

熟悉其他语言的异常机制的读者可能会对 `runtime.Stack` 能够输出函数栈信息感到吃惊，因为栈应该已经不存在了。但事实上，Go 语言的宕机机制让延迟执行的函数在栈清理之前调用。

5.10 恢复

退出程序通常是正确处理宕机的方式，但也有例外。在一定情况下是可以进行恢复的，至少有时候可以在退出前理清当前混乱的情况。比如，当 Web 服务器遇到一个未知错误时，可以先关闭所有连接，这总比让客户端阻塞在那里要好，而在开发过程中，也可以向客户端汇报当前遇到的错误。

如果内置的 `recover` 函数在延迟函数的内部调用，而且这个包含 `defer` 语句的函数发生宕机，`recover` 会终止当前的宕机状态并且返回宕机的值。函数不会从之前宕机的地方继续运行而是正常返回。如果 `recover` 在其他任何情况下运行则它没有任何效果且返回 `nil`。

为了说明这一点，假设我们开发一种语言的解析器。即使它看起来运行正常，但考虑到工作的复杂性，还是会存在只在特殊情况下发生的 bug。我们在这时会更喜欢将本该宕机的错误看作一个解析错误，不要立即终止运行，而是将一些有用的附加消息提供给用户来报告这个 bug。

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // ...解析器...
}
```

`Parse` 函数中的延迟函数会从宕机状态恢复，并使用宕机值组成一条错误消息；理想的写法是使用 `runtime.Stack` 将整个调用栈包含进来。延迟函数则将错误赋给 `err` 结果变量，从而返回给调用者。

对于宕机采用无差别的恢复措施是不可靠的，因为宕机后包内变量的状态往往没有清晰的定义和解释。可能是对某个关键数据结构的更新错误，文件或网络连接打开而未关闭，或者获得了锁却没有释放。长此以往，把异常退出变为简单地输出一条日志会使真正的 bug 难于发现。

从同一个包内发生的宕机进行恢复有助于简化处理复杂和未知的错误，但一般的原则是，你不应该尝试去恢复从另一个包内发生的宕机。公共的 API 应当直接报告错误。同样，你也不应该恢复一个宕机，而这段代码却不是由你来维护的，比如调用者提供的回调函数，因为你不清楚这样做是否安全。

举个例子，`net/http` 包提供一个 Web 服务器，后者能够把请求分配给用户定义的处理函数。与其让这些处理函数中的宕机使得整个进程退出，不如让服务器调用 `recover`，输出栈跟踪信息，然后继续工作。但是这样使用会有一定的风险，比如导致资源泄露或使失败的处理函数处于未定义的状态从而导致其他问题。

出于上面的原因，最安全的做法还是要选择性地使用 `recover`。换句话说，在宕机过后需要进行恢复的情况本来就不多。可以通过使用一个明确的、非导出类型作为宕机值，之后检测 `recover` 的返回值是否是这个类型（后面会看到这个例子）。如果是这个类型，可以像普

通的 error 那样处理宕机；如果不是，使用同一个参数调用 panic 以继续触发宕机。

下面的例子是 title 程序的变体，如果 HTML 文档包含多个 <title> 元素则会报错。如果这样，程序会通过调用 panic 并传递一个特殊的类型 bailout 作为参数退出递归。

```
gopl.io/ch5/title3
// soleTitle 返回文档中第一个非空标题元素
// 如果没有标题则返回错误
func soleTitle(doc *html.Node) (title string, err error) {
    type bailout struct{}

    defer func() {
        switch p := recover(); p {
        case nil:
            // 没有宕机
        case bailout{}:
            // "预期的"宕机
            err = fmt.Errorf("multiple title elements")
        default:
            panic(p) // 未预期的宕机；继续宕机过程
        }
    }()
}

// 如果发现多余一个非空标题，退出递归
forEachNode(doc, func(n *html.Node) {
    if n.Type == html.ElementNode && n.Data == "title" &&
        n.FirstChild != nil {
        if title != "" {
            panic(bailout{}) // 多个标题元素
        }
        title = n.FirstChild.Data
    }
}, nil)
if title == "" {
    return "", fmt.Errorf("no title element")
}
return title, nil
}
```

延迟的处理函数调用 recover，检查宕机值，如果该值是 bailout{} 则返回一个普通的错误。所有其他非空的值则说明是预料外的宕机，这时处理函数使用这个值作为参数调用 panic，忽略 recover 的作用并且继续之前的宕机状态（这个示例虽然违反了宕机不处理“预期”错误的建议，但是它简洁地展现了这种机制）。

有些情况下是没有恢复动作的。比如，内存耗尽使得 Go 运行时发生严重错误而直接终止进程。

练习 5.19： 使用 panic 和 recover 写一个函数，它没有 return 语句，但是能够返回一个非零的值。

第 6 章 |

The Go Programming Language

方 法

从 20 世纪 90 年代初开始，面向对象编程（OOP）的编程思想就已经在工业领域和教学领域占据了主导位置，而且几乎所有广泛应用的编程语言都支持了这种思想。Go 语言也不例外。

尽管没有统一的面向对象编程的定义，对我们来说，对象就是简单的一个值或者变量，并且拥有其方法，而方法是某种特定类型的函数。面向对象编程就是使用方法来描述每个数据结构的属性和操作，于是，使用者不需要了解对象本身的实现。

在之前的章节，我们了解了标准库中方法的常规使用方法，比如 `time.Duration` 类型的 `Seconds` 方法。

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

而且 2.5 节定义了我们自己的一个方法，为 `Celsius` 类型定义了 `String` 方法：

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

在这一章中，首先我们要学习如何基于面向对象编程思想，从而更有效地定义和使用方法。我们也会讲到两个关键的原则：封装和组合。

6.1 方法声明

方法的声明和普通函数的声明类似，只是在函数名字前面多了一个参数。这个参数把这个方法绑定到这个参数对应的类型上。

让我们现在尝试在一个与平面几何相关的包中写第一个方法：

```
gopl.io/ch6/geometry
package geometry

import "math"

type Point struct{ X, Y float64 }

// 普通的函数
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// Point 类型的方法
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

附加的参数 `p` 称为方法的接收者，它源自早先的面向对象语言，用来描述主调方法就像向对象发送消息。

Go 语言中，接收者不使用特殊名（比如 `this` 或者 `self`）；而是我们自己选择接收者名字，就像其他的参数变量一样。由于接收者会频繁地使用，因此最好能够选择简短且在整个方法

中名称始终保持一致的名字。最常用的方法就是取类型名称的首字母，就像 `Point` 中的 `p`。

调用方法的时候，接收者在方法名的前面。这样就和声明保持一致。

```
p := Point{1, 2}
q := Point{4, 6}
fmt.Println(Distance(p, q)) // "5", 函数调用
fmt.Println(p.Distance(q)) // "5", 方法调用
```

上面两个 `Distance` 函数声明没有冲突。第一个声明一个包级别的函数（称为 `geometry.Distance`）。第二个声明一个类型 `Point` 的方法，因此它的名字是 `Point.Distance`。

表达式 `p.Distance` 称作选择子（selector），因为它为接收者 `p` 选择合适的 `Distance` 方法。选择子也用于选择结构类型中的某些字段值，就像 `p.x` 中的字段值。由于方法和字段来自于同一个命名空间，因此在 `Point` 结构类型中声明一个叫作 `x` 的方法会与字段 `x` 冲突，编译器会报错。

因为每一个类型有它自己的命名空间，所以我们能够在其他不同的类型中使用名字 `Distance` 作为方法名。定义一个 `Path` 类型表示一条线段，同样也使用 `Distance` 作为方法名。

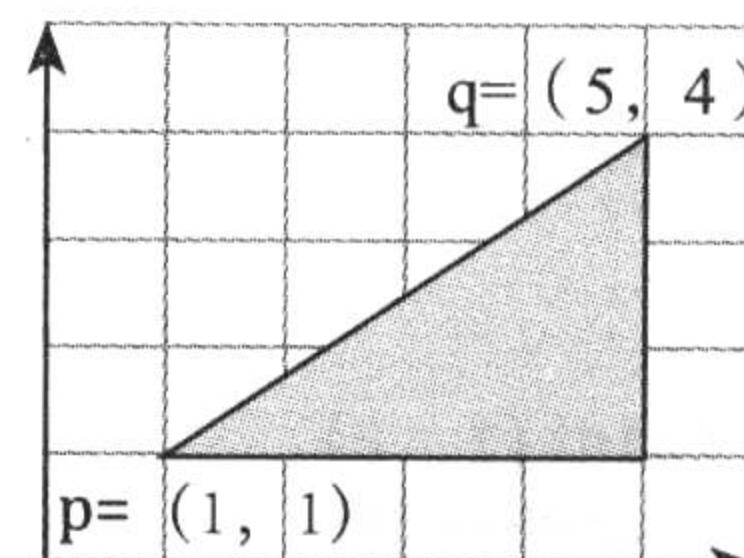
```
// Path 是连接多个点的直线段
type Path []Point
// Distance 方法返回路径的长度
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}
```

`Path` 是一个命名的 slice 类型，而非 `Point` 这样的结构体类型，但我们依旧可以给它定义方法。Go 和许多其他面向对象的语言不同，它可以将方法绑定到任何类型上。很方便地为简单的类型（如数字、字符串、slice、map，甚至函数等）定义附加的行为。同一个包下的任何类型都可以声明方法，只要它的类型既不是指针类型也不是接口类型。

这两个 `Distance` 方法拥有不同的类型。它们彼此无关，尽管 `Path.Distance` 在内部使用 `Point.Distance` 来计算线段相邻点之间的距离。

调用这个新的方法计算右边三角形的周长。

```
perim := Path{
    {1, 1},
    {5, 1},
    {5, 4},
    {1, 1},
}
fmt.Println(perim.Distance()) // "12"
```



上面两个 `Distance` 方法调用中，编译器会通过方法名和接收者的类型决定调用哪一个函数。在第一个示例中，`path[i-1]` 是 `Point` 类型，因此调用 `Point.Distance`；在第二个示例，`perim` 是 `Path` 类型，因此调用 `Path.Distance`。

类型拥有的所有方法名都必须是唯一的，但不同的类型可以使用相同的方法名，比如 `Point` 和 `Path` 类型的 `Distance` 方法；也没有必要使用附加的字段来修饰方法名（比如，`PathDistance`）以示区别。这里我们可以看到使用方法的第一个好处：命名可以比函数更简

短。在包的外部进行调用的时候，方法能够使用更加简短的名字且省略包的名字：

```
import "gopl.io/ch6/geometry"

perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
fmt.Println(geometry.PathDistance(perim)) // "12", 独立函数
fmt.Println(perim.Distance())           // "12", geometry.Path的方法
```

6.2 指针接收者的方法

由于主调函数会复制每一个实参变量，如果函数需要更新一个变量，或者如果一个实参太大而我们希望避免复制整个实参，因此我们必须使用指针来传递变量的地址。这也同样适用于更新接收者：我们将它绑定到指针类型，比如 `*Point`。

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

这个方法的名字是 `(*Point).ScaleBy`。圆括号是必需的；没有圆括号，表达式会被解析为 `*(Point.ScaleBy)`。

在真实的程序中，习惯上遵循如果 `Point` 的任何一个方法使用指针接收者，那么所有的 `Point` 方法都应该使用指针接收者，即使有些方法并不一定需要。我们在 `Point` 中打破了这个习惯只为了方便展示方法的不同使用方法。

命名类型（`Point`）与指向它们的指针（`*Point`）是唯一可以出现在接收者声明处的类型。而且，为防止混淆，不允许本身是指针的类型进行方法声明：

```
type P *int
func (P) f() { /* ... */ } // 编译错误：非法的接收者类型
```

通过提供 `*Point` 能够调用 `(*Point).ScaleBy` 方法，比如：

```
r := &Point{1, 2}
r.ScaleBy(2)
fmt.Println(*r) // "{2, 4}"
```

或者：

```
p := Point{1, 2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

或者：

```
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

但是最后两个用法虽然看上去比较别扭，但也是合法的。如果接收者 `p` 是 `Point` 类型的变量，但方法要求一个 `*Point` 接收者，我们可以使用简写：

```
p.ScaleBy(2)
```

实际上编译器会对变量进行 `&p` 的隐式转换。只有变量才允许这么做，包括结构体字段，像 `p.X` 和数组或者 `slice` 的元素，比如 `perim[0]`。不能够对一个不能取地址的 `Point` 接收者参数调用 `*Point` 方法，因为无法获取临时变量的地址。

```
Point{1, 2}.ScaleBy(2) // 编译错误：不能获得 Point 类型字面量的地址
```

但是如果实参接收者是 `*Point` 类型，以 `Point.Distance` 的方式调用 `Point` 类型的方法是合法的，因为我们有办法从地址中获取 `Point` 的值；只要解引用指向接收者的指针值即可。编译器自动插入一个隐式的 `*` 操作符。下面两个函数的调用效果是一样的：

```
pptr.Distance(q)  
(*pptr).Distance(q)
```

让我们总结一下这些例子，因为经常容易弄错。在合法的方法调用表达式中，只有符合下面三种形式的语句才能够成立。

实参接收者和形参接收者是同一个类型，比如都是 `T` 类型或都是 `*T` 类型：

```
Point{1, 2}.Distance(q) // Point  
pptr.ScaleBy(2) // *Point
```

或者实参接收者是 `T` 类型的变量而形参接收者是 `*T` 类型。编译器会隐式地获取变量的地址。

```
p.ScaleBy(2) // 隐式转换为(&p)
```

或者实参接收者是 `*T` 类型而形参接收者是 `T` 类型。编译器会隐式地解引用接收者，获得实际的取值。

```
pptr.Distance(q) // 隐式转换为(*pptr)
```

如果所有类型 `T` 方法的接收者是 `T` 自己（而非 `*T`），那么复制它的实例是安全的；调用方法的时候都必须进行一次复制。比如，`time.Duration` 的值在作为实参传递到函数的时候就会复制。但是任何方法的接收者是指针的情况下，应该避免复制 `T` 的实例，因为这么做可能会破坏内部原本的数据。比如，复制 `bytes.Buffer` 实例只会得到相当于原来 `bytes` 数组的一个别名（见 2.3.2 节）。随后的方法调用会产生不可预期的结果。

nil 是一个合法的接收者

就像一些函数允许 `nil` 指针作为实参，方法的接收者也一样，尤其是当 `nil` 是类型中有意义的零值（如 `map` 和 `slice` 类型）时，更是如此。在这个简单的整型数链表中，`nil` 代表空链表：

```
// IntList是整型链表  
// *IntList的类型nil代表空列表  
type IntList struct {  
    Value int  
    Tail  *IntList  
}  
  
// Sum返回列表元素的总和  
func (list *IntList) Sum() int {  
    if list == nil {  
        return 0  
    }  
    return list.Value + list.Tail.Sum()  
}
```

当定义一个类型允许 `nil` 作为接收者时，应当在文档注释中显式地标明，如上面的例子所示。

这是 `net/url` 包中 `Values` 类型的部分定义：

```
net/url
package url

// Values 映射字符串到字符串列表
type Values map[string][]string

// Get 返回第一个具有给定 key 的值
// 如不存在，则返回空字符串
func (v Values) Get(key string) string {
    if vs := v[key]; len(vs) > 0 {
        return vs[0]
    }
    return ""
}

// Add 添加一个键值到对应 key 列表中。
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

它的实现是 `map` 类型但也提供了一系列方法来简化 `map` 的操作，它的值是字符串 `slice`，即一个多重 `map`。使用者可以使用它固有的操作方式（`make`、`slice` 字面量、`m[key]`，等方式），或者使用它的方法，或同时使用：

```
gopl.io/ch6/urlvalues
m := url.Values{"lang": {"en"}} // 直接构造
m.Add("item", "1")
m.Add("item", "2")

fmt.Println(m.Get("lang")) // "en"
fmt.Println(m.Get("q")) // ""
fmt.Println(m.Get("item")) // "1"      (第一个值)
fmt.Println(m["item"]) // "[1 2]" (直接访问map)

m = nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3") // 容机：赋值给空的map类型
```

在最后一个 `Get` 调用中，`nil` 接收者充当一个空 `map`。它可以等同地写成 `Values(nil).Get("item")`，但是 `nil.Get("item")` 不能通过编译，因为 `nil` 的类型没有确定。相比之下，最后的 `Add` 方法会发生容机因为它尝试更新一个空的 `map`。

因为 `url.Values` 是 `map` 类型而且 `map` 间接地指向它的键 / 值对，所以 `url.Values.Add` 对 `map` 中元素的任何更新和删除操作对调用者都是可见的。然而，和普通函数一样，方法对引用本身做的任何改变，比如设置 `url.Values` 为 `nil` 或者使它指向一个不同的 `map` 数据结构，都不会在调用者身上产生作用。

6.3 通过结构体内嵌组成类型

考虑 `ColoredPoint` 类型：

```
gopl.io/ch6/coloredpoint
import "image/color"

type Point struct{ X, Y float64 }
```

```
type ColoredPoint struct {
    Point
    Color color.RGBA
}
```

我们只是想定义一个有三个字段的结构体 `ColoredPoint`, 但实际上我们内嵌了一个 `Point` 类型以提供字段 `X` 和 `Y`。在 4.4.3 节已经看到, 内嵌使我们更简便地定义了 `ColoredPoint` 类型, 它包含 `Point` 类型的所有字段以及其他更多的自有字段。如果需要, 可以直接使用 `ColoredPoint` 内所有的字段而不需要提到 `Point` 类型:

```
var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y) // "2"
```

同理, 这也适用于 `Point` 类型的方法。我们能够通过类型为 `ColoredPoint` 的接收者调用内嵌类型 `Point` 的方法, 即使在 `ColoredPoint` 类型没有声明过这个方法的情况下:

```
red := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}
fmt.Println(p.Distance(q.Point)) // "5"
p.ScaleBy(2)
q.ScaleBy(2)
fmt.Println(p.Distance(q.Point)) // "10"
```

`Point` 的方法都被纳入到 `ColoredPoint` 类型中。以这种方式, 内嵌允许构成复杂的类型, 该类型由许多字段构成, 每个字段提供一些方法。

熟悉基于类的面向对象编程语言的读者可能认为 `Point` 类型就是 `ColoredPoint` 类型的基类, 而 `ColoredPoint` 则作为子类或派生类, 或将这两个之间的关系翻译为 `ColoredPoint` 就是 `Point` 的其中一种表现。但这是个误解。注意上面调用 `Distance` 的地方。`Distance` 有一个形参 `Point`, `q` 不是 `Point`, 因此虽然 `q` 有一个内嵌的 `Point` 字段, 但是必须显式地使用它。尝试直接传递 `q` 作为参数会报错:

