因为当前目录中不一定存在\$file，它存在于\$dir中。因此该文件的全路径名是\$dir/\$file。

第15~18行：文件被打开，再次使用全路径名\$dir/\$file，如果文件没有打开，则被拒绝。

第19~23行：文件被搜索，逐行进行搜索，找出包含\$pat的这一行。匹配的行将被输出。

程序清单10-2显示了mygrep程序的输出举例。

程序清单10-2 mygrep的输出

```
1: Directory to search: /home/clintp
2: Pattern to look for: printer
3: mailbox: lot of luck re-inking Epson printer ribbons with
4: config.pl: # the following allows the user to pick a printer for
```

10.3 目录

到现在为止，本学时中一直在介绍目录结构的问题。为了打开文件，有时需要它的全路径名，readdir函数能够读取目录。但是，如果要浏览目录，添加和删除目录，或者清除目录，则需要更多的Perl功能。

10.3.1 浏览目录

当你运行软件时，操作系统将对你所在的目录保持跟踪。当你登录到一台UNIX计算机中并且运行一个软件包时，通常要进入你的主目录。如果你键入操作系统的命令pwd，shell便向你显示你是在什么目录中。如果你使用DOS或Windows操作系统，并且打开一个命令提示符，该提示符就能反映出你当时是在什么目录中，比如C:\WINDOWS。另外，你也可以在DOS提示符后面键入操作系统命令cd，这样，DOS就会告诉你是在什么目录中。你当前使用的目录称为当前目录，即你的当前工作目录。



如果你使用程序编辑器或者集成式编辑器/调试器，并且直接从那里运行你的Perl程序，那么“当前目录”可能不是你想像的那个目录。它可能是Perl程序所在的目录，编辑器所在的目录或者是任意其他的目录，这取决于你使用何种编辑器。如果要确定当前目录究竟是什么，请在你的Perl程序中使用cwd函数。

如果没有全路径名，也可以打开文件，比如：open(FH, "file")||die可以在你的当前目录中打开。若要改变当前目录，可以使用下面这个chdir函数：

```
chdir newdir;
```

chdir函数将当前工作目录改为newdir。如果newdir目录不存在，或者你不拥有对newdir的访问权，那么chdir返回假。chdir对目录的改变是暂时的，一旦Perl程序运行结束，就返回运

行Perl程序之前所在的目录。

如果运行的chdir函数不包含一个目录作为其参数，那么 chdir就会将你的目录改为你的主目录。在 UNIX系统上，主目录通常是你登录时进入的这个目录。在 Windows 95、Windows NT或DOS计算机上，chdir会使你进入HOME环境变量中指明的这个目录。如果 HOME没有设置，chdir将根本不改变当前目录。

Perl并不配有任何内置函数来确定当前目录是个什么目录，因为某些操作系统编写时所采用的方法，使得函数很难做到这一点。若要确定当前目录，必须同时使用两个语句。在程序的某个位置，最好是在靠近程序开始的地方，必须使用语句 use Cwd，然后，当你想要检查当前目录时，使用 cwd函数：

```
use Cwd;
print "Your current directory is: ", cwd, "\n";
chdir '/tmp' or warn "Directory /tmp not accessible: $!";
print "You are now in: ", cwd, "\n";
```

只能执行use Cwd语句一次，此后，可以根据需要多次使用 cwd函数。



语句use Cwd实际上让Perl加载一个称为 Cwd的模块，使Perl语言增加一些新的函数，如 Cwd。如果上面介绍的这个代码段返回一条出错消息，说 Can't locate cwd .pm in @INC (在@INC中找不到Cwd.pm)，或者你不完全理解各个模块，那么现在不必为此而担心，第 14学时将要详细介绍模块方面的知识。

10.3.2 创建和删除目录

若要创建一个新目录，可以使用 Perl的mkdir函数，mkdir函数的句法如下：

```
mkdir newdir, permissions;
```

如果目录newdir能够创建，那么 mkdir函数返回真。否则，它返回假，并且将\$！设置为mkdir运行失败的原因。只有在Perl的UNIX实现代码中，permissions才真的十分重要，不过在所有版本中都必须设置 permissions。对于下面这个例子，使用的值是 0755。这个值将在本学时后面部分中的“ UNIX系统 ”这一节中介绍。对于 DOS和Windows用户来说，只使用 0755这个值就足够了，可以省略冗长的说明。

```
print "Directory to create?";
my $newdir=<STDIN>;
chomp $newdir;
mkdir( $newdir, 0755 ) || die "Failed to create $newdir: $!";
```

若要删除目录，可以使用 rmdir函数。rmdir函数的句法如下：

```
rmdir pathname;
```

如果目录 pathname可以删除，rmdir函数返回真。如果 Pathname无法删除，rmdir返回假，并将\$！设置为rmdir运行失败的原因，如下所示：

```
print "Directory to be removed?";
my $baddir=<STDIN>;
chomp $baddir;
rmdir($baddir) || die "Failed to remove $baddir: $!";
```

rmdir函数只删除完全是空的目录。这意味着在目录被删除之前，首先必须删除目录中的

所有文件和子目录。

10.3.3 删除文件

若要从目录中删除文件，可以使用 unlink函数：

```
unlink list_of_files;
```

unlink函数能删除 list_of_files中的所有文件，并返回已经删除的文件数量。如果 list_of_files被省略，\$_中指定的文件将被删除。请看下例：

```
unlink <*.bat>;
$erased=unlink 'old.exe', 'a.out', 'personal.txt';
unlink @badfiles;
unlink;      # Removes the filename in $_
```

若要检查文件列表是否已被删除，必须对想要删除的文件数量与已删除的文件数量进行比较，如下面这个例子所示：

```
my @files=<*.txt>;
my $erased=unlink @files;

# Compare actual erased number, to original number
if ($erased != @files) {
    print "Files failed to erase: ",
          join(',', <*.txt>), "\n";
}
```

在上面这个代码段中，被 unlink删除的文件数量将存放在 \$erased中。unlink运行后，\$erased的值将与 @files中的元素的数量进行比较，它们应该相同。如果不同，便输出一条出错消息，显示“剩余的”文件。



用unlink函数删除的文件将被绝对地清除掉，它们将无法恢复，并且不是被转入“回收站”，因此使用unlink函数时应该格外小心。

10.3.4 给文件改名

在Perl中给文件或目录改名是很简单的，可以使用 rename函数，如下所示：

```
rename oldname, newname;
```

rename函数取出名字为 oldname的文件，将它的名字改为 newname。如果改名成功，该函数返回真。如果 oldname和newname是目录，那么这些目录将被改名。如果改名不成功，rename返回假，并将 \$!设置为不成功的原因，如下所示：

```
if (! rename "myfile.txt", "archive.txt") {
    warn "Could not rename myfile.txt: $!";
}
```

如果你设定了路径名，而不只是设定文件名，那么 rename函还会将文件从一个目录移到另一个目录，如下例所示：

如果文件newname已经存在，该文件将被撤销。

```
rename "myfile.txt", "/tmp/myfile.txt";  # Effectively it moves the file.
```



如果文件是在不同的文件系统上，那么 rename函将不会把文件从一个目录移到另一个目录中。

10.4 UNIX系统

下面介绍Perl用户在UNIX操作系统下工作的情况。如果你不是在UNIX系统上使用Perl，那么可以跳过本节，你不会遗漏任何重要的内容。如果你对UNIX非常有兴趣，可以阅读这一节的内容。

作为UNIX用户来说，应该知道Perl与UNIX系统之间有着很深的渊源关系，有些Perl函数直接来自UNIX命令和操作系统函数。这些函数中，大部分是你不使用的。有些函数，如unlink，虽然源于UNIX，但其含义却与UNIX毫无关系。每个操作系统都可以用来删除文件，Perl能够确保unlink会对操作系统执行正确的操作。Perl作出了很大的努力，以确保有关的特性（如文件I/O）能够在操作系统之间移植，并且它在可能的情况下将所有的兼容性问题隐藏起来，使你不必为此而担心。

Perl语言中嵌入了许多UNIX函数和命令，而Perl语言已经移植到许多非UNIX操作系统中，这满足了UNIX开发人员和管理员的要求，使他们能够将UNIX工具包中的一些工具带到任何地方去使用。



正如下一节的标题所示，这个描述不能被视为UNIX文件系统访问许可权以及如何操作文件的完整说明。若要了解它的完整说明，请参见你的操作系统文档，或者参阅关于UNIX的其他著作，如《UNIX 24学时教程》。

文件访问许可权的简要介绍

在第1学时中，我们讲到，为了使Perl程序能够像一个标准命令那样来运行，我们提供了一个命令chmod 755 scriptname，但是没有具体介绍它的含义。755是赋予文件scriptname的访问许可权的一种描述。UNIX中的chmod命令用于设置文件的访问许可权。

这行数字分别代表赋予文件所有者、文件所属小组以及其他非文件所有者和非文件所属小组的访问许可权。在上面这个例子中，文件所有者拥有的访问许可权是7，文件所属小组和其他人的访问许可权是5。表10-2列出了每种访问许可权的值。



表10-2 文件访问许可权

访问许可权值	权限
7	所有者 / 组 / 其他人可以读、写和执行该文件
6	所有者 / 组 / 其他人可以读、写该文件
5	所有者 / 组 / 其他人可以读和执行该文件
4	所有者 / 组 / 其他人可以读该文件
3	所有者 / 组 / 其他人可以写和执行该文件
2	所有者 / 组 / 其他人可以写该文件
1	所有者 / 组 / 其他人可以执行该文件

若要在Perl中设置文件的访问许可权，可以使用UNIX的内置函数chmod：

```
chmod mode, list_of_files;
```

chmod函数能够改变list_of_files的所有文件的访问许可权，并且返回已经改变访问许可权的文件数量。mode的前面必须有一个数字0（如果它是一个八进制直接量数字的话），然后是你

下载

想指明其访问许可权的数字。下面是 chmod命令的一些例子：

```
chmod 0755, 'file.pl';      # Grants RWX to owner, RX to group and non-owners
chmod 0644, 'mydata.txt';   # Grants RW to owner, and R to group and non-owners
chmod 0777, 'script.pl';    # Grants RWX to everyone (usually, not very smart)
chmod 0000, 'cia.dat';     # Nobody can do anything with this file
```

在本学时前面部分的内容中，我们介绍了 mkdir函数。mkdir的第一个参数是文件访问许可权，它与 chmod使用的许可权是相同的：

```
mkdir "/usr/tmp", 0777;  # Publically readable/writable
mkdir "myfiles", 0700;   # A very private directory
```



UNIX文件的访问许可权常常称为它的“方式”。因此 chmod是“change mode（改变方式）”的缩写。

10.5 你应该了解的关于文件的所有信息

如果你想要详细而全面地了解关于一个文件的信息，可以使用 Perl的stat函数。stat函数源于UNIX系统，它的返回值在UNIX系统中与非UNIX系统中略有不同。Stat的句法如下所示：

```
stat filehandle;
stat filename;
```

stat函数即可以用来检索已经打开的文件句柄的信息，也可以检索关于某个特定文件的信息。在任何操作系统下，stat均可返回一个包含13个元素的列表，来描述文件的属性。列表中的实际值随着运行的操作系统的不同而有所差异，因为有些操作系统包含的特性是其他操作系统所没有的。表10-3显示了stat返回值中的每个元素的含义。

表10-3 stat函数的返回值

编 号	名 字	UNIX系统	Windows系统
0	dev	设备号	驱动器号 (C : 通常是2 , D : 通常是3 , 等等)
1	ino	索引节号	总是0
2	mode	文件的方式	无
3	nlink	链接号	通常为0 ; Windows NT ; 文件系统允许链接
4	uid	文件所有者的用户ID (UID)	总是0
5	gid	文件所有者的组ID (GID)	总是0
6	rdev	特殊文件信息	驱动器号 (重复)
7	size	文件大小 (以字节计)	文件大小 (以字节计)
8	atime	上次访问的时间	上次访问的时间
9	mtime	上次修改的时间	上次修改的时间
10	ctime	Inode修改时间	文件的创建时间
11	blksz	磁盘块的大小	总是0
12	blocks	文件中的块的数量	总是0

表10-3中的许多值你可能永远不会使用，但是为了完整起见，我们在表中将它们列出来。对于含义比较含糊的值，尤其是 UNIX中的返回值，你可以查看操作系统的参考手册，以了解它的含义。

下面是将stat用于文件的一个例子：

```
@stuff=stat "myfile";
```

通常情况下，为了清楚起见，stat的返回值被拷贝到标量的一个赋值列表：

```
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
$atime, $mtime, $ctime, $blksize, $blocks)=stat("myfile");
```

若要按“文件访问许可权的简要介绍”这一节中所说的3字符形式输出文件的访问许可权，可以使用下面这个代码，其中的@stuff包含了访问许可权：

```
printf "%04o\n", $mode&0777;
```

上面这个代码所包含的元素你可能不了解，这没有什么关系。其中有些元素尚未向你介绍。在\$mode中由stat检索的元素包含了许多“额外的”信息。&0777只是提取你感兴趣的这部分信息。最后，%0是一个printf格式，用于输出采用0~7格式的八进制数字，UNIX希望用这种格式对文件访问许可权进行格式化。



八进制是以8为基数的数字表示法。由于历史原因，它用于UNIX系统，不过它也用于Perl。如果你仍然对它不太理解，请不必担心。如果需要显示文件的访问许可权，只要使用前面介绍的printf函数即可。它并不是经常出现。

表10-3中列出的3个时间戳，即访问时间、修改时间和更改（即创建）时间均以特定格式来存储。时间戳以格林威治时间1970年1月1日零点起的秒数来存储。若要以便于使用的格式来输出时间，可以使用localtime函数，如下所示：

```
print scalar localtime($mtime);
```

该函数用于输出文件的修改时间，格式为Sat Jul 3 23:35:11 EDT 1999。访问时间是指上次读取文件（或打开文件以便读取）的时间，修改时间是指上次将数据写入文件的时间。在UNIX系统下，“更改”时间是指更改关于文件的信息（文件的所有者、链接的数量、访问许可权等）的时间，它并不是文件的创建时间，不过由于巧合，它常常就是文件的创建时间。在Microsoft Windows下，ctime域实际上用于存放文件的创建时间。

有时你可能想从stat返回的列表中仅仅检索1个值。若是这样，可以用括号将整个stat函数括起来，并使用下标将你想要的值标出来：

```
print "The file has", (stat("file"))[7], " bytes of data";
```

10.6 练习：对整个文件改名

这个练习为你的工具包提供了另一个小型工具。假定有一个目录名，一个要查找的模式和要改变成的模式，使用这个实用程序，可以对目录名中的所有文件进行改名。例如，如果一个目录中包含文件名Chapter_01.rtf、Chapter_02.rtf、Chapter_04.rtf等，你就可以将所有文件改名为Hour_01.rtf、Hour_02.rtf、Hour_04.rtf等。当使用基于图形用户界面的文件浏览器时，想要用命令提示符来执行这种操作通常不容易，而且显得很笨。

使用文本编辑器，键入程序清单10-3中的程序，并将它保存为Renamer。务必按照第1学时介绍的方法使该程序成为可执行程序。

完成上述操作后，键入下面的命令行，设法运行该程序：

```
perl Renamer
```

下载

程序清单 10-3 Renamer 程序的完整清单

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:
5:  my($dir, $oldpat, $newpat);
6:  print "Directory: ";
7:  chomp($dir=<STDIN>);
8:  print "Old pattern: ";
9:  chomp($oldpat=<STDIN>);
10: print "New pattern: ";
11: chomp($newpat=<STDIN>);
12:
13: opendir(DH, $dir) || die "Cannot open $dir: $!";
14: my @files=readdir DH;
15: close(DH);
16: my $oldname;
17: foreach(@files) {
18:     $oldname=$_;
19:     s/$oldpat/$newpat/;
20:     next if (-e "$dir/$_");
21:     if (! rename "$dir/$oldname", "$dir/$_") {
22:         warn "Could not rename $oldname to $_: $!";
23:     } else {
24:         print "File $oldname renamed to $_\n";
25:     }
26: }
```

第13~15行：\$dir指明的目录中的条目被读入 @files。

第17~19行：来自 @files 的每个文件均被赋给 \$ _，并且该名字被保存在 \$ oldname 中。然后，\$ _ 中的原始文件名被改为 19 行上的新名字。

第20行：在对文件改名之前，这一行要确认目标文件名并不存在。否则，该程序可能将文件改名为一个现有文件名，撤消其原始数据。

第21~25行。该文件被改名，如果改名失败，则输出一条警告消息。请注意，原始目录名必须附加到文件名上，例如，\$dir/\$oldname，因为 @files 不包含全路径名，所以必须使用全路径名来进行改名。

程序清单 10-4 显示了该程序的示例输出。

程序清单 10-4 Rename 程序的输出示例

```

1: Directory: /tmp
2: Old Pattern: Chapter
3: New Pattern: Hour
4: File Chapter_02.rtf renamed to Hour_02.rtf
5: File Chapter_10.rtf renamed to Hour_10.rtf
```

10.7 课时小结

本学时我们介绍了如何使用 Perl 中的 mkdir、rm 和 rename 函数来创建、删除目录条目并对其改名。另外，还介绍了如何使用 stat 来查询文件系统，以便了解关于文件的信息，不光是了解文件的内容。在本学时中，两个练习提供了一些简单而实用的工具，使你的程序具备更高的效能。

10.8 课外作业

10.8.1 专家答疑

问题：我在运行下面这个程序时遇到了问题。虽然目录中有文件，但是无法读取，原因何在？

```
opendir(DIRHANDLE, "/mydir") || die;
@files=<DIRHANDLE>;
closedir(DIRHANDLE);
```

解答：问题出在第2行上。DIRHANDLE是个目录句柄，不是文件句柄。你无法用尖括号(<>)操作符来读取目录句柄。读取目录的正确方法是 @files=readdir DIRHANDLE。

问题：为什么glob(".*")不能匹配目录中的所有文件？

解答：因为“.*”只能匹配文件名中有圆点的文件名。若要匹配目录中的所有文件，请使用glob(*.*). glob函数的模式可以在许多不同操作系统之间进行移植，因此它的运行特性与DOS中的.*不同。

问题：我修改了mygrep这个练习，以便使用opendir和更多的循环来搜索子目录，但是它似乎存在某些错误。为什么？

解答：总之，你不要这样做。向下搜索目录树是个老问题，它并不很容易，以前曾经多次解决过这个问题，而你自己没有必要这样去做。（从事全部这项工作称为“重新发明车轮”。）如果你只是因为好玩而这样做，这很好，但是不要在这上面耗费太多的时间。请等到第15学时，你将会了解到如何使用File::Find这个方法。它用起来更加简单，但是更加重要，它能够进行程序调试。

问题：如果我将*.bat改为*.tmp，程序清单10-3中的程序就会出错，为什么？

解答：该程序并不希望你键入*.bat作为要搜索的模式。在正则表达式中使用*.bat是无效的，*必须放在另外某个字符的后面。如果你输入了*.bat，该程序完全可以接受这个输入，不过它不会像你期望的那样运行，因为文件名中从来没有原义字符*。

为了纠正这个错误，你可以为该程序提供它期望的输入（简单字符串），也可以将程序清单的第19行改为s/Q\$oldpat/\$newpat/，这样，正则表达式模式中的“特殊字符”将不起作用。

10.8.2 思考题

1) 若要输出文件foofile的上次修改时间，应该使用：

- print glob("foofile");
- print(stat("foofile"))[9];
- print scalar localtime(stat("foofile"))[9];

2) unlink函数返回的是：

- 实际删除的文件数量。
- 真或假，根据函数运行是否成功的情况而定。
- 试图删除的文件数量。

10.8.3 解答

1) 答案是b或c。如果选择b，输出的时间为1970年以来的秒数，没有什么用处。如果

选择c，则输出格式很好的时间。

2) 答案是a。不过，选择c，在某种情况下也是可以的。如果没有文件可以删除，unlink返回0，表示假。

10.8.4 实习

- 设法编写一个程序，列出目录中的所有文件、它的子目录中的所有文件等。这只是一个编程练习。

China-pub.com

下载

China-pub.com

下载

第11学时 系统之间的互操作性

到现在为止，我们介绍的所有Perl特性基本上都属于独立的特性。如果你想完成某项操作，那么你必须自己亲自执行这项操作，比如给数据排序，创建目录列表，插入配置信息等。问题是它的工作量很大，你必须重复进行可以在其他地方完成的工作。

关于Perl，现在有一种非常流行的说法，那就是它是一种非常出色的“胶水”语言。它的意思是说，Perl能够使用操作系统作为组件来安装的其他程序，然后将这些程序组合起来，形成一个更大的程序。它能够启动操作系统的实用程序，用它们来搜集信息，与你进行通信，然后将它们关闭。

Perl能够将这些较小的实用程序“胶合”在一起，形成一个大得多而且更加有用的实用程序。这种能力的好处是使你能够迅速编写在其他情况下需要花费很长时间来编写的代码，并且能够对代码进行调试。你应该使用对你有用的所有手段，迅速而准确地编写代码。将系统的实用程序胶合在一起，可使之具备很大的优点。

在本学时中，你将要学习：

- system()函数。
- 捕获输出。
- 代码的移植性。



本学时中的大部分代码例子都有两个版本，一个用于Windows和DOS系统，另一个用于UNIX系统。如果只有一个代码例子，那么你会在课文中找到关于如何修改该代码，以适合另一种系统的需要，这种修改通常是少量的更改。

11.1 system()函数

若要运行非Perl的命令，最简单的方法是使用system()函数。system()函数能够暂停Perl程序的运行，然后运行外部命令，接着再运行你的Perl程序。system函数的句法如下：

```
system command;
```

该语句中的command是你要运行的命令，如果一切正常，系统的返回值是0，如果出现问题，则返回非零值。请注意，这个结果与True和False这两个Perl标准返回值相反。

下面是在UNIX系统上运行system函数的一个例子：

```
system("ls -1F");          # Print a file listing
# print system's documentation
if ( system("perldoc -f system") ) {
    print "Your documentation isn't installed correctly!\n";
}
```

下面这个例子显示在DOS / Windows下运行system的情况：

```
system("dir /w");          # Print a file listing
# print system's documentation
```

```
if ( system("perldoc -f system") ) {
    print "Your documentation isn't installed correctly!\n";
}
```

总的来说，system函数在上述两种系统结构下的运行情况是相同的。应该记住的是，两种操作系统下运行的命令基本上是不同的。若要在DOS下获得一个文件列表，可以使用dir命令，而在UNIX下获得文件列表，要使用ls命令。在非常少见的情况下，比如在perldoc中，UNIX和DOS系统下的命令是相同的。

当system函数运行外部命令时，该命令的输出在屏幕上显示的情况与Perl程序输出的情况是相同的。如果该外部命令需要输入数据，那么来自终端的输入与你的Perl程序从终端读入的输入是相同的。system函数运行的命令继承了STDIN和STDOUT文件描述符，因此，外部命令执行输入/输出的位置与你的Perl程序执行输入/输出的位置是完全相同的。通过system函数来调用完全交互式的程序是可能的。

请看下面这个在UNIX下运行的代码例子：

```
$file="myfile.txt";
system("vi $file");
```

现在再看在Windows / DOS下运行的代码例子：

```
$file="myfile.txt";
system("edit $file");
```

上面的每个例子都对myfile.txt运行一个编辑器，UNIX的编辑器是vi，DOS的编辑器是edit。当然编辑器是全屏幕运行的，所有的标准编辑器命令均能运行。当编辑器退出时，控制权返回给Perl。

可以使用system函数运行任何程序，不只是控制台方式的程序（文本程序）。在UNIX下，下面这个例子运行一个图形时钟：

```
system("xclock -update 1");
```

在DOS / Windows下，下面这个例子用于启动一个图形文件编辑器：

```
system("notepad.exe myfile.txt");
```

基本的命令解释程序

system函数（以及本学时中介绍的大多数函数）允许你使用命令提示符向你提供的命令解释程序的特性。之所以能够这样做，是因为Perl的system命令能够调用一个shell（在UNIX上是/bin/sh，在DOS / Windows中是command.exe），然后将你的system命令赋予该shell。这样，你就能够在UNIX（&）下，执行重定向（>）、管道传输（|）和后台操作等任务，并且能够使用命令解释程序提供的其他特性。

例如，若要运行一个外部命令，并且捕获它在文件中的输出，可以使用下面这个命令：

```
system("perldoc perlfaq5 > faqfile.txt");
```

上面这个命令用于运行perldoc的perlfaq5，并且捕获它在文件中的输出，称为faqfile.txt。这个特定语句在DOS和UNIX中均能运行。

有些特性，比如管道传输和后台操作，在UNIX操作系统下也能按照你的期望来运行，如你在下面看到的那样：

```
# Sort the file whose name is in $f and print it
system("sort $f | lpr"); # Some systems use "lpr"
```

```
# Run "xterm" and immediately return
system("xterm &");
```

在最后一个代码举例中，xterm这个程序启动运行，但是 &将使UNIX的shell启动“后台”的进程。这意味着虽然该进程继续运行，但是 system函数已经完成运行，并将控制权返回给Perl。Perl将不等待xterm完成运行。



在UNIX下，Perl总是将 / bin/sh或类似的命令用于 system函数、管道和反引号（后面将介绍）。不管你的个人 shell被配置成什么，它都是这样使用的。这种用法提供了在各个UNIX系统之间的某种程序的可移植性。

本学时中使用 system函数、管道和反引号的例子放到 Macintosh系统上可能无法运行。详细的说明请参见MacPerl文档中的“ Macintosh的特殊特性”这一节。

11.2 捕获输出

system函数有一个很小的不足，它没有提供特别好的方法，来捕获命令的输出，并将它送往Perl进行分析。如果要以迂回方式进行这项操作，你可以使用下面这个代码：

```
# 'ls' and 'dir' used for example only. opendir/readdir
# would be more efficient in most cases.
system("dir > outfile");    # Use "ls" instead of "dir" for Unix
open(OF, "outfile") || die "Cannot open output: $!";
@data=<OF>;
close(OF);
```

在上面这个代码段中，system运行的命令让它的输出转到一个称为 outfile的文件中。然后该文件被打开并读入一个数组。这时数据 @data包含了dir命令的输入。

这个方法很麻烦，不是一个十分聪明的办法。Perl有另外一个方法处理这个问题，即反引号。用反引号（ `` ）括起来的任何命令均由 Perl作为外部命令来运行，就像通过 system运行的一样，其输出被捕获，并且作为反引号的返回值返回。请看下面这个使用反引号的代码例子：

```
$directory='dir';    # Unix users, use ls instead of dir
```

在上面这个代码段中，运行的是dir命令，其输出在\$directory中捕获。

在反引号中，可以看到所有标准的 shell处理方式：>负责重定向，|负责管道传输。在UNIX下，&负责启动后台任务。不过请记住，在后台运行的命令，或者用 >重定向其输出的命令，均没有输出可以捕获。

在标量上下文中，反引号返回的命令输出是单个字符串。如果命令的输出包含许多行文本，那么字符串中出现的所有文本行均用记录分隔符分开。在列表上下文中，命令的输出被赋予该列表，列表的每一行结尾均有记录分隔符。

现在请看下面这个代码：

```
@dir='dir';    # Use 'ls' for Unix users
foreach(@dir) {
    # Process each line individually.
}
```

在上面这个代码段中，@dir中的输出在foreach循环中处理，每次处理一行。

Perl还有另一种方法能够起到反引号的作用，你可以使用 qx{}表示法。要执行的命令放入花括号 ({}) 中，如下例所示：

```
$perldoc=qx{perldoc perl};
```

通过使用花括号，当反引号作为命令的组成部分出现时，可以不必在反引号的前面加上反斜杠，如下所示：

```
$complex='sort \'grep -l 'conf' *\''; # Somewhat messy
```

也可以将上面的代码段改写为下面的形式：

```
$complex=qx{ sort 'grep -l 'conf' *' }; # Little easier.
```

任何字符均可用来取代{}，成对的字符如<>,()和[]等均可以使用。

避免shell中的概念混乱

Perl与命令解释程序之间的界线有时会变得比较模糊，请看下面的两个例子：

在UNIX系统中：

```
$myhome='ls $HOME';
```

在DOS和Windows系统中：

```
$windows='dir %windir%'
```

在第一个例子中，\$HOME究竟是Perl变量\$HOME，还是shell的环境变量\$HOME呢？在DOS例子中，%windir%究竟是command.com变量windir，还是Perl的哈希结构%windir后随符号%呢？

问题是\$HOME被Perl进行了内插替换，也就是说，\$HOME是Perl的标量变量\$HOME，它可能不是你想要的。在反引号中，变量展开为它们各自的值，就像使用双引号（“ ”）一样。可能的变量名%windir并不在双引号中展开，只有标量和数组名进行了内插替换。

为了避免这种混乱，可以在不想让Perl进行内插替换的变量前面加上一个反斜杠，如下面这两个例子所示：

```
$myhome='ls \$HOME'; # The \ hides $HOME
```

或者

```
$windows='dir %windir%'
```

现在，\$HOME是UNIX shell的HOME变量，%winddir%是command.com的winddir变量。

另一种方法是用qx{}表示法来代替反引号，并用单引号来限定qx，如下面这些例子所示：

```
$myhome=qx' ls $HOME ';
```

或者

```
$windows=qx' dir %winddir% ';
```

Perl将qx”序列视为特殊标号，并且不展开它里面的Perl变量，因此，你可以使用反引号，并且不必用反斜杠对命令中出现的其他反引号进行转义。

11.3 管道

UNIX与DOS / Windows中的管道可用于将不同进程连接在一起，使一个进程的输出成为下一个进程序的输入。请看下面的一组命令，这些命令差不多可以在UNIX（如果将dir改为ls）或DOS中运行：

```
dir > outfile  
sort outfile > newfile  
more newfile
```

dir的输出在outfile中集中，然后使用sort对outfile排序，同时sort的输出被存放在newfile中。接着more命令显示newfile的内容，每次显示一屏内容。

管道允许你执行与上面相同的命令序列，但是不带outfile和newfile，如下所示：

```
dir | sort | more
```

dir的输出被赋予sort，然后sort对数据进行排序。sort的输出被赋予more，每次显示1页。它不需要重定向(>)或临时文件。

这种命令行称为管道命令行，两个命令之间的竖线称为管道。UNIX主要依赖管道来连接它的较小的实用程序。DOS和Windows均支持管道，但与管道一起运行的命令行实用程序要少得多。

Perl程序可以用不同方式纳入管道。首先，你可以编写一个Perl程序，以便接收输入，对输入进行转换，然后插入管道，如下例所示：

在上面这个管道中，Totaler可以是你编写的Perl程序，以便输出目录列表的合计，也可能

```
dir /B | sort | perl Totaler | more
```

是某些统计数字，同时输出目录列表本身。如果你使用UNIX系统，请将dir /B改为ls -1，管道就会按照你的要求来运行。程序清单11-1包含Totaler程序。

程序清单11-1 Totaler的完整清单

```

1:  #!/usr/bin/perl
2:
3:  use strict;
4:  my($dirs,$sizes,$total);
5:
6:  while(<STDIN>) {
7:      chomp;
8:      $total++;
9:      if (-d $_) {
10:          $dirs++;
11:          print "$_\n";
12:          next;
13:      }
14:      $sizes+=(stat($_))[7];
15:      print "$_\n";
16:  }
17:  print "$total files, $dirs directories\n";
18:  print "Average file size: ", $sizes/($total-$dirs), "\n";

```

第6行：输入的每一行均从STDIN读入，再赋予\$_。在管道上，一个程序的STDIN被连接到前一个程序的STDOUT。因此，在上面的例子中，STDIN由dir /B馈入信息。

第9~13行：如果遇到一个目录，在\$dirs中对它的号码单独相加，目录名被输出，循环再次启动运行。

第14~15行：否则，在\$sizes中对文件的大小进行累加，文件名被输出。

第17~18行：输出文件的平均大小，同时输出文件和目录的总数。

Perl参与管道运行的另一种方法是将管道视为既可以读取也可以写入的文件。这是使用Perl中的open函数来实现的，如下所示：

```
# Replace "dir /B" with "ls -1" for Unix
open(RHANDLE, "dir /B| sort |") || die "Cannot open pipe for reading: $!";
```

在上面这个代码段中，open函数打开一个管道，以便从dir/B | sort中读取数据。Perl从该

管道读取数据的这一事实是通过右边的最后一个管道（ | ）来指明的。当 open函数运行时，Perl启动执行dir /B | sort命令。当文件句柄RHANDLE被读取时，sort的输出被读入Perl程序。

现在请看下面这个例子：

```
open(WHANDLE, " | more") || die "Cannot open pipe for writing: $!";
```

这个open函数打开一个管道，以便将数据写入 more命令。左边的管道符号说明Perl正在将数据写入管道。输出到 WHANDLE文件句柄的所有数据均被 more缓存，并每次显示1页。编写这样的函数是每次显示你的一页程序的输出的好办法。

当你完成对已向程序打开的文件句柄（如 RHANDLE和WHANDLE）的操作时，应该正确地关闭句柄，这一点非常重要，因为由 open函数打开的程序必须正确地关闭，若要关闭文件句柄，可以使用 close函数来确保它的正确关闭。当完成句柄操作后，如果不能关闭文件句柄，那么即使你的Perl程序已经终止运行，你编写的程序仍会继续运行。

当关闭了管道上打开的文件句柄后，close函数会指明管道运行是否成功。因此，你应该像下面这样认真检查 close的返回值：

```
close(WHANDLE) || warn "pipe to more failed: $!";
```



open函数也许无法告诉你管道是否已经成功地启动运行，其原因与 UNIX的设计有关。当 Perl创建管道并将它启动时，它不清楚管道是否真的能够工作。如果管道组装正确，并且启动运行了，那么它认为它将能够正确地终止运行。当管道中的最后一个程序完成运行时，它应该返回一个成功退出的状态。close函数能够读取该状态，以了解是否一切运行正常，否则，就会产生一个错误。

11.4 可移植性入门

可移植性，这是Perl擅长的特性之一。无论你的Perl代码是在VMS计算机上运行，还是在UNIX、Macintosh或MS-DOS系统下运行，都具有很强的可移植性，使你编写的Perl代码能够在Perl支持的任何结构上天衣无缝地运行。当需要与基本的操作系统打交道时，比如当进行文件输入/输出时，Perl将设法隐藏所有不必要的细节，使你的代码能够实实在在地运行。



Perl之所以具有如此强的可移植性，第 16学时将对其中的某些原因进行详细说明。

不过，Perl能够向你隐藏的信息是要受到一定限制的。

在本学时中，有些代码例子讲明“这适用于 Windows和DOS，而这适用于 UNIX”，有时又说两种系统都适用，具体情况要根据你使用的系统结构而定。使你自己的程序同时适用于Windows和UNIX，这意味着每个程序必须创建两个版本，一个用于 Windows，一个用于UNIX。当你的程序运行成功并且移植到一个更加特殊的操作系统，如 MacOS 9，那么创建程序的两个版本将会带来更多的问题。

许多情况下，你为一种操作系统结构（如 Windows NT）编写了一个程序，结果却发现它是在另一种结构（比如 UNIX）中运行。由于Perl能够在那么多的不同结构中运行，因此许多人认为在Windows NT下运行Perl程序与它在UNIX下运行是相同的。Web服务器和其他应用程

序经常在不同的操作系统之间转移，因此使你的软件具备可移植性是个非常好的思路。

为每种操作系统创建一个程序的不同版本，使程序能够在任何情况下都能够运行，这是很费时间、很浪费和低效率的。遵照一些规则，你就能够创建到处都能运行的程序，至少可以设法使之到处都能运行，并且便于维护。

下面是编写“到处均可运行的”代码时遵循的一般原则：

- 始终使警告特性处于打开状态，并使用 use strict命令。这样，就可以确保你的代码能够用不同版本的Perl来运行，并且不会出现明显的错误。
- 始终都要检查来自系统请求的返回值，例如，应该使用 open || die，而决不要只使用open。检查返回值可以在将应用程序从一个服务器移到另一个服务器（而不只是在不同的操作系统之间移动）时帮助你发现错误。
- 输出表义性强的出错消息。
- 使用Perl的内置函数，执行你要用 system函数或反引号(`)`来执行的操作。
- 将依赖系统执行的操作（文件I/O、终端I/O、进程控制等）封装在函数中，检查以确保这些操作受当前操作系统的支持。

前面两个原则你已经熟悉。在本书中，所有的代码例子均已检查了关键函数的退出状态，而从第8学时起，所有较大的代码例子均展示了use strict和警告消息。

第3个原则不能忽略，因为它是输出表义性很强的出错消息。在下面这些消息中，哪个最有帮助呢？

```
(no message, or wrong output)
Died at line 15.
Cannot open Foofile.txt: No such file or directory
Cannot open Foofile.txt: No such file or directory at myscript.pl line 24
```

显然，最后一条消息最有帮助。当你安装程序后，而且几个月（或几年）后出了问题，最后一条消息能够说明哪个程序运行失败了（myscript.pl），它想要什么（Foofile.txt），它为何运行失败（没有这个文件...），它在何处运行失败了（第24行）。这些信息可以帮助你迅速排除故障。花费一点儿时间写一条很好的、表义性强的出错消息，总是值得的。

第4个原则意味着只要可能，你就应该使用Perl。若要检索目录列表，最好只使用\$dir=`dir`；但是，如果该程序移植到非Windows系统中，那么它的运行就会失败。一种好的解决方案是使用<*>，而更好的解决方案则是只要可能，就使用opendir/readdir/closedir函数。无论你的程序移植到什么地方，这些解决方案都能运行。

举例说明两个操作系统之间的差别

编写“到处均可运行的”代码时遵循的最后两个原则，即“将依赖系统的操作封装在函数中”以及“检查程序运行所在计算机上的操作系统”，还需要作一点补充说明和演示。

当你坐在计算机面前键入Perl程序时，应该记住，总有一天，你的Perl程序有可能在另一台计算机上运行。你可能建立下一个Amazon.com web站点，它可能从你的PC移植到一台大型Windows NT服务器上，再移植到一群Sun公司1000UNIX服务器上；或者你可能只有一些个人的CGI程序，并且更换了Web提供商，结果发现新提供商配备的是一种不同的服务器。这些情况经常会出现，必须加以考虑。

那么，你的程序究竟如何知道Windows NT与UNIX之间的差别呢？这个问题很简单。Perl

有一个特殊变量 \$^O，即美元符号，插入记号 ^ 和大写字母 O，这个变量包含了程序运行时所在的操作系统结构。例如，在 Windows 和 DOS 下，它包含字符串 MSWin32。在 UNIX 下，它包含你运行的 UNIX 类型，如 linux，aix 和 solaris 等。

下面是依赖你运行的操作系统而执行的一些操作任务：

- 查找关于系统配置的各种信息。
- 对磁盘和目录结构进行操作。
- 使用系统服务程序（email）。

在下面这个例子中，可以查看一个代码段，以便找出系统上的可用磁盘空间。如果有人想要将一个文件上载到一个服务器并且想要确定该文件是否适合在该服务器上运行，那么下面这个例子是有用的。若要在 Windows 系统的当前目录中找出可用的磁盘空间，可以使用类似下面的代码段：

```
# The last line of 'dir' reports something like:  
#     10 dir(s)    67,502,080 bytes free  
# Or on Win98, "MB" instead of "bytes"  
my(@dir,$free);  
@dir='dir';  
$free=$dir[$#dir];  
$free=~s/.*/(\d,]+) \w+ free/$1/;  
$free=~s/,//g;
```

上面这个代码段取出 @dir 中的目录列表的最后一行，使用正则表达式删除不包括大小（即 bytes free 前面的数字和逗号）在内的其他信息。最后，逗号被删除，这样，\$free 只包含原始的空闲磁盘空间。这种方法非常适用于 Windows 系统。对于 UNIX 系统，尤其是 Linux，则可使用下面这个代码段：

```
# Last lines of df -k . reports something like this:  
# Filesystem      1024-blocks  Used   Available Capacity Mounted on  
# /dev/hda1        938485      709863  180139    80%   /  
# And the 4th field is the number of free 1024K disk blocks  
# This format may be particular to Linux.  
my(@dir, $free);  
@dir='df -k .';  
$free=(split(/\s+/, $dir[$#dir]))[3];  
$free=$free*1024;
```

请注意上面这个代码段与前面那个代码段之间的差别。在 Windows 下查找磁盘空间的实用程序是 dir，在 UNIX 下，它是 df -k。df -k 输出的最后一行被分割成若干部分，第 4 个域被放入 \$free 中。df 的输出随着 UNIX 系统的不同而各有差异，通常情况下，报告的域的数目是不一样的，也可能它们使用不同的顺序。你的 Perl 代码只需要选择一个不同的域，就很容易得到调整。

因此，现在有两种完全不同的例程可以用来确定空闲的磁盘空间。可以将它们组合起来，并让适当的例程在每个操作系统上运行，如下所示：

```
if ( $^O eq 'MSWin32') {  
    # The last line of 'dir' reports something like:  
    #     10 dir(s)    67,502,080 bytes free  
    my(@dir,$free);  
    @dir='dir';  
    $free=$dir[$#dir];  
    $free=~s/.*/(\d,]+) \w+ free/$1/;  
    $free=~s/,//g;  
} elsif ( $^O eq 'linux' ) {  
    # Last line of df -k . reports something like this:  
    # /dev/hda1        938485  709863  180139    80%   /  
    # And the 4th field is the number of free 1024K disk blocks
```

```

my(@dir, $free);
@dir='df -k .';
$free=(split(/[\s+/, $dir[$#dir]))[3];
$free=$free*1024;
} else {
    warn "Cannot determine free space on this machine\n";
}

```

这个示例程序现在已经扩展为同时包括 DOS / Windows版本和Linux版本。如果它在任何其他操作系统的机器上运行，就会输出一条警告消息。该例程几乎已经运行结束。现在你需要做的事情是在函数中将该例程隔离出来，使得需要的变量可以声明为专用变量，并且最后的结果可以裁剪，粘贴到任何程序中，并在需要时随时使用。产生的结果代码是：

```

# Computes free space in current directory
sub freespace {
    my(@dir, $free);
    if ( $^O eq 'MSWin32' ) {
        # The last line of 'dir' reports something like:
        #      10 dir(s)   67,502,080 bytes free
        @dir='dir';
        $free=$dir[$#dir];
        $free=~s/.*(\d,)+ bytes free/$1/;
        $free=~s/,//g;
    } elsif ( $^O eq 'linux' ) {
        # Last line of df -k . reports something like this:
        # /dev/hda1      938485 709863 180139  80% /
        # And the 4th field is the number of free 1024K disk blocks
        @dir='df -k .';
        $free=(split(/[\s+/, $dir[$#dir]))[3];
        $free=$free*1024;
    } else {
        $free=0; # A default value
        warn "Cannot determine free space on this machine\n";
    }
    return $free;
}

```

这时，每当你的程序需要了解空闲磁盘空间量时，只需要调用 `freespace()` 函数即可，答案将被返回。如果想在没有列出的另一个操作系统上运行该函数，就会输出一条出错消息。但是，将另一个UNIX型OS添加给该函数，将是很难的，你只能添加另一个子句。

11.5 课时小结

本学时你学习了如何使用系统的实用程序来进行各项操作。使用 `system` 函数，可以运行一个系统实用程序（或者一个管道程序）。反引号（``）能够运行一个系统实用程序，然后捕获它的输出。接着，捕获的输出存放在一个变量中，供 Perl 使用。`open` 函数不仅可以打开文件，而且可以打开程序。该程序可以用 `print` 函数写入尖括号运算符(<>)，或者从尖括号运算符中读取数据。最后，我们介绍了将这些实用程序用于许多不同类型的操作系统的方法，但是不必为每种系统编写不同的程序。

11.6 课外作业

11.6.1 专家答疑

问题：如何打开既可以到达一个命令也可以来自一个命令的管道？例如：`open (P, "| cmd |")`

这个管道似乎并不能运行。为什么？

解答：这项操作实际上相当复杂，因为从同一个进程读取和写入数据可能导致程序“死锁”。这时，你的程序期望 cmd输出某些信息，并且等待带有<P>的数据。同时，因为存在某些混乱状态，cmd实际上等待你的程序输出带有 print P“...”的某些数据。事实上，如果你激活了警告特性，Perl将向你报一条消息：“Can’t do bidirectional pipe（无法执行双向管道操作）”。

如果你为这种问题做好了准备，IPC::Open2模块将允许你打开一个双向开关。这些模块将在第14学时中介绍。

问题：代码\$a=system (“cmd”) 无法按我期望的那样捕获 \$a中的cmd的输出。为什么？

解答：你将system与反引号(`)`的作用混淆了。system函数不能捕获cmd的输出，你需要的代码是\$a=`cmd`。

问题：当我在UNIX下运行带有反引号(`)`的外部程序时，没有捕获出错消息，原因何在？

解答：由于所有UNIX程序，包括Perl程序，都包含两个输出文件描述符，即 STDOUT和 STDERR。文件描述符 STDOUT用于捕获正常程序输出。文件描述符 STDERR用于捕获出错消息。反引号和带有管道的 open函数只能捕获 STDOUT。简单的答案是使用 shell将STDOUT重定向到STDERR，然后运行下面这个命令：

```
$a=`cmd 2>&1`; # run "cmd", capturing output and errors
```

Perl的FAQ（常见问题）详细介绍了这个命令和捕获命令的错误时使用的其他方法。键入 perldoc perlfaq8，便可打开FAQ的有关内容。

11.6.2 思考题

1) 若要使你的程序生成的数据每次显示1页，应使用：

- a. perl myprog.pl | more
- b. open(M, "| more") || die; print M "data...data...data....\n";
- c. open(M, ">more") || die; print M "data..data...data...\n";

2) \$foo的哪个值用于\$r=`dir \$foo`这个语句？

- a. \$foo的shell的值。
- b. Perl的\$foo值被替换，然后运行dir。

3) 下列操作中的哪个操作随着操作系统的不同而变更？

- a. 找出空闲磁盘空间的容量
- b. 获取目录列表
- c. 删除目录

11.6.3 解答

1) 答案是a或b。如果选择a，myprog.pl的所有输出均送入more。如果选择b，那么写入文件描述符M的任何数据均送入more，以便进行分页。

2) 答案是b。若要防止\$foo被Perl展开，你可以使用qx`dix \$foo`。

3) 答案只能是a。b的操作可以用glob,<*>或opendir和readdir来完成。C可以用rmdir

完成。

11.6.4 实习

- 使用第8学时中的统计函数，显示程序清单 11-1 中关于文件大小的更多的统计数据。
- 如果你拥有UNIX系统，将你的UNIX的特定样式添加给freespace()函数。使用Linux作为练习的开始。

China-pub.com

下载

China-pub.com

下载

第12学时 使用Perl的命令行工具

到现在为止，Perl一直只是一个非常简单的解释程序。你将一个程序键入一个文件，然后调用Perl解释程序，以便运行你的程序。不过Perl解释程序比这个程序灵活得多。

Perl解释程序中内置了一个调试程序。使用该调试程序，可以像播放录像带那样运行你的Perl程序。可以将程序倒到开头，使它慢速运行，也可以快速运行，还可以将它定格，以仔细观察程序的内部结构。在查找Perl程序中存在的问题时，调试程序常常是个使用得很不充分的工具。

Perl也能运行不是键入文件的程序。例如，可以直接从系统的命令提示符处运行一些小程序。

在本学时中，你将要学习：

- 如何使用Perl的调试程序。
- 如何使用命令行开关来编写Perl程序。

12.1 什么是调试程序

Perl调试程序是个Perl解释程序的内置特性。它使你能够取出任何一个Perl程序，然后逐个语句运行该程序。在运行过程中，你可以查看各个变量，修改这些变量，让程序运行较长的时间，中断程序的运行，或者从头开始运行该程序。

从你的程序角度来看，它与普通程序并无区别。输入仍然来自键盘，输出仍然送往屏幕。程序并不知道何时停止运行，何时它正在运行。实际上，你可以观察程序的运行情况，根本不必中断程序的运行。

12.1.1 启动调试程序

若要启动Perl调试程序，必须打开操作系统的命令提示符。如果你是DOS和Windows用户，那么要打开MS-DOS的标准提示符C:\。如果是UNIX用户，这个提示符应该是你登录时显示的提示符（通常是%或\$）。

对于运行Perl的Macintosh用户来说，只需从Script菜单中选定Debugger。这时就会为你打开带有提示符的Debugger窗口。

本节中的所有代码例子均使用第9学时的程序清单9-2中的Employee程序。你会发现很容易将一个书签放在这一页上，然后前后翻阅，查找你想要的信息。若要在提示符处启动调试程序（本例中使用DOS提示符），请键入下面这行命令：

```
C:\> perl -d Employee
```

Perl的-d开关可使Perl以调试方式启动运行。命令行上也指明了被调试的程序。然后显示关于版本信息的某些消息，如下所示：

```
Loading DB routines from perl5db.pl version 1.0401
```

```
Emacs support available.
```

Enter h or 'h h' for help.

