过$GOROOT 变量访问，在试图编译程序时会产生错误。

作为 Go 发布包的一部分，标准库的源代码是经过预编译的。这些预编译后的文件，称作归档文件（archive file），可以在$GOROOT/pkg 文件夹中找到已经安装的各目标平台和操作系统的归档文件。在图 8-3 里，可以看到扩展名是.a 的文件，这些就是归档文件。

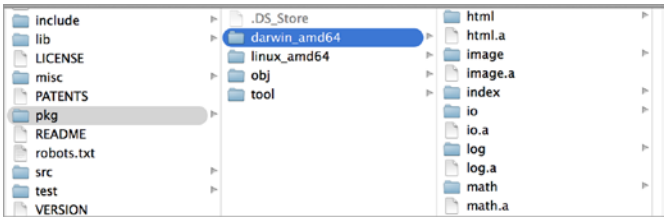

图 8-3　pkg 文件夹中的归档文件的文件夹的视图

这些文件是特殊的 Go 静态库文件，由 Go 的构建工具创建，并在编译和链接最终程序时被使用。归档文件可以让构建的速度更快。但是在构建的过程中，没办法指定这些文件，所以没办法与别人共享这些文件。Go 工具链知道什么时候可以使用已有的.a 文件，什么时候需要从机器上的源代码重新构建。

有了这些背景知识，让我们看一下标准库里的几个包，看看如何用这些包来构建自己的程序。

8.2　记录日志

即便没有表现出来，你的程序依旧可能有 bug。这在软件开发里是很自然的事情。日志是一种找到这些 bug，更好地了解程序工作状态的方法。日志是开发人员的眼睛和耳朵，可以用来跟踪、调试和分析代码。基于此，标准库提供了 `log` 包，可以对日志做一些最基本的配置。根据特殊需要，开发人员还可以自己定制日志记录器。

在 UNIX 里，日志有很长的历史。这些积累下来的经验都体现在 `log` 包的设计里。传统的 CLI（命令行界面）程序直接将输出写到名为 `stdout` 的设备上。所有的操作系统上都有这种设备，这种设备的默认目的地是标准文本输出。默认设置下，终端会显示这些写到 `stdout` 设备上的文本。这种单个目的地的输出用起来很方便，不过你总会碰到需要同时输出程序信息和输出执行细节的情况。这些执行细节被称作日志。当想要记录日志时，你希望能写到不同的目的地，这样就不会将程序的输出和日志混在一起了。

为了解决这个问题，UNIX 架构上增加了一个叫作 `stderr` 的设备。这个设备被创建为日志的默认目的地。这样开发人员就能将程序的输出和日志分离开来。如果想在程序运行时同时看到程序输出和日志，可以将终端配置为同时显示写到 `stdout` 和 `stderr` 的信息。不过，如果用户的程序只记录日志，没有程序输出，更常用的方式是将一般的日志信息写到 `stdout`，将错误或者警告信息写到 `stderr`。

8.2.1 log 包

让我们从 log 包提供的最基本的功能开始，之后再学习如何创建定制的日志记录器。记录日志的目的是跟踪程序什么时候在什么位置做了什么。这就需要通过某些配置在每个日志项上要写的一些信息，如代码清单 8-2 所示。

代码清单 8-2 跟踪日志的样例

```
TRACE: 2009/11/10 23:00:00.000000 /tmpfs/gosandbox-/prog.go:14: message
```

在代码清单 8-2 中，可以看到一个由 log 包产生的日志项。这个日志项包含前缀、日期时间戳、该日志具体是由哪个源文件记录的、源文件记录日志所在行，最后是日志消息。让我们看一下如何配置 log 包来输出这样的日志项，如代码清单 8-3 所示。

代码清单 8-3 listing03.go

```
01 // 这个示例程序展示如何使用最基本的 log 包
02 package main
03
04 import (
05     "log"
06 )
07
08 func init() {
09     log.SetPrefix("TRACE: ")
10     log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
11 }
12
13 func main() {
14     // Println 写到标准日志记录器
15     log.Println("message")
16
17     // Fatalln 在调用 Println()之后会接着调用 os.Exit(1)
18     log.Fatalln("fatal message")
19
20     // Panicln 在调用 Println()之后会接着调用 panic()
21     log.Panicln("panic message")
22 }
```

如果执行代码清单 8-3 中的程序，输出的结果会和代码清单 8-2 所示的输出类似。让我们分析一下代码清单 8-4 中的代码，看看它是如何工作的。

代码清单 8-4 listing03.go：第 08 行到第 11 行

```
08 func init() {
09     log.SetPrefix("TRACE: ")
10     log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
11 }
```

在第 08 行到第 11 行，定义的函数名为 init()。这个函数会在运行 main()之前作为程序

初始化的一部分执行。通常程序会在这个 `init()`函数里配置日志参数，这样程序一开始就能使用 `log` 包进行正确的输出。在这段程序的第 9 行，设置了一个字符串，作为每个日志项的前缀。这个字符串应该是能让用户从一般的程序输出中分辨出日志的字符串。传统上这个字符串的字符会全部大写。

有几个和 `log` 包相关联的标志，这些标志用来控制可以写到每个日志项的其他信息。代码清单 8-5 展示了目前包含的所有标志。

代码清单 8-5　golang.org/src/log/log.go

```
const (
  // 将下面的位使用或运算符连接在一起，可以控制要输出的信息。没有
  // 办法控制这些信息出现的顺序（下面会给出顺序）或者打印的格式
  // （格式在注释里描述）。这些项后面会有一个冒号：
  //    2009/01/23 01:23:23.123123 /a/b/c/d.go:23: message

  // 日期: 2009/01/23
  Ldate = 1 << iota

  // 时间: 01:23:23
  Ltime

  // 毫秒级时间: 01:23:23.123123。该设置会覆盖 Ltime 标志
  Lmicroseconds

  // 完整路径的文件名和行号: /a/b/c/d.go:23
  Llongfile

  // 最终的文件名元素和行号: d.go:23
  // 覆盖 Llongfile
  Lshortfile

  // 标准日志记录器的初始值
  LstdFlags = Ldate | Ltime
)
```

代码清单 8-5 是从 `log` 包里直接摘抄的源代码。这些标志被声明为常量，这个代码块中的第一个常量叫作 `Ldate`，使用了特殊的语法来声明，如代码清单 8-6 所示。

代码清单 8-6　声明 `Ldate` 常量

```
// 日期: 2009/01/23
Ldate = 1 << iota
```

关键字 `iota` 在常量声明区里有特殊的作用。这个关键字让编译器为每个常量复制相同的表达式，直到声明区结束，或者遇到一个新的赋值语句。关键字 `iota` 的另一个功能是，`iota` 的初始值为 0，之后 `iota` 的值在每次处理为常量后，都会自增 1。让我们更仔细地看一下这个关键字，如代码清单 8-7 所示。

