

```

type Weekday int

const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)

```

上面的声明中，Sunday 的值为 0，Monday 的值为 1，以此类推。

更复杂的表达式也可使用 iota，借用 net 包的代码举例如下，无符号整数最低 5 位数中的每一个都逐一命名，并解释为布尔值。

```

type Flags uint

const (
    FlagUp Flags = 1 << iota // 向上
    FlagBroadcast           // 支持广播访问
    FlagLoopback            // 是环回接口
    FlagPointToPoint         // 属于点对点链路
    FlagMulticast           // 支持多路广播访问
)

```

随着 iota 递增，每个常量都按 $1 \ll \text{iota}$ 赋值，这等价于 2 的连续次幂，它们分别与单个位对应。若某些函数要针对相应的位执行判定、设置或清除操作，就会用到这些常量。

```

gopl.io/ch3/netflag

func IsUp(v Flags) bool      { return v&FlagUp == FlagUp }
func TurnDown(v *Flags)     { *v &^= FlagUp }
func SetBroadcast(v *Flags) { *v |= FlagBroadcast }
func IsCast(v Flags) bool   { return v&(FlagBroadcast|FlagMulticast) != 0 }

func main() {
    var v Flags = FlagMulticast | FlagUp
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10001 true"
    TurnDown(&v)
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10000 false"
    SetBroadcast(&v)
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10010 false"
    fmt.Printf("%b %t\n", v, IsCast(v)) // "10010 true"
}

```

下例更复杂，声明的常量表示 1024 的幂。

```

const (
    _ = 1 << (10 * iota)
    KiB // 1024
    MiB // 1048576
    GiB // 1073741824
    TiB // 1099511627776          (超过  $1 \ll 32$ )
    PiB // 1125899906842624
    EiB // 1152921504606846976
    ZiB // 1180591620717411303424 (超过  $1 \ll 64$ )
    YiB // 1208925819614629174706176
)

```

然而，iota 机制存在局限。比如，因为不存在指数运算符，所以无从生成更为人熟知的 1000 的幂（KB、MB 等）。

练习 3.13：用尽可能简洁的方法声明从 KB、MB 直到 YB 的常量。

3.6.2 无类型常量

Go 的常量自有特别之处。虽然常量可以是任何基本数据类型，如 `int` 或 `float64`，也包括具名的基本类型（如 `time.Duration`），但是许多常量并不从属某一具体类型。编译器将这些从属类型待定的常量表示成某些值，这些值比基本类型的数字精度更高，且算术精度高于原生的机器精度。可以认为它们的精度至少达到 256 位。从属类型待定的常量共有 6 种，分别是无类型布尔、无类型整数、无类型文字符号、无类型浮点数、无类型复数、无类型字符串。

借助推迟确定从属类型，无类型常量不仅能暂时维持更高的精度，与类型已确定的常量相比，它们还能写进更多表达式而无需转换类型。比如，上例中 `ZiB` 和 `YiB` 的值过大，用哪种整型都无法存储，但它们都是合法常量并且可以用在下面的表达式中：

```
fmt.Println(YiB/ZiB) // "1024"
```

再例如，浮点型常量 `math.Pi` 可用于任何需要浮点值或复数的地方：

```
var x float32 = math.Pi
var y float64 = math.Pi
var z complex128 = math.Pi
```

若常量 `math.Pi` 一开始就确定从属于某具体类型，如 `float64`，就会导致结果的精度下降。另外，假使最终需要 `float32` 值或 `complex128` 值，则可能需要转换类型：

```
const Pi64 float64 = math.Pi

var x float32 = float32(Pi64)
var y float64 = Pi64
var z complex128 = complex128(Pi64)
```

字面量的类型由语法决定。`0`、`0.0`、`0i` 和 `'\u0000'` 全都表示相同的常量值，但类型相异，分别是：无类型整数、无类型浮点数、无类型复数和无类型文字符号。类似地，`true` 和 `false` 是无类型布尔值，而字符串字面量则是无类型字符串。

根据除法运算中操作数的类型，除法运算的结果可能是整型或浮点型。所以，常量除法表达式中，操作数选择不同的字面写法会影响结果：

```
var f float64 = 212
fmt.Println((f - 32) * 5 / 9)      // "100"; (f - 32) * 5 的结果是 float64 型
fmt.Println(5 / 9 * (f - 32))     // "0"; 5/9 的结果是无类型整数, 0
fmt.Println(5.0 / 9.0 * (f - 32)) // "100"; 5.0/9.0 的结果是无类型浮点数
```

只有常量才可以是无类型的。若将无类型常量声明为变量（如下面的第一条语句所示），或在类型明确的变量赋值的右方出现无类型常量（如下面的其他三条语句所示），则常量会被隐式转换成该变量的类型。

```
var f float64 = 3 + 0i // 无类型复数 -> float64
f = 2                  // 无类型整数 -> float64
f = 1e123              // 无类型浮点数 -> float64
f = 'a'                // 无类型 -> float64
```

上述语句与下面的语句等价：

```
var f float64 = float64(3 + 0i)
f = float64(2)
f = float64(1e123)
f = float64('a')
```

不论隐式或显式，常量从一种类型转换成另一种，都要求目标类型能够表示原值。实数和复数允许舍入取整：

```
const (
    deadbeef = 0xdeadbeef // 无类型整数，值为 3735928559
    a = uint32(deadbeef) // uint32，值为 3735928559
    b = float32(deadbeef) // float32，值为 3735928576 (向上取整)
    c = float64(deadbeef) // float64，值为 3735928559 (精确值)
    d = int32(deadbeef) // 编译错误：溢出，int32 无法容纳常量值
    e = float64(1e309) // 编译错误：溢出，float64 无法容纳常量值
    f = uint(-1) // 编译错误：溢出，uint 无法容纳常量值
)
```

变量声明（包括短变量声明）中，假如没有显式指定类型，无类型常量会隐式转换成该变量的默认类型，如下例所示：

```
i := 0      // 无类型整数；隐式 int(0)
r := '\000' // 无类型文字字符；隐式 rune('\000')
f := 0.0    // 无类型浮点数；隐式 float64(0.0)
c := 0i     // 无类型整数；隐式 complex128(0i)
```

注意各类型的不对称性：无类型整数可以转换成 `int`，其大小不确定，但无类型浮点数和无类型复数被转换成大小明确的 `float64` 和 `complex128`。Go 语言中，只有大小不明确的 `int` 类型，却不存在大小不确定的 `float` 类型和 `complex` 类型，原因是，如果浮点型数据的大小不明，就很难写出正确的数值算法。

要将变量转换成不同的类型，我们必须将无类型常量显式转换为期望的类型，或在声明变量时指明想要的类型，如下例所示：

```
var i = int8(0)
var i int8 = 0
```

在将无类型常量转换为接口值时（见第 7 章），这些默认类型就分外重要，因为它们决定了接口值的动态类型。

```
fmt.Printf("%T\n", 0)      // "int"
fmt.Printf("%T\n", 0.0)     // "float64"
fmt.Printf("%T\n", 0i)      // "complex128"
fmt.Printf("%T\n", '\000') // "int32" (rune)
```

至此，我们已经概述了 Go 的基本数据类型。下一步就是要说明如何将它们构建成为更大的聚合体，如数组和结构体，更进一步组成数据结构以解决实际的编程问题。第 4 章以此为主题。

复合数据类型

第 3 章讨论了 Go 程序中的基础数据类型；它们就像宇宙中的原子一样。本章介绍复合数据类型，复合数据类型是由基本数据类型以各种方式组合而构成的，就像分子由原子构成一样。我们将重点讲解四种复合数据类型，分别是数组、slice、map 和结构体。另外本章末尾将演示如何将使用这些数据类型构成的结构化数据编码为 JSON 数据，从 JSON 数据转换为结构化数据，以及从模板生成 HTML 页面。

数组和结构体都是聚合类型，它们的值由内存中的一组变量构成。数组的元素具有相同的类型，而结构体中的元素数据类型则可以不同。数组和结构体的长度都是固定的。反之，slice 和 map 都是动态数据结构，它们的长度在元素添加到结构中时可以动态增长。

4.1 数组

数组是具有固定长度且拥有零个或者多个相同数据类型元素的序列。由于数组的长度固定，所以在 Go 里面很少直接使用。slice 的长度可以增长和缩短，在很多场合下使用得更多。然而，在理解 slice 之前，我们必须先理解数组。

数组中的每个元素是通过索引来访问的，索引从 0 到数组长度减 1。Go 内置的函数 `len` 可以返回数组中的元素个数。

```
var a [3]int          // 3 个整数的数组
fmt.Println(a[0])      // 输出数组的第一个元素
fmt.Println(a[len(a)-1]) // 输出数组的最后一个元素，即 [2]
// 输出索引和元素
for i, v := range a {
    fmt.Printf("%d %d\n", i, v)
}

// 仅输出元素
for _, v := range a {
    fmt.Printf("%d\n", v)
}
```

默认情况下，一个新数组中的元素初始值为元素类型的零值，对于数字来说，就是 0。也可以使用数组字面量根据一组值来初始化一个数组。

```
var q [3]int = [3]int{1, 2, 3}
var r [3]int = [3]int{1, 2}
fmt.Println(r[2]) // "0"
```

在数组字面量中，如果省略号 “...” 出现在数组长度的位置，那么数组的长度由初始化数组的元素个数决定。以上数组 `q` 的定义可以简化为：

```
q := [...]int{1, 2, 3}
fmt.Printf("%T\n", q) // "[3]int"
```

数组的长度是数组类型的一部分，所以 `[3]int` 和 `[4]int` 是两种不同的数组类型。数组的长度必须是常量表达式，也就是说，这个表达式的值在程序编译时就可以确定。

```
q := [3]int{1, 2, 3}
q = [4]int{1, 2, 3, 4} // 编译错误：不可以将 [4]int 赋值给 [3]int
```

如我们所见，数组、slice、map 和结构体的字面语法都是相似的。上面的例子是按顺序给出一组值；也可以像这样给出一组值，这一组值同样具有索引和索引对应的值：

```
type Currency int
const (
    USD Currency = iota
    EUR
    GBP
    RMB
)
symbol := [...]string{USD: "$", EUR: "€", GBP: "£", RMB: "¥"}
fmt.Println(RMB, symbol[RMB]) // "3 ¥"
```

在这种情况下，索引可以按照任意顺序出现，并且有的时候还可以省略。和上面一样，没有指定值的索引位置的元素默认被赋予数组元素类型的零值。例如，

```
r := [...]int{99: -1}
```

定义了一个拥有 100 个元素的数组 r，除了最后一个元素值是 -1 外，该数组中的其他元素值都是 0。

如果一个数组的元素类型是可比较的，那么这个数组也是可比较的，这样我们就可以直接使用 == 操作符来比较两个数组，比较的结果是两边元素的值是否完全相同。使用 != 来比较两个数组是否不同。

```
a := [2]int{1, 2}
b := [...]int{1, 2}
c := [2]int{1, 3}
fmt.Println(a == b, a == c, b == c) // "true false false"
d := [3]int{1, 2}
fmt.Println(a == d) // 编译错误：无法比较 [2]int == [3]int
```

举一个更有意义的例子，crypto/sha256 包里面的函数 Sum256 用来为存储在任意字节 slice 中的消息使用 SHA256 加密散列算法生成一个摘要。摘要信息是 256 位，即 [32]byte。如果两个摘要信息相同，那么很有可能这两条原始消息就是相同的；如果这两个摘要信息不同，那么这两条原始消息就是不同的。下面的程序输出并比较了 "x" 和 "X" 的 SHA256 散列值：

```
gopl.io/ch4/sha256
import "crypto/sha256"

func main() {
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)
    // Output:
    // 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881
    // 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015
    // false
    // [32]uint8
}
```

这两个原始消息仅有一位 (bit) 之差，但是它们生成的摘要消息有将近一半的位不同。注意，上面的格式化字符串 %x 表示将一个数组或者 slice 里面的字节按照十六进制的方式输出，%t 表示输出一个布尔值，%T 表示输出一个值的类型。

当调用一个函数的时候，每个传入的参数都会创建一个副本，然后赋值给对应的函数变量，所以函数接受的是一个副本，而不是原始的参数。使用这种方式传递大的数组会变得很低效，并且在函数内部对数组的任何修改都仅影响副本，而不是原始数组。这种情况下，Go 把数组和其他的类型都看成值传递。而在其他的语言中，数组是隐式地使用引用传递。

当然，也可以显式地传递一个数组的指针给函数，这样在函数内部对数组的任何修改都会反映到原始数组上面。下面的程序演示如何将一个数组 [32]byte 的元素清零：

```
func zero(ptr *[32]byte) {
    for i := range ptr {
        ptr[i] = 0
    }
}
```

数组字面量 [32]byte{} 可以生成一个拥有 32 个字节元素的数组。数组中每个元素的值都是字节类型的零值，即 0。可以利用这一点来写另一个版本的数组清零程序：

```
func zero(ptr *[32]byte) {
    *ptr = [32]byte{}
}
```

使用数组指针是高效的，同时允许被调函数修改调用方数组中的元素，但是因为数组长度是固定的，所以数组本身是不可变的。例如上面的 zero 函数不能接受一个 [16]byte 这样的数组指针，同样，也无法为数组添加或者删除元素。由于数组的长度不可变的特性，除了在特殊的情况下之外，我们很少使用数组。上面关于 SHA256 的例子中，摘要的结果拥有固定的长度，我们可以使用数组作为函数参数或结果，但是更多的情况下，我们使用 slice。

练习 4.1：编写一个函数，用于统计 SHA256 散列中不同的位数（见 2.6.2 节的 PopCount）。

练习 4.2：编写一个程序，用于在默认情况下输出其标准输入的 SHA256 散列，但也支持一个输出 SHA384 或 SHA512 散列的命令行标记。

4.2 slice

slice 表示一个拥有相同类型元素的可变长度的序列。slice 通常写成 []T，其中元素的类型都是 T；它看上去像没有长度的数组类型。

数组和 slice 是紧密关联的。slice 是一种轻量级的数据结构，可以用来访问数组的部分或者全部的元素，而这个数组称为 slice 的底层数组。slice 有三个属性：指针、长度和容量。指针指向数组的第一个可以从 slice 中访问的元素，这个元素并不一定是数组的第一个元素。长度是指 slice 中的元素个数，它不能超过 slice 的容量。容量的大小通常是从 slice 的起始元素到底层数组的最后一个元素间元素的个数。Go 的内置函数 len 和 cap 用来返回 slice 的长度和容量。

一个底层数组可以对应多个 slice，这些 slice 可以引用数组的任何位置，彼此之间的元素还可以重叠。图 4-1 显示了一个月份名称的字符串数组和两个元素存在重叠的 slice。数组声明是：

```
months := [...]string{1: "January", /* ... */, 12: "December"}
```

所以 January 就是 months[1]，December 是 months[12]。一般来讲，数组中索引 0 的位置存放数组的第一个元素，但是由于月份总是从 1 开始，因此我们可以不设置索引为 0 的元素，这样它的值就是空字符串。

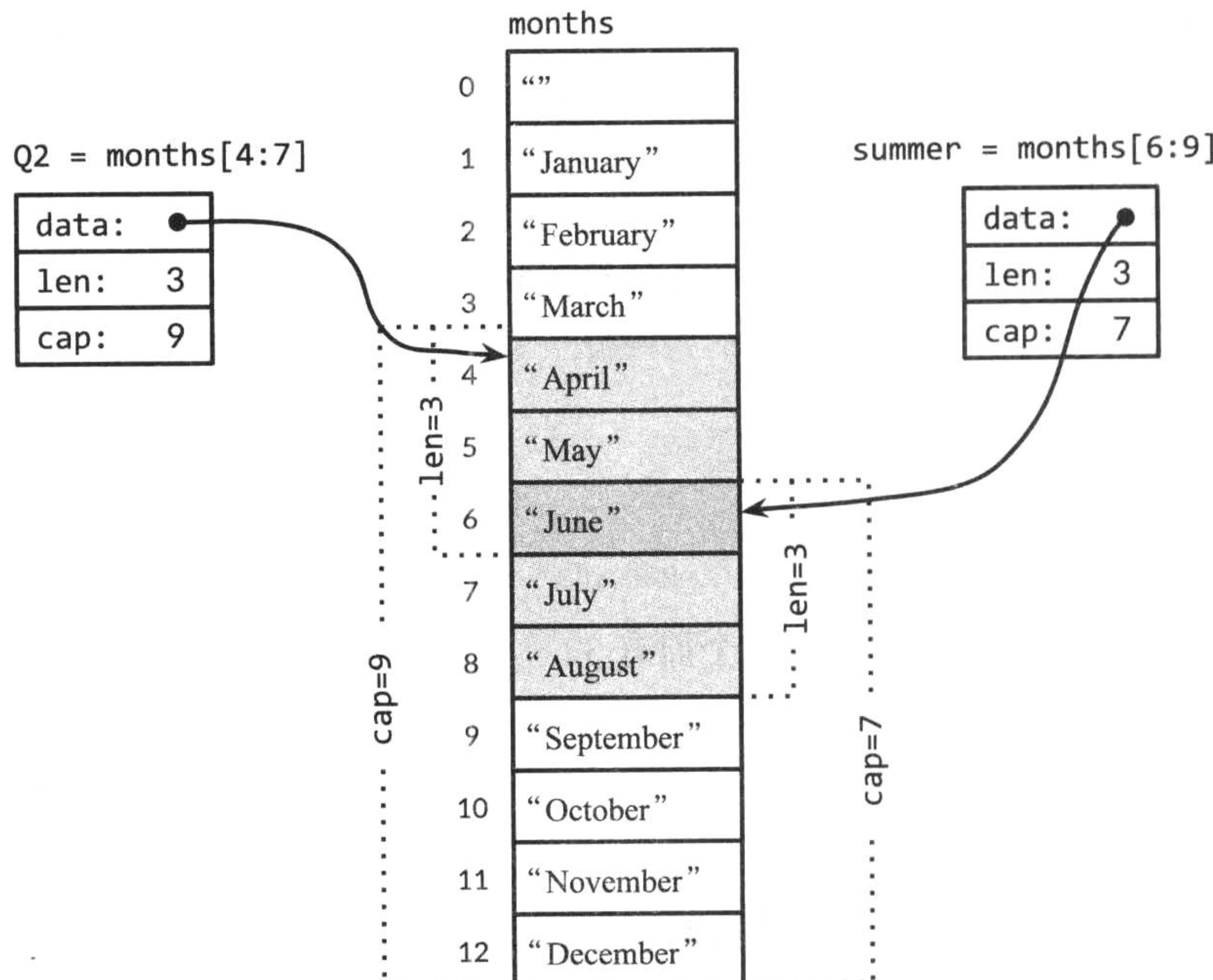


图 4-1 月份名称字符串数组对应的两个元素重叠的 slice

`slice` 操作符 `s[i:j]` (其中 $0 \leq i \leq j \leq \text{cap}(s)$) 创建了一个新的 `slice`, 这个新的 `slice` 引用了序列 `s` 中从 `i` 到 `j-1` 索引位置的所有元素, 这里的 `s` 既可以是数组或者指向数组的指针, 也可以是 `slice`。新 `slice` 的元素个数是 `j-i` 个。如果上面的表达式中省略了 `i`, 那么新 `slice` 的起始索引位置就是 0, 即 `i=0`; 如果省略了 `j`, 那么新 `slice` 的结束索引位置是 `\text{len}(s)-1`, 即 `j=\text{len}(s)`。因此 `slice months[1:13]` 引用了所有的有效月份, 同样的写法可以是 `months[1:]`。`slice months[:]` 引用了整个数组。接下来, 我们定义元素重叠的 `slice`, 分别用来表示第二季度的月份和北半球的夏季月份:

```
Q2 := months[4:7]
summer := months[6:9]
fmt.Println(Q2)      // ["April" "May" "June"]
fmt.Println(summer) // ["June" "July" "August"]
```

