了。这个特性和切片类似，保证可以用很小的成本来复制映射。

4.4 小结

- 数组是构造切片和映射的基石。
- Go 语言里切片经常用来处理数据的集合，映射用来处理具有键值对结构的数据。
- 内置函数 `make` 可以创建切片和映射，并指定原始的长度和容量。也可以直接使用切片和映射字面量，或者使用字面量作为变量的初始值。
- 切片有容量限制，不过可以使用内置的 `append` 函数扩展容量。
- 映射的增长没有容量或者任何限制。
- 内置函数 `len` 可以用来获取切片或者映射的长度。
- 内置函数 `cap` 只能用于切片。
- 通过组合，可以创建多维数组和多维切片。也可以使用切片或者其他映射作为映射的值。但是切片不能用作映射的键。
- 将切片或者映射传递给函数成本很小，并且不会复制底层的数据结构。

第 5 章　Go 语言的类型系统

本章主要内容

- 声明新的用户定义的类型
- 使用方法，为类型增加新的行为
- 了解何时使用指针，何时使用值
- 通过接口实现多态
- 通过组合来扩展或改变类型
- 公开或者未公开的标识符

Go 语言是一种静态类型的编程语言。这意味着，编译器需要在编译时知晓程序里每个值的类型。如果提前知道类型信息，编译器就可以确保程序合理地使用值。这有助于减少潜在的内存异常和 bug，并且使编译器有机会对代码进行一些性能优化，提高执行效率。

值的类型给编译器提供两部分信息：第一部分，需要分配多少内存给这个值（即值的规模）；第二部分，这段内存表示什么。对于许多内置类型的情况来说，规模和表示是类型名的一部分。`int64` 类型的值需要 8 字节（64 位），表示一个整数值；`float32` 类型的值需要 4 字节（32 位），表示一个 IEEE-754 定义的二进制浮点数；`bool` 类型的值需要 1 字节（8 位），表示布尔值 `true` 和 `false`。

有些类型的内部表示与编译代码的机器的体系结构有关。例如，根据编译所在的机器的体系结构，一个 `int` 值的大小可能是 8 字节（64 位），也可能是 4 字节（32 位）。还有一些与体系结构相关的类型，如 Go 语言里的所有引用类型。好在创建和使用这些类型的值的时候，不需要了解这些与体系结构相关的信息。但是，如果编译器不知道这些信息，就无法阻止用户做一些导致程序受损甚至机器故障的事情。

5.1　用户定义的类型

Go 语言允许用户定义类型。当用户声明一个新类型时，这个声明就给编译器提供了一个框

架，告知必要的内存大小和表示信息。声明后的类型与内置类型的运作方式类似。Go 语言里声明用户定义的类型有两种方法。最常用的方法是使用关键字 `struct`，它可以让用户创建一个结构类型。

结构类型通过组合一系列固定且唯一的字段来声明，如代码清单 5-1 所示。结构里每个字段都会用一个已知类型声明。这个已知类型可以是内置类型，也可以是其他用户定义的类型。

代码清单 5-1　声明一个结构类型

```
01 // user 在程序里定义一个用户类型
02 type user struct {
03     name       string
04     email      string
05     ext        int
06     privileged bool
07 }
```

在代码清单 5-1 中，可以看到一个结构类型的声明。这个声明以关键字 `type` 开始，之后是新类型的名字，最后是关键字 `struct`。这个结构类型有 4 个字段，每个字段都基于一个内置类型。读者可以看到这些字段是如何组合成一个数据的结构的。一旦声明了类型（如代码清单 5-2 所示），就可以使用这个类型创建值。

代码清单 5-2　使用结构类型声明变量，并初始化为其零值

```
09 // 声明 user 类型的变量
10 var bill user
```

在代码清单 5-2 的第 10 行，关键字 `var` 创建了类型为 `user` 且名为 `bill` 的变量。当声明变量时，这个变量对应的值总是会被初始化。这个值要么用指定的值初始化，要么用零值（即变量类型的默认值）做初始化。对数值类型来说，零值是 `0`；对字符串来说，零值是空字符串；对布尔类型，零值是 `false`。对这个例子里的结构，结构里每个字段都会用零值初始化。

任何时候，创建一个变量并初始化为其零值，习惯是使用关键字 `var`。这种用法是为了更明确地表示一个变量被设置为零值。如果变量被初始化为某个非零值，就配合结构字面量和短变量声明操作符来创建变量。

