

# 目 录 |

The Go Programming Language

出版者的话	
译者序	
前言	
<b>第 1 章 入门</b>	<b>1</b>
1.1 hello, world	1
1.2 命令行参数	3
1.3 找出重复行	6
1.4 GIF 动画	10
1.5 获取一个 URL	12
1.6 并发获取多个 URL	13
1.7 一个 Web 服务器	14
1.8 其他内容	17
<b>第 2 章 程序结构</b>	<b>20</b>
2.1 名称	20
2.2 声明	21
2.3 变量	22
2.3.1 短变量声明	22
2.3.2 指针	23
2.3.3 new 函数	25
2.3.4 变量的生命周期	26
2.4 赋值	27
2.4.1 多重赋值	27
2.4.2 可赋值性	28
2.5 类型声明	29
2.6 包和文件	30
2.6.1 导入	31
2.6.2 包初始化	33
2.7 作用域	34
<b>第 3 章 基本数据</b>	<b>38</b>
3.1 整数	38
3.2 浮点数	42
3.3 复数	45
3.4 布尔值	47
3.5 字符串	47
3.5.1 字符串字面量	49
3.5.2 Unicode	49
3.5.3 UTF-8	50
3.5.4 字符串和字节 slice	53
3.5.5 字符串和数字的相互转换	56
3.6 常量	56
3.6.1 常量生成器 iota	57
3.6.2 无类型常量	59
<b>第 4 章 复合数据类型</b>	<b>61</b>
4.1 数组	61
4.2 slice	63
4.2.1 append 函数	66
4.2.2 slice 就地修改	69
4.3 map	71
4.4 结构体	76
4.4.1 结构体字面量	78
4.4.2 结构体比较	80
4.4.3 结构体嵌套和匿名成员	80
4.5 JSON	82
4.6 文本和 HTML 模板	87
<b>第 5 章 函数</b>	<b>92</b>
5.1 函数声明	92
5.2 递归	93
5.3 多返回值	96
5.4 错误	98
5.4.1 错误处理策略	99
5.4.2 文件结束标识	101
5.5 函数变量	102
5.6 匿名函数	104
5.7 变长函数	110
5.8 延迟函数调用	111

5.9 容机	115	8.8 示例：并发目录遍历	192
5.10 恢复	118	8.9 取消	195
<b>第 6 章 方法</b>	<b>120</b>	8.10 示例：聊天服务器	198
6.1 方法声明	120	<b>第 9 章 使用共享变量实现并发</b>	<b>201</b>
6.2 指针接收者的方法	122	9.1 竞态	201
6.3 通过结构体内嵌组成类型	124	9.2 互斥锁： <code>sync.Mutex</code>	205
6.4 方法变量与表达式	127	9.3 读写互斥锁： <code>sync.RWMutex</code>	208
6.5 示例：位向量	128	9.4 内存同步	208
6.6 封装	130	9.5 延迟初始化： <code>sync.Once</code>	210
<b>第 7 章 接口</b>	<b>133</b>	9.6 竞态检测器	212
7.1 接口即约定	133	9.7 示例：并发非阻塞缓存	212
7.2 接口类型	135	9.8 goroutine 与线程	218
7.3 实现接口	136	9.8.1 可增长的栈	219
7.4 使用 <code>flag.Value</code> 来解析参数	139	9.8.2 goroutine 调度	219
7.5 接口值	141	9.8.3 GOMAXPROCS	219
7.6 使用 <code>sort.Interface</code> 来排序	144	9.8.4 goroutine 没有标识	220
7.7 <code>http.Handler</code> 接口	148	<b>第 10 章 包和 go 工具</b>	<b>221</b>
7.8 <code>error</code> 接口	152	10.1 引言	221
7.9 示例：表达式求值器	154	10.2 导入路径	221
7.10 类型断言	160	10.3 包的声明	222
7.11 使用类型断言来识别错误	161	10.4 导入声明	223
7.12 通过接口类型断言来查询特性	162	10.5 空导入	223
7.13 类型分支	164	10.6 包及其命名	225
7.14 示例：基于标记的 XML 解析	166	10.7 go 工具	226
7.15 一些建议	168	10.7.1 工作空间的组织	227
<b>第 8 章 goroutine 和通道</b>	<b>170</b>	10.7.2 包的下载	228
8.1 goroutine	170	10.7.3 包的构建	229
8.2 示例：并发时钟服务器	171	10.7.4 包的文档化	231
8.3 示例：并发回声服务器	174	10.7.5 内部包	232
8.4 通道	176	10.7.6 包的查询	233
8.4.1 无缓冲通道	177	<b>第 11 章 测试</b>	<b>235</b>
8.4.2 管道	178	11.1 <code>go test</code> 工具	235
8.4.3 单向通道类型	180	11.2 <code>Test</code> 函数	236
8.4.4 缓冲通道	181	11.2.1 随机测试	239
8.5 并行循环	183	11.2.2 测试命令	240
8.6 示例：并发的 Web 爬虫	187	11.2.3 白盒测试	242
8.7 使用 <code>select</code> 多路复用	190	11.2.4 外部测试包	245

11.2.5 编写有效测试	246
11.2.6 避免脆弱的测试	247
11.3 覆盖率	248
11.4 Benchmark 函数	250
11.5 性能剖析	252
11.6 Example 函数	254
<b>第 12 章 反射</b>	<b>256</b>
12.1 为什么使用反射	256
12.2 reflect.Type 和 reflect.Value	257
12.3 Display：一个递归的值显示器	259
12.4 示例：编码 S 表达式	263
12.5 使用 reflect.Value 来设置值	266
12.6 示例：解码 S 表达式	268
12.7 访问结构体字段标签	271
12.8 显示类型的方法	273
12.9 注意事项	274
<b>第 13 章 低级编程</b>	<b>276</b>
13.1 unsafe.Sizeof、Alignof 和 Offsetof	276
13.2 unsafe.Pointer	278
13.3 示例：深度相等	280
13.4 使用 cgo 调用 C 代码	282
13.5 关于安全的注意事项	286

## 入 门

本章是对于 Go 语言基本组件的一些说明。希望本章所提供的足够信息和示例，能够使您尽可能快地做一些有用的东西。本书所有的例子都是针对现实世界的任务的。本章将带您尝试体验用 Go 语言来编写各种程序：从简单的文件、图片处理到并发的客户端和服务器的互联网应用开发。虽然在一章里不能把所有东西讲清楚，但是以这类应用作为学习一门语言的开始是一种高效的方式。

学习新语言比较自然的方式，是使用新语言写一些你已经可以用其他语言实现的程序。我们试图说明和解释如何用好 Go 语言，当你写自己的代码的时候，本章的代码可以作为参考。

## 1.1 hello, world

我们依然从永恒的“hello, world”例子开始，它出现在 1978 年出版的《The C Programming Language》这本书的开头。C 对 Go 的影响非常直接，我们用“hello, world”来说明一些主要的思路：

```
gopl.io/ch1/helloworld
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Go 是编译型的语言。Go 的工具链将程序的源文件转变成机器相关的原生二进制指令。这些工具可以通过单一的 go 命令配合其子命令进行使用。最简单的子命令是 run，它将一个或多个以 .go 为后缀的源文件进行编译、链接，然后运行生成的可执行文件（本书中我们使用 \$ 符号作为命令提示符）：

```
$ go run helloworld.go
```

不出意料地，这将输出：

```
Hello, 世界
```

Go 原生地支持 Unicode，所以它可以处理所有国家的语言。

如果这个程序不是一次性的实验，那么编译输出成一个可复用的程序比较好。这通过 go build 来实现：

```
$ go build helloworld.go
```

这条命令生成了一个叫作 helloworld 的二进制程序，它可以不用进行任何其他处理，随时执行：

```
$ ./helloworld
Hello, 世界
```

我们给每一个重要的例子都加了一个标签，提示你可以从本书在 `gopl.io` 的源码库获取代码：

`gopl.io/ch1/helloworld`

如果执行 `go get gopl.io/ch1/helloworld`，它将会把源代码取到相应的目录。这将在 2.6 节和 10.7 节进行更多的讨论。

现在我们来说说该程序本身。Go 代码是使用包来组织的，包类似于其他语言中的库和模块。一个包由一个或多个 `.go` 源文件组成，放在一个文件夹中，该文件夹的名字描述了包的作用。每一个源文件的开始都用 `package` 声明，例子里面是 `package main`，指明了这个文件属于哪个包。后面跟着它导入的其他包的列表，然后是存储在文件中的程序声明。

Go 的标准库中有 100 多个包用来完成输入、输出、排序、文本处理等常规任务。例如，`fmt` 包中的函数用来格式化输出和扫描输入。`Println` 是 `fmt` 中一个基本的输出函数，它输出一个或多个用空格分隔的值，结尾使用一个换行符，这样看起来这些值是单行输出。

名为 `main` 的包比较特殊，它用来定义一个独立的可执行程序，而不是库。在 `main` 包中，函数 `main` 也是特殊的，不管在什么程序中，`main` 做什么事情，它总是程序开始执行的地方。当然，`main` 通常调用其他包中的函数来做更多的事情，比如 `fmt.Println`。

我们需要告诉编译器源文件需要哪些包，用 `package` 声明后面的 `import` 来导入这些包。“hello, world” 程序仅使用了一个来自于其他包的函数，而大多数程序可能导入更多的包。

你必须精确地导入需要的包。在缺失导入或存在不需要的包的情况下，编译会失败，这种严格的要求可以防止程序演化中引用不需要的包。

`import` 声明必须跟在 `package` 声明之后。`import` 导入声明后面，是组成程序的函数、变量、常量、类型（以 `func`、`var`、`const`、`type` 开头）声明。大部分情况下，声明的顺序是没有关系的。示例中的程序足够短，因为它只声明了一个函数，这个函数又仅仅调用了一个其他的函数。为了节省空间，在处理示例的时候，我们有时不展示 `package` 和 `import` 声明，但是它们存在于源文件中，并且编译时必不可少。

一个函数的声明由 `func` 关键字、函数名、参数列表（`main` 函数为空）、返回值列表（可以为空）、放在大括号内的函数体组成，函数体定义函数是用来做什么的，这将在第 5 章详细介绍。

Go 不需要在语句或声明后面使用分号结尾，除非有多个语句或声明出现在同一行。事实上，跟在特定符号后面的换行符被转换为分号，在什么地方进行换行会影响对 Go 代码的解析。例如，“`{`” 符号必须和关键字 `func` 在同一行，不能独自成行，并且在 `x+y` 这个表达式中，换行符可以在 `+` 操作符的后面，但是不能在 `+` 操作符的前面。

Go 对于代码的格式化要求非常严格。`gofmt` 工具将代码以标准格式重写，`go` 工具的 `fmt` 子命令使用 `gofmt` 工具来格式化指定包里的所有文件或者当前文件夹中的文件（默认情况下）。本书中包含的所有 Go 源代码文件都使用 `gofmt` 运行过，你应该养成对自己的代码使用 `gofmt` 工具的习惯。定制一个标准的格式，可以省去大量无关紧要的辩论，更重要的是，如果允许随心所欲的格式，各种自动化的源代码转换工具将不可用。

许多文本编辑器可以配置为每次在保存文件时自动运行 `gofmt`，因此源文件总可以保持正确的形式。此外，一个相关的工具 `goimports` 可以按需管理导入声明的插入和移除。它不是标准发布版的一部分，可以通过执行下面的命令获取到：

```
$ go get golang.org/x/tools/cmd/goimports
```

对大多数用户来说，按照常规方式下载、编译包，执行自带的测试，查看文档等操作，使用 `go` 工具都可以实现，这将在 10.7 节详细介绍。

## 1.2 命令行参数

大部分程序处理输入然后产生输出，这就是关于计算的大致定义。但是程序怎样获取数据的输入呢？一些程序自己生成数据，更多的时候，输入来自一个外部源：文件、网络连接、其他程序的输出、键盘、命令行参数等。随后的一些示例将从命令行参数开始讨论这些输入。

`os` 包提供一些函数和变量，以与平台无关的方式和操作系统打交道。命令行参数以 `os` 包中 `Args` 名字的变量供程序访问，在 `os` 包外面，使用 `os.Args` 这个名字。

变量 `os.Args` 是一个字符串 `slice`。`slice` 是 Go 中的基础概念，很快我们将讨论到它。现在只需理解它是一个动态容量的顺序数组 `s`，可以通过 `s[i]` 来访问单个元素，通过 `s[m:n]` 来访问一段连续子区间，数组长度用 `len(s)` 表示。与大部分编程语言一样，在 Go 中，所有的索引使用半开区间，即包含第一个索引，不包含最后一个索引，因为这样逻辑比较简单。例如，`slice s[m:n]`，其中， $0 \leq m \leq n \leq \text{len}(s)$ ，包含  $n-m$  个元素。

`os.Args` 的第一个元素是 `os.Args[0]`，它是命令本身的名字；另外的元素是程序开始执行时的参数。表达式 `s[m:n]` 表示一个从第 `m` 个到第 `n-1` 个元素的 `slice`，所以下一个示例中 `slice` 需要的元素是 `os.Args[1:len(os.Args)]`。如果 `m` 或 `n` 缺失，默认分别是 `0` 或 `len(s)`，所以我们可以将期望的 `slice` 简写为 `os.Args[1:]`。

这里有一个 UNIX `echo` 命令的实现，它将命令行参数输出到一行。该实现导入两个包，使用由圆括号括起来的列表，而不是独立的 `import` 声明。两者都是合法的，但为了方便起见，我们使用列表的方式。导入的顺序是没有关系的，`gofmt` 工具会将其按照字母顺序表进行排序（当一个示例有几个版本时，通常给它们编号以区分出当前讨论的版本）。

```
gopl.io/ch1/echo1
// echo1 输出其命令行参数
package main

import (
    "fmt"
    "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

注释以 `//` 开头。所有以 `//` 开头的文本是给程序员看的注释，编译器将会忽略它们。习惯上，在一个包声明前，使用注释对其进行描述；对于 `main` 包，注释是一个或多个完整的句子，用来对这个程序进行整体概括。

`var` 关键字声明了两个 `string` 类型的变量 `s` 和 `sep`。变量可以在声明的时候初始化。如果变量没有明确地初始化，它将隐式地初始化为这个类型的空值。例如，对于数字初始化结果是 `0`，对于字符串是空字符串 `""`。在这个示例中，`s` 和 `sep` 隐式初始化为空字符串。第 2 章将讨论变量和声明。

对于数字，Go 提供常规的算术和逻辑操作符。当应用于字符串时，`+` 操作符对字符串的值进行追加操作，所以表达式

```
sep + os.Args[i]
```

表示将 `sep` 和 `os.Args[i]` 追加到一起。程序中使用的语句

```
s += sep + os.Args[i]
```

是一个赋值语句，将 `sep` 和 `os.Args[i]` 追加到旧的 `s` 上面，并且重新赋给 `s`，它等价于下面的语句：

```
s = s + sep + os.Args[i]
```

操作符 `+=` 是一个赋值操作符。每一个算术和逻辑操作符（例如 `+` 或者 `*`）都有一个对应的赋值操作符。

`echo` 程序会循环每次输出，但是这个版本中我们通过反复追加来构建一个字符串。字符串 `s` 一开始为空字符串 `""`，每一次循环追加一些文本。在第一次迭代后，一个空格被插入，这样当循环结束时，每个参数之间都有一个空格。这是一个二次过程，如果参数数量很大成本会比较高，不过对于 `echo` 程序还好。本章和下一章会展示几个改进版本，它们会逐步处理掉低效的地方。

循环的索引变量 `i` 在 `for` 循环开始处声明。`:=` 符号用于短变量声明，这种语句声明一个或多个变量，并且根据初始化的值给予合适的类型，下一章会详细讨论它。

递增语句 `i++` 对 `i` 进行加 1，它等价于 `i += 1`，又等价于 `i = i + 1`。对应的递减语句 `i--` 对 `i` 进行减 1。这些是语句，而不像其他 C 族语言一样是表达式，所以 `j = i++` 是不合法的，并且仅支持后缀，所以 `--i` 不合法。

`for` 是 Go 里面的唯一循环语句。它有几种形式，这里展示其中一种：

```
for initialization; condition; post {
    // 零个或多个语句
}
```

`for` 循环的三个组成部分两边不用小括号。大括号是必需的，但左大括号必须和 `post`（后置）语句在同一行。

可选的 `initialization`（初始化）语句在循环开始之前执行。如果存在，它必须是一个简单的语句，比如一个简短的变量声明，一个递增或赋值语句，或者一个函数调用。`condition`（条件）是一个布尔表达式，在循环的每一次迭代开始前推演，如果推演结果是真，循环则继续执行。`post` 语句在循环体之后被执行，然后条件被再次推演。条件变成假之后循环结束。

三部分都是可以省略的。如果没有 `initialization` 和 `post` 语句，分号可以省略：

```
// 传统的 "while" 循环
for condition {
    // ...
}
```

如果条件部分都不存在，例子如下：

```
// 传统的无限循环
for {
    // ...
}
```

循环是无限的，尽管这种形式的循环可以通过如 `break` 或 `return` 等语句进行终止。

另一种形式的 `for` 循环在字符串或 `slice` 数据上迭代。为了说明，这里给出第 2 版的 `echo`：

```
gopl.io/ch1/echo2
// echo2 输出其命令行参数
package main

import (
    "fmt"
    "os"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

每一次迭代，`range` 产生一对值：索引和这个索引处元素的值。这个例子里，我们不需要索引，但是语法上 `range` 循环需要处理，因此也必须处理索引。一个主意是我们将索引赋予一个临时变量（如 `temp`）然后忽略它，但是 Go 不允许存在无用的临时变量，不然会出现编译错误。

解决方案是使用空标识符，它的名字是 `_`（即下划线）。空标识符可以用在任何语法需要变量名但是程序逻辑不需要的地方，例如丢弃每次迭代产生的无用的索引。大多数 Go 程序员喜欢搭配使用 `range` 和 `_` 来写上面的 `echo` 程序，因为索引在 `os.Args` 上面是隐式的，所以更不容易犯错。

这个版本的程序使用短的变量声明来声明和初始化 `s` 和 `sep`，但是我们可以等价地分开声明变量。以下几种声明字符串变量的方式是等价的：

```
s := ""
var s string
var s = ""
var s string = ""
```

为什么我们更喜欢某一个？第一种形式的短变量声明更加简洁，但是通常在一个函数内部使用，不适合包级别的变量。第二种形式依赖默认初始化为空字符串的 `" "`。第三种形式很少用，除非我们声明多个变量。第四种形式是显式的变量类型，在类型一致的情况下是冗余的信息，在类型不一致的情况下是必需的。实践中，我们应当使用前两种形式，使用显式的初始化来说明初始化变量的重要性，使用隐式的初始化来表明初始化变量不重要。

如上所述，每次循环，字符串 `s` 有了新的内容。`+=` 语句通过追加旧的字符串、空格字符和下一个参数，生成一个新的字符串，然后把新字符串赋给 `s`。旧的内容不再需要使用，会被例行垃圾回收。

如果有大量的数据需要处理，这样的代价会比较大。一个简单和高效的方式是使用

`strings` 包中的 `Join` 函数：

```
gopl.io/ch1/echo3
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

最后，如果我们不关心格式，只是想看值，或许只是调试，那么用 `Println` 格式化结果就可以了：

```
fmt.Println(os.Args[1:])
```

这个输出语句和我们从 `strings.Join` 得到的输出很像，不过两边有括号。任何 slice 都能够以这样的方式输出。

**练习 1.1：**修改 `echo` 程序输出 `os.Args[0]`，即命令的名字。

**练习 1.2：**修改 `echo` 程序，输出参数的索引和值，每行一个。

**练习 1.3：**尝试测量可能低效的程序和使用 `strings.Join` 的程序在执行时间上的差异。  
(1.6 节有 `time` 包，11.4 节展示如何撰写系统性的性能评估测试。)

## 1.3 找出重复行

用于文件复制、打印、检索、排序、统计的程序，通常有一个相似的结构：在输入接口上循环读取，然后对每一个元素进行一些计算，在运行时或者在最后输出结果。我们展示三个版本的 `dup` 程序，它受 UNIX 的 `uniq` 命令启发来找到相邻的重复行。这个程序使用容易适配的结构和包。

第一个版本的 `dup` 程序输出标准输入中出现次数大于 1 的行，前面是次数。这个程序引入 `if` 语句、`map` 类型和 `bufio` 包。

```
gopl.io/ch1/dup1
// dup1 输出标准输入中出现次数大于 1 的行，前面是次数
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // 注意：忽略 input.Err() 中可能的错误
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

像 `for` 一样，`if` 语句中的条件部分也从不放在圆括号里面，但是程序体中需要用到大括号。这里还可以有一个可选的 `else` 部分，当条件为 `false` 的时候执行。

map 存储一个键 / 值对集合，并且提供常量时间的操作来存储、获取或测试集合中的某个元素。键可以是其值能够进行相等（==）比较的任意类型，字符串是最常见的例子；值可以是任意类型。这个例子中，键的类型是字符串，值是 int。内置的函数 make 可以用来新建 map，它还可以有其他用途。map 将在 4.3 节中进行更多讨论。

每次 dup 从输入读取一行内容，这一行就作为 map 中的键，对应的值递增 1。语句 counts[input.Text()]++ 等价于下面的两个语句：