```
main::(Employee:5):    my @employees=(
main::(Employee:6):        'Smith,Bob,123101,9.35,40',
main::(Employee:7):        'Franklin,Alice,132912,10.15,35',
main::(Employee:8):        'Wojohowicz,Ted,198131,6.50,39',
main::(Employee:9):        'Ng,Wendy,141512,9.50,40',
main::(Employee:10):       'Cliburn,Stan,131211,11.25,40',
main::(Employee:11):      );
DB<1> __
```

该调试程序首先显示版本号（1.0401，你的版本号可能不一样）和help（帮助）提示。接着显示该程序的第一行可执行代码。由于第一个语句实际上包含7行，从“my @employees=”开始，以“”为结尾，因此所有7行语句均显示一个描述，以说明它们来自什么文件（Employee），以及它们是在文件的哪一行或哪几行上找到的（第5至第11行）。

最后，你看到调试文件的提示符DB<1>。1表示调试文件正在等待它的第一个命令。调试程序提示符后面的光标正等待你输入命令。

这时，你的Perl程序实际上暂停在第一个指令-my @employees=(的前面。每当调试程序向你显示程序中的一个语句时，它就是准备要执行的语句，而不是上一个运行的语句。

现在调试程序已经作好准备，等待你输入命令。

12.1.2 调试程序的基本命令

输入调试程序的第一个和最重要的命令是help（帮助）命令。如果在调试程序的提示符处键入h，那么调试程序的所有可用命令均被输出。也可以使用该命令的某种变形，如 h h，它能输出命令和语句的汇总，h cmd用于输出某个命令的帮助信息。

帮助命令的列表也许比较长，一个屏幕显示不下，开头的几个命令显示后，就需要向下滚动。若要每次显示一屏调试命令，可以在命令的前面加上一个|字符。因此，如果想每次查看一屏帮助命令，请使用命令|h。

调试程序的最常用特性是每次运行一个Perl代码的指令。因此，如果继续使用上面的例子，若要转至你的Perl程序的下一个语句，可以使用调试程序的命令n：

```
DB<8> l
Employee:
33:         print_emp($_);
```

当你键入命令n后，Perl就执行Employee程序的第5至11行语句。然后调试程序输出要执行的下一个语句（但尚未运行）my(\$L1,\$F1)=split(‘,’,\$a)；并显示另一个提示符。

当程序运行到这个时候，@employees被初始化为5个名字和工资等。若要查看这些信息，可以将它们输出：

```
DB<1> print @employees
Smith,Bob,123101,9.35,40Franklin,Alice,132912,10.15,35Wojohowicz,Ted,198131,6.50
,39Ng,Wendy,141512,9.50,40Cliburn,Stan,131211,11.25,40
DB<2> __
```

实际上，Perl的任何语句都可以在调试程序提示符后面运行。请注意，来自@employees的数组元素都是一道运行的。可以输入下面的命令，以便很好地将它们输出：

```
DB<2> print join("\n", @employees)
Smith,Bob,123101,9.35,40
Franklin,Alice,132912,10.15,35
Wojohowicz,Ted,198131,6.50,39
Ng,Wendy,141512,9.50,40
Cliburn,Stan,131211,11.25,40
DB<3> __
```

若要继续运行该程序，只需不断键入 n，如下所示：

```
DB<3> n
main::(Employee:23):    @employees=sort {
DB<3> n
main::(Employee:25):          my ($L2, $F2)=split(',',$b);
DB<3> n
main::(Employee:26):          return($L1 cmp $L2
main::(Employee:27):                  ||
main::(Employee:28):                  $F1 cmp $F2
main::(Employee:29):
DB<3> n
main::(Employee:23):    @employees=sort {
DB<3>
```

显然，调试程序将该程序倒退到这个位置，第 23 行准备再次运行。Perl 的 sort 语句实际上是个循环，调试程序逐步通过 sort 代码块中的每个语句。如果你不断键入 n，那么调试程序将不断循环运行，直到 sort 运行结束，这需要花费一定的时间。

若要重复运行上面的命令，也可以在调试程序的提示符处按 Enter 键。

12.1.3 断点

如果不是每次执行一个指令，逐步执行该程序，你也可以让调试程序连续运行你的 Perl 程序，直到到达某个语句，然后停止运行。这些停止运行的位置称为断点。

若要设置断点，必须在程序中选定一个要停止运行的位置。命令 l 用于列出程序的下面 10 行。再次键 l，可以列出下面的 10 行，如此类推。若要列出从某一行开始的程序，请键入 l lineno，其中 lineno 是程序的行号。也可以设定要列出的行的范围，方法是键入命令 l start-end。

在程序清单中，标号 ==> 用于指明调试程序准备执行的当前行，请看下面的代码：

```
DB<3> l 11
23==> @employees=sort {
24:           my ($L1, $F1)=split(',',$a);
25:           my ($L2, $F2)=split(',',$b);
26:           return($L1 cmp $L2
27:                   ||
28:                   $F1 cmp $F2
29: );
30:           } @employees;
31:
32:   foreach(@employees) {
33:       print_emp($_);
DB<3> __
```

在这个例子中，第 33 行是设置断点的好位置。它位于 sort 语句的后面，并且它是程序的主要循环中的第一个语句。你可以在 Perl 程序中的任何位置上设置断点，只要这个断点是个有效的 Perl 语句。但是断点不能设置在花括号（第 30 行）、标点符号（第 29 行）、空行（第 31 行）或只包含注释的代码行上。

若要设置断点，请使用 b breakpoint 命令，其中 breakpoint 可以是行号或子例程名。例如，

若要在第33行上设置断点，可以输入下面这个命令：

```
DB<3> b 33
DB<4>
```

必须知道的另一个关于断点的命令是继续命令 c。命令c向调试程序发出指令，使Perl程序运行到下一个断点或程序的结尾：

```
DB<5> c
main::(Employee:33):           print_emp($_);
DB<6>
```

在这个代码中，调试程序按照要求使Perl程序停止在第33行上，print-emp函数被调用之前。断点仍然可以设置，因此，键入另一个 c子句，可使程序继续运行print_emp()函数，并且再次停止在第33行上：

```
main::(Employee:5):      my @employees=
main::(Employee:6):          'Smith,Bob,123101,9,35,40',
main::(Employee:7):          'Franklin,Alice,132912,10,15,35',
main::(Employee:8):          'Wojehowicz,Ted,198131,6,50,39',
main::(Employee:9):          'Ng,Wendy,141512,9,50,40',
main::(Employee:10):         'Cliburn,Stan,131211,11,25,40',
main::(Employee:11):         ):
DB<1> n                  ←“下一个”指令的命令
main::(Employee:24):        my ($L1, $F1)=split(',',$a);
DB<1>
```

若要查看你在程序中已经设置的断点，可以像下面这样使用命令 L：

```
DB<7> C
131211 Stan Cliburn 11.25 40 450.00
main::(Employee:33): print_emp($_);
```

DB<8>

这个例子显示了调试程序有一个断点，在文件 Employee中的第33行上。

若要撤消程序中的断点，可以采用设置断点时的相同方法使用命令 d，比如 d line或d subname：

```
DB<9> d 33
DB<10>
```

12.1.4 其他调试程序命令

如果想查看print-emp()函数的运行情况，可以用若干不同的方法来执行这项操作。首先使用命令R，重新启动你的程序：

```
DB<11> R
```

```
Warning: some settings and command-line options may be lost!
```

```
Loading DB routines from perl5db.pl version 1.0401
```

```
Emacs support available.
```

```
Enter h or `h h' for help.
```

```
main::(Employee:5):      my @employees=
main::(Employee:6):          'Smith,Bob,123101,9,35,40',
main::(Employee:7):          'Franklin,Alice,132912,10,15,35',
main::(Employee:8):          'Wojehowicz,Ted,198131,6,50,39',
main::(Employee:9):          'Ng,Wendy,141512,9,50,40',
main::(Employee:10):         'Cliburn,Stan,131211,11,25,40',
```

```
main::(Employee:11):      ;
DB<12> b 33
```

命令R用于使Perl程序回到它的开始处，准备再次执行该程序。你已经设置的断点保持已设置状态，Perl程序中的所有变量均复位。在上面这个代码中，断点设置在第33行上。这时可用下面的命令继续运行该程序：

```
DB<13> c
main::(Employee:33):          print_emp($_);
DB<14> _
```

如果执行命令n，就可以执行下面的指令：

```
DB<14> n
131211 Stan Cliburn      11.25 40 450.00
main::(Employee:32):    foreach(@employees) {
DB<15> n
main::(Employee:33):          print_emp($_);
DB<16> _
```

但是，如果用这种方法逐步执行该程序，你就无法查看print-emp()中究竟有什么。若要单步进入print-emp()，不要使用命令n，而应该使用命令s，即单步执行命令。s命令与n命令的作用很相似，不过s命令并不仅仅执行函数，然后转入下一个指令，而是执行函数，然后在函数中的第一个指令处停止运行，如你在下面看到的情况一样：

```
main::(Employee:33):          print_emp($_);
DB<16> s
main::print_emp(Employee:14):      my($last,$first,$emp,$hourly,$time)=
main::print_emp(Employee:15):          split(',',$_[0]);
DB<17> _
```

在这里，显示了print-emp()的第一个语句。也可以用b print-emp设置一个断点，在print-emp()中停止运行。现在可以继续运行该程序，使用命令n逐步通过该函数，如下所示：

```
DB<18> n
main::print_emp(Employee:16):      my $fullname;
DB<19> n
main::print_emp(Employee:17):          $fullname=sprintf("%s %s", $first,
$last);
DB<20> n
main::print_emp(Employee:18):          printf("%6d %-20s %6.2f %3d %7.2f\n",
$emp, $fullname, $hourly, $time,
main::print_emp(Employee:19):          $hourly * $time);
main::print_emp(Employee:20):
DB<21> _
```

还可以在Perl程序运行时修改程序里的变量。例如，若要给员工每小时临时增加2.50美元的工资，可以输入下面的代码：

```
DB<21> print $hourly
10.15
DB<22> $hourly=$hourly+2.50

DB<23> n
132912 Alice Franklin      12.65 35 442.75
main::(Employee:32):    foreach(@employees) {
DB<24>
```

在上面的代码段中，变量\$hourly被输出（10.15），并增加2.50，然后程序继续运行。Printf语句输出\$hourly的新值。

最后，若要退出调试程序，只需在调试程序提示符处键入q。

12.2 练习：查找错误

这个练习向你展示如何使用调试程序来查找程序中的错误。程序清单12-1中的程序存在

一个问题（实际上是两个问题）。它应该输出下面这些消息：

```
20 glasses of Lemonade on the wall
19 glasses of Lemonade on the wall
:
1 glass of Lemonade on the wall
0 glasses of Lemonade on the wall
```

但是它并没有输出这些消息。你的任务是键入程序清单 12-1中的程序，并没法找出里面的错误。这些错误都不属于语句问题，Perl的警告特性没有被激活，同时，use strict没有输出任何消息，不过调试程序应该使得错误可以非常容易地被找到。

当你键入程序后，用调试程序运行 Perl，以便设法找到错误。请记住，要经常输出有关的变量和表达式，并单步通过各个函数调用，每次调用一个函数。

程序清单 12-1 Buggy程序

```
1:  # !/usr/bin/perl -w
2:  # This program contains TWO errors
3:  use strict;
4:
5:  sub message {
6:      my($quant)=@_;
7:      my $mess;
8:      $mess="$quant glasses of Lemonade on the wall\n";
9:      if ($quant eq 1) {
10:          $mess=s/glass/glass/;
11:      }
12:      print $mess;
13:  }
14:
15:  foreach(20..0) {
16:      &message($_);
17:  }
```

这个问题的解决办法请参见本学时结尾处的“思考题”。

12.3 其他命令行特性

调试程序并不是Perl解释程序中可以用命令行开关激活的惟一特性。实际上，许多非常有用的Perl程序都可以在命令行提示符处编写。



Macintosh用户应该通过Script菜单来运行这些命令行代码，方法是选定l-liners，然后将命令键入对话框。

12.3.1 单命令行程序 (One-Liners)

这种程序的关键是在命令行上赋予 Perl的-e开关。-e的后面可以是任何 Perl语句，如下例所示：

```
C:\> perl -e "print 'Hello, world';"
Hello, world
```

你可以使用多个-e开关来插入多个语句，或者用分号将这些语句隔开，如下所示：

```
C:\> perl -e "print 'Hello, world';" -e "print 'Howzit goin?'" 
Hello, worldHowzit goin?
```

应该注意的是，大多数命令解释程序都规定了引号的使用规则。Windows/DOS命令解释程序（command.com或NT的命令Shell）允许你使用双引号将单词括起来，比如上例中的print和Hello World，但是不能随意将双引号放入别的双引号中，也不能随意将>、<、|或^等符号放入双引号中。关于DOS/Windows中引号的详细使用规则，请查看你的操作系统手册。

在UNIX下，一般来说，只要引号配对，即每个开引号有一个闭引号，并且嵌入的引号前面有一个反斜杠\，那么你的代码就没有问题：

```
$ perl -e 'print "Hello, World\n";' -e 'print "Howzit goin?\n"'
```

上面这个代码在大多数UNIX shell(csh、ksh、bash等)下均能运行，并能输出带有正确换行符的消息。若要了解你的shell的引号使用规则的完整列表，请查看shell的在线手册页。

-e开关的一个经常的非常有用的用法是与-d组合起来使用，并将它直接放入Perl的调试程序中，但是没有需要调试的程序：

```
C:\> perl -d -e 1
```

```
Loading DB routines from perl5db.pl version 1
Emacs support available.
```

```
Enter h or `h h' for help.
```

```
main:::(-e:1): 1
DB<1> __
```

这时，调试程序就等待你输入命令了。这种特殊的开关并置方法可以用来测试Perl语句，而你不必编写完整的程序。然后再进行测试、调试、编辑、再测试和再调试。你只需要在调试程序中运行你的语句，直到它能够运行为止。命令行上的1是指最起码的Perl程序，它是计算为1并且返回1的一个表达式。

12.3.2 其他开关

Perl解释程序中的-c开关可供Perl用来查看你的代码，以便找出语句上的问题，但是它实际上并不运行程序：

```
C:\> perl -c Example.pl
Example.pl syntax OK
```

如果出现语句错误，Perl就会输出下面这样一条消息：

```
C:\> perl -c Example.pl
syntax error at Example.pl line 5, near "print"
Example.pl had compilation errors
```

与-w组合起来后，-c开关就能对你的程序进行编译，然后显示Perl认为适当的警告消息。

当你向知识更丰富的Perl用户或系统管理员了解调试代码的情况时，常常必须提供正在使用的Perl解释程序的版本。目前正在使用的Perl语言的主要版本是Perl 5。该解释程序本身有一个可以查看的版本，你可以在命令行上使用开关-v，便可了解该版本号，如下所示：

```
C:\> perl -v
```

```
This is perl, version 5.004_02
```

```
Copyright 1987-1997, 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 5.0 source kit.
```

在上面这个代码中，Perl解释程序的版本是5.004_02。若要了解更加详细的信息，比如该解释程序是如何创建的，何时创建的，等等，可以运行带有-v开关的解释程序，如下所示：

```
C:\> perl -v
Summary of my perl5 (5.0 patchlevel 4 subversion 02) configuration:
Platform:
  osname=MSWin32, osvers=4.0, archname=MSWin32
  :
Compiler:
  cc='bcc32', optimize='-O', gccversion=
  :
Characteristics of this binary (from libperl):
  Compile-time options: DEBUGGING
  Built under MSWin32
  Compiled at Aug  9 1997 21:42:37
  @INC:    C:\PERL\lib\site      C:\PERL\lib
            c:\perl\lib      c:\perl\lib\site      c:\perl\lib\site .
```

如果你试图查找Perl解释程序本身的某个问题，比如安装时出现的一个问题，那么上面这个输出可能对你有用。在这个输出的结尾处，请注意@INC的各个值。这个特定安装的Perl希望在这些目录中找到它的各个模块。当Perl安装后，它不能简单地从一个目录移动到另一个目录。该解释程序本身对于何处能找到它的各个模块有一个内置的思路，如果改变这个思路，就会导致Perl到错误的地方去查找它的模块。关于模块的问题，将在第14学时中详细介绍。

12.3.3 空的尖括号与更多的单命令行程序

迄今为止介绍的尖括号运算符(<>)具有两个功能：

- 1) 如果尖括号中间是文件句柄，尖括号运算符允许你读取文件句柄，比如<STDIN>。
- 2) 如果尖括号中间是搜索模式，尖括号运算符能返回与该模式匹配的文件列表，这称为一个glob，比如<*.bat>。

尖括号运算符还有另一个功能。一组尖括号运算符如果中间没有任何东西，那么它可以读取命令行上所有文件的内容；如果没有给出文件名，则可以读取标准输出。有时空尖括号运算符称为菱形运算符（因其形状而得名）。例如，请看下面这个小型Perl程序：

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

while(<>) {
    print $_;
}
```

如果将上面的程序保存为Example.pl，那么用下面这个命令行运行该程序：

```
C:\> perl -w Example.pl file1 file2 file3
```

就可使运算符<>读取file1的内容，每次读1行，然后读取file2，接着读取file3。如果没有设定文件，则尖括号运算符从文件句柄STDIN中读取数据。这个运行特性类似UNIX实用程序Sed、awk等的特性，如果命令行上设定了文件，则从文件中读取输入，否则读取标准输入。



Perl程序的参数，即去掉Perl的参数-w、-c、-d和-e之后，将被存放在称为@ARGV的数组中。例如，对于上面这个代码段的参数，\$ARGV[0]将包含file1，\$ARGV[1]包含file2，如此等等。

Perl程序的-n开关可用于将任何-e语句封装在该小程序中：

```
LINE:
while(<>) {
... # Your -e statements here.
```

因此，如果要创建一个简短的单命令行程序，从输入数据中删除前导空格，你可以编写下面的命令：

```
C:\> perl -n -e 's/^\\s+//g; print $_;' file1
```

上面这个命令实际上运行类似下面这个 Perl 程序：

```
LINE:
while(<>) {
    s/^\\s+//g;
    print $_;
}
```

在上面这个代码段中，名字为 file1 的文件被打开，并被赋予 while 循环中的 `$_`，每次 1 行。该行用 `S/^\\S+//g` 进行编辑，然后进行输出。`-p` 与 `-n` 开关的作用相同，差别在于语句执行后各个文件便自动输出。因此，重新编写上面这个命令行便产生下面这个命令行：

```
C:\> perl -p -e 's/^\\s+//g' file1
```

当你用 Perl 的单命令行程序来编辑一个文件时，必须注意不要在打开文件进行读取操作的同时，又试图对它进行写入操作，像下面这个例子那样：

```
C:\>perl -p -e 's/\\r//g' dosfile > dosfile
```

上面这个代码段试图从称为 dosfile 的文件中删除回车符。问题是在 Perl 命令被处理之前，dosfile 文件已经被 `> dosfile` 改写。编辑文件的正确方法应该是将输入重定向到另一个文件中，并将文件改为它的原始名字，如下所示：

```
C:\>perl -p -e 's/\\r//g' dosfile > tempfile
C:\>rename tempfile dosfile
```



有些 Perl 爱好者认为，编写简短的“单命令行程序（one-liners）”只不过是一种娱乐。他们认为，程序越复杂，功能越多，就越好。Perl Journal 是介绍 Perl 的一份季刊，它在每一期上刊登了许多单命令行程序。

12.4 课时小结

在本学时中，我们介绍了如何有效地使用调试程序来查找 Perl 程序中存在的问题；介绍了尖括号运算符 (`<>`) 的另一个功能，它使 Perl 能够处理命令行上的所有文件；另外，还介绍了如何使用 Perl 解释程序的 `-n` 和 `-p` 开关，来编写小型单行 Perl 程序。

12.5 课外作业

12.5.1 专家答疑

问题：我真的希望 Perl 有一个图形调试程序。是否存在这样的东西？

解答：是的，有几个这样的调试程序。如果你在 Windows 下使用 Perl，Activestate 拥有很好的图形调试程序。

问题：调试程序不断输出的 `main ::` 究竟是个什么东西？

解答：它与Perl的程序包命名约定有关。它的一些情况将在下一个学时中介绍，因此现在你不必对它考虑太多。

问题：Perl是否还有别的命令行开关？

解答：是的，还有一些。你可通过在线手册查看这些开关的完整列表。若要访问这些信息，请在命令提示符处键入 perldoc perlrun。

12.5.2 思考题

- 1) 程序清单12-1中存在哪些错误？
- 2) 如果命令行上没有给定文件，那么读取`<>`时将返回
 - a. `undef`。
 - b. 来自标准输入的数据行。
 - c. `True`。
- 3) Perl调试程序执行时能够输出Perl语句，这称为跟踪方式。你如何使调试程序进入跟踪方式呢？（提示：必须查看调试程序的帮助消息，才能回答这个问题。）
 - a. 使用`T`命令，使之进入跟踪方式。
 - b. 使用`t`命令，使之进入跟踪方式。

12.5.3 解答

- 1) 首先，在第15行中，范围`(20..0)`无效。范围运算符“`..`”不应该是降序，而应该是升序。这一行应该改为`for ($_=20; $_>-1; $_)`循环，将范围倒过来`(0..20)`，或者使用类似的范围。其次，在第10行上，`$mess=s/glasses/glass/`看上去像是对`$mess`的替换表达式，但实际上并非如此。替换实际上是对`$_`进行的，因为赋值运算符`(=)`应该是个连接运算符`(=~)`。
- 2) 答案是b。如果没有给定文件名，那么`<>`便开始读取STDIN。
- 3) 答案是b。`t`命令能够在程序执行时输出程序的所有语句。`T`命令用于输出堆栈跟踪记录，这是当前正在执行的函数、调用该函数的函数等的列表。

China-pub.com

下载

China-pub.com

下载

第13学时 引用与结构

如果Perl是你使用的一个编程语言，那么本学时将会使你感到颇有兴趣。在大多数编程语言中，你会发现一个概念，即一组数据实际上可以是对另一组数据的引用。有时这些引用称为指针（在pascal或C语言中），有时这种技术称为间接引用（在汇编语言中），而有些语言则根本没有指针的概念（在BASIC或Java中）。如果你以前从未使用过引用、指针或间接引用等概念，那么可能必须多次阅读本学时讲解的某些部分的内容，否则会感到混淆不清。

Perl也拥有这些特殊类型的值，不过在Perl中，它们都称为引用。在Perl中，引用可以用于许多目的，但在本学时中，你要学习的是如何使用引用来自调用带有多个参数的复杂函数和如何创建复杂的数据类型，如列表的列表。

所谓引用，它非常类似老式图书馆中的卡片目录。目录中的每个索引卡指的是图书馆中的一本书。卡片可以指明这本书是什么类型的书（比如小说、非小说、参考书等），并指明这本书放在什么位置。有些卡片目录可能配有对同一本书的若干个引用，它们是不同种类的引用，并且甚至可以参见该目录中的其他卡片。

Perl的引用类似卡片目录，可以指向各组数据。引用能够知道它指向的是何种类别的数据（如标量、数组或哈希），也知道这些数据在什么地方。引用可以被拷贝，但不改变原始数据的任何东西。对于同一组数据，可以进行多次引用。实际上一个引用可以指向其他的引用。

请牢记下面这些要点，慢慢阅读下面几页内容，并且在我们介绍有关的问题时保持清醒的头脑：

- 引用的基本概念。
- 引用的常见结构。
- 运用所有这些概念而建立的一个简要代码例子。

13.1 引用的基本概念

使用赋值运算符，可以创建和赋值一个普通的标量变量，如下所示：

```
$a="Stones"; # A normal scalar
```

在这个代码段建立后，可以创建一个称为\$a的标量变量，它包含字符串“Stones”。到现在为止，一切都很正常。这时，在计算机中的某个地方有一个标为\$a的位置，它包含了该字符串，如下图所示：



如果将标量\$b赋予\$a，比如\$a=\$b，那么会产生该数据的两个拷贝，它们使用两个不同的名字，如下图所示：



如果你想要两个独立的数据拷贝，那么拥有两个拷贝是很好的。但是，如果想让\$a和\$b都引用同一组数据，而不是引用一个数据拷贝，那么必须创建一个引用。所谓引用，它只是

指向一组数据的指针，并不包含实际数据的本身。该引用通常存放在另一个标量变量中。

若要创建对某个既定变量的引用，可以在该变量的前面加上一个反斜杠。例如，若要创建称为\$ref的对\$a的引用，只需要像下面这样将引用赋予\$ref即可：

```
$ref=\$a; # Create a reference to $a
```

这个赋值创建了类似下面这样的条件：



\$ref并不包含用于它自己的任何数据，它只是对\$a的一个引用。变量\$a根本没有改变，它仍然可以照常被赋值（\$a=“Foo”）或显示（print \$a）。

变量\$ref现在包含对\$a的引用。不能简单地对\$ref进行操作，因为它里边没有通常的标量值。实际上，如果输出\$ref，就会显示类似SCALAR(0x0000)的信息。若要通过\$ref获得\$a中的值，必须间接引用\$ref。间接引用可以被视为上面的方块图中按箭头方向的引用。若要通过引用\$ref来输出\$a的值，你可以像下面这样使用另一个\$：

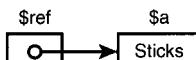
```
print $$ref;
```

在上面的代码段中，\$ref当然包含了引用。增加的一个\$告诉Perl，\$ref中的引用指的是一个标量值。\$ref引用的标量值被取出并输出。

也可以通过引用来修改原始值，这是你对数据拷贝所不能进行的操作。下面这个代码用于修改\$a中的原始值：

```
$$ref="Sticks"; # De-references $ref
```

这项修改形成了类似下面这样的引用关系：



如果你使用\$ref而不是\$\$ref

```
$ref="Break";
```

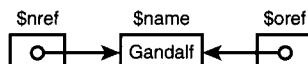
那么存放在\$ref中的引用将被撤消并被实际值取代，如下所示：



上面这个代码段运行后，\$ref不再包含一个引用，它只是一个标量。你可以像任何其他标量值那样，给引用赋值：

```
$name="Gandalf";
$nref=\$name;          # Has a reference to $name
$oref=$nref;           # Has a copy of the reference to $name
```

得到的结果如下：

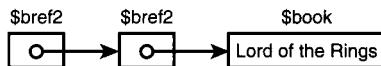


上面的代码段运行后，\$\$oref和\$\$nref均可用于获取值“Gandalf”。也可以存放对某个引用的引用，如下所示：

```
$book="Lord of the Rings";
$bref=\$book;      # A reference to $book
$oref2=\$bref;     # A reference to $bref (not to $book!)
```

下载

在这个例子中，引用链接类似下面的形式：



如果使用\$bref2来输出书名，那么该引用将是\$\$bref2，如果使用\$bref，则该引用是\$\$bref。请注意，\$\$\$bref2多了一个美元符号，它需要增加一层间接引用，才能获得原始值。

13.1.1 对数组的引用

也可以创建对数组和哈希结构的引用。可以像创建对标量的引用那样，使用反斜杠来创建对数组和哈希结构的引用：

```
$aref=\@arr;
```

现在标量变量\$aref包含了对整个数组 @arr的引用。直观地说，它类似下面的形式：



若要使用引用\$aref来访问@arr的各个部分，你可以使用下列代码之一：

`$$aref[0]` @arr的第一个元素

`@$aref[2, 3]` @arr的一个片

`@$aref` @arr的整个数组

为了清楚起见，可以使用花括号将引用与涉及数组的各个部分隔开，如下所示：

`$$aref[0]` 与 `${$aref}[0]`相同

`$$aref[2, 3]` 与 `${$aref}[2, 3]`相同

`@$aref` 与 `@{$aref}`相同

例如，若要使用数组引用\$aref，以便输出@arr的所有元素，可以使用下面这个代码：

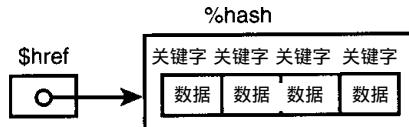
```
foreach $element (@{$aref}) {
    print $element;
}
```

13.1.2 对哈希结构的引用

若要创建对哈希结构的引用，可以使用反斜杠，就像创建标量和数组的引用那样：

```
$href=\%hash;
```

上面这个代码段用于创建对哈希结构 %hash的引用，并将它存放在 \$href中。这个代码段创建的引用结构类似下面的形式：



若要使用对哈希的引用 %href来访问%hash的各个部分，可以使用下面这些代码例子：

`$$href{key}` 访问%hash中的一个关键字，也可以是 `${$href}{key}`

`%$href` 访问整个哈希结构，也可以是 `%{$href}`

若要迭代通过该哈希结构，输出所有的值，可以使用下面这个代码：

```
foreach $key (keys %$href) {
    print $$href{$key}; # same as $hash{$key}
}
```

13.1.3 作为参数的引用

由于整个数组或哈希结构均可被引用，并且该引用可以存放在一个标量中，因此，借助这些引用，你可以调用带有多个数组或哈希结构的函数。

你可能还记得第8学时中我们讲过，下面这种代码段是不能运行的：

```
# Buggy!
sub getarrays {
    my(@a, @b)=@_;
    :
}
@fruit=qw(apples oranges banana);
@veggies=qw(carrot cabbage turnip);
getarrays(@fruit, @veggies);
```

这个代码不能运行，因为 getarrays(@fruit, @veggies) 将两个数组压缩到单个数组 @_ 中。在 getarrays() 函数中，将 @a 和 @b 赋予 @_，会导致现在存放在 @_ 中的 @fruits 和 @vegetables 的所有元素都被赋予 @a。

当所有数组挤入 @_ 之后，就没有办法知道一个数组在何时结束和下一个数组在何时开始。只有一个很大的统一的列表。

这就是引用可以发挥作用的地方。你不必将整个数组传递给 getarrays，只要传递对这些数组的引用，就能够很好地达到你的目的：

```
# Works OK!
sub getarrays {
    my($fruit_ref, $veg_ref)=@_;
    :
}
@fruit=qw(apples oranges banana);
@veggies=qw(carrot cabbage turnip);
getarrays(\@fruit, \@veggies);
```

函数 getarrays() 总是接收两个值，即两个引用，无论这些引用指向的数组有多长。这时，\$fruit_ref 和 \$veg_ref 可以用来显示或编辑数据，如下所示：

```
sub getarrays {
    my($fruit_ref, $veg_ref)=@_;

    print "Fruits:", join(',', @$fruit_ref);
    print "Veggies:", join(',', @$veg_ref);

}
```

当你将对标量、数组或哈希结构的引用作为参数传递给函数时，有几个问题必须记住。当你传递引用时，函数能够对引用指向的原始数据进行操作。请看下面这些例子：

<pre># Passing Values</pre>	<pre># Passing references</pre>
<pre>sub changehash {</pre>	<pre>sub changehash {</pre>
<pre> my(%local_hash)=@_;</pre>	<pre> my(\$href)=@_;</pre>
<pre> \$local_hash{mammal}='bear';</pre>	<pre> \$\$href{mammal}='bear';</pre>

下载

```

    return;
}

%hash=(fish => 'shark',
bird=> 'robin');

changehash(%hash);
}
}

%hash=(fish => 'shark',
bird=> 'robin');

changehash(\%hash);
}
}

```

在左边的例子中，当按正常情况传递哈希结构时，`@_`取得原始哈希结构`%hash`中每个关键字值对的各个值。在子例程`changehash()`中，现在放入`@_`中的哈希结构的各个元素被拷贝到称为`%local_hash`的新哈希结构中。哈希`%local_hash`被修改，该子例程返回。当子例程返回后，`%local_hash`就被撤销，而程序的主要部分中的`%hash`则保持不变。

在右边这个例子中，对`%hash`的引用通过`@_`被传递到子例程`changehash()`中。该引用被拷贝到标量`$href`中，它仍然指原始哈希`%hash`。在子例程中，`$href`指向的哈希结构被修改，子例程返回。`changehash()`返回后，原始哈希结构`%hash`将包含新关键字`bear`。



当数组`@_`用于传递子例程参数时，它是个引用的数组。修改`@_`数组的元素就会改变传递到函数中的原始值。修改传递给子例程的参数，通常被认为是不慎重的一种做法。如果你想让子例程修改传递给它们的参数，那么应该传递对子例程的引用。这种操作方法更加清楚。当传递一个引用时，可以认为原始值是可以修改的。

13.1.4 创建各种结构

创建对数组和哈希结构的引用，可以用来与子例程之间来回传递这些结构，并且可以用来创建下面我们要很快就要介绍的一些复杂结构。不过你应该知道，当你创建了对哈希结构或数组的引用后，就不再需要原始哈希结构或数组。只要对哈希结构或数组的引用存在，即使原始数据不再存在，Perl仍然保留着哈希结构和数组的各个元素。

在下面的代码段中，代码块中创建了一个哈希结构`%hash`，并且这个哈希结构是该代码块的专用结构：

```

my $href;
{
    my %hash=(phone=> 'Bell', light=> 'Edison');
    $href=\%hash;
}
print $$href{light};   # It will print "Edison"!

```

在这个代码块中，标量`$href`被赋予对`%hash`的引用。当该代码块存在时，即使`%hash`已经消失，`$href`中的引用仍然有效（因为`%hash`是代码块的专用结构）。当结构本身已经超出作用域之后，对该结构的引用仍然可以存在，`$href`引用的哈希结构仍然可以修改。

如果你观察上面这个代码块，就会发现，它的唯一目的是创建对哈希结构的引用。Perl提供了一个机制，可以用来创建这样的引用，而不必使用中间的哈希结构`%hash`。这个机制称为匿名存储。下面这个例子创建了一个对匿名哈希结构的引用，并把它存储在`$ahref`中：

```
$ahref={ phone => 'Bell', light => 'Edison' };
```

花括号`({})`将哈希结构括起来，返回对它的引用，但实际上并没有创建新的变量。你

可以使用前面的“对哈希结构的引用”这一节中介绍的所有方法，对匿名哈希结构进行操作。

也可以使用方括号（[]）创建匿名数组。

```
$aaref=[ qw( Crosby Stills Nash Young ) ];
```

同样，也可以使用前面的“对数组的引用”这一节中介绍的方法对数组的引用进行操作。

当引用的变量本身超出作用域时（如果它是个专用变量），那么该引用指向的数据将全部消失，如下所示：

```
{
    my $ref;
    {
        $ref=[ qw ( oats peas beans barley ) ];
    }
    print $$ref[0];           # Prints "oats", $ref is still in scope
}
print $$ref[0];           # $ref is no longer in scope--this is an error.
```

如果use strict正在运行，那么上面这个代码段甚至不进行编译。Perl将\$ref的最后一个实例视为全局变量，这是不允许的。即使没有 use strict，Perl的-w警告特性也会输出一个undefined value（未定义的值）消息。

这些匿名哈希结构和匿名数组可以组合成某些结构形式，我们将在下一节中介绍这些结构。每个哈希结构和数组的引用代表一个标量值，并且由于它是单个标量值，因此可以存放在其他数组和哈希结构中，如下所示：

```
$a=[ qw( rock pop classical ) ];
$b=[ qw( mystery action drama ) ];
$c=[ qw( biography novel periodical ) ];

# A hash of references to arrays
%media=( music => $a, film => $b, 'print' =>$c );
```

13.2 结构的配置方法

下面各节将介绍列表和哈希结构的一些常用结构配置方法。

13.2.1 一个例子：列表中的列表

在Perl中，列表中的列表常常用来代表一种称为二维数组的结构。也就是说，标准数组是多个值的线性列表，如下所示：

[0]	[1]	[2]	[3]
值1	值2	值3	值4

二维数组类似一个值的表格，里面的每个元素按照轴上的一个点来进行编址。索引的第一部分表示行号（从0开始），第二部分是列号，请看下图：

[0] [0] 数据	[0] [1] 数据	[0] [2] 数据
[1] [0] 数据	[1] [1] 数据	[1] [2] 数据
[2] [0] 数据	[2] [1] 数据	[2] [2] 数据

Perl实际上并不支持真正的二维数组。Perl允许你使用数组引用的数组，模仿建立二维数组。

若要创建数组的数组，请使用下面这个原义表达式：

```
@list_of_lists = (  
    [qw( Mustang Bronco Ranger ) ],  
    [qw( Cavalier Suburban Buick ) ],  
    [qw( LeBaron Ram ) ],  
) ;
```

请认真观察上面的代码段。它创建了一个正则列表 @list_of_lists，但是它由对其他列表的引用所组成。若要访问最里层的列表的各个元素（即二维数组中的单元格），可以使用下面这个代码：

```
$list_of_lists[0][1];      # Bronco. 1st row, 2nd entry  
$list_of_lists[1][2];      # Buick. 2nd row, 3rd entry
```

若要确定最外层的列表中的元素数目，你可以像对其他任何数组那样进行操作，使用 \$# 表示法或者使用标量上下文中的数组名：

```
$#list_of_lists;          # Last element of @list_of_lists: 2  
scalar(@list_of_lists);   # Number of rows in @list_of_lists: 3
```

若要确定里层列表中的某个列表的元素数目，可能有一点儿麻烦。语句 \$list_of_lists[1] 返回 @list_of_lists 的第二行中的引用。如果将它输出，则显示类似 ARRAY(0x00000) 这个数据。若要将 @list_of_lists 的一个元素当作数组来处理，请在它的前面加上一个符号 @，如下所示：

```
scalar(@{$list_of_lists[2]});  # From the 3rd row, 2 elements  
$#{${list_of_lists[1]} };       # From the 2nd row, last element is 2
```

若要遍历列表的列表中的每个元素，可以使用下面这个代码：

```
foreach my $outer (@list_of_lists) {  
    foreach my $inner (@{$outer}) {  
        print "$inner ";  
    }  
    print "\n";  
}
```

可以添加下面这样的结构：

```
push(@list_of_lists, [qw( Mercedes BMW Lexus ) ]);  # A new row  
push(@{$list_of_lists[0]}, qw( Taurus ) );           # A new element to one list
```

13.2.2 其他结构

在上一节中，我们介绍了如何使用引用和数组创建基本的 Perl 结构，即列表的列表。实际上可以将数量不受限制的数组、标量和哈希结构的变形组合起来，创建更为复杂的数据结构，比如下面这些结构：

- 哈希结构的列表。
- 列表的哈希结构。
- 哈希结构的哈希结构。
- 包含列表的哈希结构，而列表中又包含哈希结构，等等。

由于本书篇幅有限，无法一一介绍所有这些结构。你安装的每个 Perl 所配备的在线文档包含了一个称为“Perl Data Structures Cookbook(Perl 的数据结构大全)”文档。它详细而明白地描述了这些结构和许多其他数据结构。对于每种数据结构，“Perl Data Structures Cookbook”文档详细描述了下列信息：

- 说明你的结构（原义表示法）。
- 填充你的结构。
- 添加各个元素。
- 访问各个元素。
- 遍历整个数据结构。

若要查看“Perl Data Structures Cookbook”，请在命令提示符处键入 perldoc perldsc。

13.2.3 使用引用来调试程序

当使用引用对程序进行调试时，编程新手常常搞不清楚哪些引用指向什么种类的数据结构。另外，在你习惯之前，语句也容易混淆。Perl提供了一些工具，可以帮助你确定有关的情况。

首先，可以输出该引用。Perl能够显示该引用指向什么结构。例如，下面这个代码行：

```
print $mystery_reference;
```

可以显示

```
ARRAY(0x1231920)
```

这个结构意味着变量 \$mystery_reference 是对一个数组的引用。此外，变量也可以是对标量（SCALAR）、哈希结构（HASH）或子例程（CODE）的引用。若要输出 \$mystery_reference 指向的数组，可以将它作为数组来处理，如下所示：

```
print join(',', @{$mystery_reference});
```

Perl的调试程序也配有一些程序工具，帮助你确定某个引用指向什么数据结构。在调试程序中，你可以像通常那样输出引用。下面这个代码段显示了一个被查看的名叫 \$ref 的引用：

```
DB<1> print $ref
HASH(0x20114dac)
```

显然，\$ref是指一个哈希结构。该调试程序包括一个命令，即命令 x，它将输出该引用和它的内部结构：

```
DB<2> x $ref
0 HASH(0x20114dac)
  'fruit' => 'grape'
  'vegetable' => 'bean'
```

在这个代码中，该引用包含一个带有两个元素（关键字‘fruit’和‘vegetable’）的哈希结构。该调试程序甚至能够输出列表的列表之类的复杂数据结构，如下所示：

```
DB<1> x $a
0 ARRAY(0x20170bd4)
  0 ARRAY(0x20115484)      <-- First row in a list of lists
    0 5
    1 6                  <-- Elements in the first row
    2 7
  1 ARRAY(0x2011fbb4)      <-- Second row in the list of lists
    0 9
    1 10
    2 11
  2 ARRAY(0x2011faa0)      <-- Third row in a list of lists
    0 'a'
    1 'b'
    2 'c'
```

上面的例子显示了一个引用 \$a，它指向一个数组 ARRAY (0x20170bd4)。而这个数组又包含3个别的数据引用，即 ARRAY (0x20115484)、ARRAY (0x2011fbb4) 和 ARRAY

(0x2011faa0)，每个数组包含3个元素。

模块Data::Dumper包含的一些函数能够显示各个引用的内容。Data::Dumper是独一无二的，它的输出格式是有效的Perl格式，它可以存入文件，并在以后被检索，以提供可存储的结构。Data::Dumper模块将在第14学时中介绍。

13.3 练习：另一个游戏——迷宫

当你学习了那么多的新奇概念（引用和结构）之后，需要来一点消遣娱乐了。下面这个练习展示了一种结构和几个引用，并且你可以做一个简单的游戏。

采用探险和狩猎之类的传统游戏方式，你被置于一个迷宫之中，必须找到你的出路。这个迷宫并无奇特之处，它只是由一些房间所组成，并且每个房间至少有一个门。门可以通向位于东、南、西、北的相邻房间。这个游戏的目的是找到一间密室。你会发现通往该密室只有两条路，另外还有许多走不通的路。

首先，键入程序清单13-2，并将它保存为Maze。运行该程序，得到类似程序清单13-1的输出。

程序清单13-1 Maze的输出示例

```
1: You may move East (e)
2: Which way? e
3: You may move South West East (swe)
4: Which way? e
5: :
6: Which way? e
7: You made it through the maze!
```

程序清单13-2 Maze的完整程序清单

```
1: #!/usr/bin/perl -w
2: use strict;
3:
4: my @maze=(
5:     [qw( e   swe we ws ) ],
6:     [qw( ne new sw ns ) ],
7:     [qw( ns   -   ns wn ) ],
8:     [qw( ne w   ne w   ) ],
9: );
10: my %direction=( n=> [ -1, 0], s=> [1, 0],
11:                 e=> [ 0, 1], w=> [0, -1]);
12:
13: my %full=( e => 'East', n => 'North', w=>'West', s=>'South');
14: my($curr_x, $curr_y, $x, $y)=(0,0,3,3);
15: my $move;
16:
17: sub disp_location {
18:     my($cx, $cy)=@_;
19:     print "You may move ";
20:     while($maze[$cx][$cy]=~/([nsew])/g) {
21:         print "$full{$1} ";
22:     }
23:     print "($maze[$cx][$cy])\n";
24: }
25: sub move_to {
```

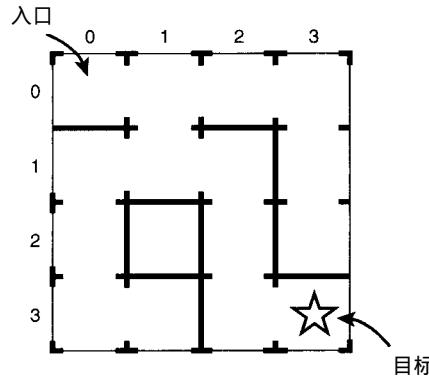
```

26:     my($new, $xref, $yref)=@_;
27:
28:     $new=substr(lc($new),0,1);
29:     if ($maze[$$xref][$$yref]!~/^$new/) {
30:         print "Invalid direction, $new.\n";
31:         return;
32:     }
33:     $$xref += $direction{$new}[0];
34:     $$yref += $direction{$new}[1];
35: }
36:
37: until ( $curr_x == $x and $curr_y == $y ) {
38:     disp_location($curr_x, $curr_y);
39:     print "Which way? ";
40:     $move=<STDIN>; chomp $move;
41:     exit if ($move=~/^q/);
42:     move_to($move, \$curr_x, \$curr_y);
43: }
44:
45: print "You made it through the maze!\n";

```

第1~2行：这两行代码是Perl程序正常的开始。-w使警告特性被激活，use strict用于捕获错误和不恰当的编程做法。

第4~9行：用于定义描述迷宫@maze的结构。显示的迷宫是个 4×4 的栅格，用一个列表的列表来表示。列表的每个元素用于描述迷宫中的任何一个房间可以通往哪些房间，因此，如果你重新设计这个迷宫，请务必留出一条出路。当前的迷宫如下所示：



有一个房间(2,1)是无法进入的，在这个结构中用一个-来表示这个房间。实际上，不能与n、s、e或w匹配的任何字符串均可使用。

第10~11行：当游戏的玩主向北或向南移动时，迷宫中的当前位置就需变更。哈希结构%direction用来根据老的位置和移动方向计算玩主的新位置。如果向“北”移动，则使玩主的x坐标移动-1(向上)，y坐标保持不变。如果向“东”移动，则玩主的x坐标不变，而y坐标增加1。你将在第33~34行代码中看到坐标的变更情况。

第13~15行：程序中使用的变量用my进行声明，以便使use strict恰当地运行。存放在\$curr_x和\$curr_y中的玩主当前位置被设置为0,0。最后目的地\$x和\$y被设置为3,3。

第17行：根据栅格中的x,y坐标，该函数显示玩主可以在每个房间中移动的方向。

第20行：在\$maze[\$cx][\$cy]的房间描述中选择字母n、s、e和w，每次选择1个字母。从哈希结构%full中显示nsew方向的相应描述。这个哈希结构只用于将短名字(n)转换成长名字

(Norfh)，供显示之用。

第25行：该函数取出一个方向（存放在 \$new 中）和对玩主的当前位置的引用。

第28行：方向用 lc 改为小写字母，substr 只取出第一个字母，并将它赋予 \$new。这样，East 变为 e，West 变为 w，s 仍为 s。

第29行：搜索当前房间的 \$maze[\$\$xref][\$yref]，找出给定的方向（n、s、e 和 w）。如果不存在给定的方向，那么它对该房间无效，然后输出一条消息。

第33~34行：玩主的 x 和 y 坐标被更改。如果方向是 e，则 \$direction{e} 是对两个元素的数组的引用（0, 1）。x 坐标将递增 0，即 \$direction{e}[0]。Y 坐标将递增 1，即 \$direction{e}[1]。

第37行：程序的主体从这里启动运行。该循环将不断运行，直到玩主的 x 和 y 坐标（\$curr_x, \$curr_y）与密室的坐标（\$x, \$y）相一致为止。

第38行：显示当前房间的“映像”。

第39行：需要的移动方向读入 \$move，用 chomp 删除换行符。如果玩主键入以 q 开头的任何信息，则游戏结束。

第42行：根据玩主当前需要做的移动和对玩主坐标的引用，调用子例程 move_to()。move_to() 子例程通过调整 \$curr_x 和 \$curr_y，使玩主作相应的移动。

若要修改迷宫，使之采用另一种布局，只需改变存放在 @maze 中的栅格。迷宫不一定需要做成正方形，也不需要给每个房间制作映像，甚至不需要存在一条有效的路径。不过请记住，迷宫不要从它边上的某个房间开始。程序不会检查迷宫的有效性，不过，如果你创建了一个无效迷宫，Perl 就会发出警告。如果要移动迷宫中的密室，只需改变它的 \$x 和 \$y 的值。

13.4 课时小结

本学时我们介绍了引用的基本概念。首先，讲述了如何创建对 Perl 的基本数据结构——标量、数组和哈希结构的引用。然后，介绍了如何使用这些引用，对原始数据结构进行操作。接着，说明了如何创建对哈希结构或数组的引用，不过这种引用没有与此相关的变量名，这种引用称为匿名存储。最后介绍了如何使用引用来创建复杂的数据结构，以及何处可以查找已有数据结构的文档资料。

13.5 课外作业

13.5.1 专家答疑

问题：当我用 print “@LOL” 输出一个列表的列表时，它输出的是 ARRAY (0x101210), ARRAY (0x101400) 等等，为什么？

解答：对于正规数组来说，print “@array” 将输出数组的元素，各个元素之间有一个空格。Print “@LOL” 也产生这样的结果，输出 @LOL 中的各个数组元素。若要输出 @LOL 中每个数组的组件，你必须使用本学时开头的“一个例子：列表中的列表”这一节中介绍的方法。

问题：我试图使用 \$ref = \(\$a, \$b, \$c) 创建一个对列表的引用，结果却产生了一个对标量值而不是列表的引用，为什么？

解答：在 Perl 中，\(\$a, \$b, \$c) 实际上是 (\\$a, \\$b, \\$c) 的简化形式。你得到的结果实际上是对括号中最后一个元素 \$c 的引用。若要获得一个对匿名数组的引用，你应该使用

\$ref=[\\$a , \$b , \$c]。

13.5.2 思考题

1) 语句\$ref=\“peanuts”；运行后，\$ref中包含了什么？

- a. 什么也不包含。该语句无效。
- b. peanuts。
- c. 对一个匿名标量的引用。

2) 下面这个结构可以创建什么？

```
$a=[  
    { name=> "Rose", kids=> [ qw( Ted, Bobby, John ) ] },  
    { name=> "Marge", kids=>[ qw( Maggie, Lisa, Bart ) ] } ,  
];
```

- a. 一个哈希结构的哈希结构，它包含一个列表
- b. 一个哈希结构的列表，它包含一个列表
- c. 一个列表的列表，它包含另一个列表

13.5.3 解答

1) 答案是c。你可以创建对任何值的引用，而不只是创建对标量、数组和哈希变量的引用。你也可以用\$ref=\100；创建对一个数字的引用。如果你的答案是a，那么最好在一个短程序或调试程序中试用一些新数据，看看它们能够产生什么结果。

2) 答案是b。在本学时中我们没有具体介绍这种结构，不过你应该能够猜到这是个什么结构。一个哈希结构（花括号中）的列表（外层方括号）包含一个列表（kids的数据）。

13.5.4 实习

- 修改Maze游戏，使之也能按对角线方向移动。你可以使用4个新关键字来表示这些方向（ne、nw、se和sw是很难编程的）。提示：关键是修改@maze时使用新符号和%direction来表示按什么方向移动，比如[1, 1][-1, -1]等。
- 设计一种结构（即使是纸上谈兵也行），来描述一种电话帐单。帐单本身包含类似哈希结构的关键字和数据（名字，电话号码，地址），帐单的某些部分是列表（明细电话项）。每个明细电话项也可以被视为一个哈希结构（电话接收方，时间）。

China-pub.com

下载

China-pub.com

下载

第14学时 使用模块

你可能已经发现，Perl是一种非常灵活的编程语言。它能够处理文件、文本、数学运算、算法和任何计算机语言中通常遇到的其他问题。该编程语言的很大一部分是专门用于编写特定目的的函数的。正则表达式是该语言的核心部分，对于Perl的使用方法来说，它们非常重要，不过许多编程语言没有正则表达式照样能够很好运行。Perl对外部程序（反引号、管道和system函数）的使用是非常广泛的，不过许多语言根本不使用它们。

程序员都希望尽可能将任何有用的特性纳入该语言的核心中。具有这样的包容性，就会形成一种规模很大并且难以使用的语言。例如，有些语言的设计者认为，支持对 world wide web访问的特性应该纳入该语言的核心中。这是个非常好的思路，但是并不是每个人都需要这个特性。如果10年后web不再像现在这样重要，那么就必须下决心去掉这个特性，许多已经编好的软件就会变得支离破碎。

Perl采取了一种不同的路子。从Perl 5开始，可以使用“模块”对语言进行扩展。模块是Perl例程的集合，它使你能扩展Perl的功能范围。你会发现这些模块能将 web浏览、图形处理、Windows OLE、数据库和几乎任何想像到的特性添加给Perl。不过请记住，Perl的运行并不一定需要这些模块，没有这些模块它照样能够很好地发挥作用。

使用模块，你就能够访问一个很大的工作代码库，以帮助你编写程序。本书的第三部分将专门介绍如何使用Perl模块来编写CGI程序。

在撰写本书时，Perl已经包含3500个以上的模块，有20多个模块已经可以销售给用户。这些模块大多数可以免费转用。可以将这些模块用在你自己的程序中，以实现你想要得到的任何功能。你想解决的许多难题都可以为你解决，你只需安装正确的模块，并且正确地使用这些模块。

在本学时中，你将要学习下面的内容：

- 学习如何在你的Perl程序中使用模块。
- 简单地了解某些内置模块的情况。
- 了解Perl提供的核心模块的列表。

14.1 模块的概述

若要在你的Perl程序中使用模块，可以使用Perl的use命令。例如，若要将Cwd模块纳入你的程序，只需将类似下面的命令插入你的代码：

```
use Cwd;
```

将use Cwd放在代码中的什么位置，这并不重要，不过为了清楚起见和便于维护，它应该放在靠近程序顶部的位置。

这个特定模块曾经用在第10学时中。不过在第10学时中，你不知道它是如何工作的。当

你运行带有use Cwd的程序时，就会出现下列情况：

- 1) Perl解释程序打开你的程序并读入所有代码，直到use Cwd语句被找到。
- 2) 当你的Perl解释程序安装时，它将得到关于它的安装目录的通知。该目录被搜索，以便找出称为Cwd的模块，该模块是包含Perl代码的一个文件。
- 3) Perl读取该模块，该模块运行时需要的所有函数和变量均被初始化。
- 4) Perl解释程序从上次终止的位置开始，继续读取和编译你的程序。

这就是该程序运行的情况。当Perl读取整个程序后，并且在它准备运行时，该模块具备的所有功能就可以供你使用。



你可能注意到use strict与use Cwd很相似。为了避免概念的混乱，use语句是个通用指令，它可以使Perl解释程序执行某项操作。如果使用use strict，它会改变解释程序的运行特性，使之对引用和裸单词变得比较严谨，不过并不存在称为strict的模块。如果使用use Cwd，它将一个模块纳入你的程序。你不必过分担心它们之间的差别，差别很小，不会对你产生很大的影响。

当你将use Cwd插入你的程序中时，一个新函数就可以供你使用，这就是函数 cwd。cwd函数能够返回你的当前工作目录的名字。

14.1.1 读取关于模块的文档

所有Perl模块都配有它们自己的文档资料。事实上，如果你能够使用某个模块，那么就可以访问它的文档，因为文档往往嵌入模块之中。

若要查看模块的文档，请使用带有模块名的 perldoc程序。例如，若要查看 Cwd的文档，只需在操作系统的命令提示符处键入下面的命令：

```
perldoc Cwd
```