```
p.Distance(q) // 编译错误: 不能将 q (ColoredPoint) 转换为 Point 类型
```

`ColoredPoint` 并不是 `Point`, 但是它包含一个 `Point`, 并且它有两个另外的方法 `Distance` 和 `ScaleBy` 来自 `Point`。如果考虑具体实现, 实际上, 内嵌的字段会告诉编译器生成额外的包装方法来调用 `Point` 声明的方法, 这相当于以下代码:

```
func (p ColoredPoint) Distance(q Point) float64 {
    return p.Point.Distance(q)
}

func (p *ColoredPoint) ScaleBy(factor float64) {
    p.Point.ScaleBy(factor)
}
```

当 `Point.Distance` 在上面的第一个包调方法内调用的时候, 接收者的值是 `p.Point` 而不是 `p`, 而且这个方法是不能访问 `ColoredPoint` (其中内嵌了 `Point`) 类型的。

匿名字段类型可以是个指向命名类型的指针, 这个时候, 字段和方法间接地来自于所指向的对象。这可以让我们共享通用的结构以及使对象之间的关系更加动态、多样化。下面的

`ColoredPoint` 声明内嵌了 `*Point`:

```
type ColoredPoint struct {
    *Point
    Color color.RGBA
}

p := ColoredPoint{&Point{1, 1}, red}
q := ColoredPoint{&Point{5, 4}, blue}
fmt.Println(p.Distance(*q.Point)) // "5"
q.Point = p.Point              // p 和 q 共享同一个 Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point) // "{2 2} {2 2}"
```

结构体类型可以拥有多个匿名字段。声明 `ColoredPoint`:

```
type ColoredPoint struct {
    Point
    color.RGBA
}
```

那么这个类型的值可以拥有 `Point` 所有的方法和 `RGBA` 所有的方法，以及任何其他直接在 `ColoredPoint` 类型中声明的方法。当编译器处理选择子（比如 `p.ScaleBy`）的时候，首先，它先查找到直接声明的方法 `ScaleBy`，之后再从来自 `ColoredPoint` 的内嵌字段的方法中进行查找，再之后从 `Point` 和 `RGBA` 中内嵌字段的方法中进行查找，以此类推。当同一个查找级别中有同名方法时，编译器会报告选择子不明确的错误。

方法只能在命名的类型（比如 `Point`）和指向它们的指针（`*Point`）中声明，但内嵌帮助我们能够在未命名的结构体类型中声明方法。

下面是个很好的示例。这个例子展示了简单的缓存实现，其中使用了两个包级别的变量——互斥锁（9.2节）和 map，互斥锁将会保护 map 的数据。

```
var (
    mu sync.Mutex // 保护 mapping
    mapping = make(map[string]string)
)

func Lookup(key string) string {
    mu.Lock()
    v := mapping[key]
    mu.Unlock()
    return v
}
```

下面这个版本的功能和上面完全相同，但是将两个相关变量放到了一个包级别的变量 `cache` 中：

```
var cache = struct {
    sync.Mutex
    mapping map[string]string
} {
    mapping: make(map[string]string),
}

func Lookup(key string) string {
    cache.Lock()
    v := cache.mapping[key]
    cache.Unlock()
    return v
}
```

新的变量名更加贴切，而且 `sync.Mutex` 是内嵌的，它的 `Lock` 和 `Unlock` 方法也包含进了结构体中，允许我们直接使用 `cache` 变量本身进行加锁。

6.4 方法变量与表达式

通常我们都在相同的表达式里使用和调用方法，就像在 `p.Distance()` 中，但是把两个操作分开也是可以的。选择子 `p.Distance` 可以赋予一个方法变量，它是一个函数，把方法 (`Point.Distance`) 绑定到一个接收者 `p` 上。函数只需要提供实参而不需要提供接收者就能够调用。

```
p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance          // 方法变量
fmt.Println(distanceFromP(q))        // "5"
var origin Point                      // {0, 0}
fmt.Println(distanceFromP(origin))    // "2.23606797749979", √5

scaleP := p.ScaleBy // 方法变量
scaleP(2)           // p 变成 (2, 4)
scaleP(3)           // 然后是 (6, 12)
scaleP(10)          // 然后是 (60, 120)
```

如果包内的 API 调用一个函数值，并且使用者期望这个函数的行为是调用一个特定接收者的方法，方法变量就非常有用。比如，函数 `time.AfterFunc` 会在指定的延迟后调用一个函数值。这个程序使用 `time.AfterFunc` 在 10s 后启动火箭 `r`：

```
type Rocket struct { /* ... */ }
func (r *Rocket) Launch() { /* ... */ }

r := new(Rocket)
time.AfterFunc(10 * time.Second, func() { r.Launch() })
```

如果使用方法变量则可以更简洁：

```
time.AfterFunc(10 * time.Second, r.Launch)
```

与方法变量相关的是方法表达式。和调用一个普通的函数不同，在调用方法的时候必须提供接收者，并且按照选择子的语法进行调用。而方法表达式写成 `T.f` 或者 `(*T).f`，其中 `T` 是类型，是一种函数变量，把原来方法的接收者替换成函数的第一个形参，因此它可以像平常的函数一样调用。

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance // 方法表达式
fmt.Println(distance(p, q)) // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p)             // "{2 4}"
fmt.Printf("%T\n", scale) // "func(*Point, float64)"
```

如果你需要用一个值来代表多个方法中的一个，而方法都属于同一个类型，方法变量可以帮助你调用这个值所对应的方法来处理不同的接收者。在下面这个例子中，变量 `op` 代表加法或减法，二者都属于 `Point` 类型的方法。`Path.TranslateBy` 调用了它计算路径上的每一个点：

```

type Point struct{ X, Y float64 }

func (p Point) Add(q Point) Point { return Point{p.X + q.X, p.Y + q.Y} }
func (p Point) Sub(q Point) Point { return Point{p.X - q.X, p.Y - q.Y} }

type Path []Point

func (path Path) TranslateBy(offset Point, add bool) {
    var op func(p, q Point) Point
    if add {
        op = Point.Add
    } else {
        op = Point.Sub
    }
    for i := range path {
        // 调用 path[i].Add(offset) 或者是 path[i].Sub(offset)
        path[i] = op(path[i], offset)
    }
}

```

6.5 示例：位向量

Go语言的集合通常使用`map[T]bool`来实现，其中`T`是元素类型。使用`map`的集合扩展性良好，但是对于一些特定问题，一个专门设计过的集合性能会更优。比如，在数据流分析领域，集合元素都是小的非负整型，集合拥有许多元素，而且集合的操作多数是求并集和交集，位向量是个理想的数据结构。

位向量使用一个无符号整型值的slice，每一位代表集合中的一个元素。如果设置第`i`位的元素，则认为集合包含`i`。下面的程序演示了一个含有三个方法的简单位向量类型。

```

gopl.io/ch6/intset

// IntSet是一个包含非负整数的集合
// 零值代表空的集合
type IntSet struct {
    words []uint64
}

// Has方法的返回值表示是否存在非负数x
func (s *IntSet) Has(x int) bool {
    word, bit := x/64, uint(x%64)
    return word < len(s.words) && s.words[word]&(1<<bit) != 0
}

// Add添加非负数x到集合中
func (s *IntSet) Add(x int) {
    word, bit := x/64, uint(x%64)
    for word >= len(s.words) {
        s.words = append(s.words, 0)
    }
    s.words[word] |= 1 << bit
}

// UnionWith将会对s和t做并集并将结果存在s中
func (s *IntSet) UnionWith(t *IntSet) {
    for i, tword := range t.words {
        if i < len(s.words) {
            s.words[i] |= tword
        } else {
            s.words = append(s.words, tword)
        }
    }
}

```

由于每一个字拥有 64 位，因此为了定位 x 位的位置，我们使用商数 $x/64$ 作为字的索引，而 $x \% 64$ 记作该字内位的索引。`UnionWith` 操作使用按位“或”操作符 `|` 来计算一次 64 个元素求并集的结果。（在练习 6.5 中会再来看 64 位字的选择。）

这个实现缺少许多需要的特性，有些会在练习中列出来，但是有一个特性不得不在这里提到：以字符串输出 `IntSet` 的方法。添加一个 `String` 方法，就像在 2.5 节里的 `Celsius` 类型那样。

```
// String方法以字符串"{1 2 3}"的形式返回集中
func (s *IntSet) String() string {
    var buf bytes.Buffer
    buf.WriteByte('{')
    for i, word := range s.words {
        if word == 0 {
            continue
        }
        for j := 0; j < 64; j++ {
            if word&(1<<uint(j)) != 0 {
                if buf.Len() > len("{") {
                    buf.WriteByte(' ')
                }
                fmt.Fprintf(&buf, "%d", 64*i+j)
            }
        }
    }
    buf.WriteByte('}')
    return buf.String()
}
```

注意，上面的 `String` 方法和 3.5.4 节的 `intsToString` 相似；在 `String` 方法中 `bytes.Buffer` 经常以这样的方式用到。`fmt` 包把具有 `String` 方法的类型进行特殊处理，于是，即使是复杂类型也可以按照友好的方式显示出来。`fmt` 默认调用 `String` 方法而不是原生的值。这个机制需要依靠接口和类型断言，第 7 章将介绍它们。

现在，可以演示 `IntSet` 了：

```
var x, y IntSet
x.Add(1)
x.Add(144)
x.Add(9)
fmt.Println(x.String()) // "{1 9 144}"

y.Add(9)
y.Add(42)
fmt.Println(y.String()) // "{9 42}"

x.UnionWith(&y)
fmt.Println(x.String()) // "{1 9 42 144}"

fmt.Println(x.Has(9), x.Has(123)) // "true false"
```

提醒一句：我们为指针类型 `*IntSet` 声明了 `String` 和 `Has` 方法并非出于需要，而是为了和其他两个方法保持一致，另外两个方法需要指针接收者，因为它们需要对 `s.words` 进行赋值。所以，`IntSet` 的值并不含有 `String` 方法，使用它可能会产生意料外的结果：

```
fmt.Println(&x)          // "{1 9 42 144}"
fmt.Println(x.String()) // "{1 9 42 144}"
fmt.Println(x)          // "[4398046511618 0 65536]"
```

第一个示例中，输出了 `*IntSet` 指针，它有一个 `String` 方法。第二个示例中，基于

`IntSet` 值调用 `String()`；编译器会帮我们隐式地插入 `&` 操作符，我们得到指针后就可以获取到 `String` 方法了。但在第三个示例中，因为 `IntSet` 值本身并没有 `String` 方法，所以 `fmt.Println` 直接输出结构体。因此，记得加上 `&` 操作符很重要。那么给 `IntSet`（而不是 `*IntSet`）加上 `String` 方法应该是个不错的主意，但这还是需要根据实际情况来看。

练习 6.1：实现这些附加的方法：

```
func (*IntSet) Len() int      // 返回元素个数
func (*IntSet) Remove(x int)  // 从集合去除元素 x
func (*IntSet) Clear()        // 删除所有元素
func (*IntSet) Copy() *IntSet // 返回集合的副本
```

练习 6.2：定义一个变长方法 `(*IntSet).AddAll(...int)`，它允许接受一串整型值作为参数，比如 `s.AddAll(1,2,3)`。

练习 6.3：`(*IntSet).UnionWith` 计算了两个集合的并集，使用 `|` 操作符对每个字进行按位“或”操作。实现交集、差集和对称差运算。（两个集合的对称差包含只在某个集合中存在的元素。）

练习 6.4：添加方法 `elems` 返回包含集合元素的 slice，这适合在 `range` 循环中使用。

练习 6.5：`IntSet` 使用的每个字的类型都是 `uint64`，但是 64 位的计算在 32 位平台上的效率不高。改写程序以使用 `uint` 类型，这是适应平台的无符号整型。除以 64 的操作可以使用一个常量来代表 32 位或 64 位。你或许可以使用一个讨巧的表达式 `32<<(^uint(0)>>63)` 来表示除数。

6.6 封装

如果变量或者方法是不能通过对象访问到的，这称作封装的变量或者方法。封装（有时候称作数据隐藏）是面向对象编程中重要的一方面。

Go 语言只有一种方式控制命名的可见性：定义的时候，首字母大写的标识符是可以从包中导出的，而首字母没有大写的则不导出。同样的机制也同样作用于结构体内的字段和类型中的方法。结论就是，要封装一个对象，必须使用结构体。

这就是为什么上一节里 `IntSet` 类型被声明为结构体但是它只有单个字段：

```
type IntSet struct {
    words []uint64
}
```

可以重新定义 `IntSet` 为一个 slice 类型，如下所示，当然，必须把方法中出现的 `s.words` 替换为 `*s`。

```
type IntSet []uint64
```

尽管这个版本的 `IntSet` 和之前的基本等同，但是它将能够允许其他包内的使用方读取和改变这个 slice。换句话说，表达式 `*s` 可以在其他包内使用，`s.words` 只能在定义 `IntSet` 的包内使用。

另一个结论就是在 Go 语言中封装的单元是包而不是类型。无论是在函数内的代码还是方法内的代码，结构体类型内的字段对于同一个包中的所有代码都是可见的。

封装提供了三个优点。第一，因为使用方不能直接修改对象的变量，所以不需要更多的语句用来检查变量的值。

第二，隐藏实现细节可以防止使用方依赖的属性发生改变，使得设计者可以更加灵活地改变 API 的实现而不破坏兼容性。

作为一个例子，考虑 `bytes.Buffer` 类型。它用来堆积非常小的字符串，因此为了优化性能，实现上会预留一部分额外的空间避免频繁申请内存。由于 `Buffer` 是结构体类型，因此这块空间使用额外的一个字段 `[64]byte`，且命名不是首字母大写。因为这个字段没有导出，`bytes` 包之外的 `Buffer` 使用者除了能感觉到性能的提升之外，不会关心其中的实现。`Buffer` 和它的 `Grow` 方法如下面的例子所示：

```
type Buffer struct {
    buf      []byte
    initial [64]byte
    /* ... */
}

// Grow 方法按需扩展缓冲区的大小
// 保证 n 个字节的空间
func (b *Buffer) Grow(n int) {
    if b.buf == nil {
        b.buf = b.initial[:0] // 最初使用预分配的空间
    }
    if len(b.buf)+n > cap(b.buf) {
        buf := make([]byte, b.Len(), 2*cap(b.buf) + n)
        copy(buf, b.buf)
        b.buf = buf
    }
}
```

第三个重要的好处，就是防止使用者肆意地改变对象内的变量。因为对象的变量只能被同一个包内的函数修改，所以包的作者能够保证所有的函数都可以维护对象内部的资源。比如，下面的 `Counter` 类型允许使用者递增计数或者重置计数器，但是不能够随意地设置当前计数器的值：

```
type Counter struct { n int }

func (c *Counter) N() int      { return c.n }
func (c *Counter) Increment() { c.n++ }
func (c *Counter) Reset()     { c.n = 0 }
```

仅仅用来获得或者修改内部变量的函数称为 `getter` 和 `setter`，就像 `log` 包里的 `Logger` 类型。然而，命名 `getter` 方法的时候，通常将 `Get` 前缀省略。这个简洁的命名习惯也同样适用在其他冗余的前缀上，比如 `Fetch`、`Find` 和 `Lookup`。

```
package log

type Logger struct {
    flags int
    prefix string
    // ...
}

func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

Go 语言也允许导出的字段。当然，一旦导出就必须要面对 API 的兼容问题，因此最初的决定需要慎重，要考虑到之后维护的复杂程度，将来发生变化的可能性，以及变化对原本

代码质量的影响等。

封装并不总是必需的。`time.Duration` 对外暴露 `int64` 的整型数用于获得微秒，这使我们能够对其进行通常的数学运算和比较操作，甚至定义常数：

```
const day = 24 * time.Hour  
fmt.Println(day.Seconds()) // "86400"
```

另一个例子可以比较 `IntSet` 和本章开头的 `geometry.Path` 类型。`Path` 定义为一个 `slice` 类型，允许它的使用者使用 `slice` 字面量的语法来构成实例，比如使用 `range` 循环遍历 `Path` 所有的点等，而 `IntSet` 则不允许这些操作。

有个明显的对比：`geometry.Path` 从本质上讲就只是连续的点，以后也不会添加新的字段，因此 `geometry` 包将 `Path` 的 `slice` 类型暴露出来是合理的做法。与它不同的是，`IntSet` 只是看上去像 `[]uint64` 的 `slice`。但它实际上完全可以是 `[]uint` 或其他复杂的集合类型，而且另外用来记录集合中元素数量的字段充当了重要的作用。基于上述原因，`IntSet` 不对外透明也合情合理。

在这一章中，我们学习了如何在命名类型中定义方法，以及如何调用它们。尽管方法是面向对象编程的关键，但这一章只讲述了其中的一部分内容。下一章会继续介绍接口相关的内容来完成这方面的学习。

接 口

接口类型是对其他类型行为的概括与抽象。通过使用接口，我们可以写出更加灵活和通用的函数，这些函数不用绑定在一个特定的类型实现上。

很多面向对象的语言都有接口这个概念，Go语言的接口的独特之处在于它是隐式实现。换句话说，对于一个具体的类型，无须声明它实现了哪些接口，只要提供接口所必需的方法即可。这种设计让你无须改变已有类型的实现，就可以为这些类型创建新的接口，对于那些不能修改包的类型，这一点特别有用。

本章首先会介绍接口类型的基本机制类型价值。然后会讨论标准库中的几种重要接口。因为在很多Go程序中，相对于新创建的接口，标准库中的接口使用得并不少。最后，我们还要了解一下类型断言（见7.10节）以及类型分支（见7.13节），以及它们如何实现另一种类型的通用化。

7.1 接口即约定

之前介绍的类型都是具体类型。具体类型指定了它所含数据的精确布局，还暴露了基于这个精确布局的内部操作。比如对于数值有算术操作，对于slice类型我们有索引、append、range等操作。具体类型还会通过其方法来提供额外的能力。总之，如果你知道了一个具体类型的数据，那么你就精确地知道了它是什么以及它能干什么。

Go语言中还有另外一种类型称为接口类型。接口是一种抽象类型，它并没有暴露所含数据的布局或者内部结构，当然也没有那些数据的基本操作，它所提供的仅仅是一些方法而已。如果你拿到一个接口类型的值，你无从知道它是什么，你能知道的仅仅是它能做什么，或者更精确地讲，仅仅是它提供了哪些方法。

本书通篇使用两个类似的函数实现字符串的格式化：`fmt.Printf` 和 `fmt.Sprintf`。前者把结果发到标准输出（标准输出其实是一个文件），后者把结果以`string`类型返回。格式化是两个函数中最复杂的部分，如果仅仅因为两个函数在输出方式上的轻微差异，就需要把格式化部分在两个函数中重复一遍，那么就太糟糕了。幸运的是，通过接口机制可以解决这个问题。其实，两个函数都封装了第三个函数`fmt.Fprintf`，而这个函数对结果实际输出到哪里毫不关心：