```
line := input.Text()  
counts[line] = counts[line] + 1
```

键在 map 中不存在时也是没有问题的。当一个新的行第一次出现时，右边的表达式 counts[line] 根据值类型被推演为零值，int 的零值是 0。

为了输出结果，我们使用基于 range 的 for 循环，这次在 map 类型的 counts 变量上遍历。像以前一样，每次迭代输出两个结果，map 里面一个元素对应的键和值。map 里面的键的迭代顺序不是固定的，通常是随机的，每次运行都不一致。这是有意设计的，以防止程序依赖某种特定的序列，此处不对排序做任何保证。

下面讨论 bufio 包，使用它可以简便和高效地处理输入和输出。其中一个最有用的特性是称为扫描器（Scanner）的类型，它可以读取输入，以行或者单词为单位断开，这是处理以行为单位的输入内容的最简单方式。

程序使用短变量的声明方式，新建一个 bufio.Scanner 类型 input 变量：

```
input := bufio.NewScanner(os.Stdin)
```

扫描器从程序的标准输入进行读取。每一次调用 input.Scan() 读取下一行，并且将结尾的换行符去掉；通过调用 input.Text() 来获取读到的内容。Scan 函数在读到新行的时候返回 true，在没有更多内容的时候返回 false。

像 C 语言或其他语言中的 printf 一样，函数 fmt.Printf 从一个表达式列表生成格式化的输出。它的第一个参数是格式化指示字符串，由它指定其他参数如何格式化。每一个参数的格式是一个转义字符、一个百分号加一个字符。例如：%d 将一个整数格式化为十进制的形式，%s 把参数展开为字符串变量的值。

Printf 函数有超过 10 个这样的转义字符，Go 程序员称为 verb。下表远不完整，但是它说明有很多可以用的功能：

verb	描述
%d	十进制整数
%x, %o, %b	十六进制、八进制、二进制整数
%f, %g, %e	浮点数：如 3.141593, 3.141592653589793, 3.141593e+00
%t	布尔型：true 或 false
%c	字符（Unicode 码点）
%s	字符串
%q	带引号字符串（如 "abc"）或者字符（如 'c'）
%v	内置格式的任何值
%T	任何值的类型
%%	百分号本身（无操作数）

程序 `dup1` 中的格式化字符串还包含一个制表符 `\t` 和一个换行符 `\n`。字符串字面量可以包含类似转义序列（escape sequence）来表示不可见字符。`Printf` 默认不写换行符。按照约定，诸如 `log.Printf` 和 `fmt.Errorf` 之类的格式化函数以 `f` 结尾，使用和 `fmt.Printf` 相同的格式化规则；而那些以 `ln` 结尾的函数（如 `Println`）则使用 `%v` 的方式来格式化参数，并在最后追加换行符。

许多程序既可以像 `dup` 一样从标准输入进行读取，也可以从具体的文件读取。下一个版本的 `dup` 程序可以从标准输入或一个文件列表进行读取，使用 `os.Open` 函数来逐个打开：

```
gopl.io/ch1/dup2

// dup2 打印输入中多次出现的行的个数和文本
// 它从 stdin 或指定的文件列表读取
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // 注意：忽略 input.Err() 中可能的错误
}
```

函数 `os.Open` 返回两个值。第一个是打开的文件 (`*os.File`)，该文件随后被 `Scanner` 读取。

第二个返回值是一个内置的 `error` 类型的值。如果 `err` 等于特殊的内置 `nil` 值，标准文件成功打开。文件在被读到结尾的时候，`Close` 函数关闭文件，然后释放相应的资源（内存等）。另一方面，如果 `err` 不是 `nil`，说明出错了。这时，`error` 的值描述错误原因。简单的错误处理是使用 `Fprintf` 和 `%v` 在标准错误流上输出一条消息，`%v` 可以使用默认格式显示任意类型的值；错误处理后，`dup` 开始处理下一个文件；`continue` 语句让循环进入下一个迭代。

为了保持示例代码简短，这里对错误处理有意进行了一定程度的忽略。很明显，必须检查 `os.Open` 返回的错误；但是，我们忽略了使用 `input.Scan` 读取文件的过程中出现概率很小的错误。我们将标记所跳过的错误检查，5.4 节将更详细地讨论错误处理。

值得注意的是，对 `countLines` 的调用出现在其声明之前。函数和其他包级别的实体可以以任意次序声明。

`map` 是一个使用 `make` 创建的数据结构的引用。当一个 `map` 被传递给一个函数时，函数接收到这个引用的副本，所以被调用函数中对于 `map` 数据结构中的改变对函数调用者使用的 `map` 引用也是可见的。在示例中，`countLines` 函数在 `counts` `map` 中插入的值，在 `main` 函数中也是可见的。

这个版本的 `dup` 使用“流式”模式读取输入，然后按需拆分为行，这样原理上这些程序可以处理海量的输入。一个可选的方式是一次读取整个输入到大块内存，一次性地分割所有行，然后处理这些行。接下去的版本 `dup3` 将以这种方式处理。这里引入一个 `ReadFile` 函数（从 `io/ioutil` 包），它读取整个命名文件的内容，还引入一个 `strings.Split` 函数，它将一个字符串分割为一个由子串组成的 `slice`。（`Split` 是前面介绍过的 `strings.Join` 的反操作。）

我们在某种程度上简化了 `dup3`：第一，它仅读取指定的文件，而非标准输入，因为 `ReadFile` 需要一个文件名作为参数；第二，我们将统计行数的工作放回 `main` 函数中，因为它当前仅在一处用到。

```
gopl.io/ch1/dup3
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

`ReadFile` 函数返回一个可以转化成字符串的字节 `slice`，这样它可以被 `strings.Split` 分割。3.5.4 节将详细讨论字符串和字节 `slice`。

实际上，`bufio.Scanner`、`ioutil.ReadFile` 以及 `ioutil.WriteFile` 使用 `*os.File` 中的 `Read` 和 `Write` 方法，但是大多数程序员很少需要直接访问底层的例程。像 `bufio` 和 `io/ioutil` 包中上层的方法更易使用。

练习 1.4：修改 dup2 程序，输出出现重复行的文件的名称。

## 1.4 GIF 动画

下一个程序展示 Go 标准的图像包的使用，用来创建一系列的位图图像，然后将位图序列编码为 GIF 动画。下面的图像叫作利萨茹图形，是 20 世纪 60 年代科幻片中的纤维状视觉效果。利萨茹图形是参数化的二维谐振曲线，如示波器  $x$  轴和  $y$  轴馈电输入的两个正弦波。图 1-1 是几个示例。

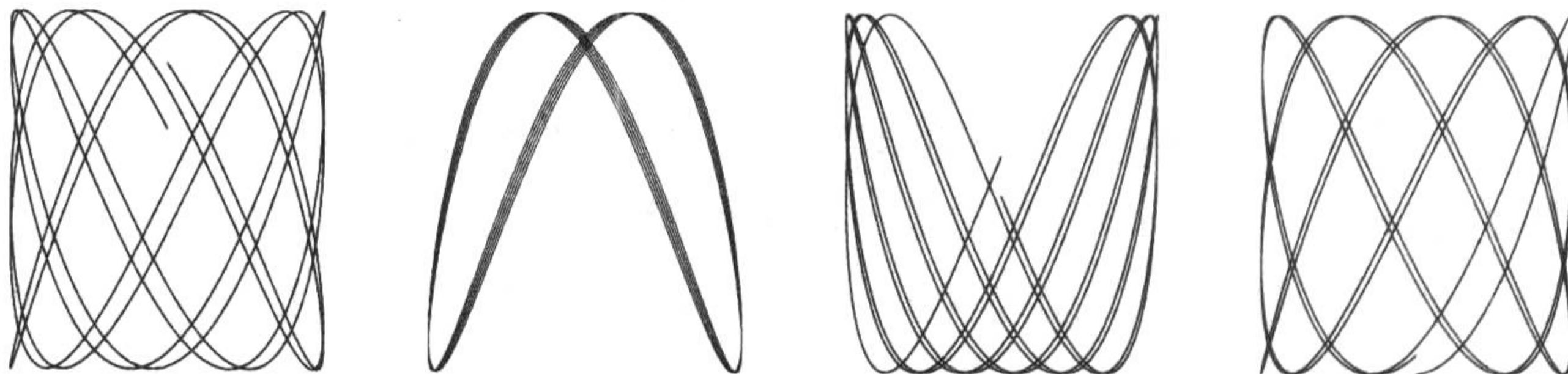


图 1-1 四种利萨茹图形

这段代码里有几个新的组成，包括 `const` 声明、结构体以及复合字面量。不像大多数例子，本例还引入了浮点运算。这个示例的主要目的是提供一些思路，表明 Go 语言看起来是怎样的，以及利用 Go 语言和它的库可以轻易完成哪些事情，这里只简短地讨论这几个主题，更多细节将放在后面章节。

```
gopl.io/ch1/lissajous
// lissajous 产生随机利萨茹图形的 GIF 动画
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)

var palette = []color.Color{color.White, color.Black}
const (
    whiteIndex = 0 // 画板中的第一种颜色
    blackIndex = 1 // 画板中的下一种颜色
)
func main() {
    rand.Seed(time.Now().UTC().UnixNano())
    if len(os.Args) > 1 && os.Args[1] == "web" {
        handler := func(w http.ResponseWriter, r *http.Request) {
            lissajous(w)
        }
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))
        return
    }
    lissajous(os.Stdout)
}
```

```
func lissajous(out io.Writer) {
    const (
        cycles = 5      // 完整的 x 振荡器变化的个数
        res   = 0.001   // 角度分辨率
        size   = 100    // 图像画布包含 [-size..+size]
        nframes = 64    // 动画中的帧数
        delay   = 8     // 以 10ms 为单位的帧间延迟
    )
    freq := rand.Float64() * 3.0 // y 振荡器的相对频率
    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0 // phase difference
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
        img := image.NewPaletted(rect, palette)
        for t := 0.0; t < cycles*2*math.Pi; t += res {
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim) // 注意：忽略编码错误
}
```

在导入那些由多段路径如 `image/color` 组成的包之后，使用路径最后的一段来引用这个包。所以变量 `color.White` 属于 `image/color` 包，`gif.GIF` 属于 `image/gif` 包。

`const` 声明（参考 3.6 节）用来给常量命名，常量是其值在编译期间固定的量，例如周期、帧数和延迟等数值参数。与 `var` 声明类似，`const` 声明可以出现在包级别（所以这些常量名字在包生命周期内都是可见的）或在一个函数内（所以名字仅在函数体内可见）。常量必须是数字、字符串或布尔值。

表达式 `[]color.Color{...}` 和 `gif.GIF{...}` 是复合字面量（参考 4.2 节、4.4.1 节），即用一系列元素的值初始化 Go 的复合类型的紧凑表达方式。这里，第一个是 `slice`，第二个是结构体。

`gif.GIF` 是一个结构体类型（参考 4.4 节）。结构体由一组称为字段的值组成，字段通常有不同的数据类型，它们一起组成单个对象，作为一个单位被对待。`anim` 变量是 `gif.GIF` 结构体类型。这个结构体字面量创建一个结构体 `LoopCount`，其值设置为 `nframes`；其他字段的值是对应类型的零值。结构体的每个字段可以通过点记法来访问，在最后两个赋值语句中，显式更新 `anim` 结构体的 `Delay` 和 `Image` 字段。

`lissajous` 函数有两个嵌套的循环。外层有 64 个迭代，每个迭代产生一个动画帧。它创建一个  $201 \times 201$  大小的画板，使用黑和白两种颜色。所有的像素值默认设置为 0（画板中的初始化颜色），这里设置为白色。每一个内层循环通过设置一些像素为黑色产生一个新的图像。结果使用内置的 `append` 参数将其追加到 `anim` 的帧列表中，并且指定 80ms 的延迟。最后帧和延迟的序列被编码成 GIF 格式，然后写入输出流 `out`。`out` 的类型是 `io.Writer`，它可以帮我们输出到很多地方，稍后即可看到。

外层循环运行两个振荡器。 $x$  方向的振荡器是正弦函数， $y$  方向也是正弦化的，但是它的频率相对于  $x$  的振动周期是  $0 \sim 3$  之间的一个随机数，它的相位相对于  $x$  的初始值为 0，然后随着每个动画帧增加。该循环在  $x$  振荡器完成 5 个完整周期后停止。每一步它都调用

`SetColorIndex` 将对应画板上面的  $(x, y)$  位置涂为黑色，在画板上的值为 1。

`main` 函数调用 `lissajous` 函数，直接写到标准输出，所以这个命令产生一个像图 1-1 那样的 GIF 动画：

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

**练习 1.5：** 改变利萨茹程序的画板颜色为绿色黑底来增加真实性。使用 `color.RGBA{0xRR, 0xGG, 0xBB, 0xff}` 创建一种 Web 颜色 `#RRGGBB`，每一对十六进制数字表示组成一个像素红、绿、蓝分量的亮度。

**练习 1.6：** 通过在画板中添加更多颜色，然后通过有趣的方式改变 `SetColorIndex` 的第三个参数，修改利萨茹程序来产生多种色彩的图片。

## 1.5 获取一个 URL

对许多应用而言，访问互联网上的信息和访问本地文件系统一样重要。Go 提供了一系列包，在 `net` 包下面分组管理，使用它们可以方便地通过互联网发送和接收信息，使用底层的网络连接，创建服务器，此时 Go 的并发特性（见第 8 章）特别有用。

程序 `fetch` 展示从互联网获取信息的最小需求，它获取每个指定 URL 的内容，然后不加解析地输出。`fetch` 来自 `curl` 这个非常重要的工具。显然可以使用这些数据做更多的事情，但这里只讲基本的思路，本书将会频繁用到这个程序：

```
gopl.io/ch1/fetch
// fetch 输出从 URL 获取的内容
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

这个程序使用的函数来自两个包：`net/http` 和 `io/ioutil`。`http.Get` 函数产生一个 HTTP 请求，如果没有出错，返回结果存在响应结构 `resp` 里面。其中 `resp` 的 `Body` 域包含服务器端响应的一个可读取数据流。随后 `ioutil.ReadAll` 读取整个响应结果并存入 `b`。关闭 `Body` 数据

流来避免资源泄漏，使用 `Printf` 将响应输出到标准输出。

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...
```

如果 HTTP 请求失败，`fetch` 报告失败：

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

无论哪种错误情况，`os.Exit(1)` 会在进程退出时返回状态码 1。

**练习 1.7：** 函数 `io.Copy(dst,src)` 从 `src` 读，并且写入 `dst`。使用它代替 `ioutil.ReadAll` 来复制响应内容到 `os.Stdout`，这样不需要装下整个响应数据流的缓冲区。确保检查 `io.Copy` 返回的错误结果。

**练习 1.8：** 修改 `fetch` 程序添加一个 `http://` 前缀（假如该 URL 参数缺失协议前缀）。可能会用到 `strings.HasPrefix`。

**练习 1.9：** 修改 `fetch` 来输出 HTTP 的状态吗，可以在 `resp.Status` 中找到它。

## 1.6 并发获取多个 URL

Go 最令人感兴趣和新颖的特点是支持并发编程。这是一个大话题，第 8 章和第 9 章将专门讨论，所以此处只是简单了解一下 Go 主要的并发机制、goroutine 和通道（channel）。

下一个程序 `fetchall` 和前一个一样获取 URL 的内容，但是它并发获取很多 URL 内容，于是这个进程使用的时间不超过耗时最长时间的获取任务，而不是所有获取任务总的时间。这个版本的 `fetchall` 丢弃响应的内容，但是报告每一个响应的大小和花费的时间：

```
gopl.io/ch1/fetchall
// fetchall 并发获取 URL 并报告它们的时间和大小
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)
func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
        go fetch(url, ch) // 启动一个 goroutine
    }
    for range os.Args[1:] {
        fmt.Println(<-ch) // 从通道 ch 接收
    }
    fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}
```

```

func fetch(url string, ch chan<- string) {
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt.Sprint(err) // 发送到通道 ch
        return
    }

    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // 不要泄露资源
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs %7d %s", secs, nbytes, url)
}

```

这有一个例子：

```

$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s      6852 https://godoc.org
0.16s      7261 https://golang.org
0.48s      2475 http://gopl.io
0.48s elapsed

```

`goroutine` 是一个并发执行的函数。通道是一种允许某一例程向另一个例程传递指定类型的值的通信机制。`main` 函数在一个 `goroutine` 中执行，然后 `go` 语句创建额外的 `goroutine`。

`main` 函数使用 `make` 创建一个字符串通道。对于每个命令行参数，`go` 语句在第一轮循环中启动一个新的 `goroutine`，它异步调用 `fetch` 来使用 `http.Get` 获取 URL 内容。`io.Copy` 函数读取响应的内容，然后通过写入 `ioutil.Discard` 输出流进行丢弃。`Copy` 返回字节数以及出现的任何错误。每一个结果返回时，`fetch` 发送一行汇总信息到通道 `ch`。`main` 中的第二轮循环接收并且输出那些汇总行。

当一个 `goroutine` 试图在一个通道上进行发送或接收操作时，它会阻塞，直到另一个 `goroutine` 试图进行接收或发送操作才传递值，并开始处理两个 `goroutine`。本例中，每一个 `fetch` 在通道 `ch` 上发送一个值 (`ch <- expression`)，`main` 函数接收它们 (`<-ch`)。由 `main` 来处理所有的输出确保了每个 `goroutine` 作为一个整体单元处理，这样就避免了两个 `goroutine` 同时完成造成输出交织所带来的风险。

**练习 1.10：**找一个产生大量数据的网站。连续两次运行 `fetchall`，看报告的时间是否会有大的变化，调查缓存情况。每一次获取的内容一样吗？修改 `fetchall` 将内容输出到文件，这样可以检查它是否一致。

**练习 1.11：**使用更长的参数列表来尝试 `fetchall`，例如使用 `alexa.com` 排名前 100 万的网站。如果一个网站没有响应，程序的行为是怎样的？（8.9 节会通过复制这个例子来描述响应的机制。）

## 1.7 一个 Web 服务器

使用 Go 的库非常容易实现一个 Web 服务器，用来响应像 `fetch` 那样的客户端请求。本节将展示一个迷你服务器，返回访问服务器的 URL 的路径部分。例如，如果请求的 URL 是 `http://localhost:8000/hello`，响应将是 `URL.Path = "/hello"`。

```
gopl.io/ch1/server1
// server1 是一个迷你回声服务器
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // 回声请求调用处理程序
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// 处理程序回显请求 URL r 的路径部分
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

这个程序只有寥寥几行代码，因为库函数做了大部分工作。`main` 函数将一个处理函数和以 / 开头的 URL 链接在一起，代表所有的 URL 使用这个函数处理，然后启动服务器监听进入 8000 端口处的请求。一个请求由一个 `http.Request` 类型的结构体表示，它包含很多关联的域，其中一个是所请求的 URL。当一个请求到达时，它被转交给处理函数，并从请求的 URL 中提取路径部分 (`/hello`)，使用 `fmt.Printf` 格式化，然后作为响应发送回去。Web 服务器将在 7.7 节进行详细讨论。

让我们在后台启动服务器。在 Mac OS X 或者 Linux 上，在命令行后添加一个 & 符号；在微软 Windows 上，不需要 & 符号，而需要单独开启一个独立的命令行窗口。