元素 June 同时包含在两个 `slice` 中。用下面的代码来输出两个 `slice` 的共同元素 (虽然效率不高),

```
for _, s := range summer {
    for _, q := range Q2 {
        if s == q {
            fmt.Printf("%s appears in both\n", s)
        }
    }
}
```

如果 `slice` 的引用超过了被引用对象的容量, 即 `cap(s)`, 那么会导致程序宕机; 但是如果 `slice` 的引用超出了被引用对象的长度, 即 `\text{len}(s)`, 那么最终 `slice` 会比原 `slice` 长:

```
fmt.Println(summer[:20]) // 宕机: 超过了被引用对象的边界
endlessSummer := summer[:5] // 在 slice 容量范围内扩展了 slice
fmt.Println(endlessSummer) // "[June July August September October]"
```

另外，注意求字符串（string）子串操作和对字节 slice（[]byte）做 slice 操作这两者的相似性。它们都写作 $x[m:n]$ ，并且都返回原始字节的一个子序列，同时它们的底层引用方式也是相同的，所以两个操作都消耗常量时间。区别在于：如果 x 是字符串，那么 $x[m:n]$ 返回的是一个字符串；如果 x 是字节 slice，那么返回的结果是字节 slice。

因为 slice 包含了指向数组元素的指针，所以将一个 slice 传递给函数的时候，可以在函数内部修改底层数组的元素。换言之，创建一个数组的 slice 等于为数组创建了一个别名（见 2.3.2 节）。下面的函数 reverse 就地反转了整型 slice 中的元素，它适用于任意长度的整型 slice。

gopl.io/ch4/rev

```
// 就地反转一个整型 slice 中的元素
func reverse(s []int) {
    for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
        s[i], s[j] = s[j], s[i]
    }
}
```

这里，反转整个数组 a ：

```
a := [...]int{0, 1, 2, 3, 4, 5}
reverse(a[:])
fmt.Println(a) // "[5 4 3 2 1 0]"
```

将一个 slice 左移 n 个元素的简单方法是连续调用 reverse 函数三次。第一次反转前 n 个元素，第二次反转剩下的元素，最后对整个 slice 再做一次反转（如果将元素右移 n 个元素，那么先做第三次调用）。

```
s := []int{0, 1, 2, 3, 4, 5}
// 向左移动两个元素
reverse(s[:2])
reverse(s[2:])
reverse(s)
fmt.Println(s) // "[2 3 4 5 0 1]"
```

注意初始化 slice s 的表达式和初始化数组 a 的表达式的区别。slice 字面量看上去和数组字面量很像，都是用逗号分隔并用花括号括起来的一个元素序列，但是 slice 没有指定长度。这种隐式区别的结果分别是创建有固定长度的数组和创建指向数组的 slice。和数组一样，slice 也按照顺序指定元素，也可以通过索引来指定元素，或者两者结合。

和数组不同的是，slice 无法做比较，因此不能用 $==$ 来测试两个 slice 是否拥有相同的元素。标准库里面提供了高度优化的函数 bytes.Equal 来比较两个字节 slice（[]byte）。但是对于其他类型的 slice，我们必须自己写函数来比较。

```
func equal(x, y []string) bool {
    if len(x) != len(y) {
        return false
    }
    for i := range x {
        if x[i] != y[i] {
            return false
        }
    }
    return true
}
```

这种深度比较看上去很简单，并且运行的时候并不比字符串数组使用 $==$ 做比较多耗费

时间。你或许奇怪为什么 slice 比较不可以直接使用 == 操作符做比较。这里有两个原因。首先，和数组元素不同，slice 的元素是非直接的，有可能 slice 可以包含它自身。虽然有办法处理这种特殊的情况，但是没有一种方法是简单、高效、直观的。

其次，因为 slice 的元素不是直接的，所以如果底层数组元素改变，同一个 slice 在不同的时间会拥有不同的元素。由于散列表（例如 Go 的 map 类型）仅对元素的键做浅拷贝，这就要求散列表里面键在散列表的整个生命周期内必须保持不变。因为 slice 需要深度比较，所以就不能用 slice 作为 map 的键。对于引用类型，例如指针和通道，操作符 == 检查的是引用相等性，即它们是否指向相同的元素。如果有一个相似的 slice 相等性比较功能，它或许会比较有用，也能解决 slice 作为 map 键的问题，但是如果操作符 == 对 slice 和数组的行为不一致，会带来困扰。所以最安全的方法就是不允许直接比较 slice。

slice 唯一允许的比较操作是和 nil 做比较，例如：

```
if summer == nil { /* ... */ }
```

slice 类型的零值是 nil。值为 nil 的 slice 没有对应的底层数组。值为 nil 的 slice 长度和容量都是零，但是也有非 nil 的 slice 长度和容量是零，例如 []int{} 或 make([]int, 3)[3:]。对于任何类型，如果它们的值可以是 nil，那么这个类型的 nil 值可以使用一种转换表达式，例如 []int(nil)。

```
var s []int // len(s) == 0, s == nil
s = nil // len(s) == 0, s == nil
s = []int(nil) // len(s) == 0, s == nil
s = []int{} // len(s) == 0, s != nil
```

所以，如果想检查一个 slice 是否是空，那么使用 len(s) == 0，而不是 s == nil，因为 s != nil 的情况下，slice 也有可能是空。除了可以和 nil 做比较之外，值为 nil 的 slice 表现和其他长度为零的 slice 一样。例如，reverse 函数调用 reverse(nil) 也是安全的。除非文档上面写明了与此相反，否则无论值是否为 nil，Go 的函数都应该以相同的方式对待所有长度为零的 slice。

内置函数 make 可以创建一个具有指定元素类型、长度和容量的 slice。其中容量参数可以省略，在这种情况下，slice 的长度和容量相等。

```
make([]T, len)
make([]T, len, cap) // 和 make([]T, cap)[:len] 功能相同
```

深入研究下，其实 make 创建了一个无名数组并返回了它的一个 slice；这个数组仅可以通过这个 slice 来访问。在上面的第一行代码中，所返回的 slice 引用了整个数组。在第二行代码中，slice 只引用了数组的前 len 个元素，但是它的容量是数组的长度，这为未来的 slice 元素留出空间。

4.2.1 append 函数

内置函数 append 用来将元素追加到 slice 的后面。

```
var runes []rune
for _, r := range "Hello, 世界" {
    runes = append(runes, r)
}
fmt.Printf("%q\n", runes) // "[H' 'e' 'l' 'l' 'o' ' ', ' ' '世' '界']"
```

虽然最方便的用法是 `[]rune("Hello, 世界")`，但是上面的循环演示了如何使用 `append` 来为一个 `rune` 类型的 `slice` 添加元素。

`append` 函数对理解 `slice` 的工作原理很重要，接下来看一个为 `[]int` 数组 `slice` 定义的方法 `appendInt`：

```
gopl.io/ch4/append

func appendInt(x []int, y int) []int {
    var z []int
    zlen := len(x) + 1
    if zlen <= cap(x) {
        // slice 仍有增长空间，扩展 slice 内容
        z = x[:zlen]
    } else {
        // slice 已无空间，为它分配一个新的底层数组
        // 为了达到分摊线性复杂性，容量扩展一倍
        zcap := zlen
        if zcap < 2*len(x) {
            zcap = 2 * len(x)
        }
        z = make([]int, zlen, zcap)
        copy(z, x) // 内置 copy 函数
    }
    z[len(x)] = y
    return z
}
```

每一次 `appendInt` 调用都必须检查 `slice` 是否仍有足够容量来存储数组中的新元素。如果 `slice` 容量足够，那么它就会定义一个新的 `slice`（仍然引用原始底层数组），然后将新元素 `y` 复制到新的位置，并返回这个新的 `slice`。输入参数 `slice x` 和函数返回值 `slice z` 拥有相同的底层数组。

如果 `slice` 的容量不够容纳增长的元素，`appendInt` 函数必须创建一个拥有足够容量的新底层数组来存储新元素，然后将元素从 `slice x` 复制到这个数组，再将新元素 `y` 追加到数组后面。返回值 `slice z` 将和输入参数 `slice x` 引用不同的底层数组。

使用循环语句来复制元素看上去直观一点，但是使用内置函数 `copy` 将更简单，`copy` 函数用来为两个拥有相同类型元素的 `slice` 复制元素。`copy` 函数的第一个参数是目标 `slice`，第二个参数是源 `slice`，`copy` 函数将源 `slice` 中的元素复制到目标 `slice` 中，这个和一般的元素赋值有点像，比如 `dest=src`。不同的 `slice` 可能对应相同的底层数组，甚至可能存在元素重叠。`copy` 函数有返回值，它返回实际上复制的元素个数，这个值是两个 `slice` 长度的较小值。所以这里不存在由于元素复制而导致的索引越界问题。

出于效率的考虑，新创建的数组容量会比实际容纳 `slice x` 和 `slice y` 所需要的最小长度更大一点。在每次数组容量扩展时，通过扩展一倍的容量来减少内存分配的次数，这样也可以保证追加一个元素所消耗的是固定时间。下面的程序演示了这个效果：

```
func main() {
    var x, y []int
    for i := 0; i < 10; i++ {
        y = appendInt(x, i)
        fmt.Printf("%d cap=%d\n", i, cap(y))
        x = y
    }
}
```

每次 slice 容量的改变都意味着一次底层数组重新分配和元素复制：

```

0 cap=1 [0]
1 cap=2 [0 1]
2 cap=4 [0 1 2]
3 cap=4 [0 1 2 3]
4 cap=8 [0 1 2 3 4]
5 cap=8 [0 1 2 3 4 5]
6 cap=8 [0 1 2 3 4 5 6]
7 cap=8 [0 1 2 3 4 5 6 7]
8 cap=16 [0 1 2 3 4 5 6 7 8]
9 cap=16 [0 1 2 3 4 5 6 7 8 9]

```

我们来仔细看一下当 $i=3$ 时的情况。这个时候 slice x 拥有三个元素 $[0 1 2]$ ，但是容量是 4，这个时候 slice 最后还有一个空位置，所以调用 `appendInt` 追加元素 3 的时候，没有发生底层数组重新分配。调用的结果是 slice 的长度和容量都是 4，并且这个结果 slice 和 x 一样拥有相同的底层数组，如图 4-2 所示。

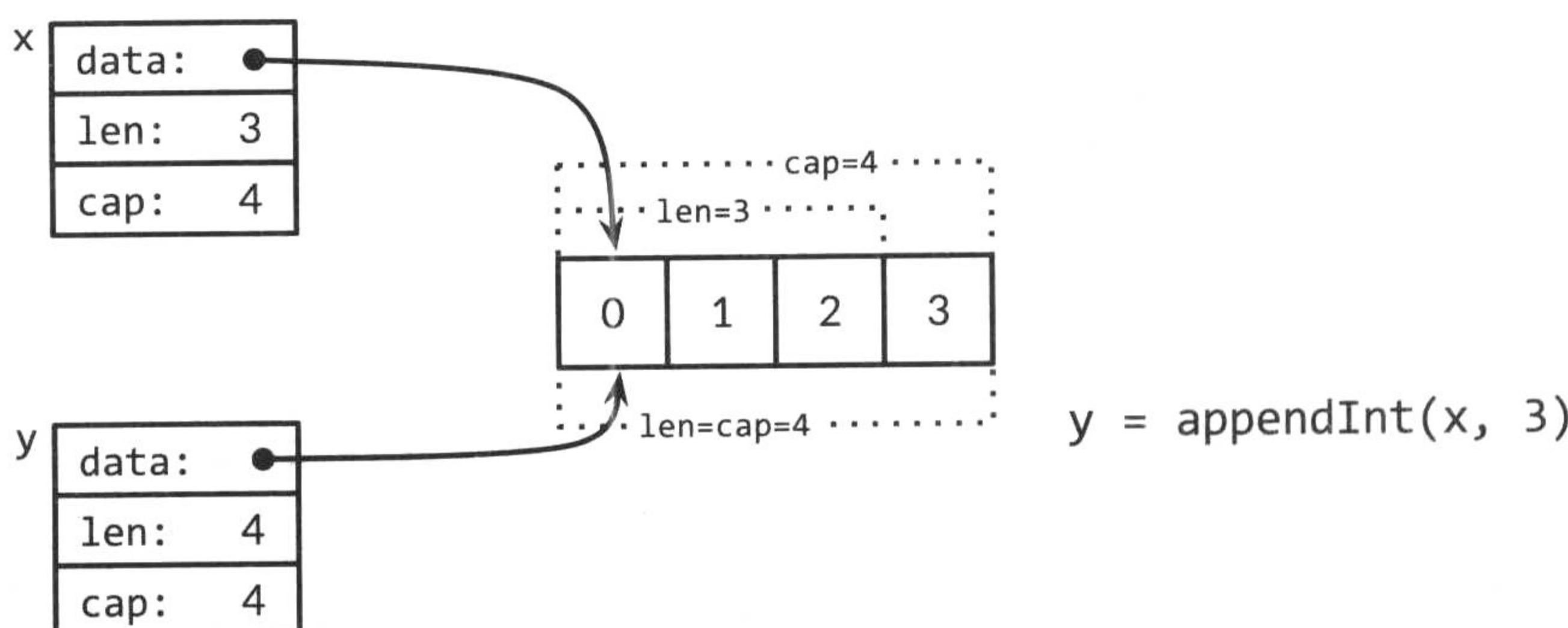


图 4-2 有容量元素的增长

在下一次循环中 $i=4$ ，这个时候原来的 slice 已经没有空间了，所以 `appendInt` 函数分配了一个长度为 8 的新数组。然后将 x 中的 4 个元素 $[0 1 2 3]$ 都复制到新的数组中，最后再追加新元素 i 。这样结果 slice 的长度就是 5，而容量是 8。多分配的三个位置就留给接下来的循环添加值使用，在接下来的三次循环中，就不需要再重新分配空间。所以 y 和 x 是不同数组的 slice。这个操作过程如图 4-3 所示。

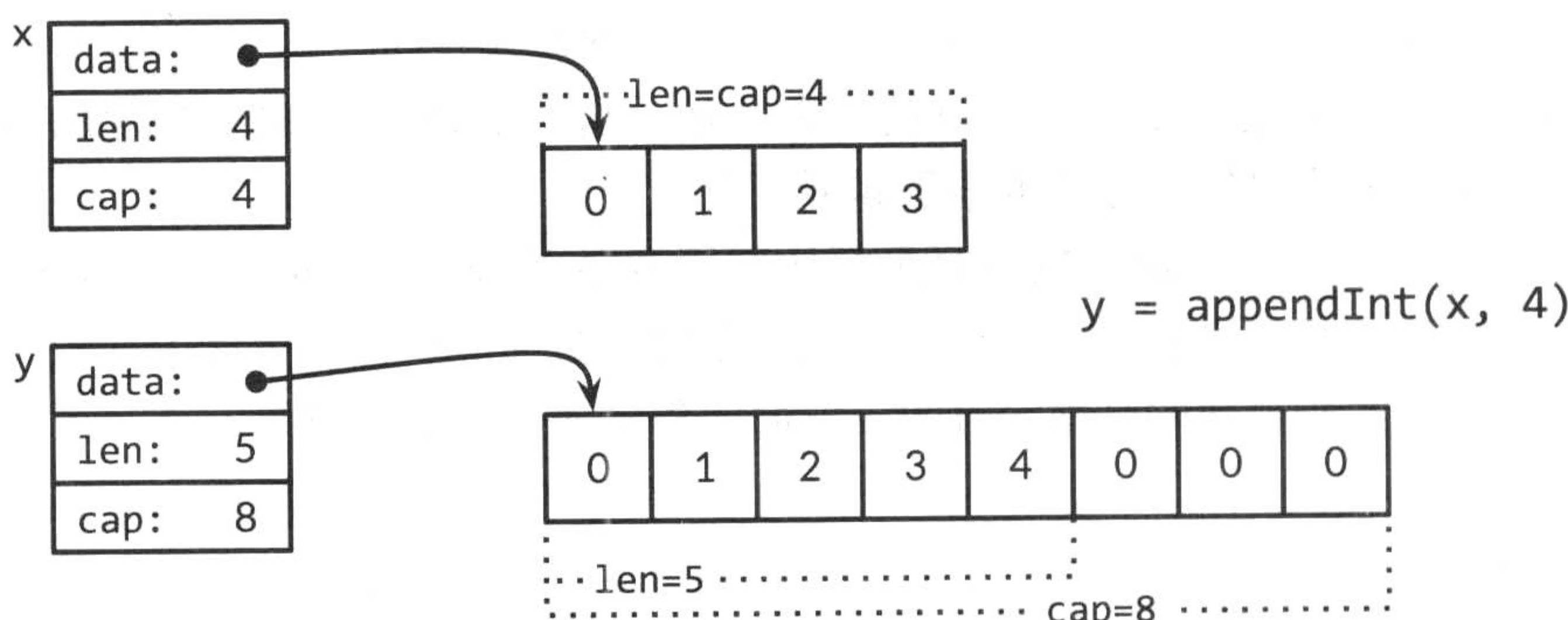


图 4-3 无容量元素的增长

内置的 `append` 函数使用了比这里的 `appendInt` 更复杂的增长策略。通常情况下，我们并不清楚一次 `append` 调用会不会导致一次新的内存分配，所以我们不能假设原始的 slice 和调