```
package fmt

func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)
func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}

func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

`Fprintf` 的前缀 F 指文件，表示格式化的输出会写入第一个实参所指代的文件。对于 `Printf`，第一个实参就是 `os.Stdout`，它属于 `*os.File` 类型。对于 `Sprintf`，尽管第一个实参不是文件，但它模拟了一个文件：`&buf` 就是一个指向内存缓冲区的指针，与文件类似，这个缓冲区也可以写入多个字节。

其实 `Fprintf` 的第一个形参也不是文件类型，而是 `io.Writer` 接口类型，其声明如下：

```
package io

// Writer 接口封装了基础的写入方法
type Writer interface {
    // Write 从 p 向底层数据流写入 len(p) 个字节的数据
    // 返回实际写入的字节数 (0 <= n <= len(p))
    // 如果没有写完，那么会返回遇到的错误
    // 在 Write 返回 n < len(p) 时，err 必须为非 nil
    // Write 不允许修改 p 的数据，即使是临时修改
    //
    // 实现时不允许残留 p 的引用
    Write(p []byte) (n int, err error)
}
```

`io.Writer` 接口定义了 `Fprintf` 和调用者之间的约定。一方面，这个约定要求调用者提供的具体类型（比如 `*os.File` 或者 `*bytes.Buffer`）包含一个与其签名和行为一致的 `Write` 方法。另一方面，这个约定保证了 `Fprintf` 能使用任何满足 `io.Writer` 接口的参数。`Fprintf` 只需要能调用参数的 `Write` 函数，无须假设它写入的是一个文件还是一段内存。

因为 `fmt.Fprintf` 仅依赖于 `io.Writer` 接口所约定的方法，对参数的具体类型没有要求，所以我们可以用任何满足 `io.Writer` 接口的具体类型作为 `fmt.Fprintf` 的第一个实参。这种可以把一种类型替换为满足同一接口的另一种类型的特性称为可取代性（substitutability），这也是面向对象语言的典型特征。

让我们创建一个新类型来测试这个特性。如下所示的 `*ByteCounter` 类型的 `Write` 方法仅仅统计传入数据的字节数，然后就不管那些数据了。（下面的代码中出现的类型转换是为了让 `len(p)` 和 `*c` 满足 `+=` 操作。）

```
gopl.io/ch7/bytectrainer

type ByteCounter int

func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // 转换 int 为 ByteCounter 类型
    return len(p), nil
}
```

因为 `*ByteCounter` 满足 `io.Writer` 接口的约定，所以可以在 `Fprintf` 中使用它，`Fprintf` 察觉不到这种类型差异，`ByteCounter` 也能正确地累积格式化后结果的长度。

```
var c ByteCounter
c.Write([]byte("hello"))
fmt.Println(c) // "5", = len("hello")

c = 0 // 重置计数器
var name = "Dolly"
fmt.Fprintf(&c, "hello, %s", name)
fmt.Println(c) // "12", = len("hello, Dolly")
```

除之 `io.Writer` 之外，`fmt` 包还有另一个重要的接口。`Fprintf` 和 `Fprintln` 提供了一个让类型控制如何输出自己的机制。在 2.5 节中，给 `Celsius` 类型定义了一个 `String` 方法，这样可以输出 "100°C" 这样的结果。在 6.5 节中，也给 `*IntSet` 类型加了一个 `String` 方法，这样可

以输出类似 "{ 1 2 3}" 的传统集合表示形式。定义一个 `String` 方法就可以让类型满足这个广泛使用的接口 `fmt.Stringer`:

```
package fmt

// 在字符串格式化时如果需要一个字符串
// 那么就调用这个方法来把当前值转化为字符串
// Print 这种不带格式化参数的输出方式也是调用这个方法
type Stringer interface {
    String() string
}
```

7.10 节会解释 `fmt` 包如何发现哪些值满足这个接口。

练习 7.1： 使用类似 `ByteCounter` 的想法，实现单词和行的计数器。实现时考虑使用 `bufio.ScanWords`。

练习 7.2： 实现一个满足如下签名的 `CountingWriter` 函数，输入一个 `io.Writer`，输出一个封装了输入值的新 `Writer`，以及一个指向 `int64` 的指针，该指针对应的值是新的 `Writer` 写入的字节数。

```
func CountingWriter(w io.Writer) (io.Writer, *int64)
```

练习 7.3： 为 `gopl.io/ch4/treesort` 中的 `*tree` 类型（见 4.4 节）写一个 `String` 方法，用于展示其中的值序列。

7.2 接口类型

一个接口类型定义了一套方法，如果一个具体类型要实现该接口，那么必须实现接口类型定义中的所有方法。

`io.Writer` 是一个广泛使用的接口，它负责所有可以写入字节的类型的抽象，包括文件、内存缓冲区、网络连接、HTTP 客户端、打包器（archiver）、散列器（hasher）等。`io` 包还定义了很多有用的接口。`Reader` 就抽象了所有可以读取字节的类型，`Closer` 抽象了所有可以关闭的类型，比如文件或者网络连接。（现在你大概已经注意到 Go 语言的单方法接口的命名约定定了。）

```
package io

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

type Closer interface {
    Close() error
}
```

另外，我们还可以发现通过组合已有接口得到的新接口，比如下面两个例子：

```
type ReadWriter interface {
    Reader
    Writer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

如上的语法称为嵌入式接口，与嵌入式结构类似，让我们可以直接使用一个接口，而不用逐一写出这个接口所包含的方法。如下所示，尽管不够简洁，但是可以不用嵌入式来声明 `io.ReadWriter`：

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

也可以混合使用两种方式：

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

三种声明的效果都是一致的。方法定义的顺序也是无意义的，真正有意义的只有接口的方法集合。

练习 7.4: `strings.NewReader` 函数输入一个字符串，返回一个从字符串读取数据且满足 `io.Reader` 接口（也满足其他接口）的值。请自己实现该函数，并且通过它来让 HTML 分析器（参考 5.2 节）支持以字符串作为输入。

练习 7.5: `io` 包中的 `LimitReader` 函数接受 `io.Reader r` 和字节数 `n`，返回一个 `Reader`，该返回值从 `r` 读取数据，但在读取 `n` 字节后报告文件结束。请实现该函数。

```
func LimitReader(r io.Reader, n int64) io.Reader
```

7.3 实现接口

如果一个类型实现了一个接口要求的所有方法，那么这个类型实现了这个接口。比如 `*os.File` 类型实现了 `io.Reader`、`Writer`、`Closer` 和 `ReaderWriter` 接口。`*bytes.Buffer` 实现了 `Reader`、`Writer` 和 `ReaderWriter`，但没有实现 `Closer`，因为它没有 `Close` 方法。为了简化表述，Go 程序员通常说一个具体类型“是一个”(is-a) 特定的接口类型，这其实代表着该具体类型实现了该接口。比如，`*bytes.Buffer` 是一个 `io.Writer`；`*os.File` 是一个 `io.ReaderWriter`。

接口的赋值规则（参考 2.4.2 节）很简单，仅当一个表达式实现了一个接口时，这个表达式才可以赋给该接口。所以：

```
var w io.Writer
w = os.Stdout          // OK: *os.File有Write方法
w = new(bytes.Buffer)  // OK: *bytes.Buffer有Write方法
w = time.Second        // 编译错误: time.Duration缺少Write方法

var rwc io.ReadWriteCloser
rwc = os.Stdout        // OK: *os.File有Read、Write、Close方法
rwc = new(bytes.Buffer) // 编译错误: *bytes.Buffer缺少Close方法
```

当右侧表达式也是一个接口时，该规则也有效：

```
w = rwc                // OK: io.ReadWriteCloser有Write方法
rwc = w                // 编译错误: io.Writer 缺少Close方法
```

因为 `ReadWriter` 和 `ReadWriteCloser` 接口包含了 `Writer` 的所有方法，所以任何实现了 `ReadWriter` 或 `ReadWriteCloser` 类型的方法都必然实现了 `Writer`。

在进一步讨论之前，我们先解释一下一个类型有某一个方法的具体含义。6.2 节曾提到，

对每一个具体类型 T ，部分方法的接收者就是 T ，而其他方法的接收者则是 $*T$ 指针。同时我们对类型 T 的变量直接调用 $*T$ 的方法也可以是合法的，只要改变量是可变的，编译器隐式地帮你完成了取地址的操作。但这仅仅是一个语法糖，类型 T 的方法没有对应的指针 $*T$ 多，所以实现的接口也可能比对应的指针少。

比如，6.5 节提到的 `IntSet` 类型的 `String` 方法，需要一个指针接收者，所以我们无法从一个无地址的 `IntSet` 值上调用该方法：

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string
var _ = IntSet{}.String() // 编译错误：String 方法需要 *IntSet 接收者
```

但可以从一个 `IntSet` 变量上调用该方法：

```
var s IntSet
var _ = s.String() // OK：s 是一个变量，&s 有 String 方法
```

因为只有 `*IntSet` 有 `String` 方法，所以也只有 `*IntSet` 实现了 `fmt.Stringer` 接口：

```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s // 编译错误：IntSet 缺少 String 方法
```

在 12.8 节，有一个程序可以输出一个任意值的方法，`godoc-analysis=type` 工具（见 10.7.4 节）也可以显示每个类型的方法，以及接口和具体类型的关系。

正如信封封装了信件，接口也封装了所对应的类型和数据，只有通过接口暴露的方法才可以调用，类型的其他方法则无法通过接口来调用：

```
os.Stdout.Write([]byte("hello")) // OK：*os.File 有 Write 方法
os.Stdout.Close() // OK：*os.File 有 Close 方法

var w io.Writer
w = os.Stdout
w.Write([]byte("hello")) // OK：io.Writer 有 Write 方法
w.Close() // 编译错误：io.Writer 缺少 Close 方法
```

一个拥有更多方法的接口，比如 `io.ReadWriter`，与 `io.Reader` 相比，给了我们它所指向数据的更多信息，当然也给实现这个接口提出更高的门槛。那么对于接口类型 `interface{}`，它完全不包含任何方法，通过这个接口能得到对应具体类型的什么信息呢？

确实什么信息也得不到。看起来这个接口没有任何用途，但实际上称为空接口类型的 `interface{}` 是不可缺少的。正因为空接口类型对其实现类型没有任何要求，所以我们可以把任何值赋给空接口类型。

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

其实在本书的第一个示例中就用了空接口类型，靠它才可以让 `fmt.Println`、`errorf`（参考 5.7 节）这类的函数能够接受任意类型的参数。

当然，即使我们创建了一个指向布尔值、浮点数、字符串、`map`、指针或者其他类型的 `interface{}` 接口，也无法直接使用其中的值，毕竟这个接口不包含任何方法。我们需要一个方法从空接口中还原出实际值，在 7.10 节中我们可以看到如何用类型断言来实现该功能。

判定是否实现接口只需要比较具体类型和接口类型的方法，所以没有必要在具体类型的

定义中声明这种关系。也就是说，偶尔在注释中标注也不坏，但对于程序来讲，这种关系声明不是必需的。如下声明在编译器就断言了 `*bytes.Buffer` 类型的一个值必然实现了 `io.Writer`:

```
// *bytes.Buffer 必须实现 io.Writer
var w io.Writer = new(bytes.Buffer)
```

我们甚至不需要创建一个新的变量，因为 `*bytes.Buffer` 的任意值都实现了这个接口，甚至 `nil`，在我们用 `(*bytes.Buffer)(nil)` 来强制类型转换后，也实现了这个接口。当然，既然我们不想引用 `w`，那么我们可以把它替换为空白标识符。基于这两点，修改后的代码可以节省不少变量：

```
// *bytes.Buffer 必须实现 io.Writer
var _ io.Writer = (*bytes.Buffer)(nil)
```

非空的接口类型（比如 `io.Writer`）通常由一个指针类型来实现，特别是当接口类型的一个或多个方法暗示会修改接收者的情形（比如 `Write` 方法）。一个指向结构的指针才是最常见的方法接收者。

指针类型肯定不是实现接口的唯一类型，即使是那些包含了会改变接收者方法的接口类型，也可以由 Go 的其他引用类型来实现。我们已经见过 `slice` 类型的方法（`geometry.Path`，参考 6.1 节），以及 `map` 类型的方法（`url.Values`，参考 6.2.1 节），稍后我们还可以看到函数类型的方法（`http.HandlerFunc`，参考 7.7 节）。基础类型也可以实现方法，比如我们会在 7.4 节见到的 `time.Duration` 类型实现了 `fmt.Stringer`。

一个具体类型可以实现很多不相关的接口。比如一个程序管理或者销售数字文化商品，比如音乐、电影和图书。那么它可能定义了如下具体类型：

```
Album
Book
Movie
Magazine
Podcast
TVEpisode
Track
```

我们可以把感兴趣的每一种抽象都用一种接口类型来表示。一些属性是所有商品都具备的，比如标题、创建日期以及创建者列表（作者或者艺术家）。

```
type Artifact interface {
    Title() string
    Creators() []string
    Created() time.Time
}
```

其他属性则局限于特定类型的商品。比如字数这个属性只与书和杂志相关，而屏幕分辨率则只与电影和电视剧相关。

```
type Text interface {
    Pages() int
    Words() int
    PageSize() int
}

type Audio interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // 比如 "MP3"、"WAV"
}
```

```
type Video interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // 比如 "MP4"、"WMV"
    Resolution() (x, y int)
}
```

这些接口只是一种把具体类型分组并暴露它们共性的方式，未来我们也可以发现其他的分组方式。比如，如果我们要把 `Audio` 和 `Video` 按照同样的方式来处理，就可以定义一个 `Streamer` 接口来呈现它们的共性，而不用修改现有的类型定义。

```
type Streamer interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
}
```

从具体类型出发、提取其共性而得出的每一种分组方式都可以表示为一种接口类型。与基于类的语言（它们显式地声明了一个类型实现的所有接口）不同的是，在 Go 语言里我们可以在需要时才定义新的抽象和分组，并且不用修改原有类型的定义。当需要使用另一个作者写的包里的具体类型时，这一点特别有用。当然，还需要这些具体类型在底层是真正有共性的。

7.4 使用 `flag.Value` 来解析参数

在本节中，我们将看到如何使用另外一个标准接口 `flag.Value` 来帮助我们定义命令行标志。考虑如下一个程序，它实现了睡眠指定时间的功能。

```
gopl.io/ch7/sleep
var period = flag.Duration("period", 1*time.Second, "sleep period")

func main() {
    flag.Parse()
    fmt.Printf("Sleeping for %v...", *period)
    time.Sleep(*period)
    fmt.Println()
}
```

在程序进入睡眠前输出了睡眠时长。`fmt` 包调用了 `time.Duration` 的 `String` 方法，可以按照一个用户友好的方式来输出，而不是输出一个以纳秒为单位的数字。

```
$ go build gopl.io/ch7/sleep
$ ./sleep
Sleeping for 1s...
```

默认的睡眠时间是 `1s`，但可以用 `-period` 命令行标志来控制。`flag.Duration` 函数创建了一个 `time.Duration` 类型的标志变量，并且允许用户用一种友好的方式来指定时长，比如可以用 `String` 方法对应的记录方法。这种对称的设计提供了一个良好的用户接口。

```
$ ./sleep -period 50ms
Sleeping for 50ms...
$ ./sleep -period 2m30s
Sleeping for 2m30s...
$ ./sleep -period 1.5h
Sleeping for 1h30m0s...
$ ./sleep -period "1 day"
invalid value "1 day" for flag -period: time: invalid duration 1 day
```

因为时间长度类的命令行标志广泛应用，所以这个功能内置到了 `flag` 包。支持自定义类型其实也不难，只须定义一个满足 `flag.Value` 接口的类型，其定义如下所示：

```
package flag

// Value 接口代表了存储在标志内的值
type Value interface {
    String() string
    Set(string) error
}
```

`String` 方法用于格式化标志对应的值，可用于输出命令行帮助消息。由于有了该方法，因此每个 `flag.Value` 其实也是 `fmt.Stringer`。`Set` 方法解析了传入的字符串参数并更新标志值。可以认为 `Set` 方法是 `String` 方法的逆操作，两个方法使用同样的记法规格是一个很好的实践。

下面定义了 `celsiusFlag` 类型来允许在参数中使用摄氏温度或华氏温度。注意，`celsiusFlag` 内嵌了一个 `Celsius` 类型（参考 2.5 节），所以已经有 `String` 方法了。为了满足 `flag.Value` 接口，只须再定义一个 `Set` 方法：

```
gopl.io/ch7/tempconv
// *celsiusFlag 满足 flag.Value 接口
type celsiusFlag struct{ Celsius }

func (f *celsiusFlag) Set(s string) error {
    var unit string
    var value float64
    fmt.Sscanf(s, "%f%s", &value, &unit) // 无须检查错误
    switch unit {
    case "C", "°C":
        f.Celsius = Celsius(value)
        return nil
    case "F", "°F":
        f.Celsius = FToC(Fahrenheit(value))
        return nil
    }
    return fmt.Errorf("invalid temperature %q", s)
}
```

`fmt.Sscanf` 函数用于从输入 `s` 解析一个浮点值 (`value`) 和一个字符串 (`unit`)。尽管通常都必须检查 `Sscanf` 的错误结果，但在这种情况下我们无须检查。因为如果出现错误，那么接下来的跳转条件没有一个会满足。

如下 `CelsiusFlag` 函数封装了上面的逻辑。这个函数返回了一个 `Celsius` 指针，它指向嵌入在 `celsiusFlag` 变量 `f` 中的一个字段。`Celsius` 字段在标志处理过程中会发生变化（经由 `Set` 方法）。调用 `Var` 方法可以把这个标志加入到程序的命令行标记集合中，即全局变量 `flag.CommandLine`。如果一个程序有非常复杂的命令行接口，那么单个全局变量 `flag.CommandLine` 就不够用了，需要有多个类似的变量来支撑。调用 `Var` 方法时会把 `*celsiusFlag` 实参赋给 `flag.Value` 形参，编译器会在此时检查 `*celsiusFlag` 类型是否有 `flag.Value` 所必需的方法。

```
// CelsiusFlag 根据给定的 name、默认值和使用方法
// 定义了一个 Celsius 标志，返回了标志值的指针
// 标志必须包含一个数值和一个单位，比如："100C"
```

```
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}
```