代码清单 8-7　使用关键字 iota

```
const (
  Ldate = 1 << iota  // 1 << 0 = 000000001 = 1
  Ltime              // 1 << 1 = 000000010 = 2
  Lmicroseconds      // 1 << 2 = 000000100 = 4
  Llongfile          // 1 << 3 = 000001000 = 8
  Lshortfile         // 1 << 4 = 000010000 = 16
  ...
)
```

代码清单 8-7 展示了常量声明背后的处理方法。操作符<<对左边的操作数执行按位左移操作。在每个常量声明时，都将 1 按位左移 iota 个位置。最终的效果使为每个常量赋予一个独立位置的位，这正好是标志希望的工作方式。

常量 LstdFlags 展示了如何使用这些标志，如代码清单 8-8 所示。

代码清单 8-8　声明 LstdFlags 常量

```
const (
  ...
  LstdFlags = Ldate(1) | Ltime(2) = 00000011 = 3
)
```

在代码清单 8-8 中看到，因为使用了复制操作符，LstdFlags 打破了 iota 常数链。由于有|运算符用于执行或操作，常量 LstdFlags 被赋值为 3。对位进行或操作等同于将每个位置的位组合在一起，作为最终的值。如果对位 1 和 2 进行或操作，最终的结果就是 3。

让我们看一下我们要如何设置日志标志，如代码清单 8-9 所示。

代码清单 8-9　listing03.go：第 08 行到第 11 行

```
08 func init() {
09     ...
10     log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
11 }
```

这里我们将 Ldate、Lmicroseconds 和 Llongfile 标志组合在一起，将该操作的值传入 SetFlags 函数。这些标志值组合在一起后，最终的值是 13，代表第 1、3 和 4 位为 1（00001101）。由于每个常量表示单独一个位，这些标志经过或操作组合后的值，可以表示每个需要的日志参数。之后 log 包会按位检查这个传入的整数值，按照需求设置日志项记录的信息。

初始完 log 包后，可以看一下 main()函数，看它是是如何写消息的，如代码清单 8-10 所示。

代码清单 8-10　listing03.go：第 13 行到第 22 行

```
13 func main() {
14     // Println 写到标准日志记录器
15     log.Println("message")
16
17     // Fatalln 在调用 Println()之后会接着调用 os.Exit(1)
```

```
18      log.Fatalln("fatal message")
19
20      // Panicln 在调用 Println()之后会接着调用 panic()
21      log.Panicln("panic message")
22  }
```

代码清单 8-10 展示了如何使用 3 个函数 `Println`、`Fatalln` 和 `Panicln` 来写日志消息。这些函数也有可以格式化消息的版本，只需要用 `f` 替换结尾的 `ln`。`Fatal` 系列函数用来写日志消息，然后使用 `os.Exit(1)`终止程序。`Panic` 系列函数用来写日志消息，然后触发一个 `panic`。除非程序执行 `recover` 函数，否则会导致程序打印调用栈后终止。`Print` 系列函数是写日志消息的标准方法。

`log` 包有一个很方便的地方就是，这些日志记录器是多 goroutine 安全的。这意味着在多个 goroutine 可以同时调用来自同一个日志记录器的这些函数，而不 会有彼此间的写冲突。标准日志记录器具有这一性质，用户定制的日志记录器也应该满足这一性质。

现在知道了如何使用和配置 `log` 包，让我们看一下如何创建一个定制的日志记录器，以便可以让不同等级的日志写到不同的目的地。

8.2.2 定制的日志记录器

要想创建一个定制的日志记录器，需要创建一个 `Logger` 类型值。可以给每个日志记录器配置一个单独的目的地，并独立设置其前缀和标志。让我们来看一个示例程序，这个示例程序展示了如何创建不同的 `Logger` 类型的指针变量来支持不同的日志等级，如代码清单 8-11 所示。

代码清单 8-11　listing11.go

```
01 // 这个示例程序展示如何创建定制的日志记录器
02 package main
03
04 import (
05      "io"
06      "io/ioutil"
07      "log"
08      "os"
09 )
10
11 var (
12      Trace   *log.Logger // 记录所有日志
13      Info    *log.Logger // 重要的信息
14      Warning *log.Logger // 需要注意的信息
15      Error   *log.Logger // 非常严重的问题
16 )
17
18 func init() {
19      file, err := os.OpenFile("errors.txt",
20          os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
21      if err != nil {
```

```
22          log.Fatalln("Failed to open error log file:", err)
23      }
24
25      Trace = log.New(ioutil.Discard,
26          "TRACE: ",
27          log.Ldate|log.Ltime|log.Lshortfile)
28
29      Info = log.New(os.Stdout,
30          "INFO: ",
31          log.Ldate|log.Ltime|log.Lshortfile)
32
33      Warning = log.New(os.Stdout,
34          "WARNING: ",
35          log.Ldate|log.Ltime|log.Lshortfile)
36
37      Error = log.New(io.MultiWriter(file, os.Stderr),
38          "ERROR: ",
39          log.Ldate|log.Ltime|log.Lshortfile)
40  }
41
42  func main() {
43      Trace.Println("I have something standard to say")
44      Info.Println("Special Information")
45      Warning.Println("There is something you need to know about")
46      Error.Println("Something has failed")
47  }
```

代码清单 8-11 展示了一段完整的程序，这段程序创建了 4 种不同的 `Logger` 类型的指针变量，分别命名为 `Trace`、`Info`、`Warning` 和 `Error`。每个变量使用不同的配置，用来表示不同的重要程度。让我们来分析一下这段代码是如何工作的。

在第 11 行到第 16 行，我们为 4 个日志等级声明了 4 个 `Logger` 类型的指针变量，如代码清单 8-12 所示。

代码清单 8-12　listing11.go：第 11 行到第 16 行

```
11  var (
12      Trace   *log.Logger // 记录所有日志
13      Info    *log.Logger // 重要的信息
14      Warning *log.Logger // 需要注意的信息
15      Error   *log.Logger // 非常严重的问题
16  )
```

在代码清单 8-12 中可以看到对 `Logger` 类型的指针变量的声明。我们使用的变量名很简短，但是含义明确。接下来，让我们看一下 `init()`函数的代码是如何创建每个 `Logger` 类型的值并将其地址赋给每个变量的，如代码清单 8-13 所示。