```
$ go run src/gopl.io/ch1/server1/main.go &
```

可以从命令行发起客户请求：

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

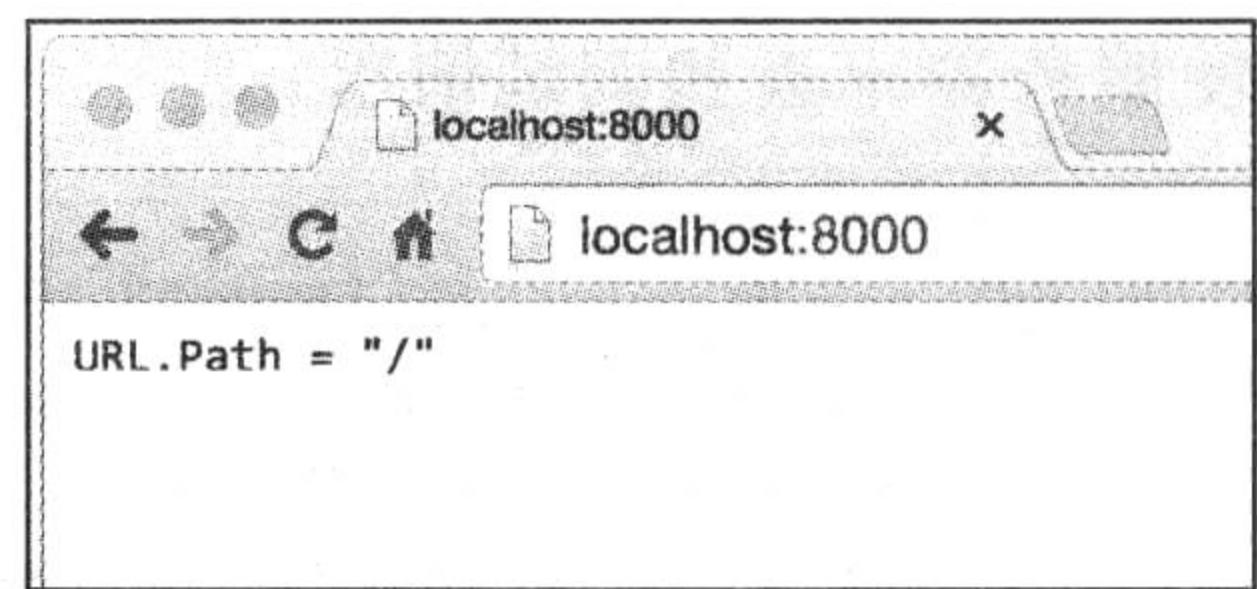


图 1-2 来自回声服务器的响应

另外，还可以通过浏览器进行访问，如图 1-2 所示。

为服务器添加功能很容易。一个有用的扩展是一个特定的 URL，它返回某种排序的状态。例如，这个版本的程序完成和回声服务器一样的事情，但同时返回请求的数量；URL `/count` 请求返回到现在为止的个数，去掉 `/count` 请求本身：

```
gopl.io/ch1/server2
// server2 是一个迷你的回声和计数器服务器
package main

import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int
```

```

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// 处理程序回显请求的 URL 的路径部分
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}

// counter 回显目前为止调用的次数
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}

```

这个服务器有两个处理函数，通过请求的 URL 来决定哪一个被调用：请求 /count 调用 counter，其他的调用 handler。以 / 结尾的处理模式匹配所有含有这个前缀的 URL。在后台，对于每个传入的请求，服务器在不同的 goroutine 中运行该处理函数，这样它可以同时处理多个请求。然而，如果两个并发的请求试图同时更新计数值 count，它可能会不一致地增加，程序会产生一个严重的竞态 bug（参考 9.1 节）。为避免该问题，必须确保最多只有一个 goroutine 在同一时间访问变量，这正是 mu.Lock() 和 mu.Unlock() 语句的作用。第 9 章将更细致地讨论共享变量的并发访问。

作为一个更完整的例子，处理函数可以报告它接收到的消息头和表单数据，这样可以方便服务器审查和调试请求：

[gopl.io/ch1/server3](http://gopl.io/ch1/server3)

```

// 处理程序回显 HTTP 请求
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}

```

这里使用 `http.Request` 结构体的成员来产生类似下面的输出：

```

GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"]
Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."]
Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"

```

```
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```

注意这里是如何在 if 语句中嵌套调用 `ParseForm` 的。Go 允许一个简单的语句（如一个局部变量声明）跟在 if 条件的前面，这在错误处理的时候特别有用。也可以这样写：

```
err := r.ParseForm()
if err != nil {
    log.Println(err)
}
```

但是合并的语句更短而且可以缩小 `err` 变量的作用域，这是一个好的实践。2.7 节将介绍作用域。

这些程序中，我们看到了作为输出流的三种非常不同的类型。`fetch` 程序复制 HTTP 响应到文件 `os.Stdout`，像 `lissajous` 一样；`fetchall` 程序通过将响应复制到 `ioutil.Discard` 中进行丢弃（在统计其长度时）；Web 服务器使用 `fmt.Fprintf` 通过写入 `http.ResponseWriter` 来让浏览器显示。

尽管三种类型细节不同，但都满足一个通用的接口（interface），该接口允许它们按需使用任何一种输出流。该接口（称为 `io.Writer`）将在 7.1 节进行讨论。

Go 的接口机制是第 7 章的内容，但是为了说明它可以做什么，我们来看一下整合 Web 服务器和 `lissajous` 函数是一件多么容易的事情，这样 GIF 动画将不再输出到标准输出而是 HTTP 客户端。简单添加这些行到 Web 服务器：

```
handler := func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/", handler)
```

或者也可以：

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
})
```

上面 `HandleFunc` 函数中立即调用的第二个参数是函数字面量，这是一个在该场景中使用它时才定义的匿名函数，这将在 5.6 节进一步解释。

一旦你完成这个改变，就可以通过浏览器访问 `http://localhost:8000`。每次加载页面，你将看到一个类似图 1-3 的动画。

**练习 1.12：**修改利萨茹服务器以通过 URL 参数读取参数值。例如，你可以通过调整它，使得像 `http://localhost:8000/?cycles=20` 这样的网址将其周期设置为 20，以替代默认的 5。使用 `strconv.Atoi` 函数来将字符串参数转化为整型。可以通过 `go doc strconv.Atoi` 来查看文档。

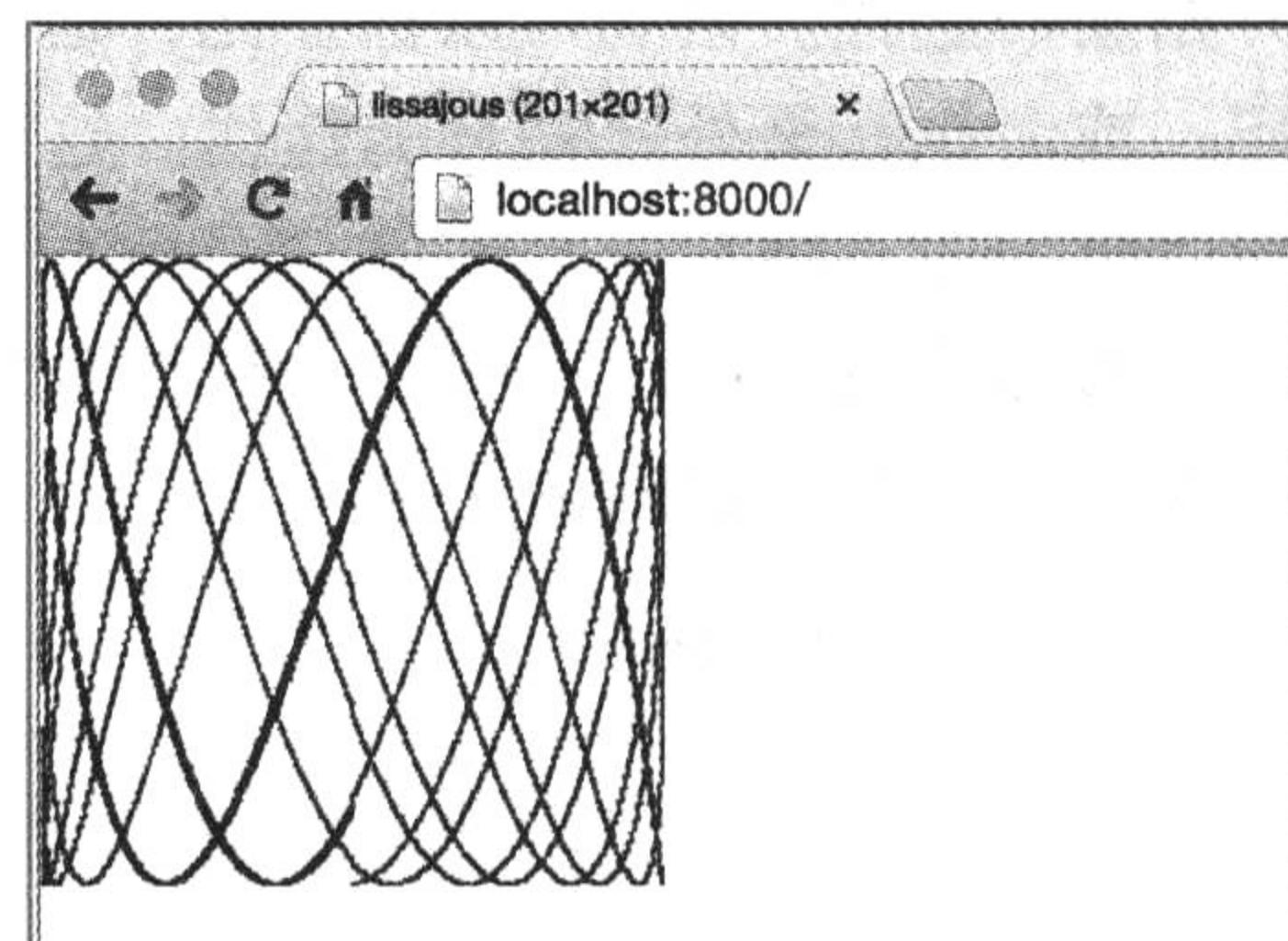


图 1-3 浏览器中的动态利萨茹图形

## 1.8 其他内容

Go 里面的东西远比这个快速入门中介绍的多。这里是一些很少提及或者完全忽略掉的

主题，下面简单地介绍一下这些主题，以便读者在用到时能够熟悉这些内容。

**控制流：**我们前面介绍了两个基础的控制语句 `if` 和 `for`，但没有介绍 `switch` 语句，它是多路分支控制。这里有一个例子：

```
switch coinflip() {
    case "heads":
        heads++
    case "tails":
        tails++
    default:
        fmt.Println("landed on edge!")
}
```

`coinflip` 的调用结果会和每一个条件的值进行比较。`case` 语句从上到下进行推演，所以第一个匹配的 `case` 语句会被执行。如果没有其他的 `case` 语句符合条件，那么可选的默认 `case` 语句将被执行。默认 `case` 语句可以放在任何地方。`case` 语句不像 C 语言那样从上到下贯穿执行（尽管有一个很少使用的 `fallthrough` 语句可以改写这个行为）。

`switch` 语句不需要操作数，它就像一个 `case` 语句列表，每条 `case` 语句都是一个布尔表达式：

```
func Signum(x int) int {
    switch {
    case x > 0:
        return +1
    default:
        return 0
    case x < 0:
        return -1
    }
}
```

这种形式称为无标签 (tagless) 选择，它等价于 `switch true`。

与 `for` 和 `if` 语句类似，`switch` 可以包含一个可选的简单语句：一个短变量声明，一个递增或赋值语句，或者一个函数调用，用来在判断条件前设置一个值。

`break` 和 `continue` 语句可以改变控制流。`break` 可以打断 `for`、`switch` 或 `select` 的最内层调用，开始执行下面的语句。正如我们在 1.3 节中看到的，`continue` 可以让 `for` 的内层循环开始新的迭代。语句可以标签化，这样方便 `break` 和 `continue` 引用它们来跳出多层嵌套的循环，或者执行最外层循环的迭代。这里还有一个 `goto` 语句，通常在机器生成的代码中使用，程序员一般不用它。

**命名类型：**`type` 声明给已有类型命名。因为结构体类型通常很长，所以它们基本上都是独立命名。一个熟悉的例子是定义一个 2D 图形系统的 `Point` 类型：

```
type Point struct {
    X, Y int
}
var p Point
```

类型声明和命名将在第 2 章讲述。

**指针：**Go 提供了指针，它的值是变量的地址。在一些语言（比如 C）中，指针基本是没有约束的。其他语言中，指针称为“引用”，并且除了到处传递之外，它不能做其他的事情。Go 做了一个折中，指针显式可见。使用 `&` 操作符可以获取一个变量的地址，使用 `*` 操作符

可以获取指针引用的变量的值，但是指针不支持算术运算。这将在 2.3.2 节进行介绍。

**方法和接口：**一个关联了命名类型的函数称为方法。Go 里面的方法可以关联到几乎所有的命名类型。方法在第 6 章讲述。接口可以用相同的方式处理不同的具体类型的抽象类型，它基于这些类型所包含的方法，而不是类型的描述或实现。接口是第 7 章的主题。

**包：**Go 自带一个可扩展并且实用的标准库，Go 社区创建和共享了更多的库。编程时，更多使用现有的包，而不是自己写所有的源码。本书将指出一些比较重要的标准库包，但是这些包太多了，本书无法一一展示，并且也无法提供诸如包的完整参考手册之类的东西。

在着手新程序前，看看是否已经有现成的包。可以在 <https://golang.org/pkg> 找到标准库包的索引，社区贡献的包可以在 <https://godoc.org> 找到。使用 `go doc` 工具可以方便地通过命令行访问这些文档：

```
$ go doc http.ListenAndServe
package http // import "net/http"

func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
    calls Serve with handler to handle requests on incoming connections.
...
```

**注释：**我们已经在程序或包的开始提到文档注释。在声明任何函数前，写一段注释来说明它的行为是一个好的风格。这个约定很重要，因为它们可以被 `go doc` 和 `godoc` 工具定位和作为文档显示（参考 10.7.4 节）。

对于跨越多行的注释，可以使用类似其他语言中的 `/*...*/` 注释。这样可以避免在文件的开始有一大块说明文本时每一行都有 `//`。在注释内部，`//` 和 `/*` 没有特殊的含义，所以注释不能嵌套。

# 程序结构

与其他编程语言一样，Go 语言中的大程序都从小的基本组件构建而来：变量存储值；简单表达式通过加和减等操作合并成大的；基本类型通过数组和结构体进行聚合；表达式通过 `if` 和 `for` 等控制语句来决定执行顺序；语句被组织成函数用于隔离和复用；函数被组织成源文件和包。

上面这些内容中的大部分已在前一章介绍过，本章将更细致地讨论 Go 程序中的基本结构元素。示例程序有意进行了简化，这有助于聚焦于语言本身而不是复杂的算法和数据结构。

## 2.1 名称

Go 中函数、变量、常量、类型、语句标签和包的名称遵循一个简单的规则：名称的开头是一个字母（Unicode 中的字符即可）或下划线，后面可以跟任意数量的字符、数字和下划线，并区分大小写。如 `heapSort` 和 `Heapsort` 是不同的名称。

Go 有 25 个像 `if` 和 `switch` 这样的关键字，只能用在语法允许的地方，它们不能作为名称：

<code>break</code>	<code>default</code>	<code>func</code>	<code>interface</code>	<code>select</code>
<code>case</code>	<code>defer</code>	<code>go</code>	<code>map</code>	<code>struct</code>
<code>chan</code>	<code>else</code>	<code>goto</code>	<code>package</code>	<code>switch</code>
<code>const</code>	<code>fallthrough</code>	<code>if</code>	<code>range</code>	<code>type</code>
<code>continue</code>	<code>for</code>	<code>import</code>	<code>return</code>	<code>var</code>

另外，还有三十几个内置的预声明的常量、类型和函数：

常量: `true` `false` `iota` `nil`  
类型: `int` `int8` `int16` `int32` `int64`  
`uint` `uint8` `uint16` `uint32` `uint64` `uintptr`  
`float32` `float64` `complex128` `complex64`  
`bool` `byte` `rune` `string` `error`  
函数: `make` `len` `cap` `new` `append` `copy` `close` `delete`  
`complex` `real` `imag`  
`panic` `recover`

这些名称不是预留的，可以在声明中使用它们。我们将在很多地方看到对其中的名称进行重声明，但是要知道这有冲突的风险。

如果一个实体在函数中声明，它只在函数局部有效。如果声明在函数外，它将对包里面的所有源文件可见。实体第一个字母的大小写决定其可见性是否跨包。如果名称以大写字母的开头，它是导出的，意味着它对包外是可见和可访问的，可以被自己包之外的其他程序所引用，像 `fmt` 包中的 `Printf`。包名本身总是由小写字母组成。

名称本身没有长度限制，但是习惯以及 Go 的编程风格倾向于使用短名称，特别是作用域较小的局部变量，你更喜欢看到一个变量叫 `i` 而不是 `theLoopIndex`。通常，名称的作用域

越大，就使用越长且更有意义的名称。

风格上，当遇到由单词组合的名称时，Go 程序员使用“驼峰式”的风格——更喜欢使用大写字母而不是下划线。所以标准库中的函数名采用 `QuoteRuneToASCII` 和 `parseRequestLine` 的形式，而不会采用 `quote_rune_to_ASCII` 或 `quote_rune_to_ASCII` 这样的形式。像 ASCII 和 HTML 这样的首字母缩写词通常使用相同的大小写，所以一个函数可以叫作 `htmlEscape`、`HTMLEscape` 或 `escapeHTML`，但不会是 `escapeHtml`。

## 2.2 声明

声明给一个程序实体命名，并且设定其部分或全部属性。有 4 个主要的声明：变量 (`var`)、常量 (`const`)、类型 (`type`) 和函数 (`func`)。本章讨论变量和类型，常量放在第 3 章讨论，函数放在第 5 章讨论。

Go 程序存储在一个或多个以 `.go` 为后缀的文件里。每一个文件以 `package` 声明开头，表明文件属于哪个包。`package` 声明后面是 `import` 声明，然后是包级别的类型、变量、常量、函数的声明，不区分顺序。例如，下面的程序声明一个常量、一个函数和一对变量：

```
gopl.io/ch2/boiling
// boiling 输出水的沸点
package main

import "fmt"

const boilingF = 212.0

func main() {
    var f = boilingF
    var c = (f - 32) * 5 / 9
    fmt.Printf("boiling point = %g°F or %g°C\n", f, c)
    // 输出：
    // boiling point = 212°F or 100°C
}
```

常量 `boilingF` 是一个包级别的声明 (`main` 包)，`f` 和 `c` 是属于 `main` 函数的局部变量。包级别的实体名字不仅对于包含其声明的源文件可见，而且对于同一个包里面的所有源文件都可见。另一方面，局部声明仅仅是在声明所在的函数内部可见，并且可能对于函数中的一小块区域可见。

函数的声明包含一个名字、一个参数列表（由函数的调用者提供的变量）、一个可选的返回值列表，以及函数体（其中包含具体逻辑语句）。如果函数不返回任何内容，返回值列表可以省略。函数的执行从第一个语句开始，直到遇到一个返回语句，或者执行到无返回结果的函数的结尾。然后程序控制和返回值（如果有的话）都返回给调用者。

我们已经看过许多函数，将来还会遇见更多，在第 5 章有更广泛的讨论，因此这里仅仅是一个概括。下面的函数 `ftoc` 封装了温度转换的逻辑，这样它可以只定义一次而在多个地方使用。这里 `main` 调用了它两次，使用两个不同的局部常量的值：

```
gopl.io/ch2/ftoc
// ftoc 输出两个华氏温度 - 摄氏温度的转换
package main

import "fmt"
```

```

func main() {
    const freezingF, boilingF = 32.0, 212.0
    fmt.Printf("%g°F = %g°C\n", freezingF, fToC(freezingF)) // "32°F = 0°C"
    fmt.Printf("%g°F = %g°C\n", boilingF, fToC(boilingF))   // "212°F = 100°C"
}

func fToC(f float64) float64 {
    return (f - 32) * 5 / 9
}

```

## 2.3 变量

`var` 声明创建一个具体类型的变量，然后给它附加一个名字，设置它的初始值。每一个声明有一个通用的形式：

```
var name type = expression
```

类型和表达式部分可以省略一个，但是不能都省略。如果类型省略，它的类型将由初始化表达式决定；如果表达式省略，其初始值对应于类型的零值——对于数字是 `0`，对于布尔值是 `false`，对于字符串是 `""`，对于接口和引用类型（slice、指针、map、通道、函数）是 `nil`。对于一个像数组或结构体这样的复合类型，零值是其所有元素或成员的零值。

零值机制保障所有的变量是良好定义的，Go 里面不存在未初始化变量。这种机制简化了代码，并且不需要额外工作就能感知边界条件的行为。例如：

```
var s string
fmt.Println(s) // ""
```

输出空字符串，而不是一些错误或不可预料的行为。Go 程序员经常花费精力来使复杂类型的零值有意义，以便变量一开始就处于一个可用状态。

可以声明一个变量列表，并选择使用对应的表达式列表对其初始化。忽略类型允许声明多个不同类型的变量。

```
var i, j, k int          // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

初始值设定可以是字面量值或者任意的表达式。包级别的初始化在 `main` 开始之前进行（参考 2.6.2 节），局部变量初始化和声明一样在函数执行期间进行。

变量可以通过调用返回多个值的函数进行初始化：

```
var f, err = os.Open(name) // os.Open 返回一个文件和一个错误
```

### 2.3.1 短变量声明

在函数中，一种称作短变量声明的可选形式可以用来声明和初始化局部变量。它使用 `name := expression` 的形式，`name` 的类型由 `expression` 的类型决定。这里是 `lissajous` 函数（参考 1.4 节）中的三个短变量声明：

```
anim := gif.GIF{LoopCount: nframes}
freq := rand.Float64() * 3.0
t := 0.0
```