现在可以在程序中使用这个新标志了：

```
gopl.io/ch7/tempflag
var temp = tempconv.CelsiusFlag("temp", 20.0, "the temperature")

func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

接下来是一些典型的使用方法：

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: invalid temperature "273.15K"
Usage of ./tempflag:
  -temp value
      the temperature (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
  -temp value
      the temperature (default 20°C)
```

练习 7.6：在 tempflag 中支持热力学温度。

练习 7.7：请解释为什么默认值 20.0 没写 °C，而帮助消息中却包含 °C。

7.5 接口值

从概念上来讲，一个接口类型的值（简称接口值）其实有两个部分：一个具体类型和该类型的一个值。二者称为接口的动态类型和动态值。

对于像 Go 这样的静态类型语言，类型仅仅是一个编译时的概念，所以类型不是一个值。在我们的概念模型中，用类型描述符来提供每个类型的具体信息，比如它的名字和方法。对于一个接口值，类型部分就用对应的类型描述符来表述。

如下四个语句中，变量 w 有三个不同的值（最初和最后是同一个值）：

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

接下来让我们详细地查看一下在每个语句之后 w 的值和相关的动态行为。第一个语句声明了 w：

```
var w io.Writer
```

在 Go 语言中，变量总是初始化为一个特定的值，接口也不例外。接口的零值就是把它

的动态类型和值都设置为 nil，如图 7-1 所示。

一个接口值是否是 nil 取决于它的动态类型，所以现在这是一个 nil 接口值。可以用 `w == nil` 或者 `w != nil` 来检测一个接口值是否是 nil。调用一个 nil 接口的任何方法都会导致崩溃：

```
w.Write([]byte("hello")) // 崩溃：对空指针取引用值
```

第二个语句把一个 `*os.File` 类型的值赋给了 `w`：

```
w = os.Stdout
```

这次赋值把一个具体类型隐式转换为一个接口类型，它与对应的显式转换 `io.Writer(os.Stdout)` 等价。不管这种类型的转换是隐式的还是显式的，它都可以转换操作数的类型和值。接口值的动态类型会设置为指针类型 `*os.File` 的类型描述符，它的动态值会设置为 `os.Stdout` 的副本，即一个指向代表进程的标准输出的 `os.File` 类型的指针，如图 7-2 所示。

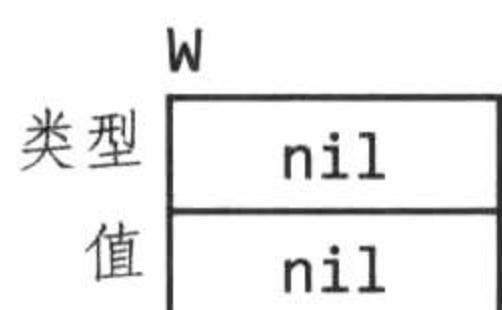


图 7-1 一个 nil 接口值



图 7-2 包含 os.File 指针的接口值

调用该接口值的 `Write` 方法，会实际调用 `(*os.File).Write` 方法，即输出 "hello"。

```
w.Write([]byte("hello")) // "hello"
```

一般来讲，在编译时我们无法知道一个接口值的动态类型会是什么，所以通过接口来做调用必然需要使用动态分发。编译器必须生成一段代码来从类型描述符拿到名为 `Write` 的方法地址，再间接调用该方法地址。调用的接收者就是接口值的动态值，即 `os.Stdout`，所以实际效果与直接调用等价：

```
os.Stdout.Write([]byte("hello")) // "hello"
```

第三个语句把一个 `*bytes.Buffer` 类型的值赋给了接口值：

```
w = new(bytes.Buffer)
```

动态类型现在是 `*bytes.Buffer`，动态值现在则是一个指向新分配缓冲区的指针，如图 7-3 所示。

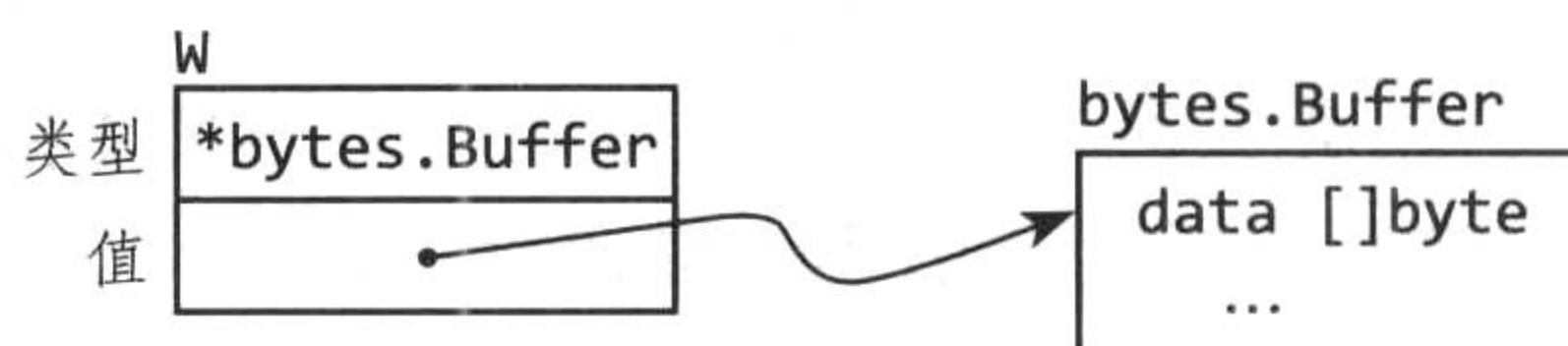


图 7-3 包含 *bytes.Buffer 指针的接口值

调用 `Write` 方法的机制也跟第二个语句一致：

```
w.Write([]byte("hello")) // 把 "hello" 写入 bytes.Buffer
```

这次，类型描述符是 `*bytes.Buffer`，所以调用的是 `(*bytes.Buffer).Write` 方法，方法的接收者是缓冲区的地址。调用该方法会追加 "hello" 到缓冲区。

最后，第四个语句把 nil 赋给了接口值：

```
w = nil
```

这个语句把动态类型和动态值都设置为 nil，把 w 恢复到了它刚声明时的状态（如图 7-1 所示）。

一个接口值可以指向多个任意大的动态值。比如，`time.Time` 类型可以表示一个时刻，它是一个包含几个非导出字段的结构。如果从它创建一个接口值：

```
var x interface{} = time.Now()
```

结果可能如图 7-4 所示。从理论上来讲，无论动态值有多大，它永远在接口值内部（当然这只是一个理论模型；实际的实现是很不同的）。

接口值可以用 == 和 != 操作符来做比较。如果两个接口值都是 nil 或者二者的动态类型完全一致且二者动态值相等（使用动态类型的 == 操作符来做比较），那么两个接口值相等。因为接口值是可以比较的，所以它们可以作为 map 的键，也可以作为 switch 语句的操作数。

需要注意的是，在比较两个接口值时，如果两个接口值的动态类型一致，但对应的动态值是不可比较的（比如 slice），那么这个比较会以崩溃的方式失败：

```
var x interface{} = []int{1, 2, 3}
fmt.Println(x == x) // 崩溃：试图比较不可比较的类型 []int
```

从这点来看，接口类型是非平凡的。其他类型要么是可以安全比较的（比如基础类型和指针），要么是完全不可比较的（比如 slice、map 和函数），但当比较接口值或者其中包含接口值的聚合类型时，我们必须小心崩溃的可能性。当把接口作为 map 的键或者 switch 语句的操作数时，也存在类似的风险。仅在能确认接口值包含的动态值可以比较时，才比较接口值。

当处理错误或者调试时，能拿到接口值的动态类型是很有帮助的。可以使用 `fmt` 包的 %T 来实现这个需求：

```
var w io.Writer
fmt.Printf("%T\n", w) // <nil>

w = os.Stdout
fmt.Printf("%T\n", w) // *os.File

w = new(bytes.Buffer)
fmt.Printf("%T\n", w) // *bytes.Buffer
```

在内部实现中，`fmt` 用反射来拿到接口动态类型的名字。第 12 章将进一步讨论反射。

注意：含有空指针的非空接口

空的接口值（其中不包含任何信息）与仅仅动态值为 nil 的接口值是不一样的。这种微妙的区别成为让每个 Go 程序员都困惑过的陷阱。

考虑如下程序，当 `debug` 设置为 true 时，主函数收集函数 f 的输出到一个缓冲区中：

```
const debug = true

func main() {
    var buf *bytes.Buffer
    if debug {
        buf = new(bytes.Buffer) // 启用输出收集
    }
}
```

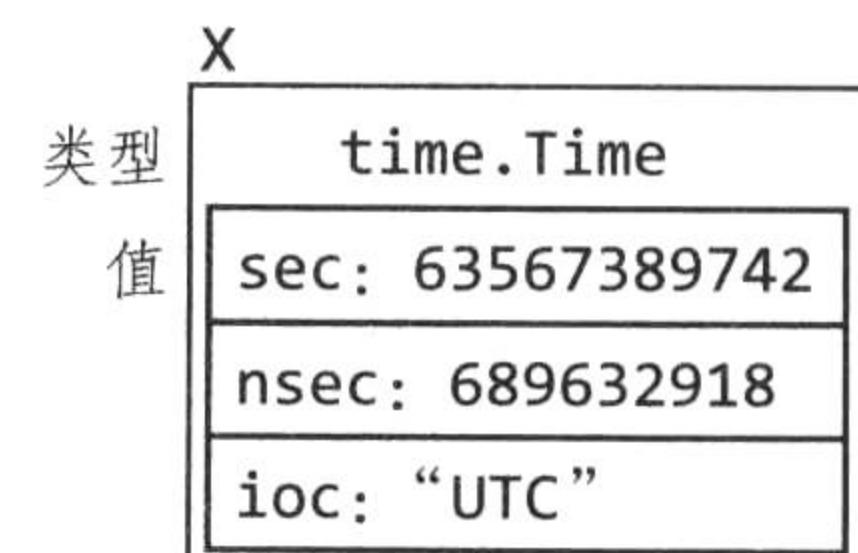


图 7-4 持有 `time.Time` 结构的接口值

```
f(buf) // 注意：微妙的错误
if debug {
    // ...使用 buf...
}

// 如果 out 不是 nil，那么会向其写入输出的数据
func f(out io.Writer) {
    // ...其他代码...
    if out != nil {
        out.Write([]byte("done!\n"))
    }
}
```

当设置 `debug` 为 `false` 时，我们会觉得仅仅是不再收集输出，但实际上会导致程序在调用 `out.Write` 时崩溃：

```
if out != nil {
    out.Write([]byte("done!\n")) // 岌机：对空指针取引用值
}
```

当 `main` 函数调用 `f` 时，它把一个类型为 `*bytes.Buffer` 的空指针赋给了 `out` 参数，所以 `out` 的动态值确实为空。但它的动态类型是 `*bytes.Buffer`，这表示 `out` 是一个包含空指针的非空接口（见图 7-5），所以防御性检查 `out != nil` 仍然是 `true`。

W	*bytes.Buffer
类型	nil

图 7-5 包含空指针的非空接口

如前所述，动态分发机制决定了我们肯定会调用 `(*bytes.Buffer).Write`，只不过这次接收者值为空。对于某些类型，比如 `*os.File`，空接收值是合法的（参考 6.2.1 节），但对于 `*bytes.Buffer` 则不行。方法尽管被调用了，但在尝试访问缓冲区时崩溃了。

问题在于，尽管一个空的 `*bytes.Buffer` 指针拥有的方法满足了该接口，但它并不满足该接口所需的一些行为。特别是，这个调用违背了 `(*bytes.Buffer).Write` 的一个隐式的前置条件，即接收者不能为空，所以把空指针赋给这个接口就是一个错误。解决方案是把 `main` 函数中的 `buf` 类型修改为 `io.Writer`，从而避免在最开始就把一个功能不完整的值赋给一个接口。

```
var buf io.Writer
if debug {
    buf = new(bytes.Buffer) // 启用输出收集
}
f(buf) // OK
```

既然我们已经了解过接口值的机制，接下来就要看一下 Go 标准库的一些重要接口。在接下来的三节中，我们将看到接口在排序、Web 服务、错误处理中的应用。

7.6 使用 `sort.Interface` 来排序

与字符串格式化类似，排序也是一个在很多程序中广泛使用的操作。尽管一个最小的快排（Quicksort）只需 15 行左右，但一个健壮的实现长很多。所以我们无法想象在每次需要时都重新写一遍或者复制一遍。

幸运的是，`sort` 包提供了针对任意序列根据任意排序函数原地排序的功能。这样的设计其实并不常见。在很多语言中，排序算法跟序列数据类型绑定，排序算法则跟序列元素类型绑定。与之相反的是，Go 语言的 `sort.Sort` 函数对序列和其中元素的布局无任何要求，它

使用 `sort.Interface` 接口来指定通用排序算法和每个具体的序列类型之间的协议（contract）。这个接口的实现确定了序列的具体布局（经常是一个 slice），以及元素期望的排序方式。

一个原地排序算法需要知道三个信息：序列长度、比较两个元素的含义以及如何交换两个元素，所以 `sort.Interface` 接口就有三个方法：

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool // i, j 是序列元素的下标
    Swap(i, j int)
}
```

要对序列排序，需要先确定一个实现了如上三个方法的类型，接着把 `sort.Sort` 函数应用到上面这类方法的实例上。我们先考虑几乎是最简单的一个例子：字符串 slice。定义的新类型 `StringSlice` 以及它的 `Len`、`Less`、`Swap` 三个方法如下所示：

```
type StringSlice []string

func (p StringSlice) Len() int { return len(p) }
func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p StringSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }
```

现在就可以对一个字符串 slice 进行排序，只须简单地把一个 slice 转换为 `StringSlice` 类型即可，如下所示：

```
sort.Sort(StringSlice(names))
```

类型转换生成了一个新的 slice，与原始的 `names` 有同样的长度、容量和底层数组，不同的就是额外增加了三个用于排序的方法。

字符串 slice 的排序太常用了，所以 `sort` 包提供了 `StringSlice` 类型，以及一个直接排序的 `Strings` 函数，于是上面的代码可以简写为 `sort.Strings(names)`。

这种技术可以方便地复用到其他排序方式，比如，忽略大小写或者特殊字符。（本书的索引词和页码排序也用了这个技术，只是加了额外的罗马数字逻辑。）对于更复杂的排序，也可以使用同样的思路，只用加上更复杂的数据结构和更复杂的 `sort.Interface` 方法实现。

这里的排序示例将是一个以表格方式显示的音乐播放列表。每首音乐占一行，每个字段都是这首音乐的一个属性，比如艺术家、标题和时间。考虑使用图形用户界面来展示一个表，单击列头会按该列对应的属性来进行排序，再次单击同一个列头会逆序排列。接下来看一下如何响应每一次单击。

如下变量 `tracks` 包含一个播放列表。（作者之一对其他作者的音乐品味表示遗憾。）每个元素都是一个指向 `Track` 的指针。尽管我们不用指针，而改为直接存储 `Tracks`，后面的代码也能运行，考虑到 `sort` 函数要交换很多对元素，所以在元素是一个指针的情况下代码运行速度会更快，毕竟，一个指针的大小只有一个字长，而一个完整的 `Track` 则需要 8 个甚至更多的字。

gopl.io/ch7/sorting

```
type Track struct {
    Title string
    Artist string
    Album string
    Year int
    Length time.Duration
}
```

```

var tracks = []*Track{
    {"Go", "Delilah", "From the Roots Up", 2012, length("3m38s")},
    {"Go", "Moby", "Moby", 1992, length("3m37s")},
    {"Go Ahead", "Alicia Keys", "As I Am", 2007, length("4m36s")},
    {"Ready 2 Go", "Martin Solveig", "Smash", 2011, length("4m24s")},
}

func length(s string) time.Duration {
    d, err := time.ParseDuration(s)
    if err != nil {
        panic(s)
    }
    return d
}

```

`printTracks` 函数将播放列表输出为一个表格。当然，一个图形界面肯定会更好，但这个例程使用的 `text/tabwriter` 包可以生成一个如下所示的干净整洁的表格。注意，`*tabwriter.Writer` 满足 `io.Writer` 接口，它先收集所有写入的数据，在 `Flush` 方法调用时才格式化整个表格并且输出到 `os.Stdout`。

```

func printTracks(tracks []*Track) {
    const format = "%v\t%v\t%v\t%v\t%v\t\n"
    tw := new(tabwriter.Writer).Init(os.Stdout, 0, 8, 2, ' ', 0)
    fmt.Fprintf(tw, format, "Title", "Artist", "Album", "Year", "Length")
    fmt.Fprintf(tw, format, "-----", "-----", "-----", "-----", "-----")
    for _, t := range tracks {
        fmt.Fprintf(tw, format, t.Title, t.Artist, t.Album, t.Year, t.Length)
    }
    tw.Flush() // 计算各列宽度并输出表格
}

```

要按照 `Artist` 字段来对播放列表排序，需要先定义一个新的 `slice` 类型，以及必需的 `Len`、`Less` 和 `Swap` 方法，正如 `StringSlice` 一样。

```

type byArtist []*Track

func (x byArtist) Len() int           { return len(x) }
func (x byArtist) Less(i, j int) bool { return x[i].Artist < x[j].Artist }
func (x byArtist) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }

```

要调用通用的排序例称，必须先把 `tracks` 转换为定义排序规则的新类型 `byArtist`：

```
sort.Sort(byArtist(tracks))
```

按照艺术家排序之后，从 `printTracks` 生成的输出如下：

Title	Artist	Album	Year	Length
-----	-----	-----	-----	-----
Go Ahead	Alicia Keys	As I Am	2007	4m36s
Go	Delilah	From the Roots Up	2012	3m38s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s
Go	Moby	Moby	1992	3m37s

如果用户第二次请求“按照艺术家排序”，就需要把这些音乐反向排序。我们不需要定义一个新的 `byReverseArtist` 类型和对应的反向 `Less` 方法，因为 `sort` 包已经提供了一个 `Reverse` 函数，它可以把任意的排序反向。

按照艺术家对 `slice` 反向排序之后，从 `printTracks` 生成的输出如下：

Title	Artist	Album	Year	Length
Go	Moby	Moby	1992	3m37s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s
Go	Delilah	From the Roots Up	2012	3m38s
Go Ahead	Alicia Keys	As I Am	2007	4m36s

`sort.Reverse` 函数值得仔细看一下，因为它使用了一个重要概念：组合（参考 6.3 节）。`sort` 包定义了一个未导出的类型 `reverse`，这个类型是一个嵌入了 `sort.Interface` 的结构。`reverse` 的 `Less` 方法直接调用了内嵌的 `sort.Interface` 值的 `Less` 方法，但只交换传入的下标，就可以颠倒排序的结果。