代码清单 8-13　listing11.go：第 25 行到第 39 行

```
25      Trace = log.New(ioutil.Discard,
26          "TRACE: ",
27          log.Ldate|log.Ltime|log.Lshortfile)
```

```
28
29      Info = log.New(os.Stdout,
30          "INFO: ",
31          log.Ldate|log.Ltime|log.Lshortfile)
32
33      Warning = log.New(os.Stdout,
34          "WARNING: ",
35          log.Ldate|log.Ltime|log.Lshortfile)
36
37      Error = log.New(io.MultiWriter(file, os.Stderr),
38          "ERROR: ",
39          log.Ldate|log.Ltime|log.Lshortfile)
```

为了创建每个日志记录器，我们使用了 log 包的 New 函数，它创建并正确初始化一个 Logger 类型的值。函数 New 会返回新创建的值的地址。在 New 函数创建对应值的时候，我们需要给它传入一些参数，如代码清单 8-14 所示。

代码清单 8-14　golang.org/src/log/log.go

```
// New 创建一个新的 Logger。out 参数设置日志数据将被写入的目的地
// 参数 prefix 会在生成的每行日志的最开始出现
// 参数 flag 定义日志记录包含哪些属性
func New(out io.Writer, prefix string, flag int) *Logger {
    return &Logger{out: out, prefix: prefix, flag: flag}
}
```

代码清单 8-14 展示了来自 log 包的源代码里的 New 函数的声明。第一个参数 out 指定了日志要写到的目的地。这个参数传入的值必须实现了 io.Writer 接口。第二个参数 prefix 是之前看到的前缀，而日志的标志则是最后一个参数。

在这个程序里，Trace 日志记录器使用了 ioutil 包里的 Discard 变量作为写到的目的地，如代码清单 8-15 所示。

代码清单 8-15　listing11.go：第 25 行到第 27 行

```
25      Trace = log.New(ioutil.Discard,
26          "TRACE: ",
27          log.Ldate|log.Ltime|log.Lshortfile)
```

变量 Discard 有一些有意思的属性，如代码清单 8-16 所示。

代码清单 8-16　golang.org/src/io/ioutil/ioutil.go

```
// devNull 是一个用 int 作为基础类型的类型
type devNull int

// Discard 是一个 io.Writer，所有的 Write 调用都不会有动作，但是会成功返回
var Discard io.Writer = devNull(0)

// io.Writer 接口的实现
func (devNull) Write(p []byte) (int, error) {
```

```
    return len(p), nil
}
```

代码清单 8-16 展示了 Discard 变量的声明以及相关的实现。Discard 变量的类型被声明为 io.Writer 接口类型，并被给定了一个 devNull 类型的值 0。基于 devNull 类型实现的 Write 方法，会忽略所有写入这一变量的数据。当某个等级的日志不重要时，使用 Discard 变量可以禁用这个等级的日志。

日志记录器 Info 和 Warning 都使用 stdout 作为日志输出，如代码清单 8-17 所示。

代码清单 8-17　listing11.go：第 29 行到第 35 行

```
29      Info = log.New(os.Stdout,
30          "INFO: ",
31          log.Ldate|log.Ltime|log.Lshortfile)
32
33      Warning = log.New(os.Stdout,
34          "WARNING: ",
35          log.Ldate|log.Ltime|log.Lshortfile)
```

变量 Stdout 的声明也有一些有意思的地方，如代码清单 8-18 所示。

代码清单 8-18　golang.org/src/os/file.go

```
// Stdin、Stdout 和 Stderr 是已经打开的文件，分别指向标准输入、标准输出和
// 标准错误的文件描述符
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)

os/file_unix.go

// NewFile 用给出的文件描述符和名字返回一个新 File
func NewFile(fd uintptr, name string) *File {
```

在代码清单 8-18 中可以看到 3 个变量的声明，分别表示所有操作系统里都有的 3 个标准输入/输出，即 Stdin、Stdout 和 Stderr。这 3 个变量都被声明为 File 类型的指针，这个类型实现了 io.Writer 接口。有了这个知识，我们来看一下最后的日志记录器 Error，如代码清单 8-19 所示。

代码清单 8-19　listing11.go：第 37 行到第 39 行

```
37      Error = log.New(io.MultiWriter(file, os.Stderr),
38          "ERROR: ",
39          log.Ldate|log.Ltime|log.Lshortfile)
```

在代码清单 8-19 中可以看到 New 函数的第一个参数来自一个特殊的函数。这个特殊的函数就是 io 包里的 MultiWriter 函数，如代码清单 8-20 所示。

代码清单 8-20　包 io 里的 MultiWriter 函数的声明

```
io.MultiWriter(file, os.Stderr)
```

代码清单 8-20 单独展示了 MultiWriter 函数的调用。这个函数调用会返回一个 io.Writer 接口类型值，这个值包含之前打开的文件 file，以及 stderr。MultiWriter 函数是一个变参函数，可以接受任意个实现了 io.Writer 接口的值。这个函数会返回一个 io.Writer 值，这个值会把所有传入的 io.Writer 的值绑在一起。当对这个返回值进行写入时，会向所有绑在一起的 io.Writer 值做写入。这让类似 log.New 这样的函数可以同时向多个 Writer 做输出。现在，当我们使用 Error 记录器记录日志时，输出会同时写到文件和 stderr。

现在知道了该如何创建定制的记录器了，让我们看一下如何使用这些记录器来写日志消息，如代码清单 8-21 所示。

代码清单 8-21　listing11.go：第 42 行到第 47 行

```
42 func main() {
43     Trace.Println("I have something standard to say")
44     Info.Println("Special Information")
45     Warning.Println("There is something you need to know about")
46     Error.Println("Something has failed")
47 }
```

代码清单 8-21 展示了代码清单 8-11 中的 main()函数。在第 43 行到第 46 行，我们用自己创建的每个记录器写一条消息。每个记录器变量都包含一组方法，这组方法与 log 包里实现的那组函数完全一致，如代码清单 8-22 所示。

代码清单 8-22 展示了为 Logger 类型实现的所有方法。

代码清单 8-22　不同的日志方法的声明

```
func (l *Logger) Fatal(v ...interface{})
func (l *Logger) Fatalf(format string, v ...interface{})
func (l *Logger) Fatalln(v ...interface{})
func (l *Logger) Flags() int
func (l *Logger) Output(calldepth int, s string) error
func (l *Logger) Panic(v ...interface{})
func (l *Logger) Panicf(format string, v ...interface{})
func (l *Logger) Panicln(v ...interface{})
func (l *Logger) Prefix() string
func (l *Logger) Print(v ...interface{})
func (l *Logger) Printf(format string, v ...interface{})
func (l *Logger) Println(v ...interface{})
func (l *Logger) SetFlags(flag int)
func (l *Logger) SetPrefix(prefix string)
```