用 `append` 后的结果 `slice` 指向同一个底层数组，也无法证明它们就指向不同的底层数组。同样，我们也无法假设旧 `slice` 上对元素的操作会或者不会影响新的 `slice` 元素。所以，通常我们将 `append` 的调用结果再次赋值给传入 `append` 函数的 `slice`：

```
runes = append(runes, r)
```

不仅仅是在调用 `append` 函数的情况下需要更新 `slice` 变量。另外，对于任何函数，只要有可能改变 `slice` 的长度或者容量，抑或是使得 `slice` 指向不同的底层数组，都需要更新 `slice` 变量。为了正确地使用 `slice`，必须记住，虽然底层数组的元素是间接引用的，但是 `slice` 的指针、长度和容量不是。要更新一个 `slice` 的指针，长度或容量必须使用如上所示的显式赋值。从这个角度看，`slice` 并不是纯引用类型，而是像下面这种聚合类型：

```
type IntSlice struct {
    ptr      *int
    len, cap int
}
```

`appendInt` 函数只能给 `slice` 添加一个元素，但是内置的 `append` 函数可以同时给 `slice` 添加多个元素，甚至添加另一个 `slice` 里的所有元素。

```
var x []int
x = append(x, 1)
x = append(x, 2, 3)
x = append(x, 4, 5, 6)
x = append(x, x...) // 追加 x 中的所有元素
fmt.Println(x)       // "[1 2 3 4 5 6 1 2 3 4 5 6]"
```

可以简单修改一下 `appendInt` 函数来匹配 `append` 的功能。函数 `appendInt` 参数声明中的省略号 “...” 表示该函数可以接受可变长度参数列表。上面例子中 `append` 函数的参数后面的省略号表示如何将一个 `slice` 转换为参数列表。5.7 节会详细解释这种机制。

```
func appendInt(x []int, y ...int) []int {
    var z []int
    zlen := len(x) + len(y)
    // ... 扩展 slice z 的长度至少到 zlen...
    copy(z[len(x):], y)
    return z
}
```

扩展 `slice` `z` 底层数组的逻辑和上面一样，所以就不重复了。

4.2.2 slice 就地修改

我们多看一些就地使用 `slice` 的例子，比如 `rotate` 和 `reverse` 这种可以就地修改 `slice` 元素的函数。下面的函数 `nonempty` 可以从给定的一个字符串列表中去除空字符串并返回一个新的 `slice`。

```
gopl.io/ch4/nonempty
// Nonempty 演示了 slice 的就地修改算法
package main

import "fmt"

// nonempty 返回一个新的 slice, slice 中的元素都是非空字符串
// 在函数的调用过程中, 底层数组的元素发生了改变
```

```
func nonempty(strings []string) []string {
    i := 0
    for _, s := range strings {
        if s != "" {
            strings[i] = s
            i++
        }
    }
    return strings[:i]
}
```

这里有一点是输入的 slice 和输出的 slice 拥有相同的底层数组，这样就避免在函数内部重新分配一个数组。当然，这种情况下，底层数组的元素只是部分被修改，示例如下：

```
data := []string{"one", "", "three"}
fmt.Printf("%q\n", nonempty(data)) // `["one" "three"]`
fmt.Printf("%q\n", data)           // `["one" "three" "three"]`
```

因此，通常我们会这样来写：`data = nonempty(data)`。

函数 `nonempty` 还可以利用 `append` 函数来写：

```
func nonempty2(strings []string) []string {
    out := strings[:0] // 引用原始 slice 的新的零长度的 slice
    for _, s := range strings {
        if s != "" {
            out = append(out, s)
        }
    }
    return out
}
```

无论使用哪种方式，重用底层数组的结果是每一个输入值的 slice 最多只有一个输出的结果 slice，很多从序列中过滤元素再组合结果的算法都是这样做的。这种精细的 slice 使用方式只是一个特例，并不是规则，但是偶尔这样做可以让实现更清晰、高效、有用。

slice 可以用来实现栈。给定一个空的 slice 元素 `stack`，可以使用 `append` 向 slice 尾部追加值：

```
stack = append(stack, v) // push v
```

栈的顶部是最后一个元素：

```
top := stack[len(stack)-1] // 栈顶
```

通过弹出最后一个元素来缩减栈：

```
stack = stack[:len(stack)-1] // pop
```

为了从 slice 的中间移除一个元素，并保留剩余元素的顺序，可以使用函数 `copy` 来将高位索引的元素向前移动来覆盖被移除元素所在位置：

```
func remove(slice []int, i int) []int {
    copy(slice[i:], slice[i+1:])
    return slice[:len(slice)-1]
}

func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 8 9]"
}
```

如果不需要维持 slice 中剩余元素的顺序，可以简单地将 slice 的最后一个元素赋值给被移除元素所在的索引位置：

```
func remove(slice []int, i int) []int {
    slice[i] = slice[len(slice)-1]
    return slice[:len(slice)-1]
}

func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 9 8"]
}
```

练习 4.3：重写函数 `reverse`，使用数组指针作为参数而不是 slice。

练习 4.4：编写一个函数 `rotate`，实现一次遍历就可以完成元素旋转。

练习 4.5：编写一个就地处理函数，用于去除 `[]string` slice 中相邻的重复字符串元素。

练习 4.6：编写一个就地处理函数，用于将一个 UTF-8 编码的字节 slice 中所有相邻的 Unicode 空白字符（查看 `unicode.IsSpace`）缩减为一个 ASCII 空白字符。

练习 4.7：修改函数 `reverse`，来翻转一个 UTF-8 编码的字符串中的字符元素，传入参数是该字符串对应的字节 slice 类型（`[]byte`）。你可以做到不需要重新分配内存就实现该功能吗？

4.3 map

散列表是设计精妙、用途广泛的数据结构之一。它是一个拥有键值对元素的无序集合。在这个集合中，键的值是唯一的，键对应的值可以通过键来获取、更新或移除。无论这个散列表有多大，这些操作基本上是通过常量时间的键比较就可以完成。

在 Go 语言中，map 是散列表的引用，map 的类型是 `map[K]V`，其中 `K` 和 `V` 是字典的键和值对应的数据类型。map 中所有的键都拥有相同的数据类型，同时所有的值也都拥有相同的数据类型，但是键的类型和值的类型不一定相同。键的类型 `K`，必须是可以通过操作符 `==` 来进行比较的数据类型，所以 map 可以检测某一个键是否已经存在。虽然浮点型是可以比较的，但是比较浮点型的相等性不是一个好主意，如第 3 章所述，尤其是在 `NaN` 可以是浮点型值的时候。当然，值类型 `V` 没有任何限制。

内置函数 `make` 可以用来创建一个 map：

```
ages := make(map[string]int) // 创建一个从 string 到 int 的 map
```

也可以使用 map 的字面量来新建一个带初始化键值对元素的字典：

```
ages := map[string]int{
    "alice": 31,
    "charlie": 34,
}
```

这个等价于：

```
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

因此，新的空 map 的另外一种表达式是：`map[string]int{}`。

map 的元素访问也是通过下标的方式：

```
ages["alice"] = 32
fmt.Println(ages["alice"]) // "32"
```

可以使用内置函数 `delete` 来从字典中根据键移除一个元素：

```
delete(ages, "alice") // 移除元素 ages["alice"]
```

即使键不在 `map` 中，上面的操作也都是安全的。`map` 使用给定的键来查找元素，如果对应的元素不存在，就返回值类型的零值。例如，下面的代码同样可以工作，尽管 "bob" 还不是 `map` 的键，因为 `ages["bob"]` 的值是 0。

```
ages["bob"] = ages["bob"] + 1 // 生日快乐
```

快捷赋值方式（如 `x+=y` 和 `x++`）对 `map` 中的元素同样适用，所以上面的代码还可以写成：

```
ages["bob"] += 1
```

或者更简洁的：

```
ages["bob"]++
```

但是 `map` 元素不是一个变量，不可以获取它的地址，比如这样是不对的：

```
_ = &ages["bob"] // 编译错误，无法获取 map 元素的地址
```

我们无法获取 `map` 元素的地址的一个原因是 `map` 的增长可能会导致已有元素被重新散列到新的存储位置，这样就可能使得获取的地址无效。

可以使用 `for` 循环（结合 `range` 关键字）来遍历 `map` 中所有的键和对应的值，就像上面遍历 `slice` 一样。循环语句的连续迭代将会使得变量 `name` 和 `age` 被赋予 `map` 中的下一对键和值。

```
for name, age := range ages {
    fmt.Printf("%s\t%d\n", name, age)
}
```

`map` 中元素的迭代顺序是不固定的，不同的实现方法会使用不同的散列算法，得到不同的元素顺序。实践中，我们认为这种顺序是随机的，从一个元素开始到后一个元素，依次执行。这个是有意为之的，这样可以使得程序在不同的散列算法实现下变得健壮。如果需要按照某种顺序来遍历 `map` 中的元素，我们必须显式地来给键排序。例如，如果键是字符串类型，可以使用 `sort` 包中的 `Strings` 函数来进行键的排序，这是一种常见的模式：

```
import "sort"

var names []string
for name := range ages {
    names = append(names, name)
}
sort.Strings(names)
for _, name := range names {
    fmt.Printf("%s\t%d\n", name, ages[name])
}
```

因为我们一开始就知道 `slice` `names` 的长度，所以直接指定一个 `slice` 的长度会更加高效。下面的语句创建了一个初始元素为空但是容量足够容纳 `ages` `map` 中所有键的 `slice`：

```
names := make([]string, 0, len(ages))
```

在上面的第一个循环中，我们只需要 `map` `ages` 的所有键，所以我们忽略了循环中的第二个变量。在第二个循环中，我们只需要使用 `slice` `names` 中的元素值，所以我们使用空白标

识符`_`来忽略第一个变量，即元素索引。

`map`类型的零值是`nil`，也就是说，没有引用任何散列表。

```
var ages map[string]int
fmt.Println(ages == nil)    // "true"
fmt.Println(len(ages) == 0) // "true"
```

大多数的`map`操作都可以安全地在`map`的零值`nil`上执行，包括查找元素，删除元素，获取`map`元素个数(`len`)，执行`range`循环，因为这和空`map`的行为一致。但是向零值`map`中设置元素会导致错误：

```
ages["carol"] = 21 // 容机：为零值 map 中的项赋值
```

设置元素之前，必须初始化`map`。

通过下标的方式访问`map`中的元素总是会有值。如果键在`map`中，你将得到键对应的值；如果键不在`map`中，你将得到`map`值类型的零值，如同对于`ages["bob"]`的操作结果。很多情况下，这个没有问题，但是有时候你需要知道一个元素是否在`map`中。例如，如果元素类型是数值类型，你需要能够辨别一个不存在的元素或者恰好这个元素的值是0，可以这样做：

```
age, ok := ages["bob"]
if !ok { /* "bob" 不是字典中的键, age == 0 */ }
```

通常这两条语句合并成一条语句，如下所示：

```
if age, ok := ages["bob"]; !ok { /* ... */ }
```

通过这种下标方式访问`map`中的元素输出两个值，第二个值是一个布尔值，用来报告该元素是否存在。这个布尔变量一般叫作`ok`，尤其是它立即用在`if`条件判断中的时候。

和`slice`一样，`map`不可比较，唯一合法的比较就是和`nil`做比较。为了判断两个`map`是否拥有相同的键和值，必须写一个循环：

```
func equal(x, y map[string]int) bool {
    if len(x) != len(y) {
        return false
    }
    for k, xv := range x {
        if yv, ok := y[k]; !ok || yv != xv {
            return false
        }
    }
    return true
}
```

注意我们如何使用`!ok`来区分“元素不存在”和“元素存在但值为零”的情况。如果我们简单地写成了`xv != y[k]`，那么下面的调用将错误地报告两个`map`是相等的。

```
// 如果 equal 函数写法错误, 结果为 True
equal(map[string]int{"A": 0}, map[string]int{"B": 42})
```

`Go`没有提供集合类型，但是既然`map`的键都是唯一的，就可以用`map`来实现这个功能。为了模拟这个功能，程序`dedup`读取一系列的行，并且只输出每个不同行一次。这个是1.3节演示过的`dup`程序的变体。程序`dedup`使用`map`的键来存储这些已经出现过的行，来确保接下来出现的相同行不会输出。

gopl.io/ch4/dedup

```
func main() {
    seen := make(map[string]bool) // 字符串集合
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
        if !seen[line] {
            seen[line] = true
            fmt.Println(line)
        }
    }
    if err := input.Err(); err != nil {
        fmt.Fprintf(os.Stderr, "dedup: %v\n", err)
        os.Exit(1)
    }
}
```

Go程序员通常把这种使用map的方式描述成字符串集合，但是请注意，并不是所有的`map[string]bool`都是简单的集合，有一些map的值会同时包含`true`和`false`的情况。

有时候，我们需要一个map并且需求它的键是slice，但是因为map的键必须是可以比较的，所以这个功能无法直接实现。然而，我们可以分两步来做。首先，定义一个帮助函数`k`将每一个键都映射到字符串，当且仅当`x`和`y`相等的时候，我们认为`k(x) == k(y)`。然后，就可以创建一个map，map的键是字符串类型，在每个键元素被访问的时候，调用这个帮助函数。

下面的例子通过一个字符串列表使用一个map来记录Add函数被调用的次数。帮助函数使用`fmt.Sprintf`来将一个字符串slice转换为一个适合做map键的字符串，使用`%q`来格式化slice并记录每个字符串的边界。

```
var m = make(map[string]int)

func k(list []string) string { return fmt.Sprintf("%q", list) }

func Add(list []string) { m[k(list)]++ }
func Count(list []string) int { return m[k(list)] }
```

同样的方法适用于任何不可直接比较的键类型，不仅仅局限于slice。甚至有的时候，你不想让键通过`==`来比较相等性，而是自定义一种比较方法，例如字符串不区分大小写的比较。同样`k(x)`的类型不一定是字符串类型，任何能够得到想要的比较结果的可比较类型都可以，例如整数、数组或者结构体。

这里还有一个关于map的例子，一个统计输入中Unicode代码点出现次数的程序。虽然存在着大量可能的字符，但是在一篇文档中仅会有这个巨大字符集的一部分，所以很自然地使用map来追踪每个字符出现的次数。

gopl.io/ch4/charcount

```
// charcount 计算 Unicode 字符的个数
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "unicode"
    "unicode/utf8"
)
```

```

func main() {
    counts := make(map[rune]int)      // Unicode 字符数量
    var utflen [utf8.UTFMax + 1]int // UTF-8 编码的长度
    invalid := 0                   // 非法 UTF-8 字符数量

    in := bufio.NewReader(os.Stdin)
    for {
        r, n, err := in.ReadRune() // 返回 rune、 nbytes、 error
        if err == io.EOF {
            break
        }
        if err != nil {
            fmt.Fprintf(os.Stderr, "charcount: %v\n", err)
            os.Exit(1)
        }
        if r == unicode.ReplacementChar && n == 1 {
            invalid++
            continue
        }
        counts[r]++
        utflen[n]++
    }
    fmt.Printf("rune\tcount\n")
    for c, n := range counts {
        fmt.Printf("%q\t%d\n", c, n)
    }
    fmt.Print("\nlen\tcount\n")
    for i, n := range utflen {
        if i > 0 {
            fmt.Printf("%d\t%d\n", i, n)
        }
    }
    if invalid > 0 {
        fmt.Printf("\n%d invalid UTF-8 characters\n", invalid)
    }
}

```

函数 `ReadRune` 解码 UTF-8 编码，并返回三个值：解码的字符、UTF-8 编码中字节的长度和错误值。这里唯一可能出现的错误是文件结束（EOF）。如果输入的不是合法的 UTF-8 字符，那么返回的字符是 `code.ReplacementChar` 并且长度是 1。

`charcount` 程序还输出了 UTF-8 编码长度出现的次数，这个时候 `map` 不再是一个合适的数据类型，因为编码的长度是变化的，一个字符对应的字节长度可能值从 1 到 `utf8.UTFMax`（这里是 4）各不相同，所以这个时候使用数组使程序更加精简一点。

作为一个实验，我们对本书运行了一次 `charcount` 程序，虽然本书英文版内容大多数是英文，但是也确实包含一些非 ASCII 字符，这里是最多的 10 个非 ASCII 字符：

° 27 世 15 界 14 é 13 × 10 ≤ 5 × 5 国 4 ◊ 4 □ 3

这里是 UTF-8 编码长度的分布：

len	count
1	765391
2	60
3	70
4	0

`map` 的值类型本身可以是复合数据类型，例如 `map` 或 `slice`。在下面的代码中，变量 `graph` 的键类型是 `string` 类型；值类型是 `map` 类型 `map[string]bool`，表示一个字符串集合。

因此，`graph` 建立了一个从字符串到字符串集合的映射。

```
gopl.io/ch4/graph
var graph = make(map[string]map[string]bool)
func addEdge(from, to string) {
    edges := graph[from]
    if edges == nil {
        edges = make(map[string]bool)
        graph[from] = edges
    }
    edges[to] = true
}
func hasEdge(from, to string) bool {
    return graph[from][to]
}
```

函数 `addEdge` 演示了一种符合语言习惯的延迟初始化 `map` 的方法，当 `map` 中的每个键第一次出现的时候初始化。函数 `hasEdge` 演示了在 `map` 中值不存在的情况下，也可以直接使用。即使 `from` 和 `to` 都不存在，`graph[from][to]` 也始终可以给出一个有意义的值。

练习 4.8：修改 `charcount` 的代码来统计字母、数字和其他在 Unicode 分类中的字符数量，可以使用函数 `unicode.IsLetter` 等。

练习 4.9：编写一个程序 `wordfreq` 来汇总输入文本文件中每个单词出现的次数。在第一次调用 `Scan` 之前，需要使用 `input.Split(bufio.ScanWords)` 来将文本行按照单词分割而不是行分割。

4.4 结构体

结构体是将零个或者多个任意类型的命名变量组合在一起的聚合数据类型。每个变量都叫作结构体的成员。在数据处理领域，结构体使用的经典实例是员工信息记录，记录中有唯一 ID、姓名、地址、出生日期、职位、薪水、直属领导等信息。所有的这些员工信息成员都作为一个整体组合在一个结构体里面，可以复制一个结构体，将它传递给函数，作为函数的返回值，将结构体存储到数组中，等等。

下面的语句定义了一个叫 `Employee` 的结构体和一个结构体变量 `dilbert`：