```
package sort

type reverse struct{ Interface } // that is, sort.Interface

func (r reverse) Less(i, j int) bool { return r.Interface.Less(j, i) }

func Reverse(data Interface) Interface { return reverse{data} }
```

`reverse` 的另外两个方法 `Len` 和 `Swap`，由内嵌的 `sort.Interface` 隐式提供。导出的函数 `Reverse` 则返回一个包含原始 `sort.Interface` 值的 `reverse` 实例。

如果要按其他列来排序，就需要定义一个新的类型，比如 `byYear`：

```
type byYear []*Track

func (x byYear) Len() int      { return len(x) }
func (x byYear) Less(i, j int) bool { return x[i].Year < x[j].Year }
func (x byYear) Swap(i, j int)   { x[i], x[j] = x[j], x[i] }
```

把 `tracks` 按照 `sort.Sort(byYear(tracks))` 排序后，`printTracks` 就可以输出一个按照年代排序的列表了：

Title	Artist	Album	Year	Length
Go	Moby	Moby	1992	3m37s
Go Ahead	Alicia Keys	As I Am	2007	4m36s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s
Go	Delilah	From the Roots Up	2012	3m38s

对于每一类 `slice` 和每一种排序函数，都需要实现一个新的 `sort.Interface`。如你所见，`Len` 和 `Swap` 方法对所有的 `slice` 类型都是一样的。在下一个例子中，具体类型 `customSort` 组合了一个 `slice` 和一个函数，让我们只写一个比较函数就可以定义一个新的排序。顺便说一下，实现 `sort.Interface` 的具体类型并不一定都是 `slice`，比如 `customSort` 就是一个结构类型：

```
type customSort struct {
    t    []*Track
    less func(x, y *Track) bool
}

func (x customSort) Len() int      { return len(x.t) }
func (x customSort) Less(i, j int) bool { return x.less(x.t[i], x.t[j]) }
func (x customSort) Swap(i, j int)   { x.t[i], x.t[j] = x.t[j], x.t[i] }
```

让我们定义一个多层次的比较函数，先按照标题（`Title`）排序，接着是年份 `Year`，最后是时长 `Length`。如下 `sort` 调用就是一个使用匿名排序函数来这样排序的例子：

```

sort.Sort(customSort{tracks, func(x, y *Track) bool {
    if x.Title != y.Title {
        return x.Title < y.Title
    }
    if x.Year != y.Year {
        return x.Year < y.Year
    }
    if x.Length != y.Length {
        return x.Length < y.Length
    }
    return false
}})
```

如下就是结果。注意，对于两首标题都是“Go”的音乐，年份较早的排序靠前：

Title	Artist	Album	Year	Length
-----	-----	-----	-----	-----
Go	Moby	Moby	1992	3m37s
Go	Delilah	From the Roots Up	2012	3m38s
Go Ahead	Alicia Keys	As I Am	2007	4m36s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s

对一个长度为 n 的序列进行排序需要 $O(n \log n)$ 次比较操作，而判断一个序列是否已经排好序则只需最多 $(n-1)$ 次比较。`sort` 包提供的 `IsSorted` 函数就可以做这个判断。与 `sort.Sort` 类似，它使用 `sort.Interface` 来抽象序列及其排序函数，只是从不调用 `Swap` 方法而已。下面的代码就演示了 `IntsAreSorted`、`Ints` 函数和 `IntSlice` 类型：

```

values := []int{3, 1, 4, 1}
fmt.Println(sort.IntsAreSorted(values)) // "false"
sort.Ints(values)
fmt.Println(values)                  // "[1 1 3 4]"
fmt.Println(sort.IntsAreSorted(values)) // "true"
sort.Sort(sort.Reverse(sort.IntSlice(values)))
fmt.Println(values)                  // "[4 3 1 1]"
fmt.Println(sort.IntsAreSorted(values)) // "false"
```

为了简便起见，`sort` 包专门提供了对于 `[]int`、`[]string`、`[]float64` 自然排序的函数和相关类型。对于其他类型，比如 `[]int64` 或者 `[]uint`，则需要自己写，反正写起来也不复杂。

练习 7.8：很多图形界面提供了一个表格控件，它支持有状态的多层排序：先按照最近单击的列来排序，接着是上一次单击的列，依次类推。请定义 `sort.Interface` 接口实现来满足如上需求。试比较这个方法与多次使用 `sort.Stable` 排序的异同。

练习 7.9：利用 `html/template` 包（见 4.6 节）来替换 `printTracks` 函数，使用 HTML 表格来显示音乐列表。结合上一个练习，来实现通过单击列头来发送 HTTP 请求，进而对表格排序。

练习 7.10：`sort.Interface` 也可以用于其他用途。试写一个函数 `IsPalindrome(s sort.Interface)bool` 来判断一个序列是否是回文，即序列反转后是否保持不变。可以假定对于下标分别为 i 、 j 的元素，如果 $\text{s.Less}(i, j) \&& \text{s.Less}(j, i)$ ，那么两个元素相等。

7.7 http.Handler 接口

第 1 章简单介绍了如何用 `net/http` 包来实现 Web 客户端（参考 1.5 节）和服务器（参考 1.7 节）。本节将进一步讨论服务端 API，以及作为其基础的 `http.Handler` 接口。

```
net/http  
package http  
  
type Handler interface {  
    ServeHTTP(w ResponseWriter, r *Request)  
}  
  
func ListenAndServe(address string, h Handler) error
```

`ListenAndServe` 函数需要一个服务器地址，比如 "localhost:8000"，以及一个 `Handler` 接口的实例（用来接受所有的请求）。这个函数会一直运行，直到服务出错（或者启动时就失败了）时返回一个非空的错误。

设想一个电子商务网站，使用一个数据库来存储商品和价格（以美元计价）的映射。如下程序将展示一个最简单的实现。它用一个 `map` 类型（命名为 `database`）来代表仓库，再加上一个 `ServeHTTP` 方法来满足 `http.Handler` 接口。这个函数遍历整个 `map` 并且输出其中的元素：

```
gopl.io/ch7/http1  
func main() {  
    db := database{"shoes": 50, "socks": 5}  
    log.Fatal(http.ListenAndServe("localhost:8000", db))  
}  
  
type dollars float32  
  
func (d dollars) String() string { return fmt.Sprintf("%.2f", d) }  
  
type database map[string]dollars  
  
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {  
    for item, price := range db {  
        fmt.Fprintf(w, "%s: %s\n", item, price)  
    }  
}
```

如果启动服务器：

```
$ go build gopl.io/ch7/http1  
$ ./http1 &
```

使用 1.5 节的 `fetch` 程序来连接服务器（也可以用 Web 浏览器），可以得到如下输出：

```
$ go build gopl.io/ch1/fetch  
$ ./fetch http://localhost:8000  
shoes: $50.00  
socks: $5.00
```

到现在为止，这个服务器只能列出所有的商品，而且是完全不管 URL，对每个请求都是如此。一个更加真实的服务器会定义多个不同 URL，每个触发不同的行为。我们把现有功能的 URL 设为 `/list`，再加上另外一个 `/price` 用来显示单个商品的价格，商品可以在请求参数中指定，比如 `/price?item=socks`：

```
gopl.io/ch7/http2  
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {  
    switch req.URL.Path {  
    case "/list":  
        for item, price := range db {  
            fmt.Fprintf(w, "%s: %s\n", item, price)  
        }  
    }
```

```

case "/price":
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
default:
    w.WriteHeader(http.StatusNotFound) // 404
    fmt.Fprintf(w, "no such page: %s\n", req.URL)
}
}

```

现在，处理函数基于 URL 的路径部分 (`req.URL.Path`) 来决定执行哪部分逻辑。如果处理函数不能识别这个路径，那么它通过调用 `w.WriteHeader(http.StatusNotFound)` 来返回一个 HTTP 错误。注意，这个调用必须在往 `w` 中写入内容之前完成（顺带说一下，`http.ResponseWriter` 也是一个接口，它扩充了 `io.Writer`，加了发送 HTTP 响应头的方法）。也可以使用 `http.Error` 这个工具函数来达到同样目的。

```

msg := fmt.Sprintf("no such page: %s\n", req.URL)
http.Error(w, msg, http.StatusNotFound) // 404

```

对于 `/price` 的场景，它调用了 URL 的 `Query` 方法，把 HTTP 的请求参数解析为一个 `map`，或者更精确来讲，解析为一个 `multimap`，由 `net/url` 包的 `url.Values` 类型（6.2.1 节）实现。它找到第一个 `item` 请求参数，查询对应的价格。如果商品没找到，则返回一个错误。

与新服务端的交互范例如下所示：

```

$ go build gopl.io/ch7/http2
$ go build gopl.io/ch1/fetch
$ ./http2 &
$ ./fetch http://localhost:8000/list
shoes: $50.00
socks: $5.00
$ ./fetch http://localhost:8000/price?item=socks
$5.00
$ ./fetch http://localhost:8000/price?item=shoes
$50.00
$ ./fetch http://localhost:8000/price?item=hat
no such item: "hat"
$ ./fetch http://localhost:8000/help
no such page: /help

```

显然，可以继续给 `ServeHTTP` 方法增加功能，但对于一个真实的应用，应当把每部分逻辑分到独立的函数或方法。进一步来讲，某些相关的 URL 可能需要类似的逻辑，比如几个图片文件的 URL 可能都是 `/images/*.png` 形式。因为这些原因，`net/http` 包提供了一个请求多工转发器 `ServeMux`，用来简化 URL 和处理程序之间的关联。一个 `ServeMux` 把多个 `http.Handler` 组合成单个 `http.Handler`。在这里，我们再次看到满足同一个接口的多个类型是可以互相替代的，Web 服务器可以把请求分发到任意一个 `http.Handler`，而不用管后面具体的类型是什么。

对于一个更复杂的应用，多个 `ServeMux` 会组合起来，用来处理更复杂的分发需求。Go 语言并没有一个类似于 Ruby 的 Rails 或者 Python 的 Django 那样的权威 Web 框架。这并不

是说那样的框架无法存在，只是 Go 语言的标准库提供的基础单元足够灵活，以至于那样的框架通常不是必需的。进一步来讲，尽管框架在项目初期带来很多便利，但框架带来了额外复杂性，增加长时间维护的难度。

在下面的代码中，创建了一个 `ServeMux`，用于将 `/list`、`/price` 这样的 URL 和对应的处理程序关联起来，这些处理程序也已经拆分到不同的方法中。最后作为主处理程序在 `ListenAndServe` 调用中使用这个 `ServerMux`：

```
gopl.io/ch7/http3
func main() {
    db := database{"shoes": 50, "socks": 5}
    mux := http.NewServeMux()
    mux.HandleFunc("/list", http.HandlerFunc(db.list))
    mux.HandleFunc("/price", http.HandlerFunc(db.price))
    log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
}
```

我们先关注一下用于注册处理程序的两次 `mux.HandleFunc` 调用。在第一个调用中，`db.list` 是一个方法值（参考 6.4 节），即如下类型的一个值：

```
func(w http.ResponseWriter, req *http.Request)
```

当调用 `db.list` 时，等价于以 `db` 为接收者调用 `database.list` 方法。所以 `db.list` 是一个实现了处理功能的函数（而不是一个实例），因为它没有接口所需的方法，所以它不满足 `http.Handler` 接口，也不能直接传给 `mux.HandleFunc`。

表达式 `http.HandlerFunc(db.list)` 其实是类型转换，而不是函数调用。注意，`http.HandlerFunc` 是一个类型。它有如下定义：

```
net/http
package http

type HandlerFunc func(w ResponseWriter, r *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

`HandlerFunc` 演示了 Go 语言接口机制的一些不常见特性。它不仅是一个函数类型，还拥有自己的方法，也满足接口 `http.Handler`。它的 `ServeHTTP` 方法就调用函数本身，所以

`HandlerFunc` 就是一个让函数值满足接口的一个适配器，在这个例子中，函数和接口的唯一方法拥有同样的签名。这个小技巧让 `database` 类型可以用不同的方式来满足 `http.Handler` 接口：一次通过 `list` 方法，一次通过 `price` 方法，依次类推。

因为这种注册处理程序的方法太常见了，所以 `ServeMux` 引入了一个 `HandleFunc` 便捷方法来简化调用，处理程序注册部分的代码可以简化为如下形式：

```
gopl.io/ch7/http3a
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

通过上面的代码，我们可以看到构造这样一个程序也是简单的：有两个不同的 Web 服务器，在不同的端口监听，定义不同的 URL，分发到不同的处理程序。只须简单地构造另外一个 `ServeMux`，再调用一次 `ListenAndServe` 即可（建议并发调用）。但对于绝大部分程序来说，一个 Web 服务就已经远远足够了。另外，一个程序可能在很多文件中来定义 HTTP 处理程序，如果每次都需要显式注册在应用本身的 `ServerMux` 实例上，那就太麻烦了。

所以，为简便起见，`net/http` 包提供一个全局的 `ServeMux` 实例 `DefaultServeMux`，以及包级别的注册函数 `http.Handle` 和 `http.HandleFunc`。要让 `DefaultServeMux` 作为服务器的主处理程序，无须把它传给 `ListenAndServe`，直接传 `nil` 即可。

服务器的主函数可以进一步简化为：

```
gopl.io/ch7/http4
func main() {
    db := database{"shoes": 50, "socks": 5}
    http.HandleFunc("/list", db.list)
    http.HandleFunc("/price", db.price)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

最后有一个重要的提示：1.7 节曾提到，Web 服务器每次都用一个新的 `goroutine` 来调用处理程序，所以处理程序必须要注意并发问题。比如在访问变量时的锁问题，这个变量可能会被其他 `goroutine` 访问，包括由同一个处理程序处理的其他请求。接下来的两章会继续讨论并发问题。

练习 7.11：增加额外的处理程序，来支持创建、读取、更新和删除数据库条目。比如，`/update?item=socks&price=6` 这样的请求将更新仓库中物品的价格，如果商品不存在或者价格无效就返回错误。（注意：这次修改会引入并发变量修改。）

练习 7.12：修改 `/list` 的处理程序，改为输出 HTML 表格，而不是纯文本。可以考虑使用 `html/template` 包（参考 4.6 节）。

7.8 error 接口

从本书的开始，我们就已经使用和创建了神秘的预定义 `error` 类型，但从来没解释过它具体是什么。实际上，它只是一个接口类型，不含一个返回错误消息的方法：

```
type error interface {
    Error() string
}
```

构造 `error` 最简单的方法是调用 `errors.New`，它会返回一个包含指定的错误消息的新

`error` 实例。完整的 `error` 包只有如下 4 行代码：

```
package errors

func New(text string) error { return &errorString{text} }

type errorString struct { text string }

func (e *errorString) Error() string { return e.text }
```

底层的 `errorString` 类型是一个结构，而没有直接用字符串，主要是为了避免将来无意间的（或者有预谋的）布局变更。满足 `error` 接口的是 `*errorString` 指针，而不是原始的 `errorString`，主要是为了让每次 `New` 分配的 `error` 实例都互不相等。我们不希望出现像 `io.EOF` 这样重要的错误，与仅仅包含同样错误消息的一个错误相等。

```
fmt.Println(errors.New("EOF") == errors.New("EOF")) // "false"
```

直接调用 `errors.New` 比较罕见，因为有一个更易用的封装函数 `fmt.Errorf`，它还额外提供了字符串格式化功能。这个函数在第 5 章中我们已经用过几次了。

```
package fmt

import "errors"

func Errorf(format string, args ...interface{}) error {
    return errors.New(Sprintf(format, args...))
}
```

尽管 `*errorString` 可能是最简单的 `error` 类型，但这样简单的 `error` 类型远不止一个。比如，`syscall` 包提供了 Go 语言的底层系统调用 API。在很多平台上，它也定义了一个满足 `error` 接口的数字类型 `Errno`。在 UNIX 平台上，`Errno` 的 `Error` 方法会从一个字符串表格中查询错误消息，如下所示：

```
package syscall

type Errno uintptr // 操作系统错误码
var errors = [...]string{
    1: ". \"operation not permitted\"", // EPERM
    2: " \"no such file or directory\"", // ENOENT
    3: " \"no such process\"", // ESRCH
    // ...
}

func (e Errno) Error() string {
    if 0 <= int(e) && int(e) < len(errors) {
        return errors[e]
    }
    return fmt.Sprintf("errno %d", e)
}
```

如下语句创建一个接口值，其中包含值为 2 的 `Errno`，这个值代表 POSIX `ENOENT` 状态：

```
var err error = syscall.Errno(2)
fmt.Println(err.Error()) // "没有文件或目录"
fmt.Println(err) // "没有文件或目录"
```

`err` 的接口值如图 7-6 所示。

`Errno` 是一个系统调用错误的高效表示手法，毕竟系统调用错误是一个有限的集合，尽管很简单，但是它也满足标

err	
类型	<code>syscall.Errno</code>
值	2

图 7-6 一个包含 `syscall.Errno` 整数的接口值

准的 `error` 接口。在 7.11 节中我们可以看到满足 `error` 接口的其他类型。

7.9 示例：表达式求值器

在本节中，我们将创建简单算术表达式的一个求值器。我们将使用一个接口 `Expr` 来代表这种语言中的任意一个表达式。现在，这个接口没有任何方法，但稍后我们会逐个添加。

```
// Expr: 算术表达式
type Expr interface{}
```

我们的表达式语言包括浮点数字面量，二元操作符 +、-、*、/，一元操作符 -x 和 +x，函数调用 `pow(x, y)`、`sin(x)` 和 `sqrt(x)`，变量（比如 `x` 和 `pi`），当然，还有圆括号和标准的操作符优先级。所有的值都是 `float64` 类型。下面是几个示例表达式：

```
sqrt(A / pi)
pow(x, 3) + pow(y, 3)
(F - 32) * 5 / 9
```

下面 5 种具体类型代表特定类型的表达式。`Var` 代表变量应用（很快我们将了解到为什么这个类型需要导出）。`literal` 代表浮点数常量。`unary` 和 `binary` 类型代表有一个或者两个操作数的操作符表达式，而操作数则可以任意的 `Expr`。`call` 代表函数调用，这里限制它的 `fn` 字段只能是 `pow`、`sin` 和 `sqrt`。

gopl.io/ch7/eval

```
// Var 表示一个变量，比如 x
type Var string

// literal 是一个数字常量，比如 3.141
type literal float64

// unary 表示一元操作符表达式，比如 -x
type unary struct {
    op rune // '+', '-' 中的一个
    x   Expr
}

// binary 表示二元操作符表达式，比如 x+y
type binary struct {
    op   rune // '+', '-', '*', '/' 中的一个
    x, y Expr
}

// call 表示函数调用表达式，比如 sin(x)
type call struct {
    fn   string // one of "pow", "sin", "sqrt" 中的一个
    args []Expr
}
```