8.2.3　结论

log 包的实现，是基于对记录日志这个需求长时间的实践和积累而形成的。将输出写到

stdout，将日志记录到 stderr，是很多基于命令行界面（CLI）的程序的惯常使用的方法。不过如果你的程序只输出日志，那么使用 stdout、stderr 和文件来记录日志是很好的做法。

标准库的 log 包包含了记录日志需要的所有功能，推荐使用这个包。我们可以完全信任这个包的实现，不仅仅是因为它是标准库的一部分，而且社区也广泛使用它。

8.3 编码/解码

许多程序都需要处理或者发布数据，不管这个程序是要使用数据库，进行网络调用，还是与分布式系统打交道。如果程序需要处理 XML 或者 JSON，可以使用标准库里名为 xml 和 json 的包，它们可以处理这些格式的数据。如果想实现自己的数据格式的编解码，可以将这些包的实现作为指导。

在今天，JSON 远比 XML 流行。这主要是因为与 XML 相比，使用 JSON 需要处理的标签更少。而这就意味着网络传输时每个消息的数据更少，从而提升整个系统的性能。而且，JSON 可以转换为 BSON（Binary JavaScript Object Notation，二进制 JavaScript 对象标记），进一步缩小每个消息的数据长度。因此，我们会学习如何在 Go 应用程序里处理并发布 JSON。处理 XML 的方法也很类似。

8.3.1 解码 JSON

我们要学习的处理 JSON 的第一个方面是，使用 json 包的 NewDecoder 函数以及 Decode 方法进行解码。如果要处理来自网络响应或者文件的 JSON，那么一定会用到这个函数及方法。让我们来看一个处理 Get 请求响应的 JSON 的例子，这个例子使用 http 包获取 Google 搜索 API 返回的 JSON。代码清单 8-23 展示了这个响应的内容。

代码清单 8-23　Google 搜索 API 的 JSON 响应例子

```
{
    "responseData": {
        "results": [
            {
                "GsearchResultClass": "GwebSearch",
                "unescapedUrl": "https://www.reddit.com/r/golang",
                "url": "https://www.reddit.com/r/golang",
                "visibleUrl": "www.reddit.com",
                "cacheUrl": "http://www.google.com/search?q=cache:W...",
                "title": "r/\u003cb\u003eGolang\u003c/b\u003e - Reddit",
                "titleNoFormatting": "r/Golang - Reddit",
                "content": "First Open Source \u003cb\u003eGolang\u..."
            },
            {
                "GsearchResultClass": "GwebSearch",
                "unescapedUrl": "http://tour.golang.org/",
```

```
                "url": "http://tour.golang.org/",
                "visibleUrl": "tour.golang.org",
                "cacheUrl": "http://www.google.com/search?q=cache:O...",
                "title": "A Tour of Go",
                "titleNoFormatting": "A Tour of Go",
                "content": "Welcome to a tour of the Go programming ..."
            }
        ]
    }
}
```

代码清单 8-24 给出的是如何获取响应并将其解码到一个结构类型里的例子。

代码清单 8-24　listing24.go

```
01 // 这个示例程序展示如何使用 json 包和 NewDecoder 函数
02 // 来解码 JSON 响应
03 package main
04
05 import (
06     "encoding/json"
07     "fmt"
08     "log"
09     "net/http"
10 )
11
12 type (
13     // gResult 映射到从搜索拿到的结果文档
14     gResult struct {
15         GsearchResultClass string `json:"GsearchResultClass"`
16         UnescapedURL       string `json:"unescapedUrl"`
17         URL                string `json:"url"`
18         VisibleURL         string `json:"visibleUrl"`
19         CacheURL           string `json:"cacheUrl"`
20         Title              string `json:"title"`
21         TitleNoFormatting  string `json:"titleNoFormatting"`
22         Content            string `json:"content"`
23     }
24
25     // gResponse 包含顶级的文档
26     gResponse struct {
27         ResponseData struct {
28             Results []gResult `json:"results"`
29         } `json:"responseData"`
30     }
31 )
32
33 func main() {
34     uri := "http://ajax.googleapis.com/ajax/services/search/web?v=1.0&rsz=8&q=golang"
35
36     // 向 Google 发起搜索
37     resp, err := http.Get(uri)
38     if err != nil {
39         log.Println("ERROR:", err)
```

```
40          return
41      }
42      defer resp.Body.Close()
43
44      // 将 JSON 响应解码到结构类型
45      var gr gResponse
46      err = json.NewDecoder(resp.Body).Decode(&gr)
47      if err != nil {
48          log.Println("ERROR:", err)
49          return
50      }
51
52      fmt.Println(gr)
53  }
```

代码清单 8-24 中代码的第 37 行，展示了程序做了一个 HTTP `Get` 调用，希望从 Google 得到一个 JSON 文档。之后，在第 46 行使用 `NewDecoder` 函数和 `Decode` 方法，将响应返回的 JSON 文档解码到第 26 行声明的一个结构类型的变量里。在第 52 行，将这个变量的值写到 `stdout`。

如果仔细看第 26 行和第 14 行的 `gResponse` 和 `gResult` 的类型声明，你会注意到每个字段最后使用单引号声明了一个字符串。这些字符串被称作标签（tag），是提供每个字段的元信息的一种机制，将 JSON 文档和结构类型里的字段一一映射起来。如果不存在标签，编码和解码过程会试图以大小写无关的方式，直接使用字段的名字进行匹配。如果无法匹配，对应的结构类型里的字段就包含其零值。

执行 HTTP `Get` 调用和解码 JSON 到结构类型的具体技术细节都由标准库包办了。让我们看一下标准库里 `NewDecoder` 函数和 `Decode` 方法的声明，如代码清单 8-25 所示。

代码清单 8-25　golang.org/src/encoding/json/stream.go

```
// NewDecoder 返回从 r 读取的解码器
//
// 解码器自己会进行缓冲，而且可能会从 r 读比解码 JSON 值
// 所需的更多的数据
func NewDecoder(r io.Reader) *Decoder

// Decode 从自己的输入里读取下一个编码好的 JSON 值，
// 并存入 v 所指向的值里
//
// 要知道从 JSON 转换为 Go 的值的细节，
// 请查看 Unmarshal 的文档
func (dec *Decoder) Decode(v interface{}) error
```