代码清单 5-3 展示了如何声明一个 `user` 类型的变量，并使用某个非零值作为初始值。在第 13 行，我们首先给出了一个变量名，之后是短变量声明操作符。这个操作符是冒号加一个等号（`:=`）。一个短变量声明操作符在一次操作中完成两件事情：声明一个变量，并初始化。短变量声明操作符会使用右侧给出的类型信息作为声明变量的类型。

代码清单 5-3　使用结构字面量来声明一个结构类型的变量

```
12 // 声明 user 类型的变量，并初始化所有字段
13 lisa := user{
14     name:       "Lisa",
15     email:      "lisa@email.com",
```

```
16     ext:        123,
17     privileged: true,
18 }
```

既然要创建并初始化一个结构类型，我们就使用结构字面量来完成这个初始化，如代码清单 5-4 所示。结构字面量使用一对大括号括住内部字段的初始值。

代码清单 5-4　使用结构字面量创建结构类型的值

```
13 user{
14     name:       "Lisa",
15     email:      "lisa@email.com",
16     ext:        123,
17     privileged: true,
18 }
```

结构字面量可以对结构类型采用两种形式。代码清单 5-4 中使用了第一种形式，这种形式在不同行声明每个字段的名字以及对应的值。字段名与值用冒号分隔，每一行以逗号结尾。这种形式对字段的声明顺序没有要求。第二种形式没有字段名，只声明对应的值，如代码清单 5-5 所示。

代码清单 5-5　不使用字段名，创建结构类型的值

```
12 // 声明 user 类型的变量
13 lisa := user{"Lisa", "lisa@email.com", 123, true}
```

每个值也可以分别占一行，不过习惯上这种形式会写在一行里，结尾不需要逗号。这种形式下，值的顺序很重要，必须要和结构声明中字段的顺序一致。当声明结构类型时，字段的类型并不限制在内置类型，也可以使用其他用户定义的类型，如代码清单 5-6 所示。

代码清单 5-6　使用其他结构类型声明字段

```
20 // admin 需要一个 user 类型作为管理者，并附加权限
21 type admin struct {
22     person user
23     level  string
24 }
```

代码清单 5-6 展示了一个名为 `admin` 的新结构类型。这个结构类型有一个名为 `person` 的 `user` 类型的字段，还声明了一个名为 `level` 的 `string` 字段。当创建具有 `person` 这种字段的结构类型的变量时，初始化用的结构字面量会有一些变化，如代码清单 5-7 所示。

代码清单 5-7　使用结构字面量来创建字段的值

```
26 // 声明 admin 类型的变量
27 fred := admin{
28     person: user{
29         name:       "Lisa",
30         email:      "lisa@email.com",
31         ext:        123,
32         privileged: true,
```

```
33     },
34     level: "super",
35 }
```

为了初始化 person 字段，我们需要创建一个 user 类型的值。代码清单 5-7 的第 28 行就是在创建这个值。这行代码使用结构字面量的形式创建了一个 user 类型的值，并赋给了 person 字段。

另一种声明用户定义的类型的方法是，基于一个已有的类型，将其作为新类型的类型说明。当需要一个可以用已有类型表示的新类型的时候，这种方法会非常好用，如代码清单 5-8 所示。标准库使用这种声明类型的方法，从内置类型创建出很多更加明确的类型，并赋予更高级的功能。

代码清单 5-8　基于 int64 声明一个新类型

```
type Duration int64
```

代码清单 5-8 展示的是标准库的 time 包里的一个类型的声明。Duration 是一种描述时间间隔的类型，单位是纳秒（ns）。这个类型使用内置的 int64 类型作为其表示。在 Duration 类型的声明中，我们把 int64 类型叫作 Duration 的基础类型。不过，虽然 int64 是基础类型，Go 并不认为 Duration 和 int64 是同一种类型。这两个类型是完全不同的有区别的类型。

为了更好地展示这种区别，来看一下代码清单 5-9 所示的小程序。这个程序本身无法通过编译。

代码清单 5-9　给不同类型的变量赋值会产生编译错误

```
01 package main
02
03 type Duration int64
04
05 func main() {
06     var dur Duration
07     dur = int64(1000)
08 }
```