因其短小、灵活，故而在局部变量的声明和初始化中主要使用短声明。`var` 声明通常是为了那些跟初始化表达式类型不一致的局部变量保留的，或者用于后面才对变量赋值以及变量初始值不重要的情况。

```
i := 100           // 一个 int 类型的变量
var boiling float64 = 100 // 一个 float64 类型的变量

var names []string
var err error
var p Point
```

与 var 声明一样，多个变量可以以短变量声明的方式声明和初始化：

```
i, j := 0, 1
```

只有当它们对于可读性有帮助的时候才使用多个初始化表达式来进行变量声明，例如短小且天然一组的 for 循环的初始化。

记住，:= 表示声明，而= 表示赋值。一个多变量的声明不能和多重赋值（参考 2.4.1 节）搞混，后者将右边的值赋给左边的对应变量：

```
i, j = j, i // 交换 i 和 j 的值
```

与普通的 var 声明类似，短变量声明也可以用来调用像 os.Open 那样返回两个或多个值的函数：

```
f, err := os.Open(name)
if err != nil {
    return err
}
// ... 使用 f...
f.Close()
```

一个容易被忽略但重要的地方是：短变量声明不需要声明所有在左边的变量。如果一些变量在同一个词法块中声明（参考 2.7 节），那么对于那些变量，短声明的行为等同于赋值。

在如下代码中，第一条语句声明了 in 和 err。第二条语句仅声明了 out，但向已有的 err 变量赋了值。

```
in, err := os.Open(infile)
// ...
out, err := os.Create(outfile)
```

短变量声明最少声明一个新变量，否则，代码编译将无法通过：

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // 编译错误：没有新的变量
```

第二个语句使用普通的赋值语句来修复这个错误。

只有在同一个词法块中已经存在变量的情况下，短声明的行为才和赋值操作一样，外层的声明将被忽略。我们在本章结尾的例子中将看到。

### 2.3.2 指针

变量是存储值的地方。借助声明创建的变量使用名字来区分，例如 x，但是许多变量仅仅使用像 x[i] 或者 x.f 这样的表达式来区分。所有这些表达式读取一个变量的值，除非它们出现在赋值操作符的左边，这个时候是给变量赋值。

指针的值是一个变量的地址。一个指针指示值所保存的位置。不是所有的值都有地址，但是所有的变量都有。使用指针，可以在无须知道变量名字的情况下，间接读取或更新变量的值。

如果一个变量声明为 `var x int`, 表达式 `&x` (`x` 的地址) 获取一个指向整型变量的指针, 它的类型是整型指针 (`*int`)。如果值叫作 `p`, 我们说 `p` 指向 `x`, 或者 `p` 包含 `x` 的地址。`p` 指向的变量写成 `*p`。表达式 `*p` 获取变量的值, 一个整型, 因为 `*p` 代表一个变量, 所以它也可以出现在赋值操作符左边, 用于更新变量的值。

```
x := 1
p := &x          // p 是整型指针, 指向 x
fmt.Println(*p) // "1"
*p = 2          // 等于 x = 2
fmt.Println(x) // 结果 "2"
```

每一个聚合类型变量的组成 (结构体的成员或数组中的元素) 都是变量, 所以也有一个地址。

变量有时候使用一个地址化的值。代表变量的表达式, 是唯一可以应用取地址操作符 `&` 的表达式。

指针类型的零值是 `nil`。测试 `p != nil`, 结果是 `true` 说明 `p` 指向一个变量。指针是可比较的, 两个指针当且仅当指向同一个变量或者两者都是 `nil` 的情况下才相等。

```
var x, y int
fmt.Println(&x == &x, &x == &y, &x == nil) // "true false false"
```

函数返回局部变量的地址是非常安全的。例如下面的代码中, 通过调用 `f` 产生的局部变量 `v` 即使在调用返回后依然存在, 指针 `p` 依然引用它:

```
var p = f()

func f() *int {
    v := 1
    return &v
}
```

每次调用 `f` 都会返回一个不同的值:

```
fmt.Println(f() == f()) // "false"
```

因为一个指针包含变量的地址, 所以传递一个指针参数给函数, 能够让函数更新间接传递的变量值。例如, 这个函数递增一个指针参数所指向的变量, 然后返回此变量的新值, 于是它可以在表达式中使用:

```
func incr(p *int) int {
    *p++ // 递增 p 所指向的值; p 自身保持不变
    return *p
}

v := 1
incr(&v)          // 副作用: v 现在等于 2
fmt.Println(incr(&v)) // "3" (v 现在是 3)
```

每次使用变量的地址或者复制一个指针, 我们就创建了新的别名或者方式来标记同一变量。例如, `*p` 是 `v` 的别名。指针别名允许我们不用变量的名字来访问变量, 这一点是非常有用的, 但是它是双刃剑: 为了找到所有访问变量的语句, 需要知道所有的别名。不仅指针产生别名, 当复制其他引用类型 (像 `slice`、`map`、通道, 甚至包含这里引用类型的结构体、数组和接口) 的值的时候, 也会产生别名。

指针对于 `flag` 包是很关键的, 它使用程序的命令行参数来设置整个程序内某些变量的值。为了说明, 下面这个变种的 `echo` 命令使用两个可选的标识参数: `-n` 使 `echo` 忽略正常输

出时结尾的换行符，`-s sep` 使用 `sep` 替换默认参数输出时使用的空格分隔符。因为这是第 4 版，所以包名字叫作 `gopl.io/ch2/echo4`。

```
gopl.io/ch2/echo4
// echo4 输出其命令行参数
package main

import (
    "flag"
    "fmt"
    "strings"
)

var n = flag.Bool("n", false, "omit trailing newline")
var sep = flag.String("s", " ", "separator")

func main() {
    flag.Parse()
    fmt.Println(strings.Join(flag.Args(), *sep))
    if !*n {
        fmt.Println()
    }
}

flag.Bool 函数创建一个新的布尔标识变量。它有三个参数：标识的名字（"n"），变量的默认值（false），以及当用户提供非法标识、非法参数抑或 -h 或 -help 参数时输出的消息。同样地，flag.String 也使用名字、默认值和消息来创建一个字符串变量。变量 sep 和 n 是指向标识变量的指针，它们必须通过 *sep 和 *n 来访问。
```

当程序运行时，在使用标识前，必须调用 `flag.Parse` 来更新标识变量的默认值。非标识参数也可以从 `flag.Args()` 返回的字符串 slice 来访问。如果 `flag.Parse` 遇到错误，它输出一条帮助消息，然后调用 `os.Exit(2)` 来结束程序。

让我们运行一些 `echo` 测试用例：

```
$ go build gopl.io/ch2/echo4
$ ./echo4 a bc def
a bc def
$ ./echo4 -s / a bc def
a/bc/def
$ ./echo4 -n a bc def
a bc def$
$ ./echo4 -help
Usage of ./echo4:
-n      omit trailing newline
-s string
        separator (default " ")
```

### 2.3.3 new 函数

另外一种创建变量的方式是使用内置的 `new` 函数。表达式 `new(T)` 创建一个未命名的 `T` 类型变量，初始化为 `T` 类型的零值，并返回其地址（地址类型为 `*T`）。

```
p := new(int) // *int 类型的 p，指向未命名的 int 变量
fmt.Println(*p) // 输出 "0"
*p = 2         // 把未命名的 int 设置为 2
fmt.Println(*p) // 输出 "2"
```

使用 `new` 创建的变量和取其地址的普通局部变量没有什么不同，只是不需要引入（和声明）一个虚拟的名字，通过 `new(T)` 就可以直接在表达式中使用。因此 `new` 只是语法上的便

利，不是一个基础概念。

下面两个 `newInt` 函数有同样的行为。

```
func newInt() *int {
    return new(int)
}
func newInt() *int {
    var dummy int
    return &dummy
}
```

每一次调用 `new` 返回一个具有唯一地址的不同变量：

```
p := new(int)
q := new(int)
fmt.Println(p == q) // "false"
```

这个规则有一个例外：两个变量的类型不携带任何信息且是零值，例如 `struct{}` 或 `[0] int`，当前的实现里面，它们有相同的地址。

因为最常见的未命名变量都是结构体类型，它的语法（参考 4.4.1 节）比较复杂，所以 `new` 函数使用得相对较少。

`new` 是一个预声明的函数，不是一个关键字，所以它可以重定义为另外的其他类型，例如：

```
func delta(old, new int) int { return new - old }
```

自然，在 `delta` 函数内，内置的 `new` 函数是不可用的。

### 2.3.4 变量的生命周期

生命周期指在程序执行过程中变量存在的时间段。包级别的变量的生命周期是整个程序的执行时间。相反，局部变量有一个动态的生命周期：每次执行声明语句时创建一个新的实体，变量一直生存到它变得不可访问，这时它占用的存储空间被回收。函数的参数和返回值也是局部变量，它们在其闭包函数被调用的时候创建。

例如，在 1.4 节中的 `lissajous` 示例程序中：

```
for t := 0.0; t < cycles*2*math.Pi; t += res {
    x := math.Sin(t)
    y := math.Sin(t*freq + phase)
    img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                      blackIndex)
}
```

变量 `t` 在每次 `for` 循环的开始创建，变量 `x` 和 `y` 在循环的每次迭代中创建。

那么垃圾回收器如何知道一个变量是否应该被回收？说来话长，基本思路是每一个包级别的变量，以及每一个当前执行函数的局部变量，可以作为追溯该变量的路径的源头，通过指针和其他方式的引用可以找到变量。如果变量的路径不存在，那么变量变得不可访问，因此它不会影响任何其他的计算过程。

因为变量的生命周期是通过它是否可达来确定的，所以局部变量可在包含它的循环的一次迭代之外继续存活。即使包含它的循环已经返回，它的存在还可能延续。

编译器可以选择使用堆或栈上的空间来分配，令人惊奇的是，这个选择不是基于使用 `var` 或 `new` 关键字来声明变量。

```

var global *int

func f() {
    var x int
    x = 1
    global = &x
}

func g() {
    y := new(int)
    *y = 1
}

```

这里，`x`一定使用堆空间，因为它在 `f` 函数返回以后还可以从 `global` 变量访问，尽管它被声明为一个局部变量。这种情况我们说 `x` 从 `f` 中逃逸。相反，当 `g` 函数返回时，变量 `*y` 变得不可访问，可回收。因为 `*y` 没有从 `g` 中逃逸，所以编译器可以安全地在栈上分配 `*y`，即便使用 `new` 函数创建它。任何情况下，逃逸的概念使你不需要额外费心来写正确的代码，但要记住它在性能优化的时候是有好处的，因为每一次变量逃逸都需要一次额外的内存分配过程。

垃圾回收对于写出正确的程序有巨大的帮助，但是免不了考虑内存的负担。不需要显式分配和释放内存，但是变量的生命周期是写出高效程序所必需清楚的。例如，在长生命周期对象中保持短生命周期对象不必要的指针，特别是在全局变量中，会阻止垃圾回收器回收短生命周期的对象空间。

## 2.4 赋值

赋值语句用来更新变量所指的值，它最简单的形式由赋值符 `=`，以及符号左边的变量和右边的表达式组成。

```

x = 1           // 有名称的变量
*p = true       // 间接变量
person.name = "bob" // 结构体成员
count[x] = count[x] * scale // 数组或 slice 或 map 的元素

```

每一个算术和二进制位操作符有一个对应的赋值操作符，例如，最后的那个语句可以重写成：

```
count[x] *= scale
```

它避免了在表达式中重复变量本身。

数字变量也可以通过 `++` 和 `--` 语句进行递增和递减：

```

v := 1
v++ // 等同于 v = v + 1; v 变成 2
v-- // 等同于 v = v - 1; v 变成 1

```

### 2.4.1 多重赋值

另一种形式的赋值是多重赋值，它允许几个变量一次性被赋值。在实际更新变量前，右边所有的表达式被推演，当变量同时出现在赋值符两侧的时候这种形式特别有用，例如，当交换两个变量的值时：

```

x, y = y, x
a[i], a[j] = a[j], a[i]

```

或者计算两个整数的最大公约数：

```

func gcd(x, y int) int {
    for y != 0 {
        x, y = y, x%y
    }
    return x
}

```

或者计算斐波那契数列的第  $n$  个数：

```
func fib(n int) int {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        x, y = y, x+y
    }
    return x
}
```

多重赋值也可以使一个普通的赋值序列变得紧凑：

```
i, j, k = 2, 3, 5
```

从风格上考虑，如果表达式比较复杂，则避免使用多重赋值形式；一系列独立的语句更容易读。

这类表达式（例如一个有多个返回值的函数调用）产生多个值。当在一个赋值语句中使用这样的调用时，左边的变量个数需要和函数的返回值一样多。

```
f, err = os.Open("foo.txt") // 函数调用返回两个值
```

通常函数使用额外的返回值来指示一些错误情况，例如通过 `os.Open` 返回的 `error` 类型，或者一个通常叫 `ok` 的 `bool` 类型变量。我们会在后面的章节中看到，这里有三个操作符也有类似的行为。如果 `map` 查询（参考 4.3 节）、类型断言（参考 7.10 节）或者通道接收动作（参考 8.4.2 节）出现在两个结果的赋值语句中，都会产生一个额外的布尔型结果：

```
v, ok = m[key]           // map 查询
v, ok = x.(T)            // 类型断言
v, ok = <-ch             // 通道接收
```

像变量声明一样，可以将不需要的值赋给空标识符：

```
_ , err = io.Copy(dst, src) // 丢弃字节个数
_ , ok = x.(T)            // 检查类型但丢弃结果
```

## 2.4.2 可赋值性

赋值语句是显式形式的赋值，但是程序中很多地方的赋值是隐式的：一个函数调用隐式地将参数的值赋给对应参数的变量；一个 `return` 语句隐式地将 `return` 操作数赋值给结果变量。复合类型的字面量表达式，例如 `slice`（参考 4.2 节）：

```
medals := []string{"gold", "silver", "bronze"}
```

隐式地给每一个元素赋值，它可以写成下面这样：

```
medals[0] = "gold"
medals[1] = "silver"
medals[2] = "bronze"
```

`map` 和通道的元素尽管不是普通变量，但它们也遵循相似的隐式赋值。

不管隐式还是显式赋值，如果左边的（变量）和右边的（值）类型相同，它就是合法的。通俗地说，赋值只有在值对于变量类型是可赋值的时才合法。

可赋值性根据类型不同有着不同的规则，我们将会在引入新类型的时候解释相应的规则。对已经讨论过的类型，规则很简单：类型必须精准匹配，`nil` 可以被赋给任何接口变量或引用类型。常量（参考 3.6 节）有更灵活的可赋值性规则来规避显式的转换。

两个值使用 `==` 和 `!=` 进行比较与可赋值性相关：任何比较中，第一个操作数相对于第二个操作数的类型必须是可赋值的，或者可以反过来赋值。与可赋值性一样，我们也将解释新类型的可比较性的相关规则。

## 2.5 类型声明

变量或表达式的类型定义这些值应有的特性，例如大小（多少位或多少个元素等）、在内部如何表达、可以对其进行何种操作以及它们所关联的方法。

任何程序中，都有一些变量使用相同的表示方式，但是含义相差非常大。例如，`int` 类型可以用于表示循环的索引、时间戳、文件描述符或月份；`float64` 类型可以表示每秒多少米的速度或精确到几位小数的温度；`string` 类型可以表示密码或者颜色的名字。

`type` 声明定义一个新的命名类型，它和某个已有类型使用同样的底层类型。命名类型提供了一种方式来区分底层类型的不同或者不兼容使用，这样它们就不会在无意中混用。

```
type name underlying-type
```

类型的声明通常出现在包级别，这里命名的类型在整个包中可见，如果名字是导出的（开头使用大写字母），其他的包也可以访问它。

为了说明类型声明，我们把不同计量单位的温度值转换为不同的类型：

```
gopl.io/ch2/tempconv0
// 包 tempconv 进行摄氏温度和华氏温度的转换计算
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC     Celsius = 0
    BoilingC      Celsius = 100
)

func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

这个包定义了两个类型——`Celsius`（摄氏温度）和 `Fahrenheit`（华氏温度），它们分别对应两种温度计量单位。即使使用相同的底层类型 `float64`，它们也不是相同的类型，所以它们不能使用算术表达式进行比较和合并。区分这些类型可以防止无意间合并不同计量单位的温度值；从 `float64` 转换为 `Celsius(t)` 或 `Fahrenheit(t)` 需要显式类型转换。`Celsius(t)` 和 `Fahrenheit(t)` 是类型转换，而不是函数调用。它们不会改变值和表达方式，但改变了显式意义。另一方面，函数 `CToF` 和 `FToC` 用来在两种温度计量单位之间转换，返回不同的数值。

对于每个类型 `T`，都有一个对应的类型转换操作 `T(x)` 将值 `x` 转换为类型 `T`。如果两个类型具有相同的底层类型或二者都是指向相同底层类型变量的未命名指针类型，则二者是可以相互转换的。类型转换不改变类型值的表达方式，仅改变类型。如果 `x` 对于类型 `T` 是可赋值的，类型转换也是允许的，但是通常是不必要的。

数字类型间的转换，字符串和一些 `slice` 类型间的转换是允许的，我们将在下一章详细讨论。这些转换会改变值的表达方式。例如，从浮点型转化为整型会丢失小数部分，从字符串转换成字节（`[]byte`）`slice` 会分配一份字符串数据副本。任何情况下，运行时的转换不会失败。

命名类型的底层类型决定了它的结构和表达方式，以及它支持的内部操作集合，这些内部操作与直接使用底层类型的情况相同。正如你所预期的，它意味着对于 Celsius 和 Fahrenheit 类型可以使用与 float64 相同的算术操作符。

```
fmt.Printf("%g\n", BoilingC-FreezingC) // "100" °C
boilingF := CToF(BoilingC)
fmt.Printf("%g\n", boilingF-CToF(FreezingC)) // "180" °F
fmt.Printf("%g\n", boilingF-FreezingC)      // 编译错误：类型不匹配
```

通过 == 和 < 之类的比较操作符，命名类型的值可以与其相同类型的值或者底层类型相同的未命名类型的值相比较。但是不同命名类型的值不能直接比较：

```
var c Celsius
var f Fahrenheit
fmt.Println(c == 0)           // "true"
fmt.Println(f >= 0)          // "true"
fmt.Println(c == f)          // 编译错误：类型不匹配
fmt.Println(c == Celsius(f)) // "true"!
```

注意最后一种情况。无论名字如何，类型转换 Celsius(f) 没有改变参数的值，只改变其类型。测试结果是真，因为 c 和 f 的值都是 0。

命名类型提供了概念上的便利，避免一遍遍地重复写复杂的类型。当底层类型是像 float64 这样简单的类型时，好处就不大了，但是对于我们将讨论到的复杂结构体类型，好处就很大，在讨论结构体时将介绍这一点。

下面的声明中，Celsius 参数 c 出现在函数名字前面，名字叫 String 的方法关联到 Celsius 类型，返回 c 变量的数字值，后面跟着摄氏温度的符号°C。

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

很多类型都声明这样一个 String 方法，在变量通过 fmt 包作为字符串输出时，它可以控制类型值的显示方式，我们将在 7.1 节中看到。

```
c := FToC(212.0)
fmt.Println(c.String()) // "100°C"
fmt.Printf("%v\n", c) // "100°C"; 不需要显式调用字符串
fmt.Printf("%s\n", c) // "100°C"
fmt.Println(c)        // "100°C"
fmt.Printf("%g\n", c) // "100"; 不调用字符串
fmt.Println(float64(c)) // "100"; 不调用字符串
```

## 2.6 包和文件

在 Go 语言中包的作用和其他语言中的库或模块作用类似，用于支持模块化、封装、编译隔离和重用。一个包的源代码保存在一个或多个以 .go 结尾的文件中，它所在目录名的尾部就是包的导入路径，例如，`gopl.io/ch1/helloworld` 包的文件存储在目录 `$GOPATH/src/gopl.io/ch1/helloworld` 中。

每一个包给它的声明提供独立的命名空间。例如，在 `image` 包中，`Decode` 标识符和 `unicode/utf16` 包中的标识符一样，但是关联了不同的函数。为了从包外部引用一个函数，我们必须明确修饰标识符来指明所指的是 `image.Decode` 或 `utf16.Decode`。

包让我们可以通过控制变量在包外面的可见性或导出情况来隐藏信息。在 Go 里，通过一条简单的规则来管理标识符是否对外可见：导出的标识符以大写字母开头。

为了说明基本原理，假设温度转换软件很受欢迎，我们想把它作为新包贡献给 Go 社

区，将要怎么做呢？

我们创建一个叫作 `gopl.io/ch2/tempconv` 的包，这是前面例子的变种（这里我们没有照惯例对例子进行顺序编号，目的是让包路径更实际一些）。包自己保存在两个文件里，以展示如何访问一个包里面多个独立文件中的声明。现实中，像这样的小包可能只需要一个文件。