```
type Employee struct {
    ID      int
    Name    string
    Address string
    DoB     time.Time
    Position string
    Salary   int
    ManagerID int
}
var dilbert Employee
```

`dilbert` 的每一个成员都通过点号方式来访问，就像 `dilbert.Name` 和 `dilbert.DoB` 这样。由于 `dilbert` 是一个变量，它的所有成员都是变量，因此可以给结构体的成员赋值：

```
dilbert.Salary -= 5000 // 写的代码太少了，降薪
```

或者获取成员变量的地址，然后通过指针来访问它：

```
position := &dilbert.Position
*position = "Senior " + *position // 工作外包给 Elbonia，所以升职
```

点号同样可以用在结构体指针上：

```
var employeeOfTheMonth *Employee = &dilbert
employeeOfTheMonth.Position += " (proactive team player)"
```

后面一条语句等价于：

```
(*employeeOfTheMonth).Position += " (proactive team player)"
```

函数 `EmployeeID` 通过给定的参数 `ID` 来返回一个指向 `Employee` 结构体的指针。可以用点号来访问它的成员变量：

```
func EmployeeByID(id int) *Employee { /* ... */ }

fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // 尖头发的老板Θ

id := dilbert.ID
EmployeeByID(id).Salary = 0 // 被开除了……不知道为什么
```

最后一条语句更新了函数 `EmployeeID` 返回的指针指向的结构体 `Employee`。如果函数 `EmployeeID` 的返回值类型变成了 `Employee` 而不是 `*Employee`，那么代码将无法通过编译，因为赋值表达式的左侧无法识别出一个变量。

结构体的成员变量通常一行写一个，变量的名称在类型的前面，但是相同类型的连续成员变量可以写在一行上，就像这里的 `Name` 和 `Address`：

```
type Employee struct {
    ID          int
    Name, Address string
    DoB         time.Time
    Position    string
    Salary      int
    ManagerID   int
}
```

成员变量的顺序对于结构体同一性很重要。如果我们将也是字符串类型的 `Position` 和 `Name`、`Address` 组合在一起或者互换了 `Name` 和 `Address` 的顺序，那么我们就在定义一个不同的结构体类型。一般来讲，我们只组合相关的成员变量。

如果一个结构体的成员变量名称是首字母大写的，那么这个变量是可导出的，这个是 Go 最主要的访问控制机制。一个结构体可以同时包含可导出和不可导出的成员变量。

因为结构体类型一个成员变量占据一行，所以通常它的定义比较长。虽然可以在每次需要它的时候写出整个结构体类型定义，即匿名结构体类型，但是重复的工作会比较累，所以通常我们会定义命名结构体类型，比如 `Employee`。

命名结构体类型 `s` 不可以定义一个拥有相同结构体类型 `s` 的成员变量，也就是一个聚合类型不可以包含它自己（同样的限制对数组也适用）。但是 `s` 中可以定义一个 `s` 的指针类型，即 `*s`，这样我们就可以创建一些递归数据结构，比如链表和树。下面的代码给出了一个利用二叉树来实现插入排序的例子。

```
gopl.io/ch4/treesort
type tree struct {
    value      int
    left, right *tree
}
```

^Θ 尖头发的老板 (pointy-haired boss) 是“呆伯特”系列漫画中的老板形象，他缺乏一般的常识以及职位所要求的管理技能，爱说大话，且富有向现实挑战的精神。——编辑注

```
// 就地排序
func Sort(values []int) {
    var root *tree
    for _, v := range values {
        root = add(root, v)
    }
    appendValues(values[:0], root)
}

// appendValues 将元素按照顺序追加到 values 里面，然后返回结果 slice
func appendValues(values []int, t *tree) []int {
    if t != nil {
        values = appendValues(values, t.left)
        values = append(values, t.value)
        values = appendValues(values, t.right)
    }
    return values
}

func add(t *tree, value int) *tree {
    if t == nil {
        // 等价于返回 &tree{value: value}
        t = new(tree)
        t.value = value
        return t
    }
    if value < t.value {
        t.left = add(t.left, value)
    } else {
        t.right = add(t.right, value)
    }
    return t
}
```

结构体的零值由结构体成员的零值组成。通常情况下，我们希望零值是一个默认自然的、合理的值。例如，在`bytes.Buffer`中，结构体的初始值就是一个可以直接使用的空缓存。另外，第9章将讲到的`sync.Mutex`也是一个可以直接使用且未锁定状态的互斥锁。有时候，这种合理的初始值实现简单，但是有时候也需要类型的设计者花费时间来进行设计。

没有任何成员变量的结构体称为空结构体，写作`struct{}`。它没有长度，也不携带任何信息，但是有的时候会很有用。有一些Go程序员用它来替代被当作集合使用的`map`中的布尔值，来强调只有键是有用的，但由于这种方式节约的内存很少并且语法复杂，所以一般尽量避免这样用。

```
seen := make(map[string]struct{}) // 字符串集合
// ...
if _, ok := seen[s]; !ok {
    seen[s] = struct{}{}
    // ...首次出现 s...
}
```

4.4.1 结构体字面量

结构体类型的值可以通过结构体字面量来设置，即通过设置结构体的成员变量来设置。

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

p := Point{1, 2}
```

有两种格式的结构体字面量。第一种格式如上，它要求按照正确的顺序，为每个成员变

量指定一个值。这会给开发和阅读代码的人增加负担，因为他们必须记住每个成员变量的顺序，另外这也使得未来结构体成员变量扩充或者重新排列的时候代码维护性差。所以，这种格式一般用在定义结构体类型的包中或者一些有明显的成员变量顺序约定的小结构体中，比如 `image.Point{x, y}` 或者 `color.RGBA{red, green, blue, alpha}`。

我们用得更多的是第二种格式，通过指定部分或者全部成员变量的名称和值来初始化结构体变量，就像 1.4 节讲述的 Lissajous 程序那样：

```
anim := gif.GIF{LoopCount: nframes}
```

如果在这种初始化方式中某个成员变量没有指定，那么它的值就是该成员变量类型的零值。因为指定了成员变量的名字，所以它们的顺序是无所谓的。

这两种初始化方式不可以混合使用，另外也无法使用第一种初始化方式来绕过不可导出变量无法在其他包中使用的规则。

```
package p
type T struct{ a, b int } // a 和 b 都是不可导出的

package q
import "p"
var _ = p.T{a: 1, b: 2} // 编译错误，无法引用 a、b
var _ = p.T{1, 2}       // 编译错误，无法引用 a、b
```

虽然上面的最后一行代码没有显式地提到不可导出变量，但是它们被隐式地引用了，所以这也是不允许的。

结构体类型的值可以作为参数传递给函数或者作为函数的返回值。例如，下面的函数将 `Point` 缩放了一个比率：

```
func Scale(p Point, factor int) Point {
    return Point{p.X * factor, p.Y * factor}
}

fmt.Println(Scale(Point{1, 2}, 5)) // "{5 10}"
```

出于效率的考虑，大型的结构体通常都使用结构体指针的方式直接传递给函数或者从函数中返回。

```
func Bonus(e *Employee, percent int) int {
    return e.Salary * percent / 100
}
```

这种方式在函数需要修改结构体内容的时候也是必需的，在 Go 这种按值调用的语言中，调用的函数接收到的是实参的一个副本，并不是实参的引用。

```
func AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105 / 100
}
```

由于通常结构体都通过指针的方式使用，因此可以使用一种简单的方式来创建、初始化一个 `struct` 类型的变量并获取它的地址：

```
pp := &Point{1, 2}
```

这个等价于：

```
pp := new(Point)
*pp = Point{1, 2}
```

但是 `&Point{1,2}` 这种方式可以直接使用在一个表达式中，例如函数调用。

4.4.2 结构体比较

如果结构体的所有成员变量都可以比较，那么这个结构体就是可比较的。两个结构体的比较可以使用 `==` 或者 `!=`。其中 `==` 操作符按照顺序比较两个结构体变量的成员变量，所以下面的两个输出语句是等价的：

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

p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q) // "false"
```

和其他可比较的类型一样，可比较的结构体类型都可以作为 `map` 的键类型。

```
type address struct {
    hostname string
    port      int
}

hits := make(map[address]int)
hits[address{"golang.org", 443}]++
```

4.4.3 结构体嵌套和匿名成员

本节将讨论 Go 中不同寻常的结构体嵌套机制，这个机制可以让我们将一个命名结构体当作另一个结构体类型的匿名成员使用；并提供了一种方便的语法，使用简单的表达式（比如 `x.f`）就可以代表连续的成员（比如 `x.d.e.f`）。

想象一下 2D 绘图程序中会提供的关于形状的库，比如矩形、椭圆、星形和车轮形。这里定义了其中可能存在的两个类型：

```
type Circle struct {
    X, Y, Radius int
}

type Wheel struct {
    X, Y, Radius, Spokes int
}
```

`Circle` 类型定义了圆心的坐标 `X` 和 `Y`，另外还有一个半径 `Radius`。`Wheel` 类型拥有 `Circle` 类型的所有属性，另外还有一个 `Spokes` 属性，即车轮中条辐的数量。创建一个 `Wheel` 类型的对象：

```
var w Wheel
w.X = 8
w.Y = 8
w.Radius = 5
w.Spokes = 20
```

在需要支持的形状变多之后，我们将意识到它们之间的相似性和重复性。所以，很自然地，我们会重构相同的部分：

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

```
type Circle struct {
    Center Point
    Radius int
}

type Wheel struct {
    Circle Circle
    Spokes int
}
```

这个程序看上去变得更清晰了，但是访问 `Wheel` 的成员变麻烦了：

```
var w Wheel
w.Circle.Center.X = 8
w.Circle.Center.Y = 8
w.Circle.Radius = 5
w.Spokes = 20
```

Go 允许我们定义不带名称的结构体成员，只需要指定类型即可；这种结构体成员称做匿名成员。这个结构体成员的类型必须是一个命名类型或者指向命名类型的指针。下面的 `Circle` 和 `Wheel` 都拥有一个匿名成员。这里称 `Point` 被嵌套到 `Circle` 中，`Circle` 被嵌套到 `Wheel` 中。

```
type Circle struct {
    Point
    Radius int
}

type Wheel struct {
    Circle
    Spokes int
}
```

正因为有了这种结构体嵌套的功能，我们才能直接访问到我们需要的变量而不是指定一大串中间变量：

```
var w Wheel
w.X = 8          // 等价于 w.Circle.Point.X = 8
w.Y = 8          // 等价于 w.Circle.Point.Y = 8
w.Radius = 5     // 等价于 w.Circle.Radius = 5
w.Spokes = 20
```

上面注释里面的方式也是正确的，但是使用“匿名成员”的说法或许不合适。上面的结构体成员 `Circle` 和 `Point` 是有名字的，就是对应类型的名字，只是这些名字在点号访问变量时是可选的。当我们访问最终需要的变量的时候可以省略中间所有的匿名成员。

遗憾的是，结构体字面量并没有什么快捷方式来初始化结构体，所以下面的语句是无法通过编译的：

```
w = Wheel{8, 8, 5, 20}           // 编译错误，未知成员变量
w = Wheel{X: 8, Y: 8, Radius: 5, Spokes: 20} // 编译错误，未知成员变量
```

结构体字面量必须遵循形状类型的定义，所以我们使用下面的两种方式来初始化，这两种方式是等价的：

gopl.io/ch4/embed

```
w = Wheel{Circle{Point{8, 8}, 5}, 20}
```

```
w = Wheel{
    Circle: Circle{
        Point: Point{X: 8, Y: 8},
        Radius: 5,
    },
    Spokes: 20, // 注意，尾部的逗号是必需的 (Radius 后面的逗号也一样)
}

fmt.Printf("%#v\n", w)
// 输出
// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}

w.X = 42

fmt.Printf("%#v\n", w)
// 输出
// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

注意副词 `#` 如何使得 `Printf` 的格式化符号 `%v` 以类似 Go 语法的方式输出对象，这个方式里面包含了成员变量的名字。

因为“匿名成员”拥有隐式的名字，所以你不能在一个结构体里面定义两个相同类型的匿名成员，否则会引起冲突。由于匿名成员的名字是由它们的类型决定的，因此它们的可导出性也是由它们的类型决定的。在上面的例子中，`Point` 和 `Circle` 这两个匿名成员是可导出的。即使这两个结构体是不可导出的（`point` 和 `circle`），我们仍然可以使用快捷方式：

```
w.X = 8 // 等价于 w.circle.point.X = 8
```

但是注释中那种显式指定中间匿名成员的方式在声明 `circle` 和 `point` 的包之外是不允许的，因为它们是不可导出的。

到目前为止，我们所看到关于结构体嵌套的使用，仅仅是关于点号访问匿名成员内部变量的语法糖。后面我们将了解到匿名成员不一定是结构体类型，任何命名类型或者指向命名类型的指针都可以。不过话说回来，嵌套一个没有子成员的类型有什么用呢？

以快捷方式访问匿名成员的内部变量同样适用于访问匿名成员的内部方法。因此，外围的结构体类型获取的不仅是匿名成员的内部变量，还有相关的方法。这个机制就是从简单类型对象组合成复杂的复合类型的主要方式。在 Go 中，组合是面向对象编程方式的核心，这将在 6.3 节进一步讲述。

4.5 JSON

JavaScript 对象表示法（JSON）是一种发送和接收格式化信息的标准。JSON 不是唯一的标准，XML（见 7.14 节）、ASN.1 和 Google 的 Protocol Buffer 都是相似的标准，各自有适用的场景。但是因为 JSON 的简单、可读性强并且支持广泛，所以使用得最多。

Go 通过标准库 `encoding/json`、`encoding/xml`、`encoding/asn1` 和其他的库对这些格式的编码和解码提供了非常好的支持，这些库都拥有相同的 API。本节对使用最多的 `encoding/json` 做一个简要的描述。

JSON 是 JavaScript 值的 Unicode 编码，这些值包括字符串、数字、布尔值、数组和对象。JSON 是基本数据类型和复合数据类型的一种高效的、可读性强的表示方法。第 3 章讲解了基础数据类型，本章讲解了复合数据类型——数组、slice、结构体和 map。

JSON 最基本的类型是数字（以十进制或者科学计数法表示）、布尔值（`true` 或 `false`）和字符串。字符串是用双引号括起来的 Unicode 代码点的序列，使用反斜杠作为转义字符，通

过和 Go 类似的方式访问成员。当然，JSON 里面的 \uhhh 数字转义得到的是 UTF-16 编码，而不是 Go 里面的字符。

这些基础类型可以通过 JSON 的数组和对象进行组合。JSON 的数组是一个有序的元素序列，每个元素之间用逗号分隔，两边使用方括号括起来。JSON 的数组用来编码 Go 里面的数组和 slice。JSON 的对象是一个从字符串到值的映射，写成 name:value 对的序列，每个元素之间用逗号分隔，两边使用花括号括起来。JSON 的对象用来编码 Go 里面的 map（键为字符串类型）和结构体。例如：

```
boolean          true
number          -273.15
string          "She said \"Hello, 世界\""
array            ["gold", "silver", "bronze"]
object           {"year": 1980,
                  "event": "archery",
                  "medals": ["gold", "silver", "bronze"]}
```

想象一个程序需要收集电影的观看次数并提供推荐。这个程序的 Movie 类型和典型的元素列表都在下面提供了。（结构体中成员 Year 和 Color 后面的字符串字面量是成员的标签，稍后会讲解它。）

```
gopl.io/ch4/movie
type Movie struct {
    Title string
    Year  int `json:"released"`
    Color bool `json:"color,omitempty"`
    Actors []string
}

var movies = []Movie{
    {Title: "Casablanca", Year: 1942, Color: false,
     Actors: []string{"Humphrey Bogart", "Ingrid Bergman"}},
    {Title: "Cool Hand Luke", Year: 1967, Color: true,
     Actors: []string{"Paul Newman"}},
    {Title: "Bullitt", Year: 1968, Color: true,
     Actors: []string{"Steve McQueen", "Jacqueline Bisset"}},
    // ...
}
```

这种类型的数据结构体最适合 JSON，无论是从 Go 对象转为 JSON 还是从 JSON 转换为 Go 对象都很容易。把 Go 的数据结构（比如 movies）转换为 JSON 称为 marshal。marshal 是通过 json.Marshal 来实现的：

```
data, err := json.Marshal(movies)
if err != nil {
    log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)
```

Marshal 生成了一个字节 slice，其中包含一个不带有任何多余空白字符的很长的字符串。把生成的结果折叠一下放在这里：

```
[{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","Ingr
id Bergman"]}, {"Title":"Cool Hand Luke","released":1967,"color":true,"Ac
tors":["Paul Newman"]}, {"Title":"Bullitt","released":1968,"color":true,
"Actors":["Steve McQueen","Jacqueline Bisset"]}]
```

这种紧凑的表示方法包含了所有的信息但是难以阅读。为了方便阅读，有一个 json.