然后就可以每次显示1页文档。下面是一页示例文档，它作了一定的压缩：

```
Cwd(3)          13/Oct/98 (perl 5.005, patch 02)          Cwd(3)
```

NAME

```
getcwd - get pathname of current working directory
```

SYNOPSIS

```
use Cwd;
$dir = cwd;
use Cwd;
$dir = getcwd;

use Cwd;
$dir = fastgetcwd;
```

DESCRIPTION

```
The getcwd() function re-implements the getcwd(3) (or
getwd(3)) functions in Perl.
```

```
The abs_path() function takes a single argument and returns
the absolute pathname for that argument. It uses the same
```

下载

```
algorithm as getcwd(). (actually getcwd() is abs_path("."))  
:  
:
```

在这个例子中，Cwd模块实际上允许你使用3个新函数，即 cwd、getcwd 和 fastgetcwd。如果你想使用这些函数，请阅读关于 Cwd 模块的文档。



如果你很想知道模块是如何工作的，就应该去了解它。模块主要是用 Perl 编写的，存放在系统的文件树中。Cwd 模块存放在 Cwd.pm 文件中。该文件的位置可以是不一样的，不过它通常存放在 Perl 的安装目录下的某个位置中。变量 @INC 包含 Cwd.pm 的可能存放位置的名字，若要输出该变量，请在命令提示符处键入 perl -v。

由于许多模块是其他 Perl 程序员免费提供的，所以模块文档的质量差异很大。比较主流的模块，即销售的标准模块，本书中提到的模块，以及流行的模块，如 TK 和 LMP 等，都配有很多好的文档。如果你不知道模块是如何工作的，请查阅第 16 学时的内容，以了解有关的资料，或者干脆询问模块的作者。

14.1.2 什么地方可能出错

如果你的 Perl 安装正确，并且根本没有受到破坏，那么不应该出现任何错误。但是，世界并不是完美无缺的，有时某些地方仍会出错。

如果你看到下面这个出错消息：

```
syntax error in file XXXX at line YYY, next two tokens "use Cwd"
```

那么你应该检查安装的 Perl 的版本。请在系统的命令行提示符处键入下面的命令：

```
perl -v
```

如果 Perl 报告的版本号小于 5，比如 4.036，那么你拥有的 Perl 版本就太旧了，必须对它进行升级。Perl 5 的许多特性它都没有，而且所有老的软件中都存在着安全隐患。实际上第 13 学时中的代码举例都无法在 Perl 4 中运行，现在你应该注意到这个问题了，请立即进行版本的升级。

另一个潜在的错误消息如下所示：

```
Can't locate Cwd.pm in @INC (@INC contains: path...path...path...)  
BEGIN failed--compilation aborted
```

这种错误通常意味着存在下列 3 个问题中的一个：

- 模块的名字拼写有误。

模块的名字是区分大小写字母的。Use Cwd 不同于 use cwd。有些模块名包含冒号 (::)，比如 File :: Find，你必须正确键入冒号。

- 要使用的模块不是标准产品的组成部分，它没有安装在系统的正确位置上。

安装的每个 Perl 均配有大约 150 个模块，这属于“标准产品”。本学时的后面部分内容中列出了其中的一些模块。所有这些模块都应该能够正确运行。你或你的系统管理员必须另外安装不是“标准产品”中的模块。

本书的附录包含如何安装这些额外模块的说明。

- 安装的 Perl 不完整，或者受到了破坏，也可能安装不正确。这种情况是经常发生的。

Perl解释程序会按照出错消息输出的 @INC中的路径查看已安装的模块。如果这些模块移动了位置，被删除，或者无法使用，最简单的解决办法是重新安装 Perl。在处理这个问题之前，首先要搞清出错的模块是否属于标准模块。已经安装的任何附加模块都可能放到其他位置中，这是正常的。关于如何在非标准位置上安装和使用模块的详细说明，请参见本书的附录。

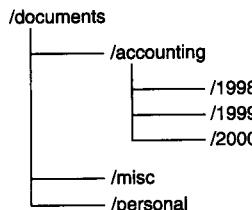
14.2 已安装模块简介

下面我们要简单介绍一下作为Perl核心产品的组成部分且已经安装在你的系统上的一些模块。

14.2.1 文件和目录简介

在第10学时中，我们介绍了如何打开目录并且读取这些目录中包含的文件名列表。接着，提出了如何读取子目录的问题，但是当时我们没有讲述这个问题。现在，我们就要说明如何遍历目录和子目录的方法。

你可以编写一个常用程序，以便在不知道文件所在的确切目录的情况下查找这个特定的文件。例如，你可能想在目录文档下的某个位置上查找名字为 important.doc的文件，如下所示：



这个插图显示了一个目录结构，它位于名字为 documents的父目录下。如果使用 opendir/readdir/closedir来查找documents下的某个位置上的一个文件，那是非常不容易的。首先，必须搜索documents来查找该文件。然后必须搜索 documents下的每个目录，即accounting、 misc 和personal，接着再搜索这些目录下的每个目录，如此等等。

这是过去30年来程序员一次又一次解决的一个老问题。如果你自己编写一个程序来解决这个问题，纯粹是浪费时间。为此，Perl的设计人员采取了一个简单的解决方案，加上了 File :: Find模块。若要在你的程序中使用 File :: Find模块，只需将下面这个命令输入到你的程序中的某个位置，最好是靠近程序的顶部：

```
use File::Find;
```

这时一个称为 find的新函数就可以供你使用了。 find函数的句法如下所示：

```
find subref, dirlist
```

find函数的第二个参数是要搜索的一个目录列表。第一个参数对你来说是新的，它是个子例程引用。你创建的子例程引用很像是标量或数组的引用，它只不过是前面加上一个反斜杠的子例程名。必须使用子例程名前面的 &，才能取得子例程的引用。然后为 dirlist中找到的每个文件和目录调用指明的子例程。

程序清单14-1显示了为查找丢失的important.doc而使用的程序。

程序清单14-1 查找一个文件时所用的程序

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use File::Find;
```

下载

```

4:
5:  sub wanted {
6:      if ($_ eq "important.doc") {
7:          print $File::Find::name;
8:      }
9:  }
10: find \&wanted, '/documents';

```

第1~2行：这两行代码是Perl程序通常开始运行时的代码。-w使警告特性激活，use strict用于捕获错误。

第3行：File::Find模块被插入你的程序。它使你可以运用find函数。

第5行：为‘/documents’下的每个文件和目录调用该函数。如果你有100个文件和12个目录，该例程将被调用112次。

第6行：当wanted()函数被调用时，\$File::Find::name将包含到达被查看的当前文件的路径，\$_只包含文件名。这行代码用于确定文件名是importamt.doc。如果是，则输出全路径名。

第10行：用子例程引用\&wanted和一个目录调用find函数。并为‘/documents’下的每个文件和目录调用wanted()函数。

由find调用的函数将可以使用下列变量：

- \$File::Find::name 当前路径名，目录和文件名。
- \$File::Find::dir 当前目录名。
- \$_ 当前文件名（不带目录）。有一点总是很重要，那就是你不能改变函数中的\$_的值。如果你改变了这个值，应该将它改回来。

程序清单14-2包含了另一个File::Find例子。这个例子用于删除C:和D:驱动器上带有扩展名.tmp的所有文件。这些文件在你的硬盘上不断增加并变得拥挤不堪。你可以很容易使用这个程序，从UNIX系统中删除文件，或者执行各种文件维护操作。

程序清单14-2 删除临时文件所用的程序

```

1:  #!/usr/bin/perl -w
2:  use strict;
3:  use File::Find;
4:
5:  sub wanted {
6:      # Check to see if the filename is not a directory
7:      if (-f $File::Find::name) {
8:          # Verify the filename ends in .tmp
9:          if ( $File::Find::name =~ /\.\tmp$/i) {
10:              print "Removing $File::Find::name";
11:              unlink $File::Find::name;
12:          }
13:      }
14:  }
15: find(\&wanted, 'c:/', 'd:/');

```

程序清单14-2中的大多数程序与程序清单14-1中的程序相似。

第7行：对传递过来的文件名进行测试，以确保它是个正规文件。请记住，这个子例程将同时为文件和目录而调用。

第9~11行：对文件名进行核实，以了解文件名的结尾是否包含.tmp。如果包含.tmp，则用

unlink将文件删除。

14.2.2 拷贝文件

另一个常见操作是拷贝文件，可以在Perl中使用下列步骤执行这项操作：

- 1) 打开源文件，以便读取该文件。
- 2) 打开目标文件，以便写入。
- 3) 读取源文件并写入目标文件。
- 4) 关闭源文件和目标文件。

当然，在执行每个操作步骤后，必须确保没有发生任何错误，并且每个写入操作均取得了成功。告诉你一个比较容易的方法，Perl提供了File::copy模块，它能进行文件的拷贝操作。下面是该模块的一个例子：

```
use File::Copy;
copy("sourcefile", "destination") || warn "Could not copy files: $!";
```

上面这个代码段用于将sourcefile的内容拷贝到destination。如果拷贝成功，copy函数返回1，如果拷贝出现问题，则返回0，并且它会将变量\$!设置为相应的错误代码。

File::copy模块也提供了一个move函数。move函数能够将文件从一个目录移到另一个目录。如果可以通过对文件改名来移动文件，那么该文件将被改名。当源文件和目标文件在同一个文件系统或磁盘上时，通常采取改名的方法。如果无法通过对文件改名来移动文件，那么文件首先拷贝到目标文件名，然后将原始文件删除。请看下面这个例子：

```
use File::Copy;
if (not move("important.doc", "d:/archives/documents/important.doc")) {
    warn "important.doc could not be moved: $!";
    unlink "d:/archives/documents/important.doc";
}
```

在上面的代码段中，文件important.doc从当前目录移到目标目录d:/archives/documents中。如果move函数运行失败，那么就可能存在不完整的目标文件。如果move运行失败，unlink函数能够删除部分拷贝的目标文件。

14.2.3 用于通信的Perl模块

Perl模块的功能并不限于对文件和目录进行操作。还可以使用Net::Ping模块来确定你的系统是否能够在网络上正确地进行通信。

Net::Ping模块是根据UNIX实用程序ping而得名的，而实用程序ping又是根据潜水艇利用声波来测定位置的“乒乓”声音而得名。ping实用程序将一个数据包发送到网络上的另一个系统。如果该系统正在运行，那么它就会发出应答，同时ping命令报告数据包发送成功。下面显示的Net::Ping的工作方式与上面完全相同：

```
use Net::Ping;
if ( pingecho("www.yahoo.com", 15) ) {
    print "Yahoo is on the network.";
} else {
    print "Yahoo is unreachable.";
}
```

在上面这个代码段中，Net::Ping模块提供了一函数叫做pingecho。该函数拥有两个参数，

下载

第一个参数是要查找的主机，在上例中是 www.yahoo.com。第二个参数用于指明pingecho应该等待多长时间才能收到对方的应答，这个时间以秒为单位计算。



由于Perl在Windows 95/98 / NT上运行时所具备的性质，在撰写本书时（1999年夏季）Net::Ping模块还不能运行。Net::Ping需要依赖alarm函数，而在Windows下该函数不能运行。Activestate是开发在Windows下运行的Perl的主要公司，它已宣布准备实现用于Windows的许多遗漏的功能，并将这些修改纳入Perl。

14.2.4 使用English模块

使用English模块，Perl的某些无名的特殊变量将采用比较长的名字，如下例所示：

```
use English;

while(<>) {
    print $ARG;
}
```

在上面的代码段中，while(<>)通常从STDIN中读取一个行输入，并将它赋予`$_`。现在它仍然如此。但是，使用use English，变量`$_`也叫做\$ARG。下面显示了特殊变量及其对应的英文变量的部分列表。

特 殊 变 量	英 文 名
<code>\$_</code>	\$ARG
<code>@_</code>	@ARG
<code>\$!</code>	\$OS_ERROR
<code>\$^O</code>	\$OSNAME
<code>\$0</code>	\$PROGRAM_NAME

若要了解特殊变量及其对应的英文变量的完整列表，请查看 English模块的在线文档。

14.2.5 diagnostics模块

Perl模块diagnostics能够帮助你查找程序中的错误。当你键入本书中的代码例子时，Perl解释程序肯定会发出你不太理解的出错消息。例如，请看下面这个短程序：

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

use strict;
print "For help, send mail to help@support.org\n";
```

它使Perl发出下面的警告消息：

```
In string, '@support' now must be written as '\@support' at line 4
Global symbol "@support" requires explicit package name at line 4
```

diagnostics模块会使Perl具体说明它的错误和警告。可以修改这个示例程序，使它像下面这样包含诊断模块：

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

```
use diagnostics;

print "For help, send mail to help@support.com\n";
```

修改后的程序能够输出一个文字更详细的诊断消息：

```
In string, @support now must be written as \@support at line 4
Global symbol "@support" requires explicit package name at ./diag.pl line 5 (#1)
```

```
(F) You've said "use strict vars", which indicates that all variables
must either be lexically scoped (using "my"), or explicitly qualified to
say which package the global variable is in (using "::").
```

如果认真观察一下这两个消息，就会发现它们之间有着明显的关系。第一条消息的意思很清楚，Perl要求你的电子邮件地址应该写成 help@support.com。第二条消息经过解释，变得更加清楚了一些。由于 use strict是有效的，@support变量应该已经用 my作了声明。但是 @support不是个变量，它是电子邮件地址的一部分，不过它被 Perl转换错了。

该消息前面的字母用于指明你遇到的是何种类型的错误。(W) 表示是个警告，(D) 表示你使用了一个不该使用的语句，(S) 是个严重警告，(F) 表示这是个致命的错误。除了(F) 之外的所有消息类型，你的 Perl程序都会继续运行。

Perl共有60页用于描述它的出错消息。如果你在理解 Perl的简要出错消息时遇到了问题，use diagnostics有时能够帮助你理解出错消息的含义。



通过浏览 perldiag 在线手册页，就可以看到出错消息和诊断消息的完整列表。

14.3 标准模块的完整列表

关于Perl中包含的模块的完整列表，本书将不作详细的说明。下面是标准 Perl产品中的模块列表及其简单的说明。如果想知道模块的作用以及它如何运行，请使用 perldoc，以查看该模块的文档资料。

模 块 名	说 明
AutoLoader	允许Perl只在需要时对函数进行编译
AutoSplit	对模块进行分割，以便自动加载
Benchmark	允许对Perl函重复定时，以便加速基准测试
CGI	允许非常容易地访问用于Web编程的Common Gateway Interface (公用网关接口，第17~24学时介绍)
CPAN	用于访问Perl模块的存档文件，以便安装新模块
Carp	生成出错消息
DirHandle	提供与目录句柄之间的对象接口
Env	将操作系统的环境映射到变量中
Exporter	允许你编写自己的模块
ExtUtils::*	允许你编写自己的模块或者安装模块
File::*	提供更多的文件操作模块，如 File::Copy
File::Spec::*	允许对文件名进行跨操作系统的操作
FileCache	打开的文件数量可以超过操作系统通常允许的数量
FindBin	找出当前正在运行的程序的名字
Getopt::*	允许你处理程序中的命令行选项

下载

(续)

模 块 名	说 明
I18N :: Collate	允许按特定语言排序
IPC :: *	用于进程间的通信，比如使用双通或三通管道进行通信
Math :: *	允许你使用带有任意精度浮点数、整数和复数的扩展数学运算库
Net :: *	允许你获得关于网络主机的信息。例如，Net :: hostent可将IP地址 (如204.71.200.68)转换成主机名(如www.Yahoo.com)
Pod :: *	用于访问Perl的Plain Old Documentation格式化例程
Symbol	允许你对Perl自己的符号表进行查看和操作
Sys :: Hostname	用于获取你的系统的IP主机名
Sys :: Syslog	允许将信息写入UNIX系统的出错记录
Term :: *	为光标位置和清屏等提供终端控制的函数接口
Text :: Abbrev	创建缩写表
Text :: ParseWords	允许对文本进行分析，以便搜索单词
Text :: Soundex	使用Soundex方法，根据标点对单词进行分类
Tie :: *	将Perl的变量与函数连接起来，使你可以实现自己的数组和哈希结构
Time :: *	允许对时间进行分析和处理。例如，你可以将“Sat Jul 24 16:21:38 EDT 1999”这种格式的时间转换成1970年1月1日以来的秒数
constant	允许定义常量值
integer	使Perl有时能够用整数而不是浮点数进行数学运算
Locale	允许进行基于语言的字符串比较(各国语言字符的字符串比较)

下一步进行的操作

如果你想免费了解能使用哪些种类的模块，请使用Web浏览器，以便访问网址 <http://www.cpan.org>。模块按类别进行大致的排列。

有些模块需要C编译器和起码的开发环境来进行安装。在Windows计算机上可能没有这些模块。Activestate的Perl包含一个名叫PPM的实用程序，它可以用来浏览和安装预安装的模块。

本书的附录包含一个按步骤操作的说明，用于在UNIX和Windows计算机上安装模块。这些操作说明将告诉你如何使用CPAN模块(用于UNIX)和Activestate的用于安装新模块的PPM实用程序。

14.4 课时小结

在本学时中，我们介绍了如何使用模块来扩展Perl语言的功能，以便执行许多其他的任务。这种将新功能添加给Perl的通用方法将在本书的其他学时内容中广泛使用。另外，本学时介绍了一些常用的模块，并且给出了标准Perl产品包含的模块的完整列表。

14.5 课外作业

14.5.1 专家答疑

问题：在File :: Find模块中，变量名中的双冒号(::)表示什么？它是否与\$File :: Find :: dir中的相同？

解答：Perl模块能够为变量名建立一些备用区域，称为名空间，这样，模块的全局变量名

与你自己的全局变量名就不会混淆在一起了。因此在 Cwd模块中的全局变量将称为 \$Cwd::x。你的大多数全局变量实际上拥有 \$main::x这个全名，而不是简名 \$x。不过就目前来说，这并不重要。

问题：我有一台安装了 Windows 95/98/NT的计算机，我想使用的模块无法通过 Activestate 的PPM实用程序进行安装。我应该如何安装该模块？

解答：很遗憾，CPAN的大多数模块要求你拥有完整的 UNIX型开发环境，可以对模块进行编译和安装，这种环境在Windows计算机上很难安装。如果你能够非常方便地使用 C编译器，则可以下载一个开发环境，创建你自己的模块，但是这样做并不容易。

问题：我有一个包含 require而不是use的老式Perl程序。require的功能是什么？

解答：require语句与 use相类似。由于 Perl 4没有use关键字，它使用的是 require。require语句可使解释程序查找一个库文件，并将它纳入你的程序，这个功能类似 use的功能。但是它们之间的主要差别是：每当 require语句在运行时（运行期），便执行require功能；而use命令则是在你的程序第一次加载（编译时）时执行其功能。

14.5.2 思考题

1) 如果你想在程序中两次使用 cwd函数，应该使用use Cwd; 几次？

- a. 一次。
- b. cwd的每个实例使用一次，因此是两次。
- c. 一次也不用，因为 cwd是个内置函数。

2) 什么模块能够为 \$_ 变量提供一个别名？

- a. LongVars
- b. English
- c. \$_ 没有别名

14.5.3 解答

1) 答案是a。当你用use将模块插入你的程序后，它的所有函数均可供程序的其余部分使用。

2) 答案是b。use English使得\$_可以使用别名\$ARG。

14.5.4 实习

- 请翻到本书的附录，设法按照那里的说明，通过 CPAN安装模块Bundle :: LWP。对于第24学时中的代码例子来说，需要使用该组模块中的一个模块。

China-pub.com

下载

China-pub.com

下载

第15学时 了解程序的运行性能

编写Perl程序，用于查找文件中的数据，或者与用户进行交互操作，这是非常有用的。但是，当程序运行结束时，将会发生什么情况呢？它的运行结果消失了，你没有得到任何东西来展示你的程序的性能，你会感到怅然若失，一无所获，就像什么也没有发生一样。

数据库能够解决这个问题。数据库可以用于存储数据，供以后使用。设计良好的数据库可以被任何种类的程序使用，以便进行数据的查询、报告和输入。若要设计数据库，必须认真考虑你想存储何种数据，以及如何对它进行存储。另外还要考虑如何访问数据，是每次由一个人访问，还是许多用户同时访问。

在本学时中，我们将要介绍两种方法，以便存储数据供以后检索。

在本学时中，你将要学习下面的内容：

- 创建DBM文件并将数据存储在该文件中。
- 将普通文本文件作为数据库来使用。
- 从文件中的随机位置读取数据和将数据存入文件中的随机位置。
- 为同时访问而锁定文件。

15.1 DBM文件

若要使你的程序能够以非常有条理的方式来存储数据，最简单的方法之一是使用DBM文件。DBM文件是已经与一个Perl的哈希结构连接起来的文件。若要读取和写入DBM文件，只需对一个哈希结构进行操作即可，就像从第7学时以来进行的操作那样。

若要将哈希结构与DBM文件连接起来，可以使用Perl函数dbmopen，如下所示：

```
dbmopen(hash, filename, mode)
```

dbmopen函数将hash与一个DBM文件连接起来。你提供的filename实际上在硬盘上创建两个不同的文件，即filename.pag和filename.dir。Perl使用这两个文件来存储哈希结构。这些文件不是文本文件，不应该对它们使用编辑器。另外，如果这两个文件中的一个为空的，或者与文件中的数据量相比似乎非常大，请不必对此担心，这是正常的。

mode是指对Perl创建的两个DBM文件的访问许可权。如果是UNIX系统，可以使用一组明确的访问许可权，它们用于控制谁能够访问你的DBM文件。例如，0666允许每个人拥有对DBM的读和写访问权。mode 0644允许你读和写这些文件，但其他人只能读这些文件。如果是Windows，只使用0666，因为你不必担心是否拥有对任何文件系统的访问许可权。

如果成功地将哈希结构键入DBM文件，那么dbmopen函数返回真，否则返回假。请看下面这个例子：

```
dbmopen(%hash, "dbmfile", 0644) || die "Cannot open DBM dbmfile: $!";
```

上面这个语句执行后，哈希结构%hash就与称为dbmfile的DBM文件相连接。Perl在你的磁盘上创建一对文件，称为dbmfile.pag和dbmfile.dir，以便存放该哈希结构。如果你将一个值赋予该哈希结构，如下所示，Perl就用该信息更新DBM文件：

```
$hash{feline}="cat";
$hash{canine}="dog";
```

如果要取出信息，Perl就从DBM文件中检索关键字和数据，如下所示：

```
print $hash{canine};
```

若要使哈希结构与DBM文件断开连接，请像下面这样使用带有哈希结构名字的 dbmclose 函数：

```
dbmclose(%hash);
```

当切断哈希结构与DBM文件的连接后，存放在它里面的项目 feline 和 canine 仍将位于该 DBM 文件中，这是 DBM 文件的重要特点。在两次调用 Perl 程序之间，存放在与 DBM 文件相连接的哈希结构中的项目均保留不变。

通常对哈希结构执行的函数，也可以对与 DBM 文件连接的哈希结构执行。哈希结构的函数 keys , values 和 delete 按通常情况运行。可以清空哈希结构和 DBM 文件，方法是将哈希结构赋予一个空列表，比如 %hash=()。也可以对哈希结构进行初始化，方法是在用 dbmopen 将哈希结构与 DBM 文件连接起来之后，将哈希结构赋予一个列表。

15.1.1 需要了解的重点

当你将哈希结构与DBM文件连接起来时，必须了解下列要点：

- 关键字和数据的长度现在是受限制的。通常情况下，哈希结构的关键字和数据长度是不受限制的，与 DBM 文件相连接的哈希结构拥有一个有限的单个关键字和一个数据，合起来的长度通常为 1024 字符左右，这是对 DBM 文件的一个限制。可以存储的关键字和值的总数没有变更，它只受文件系统的限制。
- 运行 dbmopen 以前的哈希结构中的值均被丢失，最好只使用新的哈希结构。请看下例：

```
%h=();
$h{dromedary}="camel";
dbmopen(%h, "database", 0644) || die "Cannot open: $!";
print $h{dromedary}; # Likely will print nothing at all
dbmclose(%h);
```

在上面这个代码段中，当执行 dbmopen 时，哈希结构 %h 中的值，即 dromedare 的关键字将会丢失。

- dbmclose 函数执行后，与 DBM 文件相连接时哈希结构中的值将会消失：

```
dbmopen(%h, "database", 0644) || die "Cannot open: $!";
$h{bovine}="cow";
dbmclose(%h);
print $h{bovine}; # Likely will print nothing
```

哈希结构与DBM文件连接时哈希结构中的值将保留在DBM文件中，此后，哈希结构本身将是空的。

15.1.2 遍历与DBM文件相连接的哈希结构

让我们观察一下尚未与 DBM 文件相连接的哈希结构。如果你编写一个 Perl 程序，用于将约会、电话号码和其他信息存放在哈希结构中。过一段时间后，该哈希结构就会变大。由于在两次运行你的 Perl 程序之间，哈希结构的值均被保留，因此它们决不会跑到别的地方去，除非故意将它们删除。

如果你的DBM文件（称为records）搜集了许多信息，那么下面这个代码段就会出现一些问题：

```
dbmopen(%recs, "records", 0644) || die "Cannot open records: $!";
foreach my $key (keys %recs) {
    print "$key = $recs{$key}\n";
}
dbmclose(%recs);
```

这个代码不存在什么问题。哈希结构首先与一个DBM文件相连接，然后使用keys %recs从哈希结构中取出关键字。用foreach my \$key对关键字列表进行迭代操作，然后输出每个关键字和值。

如果%recs中的关键字列表很大，那么语句 keys %recs的执行需要花费一定的时间。Perl还有另一个函数，可以允许你对哈希结构进行迭代操作，每次取出一个关键字。这个函数称为each。each的句法如下：

```
($key, $val)=each(%hash);
```

each函数返回由两个元素组成的列表，这两个元素中一个是哈希结构的关键字，另一个是它的值。对each进行的每个连续的调用，可以返回哈希结构的下一个关键字值对。当关键字用完后，each就返回一个空表。对较大的哈希结构进行迭代操作时，更好的方法是使用下面的代码：

```
dbmopen(%recs, "records", 0644) || die "Cannot open records: $!";
while( ($key, $value)=each %recs) {
    print "$key = $value\n";
}
dbmclose(%recs);
```



你不一定必须将each函数用于与DBM文件相连接的哈希结构，可以将each用于任何哈希结构。

15.2 练习：一种自由格式备忘录事板

既然你拥有一种将数据存放在磁盘上的简易方法，那么现在可以很好地运用这种方法。下面这个练习展示了一种自由格式的备忘记事板。程序清单 15-2显示了该程序（memopad）的清单。它用于按关键字存储信息，并使你可以使用简单的查询方法来搜索和检索信息。程序清单15-1显示了使用memopad的会话示例。

若要查询memopad程序，只需键入一个问题的名字，后随一个问号。若要用新的方式进行编程，键入“X is Y”形式的短语，其中X是问题，Y是与该问题相关的信息。如果键入“like pattern？”（其中的pattern是在该问题中要搜索的一个正则表达式），你就可以搜索数据库，寻找相似性。符合该正则表达式的所有问题均被输出。若要退出该程序，请在提示符处键入quit。

程序清单 15-1 使用memopad的会话示例

```
Your question: perl?
I don't know about "perl"
Your question: perl is a programming language
Ok, I'll remember "perl" as "a programming language"
Your question: perl's homepage is at http://www.perl.org
```

```

Ok, I'll remember "perl's homepage" as "at http://www.perl.org"
Your question: perl?
perl is a programming language
Your question: like perl?
perl is like perl
perl's homepage is like perl
Your question: quit

```

每当该程序运行时，赋予 memopad 程序的所有信息均被存储，因为数据均存放在与一个 DBM 文件相连接的哈希结构中。

程序清单 15-2 memopad 的完整清单

```

1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my(%answers, $subject, $info, $pattern);
5:
6:  dbmopen(%answers, "answers", 0666) || die "Cannot open answer DBM: $!";
7:  while(1) {
8:      print "Your question ('quit' to quit): ";
9:      chomp($_=lc(<STDIN>));
10:     last if (/^quit$/);
11:     if (/like\s+(.*)\?/) {
12:         $pattern=$1;
13:         while( ($subject,$info)=each(%answers) ) {
14:             if ($subject=~/$pattern/) {
15:                 print "$subject is like $pattern\n";
16:             }
17:         }
18:     } elsif (/(.*)\?/) {
19:         $subject=$1;
20:         if ($answers{$subject}) {
21:             print "$subject is $answers{$subject}\n";
22:         } else {
23:             print qq{I don't know about "$subject"\n};
24:         }
25:     } elsif (/(.*)\sis\s(.*)/) {
26:         $subject=$1;
27:         $info=$2;
28:         $answers{$subject}=$info;
29:         print qq{Ok, I'll remember "$subject" as "$info"\n};
30:     } else {
31:         print "I'm sorry, I don't understand.\n";
32:     }
33: }
34: dbmclose(%answers);

```

第1~2行：这两行是Perl程序通常的起始行，#！行带有-w，这意味着警告特性是激活的。另外，use strict命令可以防止你犯不应该犯的错误。

第6行：使用dbmopen，哈希结构%answers与DBM文件answers相连接。在磁盘上创建两个文件，即answers.pag和answers.dir。

第7行：while(1)执行该永久循环。该循环中的某个位置是个last或exit命令，用于退出该循环。

第9行：这一行代码看上去容易使人混淆，因为它会立即产生若干情况。lc将它的标量参数改为小写字母。由于<STDIN>用在标量上下文中，因此从STDIN中读入一行输入，然后转

换成小写字母，其结果被赋予`$_`。chomp用于删除结尾处的换行符。

第10行：如果输入行只包含单词quit，则退出while循环。

第11行：如果输入行（现在位于`$_`中）与单词like相匹配，然后与某个文本相匹配，再与?相匹配，那么文本被保存在\$1中，在匹配的模式中使用括号。

第12行：第11行的匹配模式中的字符串保存在\$pattern中。

第13~17行：逐个关键字对哈希结构%answers进行搜索，寻找与\$pattern中的字符串相匹配的关键字。当找到每个关键字时，便将它输出。

第18行：（这一行是第11行上开始的if语句的继续。）否则，如果输入行以问号结尾，那么问号前面的所有数据（不包含问号）将被存放在带括号的\$1中。

第19行：\$1中的模式保存到\$subject中。

第20~24行：如果关键字\$subject是在哈希结构%answers中，则输出该关键字和相关的数据。否则，该程序发出应答消息I don't know（我不知道）。

第25~27行：（这一行是第11行上开始的if语句的继续。）否则，如果输入行采用X is Y的形式，则第一部分（X）存入\$subject中，最后一部分存入\$info中。

第28行：\$info中的信息作为\$subject存放在哈希结构%answers中。

第34行：DBM文件与%answers断开连接。

15.3 将文本文件用作数据库

许多情况下，数据库是一种较小和较简单的结构，比如小型系统上的一个用户列表，小型网络上的本地主机，常用的Web站点的列表，或者个人地址文件等，它们都属于简单的数据库形式。而对于简单数据库来说，使用普通文本文件就可以了。但是首先必须考虑到这种数据库存在的某些不足。

将文本文件用作数据库，比使用复杂文件（如DBM文件）或大型数据库（如Oracle或Sybase）有着一些明显的优点。下面列举其中的一些优点：

- 文本文件数据库可以移植。它们可以在各种不同的系统之间进行移动，不会遇到太大的麻烦。
- 文本文件数据库可以使用文本编辑器进行编辑，不必使用特殊工具就能在纸张上打印。
- 文本文件数据库开始时的创建工作很简单。
- 文本文件数据库可以输入到其他程序中，如电子表格、文字处理程序和其他数据库中，没有什么太大的麻烦。凡是能够输入数据的程序，几乎都允许你输入文本。

但是将文本文件用作数据库也有它的不足。若要全面了解它的不足，你应该知道文本文件通常的结构。文本文件数据库的传统结构方式是：文本文件中的每一行都是一个记录，每一行中的各个列称为域。但是，对于你的系统来说，文本文件是个字符流。因此，下面的文本文件

Bob 555-1212

Maury 555-0912

Paul 555-0012

Ann-Marie 555-1190

实际上可以作为下面这个连续字符串来存储：

Bob[space]555-1212[newline]Maury[space]555-0912[newline]Paul[space]...

其中[space]代表一个空格字符，[newline]代表你的操作系统中的一个换行符，有时它是个回车符，有时是个回车符和换行符。每个记录和每个域的字符均封装在一个很长的字符流中，出色的列行显示格式是人能阅读的编辑器、打印机和Perl展示的数据格式。

知道这种结构的特点后，再让我们来看一看文本文件数据库的缺点：

- 不能将数据插入文本文件，只能部分或全部改写文本文件。除了将数据附加在文件的结尾处，如果将数据插入文件中的任何位置，就会拷贝文件中新插入的数据后面的所有数据。

新数据：Susan 555-6613被插入“Bob”的后面

Bob[space]555-1212[newline]Maury[space]555-0912[newline]Paul[space]...

所有这些数据必须拷贝到：

Bob[space]555-1212[newline]Susan[space]555-6613[newline]Maury[space]...

在文件中以这种方式拷贝数据很容易出错，并且速度很慢。

- 相反的情况也一样。从文本文件的中间位置删除数据是很难的。被删除的这部分数据后面的全部数据都必须拷贝到间隔位置。如果要从原始文本数据库中删除 Maury，你必须这样操作：

通过删除下面的数据

Bob[space]555-1212[newline]Maury[space]555-0912[newline]Paul[space]...

所有数据必须拷贝到：

Bob[space]555-1212[newline] Paul[space]555-0012[newline]Ann-Marie[space]...

- 若要在文本文件数据库中查找某个记录，你必须顺序搜索该文件，通常从上向下进行搜索。在DBM文件中，搜索一个记录就像在哈希结构中查找这个记录一样容易，而文本文件则必须查看其每一行，以确定它是个正确的记录。这个过程很慢，随着数据库变得越来越大，搜索过程也越来越慢。

将数据插入文本文件或从文本文件中删除数据

尽管文本文件数据库存在上述缺点，但是它并非一无是处。如果你的文本文件数据库比较小，你可以将文本文件当作一个数组来处理，这样将数据插入数据库或者从数据库中删除数据将是非常容易的。例如，如果下面这个数据库

Bob 555-1212

Maury 555-0912

Paul 555-0012

Ann-Marie 555-1190

被保存到一个称为 phone.txt 的文件中，那么用一个较短的 Perl 程序就可以将该数据库读入一个数组，如下所示：

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

sub readdata {
    open(PH, "phone.txt") || die "Cannot open phone.txt: $!";
    my @array = <PH>;
    close(PH);
    return @array;
}
```

```

my (@DATA) = <PH>;
chomp @DATA;
close(PH);
return(@DATA);
}

```

这里的readdata()函数读取文件 phone.txt，并将数据放入 @DATA中，它不带换行符，并返回该数组。如果增加另一个函数 Writedata()，就可以像下面这样读写该数据库：

```

sub writedata {
    my (@DATA)=@_;    # Accept new contents
    open(PH, ">phone.txt") || die "Cannot open phone.txt: $!";
    foreach(@DATA) {
        print PH "$_\n";
    }
    close(PH);
}

```

这时，若要将记录插入该数据库，只要使用readdata()函数将数据读入一个数组。使用push、unshift或splice函数，将记录插入该数组，然后用 writedata()函数像下面这样再次写出该数组：

```

@PHONELIST=readdata();    # Put all of the records in @PHONELIST
push(@PHONELIST, "April 555-1314");
writedata(@PHONELIST);   # Write them out again.

```

若要从文本文件数据库中删除文本，你可以在重新写入数组之前，对数组 @PHONELIST 使用Splice，pop或shift函数。也可以使用一个循环，人工编辑该数组，比如使用下面这个 grep 循环：

```

@PHONELIST=readdata();    # Read all records into @PHONELIST
# Remove everyone named "Ann" (or Annie, Annette, etc..)
@PHONELIST=grep(! /Ann/, @PHONELIST);
writedata(@PHONELIST);

```

在上面的代码段中，各个记录从 readdata()拷贝到 @PHONELIST。grep对 @PHONELIST 数组进行迭代操作，测试每个元素，以确定它是否与 Ann不相匹配。凡是不匹配的元素均被再次赋予 @PHONELIST。然后数组 @PHONELIST被送回给 writedata()，以便进行写入操作。

15.4 随机访问文件

如果你具有冒险精神，可以像前面提到的那样在文件中进行随机读写操作。下面各节将简单介绍几种工具，可以用来进行随机读写操作，不过我们不准备详细介绍这些工具，因为你并不经常需要使用它们。

15.4.1 打开文件进行读写操作

到现在为止，你已经了解到打开文件的 3种方法。文件可以被打开以便进行阅读、写入、以及将数据附加到它的结尾处。文件还可以打开以便同时进行阅读和写入操作。表 15-1列出了打开文件的不同操作方式。

两项说明：

- 设定“附加”的方式是很麻烦的。在某些系统上，比如在 UNIX上，写入文件的数据总是写到文件的结尾处，无论读取文件的指针位于何处。（后面我们很快还要说明这一点。）
- 决不应该使用+>。一旦文件打开，它的内容就会被删除。

表15-1 打开文件的不同方式

open命令	读	写	附加	如果语句不存在是否创建	是否截断现有数据
open (F , “ <file ”) 或者open(F: file ”)	是	否	否	否	否
open(F , “ >file ”)	否	是	否	是	是
open(F , “ >>file ”)	否	是	是	是	否
open(F , “ +<file ”)	是	是	否	否	否
open(F , “ +>file ”)	是	是	否	是	是
open(F , “ +>>file ”)	是	是	是	是	是

15.4.2 在读写文件中移动

当文件打开时，操作系统始终跟踪你在文件中所处的位置。这个指针称为读指针。例如，当文件初次打开以便进行阅读时，读指针位于文件的开始处，如下所示：



当你读完整个文件后，读指针位于文件的结尾处，如下所示：



若要将指针移到文件中的某个位置，你可以使用 seek函数。seek函数带有两个参数。第一个参数是个打开的文件句柄，第二个参数是文件中你想寻找到的位移。第二个参数是该位移处于什么位置。0是文件的开始处；1是文件中的当前位置；2是文件的结尾。下面是在文件中进行搜索的一些示例代码：

```
# open existing file for reading and writing
open(F, "+<file.txt") || die "file.txt error: $!";
seek(F, 0, 2);           # Seek to the end of the file
print F "On the end";    # Appended to the end of the file
seek(F, 0, 0);           # Seek back to the beginning of the file
print F "This is at the beginning";
```

tell函数返回文件中的当前读指针的位置。例如，当运行上面的代码段后，tell(F)便返回24，即“ This is at the beginning ”的长度，因为文件指针就位于该文本的后面。



这一节只是简单提到seek、tell和open等命令。关于这些命令的详细说明，请参见在线文档。seek、tell和open函数在perfunc手册页中作了说明，可以在命令提示符处键入perldoc perfunc，访问perfunc手册页。另外，在perlpenut一节中，对open命令进行了更加详细的介绍，可以在命令提示符处键入perldoc perlpenut，查看该文档。

15.5 锁定文件

假设你编写了一个非常出色的 Perl程序，并且全世界的人都想使用它。如果你使用 UNIX

或Windows NT计算机，或者使用Windows 95或Windows 98计算机，那么可能有许多人同时运行你的程序。你也可以将程序放到Web服务器上，它运行得如此频繁，以致于你的程序的许多实例互相重叠了。

现在假定你的程序将一个数据库用于它的工作，例如使用前面刚刚介绍的文本文件数据库，不过下面介绍的情况适用于任何一种数据库。请看下面这个代码，它使用上一节介绍的一些函数：

```
chomp($newrecord=<STDIN>);          # Get a new record from the user
@PHONEL=readdata();                  # Read data into @PHONEL
push(@PHONEL, $newrecord);           # Put the record into the array
writedata(@PHONEL);                 # Write out the array
```

这个代码看上去没有任何问题。如果两个人几乎同时运行你的程序，并且试图添加不同的记录，这很可能带来一些问题，它存在相当多的错误。在下面这个插图中，这一组特定的Perl语句几乎是同时在同一个系统上由两个人来运行的（第二个人运行程序的时间比第一个人稍晚一些）。请认真观察。

时间	序号	第1个人	第2个人
	1	\$newrecord = "David 555-1212";	
	2	@PHONEL = readdata();	\$newrecord = "Joy 555-6611";
	3	push (@PHONEL, \$newrecord);	@PHONEL = readdata();
	4	writedata (@PHONEL);	push (@PHONEL, \$newrecord);
	5		writedata (@PHONEL);

从第1个人的角度来看，数据是在第2步上读取的，新记录（“David”）在第3步上被添加给@PHONEL，并在第4步上进行写入操作。

从第2个人的角度来看，数据是在第3步上读取的，新记录（“Joy”）在第4步上被添加给@PHONEL，并在第5步上进行写入操作。

下面是它存在的错误：第2个人在第3步上读取的数据并不包含记录“David”。它尚未由第1个人写入。因此第2个人将“Joy”添加给数组@PHONEL，它并不包含“David”。与此同时，第1个人将@PHONEL的拷贝写入数据库，该记录包含“David”。

当第2个人的程序实例最终运行到第五步时，它将改写第1个人写入的数据。该数据库最终包含记录“Joy”而不是“David”。这显然是个错误。



这个问题实际上比你上面看到的还要严重。上面介绍的情况过分简单化了。更加使人头痛的是，writedata()函数看上去是一次性打开和写入数据的，但是实际上并非如此。多重处理的操作系统实际上能够在写入数据的中间停止程序的运行，并暂时转入另一个程序的运行，然后过几个毫秒又恢复第一个程序的运行。这两个程序都能够同时将不同的数据写入同一个文件。这会使你的数据文件遭到破坏，或者被删除。

这种类型的问题有一个正式的名字，称为竞态条件。程序中的竞态条件很难发现，因为竞态条件的出现和消失取决于一个程序有多少个实例正在同时运行。与竞态条件相关联的错误往往并不明显。

让多个程序同时更新相同的数据是很困难的，不过使用称为锁的机制就能解决这个问题。文件锁可以用于防止一个程序的多个实例同时更改一个文件。

锁定文件会带来两个问题，不过最大的问题是不同的操作系统和不同的文件系统需要使用不同类型的锁定机制。下面两节将介绍如何锁定文件以防止出现上面所说的问题。

15.5.1 锁定UNIX和NT下的文件

若要锁定UNIX和Windows NT下的文件，可以使用Perl的flock函数。flock函数提供了一个“诱导式”锁定机制。这意味着你编写的程序如果需要访问文件，那么它就必须使用flock，以确保没有其他人在同一时间将数据写入该文件。但是，其他程序如果想要修改文件，则仍然可以修改文件，这就是为什么它称为“诱导式”锁定而不是强制锁定。

你可能已经熟悉一种类型的诱导式锁定，即交通信号灯。这种信号灯是为了防止许多车辆同时进入十字路口的相同区域。但是，只有人人都按照信号灯的指示来行车，信号灯才能正常发挥作用。文件锁的情况也一样。可能同时访问一个文件的每个程序必须使用flock函数防止撞车。诱导式锁定并不能防止其他进程访问数据，它只能防止其他进程被锁定。

flock函数带有两个参数，一个是文件句柄，另一个是锁的类型，请看下面的语句：

```
use Fcntl qw(:flock);
flock(FILEHANDLE, lock_type);
```

如果锁定成功，那么flock函数返回真，否则返回假。有时，调用flock会导致你的程序暂停运行，以等待其他锁被打开。下面将很快要说明这个问题。如果使用use Fcntl qw(:flock)，那么你将使用符号名作为lock_type，而不是使用比较难以记住的数字。

文件锁有两种类型，一种是公用锁，一种是专用锁。通常情况下，当想要读取文件时，可以使用公用锁，而当将数据写入文件时，可以使用专用锁。如果一个进程拥有对文件的专用锁，那么这是那里存在的唯一的锁，其他进程则根本不拥有锁。但是，只要不存在专用锁，那么许多进程可以同时拥有公用锁。这是因为只要没有人写入文件，那么许多进程就可以安全地同时读取文件。

下面是lock_type可以使用的一些值：

- lock_SH 这个值要求在文件上设置公用锁。如果另一个进程拥有该文件的专用锁，那么flock函数就会暂停运行，直到专用锁被清除，然后再取出该文件的公用锁。
- Lock_EX 这个值要求对已经打开而用于写入的文件设置一个专用锁。如果其他进程拥有一个锁（公用锁或专用锁均可），那么flock就暂停运行，直到这些锁被清除。
- Lock_UN 这个值用于释放一个锁。但是，很少需要这样做。你只要关闭文件，就可以写出所有未写的数据并释放文件锁。如果释放仍然打开的文件上的锁，就会导致数据被破坏。



当你关闭文件或者当你的程序退出时，即使退出时存在一个错误，用flock设置的锁也会被释放。

在试图读写的文件上加锁是很复杂的。由于打开文件句柄和锁定文件至少需要两个步骤的进程，因此设置文件锁就会带来一些问题，首先必须打开文件，然后才能给文件加锁。如

果用open(FH, ">filename")，然后用flock函数给文件加了锁，那么在你获得该锁之前，你已经修改了该文件(用>对文件截尾了)。通过截尾你可能修改了该文件，而其他进程则对该文件设置了锁。

若要解决这个问题，就需要某种称为信标文件的东西。信标文件是个牺牲性文件，它没有什么重要的内容，凡是对该文件拥有锁的人，均能处理该文件。

若要使用信标文件，你只需要有一个可以用作信标的文件名和两个函数，用于将信标文件锁定和解锁。如程序清单15-3所示。这不是完整的程序，不过它可以作为其他程序的组成部分。

程序清单15-3 通用锁函数

```

1: use Fcntl qw(:flock);
2: # Any file name will do for semaphore.
3: my $semaphore_file="/tmp/sample.sem";
4:
5: # Function to lock (waits indefinitely)
6: sub get_lock {
7:     open(SEM, ">$semaphore_file")
8:         || die "Cannot create semaphore: $!";
9:     flock(SEM, LOCK_EX) || die "Lock failed: $!";
10: }
11:
12: # Function to unlock
13: sub release_lock {
14:     close(SEM);
15: }
```

这些锁函数可以将你当前不想运行的任何代码括起来，即使这些代码与读写文件毫无关系。例如，下面这个代码段(即使同时被若干个进程运行)只允许每次有一个进程输出一条消息：

```
get_lock();      # waits for a lock.
print "Hello, World!\n";
release_lock(); # Let someone else print now...
```

上面代码中的get_lock()和release_lock()函数将在需要给文件加锁时用于锁定文件。



拿着文件锁又等待用户输入(或者其他速度较慢的事件)，这不是个好主意。需要该锁的所有其他程序都会停止运行，以等待该锁被释放。你应该获取你的锁，运行你锁定的敏感代码，然后释放文件锁。

15.5.2 在加锁情况下进行读写操作

下面我们介绍的是文本文件数据库的readdata()和writedata()函数与文件锁一道运行的情况。若要进行这项操作，需要一个信标文件和上一节介绍的get_lock()和release_lock()子例程。

程序清单15-4的第一部分是上一节中的锁代码。

程序清单15-4 在加锁情况下文本文件的输入/输出

```

1: #!/usr/bin/perl -w
2: use strict;
3: use Fcntl qw(:flock);
```

```

4:
5: my $semaphore_file="/tmp/list154.sem";
6:
7: # Function to lock (waits indefinitely)
8: sub get_lock {
9:   open(SEM, ">$semaphore_file")
10:    || die "Cannot create semaphore: $!";
11:   flock(SEM, LOCK_EX) || die "Lock failed: $!";
12: }
13:
14: # Function to unlock
15: sub release_lock {
16:   close(SEM);
17: }
18:
19: sub readdata {
20:   open(PH, "phone.txt") || die "Cannot open phone.txt $!";
21:   my(@DATA)=<PH>;
22:   chomp(@DATA);
23:   close(PH);
24:   return(@DATA);
25: }
26: sub writedata {
27:   my(@DATA)=@_;
28:   open(PH, ">phone.txt") || die "Cannot open phone.txt $!";
29:   foreach(@DATA) {
30:     print PH "$_\n";
31:   }
32:   close(PH);  # Releases the lock, too
33: }
34: my @PHONE;
35:
36: get_lock();
37: @PHONE=readdata();
38: push(@PHONE, "Calvin 555-1012");
39: writedata(@PHONE);
40: release_lock();

```

程序清单 15-4中的大部分代码你已经见过。本学时的前面部分中介绍过 `get_lock()`、`release_lock()`、`readdata()`和`writedata()`等函数。

这个程序的关键部分从第 34 行开始。在这部分代码中，用 `get_lock()`函数设置了一个文件锁。这时用 `readdata()`函数将文件读入 `@PHONE`，并对数据进行操作，然后用 `writedata()`函数将数据重新写入同一个文件。当所有这些操作完成时，如果其他程序等待释放该锁的话，`release_lock()`函数将锁释放。

15.5.3 Windows 95和Windows 98下的加锁问题

情况表明，Windows 95和Windows 98不支持文件锁定。为何不支持呢？因为在这些操作系统下，每次只有一个程序能够打开文件进行写入操作，因此文件加锁就没有必要。如果在Windows 95或Windows 98系统上使用 `flock`函数，就会看到下面这个出错消息：

```
flock() unimplemented on this platform at line...
```

不过，幸好这些操作系统通常每次只支持一个用户进行文件操作。



本书中的程序清单都涉及到使用前面介绍的 `get_lock()` 和 `release_lock()` 函数进行文件锁定的问题。在 Windows 95 或 Windows 98 下使用这些函数就会出错，因为在这两个操作系统下无法实现 `flock` 函数。这些操作系统的程序清单中可以省略这些函数。程序清单中将配有相应的说明来提醒你。

15.5.4 在其他地方使用文件锁的问题

在某些情况下，你可能同时有多个程序在读写文件。由于某个原因，你无法使用 `flock` 函数。即使在能够使用 `flock` 函数的平台上，该函数也不是在所有情况下都适用的。例如，在 UNIX 系统下，对网络文件系统（NFS）上的文件使用 `flock` 函数是不可靠的。也可能你使用的是 UNIX 服务器与 Windows NT 客户机相混合的操作环境，在这种环境中，UNIX 通常支持 `flock` 函数，但是基本的文件系统都不支持 `flock`。

在 Perl 的常见问题（FAQ）列表的第五部分“文件与格式”中，你会看到如何不使用 `flock` 而对文件进行锁定的说明。若要阅读该文档，请查看该文档中的 `perlfaq5` 这一节。

15.6 课时小结

在本学时中，我们介绍了在两次调用 Perl 程序之间存储数据的几种方法。首先讲述了 DBM 文件以及如何使用 DBM 文件将哈希结构与你的硬盘连接起来，接着介绍了文本文件如何将它们用作简单的数据库，最后，为了防止同时访问文件带来的问题，介绍了如何锁定文件和保证数据安全的方法。

15.7 课外作业

15.7.1 专家答疑

问题：我能将第 13 学时中的数据结构存储在 DBM 文件或文本文件中吗？

解答：简单的回答是不行，或者说不容易。而详细的回答是行，不过首先你必须将这种“结构”转换成代表数据的字符串和包含该数据的结构，然后必须将它用作与 DBM 相连接的哈希结构中的一个值。进行这项操作的一个模块是 `Data::Dumper`。

问题：如何锁定 DBM 文件？

解答：DBM 文件可以使用前面介绍的信标锁定系统来锁定。你只需使用程序清单 15-3 中的 `get_lock()` 和 `release_lock()` 函数，将它们放在 DBM 的 `open` 和 `close` 函数的前后：

```
get_lock();
dbmopen(%hash, "foo", 0644) || die "dbmopen: $!";
$hash{newkey}="Value";
dbmclose(%hash);
release_lock();
```

问题：我是否能够以某种方式查看 `flock` 函数准备暂停但实际上并不让它暂停运行？

解答：是的，可以。一个值可以传递给 `flock` 函数，使它不暂停运行（称为非锁定 `flock`）。若要查看 `flock` 是否暂停运行，请像下面这样在锁类型的后面加上 `|Lock_NB`：