代码清单 5-9 所示的程序在第 03 行声明了 Duration 类型。之后在第 06 行声明了一个类型为 Duration 的变量 dur，并使用零值作为初值。之后，第 7 行的代码会在编译的时候产生编译错误，如代码清单 5-10 所示。

代码清单 5-10　实际产生的编译错误

```
prog.go:7: cannot use int64(1000) (type int64) as type Duration
           in assignment
```

编译器很清楚这个程序的问题：类型 int64 的值不能作为类型 Duration 的值来用。换句话说，虽然 int64 类型是基础类型，Duration 类型依然是一个独立的类型。两种不同类型的值即便互相兼容，也不能互相赋值。编译器不会对不同类型的值做隐式转换。

5.2 方法

方法能给用户定义的类型添加新的行为。方法实际上也是函数，只是在声明时，在关键字 func 和方法名之间增加了一个参数，如代码清单 5-11 所示。

代码清单 5-11　listing11.go

```
01 // 这个示例程序展示如何声明
02 // 并使用方法
03 package main
04
05 import (
06     "fmt"
07 )
08
09 // user 在程序里定义一个用户类型
10 type user struct {
11     name  string
12     email string
13 }
14
15 // notify 使用值接收者实现了一个方法
16 func (u user) notify() {
17     fmt.Printf("Sending User Email To %s<%s>\n",
18         u.name,
19         u.email)
20 }
21
22 // changeEmail 使用指针接收者实现了一个方法
23 func (u *user) changeEmail(email string) {
24     u.email = email
25 }
26
27 // main 是应用程序的入口
28 func main() {
29     // user 类型的值可以用来调用
30     // 使用值接收者声明的方法
31     bill := user{"Bill", "bill@email.com"}
32     bill.notify()
33
34     // 指向 user 类型值的指针也可以用来调用
35     // 使用值接收者声明的方法
36     lisa := &user{"Lisa", "lisa@email.com"}
37     lisa.notify()
38
39     // user 类型的值可以用来调用
40     // 使用指针接收者声明的方法
41     bill.changeEmail("bill@newdomain.com")
42     bill.notify()
43
```

```
44      // 指向 user 类型值的指针可以用来调用
45      // 使用指针接收者声明的方法
46      lisa.changeEmail("lisa@newdomain.com")
47      lisa.notify()
48  }
```

代码清单 5-11 的第 16 行和第 23 行展示了两种类型的方法。关键字 `func` 和函数名之间的参数被称作*接收者*，将函数与接收者的类型绑在一起。如果一个函数有接收者，这个函数就被称为*方法*。当运行这段程序时，会得到代码清单 5-12 所示的输出。

代码清单 5-12　listing11.go 的输出

```
Sending User Email To Bill<bill@email.com>
Sending User Email To Lisa<lisa@email.com>
Sending User Email To Bill<bill@newdomain.com>
Sending User Email To Lisa<lisa@comcast.com>
```

让我们来解释一下代码清单 5-13 所示的程序都做了什么。在第 10 行，该程序声明了名为 `user` 的结构类型，并声明了名为 `notify` 的方法。

代码清单 5-13　listing11.go：第 09 行到第 20 行

```
09 // user 在程序里定义一个用户类型
10 type user struct {
11      name  string
12      email string
13 }
14
15 // notify 使用值接收者实现了一个方法
16 func (u user) notify() {
17      fmt.Printf("Sending User Email To %s<%s>\n",
18          u.name,
19          u.email)
20 }
```

Go 语言里有两种类型的接收者：*值接收者*和*指针接收者*。在代码清单 5-13 的第 16 行，使用值接收者声明了 `notify` 方法，如代码清单 5-14 所示。

代码清单 5-14　使用值接收者声明一个方法

```
func (u user) notify() {
```

`notify` 方法的接收者被声明为 `user` 类型的值。如果使用值接收者声明方法，调用时会使用这个值的一个副本来执行。让我们跳到代码清单 5-11 的第 32 行来看一下如何调用 `notify` 方法，如代码清单 5-15 所示。

代码清单 5-15　listing11.go：第 29 行到第 32 行

```
29      // user 类型的值可以用来调用
30      // 使用值接收者声明的方法
```

```
31      bill := user{"Bill", "bill@email.com"}
32      bill.notify()
```

代码清单 5-15 展示了如何使用 user 类型的值来调用方法。第 31 行声明了一个 user 类型的变量 bill，并使用给定的名字和电子邮件地址做初始化。之后在第 32 行，使用变量 bill 来调用 notify 方法，如代码清单 5-16 所示。