`MarshalIndent` 的变体可以输出整齐格式化过的结果。这个函数有两个参数，一个是定义每行输出的前缀字符串，另外一个是定义缩进的字符串。

```
data, err := json.MarshalIndent(movies, "", "    ")
if err != nil {
    log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)
```

上面的代码输出：

```
[{
    {
        "Title": "Casablanca",
        "released": 1942,
        "Actors": [
            "Humphrey Bogart",
            "Ingrid Bergman"
        ]
    },
    {
        "Title": "Cool Hand Luke",
        "released": 1967,
        "color": true,
        "Actors": [
            "Paul Newman"
        ]
    },
    {
        "Title": "Bullitt",
        "released": 1968,
        "color": true,
        "Actors": [
            "Steve McQueen",
            "Jacqueline Bisset"
        ]
    }
}]
```

`marshal` 使用 Go 结构体成员的名称作为 JSON 对象里面字段的名称（通过反射的方式，这将在 12.6 节中介绍）。只有可导出的成员可以转换为 JSON 字段，这就是为什么我们将 Go 结构体里面的所有成员都定义为首字母大写的。

你或许注意到了，上面的结构体成员 `Year` 对应地转换为 `released`，另外 `color` 转换为 `color`。这个是通过成员标签定义（field tag）实现的。成员标签定义是结构体成员在编译期间关联的一些元信息：

```
Year int `json:"released"`
Color bool `json:"color,omitempty"`
```

成员标签定义可以是任意字符串，但是按照习惯，是由一串由空格分开的标签键值对 `key:"value"` 组成的；因为标签的值使用双引号括起来，所以一般标签都是原生的字符串字面量。键 `json` 控制包 `encoding/json` 的行为，同样其他的 `encoding/...` 包也遵循这个规则。标签值的第一部分指定了 Go 结构体成员对应 JSON 中字段的名字。成员的标签通常这样使用，比如 `total_count` 对应 Go 里面的 `TotalCount`。`Color` 的标签还有一个额外的选项，`omitempty`，它表示如果这个成员的值是零值或者为空，则不输出这个成员到 JSON 中。所以，对于《Casablanca》这部电影，就没有输出成员 `Color` 到 JSON 中。

marshal 的逆操作将 JSON 字符串解码为 Go 数据结构，这个过程叫作 unmarshal，这个是由 `json.Unmarshal` 实现的。下面的代码将电影的 JSON 数据转换到结构体 `slice` 中，这个结构体唯一的成员就是 `Title`。通过合理地定义 Go 的数据结构，我们可以选择将哪部分 JSON 数据解码到结构体对象中，哪些数据可以丢弃。当函数 `Unmarshal` 调用完成后，它将填充结构体 `slice` 中 `Title` 的值，JSON 中其他的字段就丢弃了。

```
var titles []struct{ Title string }
if err := json.Unmarshal(data, &titles); err != nil {
    log.Fatalf("JSON unmarshaling failed: %s", err)
}
fmt.Println(titles) // "[{Casablanca} {Cool Hand Luke} {Bullitt}]"
```

很多的 Web 服务都提供 JSON 接口，通过发送 HTTP 请求来获取想要得到的 JSON 信息。我们通过查询 GitHub 提供的 issue 跟踪接口来演示一下。首先，要定义需要的类型和常量：

```
gopl.io/ch4/github
// 包 github 提供了 GitHub issue 跟踪接口的 Go API
// 详细查看 https://developer.github.com/v3/search/#search-issues.
package github

import "time"

const IssuesURL = "https://api.github.com/search/issues"

type IssuesSearchResult struct {
    TotalCount int `json:"total_count"`
    Items      []*Issue
}

type Issue struct {
    Number     int
    HTMLURL   string `json:"html_url"`
    Title      string
    State      string
    User       *User
    CreatedAt time.Time `json:"created_at"`
    Body       string   // Markdown 格式
}

type User struct {
    Login     string
    HTMLURL  string `json:"html_url"`
}
```

和前面一样，即使对应的 JSON 字段的名称不是首字母大写，结构体的成员名称也必须首字母大写。由于在 unmarshal 阶段，JSON 字段的名称关联到 Go 结构体成员的名称是忽略大小写的，因此这里只需要在 JSON 中有下划线而 Go 里面没有下划线的时候使用一下成员标签定义。同样，这里选择性地对 JSON 中的字段进行解码，因为相对于这里演示的内容，GitHub 的查询回复返回相当多的信息。

函数 `SearchIssues` 发送 HTTP 请求并将回复解析为 JSON。由于用户的查询请求参数中可能存在一些字符，这些字符在 URL 中是特殊字符，比如 `?` 或者 `&`，因此使用 `url.QueryEscape` 函数来确保它们拥有正确的含义。

```
gopl.io/ch4/github
package github
```

```

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "strings"
)

// SearchIssues 函数查询 GitHub 的 issue 跟踪接口
func SearchIssues(terms []string) (*IssuesSearchResult, error) {
    q := url.QueryEscape(strings.Join(terms, " "))
    resp, err := http.Get(IssuesURL + "?q=" + q)
    if err != nil {
        return nil, err
    }

    // 我们必须在所有的可能分支上面关闭 resp.Body
    // 第 5 章将讲述 defer，它可以让代码简单一点
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("search query failed: %s", resp.Status)
    }

    var result IssuesSearchResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        resp.Body.Close()
        return nil, err
    }
    resp.Body.Close()
    return &result, nil
}

```

前面的例子使用了 `json.Unmarshal` 来将整个字节 slice 解码为单个 JSON 实体。这里变化一下，使用流式解码器（即 `json.Decoder`），可以利用它来依次从字节流里面解码出多个 JSON 实体，我们现在还用不到这个功能。你或许猜到了，也有一个叫作 `json.Encoder` 的流式编码器。

调用 `Decode` 方法来填充变量 `result`。有各种方法来将结果格式化得好看一点。最简单的就是使用下面介绍的关于 `issues` 命令的方法，使用固定宽度的表格，下一节将讨论一个基于模板的复杂一点的方法。

gopl.io/ch4/issues

```

// 将符合搜索条件的 issue 输出为一个表格
package main

import (
    "fmt"
    "log"
    "os"
    "gopl.io/ch4/github"
)

func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d issues:\n", result.TotalCount)
    for _, item := range result.Items {
        fmt.Printf("#%-5d %9.9s %.55s\n",
            item.Number, item.User.Login, item.Title)
    }
}

```

命令行参数指定搜索的条件，该命令搜索 Go 项目的 issue 跟踪接口，查找关于 JSON 编码的 Open 状态的 bug 列表。

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 issues:
#5680 eigner encoding/json: set key converter on en/decoder
#6050 gopherbot encoding/json: provide tokenizer
#8658 gopherbot encoding/json: use bufio
#8462 kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901 rsc encoding/json: allow override type marshaling
#9812 klauspost encoding/json: string tag not symmetric
#7872 extempora encoding/json: Encoder internally buffers full output
#9650 cespare encoding/json: Decoding gives errPhase when unmarshaling
#6716 gopherbot encoding/json: include field name in unmarshal error message
#6901 lukescott encoding/json, encoding/xml: option to treat unknown fields
#6384 joeshaw encoding/json: encode precise floating point integers uniformly
#6647 btracey x/tools/cmd/godoc: display type kind of each named type
#4237 gjemiller encoding/base64: URLEncoding padding is optional
```

GitHub 的 Web 服务接口 (<https://developer.github.com/v3/>) 有很多的功能，这里就不再赘述了。

练习 4.10：修改 `issues` 实例，按照时间来输出结果，比如一个月以内，一年以内或者超过一年。

练习 4.11：开发一个工具来让用户可以通过命令行创建、读取、更新或者关闭 GitHub 的 issues，当需要额外输入的时候，调用他们喜欢的文本编辑器。

练习 4.12：流行的 Web 漫画 `xkcd` 有一个 JSON 接口。例如，调用 <https://xkcd.com/571/info.0.json> 输出漫画 571 的详细描述，这个是很多人最喜欢的之一。下载每一个 URL 并且构建一个离线索引。编写一个工具 `xkcd` 来使用这个索引，可以通过命令行指定的搜索条件来查找并输出符合条件的每个漫画的 URL 和剧本。

练习 4.13：基于 JSON 开发的 Web 服务，开放电影数据库让你可以在 <https://omdbapi.com/> 上通过名字来搜索电影并下载海报图片。开发一个 `poster` 工具以通过命令行指定的电影名称来下载海报。

4.6 文本和 HTML 模板

上面的例子仅仅给出了最简单的格式化，这种情况下，`Printf` 函数足够用了。但是有的情况下格式化会比这个复杂得多，并且要求格式和代码彻底分离。这个可以通过 `text/template` 包和 `html/template` 包里面的方法来实现，这两个包提供了一种机制，可以将程序变量的值代入到文本或者 HTML 模板中。

模板是一个字符串或者文件，它包含一个或者多个两边用双大括号包围的单元——`{{...}}`，这称为操作。大多数的字符串是直接输出的，但是操作可以引发其他的行为。每个操作在模板语言里面都对应一个表达式，提供的简单但强大的功能包括：输出值，选择结构体成员，调用函数和方法，描述控制逻辑（比如 `if-else` 语句和 `range` 循环），实例化其他的模板等。一个简单的字符串模板如下所示：

```
gopl.io/ch4/issuesreport
const templ = `{{.TotalCount}} issues:
{{range .Items}}
Number: {{.Number}}
User: {{.User.Login}}
```

```
Title: {{.Title | printf "%.64s"}}
Age: {{.CreatedAt | daysAgo}} days
{{end}}
```

模板首先输出符合条件的 issue 数量，然后分别输出每个 issue 的序号、用户、标题和距离创建时间已过去的天数。在这个操作里面，有一个表示当前值的标记，用点号（.）表示。点号最开始的时候表示模板里面的参数，在这个例子中即是 `github.IssuesSearchResult`。操作 `{{.TotalCount}}` 代表 `TotalCount` 成员的值，直接输出。`{{range.Items}}` 和 `{{end}}` 操作创建一个循环，所以它们内部的值会展开很多次，这个时候点号（.）表示 `Items` 里面连续的元素。

在操作中，符号 | 会将前一个操作的结果当做下一个操作的输入，和 UNIX 的 shell 管道类似。在 `Title` 的例子中，第二个操作就是 `printf` 函数，在所有的模板中，就是内置函数 `fmt.Sprintf` 的同义词。对于 `Age` 来说，第二个操作是 `daysAgo`，这个函数使用 `time.Since` 将 `CreatedAt` 转换为已过去的时间。

```
func daysAgo(t time.Time) int {
    return int(time.Since(t).Hours() / 24)
}
```

注意，`CreatedAt` 的类型是 `time.Time` 而不是 `string` 类型。同样地，一个类型可以定义方法来控制自己的字符串格式化方式（见 2.5 节），另外也可以定义方法来控制自身 JSON 序列化和反序列化的方式。`time.Time` 的 JSON 序列化值就是该类型标准的字符串表示方法。

通过模板输出结果需要两个步骤。首先，需要解析模板并转换为内部的表示方法，然后在指定的输入上面执行。解析模板只需要执行一次。下面的代码创建并解析上面定义的文本模板 `templ`。注意方法的链式调用：`template.New` 创建并返回一个新的模板，`Funcs` 添加 `daysAgo` 到模板内部可以访问的函数列表中，然后返回这个模板对象；最后调用 `Parse` 方法。

```
report, err := template.New("report").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(tmpl)
if err != nil {
    log.Fatal(err)
}
```

由于模板通常是在编译期间就固定下来的，因此无法解析模板将是程序中的一个严重的 bug。帮助函数 `template.Must` 提供了一种便捷的错误处理方式，它接受一个模板和错误作为参数，检查错误是否为 `nil`（如果不是 `nil`，则宕机），然后返回这个模板。5.9 节将讲述这个方法。

一旦创建了模板，添加了内部可调用的函数 `daysAgo`，然后解析，再检查，就可以使用 `github.IssuesSearchResult` 作为数据源，使用 `os.Stdout` 作为输出目标执行这个模板：

```
var report = template.Must(template.New("issuelist").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(tmpl))

func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    if err := report.Execute(os.Stdout, result); err != nil {
        log.Fatal(err)
    }
}
```

这个程序输出一个纯文本，如下所示：

```
$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 issues:
-----
Number: 5680
User: eigner
Title: encoding/json: set key converter on en/decoder
Age: 750 days
-----
Number: 6050
User: gopherbot
Title: encoding/json: provide tokenizer
Age: 695 days
-----
...
```

我们再来看 `html/template` 包。它使用和 `text/template` 包里面一样的 API 和表达式语句，并且额外地对出现在 HTML、JavaScript、CSS 和 URL 中的字符串进行自动转义。这些功能可以避免生成的 HTML 引发长久以来都会有的安全问题，比如注入攻击，对方利用 issue 的标题来包含不安全的代码，在模板中如果没有合理地进行转义，会让它们能够控制整个页面。

下面的模板将 issue 输出为 HTML 的表格，注意导入不同的包：

```
gopl.io/ch4/issueshtml
import "html/template"

var issueList = template.Must(template.New("issuelist").Parse(``)
<h1>{{.TotalCount}} issues</h1>
<table>
<tr style='text-align: left'>
  <th>#</th>
  <th>State</th>
  <th>User</th>
  <th>Title</th>
</tr>
{{range .Items}}
<tr>
  <td><a href='{{.HTMLURL}}'>{{.Number}}</a></td>
  <td>{{.State}}</td>
  <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
  <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`))
```

下面的命令对查询的结果执行新的模板，这些结果和上面的稍许不同：

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml repo:golang/go commenter:gopherbot json encoder >issues.html
```

图 4-4 显示了生成的 HTML 表格在 Web 浏览器中的样子。链接指向 GitHub 上面对应的页面。

#	State	User	Title
7872	open	extemporalgenome	encoding/json: Encoder internally buffers full output
5683	open	gopherbot	encoding/json: performance slower than expected
6901	open	lukescott	encoding/json, encoding/xml: option to treat unknown fields as an error
4474	closed	gopherbot	encoding/json: json encoder fails for embedded non-struct fields
4747	closed	gopherbot	encoding/json Added tag options to ignore fields of struct for encoder/decoder separately
7767	closed	gopherbot	encoding/json: Encoder adds trailing newlines
4606	closed	gopherbot	JSON Package fails to properly escape strings
8582	closed	matt-duch	encoding/json: inconsistent behavior in *(numeric type) and string tag option
6339	closed	gopherbot	encoding/json: Marshal of nil net.IP fails
7337	closed	gopherbot	encoding/json: make "json" tag user-settable
11508	closed	josharian	cmd/go: trace http viewer: "http: multiple response.WriteHeader calls"
1017	closed	gopherbot	json crash on {} input
8592	closed	gopherbot	encoding/json: No way to avoid HTMLEscape when Marshal()-ing
7846	closed	gopherbot	encoding/json: Slice created using reflect.MakeSlice() treated as interface{}
2761	closed	gopherbot	Marshaler cannot work with omitempty in encoding/json
1133	closed	gopherbot	encoding/json: inconsistent APIs
7841	closed	gopherbot	reflect: reflect.unpackEface reflect/value.go:174 unexpected fault address 0x0

图 4-4 将获取的 Go 项目的 issue 列表的 JSON 数据以 HTML 表格显示

上图中 issue 的 HTML 信息显示没有问题，但我们可以看在 issue 标题中包含 HTML 的元字符（比如 & 和 <）来看一下效果。我们选择了两个 issue 来做演示：

```
$ ./issueshtml repo:golang/go 3133 10535 >issues2.html
```

图 4-5 显示了查询的结果。注意，`html/template` 包自动将 HTML 元字符转义，这样标题才能显示正常。如果我们错误地使用了 `text/template` 包，那么字符串“<”将被当做小于号 '<'，而字符串 "<link>" 将变成一个 `link` 元素，这将改变 HTML 的文档结构，甚至有可能产生安全问题。

我们可以通过使用命名的字符串类型 `template.HTML` 类型而不是字符串类型避免模板自动转义受信任的 HTML 数据。同样的命名类型适用于受信任的 JavaScript、CSS 和 URL。下面的程序演示了相同数据在不同类型下的效果，A 是字符串类型而 B 是 `template.HTML` 类型。

gopl.io/ch4/autoescape

```
func main() {
    const templ = `<p>A: {{.A}}</p><p>B: {{.B}}</p>`
    t := template.Must(template.New("escape").Parse(templ))
    var data struct {
        A string          // 不受信任的纯文本
        B template.HTML   // 受信任的 HTML
    }
    data.A = "<b>Hello!</b>"
    data.B = "<b>Hello!</b>"
    if err := t.Execute(os.Stdout, data); err != nil {
        log.Fatal(err)
    }
}
```

#	State	User	Title
3133	closed	ukai	html/template: escape xmldesc as <?xml
10535	open	dvyukov	x/net/html: void element <link> has child nodes

图 4-5 issue 标题中的 HTML 元字符正确地显示

图 4-6 演示了这个模板在浏览器中的输出，我们可以看出来 A 转义了而 B 没有。

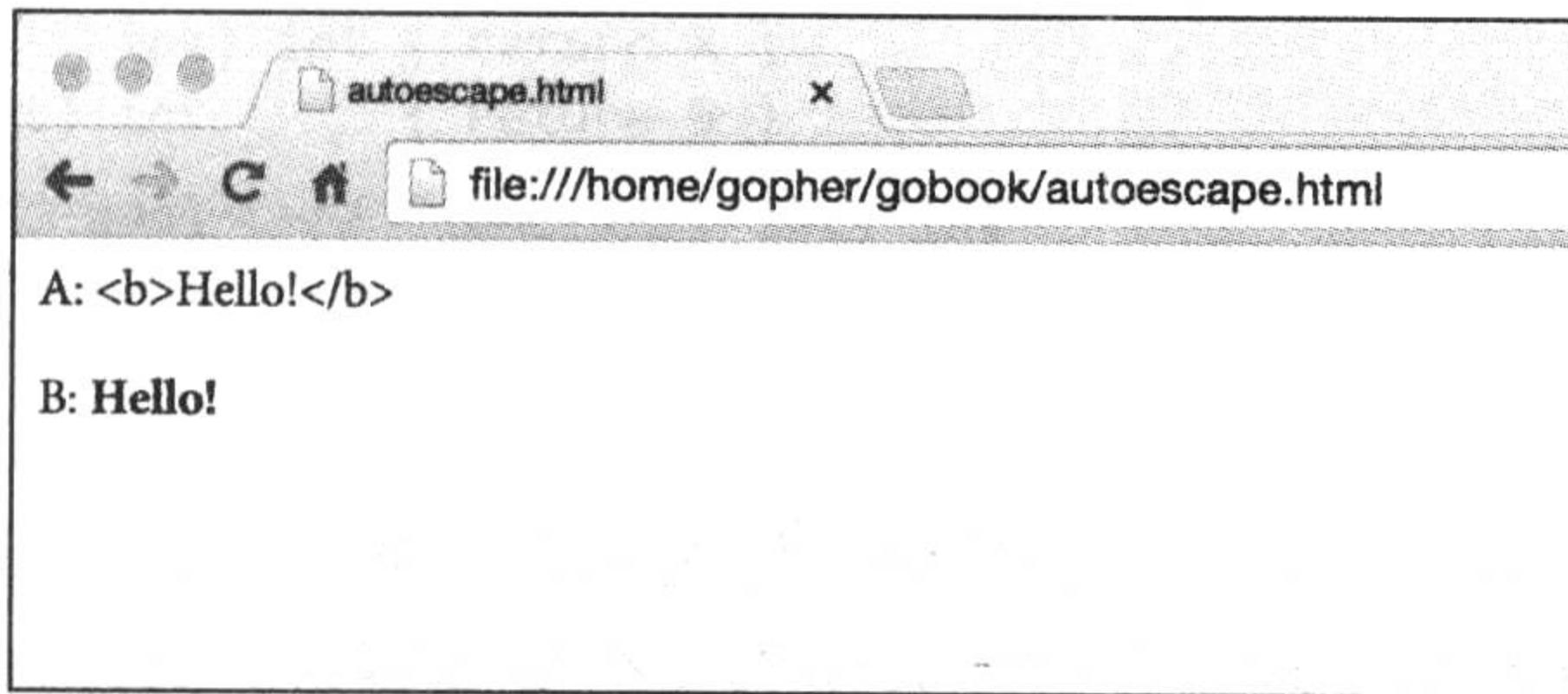


图 4-6 字符串值被 HTML 转义了，但是 `template.HTML` 值没有

这里仅仅演示了模板系统最基本的功能。如果你希望获取更多的信息，可以查询相关包的文档。

```
$ go doc text/template  
$ go doc html/template
```