在代码清单 8-25 中可以看到 `NewDecoder` 函数接受一个实现了 `io.Reader` 接口类型的值作为参数。在下一节，我们会更详细地介绍 `io.Reader` 和 `io.Writer` 接口，现在只需要知道标准库里的许多不同类型，包括 `http` 包里的一些类型，都实现了这些接口就行。只要类型实现了这些接口，就可以自动获得许多功能的支持。

函数 `NewDecoder` 返回一个指向 `Decoder` 类型的指针值。由于 Go 语言支持复合语句调用，

可以直接调用从 NewDecoder 函数返回的值的 Decode 方法，而不用把这个返回值存入变量。在代码清单 8-25 里，可以看到 Decode 方法接受一个 interface{}类型的值做参数，并返回一个 error 值。

在第 5 章中曾讨论过，任何类型都实现了一个空接口 interface{}。这意味着 Decode 方法可以接受任意类型的值。使用反射，Decode 方法会拿到传入值的类型信息。然后，在读取 JSON 响应的过程中，Decode 方法会将对应的响应解码为这个类型的值。这意味着用户不需要创建对应的值，Decode 会为用户做这件事情，如代码清单 8-26 所示。

在代码清单 8-26 中，我们向 Decode 方法传入了指向 gResponse 类型的指针变量的地址，而这个地址的实际值为 nil。该方法调用后，这个指针变量会被赋给一个 gResponse 类型的值，并根据解码后的 JSON 文档做初始化。

代码清单 8-26　使用 Decode 方法

```
var gr *gResponse
err = json.NewDecoder(resp.Body).Decode(&gr)
```

有时，需要处理的 JSON 文档会以 string 的形式存在。在这种情况下，需要将 string 转换为 byte 切片（[]byte），并使用 json 包的 Unmarshal 函数进行反序列化的处理，如代码清单 8-27 所示。

代码清单 8-27　listing27.go

```
01 // 这个示例程序展示如何解码 JSON 字符串
02 package main
03
04 import (
05     "encoding/json"
06     "fmt"
07     "log"
08 )
09
10 // Contact 结构代表我们的 JSON 字符串
11 type Contact struct {
12     Name    string `json:"name"`
13     Title   string `json:"title"`
14     Contact struct {
15         Home string `json:"home"`
16         Cell string `json:"cell"`
17     } `json:"contact"`
18 }
19
20 // JSON 包含用于反序列化的演示字符串
21 var JSON = `{
22     "name": "Gopher",
23     "title": "programmer",
24     "contact": {
25         "home": "415.333.3333",
```

```
26          "cell": "415.555.5555"
27      }
28  }`
29
30  func main() {
31      // 将 JSON 字符串反序列化到变量
32      var c Contact
33      err := json.Unmarshal([]byte(JSON), &c)
34      if err != nil {
35          log.Println("ERROR:", err)
36          return
37      }
38
39      fmt.Println(c)
40  }
```

在代码清单 8-27 中，我们的例子将 JSON 文档保存在一个字符串变量里，并使用 `Unmarshal` 函数将 JSON 文档解码到一个结构类型的值里。如果运行这个程序，会得到代码清单 8-28 所示的输出。

代码清单 8-28　listing27.go 的输出

```
{Gopher programmer {415.333.3333 415.555.5555}}
```

有时，无法为 JSON 的格式声明一个结构类型，而是需要更加灵活的方式来处理 JSON 文档。在这种情况下，可以将 JSON 文档解码到一个 `map` 变量中，如代码清单 8-29 所示。

代码清单 8-29　listing29.go

```
01 // 这个示例程序展示如何解码 JSON 字符串
02 package main
03
04 import (
05     "encoding/json"
06     "fmt"
07     "log"
08 )
09
10 // JSON 包含要反序列化的样例字符串
11 var JSON = `{
12     "name": "Gopher",
13     "title": "programmer",
14     "contact": {
15         "home": "415.333.3333",
16         "cell": "415.555.5555"
17     }
18 }`
19
20 func main() {
21     // 将 JSON 字符串反序列化到 map 变量
22     var c map[string]interface{}
23     err := json.Unmarshal([]byte(JSON), &c)
24     if err != nil {
25         log.Println("ERROR:", err)
```

```
26          return
27      }
28
29      fmt.Println("Name:", c["name"])
30      fmt.Println("Title:", c["title"])
31      fmt.Println("Contact")
32      fmt.Println("H:", c["contact"].(map[string]interface{})["home"])
33      fmt.Println("C:", c["contact"].(map[string]interface{})["cell"])
34  }
```

代码清单 8-29 中的程序修改自代码清单 8-27，将其中的结构类型变量替换为 `map` 类型的变量。变量 `c` 声明为一个 `map` 类型，其键是 `string` 类型，其值是 `interface{}`类型。这意味着这个 `map` 类型可以使用任意类型的值作为给定键的值。虽然这种方法为处理 JSON 文档带来了很大的灵活性，但是却有一个小缺点。让我们看一下访问 `contact` 子文档的 `home` 字段的代码，如代码清单 8-30 所示。

代码清单 8-30　访问解组后的映射的字段的代码

```
fmt.Println("\tHome:", c["contact"].(map[string]interface{})["home"])
```

因为每个键的值的类型都是 `interface{}`，所以必须将值转换为合适的类型，才能处理这个值。代码清单 8-30 展示了如何将 `contact` 键的值转换为另一个键是 `string` 类型，值是 `interface{}`类型的 `map` 类型。这有时会使映射里包含另一个文档的 JSON 文档处理起来不那么友好。但是，如果不需要深入正在处理的 JSON 文档，或者只打算做很少的处理，因为不需要声明新的类型，使用 `map` 类型会很快。

8.3.2　编码 JSON

我们要学习的处理 JSON 的第二个方面是，使用 `json` 包的 `MarshalIndent` 函数进行编码。这个函数可以很方便地将 Go 语言的 `map` 类型的值或者结构类型的值转换为易读格式的 JSON 文档。*序列化*（marshal）是指将数据转换为 JSON 字符串的过程。下面是一个将 `map` 类型转换为 JSON 字符串的例子，如代码清单 8-31 所示。

代码清单 8-31　listing31.go

```
01 // 这个示例程序展示如何序列化 JSON 字符串
02 package main
03
04 import (
05     "encoding/json"
06     "fmt"
07     "log"
08 )
09
10 func main() {
11     // 创建一个保存键值对的映射
```

```
    c := make(map[string]interface{})
    c["name"] = "Gopher"
    c["title"] = "programmer"
    c["contact"] = map[string]interface{}{
        "home": "415.333.3333",
        "cell": "415.555.5555",
    }

    // 将这个映射序列化到 JSON 字符串
    data, err := json.MarshalIndent(c, "", "    ")
    if err != nil {
        log.Println("ERROR:", err)
        return
    }

    fmt.Println(string(data))
}
```