将类型、它们的常量及方法的声明放在 `tempconv.go` 中：

```
gopl.io/ch2/tempconv
// tempconv 包负责摄氏温度与华氏温度的转换
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64
const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC     Celsius = 0
    BoilingC      Celsius = 100
)
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", f) }
```

将转换函数放在 `conv.go` 中：

```
package tempconv

// CToF 把摄氏温度转换为华氏温度
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }

// FToC 把华氏温度转换为摄氏温度
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

每一个文件的开头用 `package` 声明定义包的名称。当导入包时，它的成员通过诸如 `tempconv.CToF` 等方式被引用。如果包级别的名字（像类型和常量）在包的一个文件中声明，就像所有的源代码在同一个文件中一样，它们对于同一个包中的其他文件可见。注意，`tempconv.go` 导入 `fmt` 包，但是 `conv.go` 没有，因为它本身没有用到 `fmt` 包。

因为包级别的常量名字以大写字母开头，所以它们也可以使用修饰过的名称（如 `tempconv.AbsoluteZeroC`）来访问：

```
fmt.Printf("Brrrr! %v\n", tempconv.AbsoluteZeroC) // "Brrrr! -273.15°C"
```

为了在某个包里将摄氏温度转换为华氏温度，导入包 `gopl.io/ch2/tempconv`，然后编写下面的代码：

```
fmt.Println(tempconv.CToF(tempconv.BoilingC)) // "212°F"
```

`package` 声明前面紧挨着的文档注释（参考 10.7.4 节）对整个包进行描述。习惯上，应该在开头用一句话对包进行总结性的描述。每一个包里只有一个文件应该包含该包的文档注释。扩展的文档注释通常放在一个文件中，按惯例名字叫作 `doc.go`。

**练习 2.1：**添加类型、常量和函数到 `tempconv` 包中，处理以开尔文为单位（K）的温度值， $0\text{K} = -273.15\text{°C}$ ，变化  $1\text{K}$  和变化  $1\text{°C}$  是等价的。

## 2.6.1 导入

在 Go 程序里，每一个包通过称为导入路径（import path）的唯一字符串来标识。它们

出现在诸如 "gopl.io/ch2/tempconv" 之类的 import 声明中。语言的规范没有定义哪些字符串从哪来以及它们的含义，这依赖于工具来解释。当使用 go 工具（参考第 10 章）时，一个导入路径标注一个目录，目录中包含构成包的一个或多个 Go 源文件。除了导入路径之外，每个包还有一个包名，它以短名字的形式（且不必是唯一的）出现在包的声明中。按约定，包名匹配导入路径的最后一段，这样可以方便地预测 gopl.io/ch2/tempconv 的包名是 tempconv。

为了使用 gopl.io/ch2/tempconv，必须导入它：

```
gopl.io/ch2/cf
// cf 把它的数值参数转换为摄氏温度和华氏温度
package main

import (
    "fmt"
    "os"
    "strconv"

    "gopl.io/ch2/tempconv"
)

func main() {
    for _, arg := range os.Args[1:] {
        t, err := strconv.ParseFloat(arg, 64)
        if err != nil {
            fmt.Fprintf(os.Stderr, "cf: %v\n", err)
            os.Exit(1)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n",
            f, tempconv.FToC(f), c, tempconv.CToF(c))
    }
}
```

导入声明可以给导入的包绑定一个短名字，用来在整个文件中引用包的内容。上面的 import 可以使用修饰标识符来引用 gopl.io/ch2/tempconv 包里的变量名，如 tempconv.CToF。默认这个短名字是包名，在本例中是 tempconv，但是导入声明可以设定一个可选的名字来避免冲突（参考 10.4 节）。

cf 程序将一个数字型的命令行参数分别转换成摄氏温度和华氏温度：

```
$ go build gopl.io/ch2/cf
$ ./cf 32
32°F = 0°C, 32°C = 89.6°F
$ ./cf 212
212°F = 100°C, 212°C = 413.6°F
$ ./cf -40
-40°F = -40°C, -40°C = -40°F
```

如果导入一个没有被引用的包，就会触发一个错误。这个检查帮助消除代码演进过程中不再需要的依赖（尽管它在调试过程中会带来一些麻烦），因为注释掉一条诸如 log.Println("got here!") 之类的代码，可能去除了对于 log 包唯一的一个引用，导致编译器报错。这种情况下，需要注释掉或者删掉不必要的 import。

**练习 2.2：**写一个类似于 cf 的通用的单位转换程序，从命令行参数或者标准输入（如果没有参数）获取数字，然后将每一个数字转换为以摄氏温度和华氏温度表示的温度，以英寸和米表示的长度单位，以磅和千克表示的重量，等等。

## 2.6.2 包初始化

包的初始化从初始化包级别的变量开始，这些变量按照声明顺序初始化，在依赖已解析完毕的情况下，根据依赖的顺序进行。

```
var a = b + c      // 最后把 a 初始化为 3
var b = f()        // 通过调用 f 接着把 b 初始化为 2
var c = 1          // 首先初始化为 1

func f() int { return c + 1 }
```

如果包由多个 .go 文件组成，初始化按照编译器收到文件的顺序进行：go 工具会在调用编译器前将 .go 文件进行排序。

对于包级别的每一个变量，生命周期从其值被初始化开始，但是对于其他一些变量，比如数据表，初始化表达式不是简单地设置它的初始化值。这种情况下，`init` 函数的机制会比较简单。任何文件可以包含任意数量的声明如下的函数：

```
func init() { /* ... */ }
```

这个 `init` 函数不能被调用和被引用，另一方面，它也是普通的函数。在每一个文件里，当程序启动的时候，`init` 函数按照它们声明的顺序自动执行。

包的初始化按照在程序中导入的顺序来进行，依赖顺序优先，每次初始化一个包。因此，如果包 `p` 导入了包 `q`，可以确保 `q` 在 `p` 之前已完全初始化。初始化过程是自下向上的，`main` 包最后初始化。在这种方式下，在程序的 `main` 函数开始执行前，所有的包已初始化完毕。

下面的包定义了一个 `PopCount` 函数，它返回一个数字中被置位的个数，即在一个 `uint64` 的值中，值为 1 的位的个数，这称为种群统计。它使用 `init` 函数来针对每一个可能的 8 位值预计一个结果表 `pc`，这样 `PopCount` 只需要将 8 个快查表的结果相加而不用进行 64 步的计算。（这个不是最快的统计位算法，只是方便用来说明 `init` 函数，用来展示如何预计一个数值表，它是一种很有用的编程技术。）

```
gopl.io/ch2/popcount

package popcount

// pc[i] 是 i 的种群统计
var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}

// PopCount 返回 x 的种群统计（置位的个数）
func PopCount(x uint64) int {
    return int(pc[byte(x>>(0*8))] +
               pc[byte(x>>(1*8))] +
               pc[byte(x>>(2*8))] +
               pc[byte(x>>(3*8))] +
               pc[byte(x>>(4*8))] +
               pc[byte(x>>(5*8))] +
               pc[byte(x>>(6*8))] +
               pc[byte(x>>(7*8))])
}
```

注意，`init` 中的 `range` 循环只使用索引；值不是必需的，所以没必要包含进来。循环可以重写为下面的形式：

```
for i, _ := range pc {
```

我们将在下一节和 10.5 节看到 `init` 函数的其他用途。

**练习 2.3：** 使用循环重写 `PopCount` 来代替单个表达式。对比两个版本的效率。(11.4 节会展示如何系统性地对比不同实现的性能。)

**练习 2.4：** 写一个用于统计位的 `PopCount`，它在其实际参数的 64 位上执行移位操作，每次判断最右边的位，进而实现统计功能。把它与快查表版本的性能进行对比。

**练习 2.5：** 使用 `x&(x-1)` 可以清除 `x` 最右边的非零位，利用该特点写一个 `PopCount`，然后评价它的性能。

## 2.7 作用域

声明将名字和程序实体关联起来，如一个函数或一个变量。声明的作用域是指用到声明时所声明名字的源代码段。  
IP

不要将作用域和生命周期混淆。声明的作用域是声明在程序文本中出现的区域，它是一个编译时属性。变量的生命周期是变量在程序执行期间能被程序的其他部分所引用的起止时间，它是一个运行时属性。

语法块（block）是由大括号围起来的一个语句序列，比如一个循环体或函数体。在语法块内部声明的变量对块外部不可见。块把声明包围起来，并且决定了它的可见性。我们可以把块的概念推广到其他没有显式包含在大括号中的声明代码，将其统称为词法块。包含了全部源代码的词法块，叫作全局块。每一个包，每一个文件，每一个 `for`、`if` 和 `switch` 语句，以及 `switch` 和 `select` 语句中的每一个条件，都是写在一个词法块里的。当然，显式写在大括号语法里的代码块也算是一个词法块。

一个声明的词法块决定声明的作用域大小。像 `int`、`len` 和 `true` 等内置类型、函数或常量在全局块中声明并且对于整个程序可见。在包级别（就是在任何函数外）的声明，可以被同一个包里的任何文件引用。导入的包（比如在 `tempconv` 例子中的 `fmt`）是文件级别的，所以它们可以在同一个文件内引用，但是不能在没有另一个 `import` 语句的前提下被同一个包中其他文件中的东西引用。许多声明（像 `tempconv.CToF` 函数中变量 `c` 的声明）是局部的，仅可在同一个函数中或者仅仅是函数的一部分所引用。

控制流标签（如 `break`、`continue` 和 `goto` 语句使用的标签）的作用域是整个外层的函数（enclosing function）。

一个程序可以包含多个同名的声明，前提是它们在不同词法块中。例如可以声明一个和包级别变量同名的局部变量。或者像 2.3.3 节展示的，可以声明一个叫作 `new` 的参数，即使它是一个全局块中预声明的函数。然而，不要滥用，重声明所涉及的作用域越广，越可能影响其他的代码。

当编译器遇到一个名字的引用时，将从最内层的封闭词法块到全局块寻找其声明。如果没有找到，它会报“undeclared name”错误；如果在内层和外层块都存在这个声明，内层的将先被找到。这种情况下，内层声明将覆盖外部声明，使它不可访问：

```
func f() {}  
var g = "g"
```

```
func main() {
    f := "f"
    fmt.Println(f) // "f"; 局部变量 f 覆盖了包级函数 f
    fmt.Println(g) // "g"; 包级变量
    fmt.Println(h) // 编译错误: 未定义 h
}
```

在函数里面，词法块可能嵌套很深，所以一个局部变量声明可能覆盖另一个。很多词法块使用 `if` 语句和 `for` 循环这类控制流结构构建。下面的程序有三个称为 `x` 的不同的变量声明，因为每个声明出现在不同的词法块。(这个例子只是用来说明作用域的规则，风格并不完美！)

```
func main() {
    x := "hello!"
    for i := 0; i < len(x); i++ {
        x := x[i]
        if x != '!' {
            x := x + 'A' - 'a'
            fmt.Printf("%c", x) // "HELLO" (每次迭代一个字母)
        }
    }
}
```

表达式 `x[i]` 和 `x + 'A' - 'a'` 都引用了在外层声明的 `x`，稍后我们会解释它。(注意，后面的表达式不同于 `unicode.ToUpper` 函数。)

如上所述，不是所有的词法块都对应于显式大括号包围的语句序列，有一些词法块是隐式的。`for` 循环创建了两个词法块：一个是循环体本身的显式块，以及一个隐式块，它包含了一个闭合结构，其中就有初始化语句中声明的变量，如变量 `i`。隐式块中声明的变量的作用域包括条件、后置语句 (`i++`)，以及 `for` 语句体本身。

下面的例子也有三个名字为 `x` 的变量，每一个都在不同的词法块中声明：一个在函数体中，一个在 `for` 语句块中，一个在循环体中。但只有两个块是显式的：

```
func main() {
    x := "hello"
    for _, x := range x {
        x := x + 'A' - 'a'
        fmt.Printf("%c", x) // "HELLO" (每次迭代一个字母)
    }
}
```

像 `for` 循环一样，除了本身的主体块之外，`if` 和 `switch` 语句还会创建隐式的词法块。下面的 `if-else` 链展示 `x` 和 `y` 的作用域：

```
if x := f(); x == 0 {
    fmt.Println(x)
} else if y := g(x); x == y {
    fmt.Println(x, y)
} else {
    fmt.Println(x, y)
}
fmt.Println(x, y) // 编译错误: x 与 y 在这里不可见
```

第二个 `if` 语句嵌套在第一个中，所以第一个语句的初始化部分声明的变量在第二个语句中是可见的。同样的规则可以应用于 `switch` 语句：条件对应一个块，每个 `case` 语句体对应一个块。

在包级别，声明的顺序和它们的作用域没有关系，所以一个声明可以引用它自己或者跟在它后面的其他声明，使我们可以声明递归或相互递归的类型和函数。如果常量或变量声明引用它自己，则编译器会报错。

在以下程序中：

```
if f, err := os.Open(fname); err != nil { // 编译错误：未使用 f
    return err
}
f.Stat()      // 编译错误：未定义 f
f.Close()     // 编译错误：未定义 f
```

`f` 变量的作用域是 `if` 语句，所以 `f` 不能被接下来的语句访问，编译器会报错。根据编译器的不同，也可能收到其他报错：局部变量 `f` 没有使用。

所以通常需要在条件判断之前声明 `f`，使其在 `if` 语句后面可以访问：

```
f, err := os.Open(fname)
if err != nil {
    return err
}
f.Stat()
f.Close()
```

你可能希望避免在外部块中声明 `f` 和 `err`，方法是将 `Stat` 和 `Close` 的调用放到 `else` 块中：

```
if f, err := os.Open(fname); err != nil {
    return err
} else {
    // f 与 err 在这里可见
    f.Stat()
    f.Close()
}
```

通常 Go 中的做法是在 `if` 块中处理错误然后返回，这样成功执行的路径不会被变得支离破碎。

短变量声明依赖一个明确的作用域。考虑下面的程序，它获取当前的工作目录然后把它保存在一个包级别的变量里。这通过在 `main` 函数中调用 `os.Getwd` 来完成，但是最好可以从主逻辑中分离，特别是在获取目录失败是致命错误的情况下。函数 `log.Fatalf` 输出一条消息，然后调用 `os.Exit(1)` 退出。

```
var cwd string

func init() {
    cwd, err := os.Getwd() // 编译错误：未使用 cwd
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

因为 `cwd` 和 `err` 在 `init` 函数块的内部都尚未声明，所以 `:=` 语句将它们都声明为局部变量。内层 `cwd` 的声明让外部的声明不可见，所以这个语句没有按预期更新包级别的 `cwd` 变量。

当前 Go 编译器检测到局部的 `cwd` 变量没有被使用，然后报错，但是不必严格执行这种检查。进一步做一个小的修改，比如增加引用局部 `cwd` 变量的日志语句就可以让检查失效。

```
var cwd string  
func init() {  
    cwd, err := os.Getwd() // 注意：错误  
    if err != nil {  
        log.Fatalf("os.Getwd failed: %v", err)  
    }  
    log.Printf("Working directory = %s", cwd)  
}
```

全局的 `cwd` 变量依然未初始化，看起来一个普通的日志输出让 bug 变得不明显。

处理这种潜在的问题有许多方法。最直接的方法是在另一个 `var` 声明中声明 `err`，避免使用 `:=`。

```
var cwd string  
func init() {  
    var err error  
    cwd, err = os.Getwd()  
    if err != nil {  
        log.Fatalf("os.Getwd failed: %v", err)  
    }  
}
```

现在我们已经看到包、文件、声明以及语句是如何来构成程序的。接下来的两章将要讨论数据的结构。

# 基本数据

毫无疑问，计算机底层全是位，而实际操作则是基于大小固定的单元中的数值，称为字 (word)，这些值可解释为整数、浮点数、位集 (bitset) 或内存地址等，进而构成更大的聚合体，以表示数据包、像素、文件、诗集，以及其他种种。Go 的数据类型宽泛，并有多种组织方式，向下匹配硬件特性，向上满足程序员所需，从而可以方便地表示复杂数据结构。

Go 的数据类型分四大类：基础类型 (basic type)、聚合类型 (aggregate type)、引用类型 (reference type) 和接口类型 (interface type)。本章的主题是基础类型，包括数字 (number)、字符串 (string) 和布尔型 (boolean)。聚合类型——数组 (array，见 4.1 节) 和结构体 (struct，见 4.4 节) ——是通过组合各种简单类型得到的更复杂的数据类型。引用是一大分类，其中包含多种不同类型，如指针 (pointer，见 2.3.2 节)，slice (见 4.2 节)，map (见 4.3 节)，函数 (function，见第 5 章)，以及通道 (channel，见第 8 章)。它们的共同点是全都间接指向程序变量或状态，于是操作所引用数据的效果就会遍及该数据的全部引用。接口类型将在第 7 章讨论。

## 3.1 整数

Go 的数值类型包括了几种不同大小的整数、浮点数和复数。各种数值类型分别有自己的大小，对正负号支持也各异。我们从整数开始。

Go 同时具备有符号整数和无符号整数。有符号整数分四种大小：8 位、16 位、32 位、64 位，用 `int8`、`int16`、`int32`、`int64` 表示，对应的无符号整数是 `uint8`、`uint16`、`uint32`、`uint64`。

此外还有两种类型 `int` 和 `uint`。在特定平台上，其大小与原生的有符号整数\无符号整数相同，或等于该平台上的运算效率最高的值。`int` 是目前使用最广泛的数值类型。这两种类型大小相等，都是 32 位或 64 位，但不能认为它们一定就是 32 位，或一定就是 64 位；即使在同样的硬件平台上，不同的编译器可能选用不同的大小。

`rune` 类型是 `int32` 类型的同义词，常常用于指明一个值是 Unicode 码点 (code point)。这两个名称可互换使用。同样，`byte` 类型是 `uint8` 类型的同义词，强调一个值是原始数据，而非量值。

最后，还有一种无符号整数 `uintptr`，其大小并不明确，但足以完整存放指针。`uintptr` 类型仅仅用于底层编程，例如在 Go 程序与 C 程序库或操作系统的接口界面。第 13 章介绍 `unsafe` 包，将会结合 `uintptr` 举例。

`int`、`uint` 和 `uintptr` 都有别于其大小明确的相似类型的类型。就是说，`int` 和 `int32` 是不同类型，尽管 `int` 天然的大小就是 32 位，并且 `int` 值若要当作 `int32` 使用，必须显式转换；反之亦然。

有符号整数以补码表示，保留最高位作为符号位， $n$  位数字的取值范围是  $-2^{n-1} \sim 2^{(n-1)} - 1$ 。无符号整数由全部位构成其非负值，范围是  $0 \sim 2^n - 1$ 。例如，`int8` 可以从 -128 到 127 取值，而 `unit8` 从 0 到 255 取值。

Go 的二元操作符涵盖了算术、逻辑和比较等运算。按优先级的降序排列如下：

*	/	%	<<	>>	&	& <sup>^</sup>
+	-		<sup>^</sup>			
==	!=	<	<=	>	>=	
&&						

二元运算符分五大优先级。同级别的运算符满足左结合律，为求清晰，可能需要圆括号，或为使表达式内的运算符按指定次序计算，如 `mask & (1<<28)`。

上述列表中前两行的运算符（如加法运算 `+`）都有对应的赋值运算符（如 `+=`），用于简写赋值语句。

算术运算符 `+`、`-`、`*`、`/` 可应用于整数、浮点数和复数，而取模运算符 `%` 仅能用于整数。取模运算符 `%` 的行为因编程语言而异。就 Go 而言，取模余数的正负号总是与被除数一致，于是 `-5%3` 和 `-5%-3` 都得 `-2`。除法运算 `(/)` 的行为取决于操作数是否都为整型，整数相除，商会舍弃小数部分，于是 `5.0/4.0` 得到 `1.25`，而 `5/4` 结果是 `1`。