要对包含变量的表达式进行求值，需要一个上下文（environment）来把变量映射到数值：

```
type Env map[Var]float64
```

我们还需要为每种类型的表达式定义一个 `Eval` 方法来返回表达式在一个给定上下文下的值。既然每个表达式都必须提供这个方法，那么可以把它加到 `Expr` 接口中。这个包只导出了类型 `Expr`、`Env` 和 `Var`。客户端可以在不接触其他表达式类型的情况下使用这个求值器。

```
type Expr interface {
    // Eval 返回表达式在 env 上下文下的值
    Eval(env Env) float64
}
```

下面是具体的 `Eval` 方法。`Var` 的 `Eval` 方法从上下文中查询结果，如果变量不存在则返回 0。`literal` 的 `Eval` 方法则直接返回本身的值。

```
func (v Var) Eval(env Env) float64 {
    return env[v]
}

func (l literal) Eval(_ Env) float64 {
    return float64(l)
}
```

`unary` 和 `binary` 的 `Eval` 方法首先对它们的操作数递归求值，然后应用 `op` 操作。我们不把除以 0 或者无穷大当做错误（尽管它们生成的结果显然不是有穷数）。最后，`call` 方法先对 `pow`、`sin` 或者 `sqrt` 函数的参数求值，再调用 `math` 包中的对应函数。

```
func (u unary) Eval(env Env) float64 {
    switch u.op {
    case '+':
        return +u.x.Eval(env)
    case '-':
        return -u.x.Eval(env)
    }
    panic(fmt.Sprintf("unsupported unary operator: %q", u.op))
}

func (b binary) Eval(env Env) float64 {
    switch b.op {
    case '+':
        return b.x.Eval(env) + b.y.Eval(env)
    case '-':
        return b.x.Eval(env) - b.y.Eval(env)
    case '*':
        return b.x.Eval(env) * b.y.Eval(env)
    case '/':
        return b.x.Eval(env) / b.y.Eval(env)
    }
    panic(fmt.Sprintf("unsupported binary operator: %q", b.op))
}

func (c call) Eval(env Env) float64 {
    switch c.fn {
    case "pow":
        return math.Pow(c.args[0].Eval(env), c.args[1].Eval(env))
    case "sin":
        return math.Sin(c.args[0].Eval(env))
    case "sqrt":
        return math.Sqrt(c.args[0].Eval(env))
    }
    panic(fmt.Sprintf("unsupported function call: %s", c.fn))
}
```

某些方法可能会失败，比如 `call` 表达式可能会遇到未知的函数，或者参数数量不对。也有可能用 “!” 或者 “<” 这类无效的操作符构造了一个 `unary` 或 `binary` 表达式（尽管后面的 `Parse` 函数不会产生这样的结果）。这些错误都会导致 `Eval` 崩溃。其他错误（比如对一个上下文中没有定义的变量求值）仅会导致返回不正确的结果。所有这些错误都可以在求值之前做检查来发现。后面的 `Check` 方法就负责完成这个任务，但我们先测试 `Eval`。

下面的 `TestEval` 函数用于测试求值器，它使用 `testing` 包。`testing` 包的详细情况会在第 11 章介绍，现在我们只须知道调用 `t.Errorf` 来报告错误。这个函数遍历一个表格，表格中定义了三个表达式并为每个表达式准备了不同上下文。第一个表达式用于根据圆面积 `A` 求半径，第二个用于计算两个变量 `x` 和 `y` 的立方和，第三个把华氏温度 `F` 转为摄氏温度。

```
func TestEval(t *testing.T) {
    tests := []struct {
        expr string
        env  Env
        want string
    }{
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 12, "y": 1}, "1729"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
        {"5 / 9 * (F - 32)", Env{"F": 32}, "0"},
        {"5 / 9 * (F - 32)", Env{"F": 212}, "100"},
    }
    var prevExpr string
    for _, test := range tests {
        // 仅在表达式变更时才输出
        if test.expr != prevExpr {
            fmt.Printf("\n%s\n", test.expr)
            prevExpr = test.expr
        }
        expr, err := Parse(test.expr)
        if err != nil {
            t.Errorf("Parse(%s) failed: %v", test.expr, err)
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        fmt.Printf("\t%v => %s\n", test.expr, got)
        if got != test.want {
            t.Errorf("%s.Eval() = %v, want %v\n",
                test.expr, got, test.want)
        }
    }
}
```

对于表格中的每一行记录，该测试先解析表达式，在上下文中求值，再输出表达式。这里没有足够的空间来显示 `Parse` 函数，但可以通过 `go get` 来下载源码，自行查看。

`go test` 命令（参考 11.1 节）可用于运行包的测试：

```
$ go test -v gopl.io/ch7/eval
```

启用 `-v` 选项后可以看到测试的输出，通常情况下对于结果正确的测试输出就不显示了。下面就是测试中 `fmt.Printf` 语句输出的内容。

```
sqrt(A / pi)
map[A:87616 pi:3.141592653589793] => 167

pow(x, 3) + pow(y, 3)
map[x:12 y:1] => 1729
map[x:9 y:10] => 1729

5 / 9 * (F - 32)
map[F:-40] => -40
map[F:32] => 0
map[F:212] => 100
```

幸运的是，到现在为止所有的输入都是合法的，但这种幸运是不能持久的。即使在解释性语言中，通过语法检查来发现静态错误（即不用运行程序也能检测出来的错误）也是很常见的。通过分离静态检查和动态检查，我们可以更快发现错误，也可以只在运行前检查一次，而不用在表达式求值时每次都检查。

让我们给 Expr 方法加上另外一个方法。Check 方法用于在表达式语法树上检查静态错误。它的 vars 参数将稍后解释。

```
type Expr interface {
    Eval(env Env) float64
    // Check 方法报告表达式中的错误，并把表达式中的变量加入 Vars 中
    Check(vars map[Var]bool) error
}
```

具体的 Check 方法如下所示。literal 和 Var 的求值不可能出错，所以 Check 方法返回 nil。unary 和 binary 的方法首先检查操作符是否合法，再递归地检查操作数。类似地，call 的方法首先检查函数是否是已知的，然后检查参数个数是否正确，最后递归检查每个参数。

```
func (v Var) Check(vars map[Var]bool) error {
    vars[v] = true
    return nil
}

func (literal) Check(vars map[Var]bool) error {
    return nil
}

func (u unary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-", u.op) {
        return fmt.Errorf("unexpected unary op %q", u.op)
    }
    return u.x.Check(vars)
}

func (b binary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-*/", b.op) {
        return fmt.Errorf("unexpected binary op %q", b.op)
    }
    if err := b.x.Check(vars); err != nil {
        return err
    }
    return b.y.Check(vars)
}

func (c call) Check(vars map[Var]bool) error {
    arity, ok := numParams[c.fn]
    if !ok {
        return fmt.Errorf("unknown function %q", c.fn)
    }
    if len(c.args) != arity {
        return fmt.Errorf("call to %s has %d args, want %d",
            c.fn, len(c.args), arity)
    }
    for _, arg := range c.args {
        if err := arg.Check(vars); err != nil {
            return err
        }
    }
    return nil
}

var numParams = map[string]int{"pow": 2, "sin": 1, "sqrt": 1}
```

下面分两列展示了一些有错误的输入，以及它们触发的错误。Parse 函数（没有显示）报告了语法错误，Check 方法报告了语义错误。

```
x % 2           unexpected '%'
math.Pi          unexpected '.'
!true           unexpected '!'
"hello"         unexpected ''
log(10)         unknown function "log"
sqrt(1, 2)      call to sqrt has 2 args, want 1
```

Check 的输入参数是一个 Var 集合，它收集在表达中发现的变量名。要让表达式能成功求值，上下文必须包含所有的这些变量。从逻辑上来讲，这个集合应当是 Check 的输出结果而不是输入参数，但因为这个方法是递归调用的，在这种情况下使用参数更为方便。调用方在最初调用时需要提供一个空的集合。

在 3.2 节，我们绘制了一个函数 $f(x,y)$ ，不过函数是在编译时指定的。既然我们可以对字符串形式的表达式进行解析、检查和求值，那么就可以构建一个 Web 应用，在运行时从客户端接收一个表达式，并绘制函数的曲面图。可以使用 vars 集合来检查表达式是一个只有两个变量 x 、 y 的函数（为了简单起见，还提供了半径 r ，所以实际上是 3 个变量）。使用 Check 方法来拒绝掉不规范的表达式，避免了在接下来的 40000 次求值中重复检查（4 个象限中 100×100 的格子）。

下面的 parseAndCheck 函数组合了解析和检查步骤：

```
gopl.io/ch7/surface
import "gopl.io/ch7/eval"

func parseAndCheck(s string) (eval.Expr, error) {
    if s == "" {
        return nil, fmt.Errorf("empty expression")
    }
    expr, err := eval.Parse(s)
    if err != nil {
        return nil, err
    }
    vars := make(map[eval.Var]bool)
    if err := expr.Check(vars); err != nil {
        return nil, err
    }
    for v := range vars {
        if v != "x" && v != "y" && v != "r" {
            return nil, fmt.Errorf("undefined variable: %s", v)
        }
    }
    return expr, nil
}
```

要构造完这个 Web 应用，仅需要增加下面的 plot 函数，其函数签名与 http.HandlerFunc 类似：

```
func plot(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    expr, err := parseAndCheck(r.Form.Get("expr"))
    if err != nil {
        http.Error(w, "bad expr: "+err.Error(), http.StatusBadRequest)
        return
    }
```

```
w.Header().Set("Content-Type", "image/svg+xml")
surface(w, func(x, y float64) float64 {
    r := math.Hypot(x, y) // 与 (0,0) 之间的距离
    return expr.Eval(eval.Env{"x": x, "y": y, "r": r})
})
}
```

`plot` 函数解析并检查 HTTP 请求中的表达式，并用它来创建一个有两个变量的匿名函数。这个匿名函数与原始曲面图绘制程序中的 `f` 有同样的签名，且能对用户提供的表达式进行求值。上下文定义了 `x`、`y` 和半径 `r`。最后，`plot` 调用了 `surface` 函数，`surface` 函数来自 `gopl.io/ch3/surface` 中的 `main` 函数，略做修改，加了参数用于接受绘制函数和输出用的 `io.Writer`，原始版本直接使用了函数 `f` 和 `os.Stdout`。图 7-7 显示了用这个程序绘制的三张曲面图。

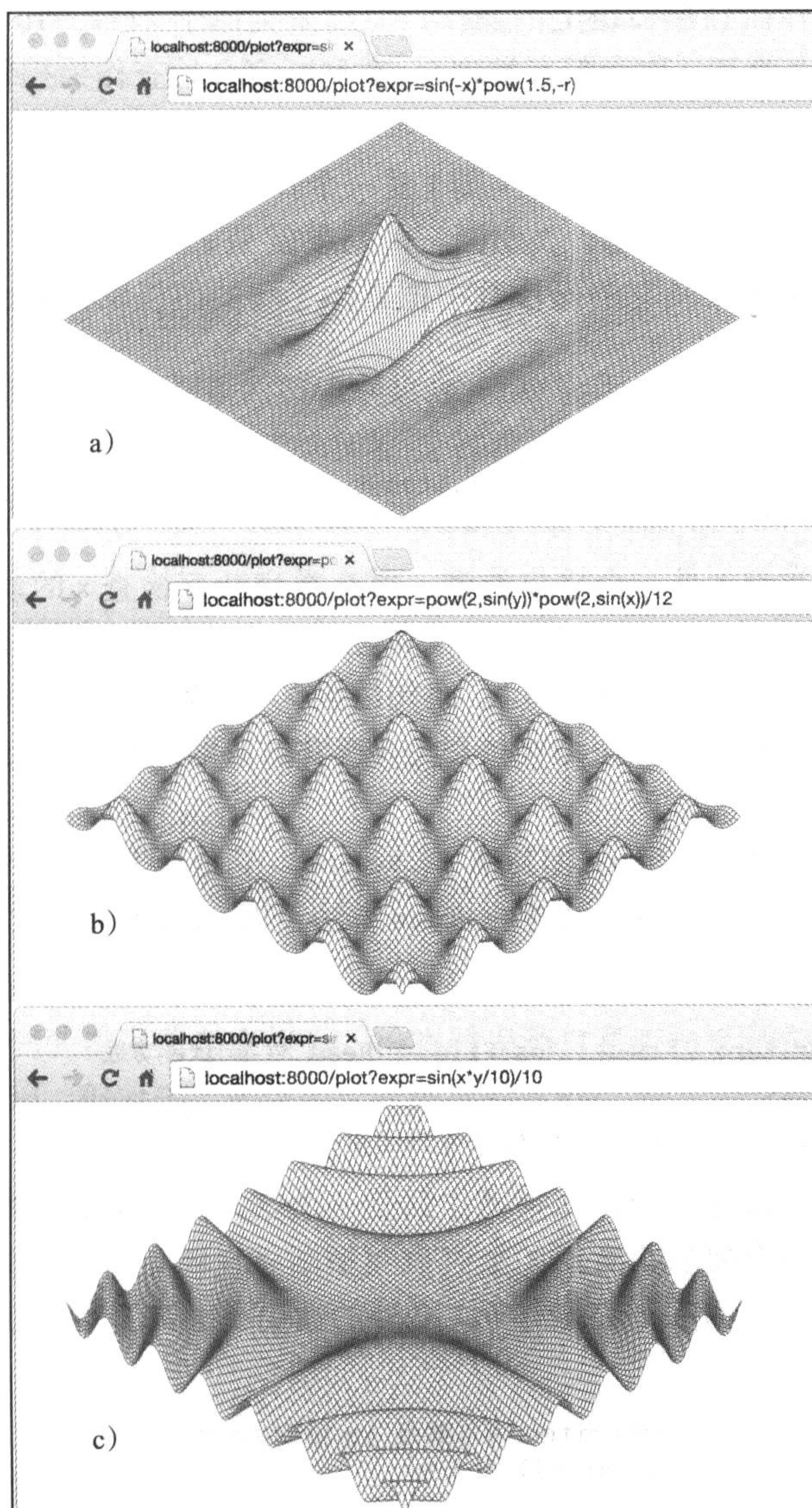


图 7-7 三个函数的曲面图：a) $\sin(-x) * \text{pow}(1.5, -r)$; b) $\text{pow}(2, \sin(y)) * \text{pow}(2, \sin(x)) / 12$;
c) $\sin(x * y / 10) / 10$

练习 7.13: 给 Expr 增加一个 `String` 方法用来美化输出语法树。要求生成的语法树重新解析后是完全一致的树。

练习 7.14: 定义一个新的满足 Expr 接口的具体类，提供一个新操作，比如计算它的操作数的最小值。因为 Parse 函数无法实例化新创建的类型，所以测试时需要直接构造语法树（当然，也可以扩充一下解析函数）。

练习 7.15: 写一个程序从标准输入读取一个表达式，提示用户输入表达式中变量的值，最后计算表达式的值。请妥善处理各种异常。

练习 7.16: 写一个基于 Web 的计算器程序。

7.10 类型断言

类型断言是一个作用在接口值上的操作，写出来类似于 `x.(T)`，其中 `x` 是一个接口类型的表达式，而 `T` 是一个类型（称为断言类型）。类型断言会检查作为操作数的动态类型是否满足指定的断言类型。

这儿有两个可能。首先，如果断言类型 `T` 是一个具体类型，那么类型断言会检查 `x` 的动态类型是否就是 `T`。如果检查成功，类型断言的结果就是 `x` 的动态值，类型当然就是 `T`。换句话说，类型断言就是用来从它的操作数中把具体类型的值提取出来的操作。如果检查失败，那么操作崩溃。比如：

```
var w io.Writer
w = os.Stdout
f := w.(*os.File)      // 成功: f == os.Stdout
c := w.(*bytes.Buffer) // 崩溃: 接口持有的是 *os.File, 不是 *bytes.Buffer
```

其次，如果断言类型 `T` 是一个接口类型，那么类型断言检查 `x` 的动态类型是否满足 `T`。如果检查成功，动态值并没有提取出来，结果仍然是一个接口值，接口值的类型和值部分也没有变更，只是结果的类型为接口类型 `T`。换句话说，类型断言是一个接口值表达式，从一个接口类型变为拥有另外一套方法的接口类型（通常方法数量是增多），但保留了接口值中的动态类型和动态值部分。

如下类型断言代码中，`w` 和 `rw` 都持有 `os.Stdout`，于是所有对应的动态类型都是 `*os.File`，但 `w` 作为 `io.Writer` 仅暴露了文件的 `Write` 方法，而 `rw` 还暴露了它的 `Read` 方法。

```
var w io.Writer
w = os.Stdout
rw := w.(io.ReadWriter) // 成功: *os.File 有 Read 和 Write 方法
w = new(ByteCounter)
rw = w.(io.ReadWriter) // 崩溃: *ByteCounter 没有 Read 方法
```

无论哪种类型作为断言类型，如果操作数是一个空接口值，类型断言都失败。很少需要从一个接口类型向一个要求更宽松的类型做类型断言，该宽松类型的接口方法比原类型的少，而且是其子集。因为除了在操作 `nil` 之外的情况下，在其他情况下这种操作与赋值一致。

```
w = rw          // io.ReadWriter 可以赋给 io.Writer
w = rw.(io.Writer) // 仅当 rw == nil 时失败
```

我们经常无法确定一个接口值的动态类型，这时就需要检测它是否是某一个特定类型。如果类型断言出现在需要两个结果的赋值表达式（比如如下的代码）中，那么断言不会在失

败时崩溃，而是会多返回一个布尔型的返回值来指示断言是否成功。

```
var w io.Writer = os.Stdout
f, ok := w.(*os.File)      // 成功: ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // 失败: !ok, b == nil
```

按照惯例，一般把第二个返回值赋给一个名为 `ok` 的变量。如果操作失败，`ok` 为 `false`，而第一个返回值为断言类型的零值，在这个例子中就是 `*bytes.Buffer` 的空指针。

`ok` 返回值通常马上就用来决定下一步做什么。下面 `if` 表达式的扩展形式就可以让我们写出相当紧凑的代码：

```
if f, ok := w.(*os.File); ok {
    // ...使用 f...
}
```