练习 4.14： 创建一个 Web 服务器，可以通过查询 GitHub 并缓存信息，然后可以浏览 bug 列表、里程碑信息以及参与用户的信息。

函 数

函数包含连续的执行语句，可以在代码中通过调用函数来执行它们。函数能够将一个复杂的工作切分成多个更小的模块，使得多人协作变得更加容易。另外，函数对它的使用者隐藏了实现细节。这几方面的特性使函数成为多数编程语言的重要特性之一。

我们之前已经见过许多函数，现在让我们更彻底地探究一下函数。本章的运行示例是一个网络爬虫，它是 Web 搜索引擎的组件之一，负责抓取网页并分析页面包含的链接，将链接指向的页面也抓取下来，循环往复。利用网络爬虫的实现，我们可以更充分地了解到 Go 语言的递归、匿名函数、错误处理等方面的函数特性。

5.1 函数声明

每个函数声明都包含一个名字、一个形参列表、一个可选的返回列表以及函数体：

```
func name(parameter-list) (result-list) {  
    body  
}
```

形参列表指定了一组变量的参数名和参数类型，这些局部变量都由调用者提供的实参传递而来。返回列表则指定了函数返回值的类型。当函数返回一个未命名的返回值或者没有返回值的时候，返回列表的圆括号可以省略。如果一个函数既省略返回列表也没有任何返回值，那么设计这个函数的目的是调用函数之后所带来的附加效果。在下面的 `hypot` 函数中：

```
func hypot(x, y float64) float64 {  
    return math.Sqrt(x*x + y*y)  
}  
  
fmt.Println(hypot(3, 4)) // "5"
```

`x` 和 `y` 是函数声明中的形参，`3` 和 `4` 是调用函数时的实参，并且函数返回一个类型为 `float64` 的值。

返回值也可以像形参一样命名。这个时候，每一个命名的返回值会声明为一个局部变量，并根据变量类型初始化为相应的 0 值。

当函数存在返回列表时，必须显式地以 `return` 语句结束，除非函数明确不会走完整个执行流程，比如在函数中抛出宕机异常或者函数体内存在一个没有 `break` 退出条件的无限 `for` 循环。

在 `hypot` 函数中使用到一种简写，如果几个形参或者返回值的类型相同，那么类型只需要写一次。以下两个声明是完全相同的：

```
func f(i, j, k int, s, t string) { /* ... */ }  
func f(i int, j int, k int, s string, t string) { /* ... */ }
```

下面使用 4 种方式声明一个带有两个形参和一个返回值的函数，所有变量都是 `int` 类型。空白标识符用来强调这个形参在函数中未使用。

```

func add(x int, y int) int { return x + y }
func sub(x, y int) (z int) { z = x - y; return }
func first(x int, _ int) int { return x }
func zero(int, int) int { return 0 }

fmt.Printf("%T\n", add) // "func(int, int) int"
fmt.Printf("%T\n", sub) // "func(int, int) int"
fmt.Printf("%T\n", first) // "func(int, int) int"
fmt.Printf("%T\n", zero) // "func(int, int) int"

```

函数的类型称作函数签名。当两个函数拥有相同的形参列表和返回列表时，认为这两个函数的类型或签名是相同的。而形参和返回值的名字不会影响到函数类型，采用简写同样也不会影响到函数的类型。

每一次调用函数都需要提供实参来对应函数的每一个形参，包括参数的调用顺序也必须一致。Go 语言没有默认参数值的概念也不能指定实参名，所以除了用于文档说明之外，形参和返回值的命名不会对调用方有任何影响。

形参变量都是函数的局部变量，初始值由调用者提供的实参传递。函数形参以及命名返回值同属于函数最外层作用域的局部变量。

实参是按值传递的，所以函数接收到的是每个实参的副本；修改函数的形参变量并不会影响到调用者提供的实参。然而，如果提供的实参包含引用类型，比如指针、slice、map、函数或者通道，那么当函数使用形参变量时就有可能会间接地修改实参变量。

你可能偶尔会看到有些函数的声明没有函数体，那说明这个函数使用除了 Go 以外的语言实现。这样的声明定义了该函数的签名。

```

package math

func Sin(x float64) float64 // 使用汇编语言实现

```

5.2 递归

函数可以递归调用，这意味着函数可以直接或间接地调用自己。递归是一种实用的技术，可以处理许多带有递归特性的数据结构。在 4.4 节使用递归实现了对一棵树进行插入排序。本节再一次使用递归处理 HTML 文件。

下面的代码示例使用了一个非标准的包 `golang.org/x/net/html`，它提供了解析 HTML 的功能。`golang.org/x/…` 下的仓库（比如网络、国际化语言处理、移动平台、图片处理、加密功能以及开发者工具）都由 Go 团队负责设计和维护。这些包并不属于标准库，原因是它们还在开发当中，或者很少被 Go 程序员使用。

我们需要的 `golang.org/x/net/html` API 如下面的代码所示。函数 `html.Parse` 读入一段字节序列，解析它们，然后返回 HTML 文档树的根节点 `html.Node`。HTML 有多种节点，比如文本、注释等。但这里我们只关心表单的元素节点 `<name key='value'>`。

```

golang.org/x/net/html
package html

type Node struct {
    Type           NodeType
    Data           string
    Attr          []Attribute
    FirstChild, NextSibling *Node
}

```

```

type NodeType int32

const (
    ErrorNode NodeType = iota
    TextNode
    DocumentNode
    ElementNode
    CommentNode
    DoctypeNode
)
type Attribute struct {
    Key, Val string
}
func Parse(r io.Reader) (*Node, error)

```

主函数从标准输入中读入 HTML，使用递归的 visit 函数获取 HTML 文本的超链接，并且把所有的超链接输出。

```

gopl.io/ch5/findlinks1
// Findlinks1 输出从标准输入中读入的 HTML 文档中的所有链接
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlinks1: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) {
        fmt.Println(link)
    }
}

```

visit 函数遍历 HTML 树上的所有节点，从 HTML 锚元素 `` 中得到 href 属性的内容，将获取到的链接内容添加到字符串 slice，最后返回这个 slice：

```

// visit 函数会将 n 节点中的每个链接添加到结果中
func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c)
    }
    return links
}

```

要对树中的任意节点 n 进行递归，visit 递归地调用自己去访问节点 n 的所有子节点，并且将访问过的节点保存在 FirstChild 链表中。

我们在 Go 的主页运行 `findlinks`，使用管道将本书 1.5 节完成的 `fetch` 程序的输出定向到 `findlinks`。稍稍修改输出，使之更加简洁。

```
$ go build gopl.io/ch1/fetch
$ go build gopl.io/ch5/findlinks1
$ ./fetch https://golang.org | ./findlinks1
#
/doc/
/pkg/
/help/
/blog/
http://play.golang.org/
//tour.golang.org/
https://golang.org/dl/
//blog.golang.org/
/LICENSE
/doc/tos.html
http://www.google.com/intl/en/policies/privacy/
```

可以注意到会获取到各种不同形式的超链接。之后我们将看到如何解析这些地址，并将链接都转换为基于 `https://golang.org` 的 URL 绝对路径。

下一个程序使用递归遍历所有 HTML 文本中的节点树，并输出树的结构。当递归遇到每个元素时，它都会将元素标签压入栈，然后输出栈。

```
gopl.io/ch5/outline
func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "outline: %v\n", err)
        os.Exit(1)
    }
    outline(nil, doc)
}

func outline(stack []string, n *html.Node) {
    if n.Type == html.ElementNode {
        stack = append(stack, n.Data) // 把标签压入栈
        fmt.Println(stack)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        outline(stack, c)
    }
}
```

注意一个细节：尽管 `outline` 会将元素压栈但并不会出栈。当 `outline` 递归调用自己时，被调用的函数会接收到栈的副本。尽管被调用者可能会对 slice 进行元素的添加、修改甚至创建新数组的操作，但它并不会修改调用者原来传递的元素，所以当被调函数返回时，调用者的栈依旧保持原样。

以下是 `https://golang.org` 页面的 `outline`：

```
$ go build gopl.io/ch5/outline
$ ./fetch https://golang.org | ./outline
[html]
[html head]
[html head meta]
[html head title]
[html head link]
```

```
[html body]
[html body div]
[html body div]
[html body div div]
[html body div div form]
[html body div div form div]
[html body div div form div a]
...

```

通过 `outline` 可以发现，大多数的 HTML 文档都只会经过几层递归处理，但即使是一些需要复杂递归处理的文档也能够轻松应对。

许多编程语言使用固定长度的函数调用栈；大小在 64KB 到 2MB 之间。递归的深度会受限于固定长度的栈大小，所以当进行深度递归调用时必须谨防栈溢出。固定长度的栈甚至会造成一定的安全隐患。相比固定长的栈，Go 语言的实现使用了可变长度的栈，栈的大小会随着使用而增长，可达到 1GB 左右的上限。这使得我们可以安全地使用递归而不用担心溢出问题。

练习 5.1：改变 `findlinks` 程序，使用递归调用 `visit`（而不是循环）遍历 `n.FirstChild` 链表。

练习 5.2：写一个函数，用于统计 HTML 文档树内所有的元素个数，如 `p`、`div`、`span` 等。

练习 5.3：写一个函数，用于输出 HTML 文档树中所有文本节点的内容。但不包括 `<script>` 或 `<style>` 元素，因为这些内容在 Web 浏览器中是不可见的。

练习 5.4：扩展 `visit` 函数，使之能够获得到其他种类的链接地址，比如图片、脚本或样式表的链接。

5.3 多返回值

一个函数能够返回不止一个结果。我们之前已经见过标准包内的许多函数返回两个值，一个期望得到的计算结果与一个错误值，或者一个表示函数调用是否正确的布尔值。下面来看看怎样写一个这样的函数。

下面程序中的 `findLinks` 函数有一个小的变化，它将自己发送 HTTP 请求，因此不再需要运行 `fetch` 函数。因为 HTTP 请求和解析操作可能会失败，所以 `findLinks` 声明了两个结果：一个是发现的链接列表，另一个是错误。另外，HTML 的解析一般能够修正错误的输入以及构造一个存在错误节点的文档，所以 `Parse` 很少失败；通常情况下，出错都是由基本的 I/O 错误引起的。

```
gopl.io/ch5/findlinks2

func main() {
    for _, url := range os.Args[1:] {
        links, err := findLinks(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)
            continue
        }
        for _, link := range links {
            fmt.Println(link)
        }
    }
}
```

```
// findLinks 发起一个 HTTP 的 GET 请求，解析返回的 HTML 页面，并返回所有链接
func findLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    return visit(nil, doc), nil
}
```

`findLinks` 函数有 4 个返回语句，每一个语句返回一对值。前 3 个返回语句将函数从 `http` 和 `html` 包中获得的错误信息传递给调用者。第一个返回语句中，错误直接返回；第二个返回语句和第三个返回语句则使用 `fmt.Errorf`（参考 7.8 节）格式化处理过的附加上下文信息。如果 `findLinks` 调用成功，最后一个返回语句将返回链接的 slice，且 `error` 为空。

我们必须保证 `resp.Body` 正确关闭使得网络资源正常释放。即使在发生错误的情况下也必须释放资源。Go 语言的垃圾回收机制将回收未使用的内存，但不能指望它会释放未使用的操作系统资源，比如打开的文件以及网络连接。必须显式地关闭它们。

调用一个涉及多值计算的函数会返回一组值。如果调用者要使用这些返回值，则必须显式地将返回值赋给变量。

```
links, err := findLinks(url)
```

忽略其中一个返回值可以将它赋给一个空标识符。

```
links, _ := findLinks(url) // 忽略的错误
```

返回一个多值结果可以是调用另一个多值返回的函数，就像下面的函数，这个函数的行为和 `findLinks` 类似，只是多了一个记录参数的动作。

```
func findLinksLog(url string) ([]string, error) {
    log.Printf("findLinks %s", url)
    return findLinks(url)
}
```

一个多值调用可以作为单独的实参传递给拥有多个形参的函数中。尽管很少在生产环境使用，但是这个特性有的时候可以方便调试，它使得我们仅仅使用一条语句就可以输出所有的结果。下面两个输出语句的效果是一致的。

```
log.Println(findLinks(url))
```

```
links, err := findLinks(url)
log.Println(links, err)
```

良好的名称可以使得返回值更加有意义。尤其在一个函数返回多个结果且类型相同时，名字的选择更加重要，比如：

```
func Size(rect image.Rectangle) (width, height int)
func Split(path string) (dir, file string)
func HourMinSec(t time.Time) (hour, minute, second int)
```

但不必始终为每个返回值单独命名。比如，习惯上，最后的一个布尔返回值表示成功与否，一个 `error` 结果通常都不需要特别说明。

一个函数如果有命名的返回值，可以省略 `return` 语句的操作数，这称为裸返回。

```
// CountWordsAndImages 发送一个 HTTP GET 请求，并且获取文档的
// 字数与图片数量
func CountWordsAndImages(url string) (words, images int, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        err = fmt.Errorf("parsing HTML: %s", err)
        return
    }
    words, images = countWordsAndImages(doc)
    return
}
func countWordsAndImages(n *html.Node) (words, images int) { /* ... */ }
```

裸返回是将每个命名返回结果按照顺序返回的快捷方法，所以在上面的函数中，每个 `return` 语句都等同于：

```
return words, images, err
```

像在这个函数中存在许多返回语句且有多个返回结果，裸返回可以消除重复代码，但是并不能使代码更加易于理解。比如，对于这种方式，在第一眼看来，不能直观地看出前两个返回等同于 `return 0, 0, err`（因为结果变量 `words` 和 `images` 初始化值为 0）而且最后一个 `return` 等同于 `return words, images, nil`。鉴于这个原因，应保守使用裸返回。

练习 5.5：实现函数 `CountWordsAndImages`（参照练习 4.9 中的单词分隔）。

练习 5.6：修改 `gopl.io/ch3/surface`（参考 3.2 节）中的函数 `corner`，以使用命名的结果以及裸返回语句。

5.4 错误

有一些函数总是成功返回的。比如，`strings.Contains` 和 `strconv.FormatBool` 对所有可能的参数变量都有定义好的返回结果，不会调用失败——尽管还有灾难性的和不可预知的场景，像内存耗尽，这类错误的表现和起因相差甚远而且恢复的希望也很渺茫。

其他的函数只要符合其前置条件就能够成功返回。比如 `time.Date` 函数始终会利用年、月等构成 `time.Time`，但是如果最后一个参数（表示时区）为 `nil` 则会导致宕机。这个宕机标志着这是一个明显的 bug，应该避免这样调用代码。

对于许多其他函数，即使在高质量的代码中，也不能保证一定能够成功返回，因为有些因素并不受程序设计者的掌控。比如任何操作 I/O 的函数都一定会面对可能的错误，只有没有经验的程序员会认为一个简单的读或写不会失败。事实上，这些地方是我们最需要关注的，很多可靠的操作都可能会毫无征兆地发生错误。

因此错误处理是包的 API 设计或者应用程序用户接口的重要部分，发生错误只是许多预料行为中的一种而已。这就是 Go 语言处理错误的方法。

如果当函数调用发生错误时返回一个附加的结果作为错误值，习惯上将错误值作为最后一个结果返回。如果错误只有一种情况，结果通常设置为布尔类型，就像下面这个查询缓存值的例子里面，往往都返回成功，只有不存在对应的键值的时候返回错误：

```
value, ok := cache.Lookup(key)
if !ok {
    // ...cache[key] 不存在...
}
```

更多时候，尤其对于 I/O 操作，错误的原因可能多种多样，而调用者则需要一些详细的信息。在这种情况下，错误的结果类型往往是 `error`。

`error` 是内置的接口类型。第 7 章将通过介绍错误处理揭示更多关于 `error` 类型的深层含义。目前我们已经了解到，一个错误可能是空值或者非空值，空值意味着成功而非空值意味着失败，且非空的错误类型有一个错误消息字符串，可以通过调用它的 `Error` 方法或者通过调用 `fmt.Println(err)` 或 `fmt.Printf("%v", err)` 直接输出错误消息：

一般当一个函数返回一个非空错误时，它其他的结果都是未定义的而且应该忽略。然而，有一些函数在调用出错的情况下会返回部分有用的结果。比如，如果在读取一个文件的时候发生错误，调用 `Read` 函数后返回能够成功读取的字节数与相对应的错误值。正确的行为通常是在调用者处理错误前先处理这些不完整的返回结果。因此在文档中清晰地说明返回值的意义是很重要的。

与许多其他语言不同，Go 语言通过使用普通的值而非异常来报告错误。尽管 Go 语言有异常机制，这将在 5.9 节进行介绍，但是 Go 语言的异常只是针对程序 bug 导致的预料外的错误，而不能作为常规的错误处理方法出现在程序中。

这样做的原因是异常会陷入带有错误消息的控制流去处理它，通常会导致预期外的结果：错误会以难以理解的栈跟踪信息报告给最终用户，这些信息大都是关于程序结构方面的而不是简单明了的错误消息。

相比之下，Go 程序使用通常的控制流机制（比如 `if` 和 `return` 语句）应对错误。这种方式在错误处理逻辑方面要求更加小心谨慎，但这恰恰是设计的要点。