不论是有符号数还是无符号数，若表示算术运算结果所需的位超出该类型的范围，就称为溢出。溢出的高位部分会无提示地丢弃。假如原本的计算结果是有符号类型，且最左侧位是 `1`，则会形成负值，以 `int8` 为例：

```
var u uint8 = 255
fmt.Println(u, u+1, u*u) // "255 0 1"

var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

下列二元比较运算符用于比较两个类型相同的整数；比较表达式本身的类型是布尔型。

==	等于
!=	不等于
<	小于
<=	小于或等于
>	大于
>=	大于等于

实际上，全部基本类型的值（布尔值、数值、字符串）都可以比较，这意味着两个相同类型的值可用 `==` 和 `!=` 运算符比较。整数、浮点数和字符串还能根据比较运算符排序。许多其他类型的值是不可比较的，也无法排序。后面介绍每种类型时，我们将分别说明比较规则。

另外，还有一元加法和一元减法运算符：

+	一元取正（无实际影响）
-	一元取负

对于整数，`+x` 是 `0+x` 的简写，而 `-x` 则为 `0-x` 的简写。对于浮点数和复数，`+x` 就是 `x`，`-x` 为 `x` 的负数。

Go 也具备下列位运算符，前四个对操作数的运算逐位独立进行，不涉及算术进位或正负号：

&	位运算 AND
	位运算 OR
<sup>^</sup>	位运算 XOR
& <sup>^</sup>	位清空 (AND NOT)
<<	左移
>>	右移

如果作为二元运算符，运算符`^`表示按位“异或”（XOR）；若作为一元前缀运算符，则它表示按位取反或按位取补，运算结果就是操作数逐位取反。运算符`&^`是按位清除（AND NOT）：表达式`z=x&^y`中，若`y`的某位是1，则`z`的对应位等于0；否则，它就等于`x`的对应位。

下面的代码说明了如何用位运算将一个`uint8`值作为位集（bitset）处理，其含有8个独立的位，高效且紧凑。`Printf`用谓词`%b`以二进制形式输出数值，副词`08`在这个输出结果前被零，补够8位。

```
var x uint8 = 1<<1 | 1<<5
var y uint8 = 1<<1 | 1<<2

fmt.Printf("%08b\n", x)    // "00100010", 集合{1, 5}
fmt.Printf("%08b\n", y)    // "00000010", 集合{1, 2}

fmt.Printf("%08b\n", x&y) // "00000010", 交集{1}
fmt.Printf("%08b\n", x|y) // "00100110", 并集{1, 2, 5}
fmt.Printf("%08b\n", x^y) // "00100100", 对称差{2, 5}
fmt.Printf("%08b\n", x&^y) // "00100000", 差集{5}

for i := uint(0); i < 8; i++ {
    if x&(1<<i) != 0 { // 元素判定
        fmt.Println(i) // "1", "5"
    }
}

fmt.Printf("%08b\n", x<<1) // "01000100", 集合{2, 6}
fmt.Printf("%08b\n", x>>1) // "00010001", 集合{0, 4}
```

（6.5节会介绍比单字节大得多的整数位集的实现。）

在移位运算`x<<n`和`x>>n`中，操作数`n`决定位移量，而且`n`必须为无符号型；操作数`x`可以是有符号型也可以是无符号型。算术上，左移运算`x<<n`等价于`x`乘以`2^n`；而右移运算`x>>n`等价于`x`除以`2^n`，向下取整。

左移以0填补右边空位，无符号整数右移同样以0填补左边空位，但有符号数的右移操作是按符号位的值填补空位。因此，请注意，如果将整数以位模式处理，须使用无符号整型。

尽管Go具备无符号整型数和相关算术运算，也尽管某些量值不可能为负，但是我们往往还采用有符号整型数，如数组的长度（即便直观上明显更应该选用`uint`）。下例从后向前输出奖牌名称，循环里用到了内置的`len`函数，它返回有符号整数：

```
medals := []string{"gold", "silver", "bronze"}
for i := len(medals) - 1; i >= 0; i-- {
    fmt.Println(medals[i]) // "bronze", "silver", "gold"
}
```

相反，假若`len`返回的结果是无符号整数，就会导致严重错误，因为`i`随之也成为`uint`型，根据定义，条件`i>=0`将恒成立。第3轮迭代后，有`i==0`，语句`i--`使得`i`变为`uint`型的最大值（例如，可能为`2^64-1`），而非`-1`，导致`medals[i]`试图越界访问元素，超出slice范围，引发运行失败或宕机（见5.9节）。

因此，无符号整数往往只用于位运算符和特定算术运算符，如实现位集时，解析二进制格式的文件，或散列和加密。一般而言，无符号整数极少用于表示非负值。

通常，将某种类型的值转换成另一种，需要显式转换。对于算术和逻辑（不含移位）的二元运算符，其操作数的类型必须相同。虽然这有时会导致表达式相对冗长，但是一整类错误得以避免，程序也更容易理解。

考虑下面的语句，它与某些其他场景类似：

```
var apples int32 = 1
var oranges int16 = 2
var compote int = apples + oranges // 编译错误
```

尝试编译这三个声明将产生错误消息：

```
非法操作：apples + oranges (int32 与 int16 类型不匹配)
```

类型不匹配（+ 的问题）有几种方法改正，最直接地，将全部操作数转换成同一类型：

```
var compote = int(apples) + int(oranges)
```

2.5 节已经提及，于每种类型  $\tau$ ，若允许转换，操作  $\tau(x)$  会将  $x$  的值转换成类型  $\tau$ 。很多整型 – 整型转换不会引起值的变化，仅告知编译器应如何解读该值。不过，缩减大小的整型转换，以及整型与浮点型的相互转换，可能改变值或损失精度：

```
f := 3.141 // a float64
i := int(f)
fmt.Println(f, i) // "3.141 3"
f = 1.99
fmt.Println(int(f)) // "1"
```

浮点型转成整型，会舍弃小数部分，趋零截尾（正值向下取整，负值向上取整）。如果有些转换的操作数的值超出了目标类型的取值范围，就应当避免这种转换，因为其行为依赖具体实现：

```
f := 1e100 // a float64
i := int(f) // 结果依赖实现
```

不论有无大小和符号限制，源码中的整数都能写成常见的十进制数；也能写成八进制数，以  $0$  开头，如  $0666$ ；还能写成十六进制数，以  $0x$  或  $0X$  开头，如  $0xdeadbeef$ 。十六进制的数字（或字母）大小写皆可。当前，八进制数似乎仅有一种用途——表示 POSIX 文件系统的权限——而十六进制数广泛用于强调其位模式，而非数值大小。

如下例所示，如果使用 `fmt` 包输出数字，我们可以用谓词 `%d`、`%o` 和 `%x` 指定进位制基数和输出格式：

```
o := 0666
fmt.Printf("%d %[1]o %#[1]o\n", o) // "438 666 0666"
x := int64(0xdeadbeef)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)
// 输出：
// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

注意 `fmt` 的两个技巧。通常 `Printf` 的格式化字符串含有多个 `%` 谓词，这要求提供相同数目的操作数，而 `%` 后的副词 `[1]` 告知 `Printf` 重复使用第一个操作数。其次，`%o`、`%x` 或 `%X` 之前的副词 `#` 告知 `Printf` 输出相应的前缀  $0$ 、 $0x$  或  $0X$ 。

源码中，文字符号（rune literal）的形式是字符写在一对单引号内。最简单的例子就是 ASCII 字符，如 `'a'`，但也可以直接使用 Unicode 码点（codepoint）或码值转义，稍后有介绍。

用 `%c` 输出文字符号，如果希望输出带有单引号则用 `%q`：

```
ascii := 'a'
unicode := '国'
newline := '\n'
fmt.Printf("%d %[1]c %[1]q\n", ascii) // "97 a 'a'"
fmt.Printf("%d %[1]c %[1]q\n", unicode) // "22269 国 '国'"
fmt.Printf("%d %[1]q\n", newline) // "10 '\n'"
```

## 3.2 浮点数

Go 具有两种大小的浮点数 `float32` 和 `float64`。其算术特性遵从 IEEE 754 标准，所有新式 CPU 都支持该标准。

这两个类型的值可从极细微到超宏大。`math` 包给出了浮点值的极限。常量 `math.MaxFloat32` 是 `float32` 的最大值，大约为 `3.4e38`，而 `math.MaxFloat64` 则大约为 `1.8e308`。相应地，最小的正浮点值大约为 `1.4e-45` 和 `4.9e-324`。

十进制下，`float32` 的有效数字大约是 6 位，`float64` 的有效数字大约是 15 位。绝大多数情况下，应优先选用 `float64`，因为除非格外小心，否则 `float32` 的运算会迅速累积误差。另外，`float32` 能精确表示的正整数范围有限：

```
var f float32 = 16777216 // 1 << 24
fmt.Println(f == f+1) // "true"
```

在源码中，浮点数可写成小数，如：

```
const e = 2.71828 // (近似值)
```

小数点前的数字可以省略 (`.707`)，后面的也可省去 (`1.`)。非常小或非常大的数字最好使用科学记数法表示，此方法在数量级指数前写字母 `e` 或 `E`：

```
const Avogadro = 6.02214129e23
const Planck = 6.62606957e-34
```

浮点值能方便地通过 `Printf` 的谓词 `%g` 输出，该谓词会自动保持足够的精度，并选择最简洁的表示方式，但是对于数据表，`%e` (有指数) 或 `%f` (无指数) 的形式可能更合适。这三个谓词都能掌控输出宽度和数值精度。

```
for x := 0; x < 8; x++ {
    fmt.Printf("x = %d e^x = %8.3f\n", x, math.Exp(float64(x)))
}
```

上面的代码按 8 个字符的宽度输出自然对数 `e` 的各个幂方，结果保留三位小数：

```
x = 0  e^x = 1.000
x = 1  e^x = 2.718
x = 2  e^x = 7.389
x = 3  e^x = 20.086
x = 4  e^x = 54.598
x = 5  e^x = 148.413
x = 6  e^x = 403.429
x = 7  e^x = 1096.633
```

除了大量常见的数学函数之外，`math` 包还有函数用于创建和判断 IEEE 754 标准定义的特殊值：正无穷大和负无穷大，它表示超出最大许可值的数及除以零的商；以及 `NaN` (Not a Number)，它表示数学上无意义的运算结果 (如 `0/0` 或 `Sqrt(-1)`)。

```
var z float64
fmt.Println(z, -z, 1/z, -1/z, z/z) // "0 -0 +Inf -Inf NaN"
```

`math.IsNaN` 函数判断其参数是否是非数值，`math.NaN` 函数则返回非数值 (`NaN`)。在数字运算中，我们倾向于将 `NaN` 当作信号值 (sentinel value)，但直接判断具体的计算结果是否为 `NaN` 可能导致潜在错误，因为与 `NaN` 的比较总不成立 (除了 `!=`，它总是与 `==` 相反)：

```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
```

一个函数的返回值是浮点型且它有可能出错，那么最好单独报错，如下：

```
func compute() (value float64, ok bool) {
    // ...
    if failed {
        return 0, false
    }
    return result, true
}
```

下一个程序以浮点绘图运算为例。它根据传入两个参数的函数  $z=f(x,y)$ ，绘出三维的网线状曲面，绘制过程中运用了可缩放矢量图形（Scalable Vector Graphics，SVG），绘制线条的一种标准 XML 格式。图 3-1 是函数  $\sin(r)/r$  的图形输出样例，其中  $r$  为  $\sqrt{x^2+y^2}$ 。

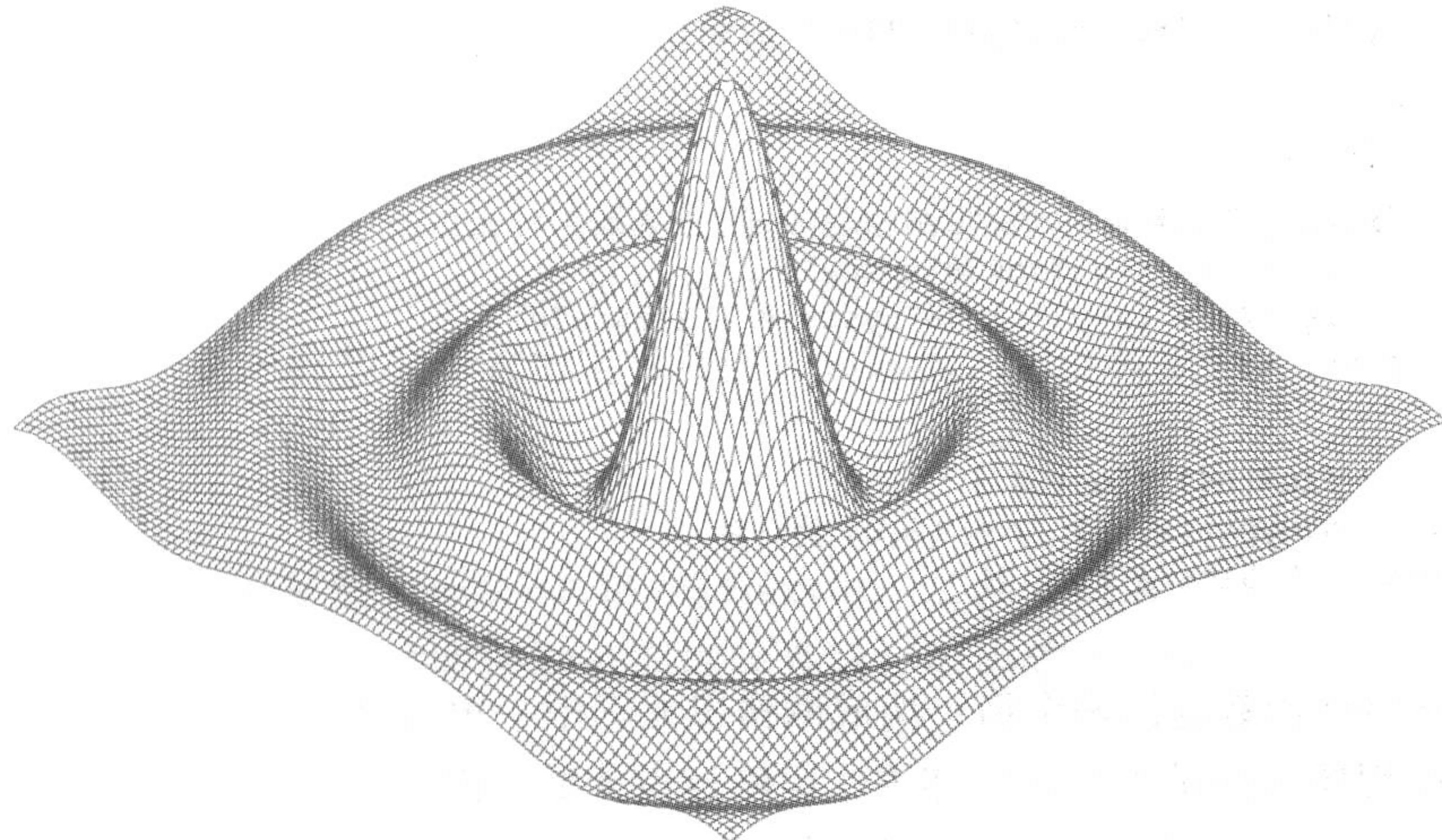


图 3-1 函数  $\sin(r)/r$  的图形输出样例

[gopl.io/ch3/surface](http://gopl.io/ch3/surface)

```
// surface 函数根据一个三维曲面函数计算并生成 SVG
package main

import (
    "fmt"
    "math"
)

const (
    width, height = 600, 320           // 以像素表示的画布大小
    cells          = 100                // 网格单元的个数
    xyrange        = 30.0               // 坐标轴的范围 (-xyrange..+xyrange)
    xyscale        = width / 2 / xyrange // x 或 y 轴上每个单位长度的像素
    zscale         = height * 0.4       // z 轴上每个单位长度的像素
    angle          = math.Pi / 6         // x、y 轴的角度 (=30°)
)

var sin30, cos30 = math.Sin(angle), math.Cos(angle) // sin(30°), cos(30°)

func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
        "width='%d' height='%d'>", width, height)
```

```

for i := 0; i < cells; i++ {
    for j := 0; j < cells; j++ {
        ax, ay := corner(i+1, j)
        bx, by := corner(i, j)
        cx, cy := corner(i, j+1)
        dx, dy := corner(i+1, j+1)
        fmt.Printf("<polygon points='%.g %.g %.g %.g %.g %.g' />\n",
                  ax, ay, bx, by, cx, cy, dx, dy)
    }
}
fmt.Println("</svg>")
}

func corner(i, j int) (float64, float64) {
    // 求出网格单元 (i,j) 的顶点坐标 (x,y)
    x := xyrange * (float64(i)/cells - 0.5)
    y := xyrange * (float64(j)/cells - 0.5)

    // 计算曲面高度 z
    z := f(x, y)

    // 将 (x,y,z) 等角投射到二维 SVG 绘图平面上, 坐标是 (sx,sy)
    sx := width/2 + (x-y)*cos30*xyscale
    sy := height/2 + (x+y)*sin30*xyscale - z*zscale
    return sx, sy
}

func f(x, y float64) float64 {
    r := math.Hypot(x, y) // 到 (0,0) 的距离
    return math.Sin(r) / r
}

```

注意, `corner` 函数返回两个值, 构成网格单元其中一角的坐标。

理解这段程序只需基本的几何知识, 但略过也无妨, 因为本例旨在说明浮点运算。这段程序本质上是三套不同坐标系的相互映射, 见图 3-2。首先是个包含  $100 \times 100$  个单元的二维网格, 每个网格单元用整数坐标  $(i, j)$  标记, 从最远处靠后的角落  $(0, 0)$  开始。我们从后向前绘制, 因而后方的多边形可能被前方的遮住。

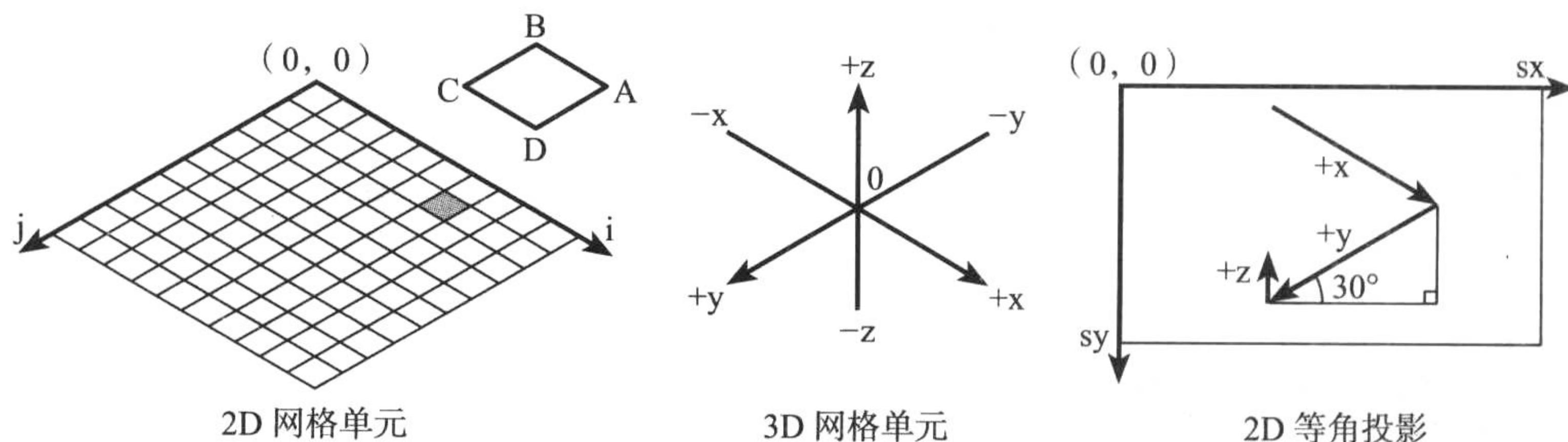


图 3-2 三套不同坐标系

第二个坐标系内, 网格由三维浮点数  $(x, y, z)$  决定, 其中  $x$  和  $y$  由  $i$  和  $j$  的线性函数决定, 经过坐标转换, 原点处于中央, 并且坐标系按照 `xyrange` 进行缩放。高度值  $z$  由曲面函数  $f(x, y)$  决定。

第三个坐标系是二维成像绘图平面 (image canvas), 原点在左上角。这个平面中点的坐标记作  $(sx, sy)$ 。我们用等角投影 (isometric projection) 将三维坐标点  $(x, y, z)$  映射到二维绘图平面上。若一个点的  $x$  值越大,  $y$  值越小, 则其在绘图平面上看起来就越接近右方。而若一个点的  $x$  值或  $y$  值越大, 且  $z$  值越小, 则其在绘图平面上看起来就越接近下方。纵向

( $x$ ) 与横向 ( $y$ ) 的缩放系数是由  $30^\circ$  角的正弦值和余弦值推导而得。 $z$  方向的缩放系数为 0.4，是个随意选定的参数值。

二维网格中的单元由 `main` 函数处理，它算出多边形  $ABCD$  在绘图平面上四个顶点的坐标，其中  $B$  对应  $(i, j)$ ， $A$ 、 $C$ 、 $D$  则为其他三个顶点，然后再输出一条 SVG 指令将其绘出。

**练习 3.1：**假如函数 `f` 返回一个 `float64` 型的无穷大值，就会导致 SVG 文件含有无效的 `<polygon>` 元素（尽管很多 SVG 绘图程序对此处理得当）。修改本程序以避免无效多边形。

**练习 3.2：**用 `math` 包的其他函数试验可视化效果。你能否生成各种曲面，分别呈鸡蛋盒状、雪坡状或马鞍状？

**练习 3.3：**按高度给每个多边形上色，使得峰顶呈红色 (#ff0000)，谷底呈蓝色 (#0000ff)。

**练习 3.4：**仿照 1.7 节的示例 Lissajous 的方法，构建一个 Web 服务器，计算并生成曲面，同时将 SVG 数据写入客户端。服务器必须如下设置 Content-Type 报头。

```
w.Header().Set("Content-Type", "image/svg+xml")
```