```
use Fcntl qw(:flock);
# Attempt to get an exclusive lock, but don't wait for it.
if (not flock(LF, LOCK_EX|LOCK_NB)) {
    print "Could not get the lock: $!";
}
```

你甚至可以等待一会儿，看看是否给文件加了锁，如果你最终没有给文件加上锁，就会输出一条消息：

```
use Fcntl qw(:flock);
$lock_attempts=3;
while (not flock(LF, LOCK_EX|LOCK_NB)) {
    sleep 5; # Wait 5 seconds
    $lock_attempts--;
    die "Could not get lock!" if (not $attempts);
}
```

15.7.2 思考题

- 1) 与DBM文件相连接的哈希结构中的关键字能够存储长度不限的关键字，是吗？
 - a. 是
 - b. 否
- 2) 为什么将数据插入普通文件非常难？
 - a. 周围的数据必须移动以便放置插入的数据。
 - b. 普通文件不能打开后同时进行读和写操作。
 - c. 文件被编辑时，必须锁定。
- 3) 关于锁定文件的说明是在FAQ的哪一节中介绍的？

15.7.3 解答

- 1) 答案是b。按照默认设置，DBM文件中关键字与值的合计长度是1024字符。
- 2) 答案是a。数据在文件中不能随意“向上”和“向下”移动，因此移动周围的数据非常困难。如果选择c也是可以的，不过只有在多个程序同时使用文件时才能成立。
- 3) FAQ的第5节“文件与格式”。

15.7.4 实习

- 编写一段小程序，用来更新文件中的计数器，使每次运行程序时计数器递增1。当程序的多个实例同时运行时，记住要使用文件锁定特性。

China-pub.com

下载

China-pub.com

下载

第16学时 Perl语言开发界

本学时你可以得到一个稍事休息的机会，让我们聊一聊 Perl语言的演变历史和它的文化背景。

也许你认为这一学时应该作为本书的附录或者前言，但如果作为附录或者前言，往往很容易被读者忽略。为了充分利用 Perl的潜在功能，你必须了解Perl语言开发界的一些情况。

了解是什么因素使得Perl语言开发界能够顺利地进行它们的开发活动，这将有助于了解你可以使用什么资源，为什么存在这些资源，这些资源如何运行，为什么 Perl能够成为这样一种编程语言。许多资源都能够帮助你回答这些问题，本学时将帮助你查找这些资源。

在本学时中，你将要学习：

- 关于Perl的一些历史知识。
- 什么是CPAN，你如何使用它。
- 何处获得帮助。

16.1 Perl究竟是一种什么语言

为了获得Perl的文化背景知识，它如何运行，以及你可以使用哪些资源，你有必要知道究竟是什么因素使得Perl能够一脉传承发展至今。

16.1.1 Perl的简单发展历史

1988年，Internet还是个非常不同的系统。首先，它的规模比较小，并且与它今天的样子大不相同。当时的Internet大约只有6万台计算机，而今天它的数量超过1千万台，并且仍在迅速增加。

当时World Wide Web尚未问世，直到1991年在的CERN计算机网络上才提出了World Wide Web的思路，到了1993年，出现了第一个图形浏览器Mosaic。

Internet上的大部分信息都是文字信息。Usenet新闻提供了一个传信系统，使得有兴趣的用户组可以互相保持接触。当时的电子邮件与今天的情况非常相似，主要是文本邮件。文件传送和远程登录形成了Internet上的拥挤信息。

1988年1月，Larry Wall宣布，他编写了另一个软件工具，以替代UNIX下的awk和sed等工具，他将它称为“Perl”。Perl的原始手册对它作了如下的描述：

Perl是一种解释性语言，它非常适合浏览各种文本文件，从这些文本文件中提取有关的信息，并且根据这些信息打印报表。另外，它也是非常适合执行许多系统管理任务的语言。该语言注重实用性（使用方便、有效、完整），而不注重形式上的美观（小巧、精致）。从语言创建者的观点来看，它综合了C、sed、awk和sh等语言的某些最佳特性，因此熟悉这些语言的用户使用Perl语言是不会遇到多大困难的。（语言发展的历史也留下了csh、Pascal甚至BASIC - PLUS的某些遗迹。）Perl的表达式句法与C语言的表达式句法非常接近。如果你有一个问题，

原先使用 sed、awk或sh来解决这个问题，但是 sed、awk和sh感到力不从心，或者这个问题需要运行得稍快一些，而你又不想用 C语言来编写解决这个问题的程序，那么可以使用 Perl。另外，也有一些翻译程序，可以将你的 sed和awk脚本转换成Perl脚本。

Perl的第二个版本于1988年6月推出，它与最新的Perl版本非常相似。Perl 2的大多数特性都很容易理解和使用。它曾经是并且现在仍然是一种功能丰富而完善的编程语言。正如 Perl手册所说，当时Perl的特性主要是用来进行文本处理和执行系统编程任务。

对于Perl来说，1991年是不寻常的一年。1月份，Larry Wall与Randal Schwartz撰写的《Programming Perl》一书的第一版出版。这本书曾经是（并且它后来的版本仍然是）Perl语言的权威参考书。这本书的粉红色封面上印有一只骆驼，这是Perl语言的正式标记。（骆驼并不是一种好看的动物，但是它稳健可靠，值得信赖，并且用处极大。）

这本书的出版时间恰好与Perl 4的推出时间相一致。Perl 4是第一个广泛销售的Perl版本，尽管它最后修改是在1992年，但是直到今天，我们仍然能够在Internet上的遥远角落看到它的踪影。如果你在网上遇到它，你不应使用它。

1994年10月，Perl 5问世。它推出了专用变量、引用、模块和对象等特性，其中“对象”我们尚未介绍。1996年10月，《Programming Perl》一书的第二版（“蓝色骆驼”）上市，它记录了这些新特性。

16.1.2 开放源

Perl取得成功的原因之一与Perl语言的开发和销售方式有关。Perl解释程序是一个开放源软件。开放源是软件开发人员给一个老概念赋予的新术语，它称为“免费分配的软件”。这种软件可免费提供给用户，凡是希望修改软件的源代码的人，都可以查看、调整和修改该源代码。采用这种模式的其他软件包是Linux和FreeBSD操作系统，Apache Web服务器，以及Netscape的开放源浏览器Mozilla。

使用开放源模式实际上是开发软件的一种非常有效的方式。由于开放源代码是由志愿者缩写的，因此软件中通常不会包含不必要的代码。他们认为必要的特性，就会建议纳入源代码中。这种软件的质量非常好，因为对软件有兴趣的每个人都有权并且有责任认真关注它的开发过程，以找出它存在的错误。查看该代码的人越多，错误就越少。



Eric S Raymond撰写了一系列生动的文章，来介绍开发源代码的开发模式，比如为什么它运行得这么好，为什么它在经济上非常合算，以及它是如何开发而成的。第一篇文章“权威与廉价市场”(The Cathedral and the Bazaar)对开放源开发模式如何工作进行了很好的介绍。这些文章的URL在本学时的“课时小结”这一节中列出。

Larry Wall给Perl解释程序申请了版权，因此他拥有Perl的版权，可以根据自己的意愿来处理该软件的版权。但是，与大多数软件一样，用户可以购买Perl的使用许可证。软件许可证说明了软件可以如何来使用和分销，当你打开从商店购买的软件时，会发现它是个印刷得很精美的软件。Larry Wall为你提供了两个不同的软件许可证，供你选择，即GNU普通公用许可证

和Perl艺术家许可证。当你阅读这两种软件版本的许可证后，就可以根据协议条款来选择你需要的许可证，以便将Perl转售给其他用户。

两个许可证的文本都很长，现在将它们的内容概括如下：

- 你可以将Perl解释程序的源代码转售给其他用户，并将版权声明复制给他。
- 你可以修改原来的源代码，只要将你的修改明确标为你自己做的修改，并且既可以放弃这些修改，也可以清楚地指明这不是Perl的标准版本。你也必须提供Perl的标准版本。
- 将Perl转售给别的用户时，你可以收取合理的费用。也可以收取一定的支持费用，但是不得将Perl本身销售给其他用户。你可以将Perl纳入你销售的其他产品中。
- 使用Perl编写的程序不受本许可证的约束。
- 对Perl不作任何担保。

你不得将类似上面这样的对Perl许可证条款的概述用于法律目的。这些概述只是为了使你对这两种许可证的条款有一个大致的了解。



在你想要将Perl纳入另一个软件包之前，应该亲自阅读许可证的内容，并且弄清你的行为是否符合这两种许可证的规定。Perl艺术家许可证包含在销售的每个Perl中，其文件名为Artistic(艺术家)。可以通过网址<http://www.gnu.org>查看GNU普通公用许可证的内容。

有了许可证，Perl就可以在开放论坛中进行开发和改进。运用这种方法，凡是想要阅读和提出修改建议的用户，都可以看到Perl的全部源代码。这种方法有助于实现出色的编程，避免陷入专用的、隐蔽的和模糊不清的软件解决方案之中。

16.1.3 Perl的开发

Perl解释程序、语言以及该语言包含的各个模块的开发是在一个邮件列表上进行的，在这个邮件列表上，Perl的开发人员可以提出修改建议，查看错误报表，对Perl源代码的修改进行争论。

凡是希望参与这个开发过程的人，均会受到欢迎，这正是开放源的特色。不过，为了防止混乱，人们提出的修改意见将由一个核心开发人员小组负责筛选，这些开发人员负责批准和拒绝人们的修改意见，并维护Perl开发的核心内容。修改意见要进行评估，看它们对Perl是否有利，这些修改究竟有多大用处，是否有的人的修改取得了成功。Larry Wall负责监督这个开发过程，担当着仁慈的总监的角色，允许进行确实有利的修改，并否决他认为对Perl不利的修改。

已经推出的Perl版本按两种方法编号。1999年8月以前，它们按照主次修补次数(major.minor_patchlevel)的格式进行编号。因此，4.036_18是指Perl第4版，第36次发布，第18次修补。有时Perl版本号不包含修补次数。截止到1999年秋撰写本书时，Perl的当前版本是5.005。

Perl的下一个版本是5.6。这个版本的编写方案比较传统，它采用主、次版本号格式。因此，Perl 5.6的下一个版本将是5.7等等。

16.2 Perl综合存档文件网 (CPAN)

Perl提供了另一些模块，以便进一步扩大你的开发环境。这些模块均包含在CPAN中。

16.2.1 什么是CPAN

Perl综合存档文件网即CPAN，它是Perl文档和软件的一个大型集合体。该软件是由为Perl语言家族编写模块、程序和文档的志愿软件人员共同开发的。

CPAN中可以使用的模块非常广泛。在撰写本书时，CPAN的建立大约已有4年历史，可供安装的模块超过3500个。这些模块涵盖的编程问题的范围极其广泛。表16-1是个简短的列表，可以使你对CPAN中的模块有个大致的了解。

表16-1 CPAN中的部分模块一览表

TK	用于Perl程序的图形接口。可以使用特定的工具箱模块来访问特定的图形库，比如Win32 API、Gtk、Gnome、Qt或XII工具箱
Net::*	网络模块。它们是用于Mail、Telnet、IRC、LDAP和40多个其他程序的接口
Math::*	包含30多个模块，用于复数、快速傅立叶转换和矩阵操作的各种结构
Date::Time::*	* 用于将日期/时间转换成各种不同格式和将各种格式转换成日期/时间并对它们进行操作的模块
Date::Tree::*	用于对链接列表和B-树状结构之类的数据结构进行操作的模块
DBI::*	数据库的通用结构
DBD::*	商用和免费数据库的接口，这些数据库包括 Oracle、informix、Ingres、ODBC、MsSQL、MySQL、Sybase和许多其他数据库
Term::*	对文本模式的屏幕（如DOS的Command窗口或UNIX终端仿真程序）进行精确控制的模块
String::*, Text::*	包含几十个模块，用于对文本进行分析和格式化
CGI::*	用于Web页的创建、服务、提取和分析的各个模块
URI::*	
HTML::*	
LWP::*	
GD, Graphics::*, Image::*	用于对图形和图像进行操作的各个模块
Win32::*, Win32API::*	用于对Microsoft Windows进行操作的模块

需要记住的最重要一点是，对于大多数问题来说，已有的模块至少可以部分地解决某个问题。CPAN中的这些解决方案已经具有相应的代码，经过了测试，并且有许多程序员对代码进行了审查，以保证它们的正确性和完整性。

CPAN中的所有模块的版权均由各自的开发人员所拥有，因此你应该阅读每个模块所附的README文件，以了解使用模块时应该遵守的条款。大多数情况下，这些模块的销售条款与Perl本身的条款相同，也就是要按照Perl艺术家许可证或GNU普通公用许可证的条款进销售。

CPAN也是一个标准模块的名字，它用于帮助用户将辅助模块安装到你的Perl中。CPAN模块在本书附录“安装模块”中作了说明。

16.2.2 为什么人们愿意提供自己的开发成果

在过去半个世纪的计算机编程发展历程中，程序员一次又一次解决着一些相同的问题。从50年代以来，搜索、排序、通信、读取、写入，这些编程问题实际上没有多大的变化。关于计算机编程理论和管理的一些著作在二三十年以后仍然可用。

一次又一次地解决相同的编程问题并不总是一件有意思的事情，并且常常会产生一些质量不高的解决方案，这叫做“仿制车轮”。最终，程序员对解决一些有意思的编程问题产生了极大的兴趣。

令程序开发人员感到尴尬的一种情况是：花费了很长的时间，投入了大量精力，以便解决一个复杂的问题，结果却发现可以使用一个简单而巧妙的方案就可以解决这个问题。这种尴尬促使程序开发人员去寻求一些方法以便与其他人共享代码。共享代码带来的一个非常有趣的副产品是可以产生更好的代码，因为其他程序员将会发现你的代码中存在的问题，而你自己却没有注意到。

CPAN是Perl语言开发界为了避免进行不必要的开发工作所作努力的结果。它所包含的模块能使你不会经历“仿制车轮”的尴尬。

大多数模块的质量是很好的，模块与Perl一样，都是在开放源生产方式下开发的。当你在系统中安装一个模块时，你将自动拥有该模块的源代码。你可以自己查看源代码，并且根据许可证条款的规定，将源代码的各个部分用于你自己的程序，并可修改源代码，甚至可以与源代码的开发者联系，提出修改建议。

从表面上看，CPAN的开发是大量程序员共同努力的结果，但开发人员为CPAN提供自己的开发成果的实际原因是千差万别的。有时是为了帮助其他人解决类似的编程问题，有时是为了达到一个很好的目标，有时是为了得到同行们的尊重和崇敬，这是一种很强的动力。不管原因是什么，最终结果是大量的成果可以用于你自己的程序。

16.3 下一步你要做的工作

当你阅读了本书前面的三分之二的内容之后，应该对Perl的基本概念有所了解了。不过你并没有学到该语言的全部知识。在我的书架上，至少放着5、6本关于Perl语言的著作，除去重复的内容，它们总共有2300页左右，尽管如此，仍然有许多问题没有包括。

你无法从一种资源中学到Perl的全部知识，不过下列各节能够告诉你下面应该做的一些工作。

这里介绍的一些资源都是按照你查找资源时应该遵循的次序列出的。虽然有些例外情况，但是总的来说，按照这个顺序来学习，可以用最快的方式解决你的问题。

16.3.1 要做的第一步工作

当你遇到一个Perl的问题时，要确定必须采取的第一步操作是很困难的。你会感到不知所措，如果你为解决这个问题花了一点儿时间之后，你可能变得心烦意乱。不过，不要慌，要相信自己能够解决所有问题的。不管怎样，这是重要的第一步。大多数人在解决一个问题时如果进展不大，就会灰心丧气，这会影响你清晰的思路，结果会把事情搞得更糟。

这时，应该暂时放下你的工作，让自己平静下来，精神放松。最终你一定能够解决你的难题。

16.3.2 最有用的工具

你的Perl工具箱中的最有用的工具就是Perl本身。首先，你必须搞清要解决的问题究竟是个什么性质的问题。通常问题可以分为两类，一类是语句错误，另一类是逻辑错误。

如果你的问题与语句相关，通常可以将它分为两个较小的问题。一个问题是某些Perl元素的使用不正确，另一个问题是键入错误。请运行你的程序，查看出错消息。出错消息通常能够指明Perl对出错代码行的最佳判断。你可以查看这个代码行，看看是否存在下列情况：

- Perl的出错消息是否专门指明你应该查看代码行上的哪个位置。然后请查看这个位置。Perl解释程序可以成为你查找错误的最佳助手。
- 左括号、左方括号和左花括号是否有匹配的右括号？
- 是否仔细检查了输入文字的拼写？请再检查一次。你会惊奇地发现有多少程序错误是拼写不正确造成的。
- 是否漏掉了什么东西？比如逗号或句号？
- 指定的某一行的前面一行是否正确？
- 如果你回到本书中介绍某种类型的语句这一节，能找到类似你编写的代码的示例代码吗？
- 如果你从另一个源代码中拷贝了代码，是否检查了其他位置，以便找出类似的一段代码？它可能存在错误。

如果你的Perl程序能够运行，但是它不能产生正确的结果，那么你的逻辑可能有问题。在你分析原因之前，请执行下列操作步骤：

- 1) 确保程序中以 # ! 开头的代码行包含一个 -w。
- 2) 确保程序顶部附近的某个位置上有 use strict。

许多明显的逻辑错误都是 -w 和 use strict 能够捕获的简单错误，请使用这些工具。如果仍有问题，请继续阅读下面的内容。

16.3.3 查找程序中的错误

如果你能够肯定你的程序的语句是正确的，但是它无法执行正确的操作，那么你就必须进行基本的程序调试。

调试程序时首先和最常用的一个方法是使用普通的 print 语句。如果你在程序中小心地使用这个语句，它能够对正在运行的程序进行某种运行期诊断。请看下面这个例子中 print 是如何运行的：

```
sub foo {
    my($a1, $a2)=@_;
    # Diagnostic added to see if everything's OK.
    print STDERR "DEBUG: Made it to foo with $a1 $a2\n"
}
```

请记住，当你的程序完成时，必须取出所有的 print 调试语句。建议你将某个字符串插入这些语句（“ DEBUG ”），这样就可以在以后将它们全部找出来。通过输入到 STDERR 文件句柄，可以将你的正常输出与诊断信息分开。如果你将原义符号 _LINE_ 和 _FILE_ 纳入你的诊断信息，Perl 就能输出当前代码行和文件的名字。

可以试用的另一种方法是使用 Perl 调试程序，几乎可以将调试程序用于任何 Perl 程序。观察程序按步骤运行的情况，会给你以很大的启发。第 12 学时对 Perl 调试程序的使用方法作了详细的说明。

16.3.4 首先要靠自己来解决问题

如果你的程序语句完全正确，代码逻辑也没有问题，但是仍然无法得到你想要的结果，那么你应该寻求外界的帮助。首先可以通过 Perl 文档来寻找解决问题的办法。

正如我们在第1学时中介绍的那样，用户安装的每个Perl产品都配有一套完整的文档资料。如果是Perl 5.005版本，它的文档资料超过1700页。每个模块，每个函数，以及Perl语言的大多数特性，都在文档资料中作了介绍，并且在常见问题列表中也有相应的说明。

若要得到可用文档资料的清单，请在命令提示符处键入 perldoc perl。它列出了手册的每一节内容和对Perl的总的描述。

常见问题列表包含了初学者和专家们对Perl编程语言提出的最常见的问题。你至少应该对它进行一次浏览，以便对里面列出的各种问题有个基本的了解，即使你并不完全理解它的答案，也值得浏览一下。

如果因为某个原因，你的系统上没有安装Perl文档资料，或者perldoc并没有显示该文档，那么首先应该告诉你的系统管理员，来查找该文档。正确安装该文档是非常重要的，因为在在线文档与你运行的Perl版本是完全匹配的。其他任何文档很可能与此不同。

如果你无法访问在线文档，也可以通过网址 <http://www.perl.com>找到该文档。

16.3.5 从别人的程序错误中吸取教训

Usenet是个分布式传信系统，80年代初开发成功，并立即应用到方兴未艾的Internet上。Usenet分为成千上万个讨论组，涉及的问题从医疗、园艺、信息处理、科幻小说到曲棍球比赛和电动剃须刀，无所不包。并且还有许多地区性讨论组，世界上每个地区都有。下面是Perl的一些特定新闻组：

comp.lang.perl.announce	关于Perl的新版本、新模块和信息的新闻
comp.lang.perl.moderated	小信息量讨论组，对Perl进行适度讨论
comp.lang.perl.misc	大信息量讨论组，讨论与Perl相关的任何问题

若要读取Usenet的新闻，需要一个新闻阅读器。找到新闻阅读器并不难，可以访问任何一个软件下载站点，抓取一个新闻阅读器。可以访问若干个Web站点，比如deja.com或Supernews.com，它们镜像了Web格式的Usenet新闻组，只要求你拥有一个Web浏览器就可以阅读新闻。

在这些新闻组中，人们可以提出他们在使用Perl时遇到的问题，其他人则可以回答这些问题，并且全部是在自愿的基础上进行。另外，这里也可以讨论与Perl有关的人们普遍感兴趣的问题。

在我的整个编程生涯中，我认为在信息处理领域中不存在什么原始的问题。你所遇到的问题，其他人以前就遇到过。关键是要找到提出问题的人和他能提供什么答案。很可能至少有一个人提出的问题与你在某个新闻组中提出的问题非常相似。

deja.com维护了关于Usenet的许多情况的在线历史记录。使用它的搜索引擎，利用几个选择准确的关键字，就能够找到你的问题的答案。

例如，你想了解如何编写用于抓取Web页的Perl程序，可以访问deja.com站点的Power Search屏幕，然后将下列信息填入屏幕：

Keywords: **fetch web page**
Forum: **comp.lang.perl.misc**

在这个例子中，你可以将所有其他域置空。当搜索结果返回时（几乎会有100个匹配的搜索结果），大多数匹配的搜索结果均与你提的问题有关。记住，你在Usenet中阅读文章时应该

注意下列要点：

- 并非所有答案都正确。任何人都可以提出问题，任何人也可以回答问题。你可以阅读几个答案，自己确定哪些答案有道理。你从这些答案中得到的收获是各不相同的。
- 如果你无法确定某个答案是否正确，可以根据这个答案，自己来核实有关的信息。可以访问关于该问题的在线手册页，现在你知道何处可以查找在线手册了。
- [deja.com](#)对5年中的新闻进行了存档。它提供的答案在5年前是正确的，但现在的正确性也许要打折扣了。

16.3.6 请求他人的帮助

如果你已经查看了在线文档，参考书和 Usenet的历史信息，但是仍然没有找到问题的答案，那么就应该求助于其他人了。

请求他人的帮助应该是你最后采取的措施，当然不是你首先采取的措施。专家是回答问题的最佳人选。他们能够接受你提出的措词糟糕的问题，并且在某个时候为你的问题提供出色的解决方案。不过与我提到的所有其他资源不同的是，人不具备解决问题的无限能力。他们会感到疲倦，他们会有心情不好的日子，他们尤其会厌倦一次又一次回答同一个问题。

虽然你所问的这个人很可能知道问题的答案，但是应该记住，你要他回答你的问题必然要占用他的时间，并且是利用他的经验。在麻烦他人帮你解决问题之前，你有责任先从其他地方寻求解决问题的方法。

若要在Usenet上提出问题，必须使用新闻阅读器或者前面讲到的一种 Web新闻接口。在你提出问题时，请遵循下列原则：

1) 在你做其他事情之前，先要看一看新闻组是否拥有常见问题列表。Perl新闻组有一个这样的列表，它是随着Perl解释程序一道提供给你的。对于其他新闻组，请先搜索 [deja.com](#)，找出该组的常见问题的列表，然后再发布你的消息。

2) 应该将问题提供给正确的新闻组。一般的Perl语言问题应该提供给 comp.lang.perl.misc 新闻组。与CGI相关的编程问题应该在 comp.infosystems.www.authoring.cgi 上发布。通过该新闻组的常见问题列表，你就会知道是否在正确的地方发布了你的问题。

3) 为你发布的问题选择一个比较好的主题行。它应该很好地描述你要提的问题，避免毫无用处的文字（“帮帮忙”、“新问题”之类的名字都是多余的），既要有表义性，也要简明扼要。

4) 确保问题的主体包含下列元素：

- 说明你究竟想干什么（甚至应该说明为什么要这样做）。
- 说明到现在为止你已经做了哪些试验。
- 说明你遇到过哪些错误。

如果你发布了你的代码的出错消息或代码的引用，那么也应该发布足够的代码，使回答问题的人能够知道你的代码的运行情况。如果你打算处理数据，则应包含一些代码行作为例子。

问题的主体不应包括下列元素：

- 大的代码段。
- EXE等二进制文件或uuencoded编码的文件。
- MIME附件。相反，你可以将例子和代码纳入文本的主体中。

5) 务必发布一个有效的电子邮件地址，以备有人想要回答你的问题但又不想公开回答时使用。

6) 最重要的一点是：你应该非常有礼貌。你是寻求素不相识的人为你提供帮助。任何人都没有帮助你的义务。你应该多说“请”和“谢谢你”，并且不要使用不礼貌的评语。不要使用欺骗性手段来谋求他人的帮助，比如说“帮助一个可怜的小女孩编写她的 CGI程序”，或者说“我将为你提供一个免费的 Web页，如果你……”这些话语显得非常失礼，而且有些低声下气。

当你将文章发布在新闻组上之后，应该等待其他人提供解决方案。Usenet新闻要花费数天时间才能传遍全世界，人们不可能跟踪和阅读每一篇文章。你应该有耐心，在等待解决方案的同时，可以再次提出这个问题。无论你做什么，不要过早在 Usenet上再次提出这个问题。至少要等待两个星期之后，再提出这个问题。你应该改变提出问题的措词，使主题行更加清楚，然后再试一次。

对你的文章的反应可能立即出现（在几分钟内），也可能在一个月或更长时间之后出现。正如在前面讲过的那样，对你提出的问题的解决方案的质量差别很大。有些很有参考价值，有些可能是错的。有些回答很有礼貌，有些则非常粗鲁。按照网络礼仪，你应该感谢为你提供解决方案的人。如果有人过分热情，你不必在意。

16.4 其他资源

如果想了解Perl、Perl编程和Perl开发界的情况，可以查看下列辅助资源：

- Larry Wall、Tom Christiansen和Randal Schwartz撰写的《Programming Perl》一书。这本书被人们视为Perl程序员的圣经。当你学习了Perl的基本知识后，可以将这本书作为最佳参考书来使用。
- Tom Christiansen和Nothn Torkington撰写的《The Perl Cookbook》一书。这是涉及Perl的各种问题、代码举例、解决方案以及对数百个问题的评论的总汇，它采用食谱的书写格式。它提出了每个问题，每个问题的解决办法，然后是每种解决办法的代码举例和说明。
- Perl季刊。这份季刊自称是“Perl语言开发界之声”。这是一份真正的技术性刊物，里面的文章都是Perl语言开发界（每天使用Perl的程序开发人员）的成员撰写的，而不是由权威学者或专业撰稿人撰写的。它的创刊号就声称：“我们的目标是……使之成为一份知识性出版物，以探讨Perl的技巧，编程的技巧和其他的一些技巧……”

若要进一步深入了解这些问题，请查看下列信息：

Internet的发展历史：Hobbe的Internet Timeline，网址为：

<http://www.isoc.org/zakon/Internet/History/HIT.html>

Perl的发展历史：CPAST，网址为：

<http://history.perl.com>

Perl季刊，网址为：

<http://www.tpj.com>

CPAN，网址为：

<http://www.perl.com/CPAN>

在线文档：

在你的系统上，也可以查看 <http://www.Perl.com>。

Eric S. Raymond的开放源文章：

<http://www.netaxs.com/~esr/writings>

16.5 课时小结

本学时你学习了关于 Perl 的发展历史以及开放源开发模式如何用于 Perl，还学习了关于 CPAN的一些知识，它为什么会存在，以及谁负责对它进行维护。最后，学习了在开发 Perl 程序时遇到问题时可以使用何种类型的资源来加以解决。

16.6 课外作业

16.6.1 专家答疑

问题：如果说 Web 是在 Perl 之后问世的，那么为什么 Perl 是个 CGI 语言？

解答：Perl之所以属于CGI语言，其原因与计算机可以用来玩游戏是相同的。原因并不在于它们为了什么目的而发明，而在于 Perl非常适合作为一种 CGI语言。下一个学时我们将要详细介绍为什么 Perl 是个非常好的CGI语言。

问题：我在 Usenet 上提出了一个编程问题，但是却得到了一个粗鲁和令人恼火的答复。我应该怎么办？

解答：首先，这个使你恼火的答复是否包含某些好的建议。如果有，那么你应该采纳这些建议，不要在乎粗鲁无礼的一面。否则你可以对这个答复不予理睬。人生苦短，不要把生命浪费在无谓的纠纷上。

问题：有没有一种简便的方法可以用来搜索 CPAN？

解答：有的。位于 <http://search.cpan.org> 上的 Web 页包含一个通用搜索函数，你可以用来浏览 CPAN 最近的修改情况，并可按分类查看各个模块。

16.6.2 思考题

1) 关于用 Perl 进行 CGI 编程的问题首先应该传送给哪个 Usenet 组？

- a.comp.infosystems.www.authoring.cgi
- b.comp.lang.perl.misc

2) 如果你的系统没有这个文档，应该怎么办？

- a. 要求系统管理员安装该文档。
- b. 将消息发送到 comp.lang.perl.misc。
- c. 设法从辅助来源那里获取该文档，比如 <http://www.Perl.com>。

16.6.3 解答

1) 答案是 a。comp.lang.Perl.misc 可以作为你提出 CGI 问题的第二个地方。

2) 答案是 a 和 c。可能是先 a 后 c 这个顺序。文档可能已经安装，系统管理员可以帮你找到它。如果不行，www.Perl.com 有一组文档的最新拷贝。

China-pub.com

下载

China-pub.com

下载

第三部分

将Perl用于CGI

第17学时 CGI 概述

第18学时 基本窗体

第19学时 复杂窗体

第20学时 对HTTP 和CGI进行操作

第21学时 cookie

第22学时 使用CGI程序发送电子邮件

第23学时 服务器推送和访问次数计数器

第24学时 建立交互式Web站点

China-pub.com

下载

第17学时 CGI概述

毫无疑问，人们普遍认为 Internet的爆炸性流行主要是因为有了 World Wide Web。自从1993年第一个图形 Web浏览器的问世以来，Internet便以惊人的速度迅速发展，1993年前后Internet上的主机数量每20个月翻一番，而目前则每12个月翻一番。专用网络即 Intranet的增长速度甚至更快。

1993年以来，Web的内容已经变得越来越杂，Web用户希望每个Web页不只是能够显示静态（不变的）Web内容。成功的 Web站点必须显示动态 Web页，也就是能够提供最新信息的Web页。要使复杂的Web页能够跟上内容的迅速变化，这几乎是不可能的，因此出现了公用网关接口（CGI）。



为了学习后面7个学时的内容，你必须具备关于超文本标记语言（HTML）的某些知识。如果你对 HTML不熟悉，不必担心，它学习起来并不困难，也不需要通过本书来学习更多的这方面的内容。

HTML是一种标记语言，常用于创建 Web页。HTML由纯文本组成，其格式化代码嵌入文本之中，以指明 Web浏览器应该如何显示文本。例如，HTML is <I>not</I> hard to learn这句话是个普通文本，而<I></I>这些标记则不属于普通文本。它们称为标记，用于描述应该使用何种格式来显示文本。在上面这个例子中，Web浏览器应该用斜体字来显示单词 not。（请记住，并非所有浏览器都具有图形显示功能。）

关于HTML的详细说明，不属于本书要讲解的范围。介绍并不困难，但是有大量的资料需要加以说明。HTML的技术规范由World Wide Web集团（W3C）负责维护，该机构的网址是 <http://www.w3c.org>，可以通过该网址找到许多很好的教材。《HTML 24学时教程》是介绍HTML的一本好书。

在本学时中，你将要学习：

- Web是如何运行的。
- 在编写CGI之前你应该具备什么知识。
- 如何编写你的第一个CGI程序。

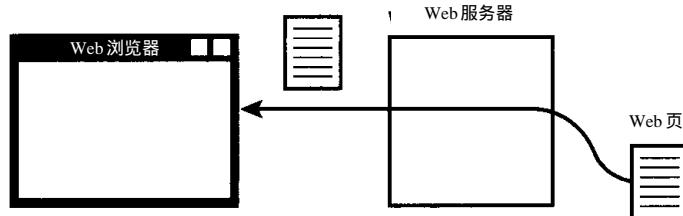
17.1 浏览Web

也许你已经知道，Web是指试图进行数据交换的两个不同系统之间进行的交互操作。试图抓取Web页的系统称为客户机系统。客户机系统通常运行一个称为 Web浏览器的程序，比如Netscape、Internet Explorer和Opera等，这是你习惯于日常使用 Web的应用范围。Web浏览器配有浏览按钮和书签，用于在屏幕上绘制 Web页。

在Web的另一端是称为 Web服务器的系统。该系统负责接收客户机查看 Web页的请求，从本地磁盘上检索 Web页，并将 Web页发送给客户机系统，即你的 Web浏览器。图17-1显示了这

种交互操作的情况。

图17-1 Web浏览器检索一个Web页



17.1.1 检索一个静态Web页

当客户机需要检索一个Web页时，它要查看统一资源定位器（URL），以确定使用什么协议、服务器，以及在该服务器上提出的是什么请求。典型的URL类似下面的形式：

`http://www.google.com:80/more.html`

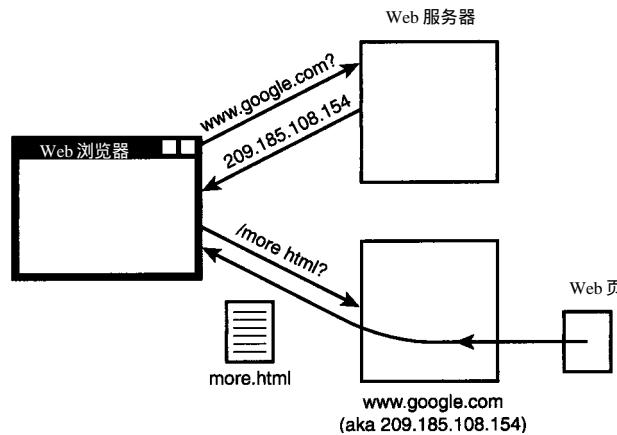
URL可以分割成下列部分：

- http 这个部分是指使用的协议。HTTP即超文本传输协议，它是传送Web页时使用的协议。你也会看到文件传输协议（ftp）或保密HTTP（https）等协议。
- www.google.com 这部分是服务器名，也称为主机名，它包含你想要的文档。有时，这部分不是主机名，而是个IP地址，通常写作4个数字，数字之间用圆点隔开，比如209.185.108.147。不过这些地址不如主机名那样可靠。
- : 80 这部分是个端口号，用于确定你的客户机与服务器是在哪个端口上互相进行连接。这部分通常是可有可无的。使用的协议决定了使用什么端口。http协议通常使用端口80。
- more.html 这部分是指对服务器提出的请求。通常这是你想检索的一个文档。有时它写作一个路径名，例如/archives/foo.html，也可以用其他字符作为结尾（？＆），不过它基本上指客户机要求向服务器检索的文档。

这时客户机为http执行下列操作步骤（见图17-2）：

- 1) 主机名（www.google.com）转换成IP地址。
- 2) 使用IP地址和端口号与www.google.com上的服务器建立连接。
- 3) 向服务器提出检索Web页more.html的请求。客户机等待服务器应答。
- 4) 服务器发出应答，在上例中，服务器发出more.html的内容，然后断开与服务器的连接。
- 5) 客户机在屏幕上显示服务器应答的Web内容。

图17-2 客户机向服务器提出检索Web页的请求



下载

客户机与服务器之间进行“通信”的详细情况将在第20学时中介绍。

17.1.2 动态Web页——使用CGI

在检索通常的Web页时，服务器只是根据你想要的文档并从它的磁盘存储器上检索这个文档，然后将它发送给客户机，如图17-3所示。

图17-3 检索静态Web的示意图

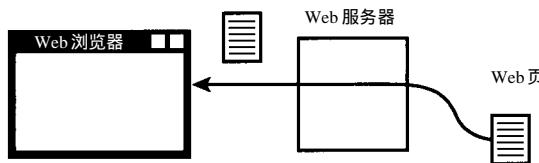
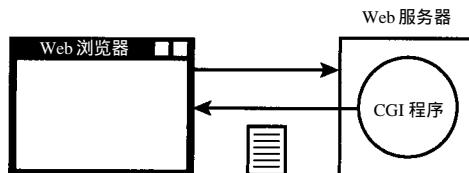


图17-3中的服务器根本不对数据进行任何处理，它只是查看客户机提出的请求，并将请求的数据传送给客户机。

在Web上创建动态内容的方法之一是使用CGI程序。CGI是Web服务器用来在服务器上运行程序以便生成Web内容的公认的方法。当URL指明CGI应该在它上面运行CGI程序来生成Web内容的服务器时，该服务器就启动该程序运行，该程序则生成Web内容，然后服务器将内容传递给客户机，如图17-4所示。

图17-4 CGI脚本生成的Web页



每当客户机请求检索一个实际上是CGI程序的Web页时，便出现下列操作：

- 1) 服务器启动CGI程序的一个新实例。
- 2) CGI程序使用它需要的信息生成一个Web页，或者生成另一个应答。
- 3) 该Web页被送回给客户机。
- 4) CGI程序退出。

CGI程序可以是任何类型的程序。它可以是个Perl脚本，就是你将要在这里学习的一项内容。它也可以是用C、UNIX shell，pascal，LISP、TCL或任何其他编程语言编写的程序。而许多CGI程序是用Perl编写的，这完全是一种巧合。Perl恰好非常适合编写用于文本处理的程序，而CGI程序的输出常常是文本。

CGI程序的输出几乎可以是以任何形式的信息。它可以是图形、HTML格式的文本、压缩文件、流式视频信息，或者你在Web上找到的任何其他类型的内容。总的来说，你编写的CGI程序将生成HTML格式的文本。



CGI不是一种语言，它与Perl之间并不存在特殊的关系，与HTML语言也没有任何关系，与HTTP之间也没有多少关系，它只是Web服务器与代表服务器运行的程序之间的一个公认接口。CGI的技术规范由美国国家超级计算机应用中心维护，该中心的网址是http://www.ncsa.uiuc.edu/cgi_interface.html。你可以在后面7个学时中了解CGI的详细特性。

17.2 不要跳过这一节内容

当你准备编写 CGI程序时，首先必须搞清几个问题，否则，第一次编写 CGI程序的经历一定不会使你感到愉快。你预先查找这些信息是比较容易的，而在调试程序时要搞清这些问题则要困难得多。

若要使用CGI，必须拥有Web服务器。CGI编程新手遇到的常见问题是他们试图在 Web服务器没有正确安装的情况下就编写 CGI程序。若要获取 Web服务器，可以从两种方法中选择一个。可以在商用 Web服务器上租用一定的空间，也可以运行自己的服务器。决定权在你的手里，可以根据你愿意支付的费用，需要的带宽，以及你的技术熟练程度来作出决定。

若要获得商用 Web服务器，你可以搜索 Web并查找一个。这些商用服务器常常称为“ Web托管”公司，它们收取的费用和提供的特性可以根据情况而千差万别。如果你打算编写 Perl CGI程序，应该确保Perl 5将作为CGI的编程语言。很少有 Web托管公司不支持Perl 5作为CGI编程语言或者根本不允许CGI编程的。应该避免使用不支持Perl 5的公司提供的服务，有许多其他公司可以供你选择。

还应该确保 Web托管公司允许你使用自己的脚本。有些公司声称它们允许使用 Perl CGI程序，但是接着又要求你使用它们公司的程序，有时要收取一定的费用。你也应该避免使用这些公司提供的服务。

另外还有一些公司，它们收取一定的费用来“审查”你的CGI程序，它之所以要收取费用，原因是你可以使用这些程序。如果你选择这些公司中的一个，应该安装你自己的服务器，以便进行相应的测试，因为“审查”是非常昂贵的。

如果你具备某些专门技能并愿意阅读全部说明的话，运行你自己的个人 Web服务器并不十分困难。首先，必须选择一个 Web服务器。如果你运行Windows，可以从几十种免费的或者接近免费的Web服务器中选择你要安装的服务器。一定要确保它们支持Perl作为CGI脚本语言。少数商用 Web服务器也可以用于Windows，比如Microsoft的Internet Information Server(IIS)。

如果你拥有一台 UNIX计算机，也可以使用少量商用 Web服务器。请与 UNIX供应商联系索取Web服务器清单。

Internet上最流行的 Web服务器是Apache，它是完全免费的。如果你拥有一个 C编译器，那么Apache Web服务器是很容易安装的。如果你习惯于编辑配置文件，那么它的运行也是非常容易的。Apache甚至可以用于 Miorosoft Windows平台。关于 Apache的信息，请访问 <http://www.apache.org>。

如果你运行自己的 Web服务器，在试图编写CGI程序之前，应该确保Web服务器运行正确，并且能够为静态 Web页提供服务。如果你的 Web服务器不能为静态 Web页提供服务，这说明你的CGI程序很可能无法运行。

还应该检查 Web服务器的配置，以确保你已将CGI脚本正确地激活。如果不激活这个特性，那么CGI编程的初学者就会遇到非常头痛的麻烦。

检验表

无论你是运行自己的 Web服务器，还是租用商用 Web服务器上的空间，都必须花费一点时间来完成下面这个检验表中的操作，确实如此。请将这些信息写下来，以后你就可以省去很多麻烦。

- 如果你是从商用 Web服务器主机上租用空间，那么该主机将为你提供所有的信息。这些信息可能位于主机的 Web站点上的FAQ中，也可能包含在你建立帐户时发送给你的文档中。如果你没有接收到这些信息，可以与 Web托管公司联系索取这些信息。如果你想使CGI程序正确运行，获取这些信息是非常重要的。
- 如果你已经配置和安装了自己的Web服务器，那么这些信息应该是配置进程的组成部分。如果你遇到了问题，请查看一下是否能够找到一个回答这些问题的FAQ，或者是否能够找到要检验的配置文件。

如果你要进行CGI编程，必须知道下列信息：

- Web服务器上Perl的位置 你必须知道Perl解释程序安装在Web服务器上的什么位置。由于你必须修改程序中的#！行代码以便反映该路径的情况，所以必须知道这个信息。如果你的Web托管公司运行Microsoft公司的操作系统，那么你可以不需要这个信息。
- Web服务器日志文件的位置 如果不知道Web服务器的错误日志保存在什么地方，就很难调试你的CGI脚本程序。你应该设法找到这个位置，这很重要。
- 用于CGI程序的扩展名 Web服务器有时要将服务器上保存的普通静态Web页与根据文件名来运行的CGI程序区分开来。CGI程序的扩展名通常是.cgi或.pl。有时则根本不用扩展名。
- CGI程序目录的位置 Web服务器有时需要CGI程序文件名的扩展名，有时则需要将文件放在一个专门的目录中。（很少同时需要这两者。）该目录通常称为 /cgi-bin，并且位于Web站点的顶层目录中（或者靠近顶层的目录中）。
- CGI目录的URL 许多情况下，你使用的Web服务器的URL中CGI目录附加在它的结尾处，例如：

http://www.myserver.com/cgi-或者http://www.myserver.com/cgi/.

17.3 编写你的第一个CGI程序

了解了上述关于CGI编程的有关说明、注意事项、检验表等信息后，你就可以准备键入你的第一个CGI程序了。程序清单17-1显示了这个程序。

键入这个程序并将它保存为hello。如果在检验表中必须将某个扩展名用于CGI程序（正如“用于CGI程序的扩展名”项中所说的那样），那么请使用该扩展名。这样，如果必须使你的CGI程序的文件名带有.cgi扩展名，那么将该脚本程序保存为hello.cgi。如果必须使用扩展名.pl，请将该脚本程序保存为hello.pl。

程序清单17-1 你的第一个CGI程序

```
1:  #!/usr/bin/perl -w
2:  use CGI qw(:standard);
3:  use strict;
4:
5:  print header;
6:  print "<B>Hello, World!</B>";
```

第1行：这一行是个标准#！行。你必须替换检验表中的“Web服务器上的Perl位置”信息项中的路径，使该脚本程序能够运行。当然-w用于激活警告特性。

第2行：CGI模块纳入了该程序。qw（:standard）使得一组标准函数从CGI输入到你的程

序中。

第3行：use strict是个很好的编程命令，对于CGI程序来说也是一样。

第5行：从CGI模块中输入header函数。它输出一个标准标题，服务器（和客户机）必须看到它后才能处理CGI程序的输出。

第6行：当标题输出后，所有输出就会正常显示在浏览器中。在本例中，当 CGI运行时，浏览器将显示Hello world。

这就是各个代码行的具体内容。

不过，事情并没有结束，你还必须安装这个CGI程序并对它进行测试，你的工作只完成了半。

17.3.1 在服务器上安装CGI程序

究竟如何安装CGI程序，主要取决于你拥有何种服务器，你是否能够在本地访问它，或者是否只能用FTP将文件发送给该服务器。下列各节将介绍如何为不同的环境安装CGI程序。

1. 本地访问UNIX Web服务器上的文件系统

如果你能够使用telnet、rlogin或其他方法登录到UNIX Web服务器上去，请使用下列说明来安装CGI程序：

1) 使用FTP，将CGI程序hello.cgi（或hello.pl）放在UNIX服务器上。也可以使用vi将该程序写入服务器，这也是个好方法。

2) 使用mv或cp命令，将CGI程序转到正确的目录中。你应该在“CGI程序目录的位置”下的检验表中找到正确的目录。

3) 在UNIX下，必须使该程序成为可执行程序。可以使用下面这个命令来执行这项操作：

```
chmod 755 hello.cgi
```

如果该程序的名字是hello.pl，则在该命令中使用该名字。该命令使得文件所有者能够写入该文件，而其他人则可读取和执行该文件（对于CGI程序来说，这是正确的）。

2. 只能用FTP来访问UNIX Web服务器

如果你只能使用FTP来访问服务器，请按下列说明来安装CGI程序：

1) 使用你的FTP客户程序将hello.cgi（或hello.pl）程序转入CGI程序目录。你应该已经在“CGI程序目录的位置”下的检验表中找到正确的目录。务必以文本方式或 ASCII方式来传送文件，不要使用二进制方式将CGI程序传送到服务器中。如果使用文本方式的FTP实用程序，那么它的默认方式通常是文本方式。

2) 必须使CGI程序成为可执行程序。对于仅为文本的CGI程序，下面这个命令应能运行：

```
quote site chmod 755 hello.cgi
```

如果hello.pl是该程序的名字，则上述命令应该使用hello.pl。该命令使得该文件可供文件所有者写入，而其他所有人则可以读取和执行该文件（对于CGI程序来说，这是正确的）。

3) 如果你拥有一个图形FTP程序（如Cute-FTP），必须找到Set Permissions（设置访问许可权），Change Mode（改变方式），Set File Attributes（设置文件属性），或Set File Access Mode（设置文件访问方式）等选项卡，以便设置访问许可权。

不管用何种方法设置访问许可权，文件所有者需要读/写/执行权限，用户组需要读/执行权限，其他用户需要读/执行权限。如果该程序需要数字式访问权限，请使用755。

下载

3. 本地访问NT Web服务器上的文件系统

如果你能够本地访问NT Web服务器的文件系统，请使用NT的Explorer或文件拷贝实用程序将CGI程序放入正确的目录，即“CGI程序目录的位置”中指定的这个目录。

4. 只能使用FTP来访问NT Web服务器

如果你只能使用FTP来访问NT Web服务器，请使用FTP客户程序将hello.cgi（或hello.pl）程序放入CGI程序目录。你应该已经找到“CGI程序目录的位置”下的检验表中的正确目录。务必用文本方式或ASCII方式来传送文件，不要使用二进制方式将CGI程序传送到服务器。如果使用文本方式的FTP实用程序，其默认方式通常是文本方式。

17.3.2 运行你的CGI程序

若要了解你的CGI程序是否能够运行，请打开浏览器，并将它指向你在检验表中设定的地址，即CGI目录的URL，并将CGI程序名附加在该URL的后面。例如，可以输入下面的URL：

`http://www.myserver.com/cgi-bin/hello.pl`

当保存CGI程序时，应该使用hello.cgi或你用于CGI程序的任何名字。

这时会发生下列两种情况中的一种：

- 1) 你的浏览器加载一个带有Hello, world消息的Web页。
- 2) 它没有加载这个Web页。

如果你的CGI程序不能运行，不管原因是什么，请查看下一节的说明。下一节专门介绍如何查找类似这样的程序问题。CGI程序的安装和调试过程非常困难，而且的确很难。不过你不要灰心，因为CGI程序的运行不会是一帆风顺的，你应该坚定信心。一旦排除了CGI程序的故障，你将不必重新对它进行调试。

如果你的CGI程序能够按照要求来运行，那就太好了。这说明你已经成功地安装了你的Web服务器和CGI程序，并且使它们能够正确运行。不过你仍然应该浏览下一节的内容。总有一天，你的某个CGI程序可能发生故障，你至少应该熟悉诊断程序故障的操作步骤。

17.4 CGI程序无法运行时怎么办

下面几节为你提供一个CGI程序的通用调试指南。在你阅读所有这些内容以便找出你的第一个CGI程序中的问题之前，请回头看一看前面的内容，以确保没有跳过任何步骤。当你学到本学时结尾的时候，应能发现你的CGI程序中存在的任何问题。

这几节中介绍的诊断操作均假设你要调试的CGI程序名是hello.cgi。如果你的程序使用别的名字，请改过来。

17.4.1 这是你的CGI程序吗

- 第一个需要解决的产生问题的原因是CGI程序本身。如果CGI程序不能运行，那么调试Web服务器的配置是毫无意义的。

CGI程序可以像所有Perl程序那样以交互方式来运行，用交互方式来运行CGI程序对于程序的调试来说是非常有用的。若要运行你的CGI程序，请在命令提示符处输入下面这个命令，将它启动：

```
perl hello.cgi
```

这时Perl解释程序应该输出下面这行信息作为应答：

```
(offline mode: enter name=value pairs on standard input)
```

这个提示行表示CGI模块试图获取你的CGI窗体的值。这些值将在第18学时中介绍。

看到这个提示后，你应该输入文件结束字符作为应答。在UNIX下，它是Ctrl+D，你只需按下Ctrl键并键入D。在Windows中，可以按下Ctrl+Z。然后Perl应该输出下面这两行消息：