代码清单 8-31 展示了如何使用 `json` 包的 `MarshalIndent` 函数将一个 `map` 值转换为 JSON 字符串。函数 `MarshalIndent` 返回一个 `byte` 切片，用来保存 JSON 字符串和一个 `error` 值。下面来看一下 `json` 包中 `MarshalIndent` 函数的声明，如代码清单 8-32 所示。

代码清单 8-32　golang.org/src/encoding/json/encode.go

```
// MarshalIndent 很像 Marshal，只是用缩进对输出进行格式化
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error) {
```

在 `MarshalIndent` 函数里再一次看到使用了空接口类型 `interface{}`。函数 `MarshalIndent` 会使用反射来确定如何将 `map` 类型转换为 JSON 字符串。

如果不需要输出带有缩进格式的 JSON 字符串，`json` 包还提供了名为 `Marshal` 的函数来进行解码。这个函数产生的 JSON 字符串很适合作为在网络响应（如 Web API）的数据。函数 `Marshal` 的工作原理和函数 `MarshalIndent` 一样，只不过没有用于前缀 `prefix` 和缩进 `indent` 的参数。

8.3.3　结论

在标准库里都已经提供了处理 JSON 和 XML 格式所需要的诸如解码、反序列化以及序列化数据的功能。随着每次 Go 语言新版本的发布，这些包的执行速度也越来越快。这些包是处理 JSON 和 XML 的最佳选择。由于有反射包和标签的支持，可以很方便地声明一个结构类型，并将其中的字段映射到需要处理和发布的文档的字段。由于 `json` 包和 `xml` 包都支持 `io.Reader` 和 `io.Writer` 接口，用户不用担心自己的 JSON 和 XML 文档源于哪里。所有的这些特性都让处理 JSON 和 XML 变得很容易。

8.4　输入和输出

类 UNIX 的操作系统如此伟大的一个原因是，一个程序的输出可以是另一个程序的输入这一

理念。依照这个哲学，这类操作系统创建了一系列的简单程序，每个程序只做一件事，并把这件事做得非常好。之后，将这些程序组合在一起，可以创建一些脚本做一些很惊艳的事情。这些程序使用 stdin 和 stdout 设备作为通道，在进程之间传递数据。

同样的理念扩展到了标准库的 io 包，而且提供的功能很神奇。这个包可以以流的方式高效处理数据，而不用考虑数据是什么，数据来自哪里，以及数据要发送到哪里的问题。与 stdout 和 stdin 对应，这个包含有 io.Writer 和 io.Reader 两个接口。所有实现了这两个接口的类型的值，都可以使用 io 包提供的所有功能，也可以用于其他包里接受这两个接口的函数以及方法。这是用接口类型来构造函数和 API 最美妙的地方。开发人员可以基于这些现有功能进行组合，利用所有已经存在的实现，专注于解决业务问题。

有了这个概念，让我们先看一下 io.Wrtier 和 io.Reader 接口的声明，然后再来分析展示了 io 包神奇功能的代码。

8.4.1 Writer 和 Reader 接口

io 包是围绕着实现了 io.Writer 和 io.Reader 接口类型的值而构建的。由于 io.Writer 和 io.Reader 提供了足够的抽象，这些 io 包里的函数和方法并不知道数据的类型，也不知道这些数据在物理上是如何读和写的。让我们先来看一下 io.Writer 接口的声明，如代码清单 8-33 所示。

代码清单 8-33 io.Writer 接口的声明

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

代码清单 8-33 展示了 io.Writer 接口的声明。这个接口声明了唯一一个方法 Write，这个方法接受一个 byte 切片，并返回两个值。第一个值是写入的字节数，第二个值是 error 错误值。代码清单 8-34 给出的是实现这个方法的一些规则。

代码清单 8-34 io.Writer 接口的文档

```
Write 从 p 里向底层的数据流写入 len(p) 字节的数据。这个方法返回从 p 里写出的字节
数（0 <= n <= len(p)），以及任何可能导致写入提前结束的错误。Write 在返回 n
< len(p) 的时候，必须返回某个非 nil 值的 error。Write 绝不能改写切片里的数据，
哪怕是临时修改也不行。
```

代码清单 8-34 中的规则来自标准库。这些规则意味着 Write 方法的实现需要试图写入被传入的 byte 切片里的所有数据。但是，如果无法全部写入，那么该方法就一定会返回一个错误。返回的写入字节数可能会小于 byte 切片的长度，但不会出现大于的情况。最后，不管什么情况，都不能修改 byte 切片里的数据。

让我们看一下 Reader 接口的声明，如代码清单 8-35 所示。

代码清单 8-35 io.Reader 接口的声明

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

代码清单 8-35 中的 io.Reader 接口声明了一个方法 Read,这个方法接受一个 byte 切片，并返回两个值。第一个值是读入的字节数，第二个值是 error 错误值。代码清单 8-36 给出的是实现这个方法的一些规则。

代码清单 8-36 io.Reader 接口的文档

```
(1) Read 最多读入 len(p)字节，保存到 p。这个方法返回读入的字节数（0 <= n
<= len(p)）和任何读取时发生的错误。即便 Read 返回的 n < len(p)，方法也可
能使用所有 p 的空间存储临时数据。如果数据可以读取，但是字节长度不足 len(p)，
习惯上 Read 会立刻返回可用的数据，而不等待更多的数据。

(2) 当成功读取 n > 0 字节后，如果遇到错误或者文件读取完成，Read 方法会返回
读入的字节数。方法可能会在本次调用返回一个非 nil 的错误，或者在下一次调用时返
回错误（同时 n == 0）。这种情况的的一个例子是，在输入的流结束时，Read 会返回
非零的读取字节数，可能会返回 err == EOF，也可能会返回 err == nil。无论如何，
下一次调用 Read 应该返回 0, EOF。

(3) 调用者在返回的 n > 0 时，总应该先处理读入的数据，再处理错误 err。这样才
能正确操作读取一部分字节后发生的 I/O 错误。EOF 也要这样处理。

(4) Read 的实现不鼓励返回 0 个读取字节的同时，返回 nil 值的错误。调用者需要将
这种返回状态视为没有做任何操作，而不是遇到读取结束。
```

标准库里列出了实现 Read 方法的 4 条规则。第一条规则表明，该实现需要试图读取数据来填满被传入的 byte 切片。允许出现读取的字节数小于 byte 切片的长度，并且如果在读取时已经读到数据但是数据不足以填满 byte 切片时，不应该等待新数据，而是要直接返回已读数据。