代码清单 5-16　使用变量来调用方法

```
bill.notify()
```

这个语法与调用一个包里的函数看起来很类似。但在这个例子里，bill 不是包名，而是变量名。这段程序在调用 notify 方法时，使用 bill 的值作为接收者进行调用，方法 notify 会接收到 bill 的值的一个副本。

也可以使用指针来调用使用值接收者声明的方法，如代码清单 5-17 所示。

代码清单 5-17　listing11.go：第 34 行到第 37 行

```
34      // 指向 user 类型值的指针也可以用来调用
35      // 使用值接收者声明的方法
36      lisa := &user{"Lisa", "lisa@email.com"}
37      lisa.notify()
```

代码清单 5-17 展示了如何使用指向 user 类型值的指针来调用 notify 方法。在第 36 行，声明了一个名为 lisa 的指针变量，并使用给定的名字和电子邮件地址做初始化。之后在第 37 行，使用这个指针变量来调用 notify 方法。为了支持这种方法调用，Go 语言调整了指针的值，来符合方法接收者的定义。可以认为 Go 语言执行了代码清单 5-18 所示的操作。

代码清单 5-18　Go 在代码背后的执行动作

```
(*lisa).notify()
```

代码清单 5-18 展示了 Go 编译器为了支持这种方法调用背后做的事情。指针被解引用为值，这样就符合了值接收者的要求。再强调一次，notify 操作的是一个副本，只不过这次操作的是从 lisa 指针指向的值的副本。

也可以使用指针接收者声明方法，如代码清单 5-19 所示。

代码清单 5-19　listing11.go：第 22 行到第 25 行

```
22 // changeEmail 使用指针接收者实现了一个方法
23 func (u *user) changeEmail(email string) {
24      u.email = email
25 }
```

代码清单 5-19 展示了 changeEmail 方法的声明。这个方法使用指针接收者声明。这个接收者的类型是指向 user 类型值的指针，而不是 user 类型的值。当调用使用指针接收者声明的方法时，这个方法会共享调用方法时接收者所指向的值，如代码清单 5-20 所示。

代码清单 5-20 listing11.go：第 36 行和第 44 行到第 46 行

```
36      lisa := &user{"Lisa", "lisa@email.com"}

44      // 指向 user 类型值的指针可以用来调用
45      // 使用指针接收者声明的方法
46      lisa.changeEmail("lisa@newdomain.com")
```

在代码清单 5-20 中，可以看到声明了 lisa 指针变量，还有第 46 行使用这个变量调用了 changeEmail 方法。一旦 changeEmail 调用返回，这个调用对值做的修改也会反映在 lisa 指针所指向的值上。这是因为 changeEmail 方法使用了指针接收者。总结一下，值接收者使用值的副本来调用方法，而指针接受者使用实际值来调用方法。

也可以使用一个值来调用使用指针接收者声明的方法，如代码清单 5-21 所示。

代码清单 5-21 listing11.go：第 31 行和第 39 行到第 41 行

```
31      bill := user{"Bill", "bill@email.com"}

39      // user 类型的值可以用来调用
40      // 使用指针接收者声明的方法
41      bill.changeEmail("bill@newdomain.com")
```

在代码清单 5-21 中可以看到声明的变量 bill，以及之后使用这个变量调用使用指针接收者声明的 changeEmail 方法。Go 语言再一次对值做了调整，使之符合函数的接收者，进行调用，如代码清单 5-22 所示。

代码清单 5-22 Go 在代码背后的执行动作

```
(&bill).changeEmail ("bill@newdomain.com")
```

代码清单 5-22 展示了 Go 编译器为了支持这种方法调用在背后做的事情。在这个例子里，首先引用 bill 值得到一个指针，这样这个指针就能够匹配方法的接收者类型，再进行调用。Go 语言既允许使用值，也允许使用指针来调用方法，不必严格符合接收者的类型。这个支持非常方便开发者编写程序。

应该使用值接收者，还是应该使用指针接收者，这个问题有时会比较迷惑人。可以遵从标准库里一些基本的指导方针来做决定。后面会进一步介绍这些指导方针。

5.3 类型的本质

在声明一个新类型之后，声明一个该类型的方法之前，需要先回答一个问题：这个类型的本质是什么。如果给这个类型增加或者删除某个值，是要创建一个新值，还是要更改当前的值？如果是要创建一个新值，该类型的方法就使用值接收者。如果是要修改当前值，就使用指针接收者。这个答案也会影响程序内部传递这个类型的值的方式：是按值做传递，还是按指针做传递。保持