5.4.1 错误处理策略

当一个函数调用返回一个错误时，调用者应当负责检查错误并采取合适的处理应对。根据情形，将有许多可能的处理场景。接下来我们看 5 个例子。

首先也最常见的情形是将错误传递下去，使得在子例程中发生的错误变为主调例程的错误。5.3 节讨论过 `findLinks` 函数的示例。如果调用 `http.Get` 失败，`findLinks` 不做任何操作立即向调用者返回这个 HTTP 错误。

```
resp, err := http.Get(url)
if err != nil {
    return nil, err
}
```

对比之下，如果调用 `html.Parse` 失败，`findLinks` 将不会直接返回 HTML 解析的错误，因为它缺失两个关键信息：解析器的出错信息与被解析文档的 URL。在这种情况下，`findLinks` 构建一个新的错误消息，其中包含我们需要的所有相关信息和解析的错误信息：

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
```

```

if err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
}

```

`fmt.Errorf` 使用 `fmt.Sprintf` 函数格式化一条错误消息并且返回一个新的错误值。我们为原始的错误消息不断地添加额外的上下文信息来建立一个可读的错误描述。当错误最终被程序的 `main` 函数处理时，它应当能够提供一个从最根本问题到总体故障的清晰因果链，这让我想到 NASA 的事故调查有这样一个例子：

```
genesis: crashed: no parachute: G-switch failed: bad relay orientation
```

因为错误消息频繁地串联起来，所以消息字符串首字母不应该大写而且应该避免换行。错误结果可能会很长，但能够使用 `grep` 这样的工具找到我们需要的信息。

设计一个错误消息的时候应当慎重，确保每一条消息的描述都是有意义的，包含充足的相关信息，并且保持一致性，不论被同一个函数还是同一个包下面的一组函数返回时，这样的错误都可以保持统一的形式和错误处理方式。

比如，`os` 包保证每一个文件操作（比如 `os.Open` 或针对打开的文件的 `Read`、`Write` 或 `Close` 方法）返回的错误不仅包括错误的信息（没有权限、路径不存在等）还包含文件的名字，因此调用者在构造错误消息的时候不需要再包含这些信息。

一般地，`f(x)` 调用只负责报告函数的行为 `f` 和参数值 `x`，因为它们和错误的上下文相关。调用者负责添加进一步的信息，但是 `f(x)` 本身并不会，就像上面函数中 `URL` 和 `html.Parse` 的关系。

我们接下来看一下第二种错误处理策略。对于不固定或者不可预测的错误，在短暂的间隔后对操作进行重试是合乎情理的，超出一定的重试次数和限定的时间后再报错退出。

```

gopl.io/ch5/wait

// WaitForServer 尝试连接URL对应的服务器
// 在一分钟内使用指数退避策略进行重试
// 所有的尝试失败后返回错误
func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil // 成功
        }
        log.Printf("server not responding (%s); retrying...", err)
        time.Sleep(time.Second << uint(tries)) // 指数退避策略
    }
    return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}

```

第三，如果依旧不能顺利进行下去，调用者能够输出错误然后优雅地停止程序，但一般这样的处理应该留给主程序部分。通常库函数应当将错误传递给调用者，除非这个错误表示一个内部一致性错误，这意味着库内部存在 bug。

```

// (In function main.)
if err := WaitForServer(url); err != nil {
    fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
    os.Exit(1)
}

```

一个更加方便的方法是通过调用 `log.Fatalf` 实现相同的效果。就和所有的日志函数一样，它默认会将时间和日期作为前缀添加到错误消息前。

```
if err := WaitForServer(url); err != nil {
    log.Fatalf("Site is down: %v\n", err)
}
```

默认的格式有助于长期运行的服务器，而对于交互式的命令行工具则意义不大：

```
2006/01/02 15:04:05 Site is down: no such domain: bad.gopl.io
```

一种更吸引人的输出方式是自己定义命令的名称作为 `log` 包的前缀，并且将日期和时间略去。

```
log.SetPrefix("wait: ")
log.SetFlags(0)
```

第四，在一些错误情况下，只记录下错误信息然后程序继续运行。同样地，可以选择使用 `log` 包来增加日志的常用前缀：

```
if err := Ping(); err != nil {
    log.Printf("ping failed: %v; networking disabled", err)
}
```

并且直接输出到标准错误流：

```
if err := Ping(); err != nil {
    fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled\n", err)
}
```

(所有 `log` 函数都会为缺少换行符的日志补充一个换行符。)

第五，在某些罕见的情况下我们可以直接安全地忽略掉整个日志：

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
    return fmt.Errorf("failed to create temp dir: %v", err)
}

// ...使用临时目录...

os.RemoveAll(dir) //忽略错误，$TMPDIR 会被周期性删除
```

调用 `os.RemoveAll` 可能会失败，但程序忽略了这个错误，原因是操作系统会周期性地清理临时目录。在这个例子中，我们有意地抛弃了错误，但程序的逻辑看上去就和我们忘记去处理了一样。要习惯考虑到每一个函数调用可能发生的出错情况，当你有意地忽略一个错误的时候，清楚地注释一下你的意图。

Go 语言的错误处理有特定的规律。进行错误检查之后，检测到失败的情况往往都在成功之前。如果检测到的失败导致函数返回，成功的逻辑一般不会放在 `else` 块中而是在外层的作用域中。函数会有一种通常的形式，就是在开头有一连串的检查用来返回错误，之后跟着实际的函数体一直到最后。

5.4.2 文件结束标识

通常，最终用户会对函数返回的多种错误感兴趣而不是中间涉及的程序逻辑。偶尔，一个程序必须针对不同各种类的错误采取不同的措施。考虑如果要从一个文件中读取 n 个字节的数据。如果 n 是文件本身的长度，任何错误都代表操作失败。另一方面，如果调用者反复

地尝试读取固定大小的块直到文件耗尽，调用者必须把读取到文件尾的情况区别于遇到其他错误的操作。为此，`io` 包保证任何由文件结束引起的读取错误，始终都将会得到一个与众不同的错误——`io.EOF`，它的定义如下：

```
package io

import "errors"

//当没有更多输入时，将会返回EOF
var EOF = errors.New("EOF")
```

调用者可以使用一个简单的比较操作来检测这种情况，在下面的循环中，不断从标准输入中读取字符。（4.3 节的 `charcount` 程序提供了一个更完整的示例。）

```
in := bufio.NewReader(os.Stdin)
for {
    r, _, err := in.ReadRune()
    if err == io.EOF {
        break // 结束读取
    }
    if err != nil {
        return fmt.Errorf("read failed: %v", err)
    }
    // ...使用 r...
}
```

除了反映这个实际情况外，因为文件结束的条件没有其他信息，所以 `io.EOF` 有一条固定的错误消息 "`EOF`"。对于其他错误，我们可能需要同时得到错误相关的本质原因和数量信息，因此一个固定的错误值并不能满足我们的需求。7.11 节将会呈现一个更加系统的方式以区分某个错误值。

5.5 函数变量

函数在 Go 语言中是头等重要的值：就像其他值，函数变量也有类型，而且它们可以赋给变量或者传递或者从其他函数中返回。函数变量可以像其他函数一样调用。比如：

```
func square(n int) int      { return n * n }
func negative(n int) int    { return -n }
func product(m, n int) int { return m * n }

f := square
fmt.Println(f(3)) // "9"

f = negative
fmt.Println(f(3))    // "-3"
fmt.Printf("%T\n", f) // "func(int) int"

f = product //编译错误：不能把类型 func(int, int) int 赋给 func(int) int
```

函数类型的零值是 `nil`（空值），调用一个空的函数变量将导致宕机。

```
var f func(int) int
f(3) // 宕机：调用空函数
```

函数变量可以和空值相比较：

```
var f func(int) int
if f != nil {
    f(3)
}
```

但它们本身不可比较，所以不可以互相进行比较或者作为键值出现在 map 中。

函数变量使得函数不仅将数据进行参数化，还将函数的行为当作参数进行传递。标准库中蕴含者大量的例子。比如，`strings.Map` 对字符串中的每一个字符使用一个函数，将结果连接起来变成另一个字符串。

```
func add1(r rune) rune { return r + 1 }

fmt.Println(strings.Map(add1, "HAL-9000")) // "IBM.:111"
fmt.Println(strings.Map(add1, "VMS"))        // "WNT"

fmt.Println(strings.Map(add1, "Admix"))      // "Benjy"
```

5.2 节中的 `findLinks` 函数使用了一个辅助函数 `visit`，它访问 HTML 文档中所有的节点而后对每一个节点进行操作。使用函数变量，可以使得我们将每个节点的操作逻辑从遍历树形结构的逻辑中分开。下面通过不同的操作重用该遍历逻辑。

gopl.io/ch5/outline2

```
// forEachNode 调用 pre(x) 和 post(x) 遍历以n为根的树中的每个节点x
// 两个函数是可选的
// pre 在子节点被访问前（前序）调用
// post 在访问后（后序）调用
func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }

    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }

    if post != nil {
        post(n)
    }
}
```

这里 `forEachNode` 函数接受两个函数作为参数，一个在本节点所有子节点都被访问前调用，另一个则在之后。这样的代码组织给调用者提供了很多的灵活性。比如，函数 `startElement` 和 `endElement` 输出 HTML 元素的起始和结束标签，如 `…`：

```
var depth int

func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth*2, "", n.Data)
        depth++
    }
}

func endElement(n *html.Node) {
    if n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s</%s>\n", depth*2, "", n.Data)
    }
}
```

这两个函数巧妙地利用 `fmt.Printf` 来缩进输出。`%*s` 中的 * 号输出带有可变数量空格的字符串。输出的宽度和字符串则由参数 `depth*2` 和 `""` 提供。

如果对 HTML 文档调用 `forEachNode` 函数，比如：

```
forEachNode(doc, startElement, endElement)
```

可以使之前的 `outline` 函数得到一个更加直观的输出。

```
$ go build gopl.io/ch5/outline2
$ ./outline2 http://gopl.io
<html>
  <head>
    <meta>
    </meta>
    <title>
    </title>
    <style>
    </style>
  </head>
  <body>
    <table>
      <tbody>
        <tr>
          <td>
            <a>
              <img>
            </img>
          ...
        ...
      ...
    ...
  ...
...

```

练习 5.7：开发 `startElement` 和 `endElement` 函数并应用到一个通用的 HTML 输出代码中。输出注释节点、文本节点和所有元素属性 (``)。当一个元素没有子节点时，使用简短的形式，比如 `` 而不是 ``。写一个测试程序保证输出可以正确解析（参考第 11 章）。

练习 5.8：修改 `forEachNode` 使得 `pre` 和 `post` 函数返回一个布尔型的结果来确定遍历是否继续下去。使用它写一个函数 `ElementByID`，该函数使用下面的函数签名并且找到第一个符合 `id` 属性的 HTML 元素。函数在找到符合条件的元素时应该尽快停止遍历。

```
func ElementByID(doc *html.Node, id string) *html.Node
```

练习 5.9：写一个函数 `expand(s string, f func(string)string)`，该函数替换参数 `s` 中每一个子字符串 `"$foo"` 为 `f("foo")` 的返回值。

5.6 匿名函数

命名函数只能在包级别的作用域进行声明，但我们能够使用函数字面量在任何表达式内指定函数变量。函数字面量就像函数声明，但在 `func` 关键字后面没有函数的名称。它是一个表达式，它的值称作匿名函数。

函数字面量在我们需要使用的时候才定义。就像下面这个例子，之前的函数调用 `strings.Map` 可以写成：

```
strings.Map(func(r rune) rune { return r + 1 }, "HAL-9000")
```

更重要的是，以这种方式定义的函数能够获取到整个词法环境，因此里层的函数可以使外层函数中的变量，如下面这个示例所示：

[gopl.io/ch5/squares](#)

```
// squares 函数返回一个函数，后者包含下一次要用到的平方数
// the next square number each time it is called.
```

```
func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}

func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```

函数 `squares` 返回了另一个函数，类型是 `func() int`。调用 `squares` 创建了一个局部变量 `x` 而且返回了一个匿名函数，每次调用 `squares` 都会递增 `x` 的值然后返回 `x` 的平方。第二次调用 `squares` 函数将创建第二个变量 `x`，然后返回一个递增 `x` 值的新匿名函数。

这个求平方的示例演示了函数变量不仅是一段代码还可以拥有状态。里层的匿名函数能够获取和更新外层 `squares` 函数的局部变量。这些隐藏的变量引用就是我们把函数归类为引用类型而且函数变量无法进行比较的原因。函数变量类似于使用闭包方法实现的变量，Go 程序员通常把函数变量称为闭包。

我们再一次看到这个例子里面变量的生命周期不是由它的作用域所决定的：变量 `x` 在 `main` 函数中返回 `squares` 函数后依旧存在（虽然 `x` 在这个时候是隐藏在函数变量 `f` 中的）。

在下面这个与学术课程相关的匿名函数例程中，考虑学习计算机科学课程的顺序，需要计算出学习每一门课程的先决条件。先决课程在下面的 `prereqs` 表中已经给出，其中给出了学习每一门课程必须提前完成的课程列表关系。

gopl.io/ch5/toposort

这样的问题是我们所熟知的拓扑排序。概念上，先决条件的内容构成一张有向图，每一个节点代表每一门课程，每一条边代表一门课程所依赖另一门课程的关系。图是无环的：没有节点可以通过图上的路径回到它自己。我们可以使用深度优先的搜索算法计算得到合法的学习路径，如以下代码所示：

```

func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}

func topoSort(m map[string][]string) []string {
    var order []string
    seen := make(map[string]bool)
    var visitAll func(items []string)
    visitAll = func(items []string) {
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                visitAll(m[item])
                order = append(order, item)
            }
        }
    }
    var keys []string
    for key := range m {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    visitAll(keys)
    return order
}

```

当一个匿名函数需要进行递归，在这个例子中，必须先声明一个变量然后将匿名函数赋给这个变量。如果将两个步骤合并成一个声明，函数字面量将不能存在于 `visitAll` 变量的作用域中，这样也就不能递归地调用自己了。

```

visitAll := func(items []string) {
    // ...
    visitAll(m[item]) // compile error: undefined: visitAll
    // ...
}

```

下面是拓扑排序的程序输出。它是确定的结果，得到令人满意的结果不容易。在这里，`prereqs` 的值都是 slice 而不是 map，所以它们的迭代顺序是确定的并且我们在调用最初的 `visitAll` 之前将 `prereqs` 的键值进行了排序。

```

1:      intro to programming
2:      discrete math
3:      data structures
4:      algorithms
5:      linear algebra
6:      calculus
7:      formal languages
8:      computer organization
9:      compilers
10:     databases
11:     operating systems
12:     networks
13:     programming languages

```

回到 `findLinks` 例子。由于在第 8 章还需要用到它，因此我们将解析链接的函数 `links.Extract` 移动到它自己的包中。我们将原本的 `visit` 函数替换为匿名函数，并直接放到存放链

接的 slice 之后，然后用 `forEachNode` 处理递归。因为 `Extract` 函数只需要 `pre` 函数，所以把 `post` 部分的参数填 `nil`。

```
gopl.io/ch5/links
// link 包提供了解析链接的函数
package links

import (
    "fmt"
    "net/http"

    "golang.org/x/net/html"
)

// Extract 函数向给定 URL 发起 HTTP GET 请求
// 解析 HTML 并返回 HTML 文档中存在的链接
func Extract(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }

    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }

    var links []string
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "a" {
            for _, a := range n.Attr {
                if a.Key != "href" {
                    continue
                }
                link, err := resp.Request.URL.Parse(a.Val)
                if err != nil {
                    continue // 忽略不合法的 URL
                }
                links = append(links, link.String())
            }
        }
    }
    forEachNode(doc, visitNode, nil)
    return links, nil
}
```

在这个版本里，我们并不是直接把 `href` 原封不动地添加到存放链接的 slice 中，而是将它解析成基于当前文档的相对路径 `resp.Request.URL`。结果的链接是绝对路径的形式，非常适用于调用函数 `http.Get`。

网页爬虫的核心是解决图的遍历。拓扑排序的示例展示了深度优先遍历；对于网络爬虫，我们使用广度优先遍历。第 8 章将探索并发遍历。

下面的示例函数展示了广度优先遍历的精髓。调用者提供一个初始列表 `worklist`，它包含要访问的项和一个函数变量 `f` 用来处理每一个项。每一个项用字符串来识别。函数 `f` 将返回一个新的项列表，其中包含需要新添加到 `worklist` 中的项。`breadthFirst` 函数将在所有节

点项都被访问后返回。它需要维护一个字符串集合用来保证每个节点只访问一次。

```
gopl.io/ch5/findlinks3

// breadthFirst 对每个worklist元素调用f
// 并将返回的内容添加到worklist中，对每一个元素，最多调用一次f
// f is called at most once for each item.
func breadthFirst(f func(item string) []string, worklist []string) {
    seen := make(map[string]bool)
    for len(worklist) > 0 {
        items := worklist
        worklist = nil
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                worklist = append(worklist, f(item)...)
            }
        }
    }
}
```

就像第 4 章介绍过的，参数 “`f(item)...`” 将会把 `f` 返回的列表中的所有项添加到 `worklist` 中。

在爬虫里，项节点都是 URL。我们提供 `crawl` 函数给 `breadthFirst` 以输出 URL，解析链接然后将它们返回，标记为已访问。

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

为了让爬虫开始工作，我们使用命令行参数指定开始的 URL。

```
func main() {
    // 开始广度遍历
    // 从命令行参数开始
    breadthFirst(crawl, os.Args[1:])
}
```

我们从 `https://golang.org` 开始爬网页。这里是一些输出的链接：

```
$ go build gopl.io/ch5/findlinks3
$ ./findlinks3 https://golang.org
https://golang.org/
https://golang.org/doc/
https://golang.org/pkg/
https://golang.org/project/
https://code.google.com/p/go-tour/
https://golang.org/doc/code.html
https://www.youtube.com/watch?v=XCsl89YtqCs
http://research.swtch.com/gotour
https://vimeo.com/53221560
...
```

整个过程将在所有可到达的网页被访问到或者内存耗尽时结束。

练习 5.10：重写 `topoSort` 以使用 `map` 代替 `slice` 并去掉开头的排序。结果不是唯一的，验证这个结果是合法的拓扑排序。

练习 5.11：现在有“线性代数”(linear algebra)这门课程，它的先决课程是微积分(calculus)。扩展 topoSort 以函数输出结果。

练习 5.12：5.5 节([gopl.io/ch5/outline2](#))的 startElement 和 endElement 函数共享一个全局变量 depth。把它们变为匿名函数以共享 outline 函数的一个局部变量。

练习 5.13：修改 crawl 函数保存找到的页面，根据需要创建目录。不要保存不同域名下的页面。比如，如果本来的页面来自 [golang.org](#)，那么就把它们保存下来但是不要保存 [vimeo.com](#) 下的页面。

练习 5.14：使用广度优先遍历搜索一个不同的拓扑结构。比如，你可以借鉴拓扑排序的例子(有向图)里的课程依赖关系，计算机文件系统的分层结构(树形结构)，或者从当前城市的官方网站上下载公共汽车或者地铁线路图(无向图)。

警告：捕获迭代变量

在这一节，我们将看到 Go 语言的词法作用域规则的陷阱，有时会得到令你吃惊的结果。我们强烈建议你先理解这个问题再进行下一节的阅读，因为即使是有经验的程序员也会掉入这些陷阱。

假设一个程序必须创建一系列的目录之后又会删除它们。可以使用一个包含函数变量的 slice 进行清理操作。(这个示例中省略了所有的错误处理逻辑。)