```
Content-Type: text/html
```

```
<B>Hello, World!</B>
```

Content-Type : text/html这条消息表示后面的信息应该转换为文本或HTML。这条消息的含义将在第20学时中全面介绍。现在，你只需要知道重要的是这条消息是你的程序用header函数输出的“第一个”信息，并且这条消息是必须输出的。如果在Content-Type消息之前输出了别的什么消息，那么CGI程序的运行将会失败。

问题：Perl应答的语句有误。

解决办法：找出语句错误。

问题：Perl应答的信息是Can't locate CGI.pm in @INC...(在@INC中无法找到CGI.pm...)。

解决办法：你安装的Perl不完整。Perl配有默认的CGI模块。如果你想要安装它，请参阅本书的附录。

17.4.2 服务器存在的问题

当排除了你的脚本程序是问题的根源之后，就应该检查脚本程序的安装和服务器的配置是否正确。

问题：服务器应答的消息是Not Found(未找到)或404 Nat Found(404未找到)。

解决办法：这些消息通常表示存在下列问题之一：

- 你使用的URL不正确。当你应该键入http://www.server.com/cgi-bin/hello.cgi时，你却键入http://www.server.com/cgi/hello.cgi。请返回到检验表，核实你的CGI目录的URL是否正确。
- 你将脚本程序放入Web服务器上的目录不正确。请核实用检验表，确定CGI程序的目录是否正确。如果不正确，请将脚本程序转到正确目录中。

问题：你的脚本程序的文本显示出来了。

解决办法：之所以显示该程序，原因是Web服务器认为该程序实际上是个文档。

- 你使用的CGI程序扩展名错了。你没有使用.pl，而是使用了.cgi或者其他错误的扩展名。请查看检验表，确保你使用了正确的CGI程序扩展名。
- 你将脚本程序放入了不正确的目录中了，同时使用了错误的URL来访问它。请将脚本程序放入正确的CGI程序目录中，并且确保你使用的URL是正确的。
- 服务器配置有误。如果你是使用自己的Web服务器，请重新阅读它的文档，并核实你的Web服务器安装是否正确。有时安装服务器时包括了一个测试用的CGI脚本程序。如果是这样，请测试这个CGI脚本程序。如果你使用一个商用Web托管服务器，请核实你将脚本程序放入了正确的目录之中，否则与Web主机联系，请求其帮助。

问题：服务器应答的信息是Forbidden(禁止)或403 Error(403错误)。

解决办法：对CGI程序的访问权限设置不正确。这个问题最有可能出现在 UNIX Web服务器上。

可以查看对hello.cgi程序的访问权限，方法是在命令提示符处键入 ls -l hello.cgi。如果你拥有对服务器的FTP访问权，可以查看文件访问权，方法是键入 dir。该访问权限应该类似下面的形式：

```
-rwxr-xr-x    1 user          93 Aug 03 23:06 hello.cgi
```

访问权限是左边的字符 rwxr-xr-x。如果不是这样，请回到安装说明，详细了解如何正确地设置对CGI程序的访问权限。

17.4.3 排除服务器内部错误或500错误

如果服务器应答的消息是 Internal Server Error (服务器内部错误) 或 500 Error (500 错误)，这意味着你的CGI程序运行失败了。这个通用故障消息是由许多不同问题产生的。

检查“ Internal Server Error ”时使用的最重要工具是服务器的日志文件。当 Web 服务器收到客户机要检索 Web 页的请求时，它就会将每个请求写入一个文件，供以后分析时使用。服务器遇到的任何错误也会记录在这个文件中，包括 CGI 程序生成的出错消息。

请查找服务器的出错日志文件的位置，你在检验表中应该看到了这个文件位置。日志文件的编写通常是将新的项目附加在日志文件的底部。若要查看 UNIX 下的最后几个日志文件项目，请在提示符后面键入下面这个命令以便查看日志文件的底部的项目：

```
tail server_log
```

有些Web服务器配有一个实用程序，它常常是 CGI 程序本身，用于查看日志文件。如果你只拥有对服务器的FTP访问权，那么必须下载该日志文件，并在你的本地 PC 上查看该日志文件，以便找出错误项。

如果你无权访问服务器的错误日志文件，那么就存在一个很大的隐患。查找“ Internal Server Error ”将是一件漫无边际的工作。按照下面显示的检验表，最终你应该能够找到存在的问题。(你在服务器的日志文件中找到的消息是不精确的信息，不同的服务器的消息文本各不相同。)

日志项：No such file or directory : exec of /cgi-bin/hello.cgi failed (没有这个文件或目录：/cgi-bin/hello.cgi运行失败)

出错的原因：

- 脚本程序的 # ! 行可能不正确。应该确保 # ! 行中 Perl 的位置与检验表中 Web 服务器上的 Perl 位置相一致。通过使用 FTP 中或本地的 ls 或 dir 命令，核实 Perl 实际上已经安装在该位置上了。
- 如果你使用 FTP 将 CGI 程序传送到服务器，可能没有使用 ASCII 方式进行传输。用二进制方式将 Windows 中编写的脚本程序转移到 UNIX 服务器（并反方向传送），这是行不通的。
- 对 CGI 程序的访问权设置不正确（ UNIX 下）。请在“ 服务器存在的问题 ” 这一节中查看关于 Forbidden 的说明。

日志项：Can 't locate CGI.pm in @INC...(@inc中找不到CGI.pm)。

出错的原因：

- 安装的Perl不完整，受到了破坏，或者太旧了。显然Perl无法找到CGI模块，CGI模块是Perl的标准组成部分。你必须重新安装该模块，或者与系统管理员联系，请他重新安装Perl。安装方法请参见本书附录。

日志项：Syntax error, warning, Global symbol requires, etc(语句错误、警告、需要全局符号等)。

出错的原因：

- 你的Perl程序显然存在键入错误或语句不正确的问题。请按“这是你的CGI程序吗”这一节中的说明，确定问题的性质。

日志项：Premature end of script headers (脚本程序标题过早结束)

出错的原因：这个出错消息说明了这样一种情况，即你的脚本程序在运行，而CGI模块的header函数输出的Content-Type标题并不是脚本程序发出的第一个消息。有时在日志文件中的这个消息前面或后面还会出现一条辅助消息。这个辅助消息更有助于确定出错的原因。你可以试用下面的方法来确定出错的原因：

- 在调用header函数前，务必不要输出任何信息，包括出错消息。在header函数之前输出的任何东西都会导致这个错误。



在程序的开始处而不是在调用header函数时，你会看到Perl CGI程序输出“Content-Type : text/html\n\n”这条消息。显然输出这个消息和调用header函数被认为是做同样的事情，但实际并非如此。header函数要考虑这样一个问题，即\n\n在每个服务器上并不总是表示相同的意思，它会为该服务器输出相应的序列。

- 一个称为输出缓冲的问题会导致system函数在header函数输出之前产生输出，并在输出中出现反引号(`)。若要确保header函数的输出总是显示在前面，可以将Perl CGI程序的开始部分重新编写为下面的形式：

```
#!/usr/bin/perl -wT
use strict;
use CGI;

$|=1;      # ensures that header's output always prints first
print header;
```

17.5 课时小结

在本学时中，我们介绍了CGI程序如何运行的基本知识，讲述了静态Web页与动态Web内容之间差别，并且在后面几个学时中还要进一步阐明这些问题。你还编写了第一个CGI程序并且使它运行了起来。

此外我们还提供了如何调试CGI程序的指南，这对今后几个学时的学习来说是非常有用的。

17.6 课外作业

17.6.1 专家答疑

问题：我没有加载CGI模块，是否必须使用该模块？

下载

解答：坦率地说，你确实必须使用该模块。CGI并不是个很容易使之正常工作的模块。目前已经发布的许多程序试图仿制CGI模块的功能，可惜都不太成功。它们存在着大量的安全漏洞，并且无法实现兼容。此外，它们不符合Internet标准。在第16学时中，我们讲述了为何“仿制车轮”是件并不高明的事情。CGI是个很难仿制的车轮，我们都无法第一次或者第100次使之恢复正常。

标准产品中包含的Perl CGI模块已经被成千上万个程序开发人员测试过，非常耐用，你可以放心地使用它。

本书附录中讲到，如果必要的话，你可以安装只供你自己使用的CGI模块。你没有理由不使用这个模块。本书中的所有代码例子都需要使用CGI模块，有关说明的前提是你已经安装了该模块。

问题：我拥有cgi-lib.pl的拷贝，可以使用它吗？

解答：你不应使用它。cgi-lib.pl的所有函数都在CGI模块中。cgi-lib.pl库非常老，并且得不到维护。

问题：为什么人人都必须将Perl用于CGI？为什么不使用C或TCL？

解答：Perl的特性对CGI特别有用。这些特性主要包括：

- Perl具有非常出色的文本处理功能。
- 你很快就会了解的Perl的出色功能将使它成为编写CGI程序的安全语言。
- Perl是一种优秀的“胶水”语言，它非常适合将操作系统的实用程序、数据库和CGI等不同技术组合在一起。
- Perl很容易使用。

问题：如果我遇到了关于Perl和CGI方面的问题，是否应该将一条消息发送给comp.lang.perl.misc新闻组？

解答：也许不合适。更合适的新闻组是comp.infosystems.www.authoring.cgi。不过首先你应该查看<http://www.w3.org/CGI/>网址上的FAQ。

17.6.2 思考题

- 1) CGI程序可以用下面的语言编写：
 - a. 只能用Perl、UNIX Shell或C语言。
 - b. 只能用C语言。
 - c. 能够在服务器上运行的所有编程语言。
- 2) Web是在Perl之前问世的。
 - a. 是。
 - b. 否。

17.6.3 解答

- 1) 答案是c。Perl并不是编写CGI程序时使用的惟一语言，它的某些特性在编写CGI程序时更加容易并且更加可靠。
- 2) 答案是b。Perl是在1987年开发而成的，而Web直到1991年才在CERN问世。

17.6.4 实习

对“Hello, world!”程序稍作修改，增加一些功能。使用localtime输出当前时间，并且用HTML标记给它增添某些颜色和一两个表格，要有创意。请记住，在你的Perl程序中输出HTML，可以在加载Web页时使该程序出现在最终的Web页中。

China-pub.com

下载

China-pub.com

下载

第18学时 基本窗体

当你浏览 Web 时，肯定会填写几个 HTML 窗体。HTML 窗体，比如电子邮件 Web 窗体、购物窗体、宾客留言簿、在线拍卖窗体、邮件列表和订单窗体等，可以用于搜集信息，如 Web 浏览器用户的登录信息和 Web 站点的首选设置等。

当用户点击这些窗体上的 Submit (提交) 按钮时，将会出现什么情况呢？几乎在所有情况下，该窗体的数据都会传递给一个 CGI 程序。在本学时中，我们将要介绍如何从窗体中取出数据，如何在你的 CGI 程序中对数据进行操作。

在本学时中，你将要学习：

- 如何处理 Perl CGI 程序中的基本窗体。
- 如何调试 CGI 窗体。
- 如何编写更加安全的 CGI 程序。

18.1 窗体是如何运行的

你肯定使用过 Web 上的窗体，甚至知道窗体是如何布局的，以及它们是如何运行的。但是，如果要确保你处理的是同一个 Web 页，那么就必须了解 HTML 窗体的基本知识。

18.1.1 HTML 窗体元素概述

在你开始学习窗体如何运行之前，首先应该了解 HTML 是如何展示窗体的，以及窗体中的所有元素起着什么样的作用。



本书中介绍的 HTML 不应该被视为最典型的 HTML。本书中讲述的 HTML 足以展示必要的 CGI 特性，但是没有涉及到太多其他的东西。在本书的代码例子中展示的 HTML 中没有使用 <HEAD> 或 <BODY> 标记，也没有使用 <DOCTYPE> 标记。此外，屏幕都非常简单朴素，你可以添加自己的 HTML，使屏幕更加生动和完整。

HTML 窗体是 HTML 文档的一个部分，用于接收用户的输入。当浏览器加载包含窗体的 HTML 文档时，各个不同的 HTML 标号便在 Web 页上建立各个用户输入区域。用户的输入被放入各个窗体元素中，比如复选框、单选按钮、选项菜单和文本输入项元素。当用户使用 Web 浏览器对输入元素的操作完成后，窗体通常被提交给 CGI 程序，以便进行处理。

程序清单 18-1 显示了一个创建的典型 HTML 窗体。

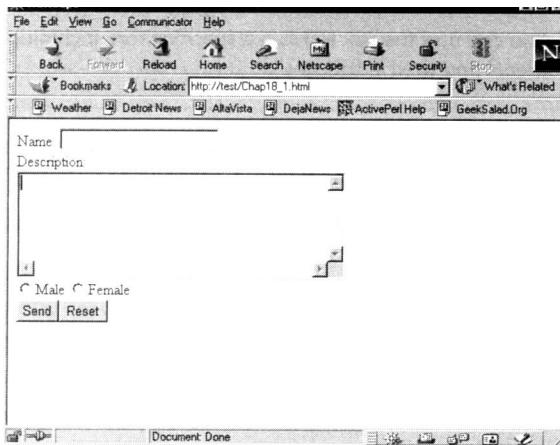
程序清单 18-1 一个小型 HTML 窗体

```
1: <FORM action="http://www.server.com/cgi-bin/submit.cgi" method="get">
2: <INPUT TYPE="text" name="name">
3: <TEXTAREA name="description" rows=5 cols=40>
4: </TEXTAREA>
5: <INPUT type="radio" name="sex" value="male">Male
6: <INPUT type="radio" name="sex" value="female">Female
```

```
7: <BR>
8: <INPUT type="submit" value="Send"><INPUT type="reset">
9: </FORM>
```

图18-1给出了：Netscape浏览器中显示的程序清单18-1的窗体。

图18-1 Netscape中显示的
程序清单18-1的窗
体



<FORM>标记用于设定完整的HTML文档中的窗体的开始。method属性用于设定该窗体是使用GET还是POST方法来提交窗体。如果这个属性没有设定，浏览器将使用GET方法将窗体提交给CGI程序。GET与POST方法的差异将在后面说明。action属性用于设定接收窗体数据的CGI程序的URL。

<INPUT>标记用于为用户提供一个输入域，在这里，它是一个空白文本框。该文本框被赋予一个名字，它恰好是“name”。

<TEXTAREA>标记用于使浏览器显示一个多行文本框，以便接收输入的数据。值得注意的重要属性是name属性，在这里，该域的名字是description。HTML窗体中的每个元素都必须拥有不同的name属性。当CGI程序被赋予该窗体以便进行处理时，name属性用来区分各个域。

“每个属性都有它自己的名字”这一原则有一个例外，那就是单选按钮。单选按钮按小组放在一起。单选按钮组中每次只能选定一个按钮。每个单选按钮组都有它自己的name属性。

最后显示submit(提交)按钮。当用户单击该按钮时，窗体的值就被传递给CGI程序，以便进行处理，这将在下一节中介绍。

HTML 4.0的技术规范包含了很多的不同窗体元素类型，因此，我们不想在本书中将它们全部完整地加以介绍。例如，许多窗体元素包含了一些属性，以便安装窗体元素的某些特性，比如前面介绍的窗体中的TEXTAREA中的rows和cols。在本书的其他学时中，每当使用HTML窗体元素时，都只使用最基本的属性。



在网址<http://www.w3c.org>上，你可以找到HTML 4.0完整的技术规范，包括有效的窗体及其属性。

18.1.2 单击submit时出现的情况

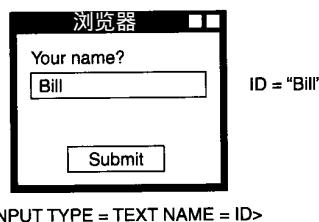
当用户在他的Web浏览器上填写窗体信息时，将会发生一连串的事件：

- Web浏览器接收窗体上的数据，放入名字与值对中（见图 18-2）。例如，在这个示例窗体中，名字为body的域接收了文本输入域的值。名字为sex的域将接收单选按钮的值。

下载

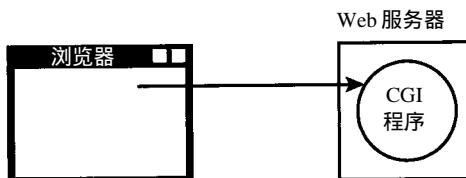
Web浏览器在发生任何情况之前执行所有这些操作。

图18-2 浏览器对数据与域的名字进行匹配



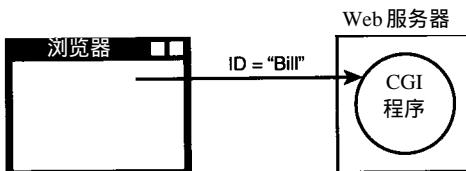
- 对窗体域的action部分的URL进行访问。这是CGI程序的URL（见图18-3）。

图18-3 浏览器与服务器进行联系



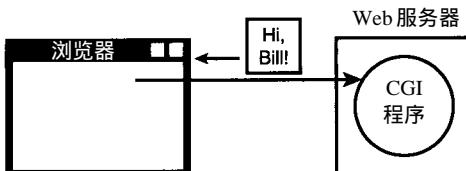
- 使用CGI方法GET或POST之一，窗体上的域的名字和值被传送到CGI程序（见图18-4）。你不必过分担心这个传输机制。

图18-4 数据被送往服务器



- CGI程序接收这些值，并生成一个应答，并将应答送回给浏览器（见图18-5）。该应答可以是个HTML页，或者包含另一个窗体的HTML页，也可以是转到另一个URL的HTML或者是CGI程序能够生成的任何其他东西。

图18-5 Web服务器的CGI程序作出的应答



18.2 将信息传递给你的CGI程序

当由于窗体的提交而使CGI程序运行时，从窗体传递过来的域的名字和值（称为参数）必须由CGI程序来处理。这是使用param函数来处理的。

如果没有任何参数，那么param函数返回传递到CGI程序中的域的名字。如果CGI程序接收到程序清单18-1中的窗体，param函数将返回body、sex、name和submit。

如果带有参数，param函数返回该参数的值。例如，param('sex')将根据你的选择，返回单选按钮的值male或female。

程序清单18-2包含了用于输出这些参数的简短的CGI程序。

程序清单18-2 用于输出参数的CGI程序

```
1: #!/usr/bin/perl -w
```

```

2: use strict;
3: use CGI qw(:standard);
4:
5: print header;
6: print "The name was", param('name'), "<BR>";
7: print "The sex selected: ", param('sex'), "<BR>";
8: print "The description was:<BR>", param('description'),
9:      "<P> ";

```

如果param函数指定的参数没有用于该窗体，则param返回undef。

GET与POST方法

在程序清单18-1中的窗体内，<FORM>标号拥有一个属性，称为method。Method属性用于设定Web浏览器应该如何将数据传送到Web服务器。目前可以使用的方法有两个。

第一个方法称为GET，如果你在<FORM>标号中没有设定方法，那么这就是默认的方法。使用GET方法，通过在URL中对窗体值进行编码，就可以将这些值传递给CGI程序。当你在Web上冲浪时，可能看到下面这样的URL：

```
http://www.server.com/cgi-bin/sample.pl?name=foo&desc=Basic%20Forms
```

CGI程序运行时，能够将URL的剩余部分解码，使之分解成域和值。当你调用param函数时，它实际上也是进行这样的操作。你不应该试图自己对这些值进行解码。param函数能够全面地进行这项操作，你没有理由使用别的方法来提取这些值。

另一个方法是POST，它产生的结果完全相同，但使用的手段不同。不是将所有的窗体值编码后放入URL，而是通过访问Web服务器，然后将HTML窗体值传送给CGI程序，作为其输入数据。另外，现在你不必清楚地了解这个过程究竟是如何运行的，CGI模块会为你处理好这个问题。只要调用param函数，就可以读取这些值，对它们解码，然后将它们传递给你的程序。



你可能已经从Internet下载了一些CGI程序，或者在别的书中见过一些例子，通过对环境变量QUERY_STRING进行解码，或者使用变量REQUEST_METHOD来确定窗体是使用GET还是POST方法。这些程序试图重复进行在标准CGI模块中已经做的工作，但是也可能没有这样做。你应该避免自己执行这项操作。

那么究竟你应该选择哪一种方法呢？每种方法都有它的优点和缺点。GET方法使得Web浏览器能给生成Web页的特定URL做上书签。例如下面的URL

```
http://www.server.com/cgi-bin/sample.pl?name=foo&desc=Basic%20Forms
```

可以做上书签，并且总是由浏览器返回。从CGI程序sample.pl的角度来看，它不知道你刚刚是否查看了该窗体。它像往常一样接收通常的CGI参数。如果能够使用GET方法的URL编码值来反复调用一个CGI程序，这称为幂等性。

但是你可能不是特别想使浏览器能在你的站点中做上书签，以便直接运行你的CGI程序，坦率地说，用于以GET方法启动CGI程序的URL是很讨厌的。

POST方法根本不对URL中的窗体数据进行编码，当它为Web页进行处理时，它依靠浏览器发送数据。但是，由于数据并没有被编码后放入URL，因此你无法使用POST方法给CGI程序生成的Web页做上书签。

18.3 Web安全性

在你将CGI程序放到World Wide Web上去之前，必须了解下面几个问题。通过将CGI程序放在Web页上，你就为远程用户（使用Web浏览器）赋予对你的系统的有限访问权。使用普通HTML文档，他们只能从你的Web站点检索静态文档。但是，使用CGI程序，他们就能在你的Web服务器上运行程序。

懂得如何编写安全而保密的CGI程序后，你与你的Web服务器管理员一定会感到更加高兴。编写这样的程序并不难，只需要掌握几条简单的注意事项。

18.3.1 建立传输明码文本的连接

当Web浏览器从Web服务器中检索Web页时，HTML是通过一个明码文本信道来发送的（见图18-6）。这意味着当数据一路通过Internet时，它并不进行加密、编码，否则就无法被对方理解。

图18-6 明码文本被传送到

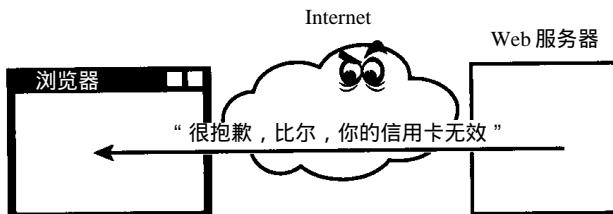
服务器



用户填入窗体然后提交给你的CGI程序的数据，传输时所用的协议与初始Web页使用的协议相同。任何人只要访问窗体，就可以查看它的所有域（见图18-7）。

图18-7 服务器用明码文本

作出应答



用明码传送数据时存在的问题确实是你应该担心的问题。Internet不是一个安全的地方，在Web浏览器与Web服务器之间的线路上的任何人都能够窃听线路上来回传送的信息。

应该记住，决不应该以普通CGI窗体来发送下列几种类型的数据：

- 任何形式的口令。
- 个人信息（社会保险号，电话号码）。
- 财务信息（帐号，个人身份识别号，信用卡号码）。

请记住这些基本原则。决不要在Internet上发送你不会写在明信片上任何信息。



你会说：“等一等，我曾经在Internet上看到一些窗体，要求查询所有上述信息，并且说这是安全的。”使用某些辅助工具，可以在Web上执行相当安全的事务处理。若要执行安全的Web事务处理，实际上必须对浏览器/服务器之间的全部会话进行加密。这是通过运用http协议的安全版本https来实现的。

18.3.2 注意不安全数据

在编写安全的CGI程序时需要考虑的另一个问题是：你编写的程序将根据Web页提供给你的输入来执行Perl命令。Internet或者你的Intranet（专用网）上有许多人属于不良之徒，他们以损害你的Web服务器为乐，并因此而感到自己了不起。还有一些并无恶意的用户可能不小心将无效数据发送给你的CGI程序。

请看程序清单18-3中的HTML窗体和程序清单18-4中的CGI程序。

程序清单18-3 目录清单Web窗体

```

1: <FORM action="/cgi-bin/directory.cgi">
2: What directory to list?
3: <INPUT TYPE=text NAME=dirname>
4: <INPUT TYPE=submit name=submit value="Run This">
5: </FORM>
```

程序清单18-4 名字为directory.cgi的不安全CGI程序

```

1: #!/usr/bin/perl -w
2: # Do NOT use this CGI program, it's very insecure
3: use strict;
4: use CGI qw(:all);
5:
6: print header;
7: my $directory=param('dirname');
8: print `ls -l $directory`; # Do a directory listing
```

程序清单18-3为用户提供了一个简短的窗体，用于接收一个目录名，再将它传送给称为directory.cgi的CGI程序。在程序清单18-3中，directory.cgi程序接收该目录，并为DOS/windows用户对它执行ls -l命令，它与dir等价，为用户提供一个目录列表。

这种类型的程序使得远程Web冲浪者能查看你的整个目录结构。CGI程序并不检查该目录名是什么，如果浏览器想要查看你的敏感数据，它就可以查看。

更重要的一个问题：\$directory可能根本不包含任何目录。如果Web浏览器为dirname发回了值/home；cat /etc/passwd，这时，CGI程序运行的命令将类似下面的形式：

```
ls -l /home; cat /etc/passwd
```

这个命令将能有效地将系统的口令文件拷贝发回给Web浏览器。实际上，所有UNIX shell命令或MS-DOS命令都可以这样运行。如果你的Web服务器尚未正确地安装，那么任何用户都可以上Internet。

Perl拥有一个机制，可以帮助你避免做这样的傻事。#！行上的-T开关可以激活数据受感染特性。当数据从外部信息源（如文件句柄、网络套接字、命令行等）接收过来时，它就被做上“tainted(受感染)”标记。受感染的数据不能用在反引号、系统函数调用（如open函数）、系统命令或可能破坏安全性的其他地方。

当受感染检查正在进行时，不能将open函数、system函或反引号用于你的Perl程序，除非首先明确设置PATH环境变量。

程序清单18-5显示了这个程序的更加安全的版本。

程序清单18-5 directory.cgi程序的更安全版本

```
1: #!/usr/bin/perl -wT
```

```
2: # tainting is enabled!
3: use strict;
4: use CGI qw(:all);
5:
6: print header;
7: # Explicitly set the path to something reasonable
8: $ENV{PATH}='/bin:/usr/bin';
9: my $dir=param('dirname');
10: # Only allow directory listings under /home/projects
11: if ($dir=~m,^(/home/projects/[\w/]+)$, ) {
12:     $dir=$1; # This "untaints" the data, see "perldoc perlsec"
13:     print 'ls -l $dir';
14: }
```



若要了解关于受感染的数据、如何消除数据的感染以及如何编写安全的 Perl程序的详细信息，请参见 Perl产品包含的 Perlsec手册页。

18.3.3 从事无法执行的操作

HTML / CGI窗体也可能遭到另一种情况的损害。请看程序清单 18-6中的HTML窗体。

程序清单 18-6 一个简单的窗体

```
1: <FORM action="/cgi-bin/doit.cgi">
2: Please type in your favorite color:
3: <INPUT TYPE=text length=15 name=color>
4: <INPUT TYPE=submit value="Submit color">
5: </FORM>
```

在这个窗体中，color域允许的最大宽度是 15。这对吗？大概差不多。HTML技术规范规定，文本域的length最多允许这么多的字符。但是，浏览器可能发生故障，有人可能故意在这个域中放入 15个以上的字符，方法是不使用你的窗体，或者创建一个新窗体。

如果你希望某个域拥有一个特定值，请不要依赖 HTML、Java或JavaScript来保证这个值的正确性。例如，如果 color域的绝对限值应该是 15，那么 Perl程序可以像下面这样对它进行处理：

```
my $color=param('color');      # Get the original field value
$color=substr($color, 0, 15);  # Get just the first 15 characters...
```

18.3.4 拒绝服务

通过拒绝服务，任何 Web服务器的性能都会受到削弱。由于 Web服务器是代表远程用户处理访问请求的，因此，如果远程用户发出的访问请求太多，Web服务器就会不堪重负。这种做法有时是恶意的，而且常常是恶意的。许多时候，一些公司为 Web提供了许多服务，结果为了响应用户的访问请求，负担太重，因此不得不关闭这些服务，重新考虑自己的做法。

静态HTML页或CGI程序也会出现拒绝服务的情况。

为了防止拒绝服务的问题，你有时会感到无能为力，除非手头有足够的服务系统，能够处理浏览器的负荷。如果你的 CGI程序花费很长时间来执行或使用相当一部分系统资源（如文件访问的频率，CPU使用的密度），以便使程序能够运行，服务器将很容易受到拒绝服务的攻击。你应该设法尽量缩小你运行的 CGI程序的规模，并使之更快地运行。

18.4 宾客留言簿

这个例子使你能够为Web站点编写定制的宾客留言簿。宾客留言簿是个HTML窗体，在这个窗体中，用户可以指明来宾的名字并配有一些说明。宾客留言簿可以用来收集关于某个问题的反馈信息，作为一个简单的消息板，或者将问题提交给帮助桌。数据保存在一个文件中，并且可以在窗体信息填满后显示出来。它也可以在它自己的Web页上显示。

程序清单18-7提供了一个简短的HTML代码段，它展示了一个用于虚构帮助桌的宾客留言簿窗体。你当然可以修改这个窗体，使之适合你自己的需要。

程序清单18-7 帮助桌窗体

```

1: <FORM action="/cgi-bin/helpdesk.cgi" name="helpdesk">
2: Problem type:
3: <INPUT TYPE=radio name=proctype value=hardware>Hardware
4: <INPUT TYPE=radio name=proctype value=software>Software
5: <BR>
6: <TEXTAREA name=problem rows=10 cols=40>
7: Describe your problem.
8: </TEXTAREA>
9: <BR>
10: Your name:
11: <INPUT TYPE=text width=40 name=name><BR>
12: <INPUT TYPE=submit name=submit value="Submit Problem">
13: </FORM>
```

这个帮助桌窗体需要运行一个名叫 /cgi-bin/helpdesk.cgi的CGI程序。程序清单18-8显示了这个CGI程序。如果你想要将该CGI程序放在另外某个位置，或者将它称为另一个名字，请务必正确的URL输入程序清单18-7的帮助桌窗体中。

程序清单18-8 帮助桌CGI程序

```

1: #!/usr/bin/perl -wT
2: use strict;
3: use CGI qw(:all);
4: use Fcntl qw(:flock);
5:
6: # Location of the guestbook log file. Change this to suit your needs
7: my $gldata="c:/temp/guestbook";
8: # Any file name will do for semaphore.
9: my $semaphore_file="/tmp/helpdesk.sem";
10:
11: # Function to lock (waits indefinitely)
12: sub get_lock {
13:     open(SEM, ">$semaphore_file")
14:         || die "Cannot create semaphore: $!";
15:     flock SEM, LOCK_EX;
16: }
17: # Function to unlock
18: sub release_lock {
19:     close(SEM);
20: }
21:
22: # This function saves a passed-in help desk HTML form to a file
23: sub save {
24:     get_lock();
25:     open(GB, ">>$gldata") || die "Cannot open $gldata: $!";
```

下载

```
25:
26:     print GB "name: ", param('name'), "\n";
27:     print GB "type: ", param('probtype'), "\n";
28:     print GB "problem: ", param('problem'), "\n";
29:     close(GB);
30:     release_lock();
31: }
32: # This function displays the contents of the help desk log file as HTML,
33: # with minimal formatting.
34: sub display {
35:     open(GB, $gbdata) || die "Cannot open $gbdata: $!";
36:     while(<GB>){
37:         print "<B>$_</B><P>"; # The name
38:         my($type,$prob);
39:         $type=<GB>;
40:         $prob=<GB>;
41:         print "$type<P>";
42:         print "$prob<BR><HR>";
43:     }
44:     close(GB);
45: }
46:
47: print header;
48: # The parameter 'submit' is only passed if this CGI program was
49: # executed by pressing the 'submit' button in the form in listing 18.7
50: if (defined param('submit')) {
51:     save;
52:     display;
53: } else {
54:     display;
55: }
```

程序清单 18-8 中的大部分代码是你已经熟悉的 Perl 程序，不过请特别注意下列几个问题：

- `get_lock()` 和 `release_lock()` 这两个函数对于这个窗体是绝对必要的。对于任何一个 CGI 程序来说，你始终必须假设任何时候都可能有 CGI 程序的多个实例正在运行。写入帮助桌日志文件的 `helpdesk.cgi` 的多个实例将会出错，因此在将信息写入文件之前，该文件应被锁定。在读取文件之前，它不锁定，因为一边读取日志文件，又一边写入文件，那将是很糟糕的。
- 这个 CGI 程序有两个目的。当作为程序清单 18-7 中的窗体的目标操作来调用时，它将新项目写入日志文件。当不使用该窗体来调用该程序时，它只显示日志文件的内容。

18.5 课时小结

在本学时中，我们介绍了 HTML 窗体与 CGI 程序如何进行交互操作，如何使用 CGI 模块的 `param` 函数使你的 CGI 程序能够转换窗体的内容。另外，还介绍了怎样才能使你的 CGI 程序更加安全，如何处理受感染的数据。我们还介绍了一个简单的 CGI 客户留言簿应用程序，你可以对它定制和修改，以便适应你自己的需要。

18.6 课外作业

18.6.1 专家答疑

问题：我无法使用窗体的提交功能，老是出错，怎么办？

解答：请使用第 17 学时中介绍的 CGI 调试指南，找出存在的问题。仅仅因为它是个窗体，

并不意味着调试该窗体与调试普通 CGI 程序有什么不同。

问题：我在 Internet 上看到了这个出色的程序，但是我不懂得为什么它试图使用 \$ENV{QUERY_STRING} 来获得窗体参数。为什么？

解答：因为该程序的开发人员决定放弃该 CGI 模块的窗体处理功能。这个情况说明它可能是该 CGI 模块以前的一个非常老的 Perl 程序，也可能程序开发人员决定使用他自己的窗体处理代码。不管属于哪种情况，这表明你应该用警惕的目光观察这个程序，并且小心地使用它。

问题：我通过命令行提示符运行程序，其 # ! 行上有一个选项 -T，我得到一条出错消息 Too late for -T option (运行-T 选项太晚了)，然后程序停止运行了。为什么？

解答：你应该尽快将 -T 选项赋予 Perl 程序，这样它就知道要去寻找受感染的数据。当你的程序中的 # ! 行被处理时，这就太晚了，Perl 已经处理了你的没有感染的命令行选项。若要从命令行提示符来运行 Perl 程序，例如在调试程序中运行，你也必须在命令行提示符上设定 -T：

```
Perl -T -d foo.cgi
```

问题：Perl 的数据受感染功能是否能使我避免在 CGI 程序中犯一些愚蠢的错误？它们现在是否能够确保安全？

解答：没有一个 CGI 程序是绝对安全的。Perl 的数据受感染功能在很大程度上可使你不犯愚蠢的错误，不过它们无法保证你编写出安全的程序。

18.6.2 思考题

- 1) 在数组上下文中，不带参数的 param 函数将返回
 - a. undef。
 - b. 窗体元素的数目。
 - c. 窗体元素名的列表。
- 2) 如果你使用 CGI 模块，POST 与 GET 方法之间的差别是清楚的。
 - a. 是。
 - b. 否。
- 3) HTML 窗体上的 password 域的输入类型是安全的，因为它在发送前会对口令进行加密。
 - a. 是。
 - b. 否。

18.6.3 解答

- 1) 答案是 c。如果不带参数，param 将返回来自提交的窗体的元素名列表。
- 2) 答案是 a。
- 3) 答案是 a。在普通 HTTP 和 CGI 程序中，所有窗体域都是以明码文本传送的，因此是不保密的。口令域输入类型只在你键入口令时将该域隐藏起来而已。

18.6.4 实习

- 对帮助桌窗体稍作修改。将时间戳添加给每个项目，并且给输出添加某些颜色。
- 问题：display() 函数从最老的项目开始输出帮助桌窗体中的各个项目。请修改 display() 函数，使之首先输出最新的项目。

China-pub.com

下载

China-pub.com

下载

第19学时 复杂窗体

Web上的窗体不只是简单的单页面窗体。有时窗体要跨越若干页。这些复杂的窗体以调查、查询和购物车等应用程序的形式出现。

这些比较复杂的窗体需要使用某些不同的编程技巧，本学时你将要学习这些技巧。

在本学时中，你将要学习：

- 如何创建多页窗体。

19.1 复杂的多页窗体

使用CGI程序来编写复杂的多页窗体时，你会遇到一个特殊的编程难题。Web浏览器与Web服务器之间的连接根本不是一个持久的连接。Web浏览器与服务器建立连接，检查Web页，然后便断开与Web服务器之间的连接。在服务器与你的Web浏览器之间并不保持不间断的连接。

更为复杂的是：浏览器每次与Web服务器连接时，Web服务器并不认为该浏览器预先访问过该站点。服务器并不每次都能很容易地识别该浏览器。

类似的一种情况是：图书馆的读者与没有记忆力的图书馆管理员之间进行谈话，读者每次只能向管理员提出一个问题。

读者向图书管理员借阅一本书，比如关于亚利桑那州的一本书，图书管理员可以检索这本书。图书管理员之所以能够检索这本书，是因为这个请求很容易满足。但是读者不能要求借阅同一个专题的另一本书。图书管理员不能记住上一个借书请求，因此他无法借给你同一个专题的另一本书。如果借书的请求改为“给我另一本关于亚利桑那州的书”，图书管理员仍然无法满足读者的要求，因为他检索的书可能与第一次检索的这本书一样。

若要检索同一专题的第二本书，惟一的办法是说：“我需要另一本关于亚利桑那州的书，我已经有一本名叫《在亚利桑那州定居》的书”。这个借阅请求带有足够的能够说明问题的信息，使图书管理员能够知道什么应答是不适当的。

为Web页编写多页窗体，也可以使用同样的解决办法。每个问题/答复会话必须包含足够的信息，使Web服务器能够知道它需要做什么。你可以用几种不同的方法来创建这样的会话，其中的一种方法，即使用隐藏的HTML域，将在本学时中介绍。

19.2 隐藏域

要使Web窗体能够“记住”信息，最容易的方法是使用隐藏域，将以前的信息嵌入Web窗体。隐藏域是HTML窗体的组成部分，它使域和值成为HTML的组成部分，但是在显示窗体时，窗体中并不出现这些域和值。在HTML中，这些域和值编写为下面的形式：

```
<INPUT type="hidden" name="fullname" value="Pink Floyd">
```

如果将上面的HTML代码放入一个窗体，新的名字（“ fullname ”）和值（“ Pink Floyd ”）将成为窗体的组成部分。如果该窗体被提交给一个Perl CGI程序，param函数将返回一个关键字和隐藏域的值。

在线商店

如果要举一个如何使用隐藏域的例子，可以看一看在线商店，它使用一系列的 Web页，使人们能够根据在线目录来选购商品。目前，我们只是向你介绍复杂窗体的运行情况，在本学时后面部分的内容中，要介绍另一个复杂的窗体，它包含用于创建一个在线调查的代码。



如果不能实现某种形式的安全 Web事务处理，那么请不要使用这个在线商店的例子，请注意，这个例子并不包含任何真实的个人信息，如电话号码或信用卡号码等，因为隐藏域就像正规的 HTML窗体，它根本不具备任何安全性。

图19-1所示的在线商店第一页显示了该商店的商品清单。

图19-1 在线商店的第一页

Featured Items	
<input type="checkbox"/>	Pickled Herring
<input checked="" type="checkbox"/>	Chocolate Bananas
Go to store	

当用户单击 Go to Store (去商店) 按钮时，CGI程序接收来自窗体的值，然后显示完整的目录，如图19-2所示。

图19-2 显示在线商店的商品目录

Online Store, full catalog	
<input checked="" type="checkbox"/>	Fish heads
<input type="checkbox"/>	Chitterlings
<input type="checkbox"/>	Head Cheese
<input type="checkbox"/>	Elephant Ears
Hidden:	
Chocolate Bananas	
Ship Items	

第二页显示完整的目录。当第一页（带有商店拥有商品的目录）提交时，CGI程序接收各个值，然后当它为完整的目录输出 HTML时，它将商品的指定数量作为隐藏域放入新窗体。

每当CGI程序接收来自HTML窗体的值时，新页将包含隐藏域中的旧值，以及普通窗体元素中的新值。

采用这个方法，你可以避免“健忘的图书管理员”存在的问题，当提交完整目录的窗体时，窗体中的隐藏域便提醒CGI程序从第一个窗体中选定哪些项目以及从当前窗体中选择哪些项目。

如果需要第三页，前两页中的值可以作为隐藏域存放在第三页上，如图 19-3所示。

关于HTML页上的隐藏域，有几个问题应该加以说明。首先，隐藏域中的值是任何人都能够查看的。若要查看这些值，用户只需要查看该页的 HTML源代码。大多数Web浏览器都配有一个选项，可以用于查看HTML源代码。

其次，隐藏域中的值可以由远程用户进行修改，如果他们确实想要这样做的话。若要修

改隐藏域的值，可以使用修改后的Web浏览器，或者使用HTTP人工提交该窗体。例如，在线商店不应该将价格存放在隐藏域中，它只能存放数量。CGI程序应该在需要显示价格时才查看价格。

图19-3 在线商店的发货信息

息

The form has two input fields: 'Name' and 'Address'. Below the fields, there is a section labeled 'Hidden:' containing the values 'Chocolate Bananas' and 'Fish Heads'. At the bottom is a 'Done' button.



当你设计窗体时，看一看别人是如何设计窗体的，这将会对你有所帮助。这样你也会对他们是否使用隐藏域来保存信息这个问题有所了解。大多数Web浏览器都有一个View Page Source（查看页源）选项。你应该将这个选项用在任意窗体上，以了解它是如何形成一个整体的。但是不要拷贝这个窗体，大多数时候，拷贝会侵犯窗体的原开发人员的版权。

19.3 多页调查窗体

调查窗体是查找跨越若干不同的Web页窗体的常见地方。有时这些窗体太长，一个Web页放不下，它们通常可以分成不同的类别。

接着，简单的多页Web调查窗体可以用于查找关于你的个人信息的各个方面。这个调查窗体可以展示4个不同的Web页，并且可以改为支持你需要的任何数目的Web页。这4个Web页是：

- 第一页用于提出一系列的一般问题，有时它们可以用来查找你拥有哪些种类的个人信息。
- 第二页用于提出一些关于你的习惯爱好的特殊问题，还有一个根据第一个调查页提出的问题。
- 第三页是供你输入你的名字和对调查的说明的Web页。
- 在调查完成后输出的一条感谢你的消息。

相同的CGI程序可以用来执行所有这4个功能。它决定了哪一页用来打开下一页。这是根据刚才显示的这一页来决定的。程序清单19-1显示了调查程序的核心。



通过包含代码 use CGI::Carp qw (fatalsToBrowser)，你的CGI程序的die()消息（它通常被写入Web服务器的日志文件）将作为Web页的组成部分来输出。当你编写更长的CGI程序时，它将有助于程序的调试。

调查的结果保存在一个文本文件中，但是该程序根本不显示该结果。该程序只不过进行调查结果的收集和存储。你必须编写另一个CGI程序，以便显示调查的结果。

程序清单19-1 调查程序的第一部分

```

1:  #!/usr/bin/perl -w
2:  use Fcntl qw(:flock);
3:  use CGI qw(:all);
4:  use CGI::Carp qw(fatalsToBrowser);
5:  use strict;
6:  my $surveyfile="/tmp/survey.txt";
7:  my @survey_answers=qw(petttype daytype clothes
8:                      castaway travel risky ownpet
9:                      realname comments);
10: my $semaphore_file="/tmp/survey.sem";
11: print header;
12: if (! param ) {
13:     page_one();      # Survey just started
14: } elsif (defined param('pageone')) {
15:     page_two();      # Answered one page, print the second
16: } elsif (defined param('pagetwo')) {
17:     page_three();    # Print the last page.
18: } else {
19:     survey_done();   # Print a thank-you note, and save
20: }
```

第6~8行：在调查过程中，每个HTML窗体都包含输入域。每个输入域的名字都出现在这个数组中。save()函数和repeat_hidden()函数将在以后使用这个数组。

第12~13行：如果不将任何参数传递给该CGI程序，也就是说，它没有作为窗体发送的结果来加载，那么就调用page_one()函数来输出调查窗体的第一页。

第14~17行：如果名叫pageone的HTML窗体参数被传递给这个CGI程序，便调用page_two()函数。如果传递的参数是pagetwo，便调用函数page_three。

第19行：如果HTML窗体参数传递给这个CGI程序，但是传递的参数不是pageone或者pagetwo，则调查完成，其结果被保存，并在survey_done()函数中输出感谢你的消息。

每个Web页上的submit按钮都提供了一个关于下面应该加载哪一页的线索，你可以在图19-4中看到这个情况。由于submit按钮的名字作为一个参数被传递给CGI程序，因此它可以用来自显示刚刚提交给程序的是Web页的哪个版本。

程序清单19-2是调查程序的第二部分。

程序清单19-2 调查程序的第二部分

```

21: sub page_one {
22:     print<<END_PAGE_ONE;
23:     <FORM>
24:     Are you a "cat person" or a "dog person"?<BR>
25:     <INPUT type=radio name=petttype value=dog>Dog<BR>
26:     <INPUT type=radio name=petttype value=cat>Cat<BR>
27:     <P>
28:     Are you more of an early-riser or a night owl?<BR>
29:     <INPUT type=radio name=daytype value=early>Early riser<BR>
30:     <INPUT type=radio name=daytype value=late>Night Owl<BR>
31:     <P>
32:     At work, if you had a choice on how to dress....<BR>
```

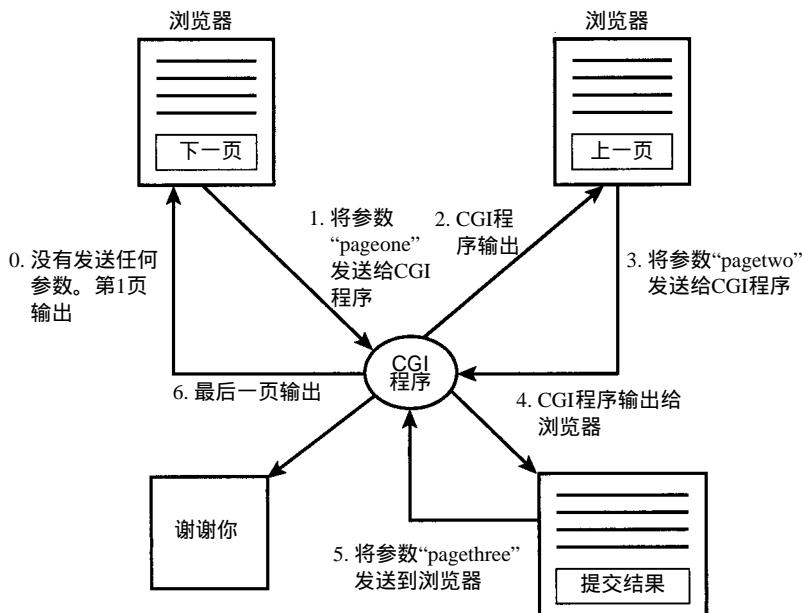
下载

```

33: <INPUT type=radio name=clothes value=casual>Casual<BR>
34: <INPUT type=radio name=clothes value=business>Business<BR>
35: <P>
36: If stranded on a desert island,
37: who would you rather be stuck with?<BR>
38: <INPUT type=radio name=castaway value=ginger>Ginger<BR>
39: <INPUT type=radio name=castaway value=marya>Mary-Anne<BR>
40: <INPUT type=radio name=castaway value=prof>Professor<BR>
41: <INPUT type=radio name=castaway value=skipper>Skipper<BR>
42: <INPUT type=submit name=pageone value="Next page">
43: </FORM>
44: END_PAGE_ONE
45: }

```

图19-4 哪个按钮用于执行
哪个操作的示意图



第22~24行是个 Perl新结构 , 你以前没有看到过 , 它称为 “ here document ”。here document使你可以设定一个跨越若干行的字符串 , 它包含其他的引号 , 可以像一个普通的双引号那样来运行。若要开始编写一个 here document , 你可以使用<< , 后随一个单词。这个引号的内容就继续下去 , 直到在一行的开头再次出现该单词为止 , 如下例所示 :

```

$a=<<END_OF_QUOTE;
This is included as part of the string.
END_OF_QUOTE

```

用于标识here document开始的单词 , 即上面这个代码段中的 END_OF_QUOTE , 或者程序清单19-2中的END_PAGE_ONE , 后面必须跟一个分号。在 here document的结尾 , 该单词必须出现在第一列的开头 , 并且后面不能有任何字符 , 如空格或分号。在 here document内 , 变量像它们在普通双引号字符串 (“ ”) 中那样展开 , 因此在 here document中 , 必须慎重使用\$和@字符。

使用here document时 , 可以将大量的HTML代码嵌入你的Perl程序 , 而不会夹杂许多引号和多个print语句 , 从而造成混乱。

程序清单19-2中的函数只是用来输出一个HTML窗体。<FORM>标记并不包含动作和方法。

当没有设定<FORM>的action属性时，当前的CGI程序（即产生窗体的CGI程序）将在提交窗体时重新加载。当不提交method属性时，便使用默认方法GET。

请注意，窗体上的提交按钮的名字是pageone。当该窗体被提交时，一个称为pageone的参数将被发送到该CGI程序，它的值并不重要。被提交的这个参数将提示CGI程序加载第二个Web页。

程序清单19-3是CGI程序的第三部分。

程序清单19-3 调查程序的第三部分

```

46: # Print out any of the responses so far as hidden fields
47: sub repeat_hidden {
48:     foreach my $answer (@survey_answers) {
49:         if (defined param($answer)) {
50:             print "<INPUT TYPE=hidden";
51:             print " name=$answer ";
52:             print " value=\"", param($answer), "\">\n";
53:         }
54:     }
55: }
56: sub page_two {
57:     my $pet=param('pettype');
58:     if (! defined $pet) {
59:         $pet="goldfish";
60:     }
61:     print<<END_PAGE_TWO;
62: <FORM>
63: Would you rather...<BR>
64: <INPUT type=radio name=travel value=travel>Travel<BR>
65: <INPUT type=radio name=travel value=home>Stay at home<BR>
66: <P>
67: Do you consider yourself...<BR>
68: <INPUT type=radio name=risky value=yes>A daredevil<BR>
69: <INPUT type=radio name=risky value=no>Cautious<BR>
70: <P>
71: Do you own a $pet?<BR>
72: <INPUT type=radio name=ownpet value=$pet>Yes<BR>
73: <INPUT type=radio name=ownpet value=no>No<BR>
74: <P>
75: <INPUT TYPE=submit name=pagetwo value="Last Page">
76: END_PAGE_TWO
77:     repeat_hidden();
78:     print "</FORM>";
79: }
```

第47行：正如第46行中的注释所表示的那样，这一行中的函数用于输出作为隐藏域的该窗体的所有域的值。数据@survey_answers包含HTML窗体上所有可能的“name=”值。当第一次运行时，大多数域将不存在，因为调查的这些部分尚未填入相应的值。

第48~49行：@survey_answers中可能的每个参数均被检查，每个参数均被定义。输出HTML标号<INPUT TYPE=hidden>，用于存放当前窗体上的值。

第56行：这个函数用于输出调查的第二页。

第57~60行：这个函数在调查的第二页中调用。如果调查的第一页填入了正确的值，那么param('pettype')将保存dog或cat，这个值将存放在\$pet中。如果被调查人跳过了这个问题，同时param('pettype')没有定义，那么就改用goldfish。

第61~76行：来自第一页的HTML窗体参数均转入该窗体，作为其隐藏域。

如果你在这时查看调查窗体，即它的第二页，那么第一页的所有答案均作为隐藏域存放在第二页的结尾处。程序清单19-4显示了第三页的代码。

程序清单19-4 调查程序的第四部分

```
80: sub page_three {
81:     print<<END_PAGE_THREE;
82: <FORM>
83: Last page! This information is optional!<BR>
84: Your name:
85: <INPUT TYPE=text name="realname"><BR>
86: Any comments about this survey:<BR>
87: <TEXTAREA NAME=comments cols=40 rows=10>
88: </TEXTAREA>
89: <P>
90: <INPUT TYPE=submit name=pagethree
91:     value="Submit survey results">
92: END_PAGE_THREE
93:     repeat_hidden();
94:     print "</FORM>";
95: }
```

函数page_three()是非常明了的。它只是输出窗体中的一个文本框和一个文本区域。在结尾处，它再次调用repeat_hidden()函数，以便将所有隐藏域放入调查窗体的第三页。程序清单19-5显示了CGI调查程序的结尾部分。

程序清单19-5 调查程序的最后部分

```
96: sub survey_done {
97:     save();
98:     print "Thank You!";
99: }
100: #
101: # Save all of the survey results to $surveyfile
102: #
103: sub save {
104:     get_lock();
105:     open(SF, ">>$surveyfile") || die "Cannot open $surveyfile: $!";
106:     foreach my $answer (@survey_answers) {
107:         if (defined param($answer)) {
108:             print SF $answer, " = ", param($answer), "\n";
109:         }
110:     }
111:     close(SF);
112:     release_lock();
113: }
114: #
115: # Locks and Unlocks the survey file so that multiple survey-takers
116: # Don't clash and write at the same time.
117: #
118: #
119: # Function to lock (waits indefinitely)
120: sub get_lock {
121:     open(SEM, ">$semaphore_file")
122:     || die "Cannot create semaphore: $!";
123:     flock SEM, LOCK_EX;
```