传递的一致性很重要。这个背后的原则是，不要只关注某个方法是如何处理这个值，而是要关注这个值的本质是什么。

5.3.1 内置类型

内置类型是由语言提供的一组类型。我们已经见过这些类型，分别是数值类型、字符串类型和布尔类型。这些类型本质上是原始的类型。因此，当对这些值进行增加或者删除的时候，会创建一个新值。基于这个结论，当把这些类型的值传递给方法或者函数时，应该传递一个对应值的副本。让我们看一下标准库里使用这些内置类型的值的函数，如代码清单 5-23 所示。

代码清单 5-23　golang.org/src/strings/strings.go：第 620 行到第 625 行

```
620 func Trim(s string, cutset string) string {
621     if s == "" || cutset == "" {
622         return s
623     }
624     return TrimFunc(s, makeCutsetFunc(cutset))
625 }
```

在代码清单 5-23 中，可以看到标准库里 `strings` 包的 `Trim` 函数。`Trim` 函数传入一个 `string` 类型的值作操作，再传入一个 `string` 类型的值用于查找。之后函数会返回一个新的 `string` 值作为操作结果。这个函数对调用者原始的 `string` 值的一个副本做操作，并返回一个新的 string 值的副本。字符串（`string`）就像整数、浮点数和布尔值一样，本质上是一种很原始的数据值，所以在函数或方法内外传递时，要传递字符串的一份副本。

让我们看一下体现内置类型具有的原始本质的第二个例子，如代码清单 5-24 所示。

代码清单 5-24　golang.org/src/os/env.go：第 38 行到第 44 行

```
38 func isShellSpecialVar(c uint8) bool {
39     switch c {
40     case '*', '#', '$', '@', '!', '?', '0', '1', '2', '3', '4', '5',
                                                    '6', '7', '8', '9':
41         return true
42     }
43     return false
44 }
```

代码清单 5-24 展示了 `env` 包里的 `isShellSpecialVar` 函数。这个函数传入了一个 `int8` 类型的值，并返回一个 `bool` 类型的值。注意，这里的参数没有使用指针来共享参数的值或者返回值。调用者传入了一个 `uint8` 值的副本，并接受一个返回值 `true` 或者 `false`。

5.3.2 引用类型

Go 语言里的引用类型有如下几个：切片、映射、通道、接口和函数类型。当声明上述类型

的变量时，创建的变量被称作标头（header）值。从技术细节上说，字符串也是一种引用类型。每个引用类型创建的标头值是包含一个指向底层数据结构的指针。每个引用类型还包含一组独特的字段，用于管理底层数据结构。因为标头值是为复制而设计的，所以永远不需要共享一个引用类型的值。标头值里包含一个指针，因此通过复制来传递一个引用类型的值的副本，本质上就是在共享底层数据结构。

让我们看一下 net 包里的类型，如代码清单 5-25 所示。

代码清单 5-25　golang.org/src/net/ip.go：第 32 行

```
32 type IP []byte
```

代码清单 5-25 展示了一个名为 IP 的类型，这个类型被声明为字节切片。当要围绕相关的内置类型或者引用类型来声明用户定义的行为时，直接基于已有类型来声明用户定义的类型会很好用。编译器只允许为命名的用户定义的类型声明方法，如代码清单 5-26 所示。

代码清单 5-26　golang.org/src/net/ip.go：第 329 行到第 337 行

```
329 func (ip IP) MarshalText() ([]byte, error) {
330     if len(ip) == 0 {
331         return []byte(""), nil
332     }
333     if len(ip) != IPv4len && len(ip) != IPv6len {
334         return nil, errors.New("invalid IP address")
335     }
336     return []byte(ip.String()), nil
337 }
```

代码清单 5-26 里定义的 MarshalText 方法是用 IP 类型的值接收者声明的。一个值接收者，正像预期的那样通过复制来传递引用，从而不需要通过指针来共享引用类型的值。这种传递方法也可以应用到函数或者方法的参数传递，如代码清单 5-27 所示。