```
var rmdir []func()
for _, d := range tempDirs() {
    dir := d // 注意，这一行是必需的
    os.MkdirAll(dir, 0755) // 也创建父目录
    rmdir = append(rmdir, func() {
        os.RemoveAll(dir)
    })
}
// ...这里做一些处理...
for _, rmdir := range rmdir {
    rmdir() // 清理
}
```

你可能会奇怪，为什么在循环体内将循环变量赋给一个新的局部变量 dir，而不是在下面这个略有错误的变体中直接使用循环变量 dir。

```
var rmdir []func()
for _, dir := range tempDirs() {
    os.MkdirAll(dir, 0755)
    rmdir = append(rmdir, func() {
        os.RemoveAll(dir) // 不正确
    })
}
```

这个原因是循环变量的作用域的规则限制。在上面的程序中，dir 在 for 循环引进的一个块作用域内进行声明。在循环里创建的所有函数变量共享相同的变量——一个可访问的存储位置，而不是固定的值。dir 变量的值在不断地迭代中更新，因此当调用清理函数时，dir 变量已经被每一次的 for 循环更新多次。因此，dir 变量的实际取值是最后一次迭代时的值并且所有的 os.RemoveAll 调用最终都试图删除同一个目录。

我们经常引入一个内部变量来解决这个问题，就像 dir 变量是一个和外部变量同名的变

量，只不过是一个副本，这看起来有些奇怪却是一个关键性的声明：

```
for _, dir := range tempDirs() {
    dir := dir // 声明内部 dir，并以外部 dir 初始化
    // ...
}
```

这样的隐患不仅仅存在于使用 `range` 的 `for` 循环里。在下面的循环中也面临由于无意间捕获的索引变量 `i` 而导致的同样问题。

```
var rmdir []func()
dirs := tempDirs()
for i := 0; i < len(dirs); i++ {
    os.MkdirAll(dirs[i], 0755) // OK
    rmdir = append(rmdir, func() {
        os.RemoveAll(dirs[i]) // 不正确
    })
}
```

在 `go` 语句（参考第 8 章）和 `defer` 语句（稍后会看到）的使用当中，迭代变量捕获的问题是最频繁的，这是因为这两个逻辑都会推迟函数的执行时机，直到循环结束。但是这个问题并不是由 `go` 或者 `defer` 语句造成的。

5.7 变长函数

变长函数被调用的时候可以有可变的参数个数。最令人熟知的例子就是 `fmt.Printf` 与其变种。`Printf` 需要在开头提供一个固定的参数，后续便可以接受任意数目的参数。

在参数列表最后的类型名称之前使用省略号 “`...`” 表示声明一个变长函数，调用这个函数的时候可以传递该类型任意数目的参数。

```
gopl.io/ch5/sum
func sum(vals ...int) int {
    total := 0
    for _, val := range vals {
        total += val
    }
    return total
}
```

上面这个 `sum` 函数返回零个或者多个 `int` 参数。在函数体内，`vals` 是一个 `int` 类型的 `slice`。调用 `sum` 的时候任何数量的参数都将提供给 `vals` 参数。

```
fmt.Println(sum())          // "0"
fmt.Println(sum(3))         // "3"
fmt.Println(sum(1, 2, 3, 4)) // "10"
```

调用者显式地申请一个数组，将实参复制给这个数组，并把一个数组 `slice` 传递给函数。上面的最后一个调用和下面的调用的作用是一样的，它展示了当实参已经存在于一个 `slice` 中的时候如何调用一个变长函数：在最后一个参数后面放一个省略号。

```
values := []int{1, 2, 3, 4}
fmt.Println(sum(values...)) // "10"
```

尽管 `...int` 参数就像函数体内的 `slice`，但变长函数的类型和一个带有普通 `slice` 参数的函数的类型不相同。

```
func f(...int) {}
func g([]int) {}
```

```
fmt.Printf("%T\n", f) // "func(...int)"
fmt.Printf("%T\n", g) // "func([]int)"
```

变长函数通常用于格式化字符串。下面的 `errorf` 函数构建一条格式化的错误消息，在消息的开头带有行号。函数的后缀 `f` 是广泛使用的命名习惯，用于可变长 `Printf` 风格的字符串格式化输出函数。

```
func errorf(linenum int, format string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, "Line %d: ", linenum)
    fmt.Fprintf(os.Stderr, format, args...)
    fmt.Println(os.Stderr)
}

linenum, name := 12, "count"
errorf(linenum, "undefined: %s", name) // "Line 12: undefined: count"
```

`interface{}` 类型意味着这个函数的最后一个参数可以接受任何值，第 7 章将解释它的用法。

练习 5.15：模仿 `sum` 写两个变长函数 `max` 和 `min`。当不带任何参数调用这些函数时应该怎么应对？编写类似函数的变种，要求至少需要一个参数。

练习 5.16：写一个变长版本的 `strings.Join` 函数。

练习 5.17：写一个变长函数 `ElementsByTagName`，已知一个 HTML 节点树和零个或多个名字，返回所有符合给出名字的元素。下面有两个示例调用：

```
func ElementsByName(doc *html.Node, name ...string) []*html.Node
images := ElementsByName(doc, "img")
headings := ElementsByName(doc, "h1", "h2", "h3", "h4")
```

5.8 延迟函数调用

`findLinks` 示例使用 `http.Get` 的输出作为 `html.Parse` 的输入。如果请求的 URL 是 HTML 那么它一定会正常工作，但是许多页面包含图片、文字和其他文件格式。如果让 HTML 解析器去解析这类文件可能会发生意料外的状况。

下面的程序获取一个 HTML 文档然后输出它的标题。`title` 函数检测从服务器端回的 `Content-Type` 头部，如果文档不是 HTML 则返回错误。

```
gopl.io/ch5/title1

func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }

    // 检查 Content-Type 是 HTML (如 "text/html; charset=utf-8")
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        resp.Body.Close()
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }

    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
```

```

doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return fmt.Errorf("parsing %s as HTML: %v", url, err)
}

visitNode := func(n *html.Node) {
    if n.Type == html.ElementNode && n.Data == "title" &&
        n.FirstChild != nil {
        fmt.Println(n.FirstChild.Data)
    }
}
forEachNode(doc, visitNode, nil)
return nil
}

```

下面是稍稍编辑后的命令行会话示例：

```

$ go build gopl.io/ch5/title1
$ ./title1 http://gopl.io
The Go Programming Language
$ ./title1 https://golang.org/doc/effective_go.html
Effective Go - The Go Programming Language
$ ./title1 https://golang.org/doc/gopher/frontpage.png
title: https://golang.org/doc/gopher/frontpage.png
    has type image/png, not text/html

```

观察重复的 `resp.Body.Close()` 调用，它保证 `title` 函数在任何执行路径下都会关闭网络连接，包括发生错误的情况。随着函数变得越来越复杂，并且需要处理更多的错误情况，这样一种重复的清理动作会造成之后的维护问题。我们看看 Go 语言的 `defer` 机制怎样让这些工作变得更简单。

语法上，一个 `defer` 语句就是一个普通的函数或方法调用，在调用之前加上关键字 `defer`。函数和参数表达式会在语句执行时求值，但是无论是正常情况下，执行 `return` 语句或函数执行完毕，还是不正常的情况下，比如发生宕机，实际的调用推迟到包含 `defer` 语句的函数结束后才执行。`defer` 语句没有限制使用次数；执行的时候以调用 `defer` 语句顺序的倒序进行。

`defer` 语句经常使用于成对的操作，比如打开和关闭，连接和断开，加锁和解锁，即使是最复杂的控制流，资源在任何情况下都能够正确释放。正确使用 `defer` 语句的地方是在成功获得资源之后。在下面的 `title` 函数，一个推迟的调用替换了先前的 `resp.Body.Close()` 调用：

```

gopl.io/ch5/title2
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }

    doc, err := html.Parse(resp.Body)
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
}

```

```
// ...输出文档的标题元素...
return nil
}
```

同样的方法可以使用在其他资源（包括网络连接）上，比如关闭一个打开的文件：

```
io/ioutil
package ioutil

func ReadFile(filename string) ([]byte, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    return ReadAll(f)
}
```

或者解锁一个互斥锁（参考 9.2 节）：

```
var mu sync.Mutex
var m = make(map[string]int)
func lookup(key string) int {
    mu.Lock()
    defer mu.Unlock()
    return m[key]
}
```

`defer` 语句也可以用来调试一个复杂的函数，即在函数的“入口”和“出口”处设置调试行为。下面的 `bigSlowOperation` 函数在开头调用 `trace` 函数，在函数刚进入的时候执行输出，然后返回一个函数变量，当其被调用的时候执行退出函数的操作。以这种方式推迟返回函数的调用，我们可以使用一个语句在函数入口和所有出口添加处理，甚至可以传递一些有用的值，比如每个操作的开始时间。但别忘了 `defer` 语句末尾的圆括号，否则入口的操作会在函数退出时执行而出口的操作永远不会调用！

```
gopl.io/ch5/trace
func bigSlowOperation() {
    defer trace("bigSlowOperation")() // 别忘记这对圆括号
    // ...这里是一些处理...
    time.Sleep(10 * time.Second) // 通过休眠仿真慢操作
}

func trace(msg string) func() {
    start := time.Now()
    log.Printf("enter %s", msg)
    return func() { log.Printf("exit %s (%s)", msg, time.Since(start)) }
}
```

每次调用 `bigSlowOperation`，它会记录进入函数入口和出口的时间与两者之间的时间差。（我们使用 `time.Sleep` 来模拟一个长时间的操作。）

```
$ go build gopl.io/ch5/trace
$ ./trace
2015/11/18 09:53:26 enter bigSlowOperation
2015/11/18 09:53:36 exit bigSlowOperation (10.000589217s)
```

延迟执行的函数在 `return` 语句之后执行，并且可以更新函数的结果变量。因为匿名函数可以得到其外层函数作用域内的变量（包括命名的结果），所以延迟执行的匿名函数可以

观察到函数的返回结果。

考虑下面的函数 `double`:

```
func double(x int) int {
    return x + x
}
```

通过命名结果变量和增加 `defer` 语句，我们能够在每次调用函数的时候输出它的参数和结果。

```
func double(x int) (result int) {
    defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()
    return x + x
}

_ = double(4)
// 输出:
// "double(4) = 8"
```

这个技巧的使用相比之前的 `double` 函数来说有些过了，但对于有很多返回语句的函数来说很有帮助。

延迟执行的匿名函数能够改变外层函数返回给调用者的结果：

```
func triple(x int) (result int) {
    defer func() { result += x }()
    return double(x)
}

fmt.Println(triple(4)) // "12"
```

因为延迟的函数不到函数的最后一刻是不会执行的。要注意循环里 `defer` 语句的使用。下面的这段代码就可能会用尽所有的文件描述符，这是因为处理完成后却没有文件关闭。

```
for _, filename := range filenames {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close() // 注意：可能会用尽文件描述符
    // ...处理文件 f...
}
```

一种解决的方式是将循环体（包括 `defer` 语句）放到另一个函数里，每此循环迭代都会调用文件关闭函数。

```
for _, filename := range filenames {
    if err := doFile(filename); err != nil {
        return err
    }
}

func doFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    // ...处理文件 f...
}
```

下面这个例子是改进过的 `fetch` 程序（参见 1.5 节），将 HTTP 的响应写到本地文件中而

不是直接显示在标准输出中。它使用 `path.Base` 函数获得 URL 路径最后的一个组成部分作为文件名。

```
gopl.io/ch5/fetch
// Fetch 下载 URL 并返回本地文件的名字和长度
func fetch(url string) (filename string, n int64, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
    }
    defer resp.Body.Close()

    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }
    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }
    n, err = io.Copy(f, resp.Body)
    // 关闭文件，并保留错误消息
    if closeErr := f.Close(); err == nil {
        err = closeErr
    }
    return local, n, err
}
```

现在应该熟悉延迟调用的 `resp.Body.Close` 了。在这个例程中，如果试图使用延迟调用 `f.Close` 去关闭一个本地文件就会有些问题，因为 `os.Create` 打开了一个文件对其进行写入、创建。在许多文件系统中，尤其是 NFS，写错误往往不是立即返回而是推迟到文件关闭的时候。如果无法检查关闭操作的结果，就会导致一系列的数据丢失。然而，如果 `io.Copy` 和 `f.Close` 同时失败，我们更加倾向于报告 `io.Copy` 的错误，因为它发生在前，更有可能告诉我们失败的原因是什么。

练习 5.18：不改变原本的行为，重写 `fetch` 函数以使用 `defer` 语句关闭打开的可写的文件。

5.9 宕机

Go 语言的类型系统会捕获许多编译时错误，但有些其他的错误（比如数组越界访问或者解引用空指针）都需要在运行时进行检查。当 Go 语言运行时检测到这些错误，它就会发生宕机。

一个典型的宕机发生时，正常的程序执行会终止，goroutine 中的所有延迟函数会执行，然后程序会异常退出并留下一条日志消息。日志消息包括宕机的值，这往往代表某种错误消息，每一个 goroutine 都会在宕机的时候显示一个函数调用的栈跟踪消息。通常可以借助这条日志消息来诊断问题的原因而不需要再一次运行该程序，因此报告一个发生宕机的程序 bug 时，总是会加上这条消息。

并不是所有宕机都是在运行时发生的。可以直接调用内置的宕机函数；内置的宕机函数可以接受任何值作为参数。如果碰到“不可能发生”的状况，宕机是最好的处理方式，比如语句执行到逻辑上不可能到达的地方时：

```

switch s := suit(drawCard()); s {
case "Spades": // ...
case "Hearts": // ...
case "Diamonds": // ...
case "Clubs": // ...
default:
    panic(fmt.Sprintf("invalid suit %q", s)) // 宕机了吗
}

```

设置函数的断言是一个良好的习惯，但是这也会带来多余的检查。除非你能够提供有效的错误消息或者能够很快地检测出错误，否则在运行时检测断言条件就毫无意义。

```

func Reset(x *Buffer) {
    if x == nil {
        panic("x is nil") // 没必要
    }
    x.elements = nil
}

```

尽管 Go 语言的宕机机制和其他语言的异常很相似，但宕机的使用场景不尽相同。由于宕机会引起程序异常退出，因此只有在发生严重的错误时才会使用宕机，比如遇到与预想的逻辑不一致的代码；用心的程序员会将所有可能会发生异常退出的情况考虑在内以证实 bug 的存在。强健的代码会优雅地处理“预期的”错误，比如错误的输入、配置或者 I/O 失败等；这时最好能够使用错误值来加以区分。

考虑函数 `regexp.Compile`，它编译了一个高效的正则表达式。如果调用时给的模式参数不合法则会报错，但是检查这个错误本身没有必要且相当烦琐，因为调用者知道这个特定的调用是不会失败的。在此情况下，使用宕机来处理这种不可能发生的错误是比较合理的。

由于大部分的正则表达式是字面量，因此 `regexp` 包提供了一个包装函数 `regexp.MustCompile` 进行这个检查：

```

package regexp

func Compile(expr string) (*Regexp, error) { /* ... */ }
func MustCompile(expr string) *Regexp {
    re, err := Compile(expr)
    if err != nil {
        panic(err)
    }
    return re
}

```

包装函数使得初始化一个包级别的正则表达式变量（带有一个编译的正则表达式）变得更加方便，如下所示：

```
var httpSchemeRE = regexp.MustCompile(`^https?:`) // "http:"或"https:"
```

当然，`MustCompile` 不应该接收到不正确的值。前缀 `Must` 是这类函数一个通用的命名习惯，比如 4.6 节介绍的 `template.Must`。

当宕机发生时，所有的延迟函数以倒序执行，从栈最上面的函数开始一直返回至 `main` 函数，如下面的程序所示：

```

gopl.io/ch5/defer1

func main() {
    f(3)
}

```

```
func f(x int) {
    fmt.Printf("f(%d)\n", x+0/x) // panics if x ==0则发生宕机
    defer fmt.Printf("defer %d\n", x)
    f(x - 1)
}
```

运行的时候，程序会输出下面的内容到标准输出。

```
f(3)
f(2)
f(1)
defer 1
defer 2
defer 3
```

当调用 `f(0)` 的时候会发生宕机，会执行三个延迟的 `fmt.Printf` 调用。之后，运行时终止了这个程序，输出宕机消息与一个栈转储信息到标准错误流（输出内容有省略）。

```
panic: runtime error: integer divide by zero
main.f(0)
    src/gopl.io/ch5/defer1/defer.go:14
main.f(1)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(2)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(3)
    src/gopl.io/ch5/defer1/defer.go:16
main.main()
    src/gopl.io/ch5/defer1/defer.go:10
```

之后会看到，函数是可以从宕机状态恢复至正常运行状态而不让程序退出。

`runtime` 包提供了转储栈的方法使程序员可以诊断错误。下面代码在 `main` 函数中延迟 `printStack` 的执行：

```
gopl.io/ch5/defer2
func main() {
    defer printStack()
    f(3)
}

func printStack() {
    var buf [4096]byte
    n := runtime.Stack(buf[:], false)
    os.Stdout.Write(buf[:n])
}
```

下面的额外信息（同样经过简化处理）输出到标准输出中：

```
goroutine 1 [running]:
main.printStack()
    src/gopl.io/ch5/defer2/defer.go:20
main.f(0)
    src/gopl.io/ch5/defer2/defer.go:27
main.f(1)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(2)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(3)
    src/gopl.io/ch5/defer2/defer.go:29
main.main()
    src/gopl.io/ch5/defer2/defer.go:15
```