```

124: }
125:
126:# Function to unlock
127: sub release_lock {
128:     close(SEM);
129: }

```

第96行：调用这个函数只是为了输出感谢你的消息。当某人遍历调查窗体的 3个页面后，这样做总是一件很好的事情。然后调用 save() 函数。

第103行：这里的 save() 函数几乎是第18学时中的 save 函数的复制品。它用 get_lock() 将调用文件锁定，再使用类似 repeat_hidden() 中的方法写入对问题的答案，然后用 release_lock() 函数对文件解锁。

你可以随意修改这个调查程序，以适应你自己的需要。它的设计非常灵活，并且可以用于许多不同的目的。

19.4 课时小结

在本学时中，你学习了如何创建多页 Web窗体的方法。当你进行这项操作时，了解到程序需要解决的几个问题，最重要的是要记住从一页转到另一页时会出现的一些情况。你还学会了如何使用隐藏域将信息存放在服务器无法记住的 Web页上，然后就可以使用隐藏域来创建框架调查窗体了。

19.5 课外作业

19.5.1 专家答疑

问题：HTML窗体难道一定是如此不顺眼吗？

解答：本书中介绍的这些窗体是简单的、缺乏特色的框架式的窗体，有的人称它们是不顺眼的窗体。本书的目的是教你进行 Perl和CGI编程，而不是教你如何使用 HTML。实际上，本书中讲到的大多数 HTML程序与标准无关，并且它是不完整的，它没有使用 <HEAD>标记，和<HTML>标记，也没有 DTD标题。通过提供基本的 HTML，我想你能够对它进行修改，使之符合你的需要。

前面讲过，给窗体增色的好办法是查找 Web，寻找你喜欢的窗体。通过查看源代码，你就会对如何将这些 Web页组合在一起有个大致的了解。

问题：我看到这样一个出错消息：Can't find string terminator “ xxxx ” anywhere before EOF at ...。这是什么意思？

解答：这个错误是因为在程序的某个位置上有一个左引号，但是没有匹配的右引号而造成的。当你使用“ here document ” 时，这意味着无法找到你给“ here document ” 做上结尾标记的单词。它的格式如下：

```

print <<MARK;
text
text
text
MARK

```

在上面这个例子中，“ here document ” 开头和结尾的单词 MARK必须完全相同。结尾的单

词这一行上，它的前面不能有任何东西，后面也不能有任何东西。MS-DOS和Windows的文本编辑器有时并不在程序的最后一行的后面放上行尾符。如果你的“here document”以文件的结尾为结束，请在它的后面放上一个空行。

19.5.2 思考题

- 1) 为了使你的程序能够记住很长的多页Web事务处理，你需要使用
 - a. 数据库和cookie。
 - b. 隐藏的HTML窗体域。
 - c. 隐藏的HTML窗体域、cookie和数据库的某种组合。
- 2) 使用HTML的<FORM>标记时，如果不带action属性，那么它将
 - a. 无法运行。
 - b. 导致submit按钮使用原先生成Web页的CGI程序。
 - c. 导致submit按钮重新加载当前页。
- 3) 上面介绍的调查程序有一个小错误，是什么错误？
 - a. print<<EOP；它在程序清单19-2中是个无效语句。
 - b. HTML不完整，因为它不带<HEAD>标记和类似的标记。
 - c. 调查程序没有输出结果。

19.5.3 解答

- 1) 答案可以是b或c。你可以只使用隐藏的HTML域，也可以只使用cookie。如果只使用数据库，那是不行的。
- 2) 答案是b。重新加载当前页将会删除当前窗体的所有答案。如果没有action属性，<FORM>标记将把当前页的URL用作替换URL。
- 3) 答案是b。print<<EOP；肯定是个有效的语句，它称为“here document”。选择c是不正确的，因为程序不是使用该方法设计的（参见“实习”这一节）。

19.5.4 实习

- 编写一段CGI程序，用于显示调查的结果。也可以创建一个表格，以下面的形式显示这些结果：

猫/狗	拥有一只	夜间活动	服装	被谁丢弃	旅行者	有危险吗
猫	否	是	普通	教授	是	是
两者之一	金鱼	否	专用	船长	否	否
狗	是	否	普通	玛丽·安尼	是	是

- 补充问题，编写一个CGI程序，将调查结果汇总成下面的形式：

猫/狗的比例	猫 40%	狗 45%	其他 15%
拥有该宠物的人	猫 20%	狗 15%	金鱼 30% 无 35%

的比例：

夜行者：	是 35%	否 40%
------	-------	-------

China-pub.com

下载

China-pub.com

下载

第20学时 对HTTP和CGI进行操作

在本学时中，你将要学习如何对 Web进行一系列有趣的操作。可以使用 CGI程序，使 Web站点变得更加灵活，并且更加便于管理。

在本学时中，你将要学习：

- 如何将HTML程序从服务器传送到你的浏览器。
- 如何使CGI程序能够发送HTML文档。
- 如何将值直接传递给CGI程序。
- 服务器端的包含程序如何运行。
- 如何查询浏览器和服务器，以便找到你要的信息。

20.1 HTTP通信概述

在第17学时中，我们介绍了 Web浏览器（Netscape和Internet Explorer等）与 Web服务器（Apache和IIS等）之间如何进行基本的通信。该学时介绍的通信方式显得过分简单了一些。现在我们对CGI程序的使用变得更加得心应手了，因此可以更加深入地探讨这个问题。在本学时的后面部分中，我们将要介绍进行这种通信时使用的一些方法，以便执行某些有意思的任务。

这种通信方式可以用一个协议来加以描述，这个协议称为超文本传输协议（HTTP）。该协议目前的两个版本是HTTP 1.0和HTTP 1.1。在本学时介绍的一些例子中，两个版本均可适用。



描述 Internet上使用的这些协议的 Internet标准文档称为“Request For Comment(说明请求)”，即通常所说的 RFC。RFC由Internet工程组负责维护，你可以通过网址 <http://www.ietf.org> 在Web上查看。专门介绍 HTTP的文档是 RFC 1945和RFC2616。请注意，这些文档的技术性很强。

当你的Web浏览器初次与 Web服务器连接时，浏览器向服务器发送一条初始消息，它类似下面的形式：

```
GET http://testserver/ HTTP/1.0
Connection: Keep-Alive
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, image/png, /*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en, en-GB, de, fr, ja, ko, zh
Host: testserver:80
User-Agent: Mozilla/4.51 [en]C-c32f404p (WinNT; U)
```

GET用于指明你试图接收的是什么 URL，以及你想要接受的是哪个版本的协议。在这个例子中，你接受的是HTTP 1.0版的协议。

connection行用于指明你希望这个连接为检索多个 Web页保持打开状态。按照默认设置，浏览器为检索每一帧、每一页和 Web页上的每个图形分别建立一个连接。命令 keep-Alive要求服务器使连接保持打开状态，以便使用相同的连接检索多个项目。

Accept行用于指明通过这个连接你愿意接受何种类型的数据。第一个 Accept行的结尾处的`/*`表示你愿意接受任何种类的数据。下一行（`iso-8959-1`等）表示字符编码可用于该文档。在这个例子中，Accept-Encoding表示gzip（GNU Zip）可用于对来自服务器的数据进行压缩，以便加快传输速度。最后，Accept-Language用于指明该浏览器能够接受何种语言（英语、大不列颠英语、德语和法语等）。

Host是你希望租用的Web站点的系统名。由于可以使用虚拟租用，因此该系统名可以不同于URL中的主机名。

最后一行，该浏览器将自己的身份通知 Web服务器，这个身份是 Mozilla/4.51[en]C-c32f404p（WinNT; U）。在Web技术中，该浏览器称为用户代理。

然后，服务器发送一个应答消息，它类似下面的形式：

```
GET http://testserver/ --> 200 OK
Date: Thu, 02 Sep 1999 19:54:39 GMT
Server: Netscape-Enterprise/3.5.1G
Content-Length: 2222
Content-Type: text/html
Last-Modified: Wed, 01 Sep 1999 17:12:03 GMT
```

这时，该应答消息后随你要检索的Web页内容。

在这个消息中的GET行用于指明服务器是否将这个Web页发送给你。状态200表示一切运行正常。服务器还在 Server行上标明自己的身份。在这个例子中，该服务器是 Netscape-Enterprise/3.51 G Web服务器。

Content-Length行表示2222字节的内容将被发送给浏览器。使用这个消息，你的浏览器就能够知道一个Web的内容完整性是50%还是60%等。Content-Type是发送给浏览器的Web页的种类。如果是HTML页，这一行就设置为text/html。如果是图形页，它就设置为image/jpeg。

Last-Modified日期表示自从该Web页上次被检索以来是否被修改了。大多数Web浏览器都将Web页缓存起来，这样你就可以两次阅读一个Web页，这时，该日期就可以与浏览器已经拥有的保存拷贝日期做比较。如果服务器上的Web页尚未修改，就没有必要再次下载整个Web页。

20.1.1 举例：人工检索Web页

如果你愿意的话，可以人工检索Web页。当想要测定Web服务器发送的是否是正确的Web页时，常常可以使用这个特性。

若要运用这个特性，需要一个专门的程序，称为 Telnet客户程序。Telnet客户程序是个远端访问程序，用于远程登录到UNIX工作站。不过它常常用于执行调试HTTP之类的任务。

如果你有一台UNIX计算机，可能已经安装了Telnet。如果你拥有一台Microsoft Windows计算机，Telnet可能已经作为你的网络实用程序的一部分安装好了。你只需要打开 Start菜单，使用Run选项，就可以运行Telnet客户程序。如果尚未安装该程序，或者使用的是Macintosh计算机，你可以在任何较好的下载站点找到免费的Telnet客户程序。

若要启动与Web服务器的通信，请在提示符处输入下面这个Telnet命令：

```
$ telnet www.webserver.com 80
```

这里的www.webserver.com是Web服务器的名字，80是你想要连接到的端口号（端口80通常是Web服务器接收信息的端口）。如果你的Telnet客户程序是个图形处理程序，你必须在对话框中设置这些值。

当Telnet进行连接时，你可能看不到提示符或连接消息。请不必担心，这是正常的。HTTP期望客户机首先发出请求，而服务器却没有发出提示。在UNIX下，你会得到一条消息，其内容如下：

```
Trying www.webserver.com
Connected to www.webserver.com
Escape character is '^']'
```

其他类型的系统，如Windows和Macintosh，则看不到这条消息。

你必须认真和迅速地键入下面这行命令：

```
GET http://www.webserver.com/ HTTP/1.0
```

键入这行命令后，请按Enter键两次。这时Web服务器应该作出响应，发出正常的HTTP标题和Web站点的顶层页，然后切断连接。

20.1.2 举例：返回非文本信息

你的CGI程序不一定将HTML信息返回给浏览器。实际上，你的浏览器能够检索的任何信息，CGI程序都能够发送。

CGI模块中的header函数告诉浏览器，它准备使用MIME内容类型（Content-Type）标题来接收何种类型的数据。Content-Type标题用于描述后随的数据内容，这样，浏览器就知道如何处理该数据。

按照默认设置，header函数将一个text/html的内容类型描述发送给浏览器。浏览器识别后随的内容是带有HTML的文本。

通过告诉浏览器将会收到不同类型的数据，你就可以控制浏览器如何来处理该数据。数据可以作为图形来显示，也可以传递给浏览器的插件，甚至可以由浏览器启动的外部程序来运行。

若要使header函数能够发送非普通text/html标题的某些信息，请使用-type选项，如下所示：

```
print header(-type => MIME_type);
```

可以发送给浏览器的某些常用MIME内容类型是text/plain（指不需要转换的文本），image/gif和image.jpeg（指GIF和JPEG图形），以及application/appname（指应用程序appname特定的数据）。一种特殊的MIME内容类型称为application/octet-stream，它是指浏览器应该保持到一个文件的原始二进制数据。

如果你需要创建一个“当日图形”的Web站点，或者创建一个Web标题广告，就可以使用这种内容类型。每天修改Web页，以便反映新图形的变化情况，这是很麻烦的。如果你出差在外，谁为你更新“当日图形”呢？为此，你可以使用一个静态HTML页，并且使用Perl CGI程序，每天自动产生一个不同的图形。

在你的Web页中，使用下面这样的HTML代码：

```
<BODY>
Today's image of the day is:
<IMG SRC="/cgi-bin/daily_image.cgi">
</BODY>
```

在上面这个HTML代码中，请注意标记的目标是个CGI程序，不是.gif或.jpg。接着，你需要一个放满图形的目录，图形的数量至少要与一个月的天数相同。你可以调用你喜欢的

任何图形，只要文件名以.jpg结尾即可。请注意，该程序能够非常容易地使用GIF图形。

CGI程序daily_image.cgi类似程序清单20-1所示的形式。

程序清单20-1 当日图形的代码

```
1: #!/usr/bin/perl -w
2:
3: use strict;
4: use CGI qw(:all);
5: my($imagedir, $day, @jpegs, $error);
6:
7: $imagedir="/web/htdocs/pic_of_day";
8: $error="/web/htdocs/images/error.jpg";
9:
10: sub display_image {
11:     my($image)=@_;
12:     open(IMAGE, "$image") || exit;
13:     binmode STDOUT; binmode IMAGE;
14:     print <IMAGE>;
15:     close(IMAGE);
16:     exit;
17: }
18:
19: print header(-type => 'image/jpeg');
20:
21: # Day of the month, 1-28, 29, 30, or 31
22: $day=(localtime)[3];
23: $day=$day-1;      # We want day 0-27, etc..
24:
25: opendir(IMGDIR, $imagedir) || display_image($error);
26: @jpegs=sort grep(/\.jpg$/, readdir IMGDIR);
27: closedir(IMGDIR);
28:
29: my $image="$imagedir/$jpegs[$day]";
30: $image=$error if (not defined $jpegs[$day]);
31: display_image($image);
```

第7行：这一行用于设定图形所在的目录。可以修改这个设置，以指明你将图形放在什么位置。

第8行：这一行非常奇怪，因为这个CGI程序并不输出文本，并且因为它嵌入到的HTML页时并不将输出显示为文本，你不能只是输出出错消息。如果无法打开\$imagedir目录，则变量\$error包含将要显示的.jpg文件的名字。

第10~16行：这个子例程将图形显示在标准输出中，它将发送到浏览器。在Windows平台上，STDOUT被视为一个文本文件，将.jpg输出到STDOUT将会损坏图形。因此，binmode用于使STDOUT和IMAGE成为二进制文件句柄。在UNIX下，你不需要使用binmode，但是它不会造成损害。请注意第12行，如果图形打不开，就没有必要输出出错消息，程序只要退出即可。

第19行：这一行用于输出标准HTTP标题，不过Content-Type将是image/jpeg，而不是通常的text/html。

第25行：图形目录被打开以便读取。如果图形目录没有打开，那么便用错误图像\$error来调用函数display-image()。

第26行：这一行比较复杂，因此要循序渐进来操作。首先用readdir读取目录。然后从该列表中取出以.jpg结尾的文件名。最后，对产生的列表进行排序，并赋予@jpegs。

20.2 如何调用CGI程序的详细说明

到现在为止，我们介绍了启动 CGI 程序时使用的两种方法。第一种方法也是最简单的方法是通过一个链接来调用 CGI 程序的 URL，或者让用户将 URL 键入浏览器。因此类似下面的这行代码可以用于启动和运行称为 time.cgi 的程序：

```
<A HREF="http://server/cgi-bin/time.cgi">Click here for the time</A>
```

当通过该链接进行操作时，CGI 程序 time.cgi 将由服务器运行，它的输出则作为一个新 Web 页来显示。这个示例代码简单明了，容易操作，与第 17 学时中的“Hello, World！”很相似。

启动 CGI 程序的另一个方法是使它成为一个 HTML 填充式窗体的目标程序。例如，当单击 Submit 按钮时，下面这个窗体便调用 CGI 程序 process.cgi：

```
<FORM METHOD=GET ACTION="/cgi-bin/process.cgi">
<INPUT TYPE=TEXT NAME=STUFF><BR>
<INPUT TYPE=SUBMIT>
</FORM>
```

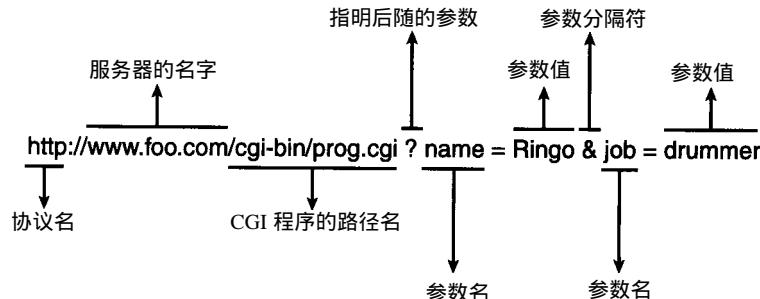
这个调用 CGI 程序的方法具有另一个优点，即你可以将参数传递给 CGI 程序，以便进行处理。好了，这就是 HTML 窗体的总体情况。

20.2.1 将参数传递给CGI程序

通过链接将信息传递给 Perl 程序，这是不是很好呢？例如，是否可以在文档中设置一个点击的链接，它将“运行 CGI 程序 foo.cgi，其 X 值等于 this，Y 值等于 that”呢？你稍加努力，就可以达到这个目的。

首先必须在 <A HREF> 标记中使用一种特殊的 URL。该 URL 的格式在图 20-1 中做了说明。

图 20-1 包含各个参数的 URL



每个参数都是你想要传递到 CGI 程序中的一个值的名字（类似一个指明的 HTML 窗体元素），该值是该名字的值。例如，若要创建一个链接，单击这个链接时，它将运行一个 CGI 程序，其参数 sign 设置为 Aries，year 设置为 1969，那么你可以输入下面这行代码：

```
<A HREF="http://www.server.com/cgi-bin/astrology.cgi?sign=Aries&year=1969">
Aries, year of the Rooster</A>
```

在这个 CGI 程序中，它的参数将像通常那样由 CGI 模块的 param 函数进行处理：

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

```
use CGI qw(:all);
use strict;
```

```
print header;
print "The year ", param('year'), " and being born under ",
      param('sign'), " indicates you are brilliant.\n";
```

你可以根据需要传递任意数量的参数。如果你想传递一个空参数，即没有值的参数，只需要像下例中的author那样将它置空即可：

```
<A HREF="http://www.server.com/cgibin/book.cgi?author=&title=Beowulf">Beowulf</A>
```

20.2.2 特殊参数

当你调用带有此类参数的CGI程序时，应该了解使用某些特殊参数时要考虑的问题。某些字符属于特殊字符，不能成为URL的组成部分。例如，？（问号）是个特殊字符，它可以作为URL的主要部分与参数之间的分隔标号。其他特殊字符还有 &、空格和引号等。



特殊字符的完整列表在Internet标准文档RFC 2396中列出。

若要将这些特殊字符中的某一个插入URL，你必须对字符进行转义。在这种情况下，对字符进行转义意味着应该将它的ASCII值转换成一个两位数十六进制数字，并在它的前面加上一个百分比符号。对“Hello,World!”的编码如下所示：

```
Hello%2C%20World
```

显然，创建一个URL转义字符串是非常麻烦的。CGI模块提供了一个函数，它能够自动为你创建这样的字符串。下面这个代码段展示了如何输出一个带有正确编码的URL：

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

use strict;
# The 'escape' function must be pulled in manually
use CGI qw(:all escape);

print header;
my $string="Hello, World!";
print '<A HREF="http://www.server.com/cgi-bin/parrot.cgi?message='
     , escape($string) , '">Click Me</A>';
```

上面这个代码可以产生一个正确进行URL字符转义的HTML链接。请注意CGI模块是如何用于代码的use CGI qw (:all escape)；。如果你使用CGI模块，那么escape函数通常不能供你的程序使用，你必须显式要求使用该函数。

下面这个程序创建了一个带有转义值的长得的URL：

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

use strict;
use CGI qw(:all escape);

my %books=( Insomnia => 'S. King', Nutshell => 'O'Reilly');
# Start with a base URL
my $url="http://www.server.com/cgi-bin/add_books.cgi?";

# Accumulate on the end of the URL with concatenation "."
foreach my $title (keys %books) {
    $url.=escape($title); # Escape the title, add it
    $url.="=";
    $url.=escape($books{$title}); # Same with Author
```

```

    $url.= "& ";
}

print header;
print "<A HREF=$url>Add books to library</A>";

```

当CGI程序用param函数取出这些参数时，在URL的结尾处的最后一个&将被该CGI程序忽略。

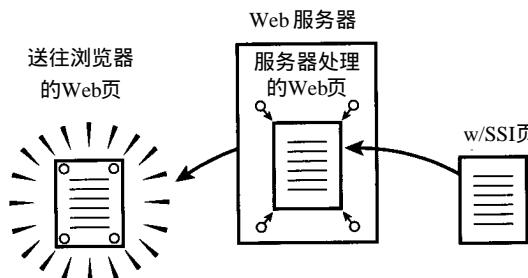
```
http://www.server.com/cgi-bin/add_books.cgi?Insomnia=S.%20King&Nutshell=O%27Reilly&
```

20.3 服务器端的包含程序

当你设计Web页时，该页上的最常见的内容是静态的。有时该页的某些部分要进行修改，但是总的来说，该页的内容将保持不变。请看这样一个Web页，它显示了一家公司的当前股票价格。该页的主要部分是静态的，比如导航栏、图形、徽标、使用信息、页眉、页脚和标题等。该页的重要部分，即股票价格，是通过读取某处的数据库和填写空格来生成的。

为了帮助你创建这种Web页，大多数Web服务器都支持一个特性，称为服务器端的包含程序（SSI），也称为服务器分析的HTML。该特性使得Web站点的开发者能够创建基本静态的HTML Web页，并使该页的某些部分由Web服务器在运行中重新编写（见图20-2）。可以将该Web页视为填空式HTML文件，而CGI程序则为你将数据填入空格。

图20-2 当HTML页被处理时，Web服务器将数据填入该页



你的服务器管理员必须激活SSI，使这些示例代码能够运行。为了使服务器能够正确地读取带有嵌入式SSI的HTML，有时你必须为HTML赋予带有.shtml或.stm扩展名的名字。请与你的服务器管理员联系，以便了解SSI是如何在你的特定Web服务器上使用的，因为它支持的命令及其语句是各不相同的。

当Web服务器从磁盘上读取静态HTML页时，它要寻找它能够替换的各个值的“标记”。在服务器分析的HTML页中的Apache Web服务器下，标号`<!--#echo var="LAST_MOCIFIED"-->` 将使Apache能够替换标记中该Web上次被修改的日期。浏览器看不到这个进程的发生，它只看到服务器替换时的这个日期。下表说明了这个进程的情况：

Web页	转换成的内容
<code><HTML></code>	<code><HTML></code>
<code><BODY></code>	<code><BODY></code>
This page was last changed:	This page was last changed:

(续)

Web页	转换成的内容
<!--#echo var= "LAST_MODIFIED" -->	Wednesday, 01-Sep-1999 21:29:31 EDT
</BODY>	</BODY>
</HTML>	</HTML>



Web服务器实现SSI的方法各有差异。有时标记的句法各不相同，有些服务器支持某些种类的标记，但是有些服务器则不支持这些标记。有些Web服务器根本不支持SSI。例如，Microsoft的Personal Web服务器就不支持SSI。本学时中使用的SSI HTML标记与Apache Web服务器及Microsoft的Internet Information服务器的标记是兼容的。在撰写本书时，后面两种服务器是Web上最流行的Web服务器。

本学时并不打算全面介绍SSI的所有特性，因为这些特性的数量太多，而且大多数特性是某些特定品牌的Web服务器所特有的。我们的目的是要介绍SSI标记#exec。你可以像下面这样将SSI #exec标号用于HTML文件：

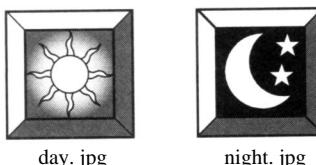
```
<!--#exec cgi="/cgi-bin/stockprice.cgi"-->
```

当Web服务器遇到这个#exec标记时，它将停下来执行 stockprice.cgi这个CGI程序。该CGI程序的输出在送往浏览器时被插入HTML数据流。当CGI程序完成运行时，#exec标记后面的HTML文件的剩余部分被送往浏览器。

举例：使用SSI

在这个例子中，你将创建一个简单的Web页，用于输出“Hello,World”，然后根据一天中的时间，输出一个定制图形，一个用于晚上，另一个用于白天，如图20-3所示。

图20-3 白天和晚上使用的两个图形：day.jpg
和night.jpg



然后你需要一个HTML样板文件，它带有问候你的信息，如下面的代码所示。如果你为自己创建这个文件，请记住必须在给HTML文件命名时使用扩展名.shtml或.stm，以便使服务器能够识别SSI标记。

```
<HTML>
<HEAD>
<TITLE>Welcome Page</TITLE>
</HEAD>
<BODY>
Welcome to this web page. Currently, out my window I see:
<!--#exec cgi="/cgi-bin/sunmoon.cgi"-->
</BODY>
```

```
</BODY>
</HTML>
```

在程序sunmoon.cgi中，可以使用程序清单20-2所示的代码。

程序清单20-2 白天和晚上的问候程序

```
1:  #!/usr/bin/perl -w
2:
3:  use CGI qw(:all);
4:
5:  # The hour from localtime() is in 24-hour format
6:  my $hour=(localtime)[2];
7:  my $image;
8:
9:  # Before 6am or after 6pm, it's nighttime
10: if ($hour<6 or $hour>18) {
11:     $image="night.jpg";
12: } else {
13:     $image="day.jpg";
14: }
15: print header;
16: print qq{<IMG SRC="$image" ALT="$image">\n};
```

第3行：由于这是个CGI程序，因此你应该使之包含CGI模块。qw (:all) 用于确保你能够使用需要的任何函数。

第6行：列表上下文中的localtime返回一个描述当前时间的元素列表。这个问题已经在第4学时中做了介绍。localtime前后的括号将它置于一个列表上下文中，[2]则使该列表的第三个元素得以返回，并赋予\$hour。元素#2是用24小时的格式来表示的时间。

第15行：标题仍然必须使用CGI的header函数来输出，尽管这个输出显示在整个Web页的中间位置上。

第16行：``标记既可以用\$day的值来输出图形，也可以用\$night的值来输出图形。如果浏览器无法显示图形，则使用ALT(替代)标记。

当浏览器在上午8点钟检索Web页时，产生Web页的源代码将如下所示：

```
<HTML>
<HEAD>
<TITLE>Welcome Page</TITLE>
</HEAD>
<BODY>
Welcome to this web page. Currently, out my window I see:
<IMG SRC="day.jpg" ALT="day.jpg" z
</BODY>
</HTML>
```

20.4 部分环境函数简介

到现在为止，来自CGI模块的大多数函数都是用于控制浏览器的(如`redirect`或`header`函数)，或者用于处理传递给CGI程序的参数(如`escape`和`param`函数)。CGI模块中的全部函数都是为了向你提供关于你当前正在运行的系统的信息的。表20-1显示了部分函数的列表，如果你要查看完整的列表，请在命令行提示符处键入`perldoc CGI`，以便查看CGI模块的在线文档。



这些函数大多需要使用Web服务器提供的值，或者使用Web浏览器使用HTTP协议发送的值。Web浏览器可能提供某些虚假的值，比如`referer`值或`user_agent`，Web服务器有时也会返回不准确的值，例如`server_name`并不总是返回你期望的值。

表20-1 部分环境函数一览表

函 数	说 明
<code>referer</code>	返回将你发送到该Web页的链接的URL。(不错，这个函数拼写错了。原先描述这个域的Internet标准包含一个拼写错误，现在国际上也将错就错，以便做到统一。)
<code>user_agent</code>	返回一个字符串，用于指明要求检索 Web页的浏览器的种类(如 Netscape、IE、Lynx)
<code>remote_host</code>	返回要求检索 Web页的系统的主机名或 IP地址。你究竟会得到哪个值，取决于你的Web服务器的配置和是否存在主机名
<code>script_name</code>	返回正在运行该程序的程序名，作为部分 URL(如 <code>/cgi-bin/foo.cgi</code>)
<code>server_name</code>	返回托管CGI程序的服务器的名字
<code>virtual_host</code>	返回用于运行该CGI程序的虚拟主机名。该函数不同于 <code>server_name</code> ，因为一个服务器常常能够托管多个Web站点。 <code>Virtual_host</code> 返回被访问的特定Web站点的名字

下面是用于展示这些函数的一个短程序：

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

use strict;
use CGI qw(:all);

print header;

print "You were sent from: ", referer, "<BR>";
print "You are apparently running: ",
      user_agent, "<BR>";
print "Your system is called: ", remote_host,
      "<BR>";
print "The name of this program is: ",
      script_name, "<BR>";
print "It's running on the server: ",
      server_name, "<BR>";
print "The server's calling itself: ",
      virtual_host, "<BR>";
```

在一个测试用的Web服务器上运行该程序，将产生下面的结果：

```
You were sent from: http://testsys.net/links.html
You are apparently running: Mozilla/4.51 [en] (Win95; I)
Your system is called: 192.168.1.2
The name of this program is: /cgi/showstuff.cgi
It's running on the server: testsys
The server's calling itself: perlbook
```

20.5 重定向

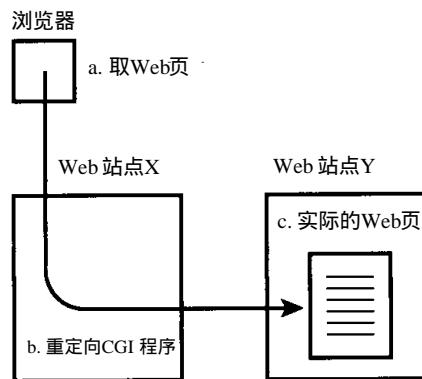
CGI程序中可以使用的一个非常有用的方法称为 HTTP重定向。当想要根据某个计算值让CGI程序加载另一个Web页时，就可以使用重定向。

例如，如果你有一连串的Web页要送往一个特定的浏览器，比如，这些Web页包含一个插件，而这个插件只有在 Microsoft Windows的Netscape浏览器下才能使用，那么可以将该Web

站点的访问者送往同一个URL，并让一个CGI程序将他们重定向到正确的Web页。图20-4说明了这个情况。

图20-4 一个Web站点重定

向到另一个Web站
点



若要实现重定向，需要使用CGI模块的redirect函数。redirect函数用于对前面介绍的HTTP通信进行操作，并使浏览器能够加载一个新Web页。

程序清单20-3包含一个短程序，用于将Windows下的Netscape用户重定向到一个Web页，并将所有其他浏览器重定向到另一个Web页。

程序清单20-3 根据浏览器进行重定向

```

1:  #!/usr/bin/perl -w
2:
3:  use CGI qw(:all);
4:  use strict;
5:  my($browser, $target);
6:
7:  # Fetch the browser's name
8:
9:  $browser=user_agent;
10: $target="http://www.server.com/generic.html";
11:
12: # Test for WinXX and Netscape
13: if ($browser=~~/Mozilla/ and $browser=~~/Win/) {
14:     $target="http://www.server.com/netscape.html";
15: }
16: print redirect( -uri => $target );

```

第9行：在\$browser中抓取浏览器类型。

第10行：默认URL被放入\$target中。任何非Netscape浏览器均被送往这里。

第13~14行：存放在\$browser中的浏览器类型被核实，以确定它是否包含Mozilla或Win，如果包含，则赋予一个新目标地址。

第16行：重定向消息被发送到浏览器。

通过CGI的重定向是天衣无缝的，而通过其他方法（如使用JavaScript和HTML扩展名）进行的重定向则存在许多问题。并非所有平台都支持JavaScript，使用JavaScript中的window.location.href赋值语句可能无法产生正确的结果。如果将HTML的<META HTTP-EQUIV=“refresh”>标记用于重定向，就会在重定向进行之前产生明显的延迟，因为浏览器必须在重定向发生之前全部加载Web页。JavaScript也同样存在这个问题。HTTP重定向是在发送任何HTML之前发生的，并且几乎是即时进行的。



对于CGI模块的user_agent函数，Netscape的浏览器将自己标识为 Mozilla。这个名字是图形 Web浏览器原先的名字 Mosaic的变形。Windows 95下的典型Netscape 4.51浏览器返回的user_agent名字类似 Mozilla/4.51 - (Win95 ; I)。

20.6 课时小结

在本学时中，我们介绍了从服务器中检索 Web页时程序运行的情况，并且简要地讲述了 HTTP协议。还介绍了如何通过链接来调用CGI程序，并将参数传递给程序，这可以用于SSI。此外，还介绍了如何进行HTTP通信，以便执行重定向操作，并获取关于浏览器和服务器的信息。

20.7 课外作业

20.7.1 专家答疑

问题：SSI示例代码似乎无法运行，为什么？

解答：SSI代码不能运行的原因很多。首先，应该检查你的 Web浏览器是否支持SSI，并非所有的Web浏览器都支持SSI。其次，应该确保Web服务器激活了SSI特性。第三，应该确保你的HTML文件拥有正确的扩展名，以便激活SSI。你可以与服务器管理员取得联系，了解上述信息。最后，应该确保你的HTML SSI标号使用的句法的正确。

如果你使用<!--#exec cgi-->标记，应该确保你在不使用SSI来运行程序时你的CGI程序运行正确。

在Web页已经加载的情况下使用浏览器中的“view source（查看源代码）”选项，就能够知道服务器是否正在执行你的SSI程序。如果你看到Web页源代码中的SSI标记，服务器就不能识别和分析它们。

问题：Telnet示例代码无法运行，为什么？

解答：如果Telnet未能建立连接，那么应该确保你是针对Web服务器的名字来使用Telnet的，并且使用的端口是正确的，也许它是端口80。你必须查看Telnet客户程序的文档，以便正确地设置端口号。

另一个常见问题是无法看到自己键入的字符。有些Telnet客户程序能够将你键入的字符反馈给你，有些则不能。请不必对此担心，你只需要认真进行操作就行了。这些字符必须认真发送。当你键入GET行后，务必按两下Enter键。

20.7.2 思考题

1) 下面这个URL能够按照你的期望运行吗？

```
<A href="/cgi/foo.pl? name=Ben Franklin&Job=printer">
```

- a. 是。
- b. 否。你不能像这样将两个参数传递给一个CGI程序。
- c. 否。名字Ben Franklin中的空格是不允许的。

2) 服务器端的包含程序由什么来进行处理和展开？

- a. 浏览器。
- b. Web服务器。
- c. 操作系统。

20.7.3 解答

- 1) 答案是c。你应该使用转义符正确地隐藏空格和其他特殊字符。
- 2) 答案是b。Web服务器负责将SSI HTML标记转换成它们的值，然后将它们发送给浏览器。

20.7.4 实习

- 使用Telnet客户程序，连接到你喜欢的Web站点之一，并设法人工检索Web页。

China-pub.com

下载

China-pub.com

下载

第21学时 cookie

在第19学时中，我们讲述了如何使用HTML中的隐藏域使你的Web浏览器记住各个Web之间的信息。你必须理解这个进程，因为从CGI程序的一个实例到另一个实例，有时需要在它们之间传递信息。进行这项操作的唯一方法是将一些信息存储在浏览器中。

将信息存储在浏览器中的另一个方法是使用HTTP的cookie。正如它的名字所表示的那样，HTTP Cookie是指在HTTP连接期间浏览器与CGI程序之间传递的信息。使用Cookie，可以比使用HTML隐藏域更加灵活地用浏览器来存储信息。

在本学时中，你将要学习

- 什么是cookie。
- 如何编写和检索cookie。
- 如何处理和避免cookie的常见问题。

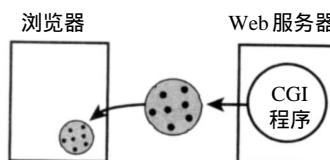
21.1 什么是cookie

可以将cookie视为电影院的入场券。你可以到电影院购买一张入场券，以便在以后的某个时间拿着入场券到电影院去看电影。看完电影你就可以离开电影院，往回家路上走，买一点爆玉米，并做你喜欢做的任何事情。当你准备看电影时，你向电影院的收票员出示电影票。收票员并不知道你如何、何时和为何购买电影票，但是，只要你持有电影票，收票员就允许你进入电影院。电影票使持票人有权在以后进入电影院去看电影。

HTTP cookie只不过是CGI程序要求浏览器持有的一个信息包。这个信息包可以由另一个CGI程序或原来的程序在任何时候回收。当有人要检索正常的HTML Web页时，cookie甚至可以重新传回给服务器。cookie可以包含任何种类的信息，比如关于多页Web窗体的信息、访问信息、用户喜欢的信息等。

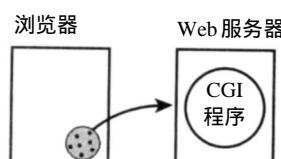
每当CGI程序要求创建cookie时，cookie可以从服务器传送到浏览器（见图21-1），这个进程称为安装cookie。

图21-1 cookie从CGI程序
传送到浏览器



CGI程序可以在晚些时候回收，以便检索存储在cookie中的信息，如图21-2所示。

图21-2 浏览器将 cookie 送
回到服务器



cookie因何而得名

在计算机界，cookie是个非常老的术语。它是指例程或程序之间传递的任何一组信息，它使cookie的持有者能够执行某项操作。某些类型的 cookie称为神秘的 cookie，因为它们包含的数据非常神秘，只有 cookie的发送者和接收者才能理解其含义。CGI cookie并不神秘。

21.1.1 如何创建cookie

若要创建cookie，你可以使用CGI函数cookie。cookie函数的句法如下：

```
$cookie_object=cookie( -name => cookie_name,          # Optional
                      -value => cookie_value,
                      -expires => expiration_date, # Optional
                      -path   => path_info,       # Optional
                      -domain=> domain_info,    # Optional
                      -secure => true/false     # Optional
);
```

cookie函数以一种特殊的方式使用参数。调用 cookie时使用的每个参数都带有名字。实际上，在Perl中以这种方法将参数传递给函数是非常方便的，因为你不必记住参数的顺序，它们将按你使用它们时的顺序进行命名。

当你用这个句法来调用 cookie函数时，该函数便返回一个 cookie（该cookie应该存放在一个标量变量中），然后该cookie可以被赋予CGI模块的header函数，以便发送给浏览器。创建 cookie时必须要的唯一参数是 -value。-name参数允许同时将若干个 cookie发送给浏览器，而检索时则可以单个检索，也可以成组检索。其他参数如 -expires、-path、-domain和-secure等，将在下一节介绍。

CGI模块中的header函数负责管理将 cookie发送给浏览器的实际操作。这意味着必须使用 cookie函数来创建 cookie，然后紧接着就调用 header函数。在cookie和标题发送之前，不应该将任何其他种类的数据发送给浏览器。

若要使用CGI程序创建一个cookie并将它发送给浏览器，你可以使用类似下面这样的CGI程序：

```
#!/usr/bin/perl -w
use CGI qw(:all);
use strict;

my $cookie=cookie(-name => 'Sample',
                  -value => 'This cookie contains no MSG');

# Transmit the cookie to the browser
print header(-cookie => $cookie);
```

当上面这个代码段运行之后，浏览器上就安装了一个称为 sample的cookie。该cookie包含了“ This cookie contains no MSG（该cookie不包含任何消息）”这样一个信息。



实际上该 cookie并没有安装。浏览器可以因为许多原因而拒绝接受某个 cookie。请参见本学时后面部分中的“ cookie存在的问题”这一节。

若要在你的CGI程序中从浏览器中检索 cookie，可以使用相同的cookie函数。如下面的例子所示，如果不带任何参数，cookie函数返回浏览器拥有的服务器的一个 cookie列表：

```
@cookie_list=cookie(); # Returns names of all cookies set
--or--
```

```
# Returns the value for a particular cookie
$cookie_value=cookie($cookie_name);
```

按照默认设置，当 cookie 安装在浏览器上之后，它将返回给驻留在同一个服务器上的任何 CGI 程序。也就是说，只有安装 cookie 的服务器才能检索这些 cookie。若要查看以前创建的 Sample cookie，可以使用另一个 CGI 程序：

```
#!/usr/bin/perl -wT
use CGI qw(:all);
use strict;

print header(); # Print out the standard header
print "Sample's value: ", cookie('Sample'), "<P>";
```

上面的代码段使用带有一个参数的 cookie 函数，这个参数就是你想查看其值的 cookie 的名字。该值被检索并输出。

cookie 应该被浏览器保留到浏览器运行终止。当浏览器重新启动时，cookie sample 将不复存在。如果你想创建一个比较永久的 cookie，请参见本学时后面部分中的“设置 cookie 终止运行的时间”这一节。



大多数浏览器都配有一个选项，用于在 cookie 被安装时查看这些 cookie。在 Netscape 中，你可以在 Advanced 选项卡上的 Preferences 选项下找到查看 cookie 的各个选项。在 Internet Explorer 中，这个选项出现在 Internet Options 对话框的 Advanced 选项卡上，还有一个单选按钮可用于控制你是否可以在安装 cookie 时查看它们。

21.1.2 举例：使用 cookie

使用这个例子，你可以创建一个小程序，让用户可以使用 Web 浏览器来设置他查看的 Web 页的颜色。该程序实际上能够同时执行若干项操作：

- 1) 通过查看程序的各个参数，以便观察默认背景色的变化。
- 2) 用正确的背景色在浏览器上设置 cookie。
- 3) 将 Web 页的背景色设置为正确的颜色。
- 4) 显示一个 CGI 窗体，使你能够改变其颜色。

程序清单 21-1 包含改变颜色的程序。

程序清单 21-1 ColorChanger 程序的完整清单

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use CGI qw(:all);
4:  use CGI::Carp qw(fatalsToBrowser);
5:  my($requested_color, $old_color, $color_cookie)=("", "");
6:  $old_color="blue"; # Default value
7:  # Is there a new color requested?
```

```

8:   if (defined param('color')) {
9:     $requested_color=param('color');
10:   }
11: # What was the old color, if any?
12: if (defined cookie('bgcolor')) {
13:   $old_color=cookie('bgcolor');
14: }
15: if ($requested_color and ($old_color ne $requested_color)) {
16:   # Set the cookie in the browser
17:   $color_cookie=cookie(-name => 'bgcolor',
18:                         -value => $requested_color);
19:   print header(-cookie => $color_cookie);
20: } else {
21:   # Nothing's changed, no need to set the cookie
22:   $requested_color=$old_color;
23:   print header;
24: }
25: print<<END_OF_HTML;
26: <HTML>
27: <HEAD>
28: <TITLE>Set your background color</TITLE>
29: </HEAD>
30: <BODY BGCOLOR="$requested_color">
31: <FORM>
32: <SELECT NAME="color">
33:   <OPTION value='red'>Red
34:   <OPTION value='blue'>Blue
35:   <OPTION value='yellow'>Yellow
36:   <OPTION value='white'>White
37: </SELECT>
38: <INPUT TYPE=SUBMIT VALUE="Set the color">
39: </FORM>
40: </BODY>
41: </HTML>
42: END_OF_HTML

```

第7~10行：如果该程序作为CGI窗体的目标程序来调用，那么param('color')函数值返回一个定义的值，即一个新颜色。否则，它不返回任何值，\$requested_color则保持未设定状态。

第12~14行：这些行用于检索名叫bgcolor的cookie。它可能存在，也可能不存在。如果它不存在，那么它存放在\$old_color中，这是上次保存到cookie中的屏幕颜色值。

第15~19行：如果颜色已经改变（即cookie的值与新值不一致），那么新cookie必须用新值进行设置。

第20~24行：否则，输出一个纯标题，不带cookie。请记住，浏览器将无限期保留以前的cookie。

第25~42行：这些代码行用于创建一个标准HTML窗体。不过请注意第30行，在这一行上，被取代的颜色被送入HTML输出。

21.1.3 另一个例子：cookie查看器

程序清单21-2中列出的一个非常短的程序是个cookie查看器，它用于帮助你调试使用cookie的CGI程序。它列出了存储在Web浏览器上的所有cookie，这些cookie恰好来自同一个Web服务器。

程序清单21-2 Cookie查看器

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use CGI qw(:all);
5:
6:  print header();
7:
8:  print "Cookies set that can be seen:<P>";
9:
10: foreach my $cookie (cookie()) {
11:     print "Cookie name: $cookie <BR>";
12:     print qq{Cookie value: "}, cookie($cookie), qq{"<P><HR>};
13: }
```

第10行：用cookie函数检查所有cookie的名字，并赋予\$cookie，每次检索1个cookie。

第11~12行：输出每个cookie的名字和值。

该cookie查看器运行时，可以获取使用cookie()函数能够得到的所有cookie的列表，然后对这些名字迭代运行cookie，输出每个cookie的名字和值。

21.2 高级cookie特性

cookie的基本概念简单明了，你将cookie赋予浏览器，过一会儿浏览器又将它送回给服务器。不过cookie的基本特性并不止此。可以将cookie设置为可以存在较长的时间，这种cookie称为永久性cookie。你可以让这些cookie只返回到另一个特定的URL，它们能够指明关于你的连接的安全程度之类的信息。

21.2.1 设置cookie终止运行的时间

到现在为止，你在浏览器上安装的cookie都是临时的。一旦浏览器关闭，cookie就消失。当你使用cookie将值保存在窗体上的多个页中（而不是保存隐藏的HTML值）时，使用临时cookie是完全合适的。当一个新浏览器启动时，你不想让cookie返回给服务器，因为用户不会从中间开始填写窗体，他将再次从头开始时填写。

在有些情况下，你可能希望cookie能够保留更长的时间。也许你想在浏览器关闭和重新启动之后使cookie持续数天、数周、数月时间。用Perl的CGI模块来创建这种cookie是非常容易的。

若要为cookie设置一个终止日期，可以在创建cookie时使用-expire选项。-expire选项必须后随一个想使cookie终止运行的日期。可以如表21-1所示用若干种格式设置这个日期。

表21-1 cookie的终止日期格式

格 式	示 例	含 义
秒数	+30s	从现在起30秒后终止
分钟数	+15m	从现在起15分钟后终止
小时数	+12h	从现在起12小时后终止
月数	+6M	从现在起6个月后终止
年数	+1Y	从现在起1年后终止
	now	cookie立即终止运行
任何负时间值	-10m	cookie立即终止运行
一个特定时间	Saturday,28-Aug-1999 22:51:05 GMT	

当设定一个特定时间时，必须完全使用表 21-1中列出的时间格式。所有其他的各种设置值都是指从当前时间起的时间偏移量。系统将为你计算出完全合格的时间值，然后发送给浏览器。

下面这个小程序用于在浏览器上安装一个将在 8天后终止运行的cookie：

```
#!/usr/bin/perl -w
use CGI qw(:all);
use strict;

my $cookie=cookie(-name => 'Favorite',
    -value => 'soft oatmeal raisin cookies',
    -expires => '+8d' );

# Transmit the cookie to the browser
print header(-cookie => $cookie);
```

21.2.2 cookie的局限性

要使cookie能够永久运行是做不到的。这就是说，如果将一个 cookie发送给浏览器，希望从现在起该 cookie能够在数周、数月或者数年内保持运行，那么你一定会大失所望的。

当你读到后面的“ cookie存在的问题”这一节内容时，就会知道浏览器并不是必须将 cookie存储起来的。实际上，它们根本不接受你的 cookie，它们并不通知你这些 cookie并没有保留起来。

浏览器可以随时清除它们的 cookie，以便为来自其他站点的新 cookie腾出地方，或者根本毫无理由就这样做了。有些浏览器允许用户编辑 cookie，或者添加新的cookie。

用户可能不小心删除 cookie，也可能故意将cookie删除掉。如果用户安装了浏览器或操作系统的 new 版本，cookie就会被清除，或者放到别的什么地方。只要改用另一种浏览器， cookie就会“不知去向”。当浏览器尚未激活时， cookie通常存放在一个文件中，该文件可以供用户编辑，删除，或者遭到损坏。



如果你有兴趣的话，我们可以告诉你，大多数浏览器是在没有激活时将 cookie存放在文件中的，这些文件通常是文本文件，你可以使用编辑器查看这些文件。Netscape将cookie存放在用户主目录下的 cookies.txt文件中（不同的系统下该目录将各不相同）。Internet Explorer将cookie存放在\Windows\ Cookies下。

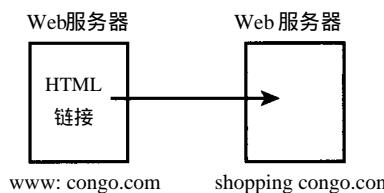
因此，将重要信息存放在一个 HTTP cookie中真的不是个好主意。你想永久存放在 cookie中的任何信息不应该被轻易改变位置，这些信息包括用户喜欢的信息，输入指定 Web页的可替换项目关键字，上次刚刚访问的信息等。

21.2.3 将cookie发送到其他地方

按照默认设置， cookie只能送回到曾经发出 cookie的服务器。有时，你希望将 cookie送回到服务器，但有时你并不希望如此。以神秘的 Web站点Congo.com为例，这个销售书籍的 Web站点拥有两个Web服务器，即www.congo.com和shopping.congo.com，如图21-3所示。主要的 Web站点（www.congo.com）包含公司的所有信息，可以连接到其他站点，并且最重要的是可

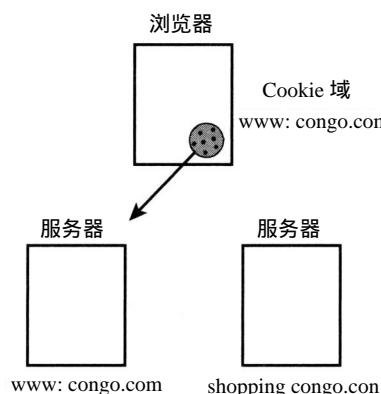
以连接到在线书店。

图21-3 两个互相连接的Web站点



www.congo.com包含一个注册用的HTML窗体 / CGI程序，使用户可以将他们的名字添加到电子邮件的地址列表，设置他们喜欢什么类型的书籍。以后，当用户浏览 www.congo.com 时，他就可以阅读关于他感兴趣的新书的信息。用户浏览器上的 cookie负责告诉 www.congo.com，应该向他介绍哪些书籍的情况（见图 21-4）。

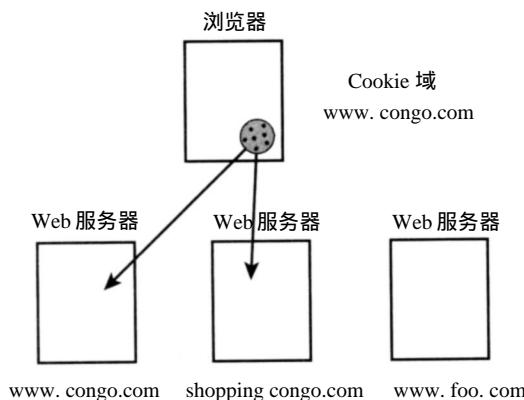
图21-4 只返回给单个 Web 站点的cookie



问题是当用户从 www.congo.com 转到位于 shopping.congo.com 站点上的在线书店时，cookie没有被发送到 shopping.congo.com 服务器。HTTP cookie只返回给原先发送 cookie 的这个服务器。如果 www.congo.com 发送了该 cookie，它并不发回给 shopping.congo. com。

那么你应该怎么办呢？如果让用户填写另一个首选项窗体，并且从 shopping.congo.com 给他发送一个新 cookie，那将是不切实际的。更好的办法是限制这个 cookie只能使用一个特定的域名。例如，当原始 cookie 从 www.congo.com 发送出来时，可以将该 cookie 送回给任何 congo.com Web 站点，如图 21-5 所示。