代码清单 5-27　golang.org/src/net/ip.go：第 318 行到第 325 行

```
318 // ipEmptyString 像 ip.String 一样，
319 // 只不过在没有设置 ip 时会返回一个空字符串
320 func ipEmptyString(ip IP) string {
321     if len(ip) == 0 {
322         return ""
323     }
324     return ip.String()
325 }
```

在代码清单 5-27 里，有一个 ipEmptyString 函数。这个函数需要传入一个 IP 类型的值。再一次，你可以看到调用者传入的是这个引用类型的值，而不是通过引用共享给这个函数。调用者将引用类型的值的副本传入这个函数。这种方法也适用于函数的返回值。最后要说的是，引用类型的值在其他方面像原始的数据类型的值一样对待。

5.3.3 结构类型

结构类型可以用来描述一组数据值，这组值的本质即可以是原始的，也可以是非原始的。如果决定在某些东西需要删除或者添加某个结构类型的值时该结构类型的值不应该被更改，那么需要遵守之前提到的内置类型和引用类型的规范。让我们从标准库里的一个原始本质的类型的结构实现开始，如代码清单 5-28 所示。

代码清单 5-28　golang.org/src/time/time.go：第 39 行到第 55 行

```
39 type Time struct {
40     // sec 给出自公元 1 年 1 月 1 日 00:00:00
41     // 开始的秒数
42     sec int64
43
44     // nsec 指定了一秒内的纳秒偏移，
45     // 这个值是非零值，
46     // 必须在[0, 999999999]范围内
47     nsec int32
48
49     // loc 指定了一个 Location，
50     // 用于决定该时间对应的当地的分、小时、
51     // 天和年的值
52     // 只有 Time 的零值，其 loc 的值是 nil
53     // 这种情况下，认为处于 UTC 时区
54     loc *Location
55 }
```

代码清单 5-28 中的 `Time` 结构选自 `time` 包。当思考时间的值时，你应该意识到给定的一个时间点的时间是不能修改的。所以标准库里也是这样实现 `Time` 类型的。让我们看一下 `Now` 函数是如何创建 `Time` 类型的值的，如代码清单 5-29 所示。

代码清单 5-29　golang.org/src/time/time.go：第 781 行到第 784 行

```
781 func Now() Time {
782     sec, nsec := now()
783     return Time{sec + unixToInternal, nsec, Local}
784 }
```

代码清单 5-29 中的代码展示了 `Now` 函数的实现。这个函数创建了一个 `Time` 类型的值，并给调用者返回了 `Time` 值的副本。这个函数没有使用指针来共享 `Time` 值。之后，让我们来看一个 `Time` 类型的方法，如代码清单 5-30 所示。

代码清单 5-30　golang.org/src/time/time.go：第 610 行到第 622 行

```
610 func (t Time) Add(d Duration) Time {
611     t.sec += int64(d / 1e9)
612     nsec := int32(t.nsec) + int32(d%1e9)
```

```
613     if nsec >= 1e9 {
614         t.sec++
615         nsec -= 1e9
616     } else if nsec < 0 {
617         t.sec--
618         nsec += 1e9
619     }
620     t.nsec = nsec
621     return t
622 }
```

代码清单 5-30 中的 Add 方法是展示标准库如何将 Time 类型作为本质是原始的类型的绝佳例子。这个方法使用值接收者，并返回了一个新的 Time 值。该方法操作的是调用者传入的 Time 值的副本，并且给调用者返回了一个方法内的 Time 值的副本。至于是使用返回的值替换原来的 Time 值，还是创建一个新的 Time 变量来保存结果，是由调用者决定的事情。

大多数情况下，结构类型的本质并不是原始的，而是非原始的。这种情况下，对这个类型的值做增加或者删除的操作应该更改值本身。当需要修改值本身时，在程序中其他地方，需要使用指针来共享这个值。让我们看一个由标准库中实现的具有非原始本质的结构类型的例子，如代码清单 5-31 所示。

代码清单 5-31　golang.org/src/os/file_unix.go：第 15 行到第 29 行

```
15 // File 表示一个打开的文件描述符
16 type File struct {
17     *file
18 }
19
20 // file 是*File 的实际表示
21 // 额外的一层结构保证没有哪个 os 的客户端
22 // 能够覆盖这些数据。如果覆盖这些数据，
23 // 可能在变量终结时关闭错误的文件描述符
24 type file struct {
25     fd int
26     name string
27     dirinfo *dirInfo // 除了目录结构，此字段为 nil
28     nepipe int32 // Write 操作时遇到连续 EPIPE 的次数
29 }
```