（在 Lissajous 示例中，这一步并不强制要求，因为该服务器使用标准的启发式规则，根据响应内容最前面的 512 字节来识别常见的格式（如 PNG），并生成正确的 HTTP 报头。）允许客户端通过 HTTP 请求参数的形式指定各种值，如高度、宽度和颜色。

### 3.3 复数

Go 具备两种大小的复数 `complex64` 和 `complex128`，二者分别由 `float32` 和 `float64` 构成。内置的 `complex` 函数根据给定的实部和虚部创建复数，而内置的 `real` 函数和 `imag` 函数则分别提取复数的实部和虚部：

```
var x complex128 = complex(1, 2) // 1+2i
var y complex128 = complex(3, 4) // 3+4i
fmt.Println(x*y)                // "(-5+10i)"
fmt.Println(real(x*y))          // "-5"
fmt.Println(imag(x*y))          // "10"
```

源码中，如果在浮点数或十进制整数后面紧接着写字母 `i`，如 `3.141592i` 或 `2i`，它就变成一个虚数，表示一个实部为 0 的复数：

```
fmt.Println(1i * 1i) // "(-1+0i)", i2 = -1
```

根据常量运算规则，复数常量可以和其他常量相加（整型或浮点型，实数和虚数皆可），这让我们可以自然地写出复数，如 `1+2i`，或等价地，`2i+1`。前面 `x` 和 `y` 的声明可以简写为：

```
x := 1 + 2i
y := 3 + 4i
```

可以用 `==` 或 `!=` 判断复数是否等值。若两个复数的实部和虚部都相等，则它们相等。`math/cmplx` 包提供了复数运算所需的库函数，例如复数的平方根函数和复数的幂函数。

```
fmt.Println(cmplx.Sqrt(-1)) // "(0+1i)"
```

下面的程序用 `complex128` 运算生成一个 Mandelbrot 集。

```
gopl.io/ch3/mandelbrot
// mandelbrot 函数生成一个 PNG 格式的 Mandelbrot 分形图
package main
```

```

import (
    "image"
    "image/color"
    "image/png"
    "math/cmplx"
    "os"
)
func main() {
    const (
        xmin, ymin, xmax, ymax = -2, -2, +2, +2
        width, height           = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0, 0, width, height))
    for py := 0; py < height; py++ {
        y := float64(py)/height*(ymax-ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px)/width*(xmax-xmin) + xmin
            z := complex(x, y)
            // 点(px, py) 表示复数值z
            img.Set(px, py, mandelbrot(z))
        }
    }
    png.Encode(os.Stdout, img) // 注意：忽略错误
}

func mandelbrot(z complex128) color.Color {
    const iterations = 200
    const contrast = 15

    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v*v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast*n}
        }
    }
    return color.Black
}

```

两个嵌套循环在  $1024 \times 1024$  的灰度图上逐行扫描每个点，这个图表示复平面上  $-2 \sim +2$  的区域，每个点都对应一个复数。该程序针对各个点反复迭代计算其平方与自身的和，判断其最终能否“超出”半径为 2 的圆。若然，就根据超出圆边界所需的迭代次数设定该点的灰度。否则，该点属于 Mandelbrot 集，颜色留黑。最后，程序将标准输出的数据写入 PNG 图，得到一个标志性的分形，见图 3-3。

**练习 3.5：**用 `image.NewRGBA` 函数和 `color.RGBA` 类型或 `color.YCbCr` 类型实现一个 Mandelbrot 集的全彩图。

**练习 3.6：**超采样 (supersampling) 通过对几个临近像素颜色值取样并取均值，是一种减少锯齿化的方法。最简单的做法是将每个像素分成 4 个“子像素”。给出实现方式。

**练习 3.7：**另一种简单的分形是运用牛顿法求某个函数的复数解，比如  $z^4-1=0$ 。以平面上各点作为牛顿法的起始，根据逼近其中一个根（共有 4 个根）所需的迭代次数对该点设定灰度。再根据求得的根对每个点进行全彩上色。

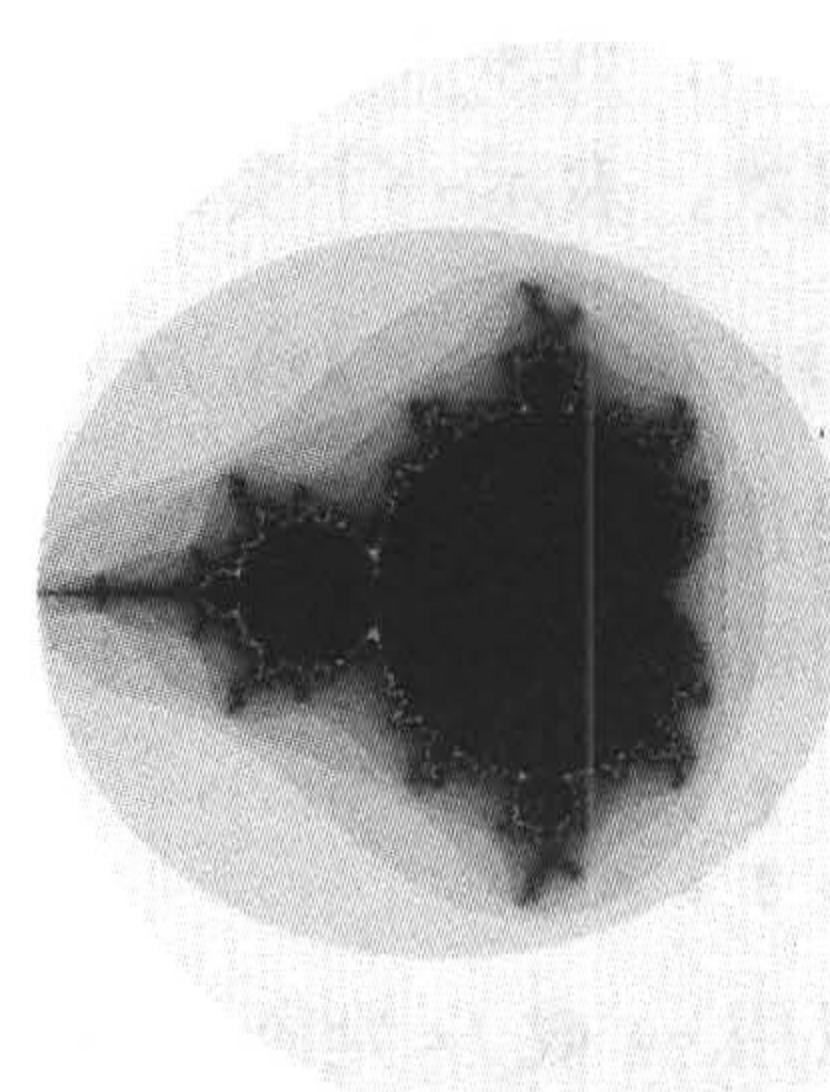


图 3-3 Mandelbrot 集

**练习 3.8：**生成高度放大的分形需要极高的数学精度。分别用以下 4 种类型（`complex64`、`complex128`、`big.Float` 和 `big.Rat`）表示数据实现同一个分形（后面两种类型由 `math/big` 包给出。`big.Float` 类型随意选用 `float32`/`float64` 浮点数，但精度有限；`big.Rat` 类型使用无限精度的有理数。）它们在计算性能和内存消耗上相比如何？放大到什么程度，渲染的失真变得可见？

**练习 3.9：**编写一个 Web 服务器，它生成分形并将图像数据写入客户端。要让客户端得以通过 HTTP 请求的参数指定  $x$ 、 $y$  值和放大系数。

## 3.4 布尔值

`bool` 型的值或布尔值（`boolean`）只有两种可能：真（`true`）和假（`false`）。`if` 和 `for` 语句里的条件就是布尔值，比较操作符（如 `==` 和 `<`）也能得出布尔值结果。一元操作符（`!`）表示逻辑取反，因此 `!true` 就是 `false`，或者可以说 `(!true==false)==true`。比如，考虑到代码风格，布尔表达式 `x==true` 相对冗长，我们总是简化为 `x`。

布尔值可以由运算符 `&&(AND)` 以及 `||(OR)` 组合运算，这可能引起短路行为：如果运算符左边的操作数已经能直接确定总体结果，则右边的操作数不会计算在内，所以下面的表达式是安全的：

```
s != "" && s[0] == 'x'
```

其中，如果作用于空字符串，`s[0]` 会触发宕机异常。

因为 `&&` 较 `||` 优先级更高（助记窍门：`&&` 表示逻辑乘法，`||` 表示逻辑加法），所以如下形式的条件无须加圆括号：

```
if 'a' <= c && c <= 'z' ||
    'A' <= c && c <= 'Z' ||
    '0' <= c && c <= '9' {
    // ...ASCII 字母或数字
}
```

布尔值无法隐式转换成数值（如 0 或 1），反之也不行。如下状况下就有必要使用显式 `if`：

```
i := 0
if b {
    i = 1
}
```

假如转换操作常常用到，就值得专门为此写个函数：

```
// 如果 b 为真，btoi 返回 1；如果 b 为假，则返回 0
func btoi(b bool) int {
    if b {
        return 1
    }
    return 0
}
```

反向转换操作过于简单，无须专门撰写函数，但为了与 `btoi` 对应，这里还是给出其代码：

```
// itob 报告 i 是否为非零值
func itob(i int) bool { return i != 0 }
```

## 3.5 字符串

字符串是不可变的字节序列，它可以包含任意数据，包括 0 值字节，但主要是人类可读

的文本。习惯上，文本字符串被解读成按 UTF-8 编码的 Unicode 码点（文字符号）序列，稍后将细究相关内容。

内置的 `len` 函数返回字符串的字节数（并非文字符号的数目），下标访问操作 `s[i]` 则取得第  $i$  个字符，其中  $0 \leq i < \text{len}(s)$ 。

```
s := "hello, world"
fmt.Println(len(s))      // "12"
fmt.Println(s[0], s[7]) // "104 119" ('h' and 'w')
```

试图访问许可范围以外的字节会触发宕机异常：

```
c := s[len(s)] // 岩机：下标越界
```

字符串的第  $i$  个字节不一定就是第  $i$  个字符，因为非 ASCII 字符的 UTF-8 码点需要两个字节或多个字节。稍后将讨论如何使用字符。

子串生成操作 `s[i:j]` 产生一个新字符串，内容取自原字符串的字节，下标从  $i$ （含边界值）开始，直到  $j$ （不含边界值）。结果的大小是  $j-i$  个字节。

```
fmt.Println(s[0:5]) // "hello"
```

再次强调，若下标越界，或者  $j$  的值小于  $i$ ，将触发宕机异常。

操作数  $i$  与  $j$  的默认值分别是  $0$ （字符串起始位置）和  $\text{len}(s)$ （字符串终止位置），若省略  $i$  或  $j$ ，或两者，则取默认值。

```
fmt.Println(s[:5]) // "hello"
fmt.Println(s[7:]) // "world"
fmt.Println(s[:]) // "hello, world"
```

加号 (+) 运算符连接两个字符串而生成一个新字符串：

```
fmt.Println("goodbye" + s[5:]) // "goodbye, world"
```

字符串可以通过比较运算符做比较，如 `==` 和 `<`；比较运算按字节进行，结果服从本身的字典排序。

尽管肯定可以将新值赋予字符串变量，但是字符串值无法改变：字符串值本身所包含的字节序列永不可变。要在一个字符串后面添加另一个字符串，可以这样编写代码：

```
s := "left foot"
t := s
s += ", right foot"
```

这并不改变 `s` 原有的字符串值，只是将 `+=` 语句生成的新字符串赋予 `s`。同时，`t` 仍然持有旧的字符串值。

```
fmt.Println(s) // "left foot, right foot"
fmt.Println(t) // "left foot"
```

因为字符串不可改变，所以字符串内部的数据不允许修改：

```
s[0] = 'L' // 编译错误：s[0] 无法赋值
```

不可变意味着两个字符串能安全地共用同一段底层内存，使得复制任何长度字符串的开销都低廉。类似地，字符串 `s` 及其子串（如 `s[7:]`）可以安全地共用数据，因此子串生成操作的开销低廉。这两种情况下都没有分配新内存。图 3-4 展示了一个字符串及其两个子字符串的内存布局，它们共用底层字节数组。

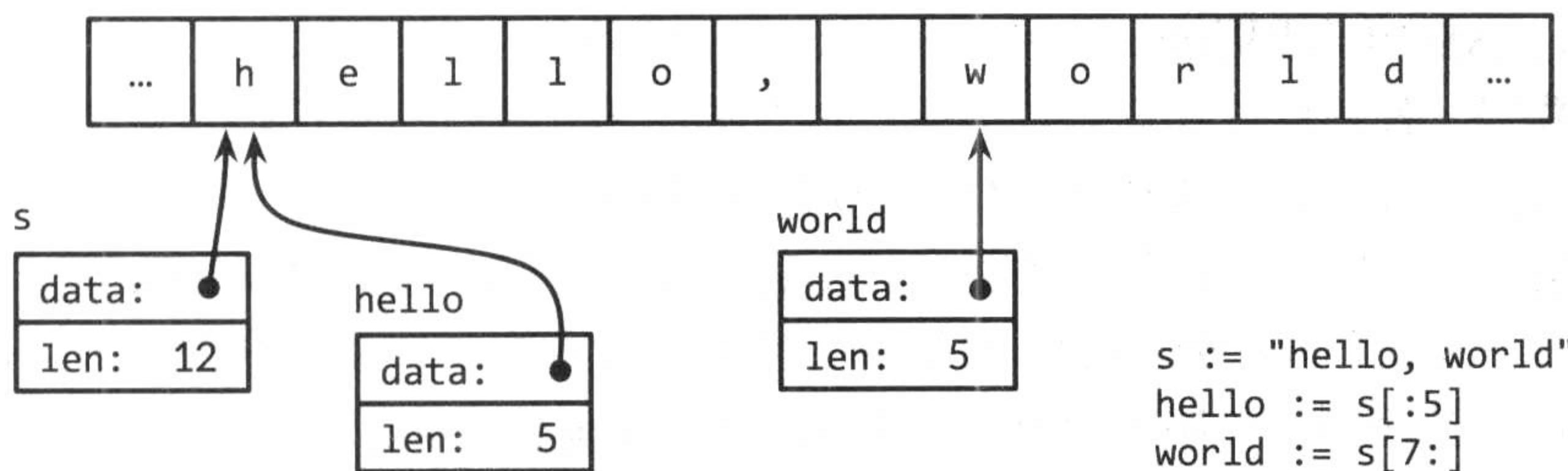


图 3-4 字符串 "hello,world" 及其两个子字符串

### 3.5.1 字符串字面量

字符串的值可以直接写成字符串字面量 (string literal)，形式上就是带双引号的字节序列：

"Hello, 世界"

因为 Go 的源文件总是按 UTF-8 编码，并且习惯上 Go 的字符串会按 UTF-8 解读，所以在源码中我们可以将 Unicode 码点写入字符串字面量。

在带双引号的字符串字面量中，转义序列以反斜杠 (\) 开始，可以将任意值的字节插入字符串中。下面是一组转义符，表示 ASCII 控制码，如换行符、回车符和制表符。

\a	“警告”或响铃
\b	退格符
\f	换页符
\n	换行符（指直接跳到下一行的同一位置）
\r	回车符（指返回行首）
\t	制表符
\v	垂直制表符
\'	单引号（仅用于文字字符字面量 '\''）
\\"	双引号（仅用于 “...” 字面量内部）
\\\	反斜杠

源码中的字符串也可以包含十六进制或八进制的任意字节。十六进制的转义字符写成 \xhh 的形式，h 是十六进制数字（大小写皆可），且必须是两位。八进制的转义字符写成 \ooo 的形式，必须使用三位八进制数字（0 ~ 7），且不能超过 \377。这两者都表示单个字节，内容是给定值。后面，我们将看到如何将数值形式的 Unicode 码点嵌入字符串字面量。

原生的字符串字面量的书写形式是 `...`，使用反引号而不是双引号。原生的字符串字面量内，转义序列不起作用；实质内容与字面写法严格一致，包括反斜杠和换行符，因此，在程序源码中，原生的字符串字面量可以展开多行。唯一的特殊处理是回车符会被删除（换行符会保留），使得同一字符串在所有平台上的值都有相同，包括习惯在文本文件存入换行符的系统。

正则表达式往往含有大量反斜杠，可以方便地写成原生的字符串字面量。原生的字面量也适用于 HTML 模板、JSON 字面量、命令行提示信息，以及需要多行文本表达的场景。

```
const GoUsage = `Go is a tool for managing Go source code.
```

Usage:

```
  go command [arguments]
```

...

### 3.5.2 Unicode

从前，事情简单明晰，至少，狭隘地看，软件只须处理一个字符集：ASCII（美国信息

交换标准码)。ASCII(或更确切地说, US-ASCII)码使用7位表示128个“字符”:大小写英文字母、数字、各种标点和设备控制符。这对早期的计算机行业已经足够了,但是让世界上众多使用其他语言的人无法在计算机上使用自己的文书体系。随着互联网的兴起,包含纷繁语言的数据屡见不鲜。到底怎样才能应付语言的繁杂多样,还能兼顾高效率?

答案是 Unicode ([unicode.org](http://unicode.org)),它囊括了世界上所有文书体系的全部字符,还有重音符和其他变音符,控制码(如制表符和回车符),以及许多特有文字,对它们各自赋予一个叫 Unicode 码点的标准数字。在 Go 的术语中,这些字符记号称为文字符号(rune)。

Unicode 第 8 版定义了超过一百种语言文字的 12 万个字符的码点。它们在计算机程序和数据中如何表示?天然适合保存单个文字符号的数据类型就是 int32,为 Go 所采用;正因如此, rune 类型作为 int32 类型的别名。

我们可以将文字符号的序列表示成 int32 值序列,这种表示方式称作 UTF-32 或 UCS-4,每个 Unicode 码点的编码长度相同,都是 32 位。这种编码简单划一,可是因为大多数面向计算机的可读文本是 ASCII 码,每个字符只需 8 位,也就是 1 字节,导致了不必要的存储空间消耗。而使用广泛的字符的数目也少于 65536 个,字符用 16 位就能容纳。我们能作改进吗?