图21-5 返回给两个 Web 站点的cookie



若要进行上述操作，可以在创建 cookie 时使用带有 -domain 参数的 cookie 函数：

```
$cookie=cookie( -name => 'preferences',
    -value => 'mysteries, horror',
    -domain => 'congo.com');
print header(-cookie => $cookie);
```

在上面这个代码段中， cookie \$ cookie得以创建，并且限制为 congo.com域。任何 Web服务器，如果其主机名以 congo.com为结尾，将使它的 cookie由浏览器返回给该服务器。

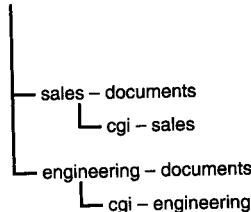


域的参数至少必须由两个部分组成，并且不能是绝对的顶层域，即 .com 或.net。这样，就可以避免浏览器将 cookie从一个.com域移植到另一个 .com 域。

21.2.4 限制cookie返回到的位置

将cookie限制为只能返回到某个服务器，这也是可能的。当你创建一个 cookie时，按照默认设置，该cookie可以返回到Web站点上的任何URL，包括非CGI URL。例如，如图21-6所示的汽车销售Web站点的组织结构。

图21-6 一个多用户 Web站
点的目录树结构



让sales（销售）CGI程序和engineering（工程设计）CGI程序驻留在不同的目录中，是有意义的。如果sales CGI程序准备建立一个 cookie，那么engineering CGI程序便将它接收过来，反过来也一样。这样的结果是人们所不希望的，为两个站点同时编写 CGI程序的开发人员必须采取协调措施，以确保不会重复使用对方的 cookie名字。

为了解决这个问题，可以使用 cookie函数的-path选项。该选项用于指明 cookie应该返回到的路径名（相对于URL顶层的路径名）。例如，若要发送只返回到 sales CGI程序的一个 cookie，可以使用下面的代码段：

```
# Cookie only visible to sales CGI programs
$cookie=cookie( -name => 'profile',
    -value => 'sedan, luxury, 2-door',
    -path => '/cgi-sales');
print header(-cookie => $cookie);
```

按照默认设置， cookie返回到服务器上的每个站点，就像已经使用了选项 -path=> ‘ / ’一样。若要限制只能返回到一个 CGI程序，可以在-path选项中使用 CGI程序的URL：

```
# Return the cookie only to this program
$cookie=cookie( -name => 'profile',
    -value => 'sedan, luxury, 2-door',
    -path => script_name());
print header(-cookie => $cookie);
```

上个学时中我们讲过， CGI模块中的script_name函数能够返回当前CGI程序的部分 URL。这可以有效地创建这样一个 cookie，它只返回到在浏览器上安装该 cookie的程序。

21.2.5 带有安全性的cookie

有些cookie你可能只想在一条安全的连接上传输它们。使用 cookie函数的 -secure参数，就可以只在连接是安全的时候从浏览器发送 cookie。下面的代码用于将一个包含账号的 cookie发给浏览器。包含此类敏感信息的 cookie只能在安全的连接上发送。

```
# Caution! Send this only over an https connection
$cookie=cookie( -name => 'account',
                 -value => '00-12-3-122-1313',
                 -secure => 1);
print header(-cookie => $cookie);
```

以后，如果你要检索该cookie，只要像平常那样使用 cookie函数即可。如果连接是安全的，而且该cookie是在该浏览器上，那么该浏览器就可以在需要时将 cookie发回给服务器。

```
# Fetch the account number from the browser.
$account_number=cookie('account');
```

你不应该依赖这个方法来检查连接是否安全，也不应该依赖账号的准确性。请记住，用户负责控制Web浏览器及其cookie文件。cookie可以在不安全的连接上发回给服务器，甚至可以有一个无效号码。

21.3 cookie存在的问题

在你将cookie投入应用之前，应该知道与 cookie相关的一些问题。由于这些原因和将来可能产生的其他原因，你应该认真设计你的Web页和CGI程序，使得cookie完全成为可以选择的选项。

例如，如果你使用 cookie来存放用户的首选项，那么倘若 cookie无法使用，你应该使用一组默认首选项。编码时请采取相应的防范措施。

21.3.1 cookie的生存期很短

本学时中多次讲到，cookie的寿命很短。Cookie可以从用户的系统中删除，可以由用户编辑，也可以毫无理由地被浏览器甩掉。

浏览器可以接受 cookie，将它使用一会儿，然后毫无理由就将它忘掉。如果你使用 -expire 选项安装了一个永久性 cookie，浏览器仍然可以甩掉这个 cookie，并且根本不通知用户。

21.3.2 并非所有浏览器都支持cookie

并非所有浏览器都支持 HTTP cookie，这是千真万确的事实。适用于 HTTP和Web信息传输的Internet标准并不能保证浏览器必须支持 cookie。

并不是说大多数浏览器都不支持 cookie，大多数浏览器是支持 cookie的。Netscape（自从1.1版以来），Internet Explorer（所有版本），Lynx,Opera，以及大多数流行的 Web浏览器都支持cookie。在大多数浏览器中，有一个选项可供用户关闭对 cookie的支持。

即使你使用 CGI模块的 user_agent函数，确定你想使用的浏览器应能支持 cookie，也不要完全指望它。

21.3.3 有些人不喜欢cookie

这一节的标题也许很难理解，为什么世界上竟然有人不喜欢 cookie呢？

在Web上冲浪实际上是一种匿名活动。正如你在上一学时中看到的那样，当浏览器要求检索一个Web页时，这个检索请求是在真空中发生的。服务器不一定知道浏览器所在的位置，也不知道该浏览器上次曾经要求检索过该站点上的一个Web页。



请记住，一个浏览器不一定代表一个用户，一个浏览器可以被一个家庭、网吧、Internet网吧或公共访问点（如图书馆）中的许多人共享。为一个人安装（或修改）一个cookie，实际上也为若干人安装了cookie。

cookie可以用来跟踪人们曾经访问过某个站点的哪个位置以及他们曾经点击过什么。如果你非常在乎隐私问题，那么这个情况你应该注意。

例如，前面的“将 cookie发送到其他地方”这一节中我们提到的一个虚构在线书店congo.com能够跟踪Web冲浪者点击了哪些书籍以便了解其详细信息，并使用该信息编写符合读者需要的书目，提供给Web冲浪者。

从表面上看，这些特性很好。但是对于那些想要维护隐私权的人来说，这会带来两个问题。首先，现在有一个机构负责跟踪 Web冲浪者感兴趣的是什么种类的书籍。如果这些信息与Web冲浪者的名字和地址有关（也许这些信息是从 congo.com共享信息的另一个站点的填写式窗体中获得的），那么Web冲浪者将会收到与他选购书籍相关的垃圾邮件。与 cookie搜集站点共享的信息越多，就能获得关于 Web冲浪者更详细的信息。

除了隐私问题外，如果 Web冲浪者查看的头两本书属于“计算机”书籍，Web站点就会停止向Web冲浪者提供“传奇”类和“烹饪”类书籍。Web站点将把Web冲浪者“转移”到他们想要的书籍类别。



你会惊奇地发现 cookie是多么频繁地用在你的浏览器上搜集和存储信息。请打开你的浏览器上的cookie确认特性，以便访问流行的Web站点。

为了避开对cookie的使用，人们想了多办法。支持 cookie的Web浏览器均配有关闭 cookie的特性，有些浏览器在安装 cookie时允许你查看这些 cookie。可以使用某些辅助软件包对发送到浏览器和浏览器返回的 cookie进行筛选，还可以对它们进行编辑。Web站点的设计使你可以对其他Web站点进行匿名冲浪，而 cookie不会搜集关于你的信息。

总之，有些人将HTTP cookie视为侵犯隐私权的一个特性，因此你在使用cookie时应该慎重。

21.4 课时小结

在本学时中，我们全面介绍了如何使用HTTP cookie在浏览器上存储信息，供别的CGI程序在以后使用。还介绍了按照预定时间使cookie终止运行，仅为特定Web服务器激活，或者为特定目录激活cookie等特性。最后，讲述了不使用cookie的许多理由以及使用cookie会带来的问题。

21.5 课外作业

21.5.1 专家答疑

问题：我应该如何将多个项目放入一个HTTP cookie？

下载

解答：最容易的方法是将多个项目组合在单个 cookie中，用域分隔符将各个项分开，如下例所示：

```
$cookie=cookie(-name => 'preferences',
               -value => 'bgcolor=blue,fgcolor=red,banners=no,java=no');
```

然后，当你检索cookie时，可以使用Split将各个项目分开：

```
$cookie=cookie('preferences');
@options=split(/,/, $cookie);
# Now, make a hash with the option as the key,
# and the option's value as the hash value
foreach $option (@options) {
    ($key,$value)=split(/=/, $option)
    $Options{$key}=$value;
}
```

问题：如何使用cookie来跟踪用户在Web页上点击了哪些链接？

解答：在解答这个问题之前，必须指出，有些人将这种跟踪视为是侵犯他人的隐私权。说明这一情况后，再来说明跟踪的一般方法：

1) 编写你的< A HREF >链接，将它们纳入一个CGI程序，将真实的目标URL作为参数来传递：

```
<A
  HREF="http://server/cgi/redirect.pl?target=http://www.congo.com">Congo
</A>
```

2) 上例中的redirect.pl程序应该使用CGI模块的param函数，以便从参数target中获得真实的URL（http://www.congo.com）：

```
target:
$target_url=param('target');
```

3) 然后使用该值中的目标URL创建一个cookie，其名字你可以在以后查看，如下所示：

```
$tracking_cookie=cookie(-name => 'tracker',
                        -value => $target_url,
                        -expires => '+1w');
```

4) 然后将重定向的项目与cookie一同发送给浏览器：

```
print redirect(-uri => $target_url,
               -cookie => $tracking_cookie);
```

以后，当浏览器返回到你的Web站点时，你就可以查找名字为tracker的cookie，它包含了用户退出你的站点时访问过的URL。

问题：我在传送cookie时能够将浏览器重定向到另一个Web页吗？

解答：当然可以。CGI模块的redirect函数也能像header函数那样带有一个-cookie参数。

```
my $cookie=cookie(-name => 'target',
                  -value => 'redirected to foo.html');
print redirect(-uri => "http://www.server.com/foo.html",
               -cookie => $cookie);
```

21.5.2 思考题

1) 用cookie来长期存储信息，为什么有时会失败？

- a. 浏览器会“甩掉”cookie的信息。
- b. 软件更新时cookie可能丢失。

c. 用户可能关闭浏览器对 cookie 的支持。

2) 若要使 cookie 在一周后终止运行 , cookie 函数的 -expire 选项应该使用什么参数 ?

- a. +7d
- b. +1w
- c. +10080m

3) 为什么有些人认为 cookie 会侵犯隐私权 ?

- a. cookie 可以用来跟踪用户点击的链接。
- b. 被跟踪的 cookie 信息可以共享 , 以便建立关于用户的档案资料。
- c. cookie 信息可用于将某些类别的信息 “ 传递 ” 给用户。

21.5.3 解答

- 1) 3 个答案均成立。
- 2) a 和 c 均成立。参数 +1w 无效。
- 3) 3 个答案均成立。

21.5.4 实习

- 扩展背景色修改程序 , 以便设置前景色和字体 , 并随机选定一个图形 , 以便显示在 Web 页上 , 方法是编辑 < IMG > 标记的目标对象。

China-pub.com

下载

第22学时 使用CGI程序发送电子邮件

毫无疑问，在你进行 Web冲浪时，要填写一个窗体，以便在以后用来发送电子邮件。这些窗体常常用作信址列表、故障报告、客户支持、爱好者邮件和其他各种可以想像到的用途。

在本学时中，我们将要介绍如何用 Perl程序发送邮件，并且讲述一个简短的 Web页示例，你可以用它来生成电子邮件。我们将使你能够创造性地使用这个 Web页。

在本学时中，你将要学习：

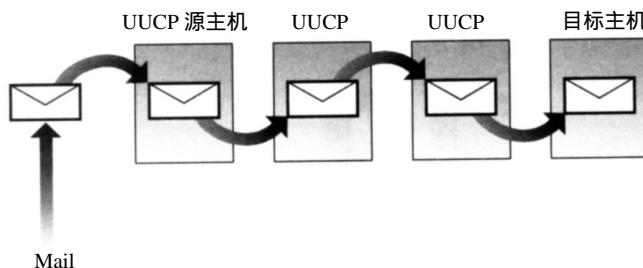
- 关于如何运行Internet电子邮件特性的简单介绍。
- 如何在UNIX和非UNIX系统下发送邮件。
- 如何建立发送邮件的Web窗体。

22.1 Internet邮件入门

在你将编程技巧用于以 Perl来发送电子邮件之前，首先必须学习一些关于电子邮件特性如何在Internet上运行的一些知识。

在Perl问世之前，在美国的国家计算机安全委员会（ NCSA ）尚未注意到 Web的远大前景并且调制解调器的速度还比较慢的时候，全球的许多人就已经在使用电子邮件在所谓的 UNIX至UNIX拷贝（ UNIX-to-UNIX copy, UUCP ）的系统上进行通信了。当你在这个老式系统上发送电子邮件时，本地系统把你的电子邮件封装好，然后转发给系统链中的下一个系统，下一个系统又将电子邮件封装好，转发给下一个系统，如此传递下去。线路上的每个系统都要给邮件添加一点信息，表示它对邮件进行了处理，然后传递下去，如图 22-1所示。

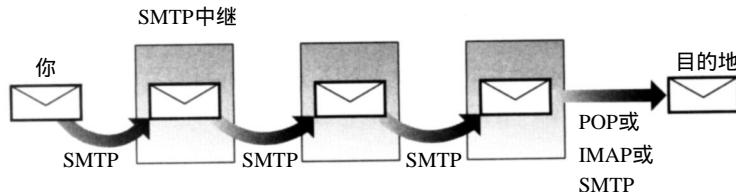
图22-1 将邮件从一个系统
传递到下一个系统



很明显，这种邮件传递的方法可以称为存储与转发法。后来UUCP系统被别的方法所取代，不过存储与转发的基本方法仍然没有变。当你从你的 PC发送电子邮件时，另一个系统负责接收该邮件，再将它转发给另一个系统，然后该系统又将邮件转发给下一个系统，直到最后由目标系统接收到邮件为止。

不过，如今这些协议完全发生了变化。目前最常用的方法是使用简单邮件传输协议（ Simple Mail Transport Protocol, SMTP ）将邮件发送到系统链上（见图 22-2）。若要检索邮件，连接的目标端通常使用邮局协议（ Post Office Portocol, POP ）或 Internet邮件访问协议（ Internet Message Access Protocol, IMAP ）。下面用于发送电子邮件的协议是 SMTP。

图22-2 发送电子邮件时使用不同的协议



22.1.1 发送电子邮件

若要发送电子邮件，需要两样东西，即邮件传输代理或 SMTP中继主机。

遗憾的是，它们都是很难理解的术语，不过下面将对它们加以解释。

邮件传输代理（Mail Transport Agent, MTA）是驻留在你的计算机上的一个程序，它通常是你操作系统所配备的一个程序，负责接收电子邮件并正确地将它们转发。当你的操作系统安装时，MTA通常已经作好正确的配置。UNIX系统上的常用MTA称为sendmail。sendmail程序负责取出一个电子邮件并确定如何将它传递到目的地。

若要在UNIX下发送电子邮件，请在命令行上使用下面这个语句：

```
$ /bin/echo "Subject:Test\n\nHello, World!" | sendmail foo@bar.com
```

上面这个代码段将一个短邮件发送到 foo@bar.com。sendmail程序负责为你解决所有难以处理的工作，比如决定使用哪个邮件中继主机，处理被拒绝的返回邮件等。

如果你使用Microsoft Windows或Macintosh操作系统，那么你将不具备内置的MTA。不过Perl模块使你能够直接发送邮件。Net::SMTP模块可以在没有介入的MTA的情况下发送邮件，但是你必须知道你的SMTP中继主机的名字。这个名字是用于发送邮件的“邮件主机”的主机名，当你用你的帐户进行登录时，你将被赋予该主机名。请索取中继主机的名字，并将它写在某个地方，以后你会用到它。



你可以使用不同的“邮件主机”，以便发送和接收邮件。本学时中你需要发送邮件的主机名。

请记住，依靠SMTP中继的程序必须将正确的中继主机内置于软件之中，否则该进程将不能运行。



正确的“SMTP中继主机名”取决于你从何处发送你的邮件。如果你从家中发送邮件，那么你的家庭Internet服务提供商（ISP）帐户为你赋予一个SMTP中继主机名。如果你用租用的Web服务器上的帐户发送邮件，那么就需要该服务器的中继主机的名字。当邮件从中继主机并不知道的一个系统发送过来，邮件中继主机便拒绝转发该邮件。

22.1.2 发送邮件时首先应该注意的问题

在下一节中，我们将要介绍一个新函数，即 `send_mail`，使用这个函数，你就能够用Perl程序发送电子邮件。这个函数虽然非常有用，但同时它也有很大的危险性。将邮件发送给某个人，将会在一定程度上侵犯他的隐私权。你会要求邮件的收件人在你的邮件上耗费一定的

时间和磁盘空间，还会要求你与收件人之间的每个系统为你中继该邮件。对于一个完全陌生的人来说，这样做是很不合适的。

下面是你在使用Perl或任何其他工具发送电子邮件时应该注意的问题：

- 首先使用众所周知的地址（比如你自己的地址）测试你的代码并发送一些短邮件。这时，随时都可能产生一些问题，你应该设法避免发生问题。
- 不要发送有人主动提供的商业性电子邮件。这类商业性电子邮件通常称为垃圾邮件，这类邮件已经成为Internet上的一个令人头痛的大问题。少数人喜欢接收这类邮件，而其他人的反应则不同，他们有的对垃圾邮件非常反感，有的则痛恨之极。发送此类邮件的企业将会成为许多人唾骂的对象。当你得到一个邮件地址后，应该问一问是否可以在以后向它发送电子邮件。如果有人要求从你的邮件地址列表中删除他的地址，那么你应该尊重他的要求。
- 无论对方要求还是没有要求，都不要一次就发送很长的邮件，要按适当的速度来发送。首先，你的本地邮件中继主机会因为急匆匆发送邮件而不堪重负，你的本地ISP将会终止你的帐户，以控制受损害的程度。其次，如果目标ISP因为你的邮件太大而无法承受，该ISP就会阻塞从你的域发送过来的全部邮件。如果根本无法向较大的域（如aol.com、hotmail.com等）发送邮件，那么你的日子一定不会好过，并且很可能使你的帐户与你的ISP之间的联系被中断，结果造成人们对你的指控。
- 应该提供很好的返回邮件的地址，尤其是在邮件报头中要写明这个地址。应该确保你的电子邮件的From：(或Reply To：)地址正确无误，尤其是当邮件是从一台计算机发送时更应保证地址的正确性。你可以使用Perl伪造电子邮件，但是伪造的邮件包含一个返回给你的指针。伪造的邮件会使你陷入巨大的麻烦之中。
- 请始终都使用你自己的邮件中继主机。滥用其他系统的邮件中继主机会使你的帐户迅速停用，并使你遭人指控，甚至出现更糟糕的问题。
- 不要将很长的电子邮件或者许多很短的邮件发送给靠不住的人，这称为邮件炸弹，可能导致你的帐户被停用，并引起法律上的麻烦。

上面这些建议并非全部仅仅是一些好的网上礼仪。如果违背这些原则，ISP可能将你从它的服务对象列表中删除掉，而且ISP和邮件的收件人会指控你。当注册你的ISP帐户时，ISP会告诉你，上述原则会成为中断对你提供服务的理由，并且可能让你对系统受到的损害负责。

对于你自己的行为，应该有所约束，对于你接受他们的恩惠，不要苛求。



Internet具有长期的记忆能力。真的发送过垃圾邮件的人将会被人们长久记住并遭到唾骂。一旦因为发送垃圾邮件而变得臭名昭著，要想挽回名誉是很难的。

22.2 邮件发送函数

下面各节将介绍如何编写一个Perl短函数，供你在CGI程序中用来发送电子邮件。不过这里存在一个问题。该函数运行的方式主要取决于你是否拥有本地MTA（如sendmail程序），或者是否亲自将邮件发送到SMTP中继主机。因此请预先考虑好，确定需要将下面的哪一节中的

函数用于你的特定程序。

22.2.1 用于UNIX系统的邮件函数

如果你拥有 UNIX 系统，并且 sendmail 可能已经配置好了（也许尚未配置好），那么你阅读本节内容是对的。如果你没有 UNIX 或 sendmail，只是因为好奇而阅读本节内容，这也对你有好处，不过，程序清单 22-1 中展示的函数也许对你没有多大帮助。



即使你拥有 UNIX 系统，下一节“用于非 UNIX 系统的邮件函数”也是值得一读的。下一节将介绍使用模块（即面向对象的模块）的新方法。

程序清单 22-1 send_mail 函数

```

1:  # Function for sending mail with an MTA like sendmail
2:  sub send_mail {
3:      my($to, $from, $subject, @body)=@_;
4:
5:      # Change this as necessary for your system
6:      my $sendmail="/usr/lib/sendmail -t -oi -odq";
7:
8:      open(MAIL, "|$sendmail") || die "Can't start sendmail: $!";
9:      print MAIL<<END_OF_HEADER;
10:     From: $from
11:     To: $to
12:     Subject: $subject
13:
14: END_OF_HEADER
15:     foreach (@body) {
16:         print MAIL "$_\n";
17:     }
18:     close(MAIL);
19: }
```

第6行：sendmail 的位置和它需要的参数在这里被放到一个变量中。 sendmail 程序可能位于你的系统上的不同位置，也可以带有不同的参数。

第8行：\$sendmail 中设定的 sendmail 程序启动并打开，以便对文件句柄 MAIL 进行写入操作。

第9~14行：电子邮件的报头被写入 MAIL。

第15~17行：邮件的正文被写入 MAIL 文件句柄。每行都附加了一个 \n。

若要使用该函数，只要像下面这样用 4 个参数调用它：

```

@body=("Lower mine, please.", "Thanks!");
send_mail('president@whitehouse.gov', 'owner@geeksalad.org',
          'Taxes', @body);
```

该函数的运行要求你在系统上正确安装和配置 sendmail。如果没有安装和配置，请阅读下一节“用于非 UNIX 系统的邮件函数”，那里介绍的解决方案也可以在 UNIX 下使用。

必须将变量 \$sendmail 改为你的系统上的 sendmail 程序的正确位置。它的位置通常是 /usr/lib，不过它也可以是 /usr/sbin/lib，或者你的系统上的任何其他目录。你必须花一点时间才能找到它。



如果程序的运行没有按你的期望进行，请确保你的系统上的邮件程序配置正确。可以使用 mail或pine之类的邮件实用程序来发送测试邮件。如果这些实用程序不能正确运行，那么说明 sendmail的安装很可能不正确。你必须首先解决这个问题，或者使用下一节介绍的方法来运行这些实用程序。

在程序清单 21-1中，sendmail程序是用下列选项启动的，你可以根据情况修改这些选项。

- -t 从输入数据而不是命令行中获得邮件的报头 (From、To、Subject等)。
- -oi 忽略单行程序上的“.”(圆点)。如果不使用本选项，就会中断你的邮件。
- -odq 对邮件进行排队，而不是立即将它们发送出去。如果你愿意，可以不使用本选项。但是，如果有太多的邮件要立即发送，那么你的邮件系统将会应接不暇。使用 -odq是一种很礼貌的做法。

send_mail()函数的其余部分的功能是不言自明的。

22.2.2 用于非UNIX系统的邮件函数

在没有安装 sendmail之类的内置MTA的Windows和其他操作系统下，你会遇到一些复杂的问题。MTA不是个简单的邮件传输工具，试图用几行 Perl代码就复制它的功能，是很不容易的事情。不过这是可能做到的。

首先，使用Perl模块Net::SMTP，你可以通过Perl运行的任何操作系统来发送邮件。使用该模块，你就能够非常容易地发送邮件而不会遇到太大的困难。

问题是在标准的Perl产品上并没有安装该模块。为了获得该模块，必须将它加载到Web服务器所在的系统上，或者加载到你想发送邮件的任何位置上。Net :: SMTP模块是libnet组件的组成部分，它包含各种非常有用的网络模块。Libnet组件位于本书所附光盘上。



本书的附录“安装模块”提供了相当详细的如何安装 Perl模块的指南。它讲述了如何在UNIX、Windows和Macintosh操作系统下，安装各个Perl模块。此外，如果你的系统管理员没有安装模块的公用拷贝，你还会在附录中找到如何安装模块的专用拷贝的说明。

程序清单 22-2显示了用于不带MTA的操作系统的send_mail函数。它包含某些非常奇特的新语句，你可能对它们不太熟悉。请务必要阅读后面的说明。

程序清单 22-2 用于非MTA系统的send_mail函数

```
1: # Function for sending mail for systems without an MTA
2: sub send_mail {
3:     my($to, $from, $subject, @body)=@_;
4:
5:     use Net::SMTP;
6:
7:     # You will need to change the following line
8:     # to your mail relay host
9:     my $relay="relayhost.yourisp.com";
10:    my $smtp = Net::SMTP->new($relay);
11:    die "Could not open connection: $" if (! defined $smtp);
```

```

12:
13:     $smtp->mail($from);
14:     $smtp->to($to);
15:
16:     $smtp->data();
17:     $smtp->datasend("To: $to\n");
18:     $smtp->datasend("From: $from\n");
19:     $smtp->datasend("Subject: $subject\n");
20:     $smtp->datasend("\n");
21:     foreach(@body) {
22:         $smtp->datasend("$_\n");
23:     }
24:     $smtp->dataend(); # Note the spelling: no "s"
25:     $smtp->quit;
26: }
```

第5行：引入Net::SMTP模块，使邮件的发送稍为容易一些。

第10行：Net::SMTP对象得以创建，并与正确的中继主机相连接，该主机是你在第9行上设置的。

第13~23行：电子邮件的报头和正文被发送到中继主机。详细说明请参见后面的各个Net::SMTP函数。

若要使用该函数，只需使用代表电子邮件各个部分的4个参数来调用它：

```

@body=("Lower mine, please.", "Thanks!");
send_mail('president@whitehouse.gov', 'owner@geeksalad.org',
          'Taxes', @body);
```

这个函数令你感到奇怪的第一件事情是 \$smtp=Net::SMTP->new (\$relay)；这行代码。这行代码用于创建一个称为“对象”的东西。“对象”实际上并不是一个标量，也不是哈希结构或者数组，它是个稍有不同的东西。\$smtp中的值现在代表一个到达邮件程序的连接，你可以对这个连接进行各种操作，请将它视为一个特殊种类的值，可以用它来调用与该值相关的函数。

你感到奇怪的下一件事情是 \$smtp->mail(\$from)；这行代码。->用于将一个对象连接到一个对它进行调用的函数，因此，mail是个使用上一行创建的\$smtp对象来调用的函数。

为了使用Net::SMTP模块，你并不需要理解对象语句的全部特征，只需顺便了解一下就够了。对于Net::SMTP对象，可以使用的函数包括下列几个：

- \$smtp->mail(addr) mail函数用于指明你发送邮件时使用的是什么身份。当然，有时你可以就你的身份问题撒点儿谎。
- \$smtp->to(addr) to函数用于指明你要将邮件发送给谁。如果你调用的to函数带有一个名字列表，那么每人都会收到一个邮件拷贝。这些人的名字列表不一定出现在邮件正文中，除非你亲自将这些名字明确放入邮件正文中，比如发送BCC。
- \$smtp->data()； data函数用于指明你准备发送邮件正文。
- \$smtp->datasend(data) 这个函数用于发送邮件的实际文本。你必须输出你自己的报头域（To:、From:等）。报头域，比如Date:和Received:，是自动生成的。在报头与正文之间，还必须输出一个空行——\$smtp->datasend(" \n ")。你的邮件正文跟随在这个空行的后面，并且也用\$smtp->datasend()来发送。
- \$smtp->dataend() dataend函数用于指明你已完成邮件正文的发送，在运行这个函数之

前，邮件并未发送。

- \$smtp->quit() 本函数用于断开与SMTP服务器的连接。

22.3 从Web页发送邮件

既然你有了一个邮件发送函数 send_mail()，那么从 Web 页来发送邮件的其余工作就非常简单了。只要设计一个 Web 页，编写一个 CGI 程序与它配合运行。程序清单 22-3 显示了一个电子邮件示例的 HTML 窗体。该窗体并非完美无缺，你可以随意使用自己的设计风格来改进这个窗体。

程序清单 22-3 用于发送电子邮件的 HTML 窗体

```
1. <!--assumes a program called /cgi-bin/mail.cgi exists-->
2. <FORM METHOD=POST ACTION="/cgi-bin/mail.cgi">
3. Your address: <INPUT TYPE=text NAME=return_addr><BR>
4. Subject: <INPUT TYPE=text NAME=subject><BR>
5. <BR>
6. Message:<BR>
7. <TEXTAREA NAME=body ROWS=20 COLS=60 WRAP=hard>
8. Type your message here
9. </TEXTAREA>
10. <BR>
11. <INPUT TYPE=SUBMIT VALUE="Send Message">
12. </FORM>
```

用于发送邮件的 CGI 程序并不比它大多少。下面显示了这个 CGI 程序：

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:all);
use CGI::Carp qw(fatalsToBrowser);
#
# Insert the send_mail function
# from Listing 22.1 or 22.2 here!
#
print header;
my $return=param("return_addr");
if (! defined $return or ! $return) {
    print "You must supply an e-mail address<P>";
    exit;
}
my $subject=param("subject");
if (! defined $subject or ! $subject) {
    print "You must supply a subject<P>";
    exit;
}
# Change this address to wherever you want your
# mail sent
send_mail('webmaster@myhost.com',
          param($return),
          param($subject),
          param("body"));

print "Mail sent.";
```

在上面这个代码中的小程序中，有几个问题你应该注意。首先，必须将程序清单 22-1或 22 - 2 中的 send_mail 函数插入该程序，使该程序能够运行。哪个程序清单中的函数最好，并且适合于你，就使用该程序清单中的那个函数。

其次，注意 To: 地址是通过硬连线与程序相连接的，正如 Webmaster@myhost.com 的情况那样。必须将这个地址改为你想要将邮件发送到的那个地址。该地址不是从用户那里获得的原因很简单，因为你不希望用户使用 Web 窗体将邮件发往任意的地址。如果有人滥用你的窗体，将恶意邮件发送给某个人，那么你和你的系统将成为人们指责的目标。因此这不是个好主意。

如果你希望用一个窗体将邮件发送到多个目的地，请使用下拉列表（或者单选按钮），为你提供一个地址选择表：

```
<INPUT TYPE=radio NAME=target Value=1 CHECKED>Support Department
<INPUT TYPE=radio NAME=target Value=2>Sales Department
<INPUT TYPE=radio NAME=target Value=3>Legal Department
```

然后，在你的程序中，使用下面这样的代码段：

```
$formtarget=param('target');
%targets=( 1=> 'support@myhost.com',
           2=> 'sales@myhost.com',
           3=> 'legal@myhost.com');
if (exists($targets{$formtarget})) {
    $target=$targets{$formtarget};
} else {
    $target='webmaster@myhost.com';
}
print $target;
```

无论你如何进行操作，不要让实际的 To：地址从窗体传递过来并用在你的程序中。请传递一个没有问题的值（在上面的例子中是 1 至 3），并在你的 CGI 程序中对该值进行相应的转换，即使看起来不可能，也要允许传递不正确的值（上面的例子中的 else 语句）。

核实电子邮件地址

也许你已经发现 CGI 程序并不试图确定用户输入的电子邮件地址是否有效。它这样做是很有理由的，因为它无法确定该地址是否有效。

这个原因一定会使你大吃一惊。

设计 Internet 上的电子邮件系统的要求之一是要能够了解目的地址是否有效。然而这是不可能的。

困难源于本学时开头介绍的程序清单 22-1 和 22-2。从发送邮件系统的角度来看，它无法看到邮件传输链的结尾环节。它必须将邮件全部传递给传输链上的第二个系统，第二个系统又将邮件传递给第三个系统，以此类推。这些“传递”过程的延迟时间是很重要的，更重要的是，发送邮件的系统在将邮件送出去后就无法控制邮件了。

标准的解决办法是设法清除掉显然无效的地址，无法确定是否有效的地址则属例外。电子邮件地址的 Internet 标准（RFC-822）有一个标准电子邮件地址的模板。但是，有些符合 RFC-822 标准的有效地址实际上是无效的，而有些不符合 RFC-822 标准的地址却是有效的、可以传递邮件的地址。

编写对电子邮件地址进行匹配的正则表达式是不行的。例如，表达式 `/^[\w.-]+@\([\w,-]\.\.)+\w+$/` 看上去是可行的，它甚至与 `me@somewhere.com` 这个地址相匹配。但是，它拒绝下面这个完全有效的电子邮件地址：

```
*@qz.az  
clintp!sol2!westwood@dec.net  
relay@me@host.com  
"barney&fred"@flintstones.net
```

与符合 RFC-822 标准的电子邮件地址相匹配的一个正则表达式长达 4700 个字符，因为太长，所以本书没有将它列出，你也很难键入。同时它也无法与 Internet 上的每个传输邮件的地址相匹配。

那么究竟怎么办呢？

若要确定电子邮件地址是否有效，唯一的办法是将一个邮件发送到该地址，然后等待对方的答复。如果由于某个原因，你希望确保对方地址上有人（比如将来将邮件发送给他，因为他要求发送），请发送一个电子邮件，要求他回答。当对方的答复返回时，就知道你发送了一份有效的电子邮件。

22.4 课时小结

在本学时中，我们介绍了如何从 Web 页发送电子邮件。同时，介绍了 `send_mail()` 函数的两个版本，它们可以用在任何 Perl 程序中来发送电子邮件。我们还讲述了 Internet 电子邮件的基础知识以及基本的电子邮件礼仪。

22.5 课外作业

22.5.1 专家答疑

问题：能不能使用从浏览器中搜集到的信息来获取 Web 冲浪者的电子邮件地址？

解答：虽然能够这样做看起来是很好的（它可以消除获取电子邮件地址时的错误），但这是不可能的。浏览器并不包含用户的电子邮件地址。CGI 模块中的 `remote_host` 函数返回的值实际上并不是用户接收电子邮件时使用的地址。如果你使用安全的 Web 事务处理，那么 `remote_user` 函数也许不是用户的电子邮件地址中的“名字”部分。同时请记住，浏览器可能提供某些不准确的此类信息，Netscape 和 Internet Explorer 的某些插件也会这样说谎。

另外，用户可能使用图书馆、朋友家、办公室或网吧中的 Web 浏览器，因此浏览器的地址甚至与用户的电子邮件地址并无关系。

问题：我能核实电子邮件地址吗？

解答：你可以试试。例如，大多数最新的电子邮件地址包含 @（at 符号），你可以用它进行测试。但是，本地计算机（例如 `postmaster`、`root`）上的计算机不需要 @。

问题：我试着运行 CGI 电子邮件程序，但在消息中出现“From nobody.....(来自无人.....)”这行文字，为什么？

解答：是这样的：`sendmail` 程序记录了电子邮件发送者的用户 ID。实际上，电子邮件的发送“人”是 Web 服务器本身。Web 服务器常常以一个特殊用户 ID——`nobody`、`Web`、`httpd` 或 `root` 来运行，该地址记录在电子邮件报头中。不必担心，只要你输出一个正确的 `From:` 行，作

为邮件报头的一部分，当用户答复该邮件时，那么这就是你看到的一行信息。

问题：我应该如何将文件附加给电子邮件消息？

解答：你应该查看CPAN中的MIME模块。

22.5.2 思考题

1) \$foo=Net::SMTP->new(‘ mailhost ’)这个模块有何功能？

(如果你没有阅读“用于非UNIX系统的邮件函数”这一节，请现在阅读。)

- a. 它会产生一个句法错误。
 - b. 它创建一个对象，称为\$foo，代表与SMTP邮件服务器的连接。
 - c. 它将Net::SMTP模块纳入当前程序之中。
- 2) 下面几个电子邮件地址中哪一个可能是无效地址？
- a.foo!bar!baz!quux
 - b. “ ” @bar.com
 - c.stuff%junk! “ Wowzers ” !foo.com!blat

22.5.3 解答

1) 答案是b。如果你回答是a，那么可能出现了键入错误，也可能运行了Perl 4。选择c是不正确的，因为它实际上描述的语句是use Net::SMTP。

2) 这是个巧妙的问题。这几个地址都可能是有效的电子邮件地址。

22.5.4 实习

- 对简单的CGI电子邮件程序进行下列简单的修改：
- 搜集用户的浏览器信息，将它附加给邮件正文。
- 给用户发送一个礼仪邮件拷贝（如果你在真实的Web站点上发送这样的邮件，请务必告诉他这一情况）。这样做时你也要小心，因为有人不喜欢这样的邮件。
- 让用户在发送邮件之前能够“预览”邮件。必须使用第19学时中介绍的方法之一，使第一页（电子邮件输入屏）中的数据可供第二页（电子邮件核实屏幕）使用，最后供邮件发送程序使用。

China-pub.com

下载

China-pub.com

下载

第23学时 服务器推送和访问次数计数器

在本学时中，我们将要讲述两个常用的 CGI编程方法。你可以使用这些方法制作更加有趣的Web页，使之具备初步的动画功能，或者让人们竞相使用。

在本学时中你要学习：

- 使用服务器推送方法来刷新 Web页。
- 访问次数计数器。
- 代理和缓存。

23.1 什么是服务器推送

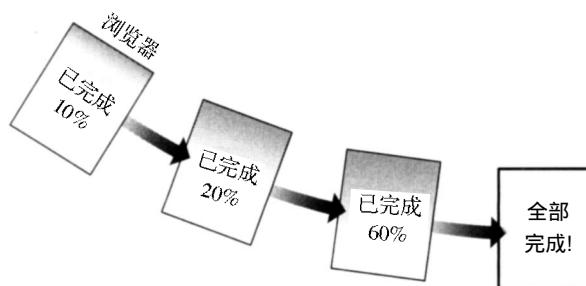
在传统的Web页中，加载速度慢或者不断增大的文档是很难使用的，因为 Web页只能一次看一页。比如，运行CGI程序的Web页需要花费很长的时间来运行。

首先，浏览器为了等待CGI程序结束运行，可能会超过原定的时间。浏览器为了等待程序运行的结果，通常等90秒钟左右，然后显示了一条消息，声称无法访问该站点。

其次，CGI程序有时会输出一条消息说：“ I’m still working, 20% complete (我仍在运行，已完成10%)”，过一会儿又说：“ I’m still working ,20% complete(我仍在运行，已完成20%)”，等等。输出这些消息是好的，问题是这些消息并不按固定间隔出现（因为缓存的缘故），当你完成程序的运行时，会有一个很长很长的 Web页。

你希望的是浏览器显示如图 23-1所示的信息。

图23-1 浏览器显示进度递增的信息



服务器推送技术利用了这样一个特性，即浏览器能够按各个部分来接收 Web页，然后依次重新显示这些Web页，就像你是依次取出不同的 Web页一样。

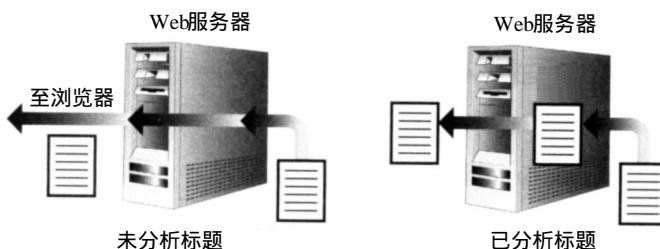


在撰写本书时，Microsoft的Internet Explorer并不支持执行服务器推送技术所需的协议。这是很遗憾的，因为使用服务器推送技术是使 Web页的内容动起来的一种简便方法。对于需要支持 Internet Explorer或者不支持这个特性的其他浏览器的 Web页来说，你应该使用客户机拖拉之类的其他技术。

23.1.1 激活服务器推送特性

你的Web服务器必须正确地安装，以便激活服务器的推送特性。为此，必须将 CGI程序作为未分析标题(nonparsed header)CGI程序来运行。当你使用未分析标题CGI程序时，服务器并不要求输出CGI标题，数据应该按原始状态直接发送给浏览器。通常来说，Web服务器要检查来自CGI程序的输出，以确保它们的正确性，因此，当CGI程序运行失败时，便出现错误500。未分析标题CGI程序将它们的输出直接发送到浏览器，如图23-2所示。

图23-2 未分析数据不经过
检查就通过了服务
器



你如何运行CGI程序以及服务器如何对标题不进行检查，这取决于Web服务器本身。例如，如果是Apache Web服务器，那么在CGI程序的文件名前面加上前缀nph-，就能使程序作为未分析标题程序来运行。例如，push.cgi是个已分析标题CGI程序，nph-push.cgi便是未分析标题CGI程序。但是Web服务器管理员可以修改这个命名规则的运行方式。

在Microsoft的Internet信息服务器（Internet Information Server, IIS）下，所有CGI程序都是作为未分析标题程序来运行的。CGI模块的header函数通常向你隐藏了这个情况，因此，在IIS下不必为服务器推送特性作任何修改。

如果你不清楚如何运行带有未分析标题的CGI程序，可以查看Web服务器的文档资料，或者求助于你的系统管理员。

23.1.2 一个小例子：更新Web页上的时钟

服务器推送技术的第一个例子是：编写一个简单的程序，以便更新Web页上的时钟。该时钟的运行方式是：让Web服务器每隔5秒钟左右推送出一个Web页，而Web页上将显示新的时间。Web服务器将不断推送出新的时间，直到浏览器删除该页，或者用户点击浏览器的Stop按钮，停止加载该页为止。

该CGI模块包含一组函数，目的是使服务器的推送操作比较容易一些。服务器推送的Web页也称为多部分文档。

程序清单23-1包含了HTML时钟的源代码。你必须键入该代码，然后用一个名字保存该代码，使Web服务器能够像上一节介绍的那样，将该程序作为未分析标题CGI程序来运行。

程序清单23-1 HTML时钟的源代码

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use CGI qw(:push -nph);
5:
6:  $|=1;    # Enable automatic buffer flushing
7:
```

```

8:   print multipart_init;
9:   while(1) {
10:     print multipart_start;
11:     print "The time is <H1>", scalar(localtime), "</H1>\n";
12:     print multipart_end;
13:     sleep 5;
14:   }

```

第4行：当CGI模块加载时，你必须指明正在执行服务器推送操作，因此这一行代码将命令:push赋予该CGI模块。另外，当你编写未分析标题脚本程序时，必须使用-nph将这个情况通知CGI模块。

第8行：multipart_init负责告诉浏览器，它后随的是个多部分Web页。它输出的是multipart_init，而不是通常用在普通Web页上的header函数。

第9行：while(1)能够有效地创建一个while循环，该循环将永远重复运行下去，这种循环称为无限循环。

第10行：multipart_start给要刷新的Web页的开始做上标号。如果一个Web页已经显示，本行代码将使浏览器清除该Web页，并等待接收新的内容。

第11行：这一行代码用于指明该页的内容。第4学时我们曾经讲过，标量上下文中的localtime用于输出格式为“Sun Sep 5 15:15:30 1999”的时间。

第12行：multipart_end用于给要刷新的Web页的结尾做上标号。本行代码只应该后随另一个multipart_start或者程序的结尾。

请注意while循环如何给multipart_init和multipart_end函数加上方括号。该循环能够有效地一次又一次重复显示同一个Web页，只有Web页上的时间是变化的。

23.1.3 另一个例子：动画

在程序清单23-2中显示的下一个例子（与上一个例子非常相似）中，显示了来自目录/images的一连串图形。这些图形使用服务器推送方法每次显示一个图形。目录中的每个文件被读取，并且作为一连串推送的Web页显示在浏览器中。

程序清单23-2 用服务器推送方法实现的动画

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use CGI qw(:push -nph);
5:  my($imagedir, @JPEGS);
6:  # Change the directory name below to suit your needs
7:  $imagedir="/web/Clinton_Test_Area/images";
8:  opendir(ID, $imagedir) || die "Cannot open $imagedir: $!";
9:  @JPEGS=sort grep(/\.\jpg$/, readdir ID);
10: closedir(ID);
11:
12: $|=1;    # Enable automatic buffer flushing
13:
14: print multipart_init;
15: foreach my $image (@JPEGS) {
16:   print multipart_start(-type => 'image/jpeg');
17:
18:   open(IMAGE, "$imagedir/$image") || die "Cannot open $image: $!";

```

```

19:     binmode(STDOUT); binmode(IMAGE); # Windows NT/95/98 only
20:     print <IMAGE>;
21:     close(IMAGE);
22:     print multipart_end;
23:     sleep 5;
24: }

```

程序清单 23-2 中的程序，大部分都与第 20 学时中介绍的“当日图形”和程序清单 23-1 的例子非常相似。

需要介绍的重要代码是第 16 行，即 `multipart_start(-type=> 'image/jpeg')`，它用于指明 CGI 程序并不在连续的 Web 页上输出普通文本或 HTML 输出信息，而是输出 JPG 图形。为了执行动画操作，该程序既可以直接输出 JPEG，也可以输出包含 `` 标号的 HTML。

23.1.4 客户机拖拉技术

使 Web 页依次加载的另一种技术称为客户机拖拉。使用客户机拖拉技术时，HTML 中嵌入了一些标记，以便告诉浏览器在一个间隔时间之后重新加载 Web 页（或另一个 URL）。例如，下面这个在 Web 页的 `<HEAD>` 节中的 HTML

```
<META HTTP-EQUIV="refresh" CONTENT="6;http://foo.bar.com">
```

将使浏览器在 6 秒钟后加载 Web 页 `http://foo.bar.com`。CGI 模块直接支持客户机拖拉命令。

当 Web 页的标题输出时，你可以设定应该重新加载的 Web 页，或者加载另一个 Web 页取代它的位置，方法是使用 CGI 模块的 `header` 函数的 `-Refresh` 选项，如下所示：

```
print header(-Refresh => '6;URL=http://foo.bar.com');
```

客户机拖拉方法实际上用于加载两次“刷新”之间的一个完整新页。这意味着如果你的 Web 页需要依次显示，比如像幻灯片那样来显示，就必须使用 URL 中嵌入的 cookie 或参数来跟踪下面显示哪个 Web 页。在两次刷新之间，必须在服务器与客户机之间建立一个新的连接，并且 Web 服务器必须为每次刷新启动一个新的 Perl 程序。这意味着不能太频繁地进行 Web 页的刷新。

使用客户机拖拉方法和服务器推送方法时遇到的主要问题是：

- 有些浏览器（比如 Internet Explorer）不支持服务器推送技术。
- 有些浏览器不支持客户机拖拉方法。

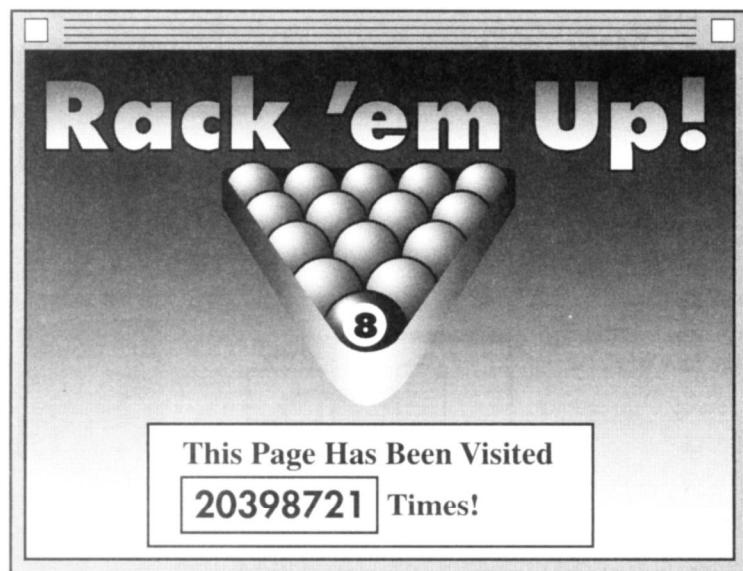
无论你用何种方法来编写你的程序，一种浏览器与另一种浏览器之间有些特性是互不兼容的。你必须决定你将容许存在哪些错误，并且根据情况编写代码。

23.2 访问次数计数器

在 Web 页上，你常常会看到一种称为访问次数计数器即访问者数量指示器的东西。可以想像，它表示 Web 页已被人们访问了多少次。图 23-3 显示了一个计数器的例子。

访问次数计数器有许多问题值得注意。首要的问题是计数器中的数字表示什么意思。“访问次数”的数目很大，表示访问这个 Web 页的人很多。如果访问的人很多，是否表示这是个很好的 Web 页呢？不一定。如果你访问一个 Web 页，这个 Web 页中可能有你需要的信息，也可能没有你要的信息。Web 页的质量好坏在于它是否含有对你有价值的信息，而不在于对其他人是否有价值。

图23-3 访问次数计数器的举例



实际上访问次数计数器是一种瞎子当裁判的选美比赛。计数器中的数字不一定是访问你的Web页的人数，它最多只不过是一个很不准确的估计数字。为什么这些计数器如此不准确呢？我想有下列几个原因。

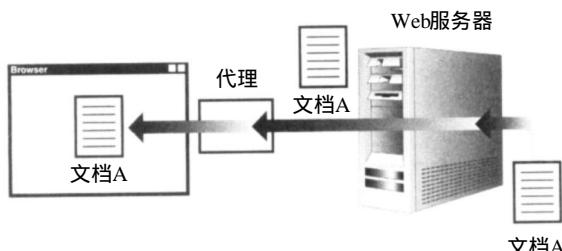
首先，没有一条规定说，访问次数计数器必须从0开始计数。当你发放最后一本支票簿时，第一张支票是1号支票吗？当你给支票排序时，完全可以选择你的起始序号。如果你很聪明，你会选择一个大号码，这样，你看上去在银行中开立了一个长期帐户。如果支票号码很小，那么商店服务员一定会对你的ID号码多看两眼，并且也许根本不会考虑接收你的支票。Web站点操作员常常在开始时将访问次数计数器的数字设置得比较大，使他们的Web点看上去比实际上更“受欢迎”。

访问次数计数器存在的第二个问题是Web机器人程序，也叫做Web蜘蛛、Web爬虫等。这些自动化进程能够搜索Web上的数据，有时只是为了查看一组特定的数据，有时为了建立感兴趣的Web站点的索引。你是否想过为什么AltaVista、Google或HotBot要建立它们的索引呢？它们搜索Web，检索Web页，最后访问次数计数器的数字升高到比它们的实际访问次数高。

第三个问题是Web浏览器上的Refresh（刷新）按钮。每次在你的Web页被刷新时，访问次数计数器就会升高一格。如果有人点击重新加载按钮，实际上你并没有计算你的Web站点的“访问者”数目，是不是？

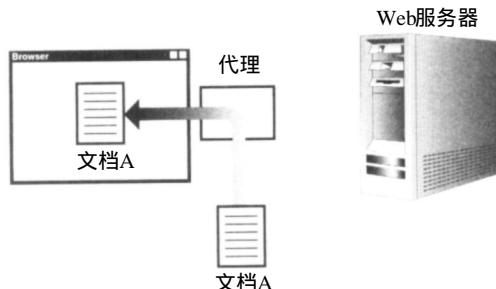
最后也是最重要的一个问题就是缓存问题。在第17学时中，我们介绍了浏览器如何与Web服务器进行通信的方块图。它展示了一个重要细节，如图23-4所示。

图23-4 代理服务器为浏览器检索Web页



如果Web浏览器位于大型ISP如aol.com或home.com等的域中，这些ISP拥有数百万个用户，那么这些ISP通常要使用缓存代理。缓存代理位于你的Web浏览器与Web服务器之间。当你检索一个Web页时，检索请求先送到缓存代理那里，然后由缓存代理为你在Internet上取出该Web页，再将该Web页发送给你的浏览器，此前它要为自己将该Web页的拷贝存储起来（见图23-5）。如果同一个域中的另一个人想要检索该Web页，缓存代理就不必到Internet上去检索这一页，它可以使用保存在缓存中的拷贝。

图23-5 代理服务器从它的缓存中检索Web页



存储Web页拷贝的代理服务器人为地降低了访问次数计数器指示的访问次数。奇怪的是，它还使remote_host值多次重复，因为该Web页被许多人所检索。



大公司和大学中的Web冲浪者常常位于作为缓存代理的防火墙的后面。从这些站点之一检索的每一页都有可能没有计入访问次数计数器，因为它被缓存代理挡住了。

23.2.1 编写一个访问次数计数器程序

阅读了上一节内容后，如果你接着读下去，一定有兴趣为你的Web页编写一个访问次数计数器程序。访问次数计数器有两个基本类型，一种是简单的文本计数器，另一种是图形计数器。下面介绍的第一个计数器例子是文本计数器，第二个计数器是图形计数器，并且我们还讲述制作非常出色的访问次数计数器的一些思路。

若要使用该访问次数计数器，请将它用作服务器端的包含程序的一部分，我们在第20学时中已经讲过这种包含程序。如果你调用访问次数计数器CGI程序hits.cgi，可以使用下面这个SSI，将它纳入任何Web页：