第二条规则提供了应该如何处理达到文件末尾（EOF）的情况的指导。当读到最后一个字节时，可以有两种选择。一种是 Read 返回最终读到的字节数，并且返回 EOF 作为错误值，另一种是返回最终读到的字节数，并返回 nil 作为错误值。在后一种情况下，下一次读取的时候，由于没有更多的数据可供读取，需要返回 0 作为读到的字节数，以及 EOF 作为错误值。

第三条规则是给调用 Read 的人的建议。任何时候 Read 返回了读取的字节数，都应该优先处理这些读取到的字节，再去检查 EOF 错误值或者其他错误值。最终，第四条约束建议 Read 方法的实现永远不要返回 0 个读取字节的同时返回 nil 作为错误值。如果没有读到值，Read 应该总是返回一个错误。

现在知道了 io.Writer 和 io.Reader 接口是什么样子的，以及期盼的行为是什么，让我们看一下如何在程序里使用这些接口以及 io 包。

8.4.2 整合并完成工作

这个例子展示标准库里不同包是如何通过支持实现了 io.Writer 接口类型的值来一起完成

工作的。这个示例里使用了 bytes、fmt 和 os 包来进行缓冲、拼接和写字符串到 stdout，如代码清单 8-37 所示。

代码清单 8-37 listing37.go

```
01 // 这个示例程序展示来自不同标准库的不同函数是如何
02 // 使用 io.Writer 接口的
03 package main
04
05 import (
06     "bytes"
07     "fmt"
08     "os"
09 )
10
11 // main 是应用程序的入口
12 func main() {
13     // 创建一个 Buffer 值，并将一个字符串写入 Buffer
14     // 使用实现 io.Writer 的 Write 方法
15     var b bytes.Buffer
16     b.Write([]byte("Hello "))
17
18     // 使用 Fprintf 来将一个字符串拼接到 Buffer 里
19     // 将 bytes.Buffer 的地址作为 io.Writer 类型值传入
20     fmt.Fprintf(&b, "World!")
21
22     // 将 Buffer 的内容输出到标准输出设备
23     // 将 os.File 值的地址作为 io.Writer 类型值传入
24     b.WriteTo(os.Stdout)
25 }
```

运行代码清单 8-37 中的程序会得到代码清单 8-38 所示的输出。

代码清单 8-38 listing37.go 的输出

```
Hello World!
```

这个程序使用了标准库的 3 个包来将"Hello World!"输出到终端窗口。一开始，程序在第 15 行声明了一个 bytes 包里的 Buffer 类型的变量，并使用零值初始化。在第 16 行创建了一个 byte 切片，并用字符串"Hello"初始化了这个切片。byte 切片随后被传入 Write 方法，成为 Buffer 类型变量里的初始内容。

第 20 行使用 fmt 包里的 Fprintf 函数将字符串"World!"追加到 Buffer 类型变量里。让我们看一下 Fprintf 函数的声明，如代码清单 8-39 所示。

代码清单 8-39 golang.org/src/fmt/print.go

```
// Fprintf 根据格式化说明符来格式写入内容，并输出到 w
// 这个函数返回写入的字节数，以及任何遇到的错误
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

需要注意 Fprintf 函数的第一个参数。这个参数需要接收一个实现了 io.Writer 接口类型的值。因为我们传入了之前创建的 Buffer 类型值的地址，这意味着 bytes 包里的 Buffer 类型必须实现了这个接口。那么在 bytes 包的源代码里，我们应该能找到为 Buffer 类型声明的 Write 方法，如代码清单 8-40 所示。

代码清单 8-40 golang.org/src/bytes/buffer.go

```
// Write 将 p 的内容追加到缓冲区，如果需要，会增大缓冲区的空间。返回值 n 是
// p 的长度，err 总是 nil。如果缓冲区变得太大，Write 会引起崩溃…
func (b *Buffer) Write(p []byte) (n int, err error) {
    b.lastRead = opInvalid
    m := b.grow(len(p))
    return copy(b.buf[m:], p), nil
}
```

代码清单 8-40 展示了 Buffer 类型的 Write 方法的当前版本的实现。由于实现了这个方法，指向 Buffer 类型的指针就满足了 io.Writer 接口，可以将指针作为第一个参数传入 Fprintf。在这个例子里，我们使用 Fprintf 函数，最终通过 Buffer 实现的 Write 方法，将"World!"字符串追加到 Buffer 类型变量的内部缓冲区。

让我们看一下代码清单 8-37 的最后几行，如代码清单 8-41 所示，将整个 Buffer 类型变量的内容写到 stdout。

代码清单 8-41 listing37.go：第 22 行到第 25 行

```
22 // 将 Buffer 的内容输出到标准输出设备
23     // 将 os.File 值的地址作为 io.Writer 类型值传入
24     b.WriteTo(os.Stdout)
25 }
```

在代码清单 8-37 的第 24 行，使用 WriteTo 方法将 Buffer 类型的变量的内容写到 stdout 设备。这个方法接受一个实现了 io.Writer 接口的值。在这个程序里，传入的值是 os 包的 Stdout 变量的值，如代码清单 8-42 所示。

代码清单 8-42 golang.org/src/os/file.go

```
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

这些变量自动声明为 NewFile 函数返回的类型，如代码清单 8-43 所示。

代码清单 8-43 golang.org/src/os/file_unix.go

```
// NewFile 返回一个具有给定的文件描述符和名字的新 File
func NewFile(fd uintptr, name string) *File {
    fdi := int(fd)
```

```
    if fdi < 0 {
        return nil
    }
    f := &File{&file{fd: fdi, name: name}}
    runtime.SetFinalizer(f.file, (*file).close)
    return f
}
```

就像在代码清单 8-43 里看到的那样，NewFile 函数返回一个指向 File 类型的指针。这就是 Stdout 变量的类型。既然我们可以将这个类型的指针作为参数传入 WriteTo 方法，那么这个类型一定实现了 io.Writer 接口。在 os 包的源代码里，我们应该能找到 Write 方法，如代码清单 8-44 所示。

代码清单 8-44 golang.org/src/os/file.go

```
// Write 将 len(b) 个字节写入 File
// 这个方法返回写入的字节数，如果有错误，也会返回错误
// 如果 n != len(b)，Write 会返回一个非 nil 的错误
func (f *File) Write(b []byte) (n int, err error) {
    if f == nil {
        return 0, ErrInvalid
    }
    n, e := f.write(b)
    if n < 0 {
        n = 0
    }
    if n != len(b) {
        err = io.ErrShortWrite
    }

    epipecheck(f, e)
    if e != nil {
        err = &PathError{"write", f.name, e}
    }
    return n, err
}
```