当类型断言的操作数是一个变量时，有时你会看到返回值的名字与操作数变量名一致，原有的值就被新的返回值掩盖了，比如：

```
if w, ok := w.(*os.File); ok {
    // ...use w...
}
```

7.11 使用类型断言来识别错误

考虑一下 `os` 包中的文件操作返回的错误集合，I/O 会因为很多原因失败，但有三类原因通常必须单独处理：文件已存储（创建操作），文件没找到（读取操作）以及权限不足。`os` 包提供了三个帮助函数用来对错误进行分类：

```
package os

func IsExist(err error) bool
func IsNotExists(err error) bool
func IsPermission(err error) bool
```

一个幼稚的实现会通过检查错误消息是否包含特定的字符串来做判断：

```
func IsNotExists(err error) bool {
    // 注意：不健壮
    return strings.Contains(err.Error(), "file does not exist")
}
```

但由于处理 I/O 错误的逻辑会随着平台的变化而变化，因此这种方法很不健壮，同样的错误可能会用完全不同的错误消息来报告。检查错误消息是否包含特定的字符串，这种方法在单元测试中还算够用，但对于生产级的代码则远远不够。

一个更可靠的方法是用专门的类型来表示结构化的错误值。`os` 包定义了一个 `PathError` 类型来表示在与一个文件路径相关的操作上发生错误（比如 `Open` 或者 `Delete`），一个类似的 `LinkError` 用来表述在与两个文件路径相关的操作上发生错误（比如 `Symlink` 和 `Rename`）。下面是 `os.PathError` 的定义：

```
package os

// PathError 记录了错误以及错误相关的操作和文件路径
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

```
func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

很多客户端忽略了 `PathError`, 改用一种统一的方法来处理所有的错误, 即调用 `Error` 方法。`PathError` 的 `Error` 方法只是拼接了所有的字段, 而 `PathError` 的结构则保留了错误所有的底层信息。对于那些需要区分错误的客户端, 可以使用类型断言来检查错误的特定类型, 这些类型包含的细节远远多于一个简单的字符串。

```
_ , err := os.Open("/no/such/file")
fmt.Println(err) // "open /no/such/file: No such file or directory"
fmt.Printf("%#v\n", err)
// 输出:
// &os.PathError{Op:"open", Path:"/no/such/file", Err:0x2}
```

这也是之前三个帮助函数的工作方式。比如, 如下所示的 `IsNotExist` 判断错误是否等于 `syscall.ENOENT` (参见 7.8 节), 或者等于另一个错误 `os.ErrNotExist` (参见 5.4.2 节的 `io.EOF`), 或者是一个 `*PathError`, 并且底层的错误是上面二者之一。

```
import (
    "errors"
    "syscall"
)

var ErrNotExist = errors.New("file does not exist")

// IsNotExist 返回一个布尔值, 该值表明错误是否代表文件或目录不存在
// report that a file or directory does not exist. It is satisfied by
// ErrNotExist 和其他一些系统调用错误会返回 true
func IsNotExist(err error) bool {
    if pe, ok := err.(*PathError); ok {
        err = pe.Err
    }
    return err == syscall.ENOENT || err == ErrNotExist
}
```

实际使用情况如下:

```
_ , err := os.Open("/no/such/file")
fmt.Println(os.IsNotExist(err)) // "true"
```

当然, 如果错误消息已被 `fmt.Errorf` 这类的方法合并到一个大字符串中, 那么 `PathError` 的结构信息就丢失了。错误识别通常必须在失败操作发生时马上处理, 而不是等到错误消息返回给调用者之后。

7.12 通过接口类型断言来查询特性

下面这段代码的逻辑类似于 `net/http` 包中的 Web 服务器向客户端响应诸如 "Content-type:text/html" 这样的 HTTP 头字段。`io.Writer w` 代表 HTTP 响应, 写入的字节最终会发到某人的 Web 浏览器上。

```
func writeHeader(w io.Writer, contentType string) error {
    if _, err := w.Write([]byte("Content-Type: ")); err != nil {
        return err
    }
    if _, err := w.Write([]byte(contentType)); err != nil {
        return err
    }
    // ...
}
```

因为 `Write` 方法需要一个字节 `slice`，而我们想写入的是一个字符串，所以 `[]byte(...)` 转换就是必需的。这种转换需要进行内存分配和内存复制，但复制后的内存又会被马上抛弃。让我们假装这是 Web 服务器的核心部分，而且性能分析表明这个内存分配导致性能下降。那么我们能否避开内存分配呢？

从 `io.Writer` 接口我们仅仅能知道 `w` 中具体类型的一个信息，那就是可以写入字节 `slice`。但如果我们深入 `net/http` 包查看，可以看到 `w` 对应的动态类型还支持一个能高效写入字符串的 `WriteString` 方法，这个方法避免了临时内存的分配和复制。（这个有点盲目猜测，但很多实现了 `io.Writer` 的重要类也有 `WriteString` 方法，比如 `*bytes.Buffer`、`*os.File` 和 `*bufio.Writer`。）

我们无法假定任意一个 `io.Writer w` 也有 `WriteString` 方法。但可以定义一个新的接口，这个接口只包含 `WriteString` 方法，然后使用类型断言来判断 `w` 的动态类型是否满足这个新接口。

```
// writeString 将 s 写入 w
// 如果 w 有 WriteString 方法，那么将直接调用该方法
func writeString(w io.Writer, s string) (n int, err error) {
    type stringWriter interface {
        WriteString(string) (n int, err error)
    }
    if sw, ok := w.(stringWriter); ok {
        return sw.WriteString(s) // 避免了内存复制
    }
    return w.Write([]byte(s)) // 分配了临时内存
}

func writeHeader(w io.Writer, contentType string) error {
    if _, err := writeString(w, "Content-Type: "); err != nil {
        return err
    }
    if _, err := writeString(w, contentType); err != nil {
        return err
    }
    // ...
}
```

为了避免代码重复，我们把检查挪到了工具函数 `writeString` 中。实际上，标准库提供了 `io.WriteString`，而且这也是向 `io.Writer` 写入字符串的推荐方法。

这个例子中比较古怪的地方是并没有一个标准的接口定义了 `WriteString` 方法并且指定它应满足的规范。进一步讲，一个具体的类型是否满足 `stringWriter` 接口仅仅由它拥有的方法来决定，而不是这个类型与一个接口类型之间的一个关系声明。这意味着上面的技术依赖于一个假定，即如果一个类型满足下面的接口，那么 `WriteString(s)` 必须与 `Write([]byte(s))` 等效。

```
interface {
    io.Writer
    WriteString(s string) (n int, err error)
}
```

尽管 `io.WriteString` 文档中提到了这个假定，但在调用它的函数的文档中就很少提到这个假定了。给一个特定类型多定义一个方法，就隐式地接受了一个特性约定。Go 语言的初学者，特别是那些具有强类型语言背景的人，会对这种缺乏显式约定的方式感到不安，但在

实践中很少产生问题。撇开空接口 `interface{}` 不谈，很少有因为无意识的巧合导致错误的接口匹配。

前面的 `writeString` 函数使用类型断言来判定一个更普适接口类型的值是否满足一个更专用的接口类型，如果满足，则可以使用后者所定义的方法。这种技术不仅适用于 `io.ReadWriter` 这种标准接口，还适用于 `stringWriter` 这种自定义类型。

这个方法也用在了 `fmt.Printf` 中，用于从通用类型中识别出 `error` 或者 `fmt.Stringer`。在 `fmt.Fprintf` 内部，有一步是把单个操作数转换为一个字符串，如下所示：

```
package fmt

func formatOneValue(x interface{}) string {
    if err, ok := x.(error); ok {
        return err.Error()
    }
    if str, ok := x.(Stringer); ok {
        return str.String()
    }
    // ...所有其他类型...
}
```

如果 `x` 满足这两种接口中的一个，就直接确定格式化方法。如果不满足，默认处理部分大致会使用反射来处理所有其他类型，详细情况在第 12 章讨论。

再说一次，上面的代码给出了一个假定，任何有 `String` 方法的类型都满足了 `fmt.Stringer` 的约定，即把类型转化为一个适合输出的字符串。

7.13 类型分支

接口有两种不同的风格。第一种风格下，典型的比如 `io.Reader`、`io.Writer`、`fmt.Stringer`、`sort.Interface`、`http.Handler` 和 `error`，接口上的各种方法突出了满足这个接口的具体类型之间的相似性，但隐藏了各个具体类型的布局和各自特有的功能。这种风格强调了方法，而不是具体类型。

第二种风格则充分利用了接口值能够容纳各种具体类型的能力，它把接口作为这些类型的联合（union）来使用。类型断言用来在运行时区分这些类型并分别处理。在这种风格中，强调的是满足这个接口的具体类型，而不是这个接口的方法（何况经常没有），也不注重信息隐藏。我们把这种风格的接口使用方式称为可识别联合（discriminated union）。

如果你对面向对象编程很熟悉，那么你就知道这两种风格分别对应子类型多态（subtype polymorphism）和特设多态（ad hoc polymorphism），当然这些名词并不重要。本章其余部分将结合示例对第二种风格的接口进行讲解。

与其他语言一样，Go 语言的数据库 SQL 查询 API 也允许我们干净地分离查询中的不变部分和可变部分。一个示例客户端如下所示：

```
import "database/sql"

func listTracks(db sql.DB, artist string, minYear, maxYear int) {
    result, err := db.Exec(
        "SELECT * FROM tracks WHERE artist = ? AND ? <= year AND year <= ?",
        artist, minYear, maxYear)
    // ...
}
```

`Exec` 方法把查询字符串中的每一个“?”都替换为与相应参数值对应的 SQL 字面量，这些参数可能是布尔型、数字、字符串或者 `nil`。通过这种方式构造请求可以帮助避免 SQL 注入攻击，攻击者可以通过在输入数据中加入不恰当的引号来控制你的查询。在 `Exec` 的实现代码中，可以发现一个类似如下的函数，将每个参数值转为对应的 SQL 字面量。

```
func sqlQuote(x interface{}) string {
    if x == nil {
        return "NULL"
    } else if _, ok := x.(int); ok {
        return fmt.Sprintf("%d", x)
    } else if _, ok := x.(uint); ok {
        return fmt.Sprintf("%d", x)
    } else if b, ok := x.(bool); ok {
        if b {
            return "TRUE"
        }
        return "FALSE"
    } else if s, ok := x.(string); ok {
        return sqlQuoteString(s) // (not shown)
    } else {
        panic(fmt.Sprintf("unexpected type %T: %v", x, x))
    }
}
```

一个 `switch` 语句可以把包含一长串值相等比较的 `if-else` 语句简化掉。一个相似的类型分支（type switch）语句则可以用来简化一长串的类型断言 `if-else` 语句。

类型分支的最简单形式与普通分支语句类似，两个的差别是操作数改为 `x.(type)`（注意：这里直接写关键词 `type`，而不是一个特定类型），每个分支是一个或者多个类型。类型分支的分支判定基于接口值的动态类型，其中 `nil` 分支需要 `x == nil`，而 `default` 分支则在其他分支都没有满足时才运行。`sqlQuote` 的类型分支会有如下几个：

```
switch x.(type) {
case nil:      // ...
case int, uint: // ...
case bool:     // ...
case string:   // ...
default:       // ...
}
```

与普通的 `switch` 语句（参考 1.8 节）类似，分支是按顺序来判定的，当一个分支符合时，对应的代码会执行。分支的顺序在一个或多个分支是接口类型时会变得重要，因为有可能两个分支都能满足。`default` 分支的位置是无关紧要的。另外，类型分支不允许使用 `fallthrough`。

注意，在原来的代码中，`bool` 和 `string` 分支的逻辑需要访问由类型断言提取出来的原始值。这个需求比较典型，所以类型分支语句也有一种扩展形式，它用来把每个分支中提取出来的原始值绑定到一个新的变量：

```
switch x := x.(type) { /* ... */ }
```

这里把新的变量也命名为 `x`，与类型断言类似，重用变量名也很普遍。与 `switch` 语句类似，类型分支也隐式创建了一个词法块，所以声明一个新变量叫 `x` 并不与外部块中的变量 `x` 冲突。每个分支也会隐式创建各自的词法块。

用类型分支的扩展形式重写后的 `sqlQuote` 就更加清晰易读了：

```

func sqlQuote(x interface{}) string {
    switch x := x.(type) {
    case nil:
        return "NULL"
    case int, uint:
        return fmt.Sprintf("%d", x) // 这里 x 类型为 interface{}
    case bool:
        if x {
            return "TRUE"
        }
        return "FALSE"
    case string:
        return sqlQuoteString(x) // (未显示具体代码)
    default:
        panic(fmt.Sprintf("unexpected type %T: %v", x, x))
    }
}

```

在这个版本中，每个单一类型的分支块内，变量 `x` 的类型都与该分支的类型一致。比如，在 `bool` 分支中 `x` 的类型是 `bool`，在 `string` 分支中则是 `string`。在其他分支中，`x` 的类型则与 `switch` 的操作数一致，在这个例子中就是 `interface{}`。如果多个分支执行的代码一致，比如本例中的 `int` 和 `uint`，使用类型分支语句就方便很多。

尽管 `sqlQuote` 支持任意类型的实参，但仅当实参类型能够符合类型分支中的一个时才能正常运行到结束，对于其他情况就会崩溃并抛出一条“unexpected type”（非期望类型）消息。表面上 `x` 的类型是 `interface{}`，实际上我们把它当作 `int`、`uint`、`bool`、`string` 和 `nil` 的一个可识别联合。

7.14 示例：基于标记的 XML 解析

4.5 节展示了如何用 `encoding/json` 包的 `Marshal` 和 `Unmarshal` 函数来把 JSON 文档解析为 Go 语言的数据结构。`encoding/xml` 包提供了一个相似的 API。当需要构造一个完整文档树的结构时这很方便，但对于很多程序这是不必要的。`encoding/xml` 还为解析 API 提供了一个基于标记的底层 XML。在这些 API 中，解析器读入输入文本，然后输出一个标记流。标记流中主要包含四种类型：`StartElement`、`EndElement`、`CharData` 和 `Comment`，这四种类型都是 `encoding/xml` 包中的一个具体类型。每次调用 `(*xml.Decoder).Token` 都会返回一个标记。

API 相关的部分如下所示。

```

encoding/xml

package xml

type Name struct {
    Local string // 比如 "Title" 或者 "id"
}

type Attr struct { // 比如 name="value"
    Name Name
    Value string
}

// Token 包括 StartElement、EndElement、CharData 和 Comment
// 以及其他一些晦涩的类型（未显示）
type Token interface{}

type StartElement struct { // 比如 <name>
    Name Name
    Attr []Attr
}

```

```
type EndElement struct { Name Name } // 比如 </name>
type CharData []byte           // 比如 <p>CharData</p>
type Comment []byte            // 比如 <!-- Comment -->

type Decoder struct{ /* ... */ }

func NewDecoder(io.Reader) *Decoder
func (*Decoder) Token() (Token, error) // 返回序列中的下一个标记
```

Token 的接口没有任何方法，这也是一个可识别联合的典型示例。一个传统的接口（比如 io.Reader）的目标是隐藏具体类型的细节，这样可以轻松创建满足接口的新实现，对于每一种实现，使用方的处理方式都是一样的。可识别联合类型的接口正好与之相反，它的实现类型是固定的而不是随意增加的，实现类型是暴露的而不是隐藏的。可识别联合类型很少有方法，操作它的函数经常会使用类型 switch，然后对每种类型应用不同的逻辑。

下面的 xmlselect 程序提取并输出 XML 文档树中特定元素下的文本。利用上面的 API，可以在一遍扫描中就完成这个任务，还不用生成相应的文档树。

```
gopl.io/ch7/xmlselect
// Xmlselect 输出 XML 文档中指定元素下的文本
package main

import (
    "encoding/xml"
    "fmt"
    "io"
    "os"
    "strings"
)

func main() {
    dec := xml.NewDecoder(os.Stdin)
    var stack []string // 元素名的栈
    for {
        tok, err := dec.Token()
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Fprintf(os.Stderr, "xmlselect: %v\n", err)
            os.Exit(1)
        }
        switch tok := tok.(type) {
        case xml.StartElement:
            stack = append(stack, tok.Name.Local) // 入栈
        case xml.EndElement:
            stack = stack[:len(stack)-1] // 出栈
        case xml.CharData:
            if containsAll(stack, os.Args[1:]) {
                fmt.Printf("%s: %s\n", strings.Join(stack, " "), tok)
            }
        }
    }
}

// containsAll 判断 x 是否包含 y 中的所有元素，且顺序一致
func containsAll(x, y []string) bool {
    for len(y) <= len(x) {
        if len(y) == 0 {
            return true
        }
    }
}
```

```

    if x[0] == y[0] {
        y = y[1:]
    }
    x = x[1:]
}
return false
}

```

在 `main` 函数的每次循环中，如果遇到 `StartElement`，就把元素的名字入栈，遇到 `EndElement` 则把元素名字出栈。API 保证了 `StartElement` 和 `EndElement` 标记是正确匹配的，对于不规范的文档也是如此。`Comments` 被忽略了。当 `xmlselect` 遇到 `CharData` 时，如果栈中的元素名按顺序包含命令行参数中给定的名称，就输出对应的文本。

如下命令输出了在两层 `div` 元素下 `h2` 元素的内容。输入的内容是 XML 规范，这份规范本身也是一个 XML 文档：

```

$ go build gopl.io/ch1/fetch
$ ./fetch http://www.w3.org/TR/2006/REC-xml11-20060816 |
  ./xmlselect div div h2
html body div div h2: 1 Introduction
html body div div h2: 2 Documents
html body div div h2: 3 Logical Structures
html body div div h2: 4 Physical Structures
html body div div h2: 5 Conformance
html body div div h2: 6 Notation
html body div div h2: A References
html body div div h2: B Definitions for Character Normalization
...