```
<!--#exec cgi="/cgi-bin/hits.cgi"-->
```

程序清单23-3显示了该访问次数计数器的源代码。

程序清单23-3 访问次数计数器程序

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use Fcntl qw(:flock);
5:  use CGI qw(:all);
6:
7:  my $semaphore_file='/tmp/webcount_lock';
8:  my $counterfile='/web/httpd/countfile';
9:  sub get_lock {

```

```
10:     open(SEM, ">$semaphore_file")
11:         || die "Cannot create semaphore: $!";
12:     flock(SEM, LOCK_EX) || die "Lock failed: $!";
13: }
14: # Function to unlock
15: sub release_lock {
16:     close(SEM);
17: }
18: get_lock(); # Get a lock, and wait for it.
19: my $hits=0;
20: if ( open(CF, $counterfile) ) {
21:     $hits=<CF>;
22:     close(CF);
23: }
24: $hits++; # Increase the hits by 1.
25: print header;
26: print "You have had $hits visitors";
27:
28: open(CF, ">$counterfile") || die "Cannot open $counterfile: $!";
29: print CF $hits;
30: close(CF);
31:
32: release_lock(); # Release the lock
```

第18行：这里需要一个锁，因为访问次数计数器文件可能被许多进程同时读取和写入。

第20~23行：\$counterfile中的文件内容被读取。它是迄今为止的访问次数。

第28~30行：访问次数计数器的内容被重新写回到 \$counterfile中的文件。

第32行：最后，锁被释放。

程序清单 23-3 中的大部分代码你并不会感到有什么特殊。但是请注意，它使用了文件锁，并且这个示例程序遵循第 15 学时中介绍的文件锁定原则。

当两个人几乎同时加载 Web 页时，就有必要对文件加锁。如果对 Web 访问次数计数器文件的读取和写入操作稍稍失去同步，那么计数器的数字就会增加得太快或太慢，也可能会产生一个受到破坏的文件。这些结果将会进一步降低计数器的准确性。

23.2.2 图形访问次数计数器

若要改进访问次数计数器，可以采取 3 种不同的方法。首先，可以制作一个图形，代表计数器的每个可能的值，并且根据需要来显示该图形。如果你接到多个访问者对 Web 站点的访问请求，那么这种办法比较费时。

第二种方法是让一个 Perl CGI 程序生成必要的图形，以便显示访问次数计数器本身。CPAN 中的 GD 模块可以用于以 Perl 程序来创建图形，因此你可以将它用于这个目的。不过关于 GD 模块的具体特性不在本书讲解的范围之内。

最容易的方法是创建 10 个图形，分别代表 0 至 9 的 10 个数字。然后，当计数器中的数字递增时，你的程序只需输出带有 < IMG > 标记的 HTML，将数字放入正确的位置（见图 23-6）。当然，必须创建代表数字的图形。程序清单 23-4 中的 Perl CGI 程序将图形命名为 digit_0.jpg，digit_1.jpg，直至 digit_9.jpg。

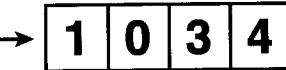
若要使用图形访问次数计数器，可以将它用作服务端的包含程序的一部分，如第 20 学时中描述的那样。如果你调用访问次数计数器的 CGI 程序 graphical_hits.cgi，你就可以将它纳入任何 Web 页，如下所示：

```
<!--#exec cgi="/cgi-bin/graphical_hits.cgi"-->
```

程序清单 23-4 显示了该图形访问次数计数器程序的源代码。

图23-6 图形访问次数计数器的输出

```
<IMG SRC = "digit_1.jpg">
<IMG SRC = "digit_0.jpg">
<IMG SRC = "digit_3.jpg">
<IMG SRC = "digit_4.jpg">
```



程序清单 23-4 图形访问次数计数器的程序

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use Fcntl qw(:flock);
5:  use CGI qw(:all);
6:
7:  my $lockfile='/tmp/webcount_lock';
8:  my $counterfile='/web/httpd/countfile';
9:  my $image_url='http://www.server.com/images';
10:
11: sub get_lock {
12:   open(SEM, ">$lockfile")
13:   || die "Cannot create semaphore: $!";
14:   flock(SEM, LOCK_EX) || die "Lock failed: $!";
15: }
16: sub release_lock {
17:   close(SEM);
18: }
19: get_lock(); # Get a lock, and wait for it.
20: my $hits=0;
21: if ( open(CF, $counterfile) ) {
22:   $hits=<CF>;
23:   close(CF);
24: }
25: $hits++;
26:
27: open(CF, ">$counterfile") || die "Cannot open $counterfile: $!";
28: print CF $hits;
29: close(CF);
30: release_lock(); # Release the lock
31:
32: # Now, create the <IMG> tags.
33: print header;
34: foreach my $digit (split(//, $hits)) {
35:   print "<IMG SRC=$image_url/digit_$digit.jpg>";
36: }
```

程序清单 23-4 实际上与程序清单 23-3 相同，只有某些很小的修改。

第9行：这行代码在\$image_url中包含了构成数字的各个图形的基本 URL。请记住，它必须是浏览器加载图形时查看的 URL，而不是到达本地磁盘上的图形的路径。

第34~35行：访问次数计数器中的数字 \$hits 对每个字符进行分割，再赋予 \$digit，每次赋予一个数字。然后为每个数字输出 < IMG > 标号。

23.3 课时小结

在本学时中，我们讲述了在 Web 页上实现动画的两种方法。可以使用服务器推送技术，

迫使浏览器连续更新 Web 页。如果这种方法不行，或者在某种浏览器上无法实现，可以使用客户机拖拉技术，获得类似的效果。然后介绍了访问次数计数器，并且说明了计数器为什么不那么准确。

23.4 课外作业

23.4.1 专家答疑

问题：服务器推送技术不起作用，为什么？

解答：许多因素会导致服务器推送技术不起作用。首先，浏览器必须支持服务器推送技术；第二，Web 服务器必须使用未分析标题；最后，你的 CGI 程序必须正确地运行。如果从命令行提示符处以交互方式运行 CGI 程序，应该确保它的输出按固定时间间隔产生，并且输出必须正确。

问题：如果访问次数计数器很不准确，有没有别的办法可以用来测定对 Web 站点的访问次数？

解答：几乎没有。查看服务器日志与使用访问次数计数器同样不可靠。测定访问站点次数的方法之一是使用实现重定向（参见第 20 学时）的点击链接，另一种方法是让访问者填写一个窗体。使用 HTML 窗体中的 POST 方法，是避免你的 Web 页被缓存代理进行缓存的唯一完全可靠的方法，它可以确保 POST 方法提交的窗体不能被任何代理进行缓存。

23.4.2 思考题

1) 为了执行服务器推送操作，需要使用 CGI 模块中的哪些函数？

- a. multipart_start 和 multipart_end
- b. multipart_init, multipart_start 和 multipart_end
- c. push_start 和 push_end

2) 所有浏览器都支持客户机拖拉技术，因为它是 HTML 标准的组成部分。

- a. 是。
- b. 否。

3) 缓存代理能够保证不对何种类型的 Web 页进行缓存？

- a. 来自使用 POST 方法的 HTML 窗体的答复页。
- b. 服务器推送的内容。
- c. CGI 程序的任何输出。

23.4.3 解答

1) 答案是 b。multipart_init 用于使浏览器准备接收多部分构成的 Web 页。multipart_start 和 multipart_end 用于给每个 Web 页做上开始和结束标记。

2) 答案是 b。完全不是。< META > 标记可以被浏览器忽略，这是 HTML 标准规定的。另外，使用 header 函数中的 -Refresh 选项并不能保证浏览器按需要重新加载 Web 页，浏览器也有一个该命令的选项。

3) 答案是a。原因已经在“专家答疑”这一节中作了解释。

23.4.4 实习

- 修改程序清单23-3（或程序清单23-4）中的访问次数计数器程序，为不同类型的浏览器保存不同的计数。为此，你必须为你要跟踪的每种浏览器建立不同的文件。不要忘记为你不能识别其身份的浏览器保留一个额外的文件。

China-pub.com

下载

China-pub.com

下载

第24学时 建立交互式Web站点

如果你在 Web 站点上使用 CGI 程序的目的是使访问者能够进入你的站点，并且为他们提供一个可以访问的有价值的站点，那么除了建立访问计数器外，还必须设计一个更好的站点。

Web 上最有价值的站点是能够提供频繁更新内容的站点。如果你的 Web 页上的信息是静态的，人们就没有理由再次访问它。经过几次访问后，他们就会知道你的站点没有太大的变化，因此不会再访问。

要使人们重访你的 Web 站点，方法之一是让他们在某种程度上参与站点的活动。人们作为一个群体，喜欢互相交谈。人们希望成为一个群体的成员，并且具有参与感，这是人的一个共性。

本学时中介绍的程序提供了一个进行每项操作的工具。第一个程序提供了一种方法，用于扫描 Internet 上的另一个信息源的内容，改变该内容的格式，并且在你的站点上显示这些内容，同时加上一些警告。第二个程序使得你的站点的访问者可以参与一个调查活动。

在本学时中，你将要学习：

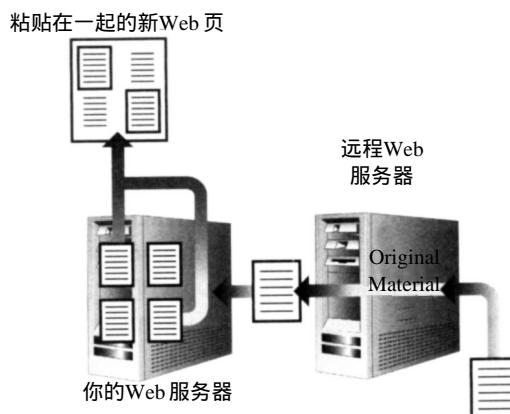
- 如何借用另一个 Web 站点的内容。
- 如何创建一个交互式民意测验站点。

24.1 借用另一个站点的内容

也许你看到过这样的 Web 站点，在 Web 页的某个位置显示了最新的股票行情信息、新闻标题或体育比赛的比分，尽管运行该站点的人与发布这些内容的机构并无关系。

出现这种情况的通常原因是：在你查看的系统上运行的程序常常可以访问信息的原始站点，并将信息拖拉过来，然后改变信息的格式，显示在目标 Web 页上。图 24-1 显示了这个进程的运行情况。

图 24-1 取出一个 Web 页，
改变其格式，然后
将它重新显示



这里你会看到你的 Web 服务器通过它的 CGI 程序，已经变成一个 Web 客户机。当服务器为自己检索 Web 页时，它能将这些页连接在一起，然后再次显示该信息。

24.1.1 注意内容的版权问题

在你继续读下去，了解如何借用其他站点的内容之前，有几个问题必须了解。首先，如

果你要显示的信息不是你自己的信息，而是来自另一个 Web站点或数据库的信息，那么这些信息也许会受到版权法的保护。从另一个 Web站点那里借用信息，然后在你自己站点上显示，这会使你陷入严重的法律争端。如果触犯版权法，可能导致你的 Web站点和ISP被关闭，你可能被罚款、监禁或者受到法律指控。

触犯版权法也是一种粗鲁的行为。

如果你想在自己的 Web站点上使用其他来源的信息，首先要获得其他信息源的许可。大多数Web站点运营商允许你显示来自他们站点的内容。他们通常要求你考虑下列几个问题：

- 你应该清楚地注明内容的来源，可以用标题、链接和文字来注明。
- 你应该清楚地注明内容的版权，说明该内容是经过允许而使用的。
- 也许你无权通过“深层次链接”访问他们的 Web站点，也就是说通过几个层次的链接访问他们站点的Web页。他们希望你只链接到顶层的 Web页。
- 你只能偶尔更新你的 Web页版本。将其他站点的服务器隐藏在信息之下，以便使你的站点看上去更加出色，这种做法是不可取的。

Slashdot.org是个为技术用户提供内容的 Web站点，它的经营者允许我们使用他们的站点来演示本书中的示例代码。当你在自己的 Web页上运行这些示例代码之前，应该征得该站点的同意。如果要与 Slashdot.org取得联系，你可以在 <http://www.slashdot.org>上通过他们的 Web站点和FAQ找到有关的详细说明。

24.1.2 举例：检索标题

如果要在你的 Web站点上显示Slashdot站点的新闻标题，请按下列步骤操作：

- 1) 通过服务器分析的HTML Web页，启动headlines.cgi CGI程序。
- 2) 然后该CGI程序查看它是否拥有存储在磁盘上的最新新闻标题拷贝。如果有，就使用之。如果没有，便从 Slashdot.org的Web站点检索这些标题。
- 3) 接着CGI程序分析标题文件并显示标题。

若要从另一个 Web站点中检索 Web页或其他内容，需要一个模块，它不是标准 Perl模块 LWP::Simple的组成部分。LWP模块使你能够从 Internet上检索所有类型的信息，比如 Web页、FTP数据、新闻组文章等。



LWP :: Simple模块被封装为 libwww-perl模块包的一部分。这个模块包含了许多个模块，分别用于检索 Web页、分析HTML、分析URL，遍历Web站点，还可以做许多其他事情。使用这些模块的好处是它们的安装是非常值得的。Libwww-perl模块包位于本书所附的光盘上。

LWP :: Simple模块一旦安装，你就可以像下面这样检索 Web页：

```
use LWP::Simple qw(get);
$content=get("http://www.slashdot.org");
```

现在\$content包含了该URL上的Web页的文本。这不是非常容易吗？

程序清单24-1到24-3展示了检索Slashdot的标题并显示这些标题时使用的程序。

程序清单24-1 Slashdot的标题程序的第一部分

```
1:  #!/usr/bin/perl -w
```

```

2:
3:  use strict;
4:  use Fcntl qw(:flock);
5:  use LWP::Simple qw(get);
6:  use CGI qw(:all);
7:
8:  my $url="http://slashdot.org/slashdot.xml";
9:  my $cache="/tmp/slashcache";
10: my $lockfile="/tmp/slashlock";
11: sub get_lock {
12:   open(SEM, ">$lockfile")
13:     || die "Cannot create lockfilee: $!";
14:   flock(SEM, LOCK_EX) || die "Lock failed: $!";
15: }
16: sub release_lock {
17:   close(SEM);
18: }
```

程序清单24-2 Slashdot的标题程序的第二部分

```

19:
20: print header;
21: # If the cache is older than about an hour, rebuild it
22: get_lock();
23: if ( (not -e $cache) or ( (-M $cache) > .04)) {
24:   my $doc=get($url);
25:   if (defined $doc) {
26:     open(CF, ">$cache") || die "Writing to cache: $!";
27:     print CF $doc;
28:     close(CF);
29:   }
30: }
31: release_lock();
32:
```

程序清单24-3 Slashdot的标题程序的第三部分

```

33: print "<H2>Slashdot.Org's Headlines as of ",
34:      scalar(gmtime((stat $cache)[9])),
35:      "GMT </H2>Updated Hourly!<P>";
36:
37: open(CF, $cache) || die "Cannot open the cache: $!";
38: my($title, $link);
39: while(<CF>) {
40:   if (m,<title>(.*)</title>,) {
41:     $title=$1;
42:   }
43:   if (m,<url>(.*)</url>,) {
44:     $link=$1;
45:     print qq{<A HREF="$link">$title</A><BR>\n};
46:
47:   }
48: }
49: print "Copyright Slashdot.Org, used with permission.";
50: close(CF);
```

第3~6行：为了编写本程序，需要许多不同的模块。应该使用模块 Fcntl，因为你必须锁定该程序的一部分，这样每次就只能由一个用户来运行该程序。必须使用 LWP::Simple模块（尤

其是get函数），以便从Slashdot的Web站点检索新闻标题。当然，你也需要CGI模块，因为这是个CGI程序。

第8行：它包含文件的URL，该文件只包含新闻标题。文件的格式类似下面的形式：

```
<story>
<title>Ask Slashdot: Internet Voting?</title>
<url>http://slashdot.org/askslashdot/99/09/05/1732249.shtml</url>
<time>1999-09-05 21:34:36</time>
<author>Cliff</author>
:
```

这里的每条新闻都用<story>标记括了起来。该文件采用称为XML标记语言的某种简化形式。这使得Perl程序能够很容易地处理该文件，你在下面可以看到这一点。

第9行的变量\$cache包含了你临时存放Slashdot标题所用文件的名字。使用该文件后，每当程序被调用时，你就不必访问Slashdot的服务器来检索信息，因为你拥有一个本地拷贝。

当然，现在你应该熟悉get_lock()和release_lock()这两个子例程，因为你已经在另外3个学时中看到过这些函数。之所以需要使用这些函数，原因是\$cache中的文件不应该每次都被多个程序更新，因此它必须锁定。

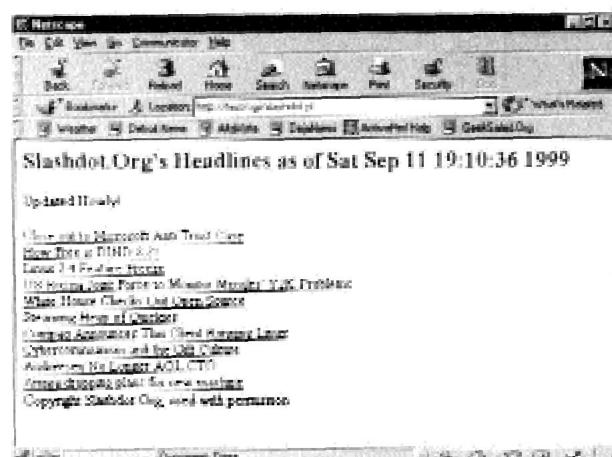
第23行：如果缓存文件不存在，请找出该文件，如果缓存文件已经存在60分钟以上，就应该重建该文件。Perl中的-M函数返回Perl程序启动以来该文件的修改时间，但返回的时间采用小数表示的格式。因此，如果该文件已经存在了一天，-M函数返回1，如果文件存在了6个小时，-M返回0.25(四分之一天)，如果文件存在了1小时，-M返回0.0416666(约1/24天)。

第24行：用于检索包含标题的URL，如前面介绍的LWP::Simple模块的get函数。下面几行用于将被检索的\$doc中的文档写入缓存文件。如果get方法运行失败，它返回undef，并且在第25行中对此进行检查。

请注意，get_lock()和release_lock()函数序列位于if语句的外面。这一点非常重要。如果CGI程序的一个实例正忙于更新缓存文件，你就不需要另一个实例来查看缓存文件是否存在，或者上次它是何时被修改的。

该程序的最后一部分最简单明了。第33~35行用于输出简介和上次更新缓存文件的时间。第34行有点儿复杂，我们对此作一说明。首先，stat用于获取关于\$cache中文件的信息，并将它作为一个列表返回。第二，列表的第9个元素（上次修改时间）被取出。第三，用该时间用

图24-2 Slashdot.cgi程序的输出



localtime，在标量上下文中，它返回格式很好的时间字符串。

第40和43行用于从Slashdot中取出标题文件的<title>和<url>节。与标题和URL相匹配的部分由正则表达式保存在\$1中，然后分别赋予\$title和\$link。由于<url>元素总是出现在<title>元素之后，因此，当<url>元素被看到时，\$title和\$link均可在第45行上输出。

在一般情况下，这些正则表达式不应该用来与HTML相匹配。它们在这里起作用，原因是Slashdot的XML标题文件格式化很好，每一行只有一个XML元素。如果文件格式要变更，该程序无法进行处理，你应该查看Slashdot的FAQ，以了解什么发生了变化。

当程序最后运行起来时，其输出将类似图24-2显示的形式。

当然，你应该运用你的HTML技巧使这个输出更加漂亮一些。

24.2 调查窗体

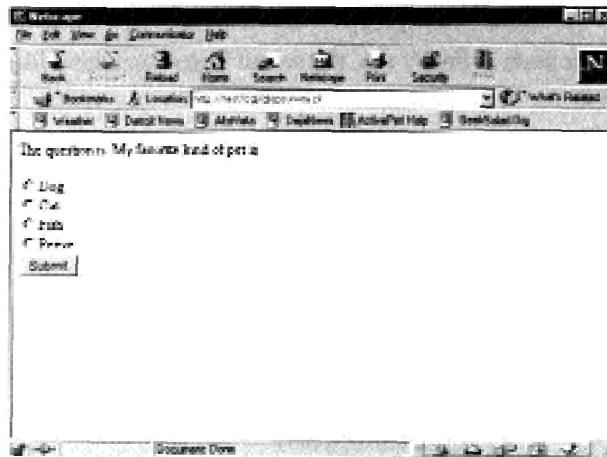
人人都希望成为某个重要人物。人人都希望他的观点能够引起人们的重视，甚至成为一个非常重要的观点，而且每个人都希望知道他的观点与别人的观点相比较的结果。这就是调查要达到的目的。

在下面这个练习中，我们将要介绍一个小程序，用来创建一个调查窗体，然后再创建一个用于输出调查结果的程序。调查程序是个文本文件，它包含一个问题，后面是一些选项，该文件被放入Web服务器上的一个目录下，其名字带有扩展名.txt。该文本文件类似下面的形式，但是没有其他标点符号或空行：

```
My favorite kind of pet is:  
Dog  
Cat  
Fish  
Peeve
```

第一个程序用于查看该目录，找出带有扩展名.txt的文件（如果存在多个文件，则取其最后一个文件），然后将问题作为窗体的一部分来显示，如图24-3所示。

图24-3 调查窗体



使用纯文本文件的优点是：CGI程序可以运用文本文件来显示问题，并在以后显示问题的答复。如果你想增加新的调查文件，可以将另一个.txt文件添加到该目录中，CGI程序能够自动开始使用该文件。它不需要进行很多的维护。

当用户选定一个选项且提交该窗体时，第二个 CGI 程序便取出问题的答复，并将它写入与问题在同一个目录中的一个文件中。如果问题文件称为 `foo.txt`，那么答复将存放在 `foo.answer` 文件中。当程序写完答复后，它将重新读取所有的答复，并且显示调查结果。

24.2.1 调查窗体程序的第一部分：提出问题

在这个调查中提出问题的程序是非常简单明了的。比较复杂的部分是遍历存放调查信息的目录，找出目录中最后一个扩展名为 `.txt` 的文件。实际上，由于提出问题和写出调查结果的这两个程序都需要查找该文件，因此可以将这部分程序编写成可以重复使用的函数，这样，你就可以在两个地方使用它了。程序清单 24-4 显示了调查窗体程序的第一部分。

程序清单 24-4 显示调查窗体的程序的第一部分

```

1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use CGI qw(:all);
5:  my($survey_dir);
6:  $survey_dir="/web/htdocs/poll";
7:
8:  sub find_last_file {
9:      my($type)=@_;
10:     my(@files, $last_file);
11:     # Open the directory, get the last file
12:     # of the correct type.
13:     opendir(SD, $survey_dir) || die "Can't open $survey_dir: $!";
14:     @files=reverse sort grep(/^.{$type}$/, readdir SD);
15:     closedir(SD);
16:     $last_file=$files[$#files];
17:     return($last_file);
18: }
19:
20: sub get_file_contents {
21:     my($type)=@_;
22:     my(@answers, $last_file);
23:
24:     $last_file=find_last_file($type);
25:
26:     return if (not defined $last_file);
27:     # Open that file, get the contents and return it.
28:     open(QF, "$survey_dir/$last_file")
29:         || die "Can't open $last_file: $!";
30:     @answers=<QF>;
31:     close(QF);
32:     chomp @answers;    # Remove the newlines
33:     return(@answers);
34: }
```

在第6行上，`$survey_dir` 包含了调查文件所在的目录。若要创建一个新调查文件，只要将一个扩展名为 `.txt` 的文本文件放入该目录即可，如本节开头介绍的那样进行操作。该目录必须是可供 Web 服务器的进程写入的目录。能够写入的目录意味着至少拥有 755（在 UNIX 系统中）访问权，或者拥有声明的客户写入权限（在 Windows 下）。

函数 `find_last_file()` 根据扩展名 `.txt` 或 `.answer`，在 `$survey_dir` 中按字母顺序寻找带有该扩展名的最后一个文件。这个通用型函数在以后供 `get_file_contents()` 函数使用，并且用于下一节

中的调查写入程序。如果目录中不存在该类型的任何文件，那么 `find_last_file()` 返回 `undef`。

函数 `get_file_contents()` 再次将扩展名 `.txt` 或 `.answer` 作为参数，返回该目录中最后一个文件的内容。为了找到该文件名，它使用 `find_last_file()` 函数。

程序清单 24-5 中显示的该程序的剩余部分是很短的。

程序清单 24-5 显示调查窗体程序的第二部分

```

35:
36: # Get the contents of the last text file, Q & A
37: my($question, @answers)=get_file_contents("txt");
38:
39: print header;
40: print qq{<FORM ACTION="/cgi/writesurvey.cgi" METHOD=POST>\n};
41: print "The question is: $question<P>\n";
42: my $answer=0;
43: foreach(@answers) {
44:     print "<INPUT TYPE=RADIO NAME=answer value=$answer>";
45:     print "$_\n";
46:     $answer++;
47: }
48: print qq{<INPUT TYPE=SUBMIT VALUE="Submit">};
49: print qq{</FORM>};

```

当这个代码从第 36 行上开始运行时，函数 `get_file_contents()` 将来自最后一个 `.txt` 文件的内容的第一行加载到 `$question` 中，将文件的其余部分加载到 `@answers` 中。

必须将第 40 行中的 `/cgi/writesurvey.cgi` 改为用于 CGI 调查程序第二部分的任何一个名字。

从那里开始，输出标题，该窗体的开始部分被发送到浏览器。`@answers` 中的每一行输出时旁边都有一个单选按钮。第一个答复 / 单选按钮的值为 0，第二个的值为 1，以此类推，直到 `@answers` 中不再遗留任何答复为止。窗体的正文如下所示：

```

<INPUT TYPE=RADIO NAME=answer value=0>Dog<BR>
<INPUT TYPE=RADIO NAME=answer value=1>Cat<BR>
<INPUT TYPE=RADIO NAME=answer value=2>Fish<BR>
<INPUT TYPE=RADIO NAME=answer value=3>Peeve<BR>

```

当该窗体被提交时，`answer` 的参数被传递给负责写出答复的 CGI 程序，该程序称为第 40 行中的 `/cgi/writesurvey.cgi`，但是你可以修改这个名字。该程序在下一节中介绍。

24.2.2 调查窗体程序的第二部分：计算调查结果

当用户点击调查窗体上的 `Submit` 按钮后，实际程序就开始运行了。用户的选择被记录到一个文件中，调查结果必须制成表格，然后显示出来。

下面这个程序清单看上去很长，但是它的主体部分是你在以前已经见过的子例程。你在本书中的许多地方见过的文件锁定子例程 `get_lock()` 和 `release_lock()`，以及显示调查窗体程序中的 `get_file_contents()` 和 `find_last_file()` 子例程，构成了这个 CGI 程序的主体。

程序清单 24-6 中的代码是该程序的开始部分。请记住，由于你在以前已经看到过该程序的大部分代码，因此不应该对它的长度感到害怕。

程序清单 24-6 接收调查结果的程序的第一部分

```

1:  #!/usr/bin/perl -w
2:

```

```

3:  use strict;
4:  use Fcntl qw(:flock);
5:  use CGI qw(:all);
6:
7:  my($survey_dir, $lockfile);
8:  $survey_dir="/web/htdocs/poll";
9:  $lockfile="/tmp/surveylock";
10:
11: sub find_last_file {
12:     my($type)=@_;
13:     my(@files, $last_file);
14:         # Open the directory, get the last file
15:         # of the correct type.
16:         opendir(SD, $survey_dir) || die "Can't open $survey_dir: $!";
17:         @files=reverse sort grep(/^\.$type$/, readdir SD);
18:         closedir(SD);
19:         $last_file=$files[$#files];
20:     return($last_file);
21: }
22:
23: sub get_file_contents {
24:     my($type)=@_;
25:     my(@answers, $last_file);
26:
27:     $last_file=find_last_file($type);
28:
29:     return if (not defined $last_file);
30:         # Open that file, get the contents and return it.
31:         open(QF, "$survey_dir/$last_file")
32:             || die "Can't open $last_file: $!";
33:         @answers=<QF>;
34:         close(QF);
35:         chomp @answers;    # Remove the newlines
36:     return(@answers);
37: }
38: sub get_lock {
39:     open(SEM, ">$lockfile")
40:         || die "Cannot create lockfile: $!";
41:     flock(SEM, LOCK_EX) || die "Lock failed: $!";
42: }
43: sub release_lock {
44:     close(SEM);
45: }
```

到现在为止，程序清单 24-6中的所有代码对你来说应该是熟悉的。这里定义的子例程既可以来自上一个程序（比如 `get_file_contents()` 和 `find_last_file()`），也可能是 `get_lock()` 和 `relase_lock()` 例程。同样，必须确保保存放在 `$survey_dir` 中的目录能够被 Web 服务器写入。

由于这部分代码都很简单明了，不必多加说明，因此我们可以转入程序清单 24-7的介绍。

程序清单 24-7 接收调查结果的程序的第二部分

```

46: my($question, @poss_answers)=get_file_contents("txt");
47:
48: print header;
49:
50: # Add their answer to the answer file
51: if (defined param("answer")) {
52:     my($lastfile);
53:     get_lock();
```

```
54:     # Find the last survey file name, and create
55:     # the answer filename from that.
56:     $lastfile=find_last_file("txt");
57:     $lastfile=~s/.txt/answer/;
58:
59:     open(ANS, ">>$survey_dir/$lastfile")
60:         || die "Can't write to $lastfile: $!";
61:     print ANS param("answer"), "\n";
62:     close(ANS);
63:     release_lock();
64: }
65:
66: my(@answers)=get_file_contents("answer");
67: my(%results);
68: # This accumulates how many times each
69: # answer was given in a hash.
70: foreach(@answers) {
71:     $results{$_]++;
72: }
73: my $ansno=0;
74: foreach my $ans (@poss_answers) {
75:     $results{$ansno}=0 if (! exists $results{$ansno});
76:     print "$ans was selected $results{$ansno} times<BR>";
77:     $ansno++;
78: }
```

这部分程序紧接着上一部分程序的结尾。当前提出的问题和对问题的答复分别存放在第46行中的\$question和@poss_answers中。

从第50行起，本程序要查看用户是否提供了对调查问题的答复。请记住，如果用户愿意的话，他可以只点击 Submit按钮，而不提供答复。如果提供了答复，便在第 53行上用get_lock()函数取出一个锁，以防止多人同时更新调查结果。

在第56行上，找到了最后一个 .txt调查文件，比如说 first.txt，同时，可以用替换方式将.txt改为.answer，以便给出first.answer。答复文件被打开，当前的答复被附加给该文件，同时，relase_lock()函数对文件进行解锁，因为现在其他人可以安全地打开该文件了。

第66行：get_file_contents()函数用于获取调查结果，而不是调查的问题。这时，名叫%results的哈希结构得以创建，其关键字是问题的答复，即数字 0、1、2等，它的值是看到每个关键字的次数。

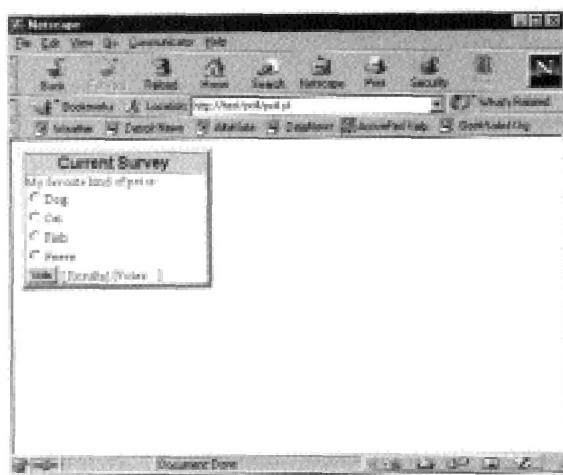
第74行起输出每个可能的答复。如果 %results中没有任何项目与某个答复相对应，那么调查结果必须是0。答复与给出的次数将在第 76行上输出。

图24-4显示了产生的调查窗体。如果你想使这个调查窗体看上去更好些，可以让 CGI程序显示对调查问题的答复（用带颜色的表格显示调查结果）和其他所有特性，使 HTML页看起来更加美观。



用于保存调查程序的目录（上例中的/web/htdocs/poll）必须是可供全世界的人都能写入的目录，以便使该程序能够运行。在Windows NT下，可以设置该目录的属性，使它可以被Guest（客方）写入。在UNIX下，可以使用chmod命令将访问权限设置为777。另外，如果用人工运行调查结果的接收程序，而不使用浏览器，那么它会创建一个Web服务器无法写入的.answer文件。如果是这样的话，你必须删除该文件，然后调查程序才能正确运行。

图24-4 经过格式美化的调查窗体



24.3 课时小结

在本学时中，创建了两个程序，使你的 Web页变得更加丰富多彩。首先，创建了一个从其他站点检索内容并将内容显示在你自己的 Web页上的程序。我们还讲述了向其他人借用 Web 内容时应该注意的一些问题。接着又创建了一个调查程序，使 Web站点的访问者能够参与站点正在进行的活动。

24.4 课外作业

24.4.1 专家答疑

问题：在Web服务器上拥有一个可供所有人写入的目录（用于民意测试），是否会带来安全上的漏洞？

解答：是的，不过这个漏洞并不大。如果你的服务器是以一种合理的方式安装的，那么就无人能够将内容上载到你的站点。但是，如果你的服务器允许任何人将任何东西上载到任何地方，那么那里就可能存在滥用危险。如果你愿意的话，可以在创建 .txt文件的同时，创建.answer文件，以便解决这个问题。请记住使用 chmod使.answer文件成为所有人都能写入的文件。

问题：如果我仅仅从某个站点借用了一些新闻标题，我会不会受到指控？

解答：是的，你会受到指控，以前曾经发生过这样的的事情。1999年2月，Microsoft与Ticketmaster两公司之间就因为这样的问题而对簿公堂。据说 Microsoft公司使用“深层次链接”进入了Ticketmaster公司的Web站点，为此Ticketmaster提出了诉讼。在这个案件中，没有出现侵犯版权的问题，但是有足够的证据说明 Microsoft公司进入了“深层次链接”。如果涉及到侵犯版权的问题，问题就严重了。

问题：有一个站点，我想从中借用其新闻标题。它类似 Slashdot的站点。但是该站点不具备很好的XML或文件供分析，因此我必须改用普通的 HTML文件。我应该如何分析该文件？

解答：如果你要分析 HTML文件，请不要使用正则表达式，也不要自己对它进行分析。对HTML文件进行分析并不像它看起来那样容易，并且几乎无法得到正确的结果。另外，即

使你试图用正则表达式来分析某些HTML文件，它也不是到处都能取得成功的。CPAN包含用于分析HTML文件的一些模块，这些模块都在CPAN的HTML节下，即HTML::*下。

24.4.2 思考题

- 1) 若要从Web服务器检索一个HTML文件，我应该怎么做？
 - a. 使用LWP。
 - b. 打开到达该系统的套接字，然后检索数据。
 - c. 使用‘lynx -dump’或‘netscape -print’。
- 2) 如果LWP::Simple模块的get函数运行失败，它返回什么？
 - a. 出错消息，即“No Document（无文档）”
 - b. 空字符串，即“ ”
 - c. undef

24.4.3 解答

- 1) 答案是a。虽然b和c也可以，但是这两种操作方法很不可靠，使用起来也很难。
- 2) 答案是c。这个答案在程序清单24-2后面的程序分析中作了解释。

24.4.4 实习

- 即使不使用图形模块，你也能很容易创建图24-4中表示民意测验结果的条形图。为了创建这个条形图，你需要一个颜色正确的1×1（或非常小的）.gif文件。若要制作条形图，你只需显示带有相应高度和宽度标号的.gif文件，如下所示：

```
<IMG SRC="small.gif" HEIGHT=20 WIDTH=200 alt="bar graph">
```

大多数图形浏览器都能将这个小型.gif文件放大为规定的大小。

你的任务是：使该民意测验程序输出带有条形图的结果。你必须计算各种类别投票的总数，再将这个总数除以每种类别投票的数量，确定条形图的正确宽度。例如，投票总数是100，一个类别的投票数目为40，那么可以将条形图的最大宽度乘以.4。

- 程序清单24-4中的调查程序可能因为有人多次投票而使调查结果产生偏差。你能设法避免这种情况吗？你可以保存一个文件，里面是所有被调查人地址的列表，不允许出现重复。这可以防止有人在缓存代理的后面进行投票（第23学时介绍了缓存代理）。请设计一种方法，只允许访问该站点的每个人投票一次。（正如你所知道的那样，这种方法不可能做到非常简单明了，它只是一种思路验证方法。）

China-pub.com

下载

China-pub.com

下载

China-pub.com

下载

第四部分

附录

附录 安装模块

China-pub.com

下载

附录 安装模块

在Perl中安装模块并不困难，如果你想真正掌握 Perl，那么学会如何安装这些模块是非常重要的。本附录包含了关于如何安装你需要的模块的信息。



在Perl的文档资料中，你可以得到在各种操作系统下安装模块的详细说明。名叫“Perlmodinstall”的文档甚至包含了在OS / 2和VMS之类的操作系统下安装模块的说明。

A.1 选择正确的模块

首先，必须选择正确的模块。可以通过站点 <http://www.perl.com/CPAN> 上的CPAN寻找你要的模块。你必须确定对哪个模块感兴趣。

CPAN模块大体上是按它们的功能来命名的。例如，Image::size带有一个图形，并且能够报告该图形的大小，该模块可用来与Web页一道运行。不过，有些模块使用一些特殊的名称。LWP是根据Perl库libwww-perl而得名的。

还可以在CPAN上找到模块包。这些模块包含有若干相关的模块，这些模块通常是一些必须要有的模块，它们全部放在一个大模块包中。例如，libnet模块包可以像一个模块那样来安装，不过在安装过程中，你会得到若干个与网络相关的模块。LWP就是libnet模块包中的一部分。



当你安装一个模块时，还会自动获得该模块需要的所有文档。

A.2 在何种操作系统下安装

在下面各节中的每个安装模块的例子中，你将安装来自CPAN的Date::Manip模块。若要安装你自己的模块或模块包，只要用你的模块包取代Date::Manip即可。

A.2.1 在Windows95 / 98 / NT下安装

在Windows下，假定已经安装了来自ActiveState Tool公司的Perl，安装模块的最容易的方法是使用ActiveState Tool公司已打包的模块。

若要在Windows下安装预装模块，首先必须启动Perl Package Manager(PPM)。该实用程序通过提供一个用于模块安装的交互式界面，从而简化了模块安装进程。为了启动PPM，你必须显示一个DOS命令提示符，如图A-1所示，应该连接到Internet。

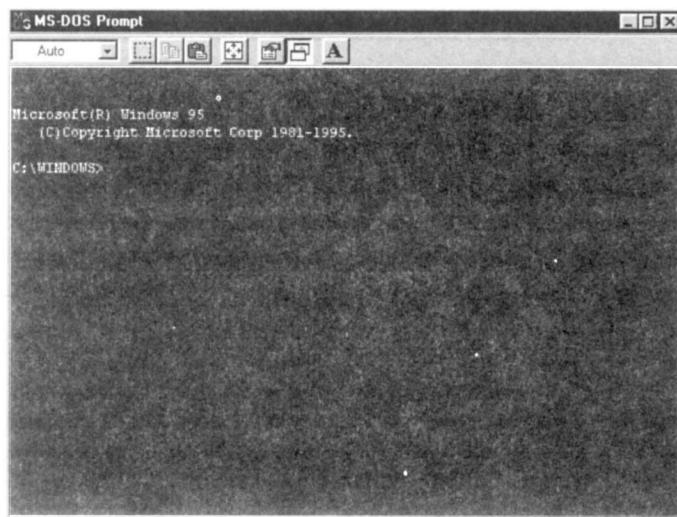
在命令提示符处，键入如下所示的PPM。这时PPM实用程序应该启动运行。如果它没有启动运行，你必须查找与ActiveState Perl一道安装的ppm.bat文件，并用全路径名运行它：

```
C:\Windows>ppm  
PPM interactive shell (1.0.0) - type 'help' for available commands  
PPM>
```

若要搜索某个模块，请使用下面所示的 search命令。之所以你必须使用该命令，原因是 ActiveState并没有CPAN中的所有模块的预装模块包，它只有比较常用的模块。另外，为了进行安装，必须正确拼写模块名。

```
PPM> search Date
Packages available from http://www.ActiveState.com/packages:
  Date-Calc
  Date-Manip
  TimeDate
PPM>
```

图A-1 可以从 DOS命令提示符处开始安装模块的操作



当你找到你想要的模块（比如这个例子中的 Date-Manip）后，就可以使用 install命令，对该模块进行安装，如下所示：

```
PPM> install Date-Manip
Install package 'Date-Manip?' (y/N): y
Installing C:\Perl\html\lib\Date\Manip.html
Installing C:\Perl\htmlhelp\pkg-Date-Manip.chm
Installing C:\Perl\htmlhelp\pkg-Date-Manip.hhc
Installing C:\Perl\site\lib\Date\Manip.pm
Writing C:\Perl\site\lib\auto\Date-Manip/.packlist
PPM>
```

这时Date :: Manip模块就安装好了。

如果你想下载模块包并用人工进行安装（也许PC没有与Internet连网或者它位于防火墙的后面），可以在ActiveState的Web站点（<http://www.ActiveState.com>）上找到下载和人工安装模块的说明。ActiveState维护了一个关于它销售的 Perl产品的特定FAQ，你可以在那里找到必要的说明。



不使用PPM来安装模块，比如使用Windows下你自己的C编译器来进行安装，这不是本书要讲解的内容。Perl的原始产品中包含了在Windows下你自己安装Perl的说明，但这不是初学者能够做的工作。如果你能够进行这项操作，那么自己来安装模块就不会太难，因为安装过程是大致相同的。

A.2.2 在UNIX下使用CPAN来安装模块

在UNIX下安装模块是很有趣并且会遇到许多问题的，但是它也可能是非常容易的。你需要一个ANSI C编译器（用于安装Perl的编译器就很好），如果供应商要求的话，你还必须拥有编译器许可证。你不需要GNU压缩程序gzip / gunzip的拷贝，有些UNIX供应商将它作为一个标准实用程序提供给用户使用。如果你没有这个拷贝，可以从网址 <http://www.fsf.org> 下载一个拷贝。



有些UNIX供应商（比如HP公司）在它们的操作系统中配备了一个C编译器，但是它不是ANSI C编译器，这是C编译器的一个非常简化了的版本，因此你必须花钱购买实际的C编译器，或者交费下载和安装GNU C编译器。

最后一个问题是：你在安装模块的计算机上必须拥有根（管理员）访问权限。通常情况下，Perl是作为整个系统范围的实用程序来安装的。将模块安装到系统目录中，你必须拥有足够的访问权限（即根权限）才能进行这种操作。

Perl产品配有一个称为CPAN的模块，用来帮助你安装其他的模块。若要开始安装操作，你必须使用CPAN模块的shell命令来启动Perl，如下所示：

```
$ perl -MCPAN -e shell
```

如果你是初次运行该命令，CPAN模块就会要求你确定从何处取得Perl的模块以及你想要如何安装这些模块。大多数情况下，默认答案就足以满足你的要求。然后它会问你临时目录的位置在什么地方（这是CPAN对你想使用的目录进行镜像的目录），并且问你是否通过代理程序来访问Internet。

当CPAN结束对你的提问后，你会看到下面这个提示：

```
cpan shell -- CPAN exploration and modules installation (v1.3901)
ReadLine support available (try `install Bundle::CPAN'')
```

```
cpan>
```

在这个提示后面，你可以使用命令*i / pat /*，搜索关于模块包的信息，其中*pat*用于说明你要搜索的模式。例如，若要查找Date::Manip模块，请输入下面这个命令：

```
cpan> i /Manip/
```

CPAN模块必须与一个CPAN服务器取得联系，以使获取该索引的新拷贝。这种情况只有在需要时才会出现，并且这个进程只需很短时间就能完成。当查询结束时，CPAN就会答复下面这样的信息：

```
Distribution      SBECK/DateManip-5.35.tar.gz
Module           Date::Manip      (SBECK/DateManip-5.35.tar.gz)
```

若要安装该模块，请键入下面的命令：

```
cpan> install Date::Manip
```

这时，CPAN模块开始按步骤执行索取、编译、测试和安装模块的各个进程。它显示的信息相当零乱，不过它类似下面这个大大简化了的例子（#后面的注释通常并不出现，这里增加了注释，目的是使它更加清楚）：

```

Running make for SBECK/DateManip-5.35.tar.gz
Fetching with LWP:           # Fetching the module

ftp://ftp.perl.org/pub/perl/CPAN/authors/id/SBECK/DateManip-5.35.t
ar.gz
Writing Makefile for Date::Manip
mkdir blib               # Building the module
mkdir blib/lib
Target "makemakercf" is up to date.
/usr/bin/make -- OK
Running make test          # Testing to ensure it works
    PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib
-I/usr/local/lib/
perl5/5.00502/aix -I/usr/local/lib/perl5/5.00502 -e 'use
Test::Harness qw(&runte
sts $verbose); $verbose=0; runtests @ARGV;' t/*
t/settime.....ok
t/unixdate.....ok
All tests successful.
Files=30, Tests=826, 178 wallclock secs (168.85 cusr + 5.23
csys = 174.08 CPU)
Target "test" is up to date.
/usr/bin/make test -- OK
Running make install        # Installing the module
Target "install" is up to date.
/usr/bin/make install -- OK

```

你得到的输出可能与上面的情况有很大的不同。现在该模块已经测试和安装好了。

A.2.3 在UNIX下用另一种方法安装模块

虽然你可以不使用CPAN模块在UNIX下安装各个模块，但是大多数情况下不需要用下面这种方法来安装模块。我们只是为了完整起见才介绍这种安装方法，但是只要可能，都应该使用CPAN模块来安装各个模块。

首先，必须从CPAN下载你要安装的模块。它是个压缩了的综合模块包。例如，如果要安装的模块是Date::Calc，你必须得到它的新版本，它的名字类似 Date-Calc-X.Y.tar.gz。当你下载了该模块包后，进入该目录，对该模块包进行拆包操作，如下所示：

```
$ gunzip Date-Calc-4_2.tar.gz
$ tar xf Date-Calc-4_2.tar
```

拆包后便产生一个子目录，称为 Date-Calc-4.2。若要转入该子目录，请使用 cd，并键入下面的命令：

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Date::Calc
```

现在你就拥有一个make程序的描述文件，这对于安装进程来说是个必不可少的文件。接着，使用下面这样的make命令，安装该模块：

```
$ make
mkdir blib
mkdir blib/lib
:
Manifying blib/man3/Date::Calc.3
Target "makemakercf" is up to date.
```

这个进程的运行需要花费一定的时间。

下载

在下一个提示符后面，你必须测试该模块，以了解它的安装是否正确。请键入下面这个make test命令：

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib
-I/usr/local/lib/
perl5/5.00502/aix -I/usr/local/lib/perl5/5.00502 -e 'use
Test::Harness qw(&runte
sts $verbose); $verbose=0; runtests @ARGV;' t/*
t/f000.....ok
t/f001.....ok
:
t/f032.....ok
t/f033.....ok
All tests successful.
Files=34, Tests=1823, 14 wallclock secs ( 9.81 cusr + 1.10 csys
= 10.91 CPU)
Target "test" is up to date.
```

你始终都应该运行make test命令，以确保模块安装正确。它能省去你以后好几个小时的调试时间。当测试完成后，必须像下面这样安装该模块。这个操作步骤通常是以根用户身份来进行的，因为安装时必须写入系统目录：

```
$ su
Password: *****
# make install
Installing
/usr/local/lib/perl5/site_perl/5.005/aix/auto/Date/Calc/Calc.so
:
Appending installation info to
/usr/local/lib/perl5/5.00502/aix/perllocal.pod
Target "install" is up to date.
#
```

这样，你的安装操作就完成了。

A.2.4 在Macintosh系统上安装模块

在Macintosh系统上安装模块是比较困难的。你应该查看 MacPerl的FAQ，了解关于可以用来安装模块的方法的信息。MacPerl FAQ可以在网址<http://WWW.macperl.com>上找到。

A.3 当不允许你安装模块时该怎么办

如果你能够在系统上安装程序，你就能够安装模块。你能够这样做，取决于模块的复杂程度和你会遇到何种困难。有时系统管理员不允许你安装某个模块，因为他不想让其他人使用该模块。在某些情况下，只有你或者一组人才想使用某些特定模块，在整个系统范围内安装这些模块太复杂了。

无论哪种情况，在你自己的目录中安装Perl模块的专用拷贝并不难。

首先，必须使用前面给出的说明（只有一些小的例外）安装模块。你可以指定安装程序，将模块安装到特写的目录中。如果在Windows下使用PPM，在你安装模块前，必须告诉PPM，你想将模块安装到另一个目录。为此可以使用下面这样的set命令：

```
PPM> set root c:\myperl
PPM> set build c:\myperl
```

然后该模块被组装在目录C:\myperl中。

在UNIX下，当你使用CPAN模块时，可以使用下面的makepl_arg设置项来设定安装目录：

```
cpan> o conf makepl_arg PREFIX="/home/clintp/perl/lib"
```

或者，如果你使用make实用程序人工安装模块，你可以在第一个代码行上使用PREFIX参数，设定安装目录：

```
$ perl Makefile.PL PREFIX="/home/clintp/perl/lib"
```

无论使用哪种方法，你要安装的模块将被安装到 /home/clintp/perl/lib目录中。如果需要的话，你可以再将该模块移到另一个目录中。

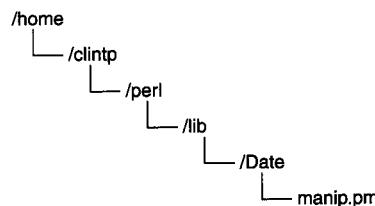


你应该注意，不要将模块在不同操作系统的计算机之间移动。经过编译的模块只能在一种类型的操作系统上运行，这与 Perl本身的情况是一样的。另外，不要试图在不同版本的 Perl之间移动模块，有时它不能运行。在这种情况下，你必须重新安装该模块。

使用安装在特殊位置中的模块

若要使用安装在非标准目录中的模块，必须使用命令 use lib。例如，如果你使用上一节中的说明将模块Date::Manip安装在目录 / home/clintp/perl/lib中，就会得到一个图 A-2所示的文件树。

图A-2 安装Date::Manip模块后形成的文件树



在你的程序开始处，只需要使用下面的代码：

```
use lib '/home/clintp/perl/lib'; # Look for module here
else
use Date::Manip;
```

这时Perl在搜索它自己的目录之前，首先搜索该目录，找出它要的模块。还可以使用这种方法将模块的新版本安装在系统上（以便达到测试目的），但不会改写老的版本，也不会带来不兼容的问题。

China-pub.com

下载

China-pub.com

下载