没错，代码清单 8-44 中的代码展示了 File 类型指针实现 io.Writer 接口类型的代码。让我们再看一下代码清单 8-37 的第 24 行，如代码清单 8-45 所示。

代码清单 8-45 listing37.go：第 22 行到第 25 行

```
22      // 将 Buffer 的内容输出到标准输出设备
23      // 将 os.File 值的地址作为 io.Writer 类型值传入
24      b.WriteTo(os.Stdout)
25  }
```

可以看到，WriteTo 方法可以将 Buffer 类型变量的内容写到 stdout，结果就是在终端窗口上显示了"Hello World!"字符串。这个方法会通过接口值，调用 File 类型实现的 Write 方法。

这个例子展示了接口的优雅以及它带给语言的强大的能力。得益于 bytes.Buffer 和 os.File 类型都实现了 Writer 接口，我们可以使用标准库里已有的功能，将这些类型组合在一起完成工作。接下来让我们看一个更加实用的例子。

8.4.3 简单的 curl

在 Linux 和 MacOS（曾用名 Mac OS X）系统里可以找到一个名为 curl 的命令行工具。这个工具可以对指定的 URL 发起 HTTP 请求，并保存返回的内容。通过使用 http、io 和 os 包，我们可以用很少的几行代码来实现一个自己的 curl 工具。

让我们来看一下实现了基础 curl 功能的例子，如代码清单 8-46 所示。

代码清单 8-46　listing46.go

```
// 这个示例程序展示如何使用 io.Reader 和 io.Writer 接口
// 写一个简单版本的 curl
package main

import (
    "io"
    "log"
    "net/http"
    "os"
)

// main 是应用程序的入口
func main() {
    // 这里的 r 是一个响应，r.Body 是 io.Reader
    r, err := http.Get(os.Args[1])
    if err != nil {
        log.Fatalln(err)
    }

    // 创建文件来保存响应内容
    file, err := os.Create(os.Args[2])
    if err != nil {
        log.Fatalln(err)
    }
    defer file.Close()

    // 使用 MultiWriter，这样就可以同时向文件和标准输出设备
    // 进行写操作
    dest := io.MultiWriter(os.Stdout, file)

    // 读出响应的内容，并写到两个目的地
    io.Copy(dest, r.Body)
    if err := r.Body.Close(); err != nil {
        log.Println(err)
    }
}
```

代码清单 8-46 展示了一个实现了基本骨架功能的 curl，它可以下载、展示并保存任意的 HTTP Get 请求的内容。这个例子会将响应的结果同时写入文件以及 stdout。为了让例子保持简单，这个程序没有检查命令行输入参数的有效性，也没有支持更高级的选项。

在这个程序的第 15 行，使用来自命令行的第一个参数来执行 HTTP Get 请求。如果这个参数是一个 URL，而且请求没有发生错误，变量 r 里就包含了该请求的响应结果。在第 21 行，我们使用命令行的第二个参数打开了一个文件。如果这个文件打开成功，那么在第 25 行会使用 defer 语句安排在函数退出时执行文件的关闭操作。

因为我们希望同时向 stdout 和指定的文件里写请求的内容，所以在第 29 行我们使用 io 包里的 MultiWriter 函数将文件和 stdout 整合为一个 io.Writer 值。在第 33 行，我们使用 io 包的 Copy 函数从响应的结果里读取内容，并写入两个目的地。由于有 MultiWriter 函数提供的值的支持，我们可使用一次 Copy 调用，将内容同时写到两个目的地。

利用 io 包里已经提供的支持，以及 http 和 os 包里已经实现了 io.Writer 和 io.Reader 接口类型的实现，我们不需要编写任何代码来完成这些底层的函数，借助已经存在的功能，将注意力集中在需要解决的问题上。如果我们自己的类型也实现了这些接口，就可以立刻支持已有的大量功能。

8.4.4 结论

可以在 io 包里找到大量的支持不同功能的函数，这些函数都能通过实现了 io.Writer 和 io.Reader 接口类型的值进行调用。其他包，如 http 包，也使用类似的模式，将接口声明为包的 API 的一部分，并提供对 io 包的支持。应该花时间看一下标准库中提供了些什么，以及它是如何实现的——不仅要防止重新造轮子，还要理解 Go 语言的设计者的习惯，并将这些习惯应用到自己的包和 API 的设计上。

8.5 小结

- 标准库有特殊的保证，并且被社区广泛应用。
- 使用标准库的包会让你的代码更易于管理，别人也会更信任你的代码。
- 100 余个包被合理组织，分布在 38 个类别里。
- 标准库里的 log 包拥有记录日志所需的一切功能。
- 标准库里的 xml 和 json 包让处理这两种数据格式变得很简单。
- io 包支持以流的方式高效处理数据。
- 接口允许你的代码组合已有的功能。
- 阅读标准库的代码是熟悉 Go 语言习惯的好方法。

第 9 章　测试和性能

本章主要内容

- 编写单元测试来验证代码的正确性
- 使用 httptest 来模拟基于 HTTP 的请求和响应
- 使用示例代码来给包写文档
- 通过基准测试来检查性能

作为一名合格的开发者，不应该在程序开发完之后才开始写测试代码。使用 Go 语言的测试框架，可以在开发的过程中就进行单元测试和基准测试。和 `go build` 命令类似，`go test` 命令可以用来执行写好的测试代码，需要做的就是遵守一些规则来写测试。而且，可以将测试无缝地集成到代码工程和持续集成系统里。

9.1　单元测试

*单元测试*是用来测试包或者程序的一部分代码或者一组代码的函数。测试的目的是确认目标代码在给定的场景下，有没有按照期望工作。一个场景是正向路经测试，就是在正常执行的情况下，保证代码不产生错误的测试。这种测试可以用来确认代码可以成功地向数据库中插入一条工作记录。

另外一些单元测试可能会测试负向路径的场景，保证代码不仅会产生错误，而且是预期的错误。这种场景下的测试可能是对数据库进行查询时没有找到任何结果，或者对数据库做了无效的更新。在这两种情况下，测试都要验证确实产生了错误，且产生的是预期的错误。总之，不管如何调用或者执行代码，所写的代码行为都是可预期的。

在 Go 语言里有几种方法写单元测试。*基础测试*（basic test）只使用一组参数和结果来测试一段代码。*表组测试*（table test）也会测试一段代码，但是会使用多组参数和结果进行测试。也可以使用一些方法来模仿（mock）测试代码需要使用到的外部资源，如数据库或者网络服务器。这有助于让测试在没有所需的外部资源可用的时候，模拟这些资源的行为使测试正常进行。最后，