```

练习 7.17：扩展 `xmlselect`，让我们不仅可以用名字，还可以用 CSS 风格的属性来做选择。比如一个 `<div id="page" class="wide">` 元素，不仅可以通过名字，还可以通过 `id` 和 `class` 来做匹配

练习 7.18：使用基于标记的解析 API，写一个程序来读入一个任意的 XML 文档，构造出一棵树来展现 XML 中的主要节点。节点包括两种类型：`CharData` 节点表示文本字符串，`Element` 节点表示元素及其属性。每个元素节点包含它的子节点数组。

可以参考如下类型定义：

```

import "encoding/xml"

type Node interface{} // CharData 或 *Element

type CharData string

type Element struct {
    Type      xml.Name
    Attr      []xml.Attr
    Children  []Node
}

```

7.15 一些建议

当设计一个新包时，一个新手 Go 程序员会首先创建一系列接口，然后再定义满足这些接口的具体类型。这种方式会产生很多接口，但这些接口只有一个单独的实现。不要这样做。这种接口是不必要的抽象，还有运行时的成本。可以用导出机制（参考 6.6 节）来限制一个类型的哪些方法或结构体的哪些字段是对包外可见的。仅在有两个或者多个具体类型需

要按统一的方式处理时才需要接口。

这个规则也有特例，如果接口和类型实现出于依赖的原因不能放在同一个包里边，那么一个接口只有一个具体类型实现也是可以的。在这种情况下，接口是一种解耦两个包的好方式。

因为接口仅在有两个或者多个类型满足的情况下存在，所以它就必然会抽象掉那些特有的实现细节。这种设计的结果就是出现了具有更简单和更少方法的接口，比如 `io.Writer` 和 `fmt.Stringer` 都只有一个方法。设计新类型时越小的接口越容易满足。一个不错的接口设计经验是仅要求你需要的。

本章关于方法和接口的讲解就结束了。Go 语言能很好地支持面向对象编程风格，但这并不意味着你只能使用它。不是所有东西都必须是一个对象，全局函数应该有它们的位置，不完全封装的数据类型也应该有位置。综合来看，在本书第 1 章～第 5 章的示例中，我们用到的方法（比如 `input.Scan`）不超过两打，这与诸如 `fmt.Printf` 之类的普通函数比起来并不多。

goroutine 和通道

并发编程表现为程序由若干个自主的活动单元组成，它从来没有像今天这样重要。Web 服务器可以一次处理数千个请求。平板电脑和手机应用在渲染用户界面的同时，后端还同步进行着计算和处理网络请求。甚至传统的批处理任务——读取数据、计算、将结果输出——也使用并发来隐藏 I/O 操作的延迟，充分利用现代的多核计算机，内核的个数每年变多，但是速度没什么变化。

Go 有两种并发编程的风格。这一章展示 goroutine 和通道（channel），它们支持通信顺序进程（Communicating Sequential Process, CSP），CSP 是一个并发的模式，在不同的执行体（goroutine）之间传递值，但是变量本身局限于单一的执行体。第 9 章涵盖一些共享内存多线程的传统模型，它们和在其他主流语言中使用线程类似。第 9 章也会指出一些关于并发编程的重要难题和陷阱，这里暂不深入介绍。

即使 Go 对并发的支持是其很大的长处，并发编程在本质上也比顺序编程要困难一些，从顺序编程获取的直觉让我们加倍地迷茫。如果这是你第一次遇到并发，建议多花一点时间思考这两章的例子。

8.1 goroutine

在 Go 里，每一个并发执行的活动称为 goroutine。考虑一个程序，它有两个函数，一个做一些计算工作，另一个将结果输出，假设它们不相互调用。顺序程序可能调用一个函数，然后调用另一个，但是在有两个或多个 goroutine 的并发程序中，两个函数可以同时执行。很快我们将看到这样的程序。

如果你使用过操作系统或者其他语言中的线程，可以假设 goroutine 类似于线程，然后写出正确的程序。goroutine 和线程之间在数量上有非常大的差别，这将在 9.8 节进行讨论。

当一个程序启动时，只有一个 goroutine 来调用 `main` 函数，称它为主 goroutine。新的 goroutine 通过 `go` 语句进行创建。语法上，一个 `go` 语句是在普通的函数或者方法调用前加上 `go` 关键字前缀。`go` 语句使函数在一个新创建的 goroutine 中调用。`go` 语句本身的执行立即完成：

```
f() // 调用 f(); 等待它返回
go f() // 新建一个调用 f() 的 goroutine, 不用等待
```

下面的例子中，主 goroutine 计算第 45 个斐波那契数。因为它使用非常低效的递归算法，所以它需要大量的时间来执行，在此期间我们提供一个可见的提示，显示一个字符串“spinner”来指示程序依然在运行。

```
gopl.io/ch8/spinner
func main() {
    go spinner(100 * time.Millisecond)
    const n = 45
    fibN := fib(n) // slow
    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}
```

```

func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/-` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}

```

若干秒后，fib(45) 返回，main 函数输出结果：

```
Fibonacci(45) = 1134903170
```

然后 main 函数返回，当它发生时，所有的 goroutine 都暴力地直接终结，然后程序退出。除了从 main 返回或者退出程序之外，没有程序化的方法让一个 goroutine 来停止另一个，但是像我们将要看到的那样，有办法和 goroutine 通信来要求它自己停止。

注意程序如何由两个自主的活动（指示器和斐波那契数计算）来表达。它们写成独立的函数，但是同时在运行。

8.2 示例：并发时钟服务器

网络是一个自然使用并发的领域，因为服务器通常一次处理很多来自客户端的连接，每一个客户端通常和其他客户端保持独立。本节介绍 net 包，它提供构建客户端和服务器程序的组件，这些程序通过 TCP、UDP 或者 UNIX 套接字进行通信。第 1 章使用过的 net/http 包是在 net 包基础上构建的。

第一个例子是顺序时钟服务器，它以每秒钟一次的频率向客户端发送当前时间：

```

gopl.io/ch8/clock1
// clock1 是一个定期报告时间的 TCP 服务器
package main

import (
    "io"
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // 例如，连接中止
            continue
        }
        handleConn(conn) // 一次处理一个连接
    }
}

```

```

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return // 例如, 连接断开
        }
        time.Sleep(1 * time.Second)
    }
}

```

`Listen` 函数创建一个 `net.Listener` 对象，它在一个网络端口上监听进来的连接，这里是 TCP 端口 `localhost:8000`。监听器的 `Accept` 方法被阻塞，直到有连接请求进来，然后返回 `net.Conn` 对象来代表一个连接。

`handleConn` 函数处理一个完整的客户连接。在循环里，它将 `time.Now()` 获取的当前时间发送给客户端。因为 `net.Conn` 满足 `io.Writer` 接口，所以可以直接向它进行写入。当写入失败时循环结束，很多时候是客户端断开连接，这时 `handleConn` 函数使用延迟的 `close` 调用关闭自己这边的连接，然后继续等待下一个连接请求。

`time.Time.Format` 方法提供了格式化日期和时间信息的方式。它的参数是一个模板，指示如何格式化一个参考时间，具体如 `Mon Jan 2 03:04:05PM 2006 UTC-0700` 这样的形式。参考时间有 8 个部分（本周第几天、月、本月第几天，等等）。它们可以以任意的组合和对应数目的格式化字符出现在格式化模板中，所选择的日期和时间将通过所选择的格式进行显示。这里只使用时间的小时、分钟和秒部分。`time` 包定义了许多标准时间格式的模板，如 `time.RFC1123`。相反，当解析一个代表时间的字符串的时候使用相同的机制。

为了连接到服务器，需要一个像 `nc`（“netcat”）这样的程序，以及一个用来操作网络连接的标准工具：

```

$ go build gopl.io/ch8/clock1
$ ./clock1 &
$ nc localhost 8000
13:58:54
13:58:55
13:58:56
13:58:57
^C

```

客户端显示每秒从服务器发送的时间，直到使用 Control+C 快捷键中断它，UNIX 系统 shell 上面回显为 `^C`。如果系统上没有安装 `nc` 或 `netcat`，可以使用 `telnet` 或者一个使用 `net.Dial` 实现的 Go 版的 `netcat` 来连接 TCP 服务器：

```

gopl.io/ch8/netcat1
// netcat1 是一个只读的 TCP 客户端程序
package main

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

```

```

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    mustCopy(os.Stdout, conn)
}

func mustCopy(dst io.Writer, src io.Reader) {
    if _, err := io.Copy(dst, src); err != nil {
        log.Fatal(err)
    }
}

```

这个程序从网络连接中读取，然后写到标准输出，直到到达 EOF 或者出错。`mustCopy` 函数是这一节的多个例子中使用的一个实用程序。在不同的终端上同时运行两个客户端，一个显示在左边，一个在右边：

```

$ go build gopl.io/ch8/netcat1
$ ./netcat1
13:58:54
13:58:55
13:58:56
^C
$ ./netcat1
13:58:57
13:58:58
13:58:59
^C
$ killall clock1

```

`killall` 命令是 UNIX 的一个实用程序，用来终止所有指定名字的进程。

第二个客户端必须等到第一个结束才能正常工作，因为服务器是顺序的，一次只能处理一个客户请求。让服务器支持并发只需要一个很小的改变：在调用 `handleConn` 的地方添加一个 `go` 关键字，使它在自己的 goroutine 内执行。

[gopl.io/ch8/clock2](#)

```

for {
    conn, err := listener.Accept()
    if err != nil {
        log.Print(err) // 例如，连接中止
        continue
    }
    go handleConn(conn) // 并发处理连接
}

```

现在，多个客户端可以同时接收到时间：

```

$ go build gopl.io/ch8/clock2
$ ./clock2 &
$ go build gopl.io/ch8/netcat1
$ ./netcat1
14:02:54
14:02:55
14:02:56
14:02:57
14:02:58
14:02:59
14:03:00
14:03:01
$ ./netcat1
14:02:55
14:02:56
^C
$ ./netcat1
14:03:00
14:03:01

```

```
^C                                14:03:02
                                  ^C
$ killall clock2
```

练习 8.1：修改 `clock2` 来接收一个端口号，写一个程序 `clockwall`，作为多个时钟服务器的客户端，读取每一个服务器的时间，类似于不同地区办公室的时钟，然后显示在一个表中。如果可以访问不同地域的计算机，可以远程运行示例程序；否则可以伪装不同的时区，在不同的端口上本地运行：

```
$ TZ=US/Eastern ./clock2 -port 8010 &
$ TZ=Asia/Tokyo   ./clock2 -port 8030 &
$ TZ=Europe/London ./clock2 -port 8020 &
$ clockwall NewYork=localhost:8010 London=localhost:8020 Tokyo=localhost:8030
```

练习 8.2：实现一个并发的 FTP 服务器。服务器可以解释从客户端发来的命令，例如 `cd` 用来改变目录，`ls` 用来列出目录，`get` 用来发送一个文件的内容，`close` 用来关闭连接。可以使用标准的 `ftp` 命令作为客户端，或者自己写一个。

8.3 示例：并发回声服务器

时钟服务器每个连接使用一个 `goroutine`。在这一节，我们要构建一个回声服务器，每个连接使用多个 `goroutine` 来处理。大多数的回声服务器仅仅将读到的内容写回去，它可以使用下面简单的 `handleConn` 版本完成：

```
func handleConn(c net.Conn) {
    io.Copy(c, c) // 注意：忽略错误
    c.Close()
}
```

更有趣的回声服务器可以模仿真实的回声，第一次大的回声（"HELLO!"），在一定延迟后中等音量的回声（"Hello!"），然后安静的回声（"hello!"），最后什么都没有了，如下面这个版本的 `handleConn` 所示：

gopl.io/ch8/reverb1

```
func echo(c net.Conn, shout string, delay time.Duration) {
    fmt.Fprintln(c, "\t", strings.ToUpper(shout))
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", shout)
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        echo(c, input.Text(), 1*time.Second)
    }
    // 注意：忽略 input.Err() 中可能的错误
    c.Close()
}
```

我们需要升级客户端程序，使它可以在终端上向服务器输入，还可以将服务器的回复复制到输出，这里提供了另一个使用并发的机会：

```
gopl.io/ch8/netcat2
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    go mustCopy(os.Stdout, conn)
    mustCopy(conn, os.Stdin)
}
```

当主 goroutine 从标准输入读取并发送到服务器的时候，第二个 goroutine 读取服务器的回复并且输出。当主 goroutine 的输入结束时，例如用户在终端按入 Control+D (^D) 组合键（或者在微软 Windows 平台上按 Control+Z 组合键）时，这个程序停止，即使其他的 goroutine 还在运行。（8.4.1 节展示如何通过引入通道来等待两边一起结束。）

下面这段场景中，客户端的输入左对齐，服务器的回复是缩进的。客户端向回声服务器呼叫三次：

```
$ go build gopl.io/ch8/reverb1
$ ./reverb1 &
$ go build gopl.io/ch8/netcat2
$ ./netcat2
Hello?
    HELLO?
    Hello?
    hello?
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    is there anybody there?
    YOOO-HOOO!
    Yooo-hooo!
    yooo-hooo!
^D
$ killall reverb1
```

注意，第三次从客户端进行的呼叫直到第二次回声枯竭才进行处理，这个不是非常切合现实。真实的回声会由三个独立的呼喊回声叠加组成。为了模仿它，我们需要更多的 goroutine。再一次，在调用 echo 时加入 go 关键字：

```
gopl.io/ch8/reverb2
func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        go echo(c, input.Text(), 1*time.Second)
    }
    // 注意：忽略 input.Err() 中可能的错误
    c.Close()
}
```

当 go 语句执行的时候，计算 echo 函数所对应的参数；所以 input.Text() 是在主 goroutine 中推演。

现在的回声是并发的，在时间上面相互重合：

```
$ go build gopl.io/ch8/reverb2
$ ./reverb2 &
$ ./netcat2
Is there anybody there?
IS THERE ANYBODY THERE?

Yooo-hooo!
Is there anybody there?
Y000-H000!
is there anybody there?
Yooo-hooo!
yooo-hooo!

^D
$ killall reverb2
```

这就是使服务器变成并发所要做的，不仅处理来自多个客户端的链接，还包括在一个连接处理中，使用多个 `go` 关键字。

然而，在添加这些 `go` 关键字的同时，必须要仔细考虑方法 `net.Conn` 的并发调用是不是安全的，对大多数类型来讲，这都是不安全的。下一章讨论并发的安全性问题。

8.4 通道

如果说 goroutine 是 Go 程序并发的执行体，通道就是它们之间的连接。通道是可以让一个 goroutine 发送特定值到另一个 goroutine 的通信机制。每一个通道是一个具体类型的导管，叫作通道的元素类型。一个有 `int` 类型元素的通道写为 `chan int`。

使用内置的 `make` 函数来创建一个通道：

```
ch := make(chan int) // ch 的类型是'chan int'
```

像 `map` 一样，通道是一个使用 `make` 创建的数据结构的引用。当复制或者作为参数传递到一个函数时，复制的是引用，这样调用者和被调用者都引用同一份数据结构。和其他引用类型一样，通道的零值是 `nil`。

同种类型的通道可以使用 `==` 符号进行比较。当二者都是同一通道数据的引用时，比较值为 `true`。通道也可以和 `nil` 进行比较。

通道有两个主要操作：发送（`send`）和接收（`receive`），两者统称为通信。`send` 语句从一个 goroutine 传输一个值到另一个在执行接收表达式的 goroutine。两个操作都使用 `<-` 操作符书写。发送语句中，通道和值分别在 `<-` 的左右两边。在接收表达式中，`<-` 放在通道操作数前面。在接收表达式中，其结果未被使用也是合法的。

```
ch <- x // 发送语句
x = <-ch // 赋值语句中的接收表达式
<-ch // 接收语句，丢弃结果
```

通道支持第三个操作：关闭（`close`），它设置一个标志位来指示值当前已经发送完毕，这个通道后面没有值了；关闭后的发送操作将导致宕机。在一个已经关闭的通道上进行接收操作，将获取所有已经发送的值，直到通道为空；这时任何接收操作会立即完成，同时获取到一个通道元素类型对应的零值。

调用内置的 `close` 函数来关闭通道：

```
close(ch)
```

使用简单的 `make` 调用创建的通道叫无缓冲（`unbuffered`）通道，但 `make` 还可以接受第二

个可选参数，一个表示通道容量的整数。如果容量是 0，`make` 创建一个无缓冲通道：

```
ch = make(chan int)    // 无缓冲通道
ch = make(chan int, 0) // 无缓冲通道
ch = make(chan int, 3) // 容量为3的缓冲通道
```

首先介绍无缓冲通道，缓冲通道将在 8.4.4 节讨论。

8.4.1 无缓冲通道

无缓冲通道上的发送操作将会阻塞，直到另一个 goroutine 在对应的通道上执行接收操作，这时值传送完成，两个 goroutine 都可以继续执行。相反，如果接收操作先执行，接收方 goroutine 将阻塞，直到另一个 goroutine 在同一个通道上发送一个值。

使用无缓冲通道进行的通信导致发送和接收 goroutine 同步化。因此，无缓冲通道也称为同步通道。当一个值在无缓冲通道上传递时，接收值后发送方 goroutine 才被再次唤醒。

在讨论并发的时候，当我们说 x 早于 y 发生时，不仅仅是说 x 发生的时间早于 y ，而是说保证它是这样，并且是可预期的，比如更新变量，我们可以依赖这个机制。

当 x 既不比 y 早也不比 y 晚时，我们说 x 和 y 并发。这不意味着， x 和 y 一定同时发生，只说明我们不能假设它们的顺序。下一章中我们将看到，在两个 goroutine 并发地访问同一个变量的时候，有必要对这样的事件进行排序，避免程序的执行发生问题。

8.3 节中的客户端程序在主 goroutine 中将输入复制到服务器中，这样客户端在输入接收后立即退出，即使后台的 goroutine 还在继续。为了让程序等待后台的 goroutine 在完成后再退出，使用一个通道来同步两个 goroutine：

```
gopl.io/ch8/netcat3
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    done := make(chan struct{})
    go func() {
        io.Copy(os.Stdout, conn) // 注意：忽略错误
        log.Println("done")
        done <- struct{}{} // 指示主 goroutine
    }()
    mustCopy(conn, os.Stdin)
    conn.Close()
    <-done // 等待后台 goroutine 完成
}
```

当用户关闭标准输入流时，`mustCopy` 返回，主 goroutine 调用 `conn.Close()` 来关闭两端网络连接。关闭写半边的连接会导致服务器看到 EOF。关闭读半边的连接导致后台 goroutine 调用 `io.Copy` 返回“read from closed connection”错误，这也是我们去掉错误日志的原因；练习 8.3 给出了更好的解决方案。（注意，`go` 语句调用一个字面量函数，一个通用的构造方式）。

在它返回前，后台 goroutine 记录一条消息，然后发送一个值到 `done` 通道。主 goroutine 在退出前一直等待，直到它接收到这个值。最终效果是程序总是在退出前记录“done”消息。

通过通道发送消息有两个重要的方面需要考虑。每一条消息有一个值，但有时候通信本身以及通信发生的时间也很重要。当我们强调这方面的时候，把消息叫作事件（event）。当事件没有携带额外的信息时，它单纯的目的就是进行同步。我们通过使用一个 `struct{}` 元素类型的通道来强调它，尽管通常使用 `bool` 或 `int` 类型的通道来做相同的事情，因为 `done <-1` 比