可以在代码清单 5-31 里看到标准库中声明的 File 类型。这个类型的本质是非原始的。这个类型的值实际上不能安全复制。对内部未公开的类型的注释，解释了不安全的原因。因为没有方法阻止程序员进行复制，所以 File 类型的实现使用了一个嵌入的指针，指向一个未公开的类型。本章后面会继续探讨内嵌类型。正是这层额外的内嵌类型阻止了复制。不是所有的结构类型都需要或者应该实现类似的额外保护。程序员需要能识别出每个类型的本质，并使用这个本质来决定如何组织类型。

让我们看一下 Open 函数的实现，如代码清单 5-32 所示。

代码清单 5-32　golang.org/src/os/file.go：第 238 行到第 240 行

```
238 func Open(name string) (file *File, err error) {
239     return OpenFile(name, O_RDONLY, 0)
240 }
```

代码清单 5-32 展示了 `Open` 函数的实现，调用者得到的是一个指向 `File` 类型值的指针。`Open` 创建了 `File` 类型的值，并返回指向这个值的指针。如果一个创建用的工厂函数返回了一个指针，就表示这个被返回的值的本质是非原始的。

即便函数或者方法没有直接改变非原始的值的状态，依旧应该使用共享的方式传递，如代码清单 5-33 所示。

代码清单 5-33　golang.org/src/os/file.go：第 224 行到第 232 行

```
224 func (f *File) Chdir() error {
225     if f == nil {
226         return ErrInvalid
227     }
228     if e := syscall.Fchdir(f.fd); e != nil {
229         return &PathError{"chdir", f.name, e}
230     }
231     return nil
232 }
```

代码清单 5-33 中的 `Chdir` 方法展示了，即使没有修改接收者的值，依然是用指针接收者来声明的。因为 `File` 类型的值具备非原始的本质，所以总是应该被共享，而不是被复制。

是使用值接收者还是指针接收者，不应该由该方法是否修改了接收到的值来决定。这个决策应该基于该类型的本质。这条规则的一个例外是，需要让类型值符合某个接口的时候，即便类型的本质是非原始本质的，也可以选择使用值接收者声明方法。这样做完全符合接口值调用方法的机制。5.4 节会讲解什么是接口值，以及使用接口值调用方法的机制。

5.4　接口

多态是指代码可以根据类型的具体实现采取不同行为的能力。如果一个类型实现了某个接口，所有使用这个接口的地方，都可以支持这种类型的值。标准库里有很好的例子，如 `io` 包里实现的流式处理接口。`io` 包提供了一组构造得非常好的接口和函数，来让代码轻松支持流式数据处理。只要实现两个接口，就能利用整个 `io` 包背后的所有强大能力。

不过，我们的程序在声明和实现接口时会涉及很多细节。即便实现的是已有接口，也需要了解这些接口是如何工作的。在探究接口如何工作以及实现的细节之前，我们先来看一下使用标准库里的接口的例子。

5.4.1 标准库

我们先来看一个示例程序，这个程序实现了流行程序 curl 的功能，如代码清单 5-34 所示。

代码清单 5-34 listing34.go

```
01 // 这个示例程序展示如何使用 io.Reader 和 io.Writer 接口
02 // 写一个简单版本的 curl 程序
03 package main
04
05 import (
06     "fmt"
07     "io"
08     "net/http"
09     "os"
10 )
11
12 // init 在 main 函数之前调用
13 func init() {
14     if len(os.Args) != 2 {
15         fmt.Println("Usage: ./example2 <url>")
16         os.Exit(-1)
17     }
18 }
19
20 // main 是应用程序的入口
21 func main() {
22     // 从 Web 服务器得到响应
23     r, err := http.Get(os.Args[1])
24     if err != nil {
25         fmt.Println(err)
26         return
27     }
28
29     // 从 Body 复制到 Stdout
30     io.Copy(os.Stdout, r.Body)
31     if err := r.Body.Close(); err != nil {
32         fmt.Println(err)
33     }
34 }
```

代码清单 5-34 展示了接口的能力以及在标准库里的应用。只用了几行代码我们就通过两个函数以及配套的接口，完成了 curl 程序。在第 23 行，调用了 `http` 包的 `Get` 函数。在与服务器成功通信后，`http.Get` 函数会返回一个 `http.Response` 类型的指针。`http.Response` 类型包含一个名为 `Body` 的字段，这个字段是一个 `io.ReadCloser` 接口类型的值。