### 3.5.3 UTF-8

UTF-8 以字节为单位对 Unicode 码点作变长编码。UTF-8 是现行的一种 Unicode 标准,由 Go 的两位创建者 Ken Thompson 和 Rob Pike 发明。每个文字符号用 1 ~ 4 个字节表示,ASCII 字符的编码仅占 1 个字节,而其他常用的文书字符的编码只是 2 或 3 个字节。一个文字符号编码的首字节的高位指明了后面还有多少字节。若最高位为 0,则标示着它是 7 位的 ASCII 码,其文字符号的编码仅占 1 字节,这样就与传统的 ASCII 码一致。若最高几位是 110,则文字符号的编码占用 2 个字节,第二个字节以 10 开始。更长的编码以此类推。

0xxxxxxx	文字符号 0 ~ 127	(ASCII)
110xxxxx 10xxxxxx	128 ~ 2047	少于 128 个未使用的值
1110xxxx 10xxxxxx 10xxxxxx	2048 ~ 65535	少于 2048 个未使用的值
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	65536 ~ 0x10ffff	其他未使用的值

变长编码的字符串无法按下标直接访问第  $n$  个字符,然而有失有得,UTF-8 换来许多有用的特性。UTF-8 编码紧凑,兼容 ASCII,并且自同步:最多追溯 3 字节,就能定位一个字符的起始位置。UTF-8 还是前缀编码,因此它能从左向右解码而不产生歧义,也无须超前预读。于是查找文字符号仅须搜索它自身的字节,不必考虑前文内容。文字符号的字典字节顺序与 Unicode 码点顺序一致(Unicode 设计如此),因此按 UTF-8 编码排序自然就是对文字符号排序。UTF-8 编码本身不会嵌入 NUL 字节(0 值),这便于某些程序语言用 NUL 标记字符串结尾。

Go 的源文件总是以 UTF-8 编码,同时,需要用 Go 程序操作的文本字符串也优先采用 UTF-8 编码。`unicode` 包具备针对单个文字符号的函数(例如区分字母和数字,转换大小写),而 `unicode/utf8` 包则提供了按 UTF-8 编码和解码文字符号的函数。

许多 Unicode 字符难以直接从键盘输入;有的看起来十分相似几乎无法分辨;有些甚至不可见。Go 语言中,字符串字面量的转义让我们得以用码点的值来指明 Unicode 字符。有两种形式,`\uhhhh` 表示 16 位码点值,`\uhhhhhhhhh` 表示 32 位码点值,其中每个  $h$  代表一个十六进制数字;32 位形式的码点值几乎不需要用到。这两种形式都以 UTF-8 编码表示出给定的码点。因此,下面几个字符串字面量都表示长度为 6 字节的相同串:

```
"世界"
"\xe4\xb8\x96\xe7\x95\x8c"
"\u4e16\u754c"
"\u00004e16\u0000754c"
```

后面三行的转义序列用不同形式表示第一行的字符串，但实质上它们的字符串值都一样。

Unicode 转义符也能用于文字符号。下列字符是等价的：

```
'世' '\u4e16' '\u00004e16'
```

码点值小于 256 的文字符号可以写成单个十六进制数转义的形式，如 'A' 写成 '\x41'；而更高的码点值则必须使用 \u 或 \u 转义。这就导致，'\xe4\xb8\x96' 不是合法的文字符号，虽然这三个字节构成某个有效的 UTF-8 编码码点。

由于 UTF-8 的优良特性，许多字符串操作都无须解码。我们可以直接判断某个字符串是否为另一个的前缀：

```
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[:len(prefix)] == prefix
}
```

或者它是否为另一个字符串的后缀：

```
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

或者它是否为另一个的子字符串：

```
func Contains(s, substr string) bool {
    for i := 0; i < len(s); i++ {
        if HasPrefix(s[i:], substr) {
            return true
        }
    }
    return false
}
```

按 UTF-8 编码的文本的逻辑同样也适用原生字节序列，但其他编码则无法如此。（上面的函数取自 `strings` 包，其实 `Contains` 函数的具体实现使用了散列方法让搜索更高效。）

另一方面，如果我们真的要逐个逐个处理 Unicode 字符，则必须使用其他编码机制。考虑我们第一个例子的字符串（见 3.5.1 节），它包含两个东亚字符。图 3-5 说明了该字符串的内存布局。它含有 13 个字节，而按作 UTF-8 解读，本质是 9 个码点或文字符号的编码：

```
import "unicode/utf8"

s := "Hello, 世界"
fmt.Println(len(s))           // "13"
fmt.Println(utf8.RuneCountInString(s)) // "9"
```

我们需要 UTF-8 解码器来处理这些字符，`unicode/utf8` 包就具备一个：

```
for i := 0; i < len(s); {
    r, size := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%d\t%c\n", i, r)
    i += size
}
```

每次 `DecodeRuneInString` 的调用都返回 `r`（文字符号本身）和一个值（表示 `r` 按 UTF-8 编码所占用的字节数）。这个值用来更新下标 `i`，定位字符串内的下一个文字符号。可是按此方法，我们总是需要使用上例中的循环形式。所幸，Go 的 `range` 循环也适用于字符串，按 UTF-8 隐式解码。图 3-5 也展示了以下循环的输出。注意，对于非 ASCII 文字符号，下标增量大于 1。

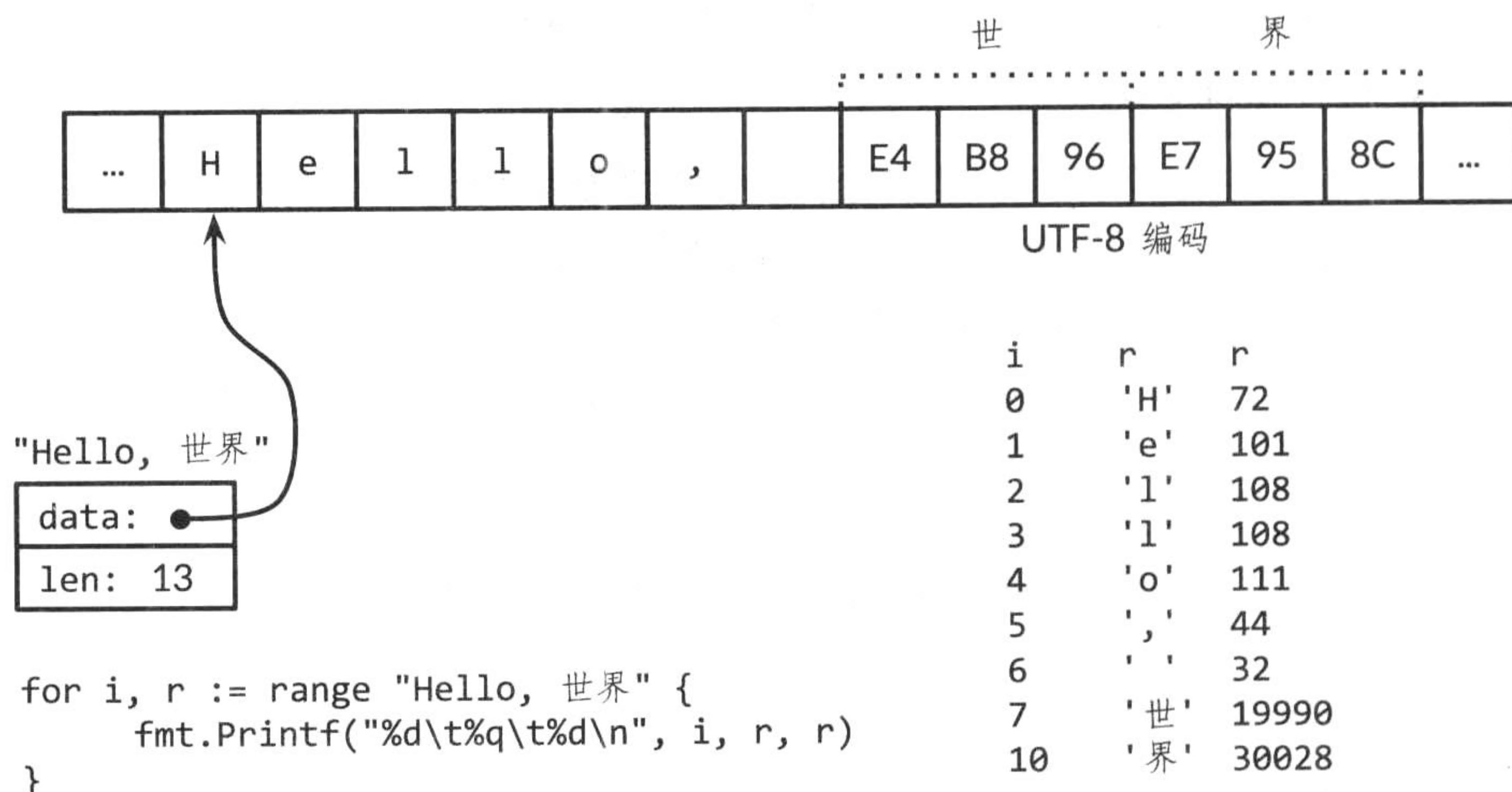


图 3-5 一个按 UTF-8 编码的字符串在 `range` 循环内解码

```

for i, r := range "Hello, 世界" {
    fmt.Printf("%d\t%q\t%d\n", i, r, r)
}

```

我们可用简单的 `range` 循环统计字符串中的文字符号数目，如下所示：

```

n := 0
for _, _ = range s {
    n++
}

```

与其他形式的 `range` 循环一样，可以忽略没用的变量：

```

n := 0
for range s {
    n++
}

```

或者，直截了当地调用 `utf8.RuneCountInString(s)`。

之前提到过，文本字符串作为按 UTF-8 编码的 Unicode 码点序列解读，很大程度是出于习惯，但为了确保使用 `range` 循环能正确处理字符串，则必须要求而不仅仅是按照习惯。如果字符串含有任意二进制数，也就是说，UTF-8 数据出错，而我们对它做 `range` 循环，会发生什么？

每次 UTF-8 解码器读入一个不合理的字节，无论是显式调用 `utf8.DecodeRuneInString`，还是在 `range` 循环内隐式读取，都会产生一个专门的 Unicode 字符 '`\uFFFD`' 替换它，其输出通常是个黑色六角形或类似钻石的形状，里面有个白色问号。如果程序碰到这个文字符号值，通常意味着，生成字符串数据的系统上游部分在处理文本编码方面存在瑕疵。

UTF-8 是一种分外便捷的交互格式，而在程序内部使用文字字符类型可能更加方便，因

为它们大小一致，便于在数组和 slice 中用下标访问。

当 []rune 转换作用于 UTF-8 编码的字符串时，返回该字符串的 Unicode 码点序列：

```
// 日语片假名"程序"
s := "プログラム"
fmt.Printf("%x\n", s) // "e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"
r := []rune(s)
fmt.Printf("%x\n", r) // "[30d7 30ed 30b0 30e9 30e0]"
```

(第一个 `Printf` 里的谓词 `%x` (注意，% 和 x 之间有空格) 以十六进制数形式输出，并在每两个数位间插入空格。)

如果把文字符号类型的 slice 转换成一个字符串，它会输出各个文字符号的 UTF-8 编码拼接结果：

```
fmt.Println(string(r)) // "プログラム"
```

若将一个整数值转换成字符串，其值按文字符号类型解读，并且产生代表该文字符号值的 UTF-8 码：

```
fmt.Println(string(65)) // "A", 而不是 "65"
fmt.Println(string(0x4eac)) // "京"
```

如果文字符号值非法，将被专门的替换字符取代（见前面的 '\uFFFD'）。

```
fmt.Println(string(1234567)) // "�"
```

### 3.5.4 字符串和字节 slice

4 个标准包对字符串操作特别重要：`bytes`、`strings`、`strconv` 和 `unicode`。

`strings` 包提供了许多函数，用于搜索、替换、比较、修整、切分与连接字符串。

`bytes` 包也有类似的函数，用于操作字节 slice([]byte 类型，其某些属性和字符串相同)。由于字符串不可变，因此按增量方式构建字符串会导致多次内存分配和复制。这种情况下，使用 `bytes.Buffer` 类型会更高效，范例见后。

`strconv` 包具备的函数，主要用于转换布尔值、整数、浮点数为与之对应的字符串形式，或者把字符串转换为布尔值、整数、浮点数，另外还有为字符串添加/去除引号的函数。

`unicode` 包备有判别文字符号值特性的函数，如 `IsDigit`、`IsLetter`、`IsUpper` 和 `IsLower`。每个函数以单个文字符号值作为参数，并返回布尔值。若文字符号值是英文字母，转换函数（如 `ToUpper` 和 `ToLower`）将其转换成指定的大小写。上面所有函数都遵循 Unicode 标准对字母数字等的分类原则。`strings` 包也有类似的函数，函数名也是 `ToUpper` 和 `ToLower`，它们对原字符串的每个字符做指定变换，生成并返回一个新字符串。

下例中，`basename` 函数模仿 UNIX shell 中的同名实用程序。只要 `s` 的前缀看起来像是文件系统路径（各部分由斜杠分隔），该版本的 `basename(s)` 就将其移除，貌似文件类型的后缀也被移除：

```
fmt.Println(basename("a/b/c.go")) // "c"
fmt.Println(basename("c.d.go")) // "c.d"
fmt.Println(basename("abc")) // "abc"
```

初版的 `basename` 独自完成全部工作，并不依赖任何库：

gopl.io/ch3 basename1

```
// basename 移除路径部分和 . 后缀
// e.g., a => a, a.go => a, a/b/c.go => c, a/b.c.go => b.c
func basename(s string) string {
    // 将最后一个 '/' 和之前的部分全都舍弃
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '/' {
            s = s[i+1:]
            break
        }
    }
    // 保留最后一个 '.' 之前的所有内容
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '.' {
            s = s[:i]
            break
        }
    }
    return s
}
```

简化版利用库函数 `string.LastIndex`:

gopl.io/ch3 basename2

```
func basename(s string) string {
    slash := strings.LastIndex(s, "/") // 如果没找到 "/", 则 slash 取值 -1
    s = s[slash+1:]
    if dot := strings.LastIndex(s, "."); dot >= 0 {
        s = s[:dot]
    }
    return s
}
```

`path` 包和 `path/filepath` 包提供了一组更加普遍适用的函数，用来操作文件路径等具有层次结构的名字。`path` 包处理以斜杠 '/' 分段的路径字符串，不分平台。它不适合用于处理文件名，却适合其他领域，像 URL 地址的路径部分。相反地，`path/filepath` 包根据宿主平台 (host platform) 的规则处理文件名，例如 POSIX 系统使用 `/foo/bar`，而 Microsoft Windows 系统使用 `c:\foo\bar`。

我们继续看另一个例子，它涉及子字符串操作。任务是接受一个表示整数的字符串，如 "12345"，从右侧开始每三位数字后面就插入一个逗号，形如 "12,345"。这个版本仅对整数有效。对浮点数的处理方式留作练习。

gopl.io/ch3 comma

```
// 函数向表示十进制非负整数的字符串中插入逗号
func comma(s string) string {
    n := len(s)
    if n <= 3 {
        return s
    }
    return comma(s[:n-3]) + "," + s[n-3:]
}
```

`comma` 函数的参数是一个字符串。若字符串长度小于等于 3，则不插入逗号。否则，`comma` 以仅包含字符串最后三个字符的子字符串作为参数，递归调用自己，最后在递归调用的结果后面添加一个逗号和最后三个字符。

若字符串包含一个字节数组，创建后它就无法改变。相反地，字节 slice 的元素允许随

意修改。

字符串可以和字节 slice 相互转换：

```
s := "abc"
b := []byte(s)
s2 := string(b)
```

概念上，`[]byte(s)` 转换操作会分配新的字节数组，拷贝填入 `s` 含有的字节，并生成一个 slice 引用，指向整个数组。具备优化功能的编译器在某些情况下可能会避免分配内存和复制内容，但一般而言，复制有必要确保 `s` 的字节维持不变（即使 `b` 的字节在转换后发生改变）。反之，用 `string(b)` 将字节 slice 转换成字符串也会产生一份副本，保证 `s2` 也不可变。

为了避免转换和不必要的内存分配，`bytes` 包和 `strings` 包都预备了许多对应的实用函数（utility function），它们两两相对应。例如，`strings` 包具备下面 6 个函数：

```
func Contains(s, substr string) bool
func Count(s, sep string) int
func Fields(s string) []string
func HasPrefix(s, prefix string) bool
func Index(s, sep string) int
func Join(a []string, sep string) string
```

`bytes` 包里面的对应函数为：

```
func Contains(b, subslice []byte) bool
func Count(s, sep []byte) int
func Fields(s []byte) [][]byte
func HasPrefix(s, prefix []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte
```

唯一的不同是，操作对象由字符串变为字节 slice。

`bytes` 包为高效处理字节 slice 提供了 `Buffer` 类型。`Buffer` 起初为空，其大小随着各种类型数据的写入而增长，如 `string`、`byte` 和 `[]byte`。如下例所示，`bytes.Buffer` 变量无须初始化，原因是零值本来就有效：

```
gopl.io/ch3/printints
// intsToString 与 fmt.Sprint(values) 类似，但插入了逗号
func intsToString(values []int) string {
    var buf bytes.Buffer
    buf.WriteByte('[')
    for i, v := range values {
        if i > 0 {
            buf.WriteString(", ")
        }
        fmt.Fprintf(&buf, "%d", v)
    }
    buf.WriteByte(']')
    return buf.String()
}

func main() {
    fmt.Println(intsToString([]int{1, 2, 3})) // "[1, 2, 3]"
}
```

若要在 `bytes.Buffer` 变量后面添加任意文字符号的 UTF-8 编码，最好使用 `bytes.Buffer` 的 `WriteRune` 方法，而追加 ASCII 字符，如 '`[`' 和 '`]`'，则使用 `WriteByte` 亦可。

`bytes.Buffer` 类型用途极广，在第 7 章讨论接口的时候，假若 I/O 函数需要一个字节接

收器 (`io.Writer`) 或字节发生器 (`io.Reader`)，我们将看到能如何用其来代替文件，其中接收器的作用就如上例中的 `Fprintf` 一样。

**练习 3.10：**编写一个非递归的 `comma` 函数，运用 `bytes.Buffer`，而不是简单的字符串拼接。

**练习 3.11：**增强 `comma` 函数的功能，让其正确处理浮点数，以及带有可选正负号的数字。

**练习 3.12：**编写一个函数判断两个字符串是否同文异构，也就是，它们都含有相同的字符但排列顺序不同。

### 3.5.5 字符串和数字的相互转换

除了字符串、文字符号和字节之间的转换，我们常常也需要相互转换数值及其字符串表示形式。这由 `strconv` 包的函数完成。

要将整数转换成字符串，一种选择是使用 `fmt.Sprintf`，另一种做法是用函数 `strconv.Itoa` (“integer to ASCII”):

```
x := 123
y := fmt.Sprintf("%d", x)
fmt.Println(y, strconv.Itoa(x)) // "123 123"
```

`FormatInt` 和 `FormatUint` 可以按不同的进位制格式化数字：

```
fmt.Println(strconv.FormatInt(int64(x), 2)) // "1111011"
```

`fmt.Printf` 里的谓词 `%b`、`%d`、`%o` 和 `%x` 往往比 `Format` 函数方便，若要包含数字以外的附加信息，它就尤其有用：

```
s := fmt.Sprintf("x=%b", x) // "x=1111011"
```

`strconv` 包内的 `Atoi` 函数或 `ParseInt` 函数用于解释表示整数的字符串，而 `ParseUint` 用于无符号整数：

```
x, err := strconv.Atoi("123")           // x 是整型
y, err := strconv.ParseInt("123", 10, 64) // 十进制，最长为 64 位
```

`ParseInt` 的第三个参数指定结果必须匹配何种大小的整型；例如，16 表示 `int16`，而 0 作为特殊值表示 `int`。任何情况下，结果 `y` 的类型总是 `int64`，可将他另外转换成较小的类型。

有时候，单行输入由字符串和数字依次混合构成，需要用 `fmt.Scanf` 解释，可惜 `fmt.Scanf` 也许不够灵活，处理不完整或不规则输入时尤甚。

## 3.6 常量

常量是一种表达式，其可以保证在编译阶段就计算出表达式的值，并不需要等到运行时，从而使编译器得以知晓其值。所有常量本质上都属于基本类型：布尔型、字符串或数字。

常量的声明定义了具名的值，它看起来在语法上与变量类似，但该值恒定，这防止了程序运行过程中的意外（或恶意）修改。例如，要表示数学常量，像圆周率，在 Go 程序中用常量比变量更适合，因其值恒定不变：

```
const pi = 3.14159 // 近似数；math.Pi 是更精准的近似
```

与变量类似，同一个声明可以定义一系列常量，这适用于一组相关的值：

```
const (
    e = 2.71828182845904523536028747135266249775724709369995957496696763
    pi = 3.14159265358979323846264338327950288419716939937510582097494459
)
```

许多针对常量的计算完全可以在编译时就完成，以减免运行时的工作量并让其他编译器优化得以实现。某些错误通常要在运行时才能检测到，但如果操作数是常量，编译时就会报错，例如整数除以 0，字符串下标越界，以及任何产生无限大值的浮点数运算。

对于常量操作数，所有数学运算、逻辑运算和比较运算的结果依然是常量，常量的类型转换结果和某些内置函数的返回值，例如 `len`、`cap`、`real`、`imag`、`complex` 和 `unsafe.Sizeof`，同样是常量。

因为编译器知晓其值，常量表达式可以出现在涉及类型的声明中，具体而言就是数组类型的长度：

```
const IPv4Len = 4

// parseIPv4 函数解释一个 IPv4 地址 (d.d.d.d)
func parseIPv4(s string) IP {
    var p [IPv4Len]byte
    // ...
}
```

常量声明可以同时指定类型和值，如果没有显式指定类型，则类型根据右边的表达式推断。下例中，`time.Duration` 是一种具名类型，其基本类型是 `int64`，`time.Minute` 也是基于 `int64` 的常量。下面声明的两个常量都属于 `time.Duration` 类型，通过 `%T` 展示：

```
const noDelay time.Duration = 0
const timeout = 5 * time.Minute
fmt.Printf("%T %[1]v\n", noDelay)      // "time.Duration 0"
fmt.Printf("%T %[1]v\n", timeout)       // "time.Duration 5m0s"
fmt.Printf("%T %[1]v\n", time.Minute) // "time.Duration 1m0s"
```

若同时声明一组常量，除了第一项之外，其他项在等号右侧的表达式都可以省略，这意味着会复用前面一项的表达式及其类型。例如：

```
const (
    a = 1
    b
    c = 2
    d
)

fmt.Println(a, b, c, d) // "1 1 2 2"
```

如果复用右侧表达式导致计算结果总是相同，这就并不太实用。假若该结果可变该怎么办呢？我们来看看 `iota`。

### 3.6.1 常量生成器 `iota`

常量的声明可以使用常量生成器 `iota`，它创建一系列相关值，而不是逐个值显式写出。常量声明中，`iota` 从 0 开始取值，逐项加 1。

下例取自 `time` 包，它定义了 `Weekday` 的具名类型，并声明每周的 7 天为该类型的常量，从 `Sunday` 开始，其值为 0。这种类型通常称为枚举型（enumeration，或缩写成 `enum`）。