在第 30 行，`Body` 字段作为第二个参数传给 `io.Copy` 函数。`io.Copy` 函数的第二个参数，接受一个 `io.Reader` 接口类型的值，这个值表示数据流入的源。`Body` 字段实现了 `io.Reader`

接口，因此我们可以将 Body 字段传入 io.Copy，使用 Web 服务器的返回内容作为源。

io.Copy 的第一个参数是复制到的目标，这个参数必须是一个实现了 io.Writer 接口的值。对于这个目标，我们传入了 os 包里的一个特殊值 Stdout。这个接口值表示标准输出设备，并且已经实现了 io.Writer 接口。当我们将 Body 和 Stdout 这两个值传给 io.Copy 函数后，这个函数会把服务器的数据分成小段，源源不断地传给终端窗口，直到最后一个片段读取并写入终端，io.Copy 函数才返回。

io.Copy 函数可以以这种工作流的方式处理很多标准库里已有的类型，如代码清单 5-35 所示。

代码清单 5-35　listing35.go

```
01 // 这个示例程序展示 bytes.Buffer 也可以
02 // 用于 io.Copy 函数
03 package main
04
05 import (
06     "bytes"
07     "fmt"
08     "io"
09     "os"
10 )
11
12 // main 是应用程序的入口
13 func main() {
14     var b bytes.Buffer
15
16     // 将字符串写入 Buffer
17     b.Write([]byte("Hello"))
18
19     // 使用 Fprintf 将字符串拼接到 Buffer
20     fmt.Fprintf(&b, "World!")
21
22     // 将 Buffer 的内容写到 Stdout
23     io.Copy(os.Stdout, &b)
24 }
```

代码清单 5-35 展示了一个程序，这个程序使用接口来拼接字符串，并将数据以流的方式输出到标准输出设备。在第 14 行，创建了一个 bytes 包里的 Buffer 类型的变量 b，用于缓冲数据。之后在第 17 行使用 Write 方法将字符串 Hello 写入这个缓冲区 b。第 20 行，调用 fmt 包里的 Fprintf 函数，将第二个字符串追加到缓冲区 b 里。

fmt.Fprintf 函数接受一个 io.Writer 类型的接口值作为其第一个参数。由于 bytes.Buffer 类型的指针实现了 io.Writer 接口，所以可以将缓存 b 传入 fmt.Fprintf 函数，并执行追加操作。最后，在第 23 行，再次使用 io.Copy 函数，将字符写到终端窗口。由于 bytes.Buffer 类型的指针也实现了 io.Reader 接口，io.Copy 函数可以用于在终端窗

口显示缓冲区 b 的内容。

希望这两个小程序展示出接口的好处，以及标准库内部是如何使用接口的。下一步，让我们看一下实现接口的细节。

5.4.2 实现

接口是用来定义行为的类型。这些被定义的行为不由接口直接实现，而是通过方法由用户定义的类型实现。如果用户定义的类型实现了某个接口类型声明的一组方法，那么这个用户定义的类型的值就可以赋给这个接口类型的值。这个赋值会把用户定义的类型的值存入接口类型的值。

对接口值方法的调用会执行接口值里存储的用户定义的类型的值对应的方法。因为任何用户定义的类型都可以实现任何接口，所以对接口值方法的调用自然就是一种多态。在这个关系里，用户定义的类型通常叫作*实体类型*，原因是如果离开内部存储的用户定义的类型的值的实现，接口值并没有具体的行为。

并不是所有值都完全等同，用户定义的类型的值或者指针要满足接口的实现，需要遵守一些规则。这些规则在 5.4.3 节介绍方法集时有详细说明。探寻方法集的细节之前，了解接口类型值大概的形式以及用户定义的类型的值是如何存入接口的，会有很多帮助。

图 5-1 展示了在 `user` 类型值赋值后接口变量的值的内部布局。接口值是一个两个字长度的数据结构，第一个字包含一个指向内部表的指针。这个内部表叫作 iTable，包含了所存储的值的类型信息。iTable 包含了已存储的值的类型信息以及与这个值相关联的一组方法。第二个字是一个指向所存储值的指针。将类型信息和指针组合在一起，就将这两个值组成了一种特殊的关系。

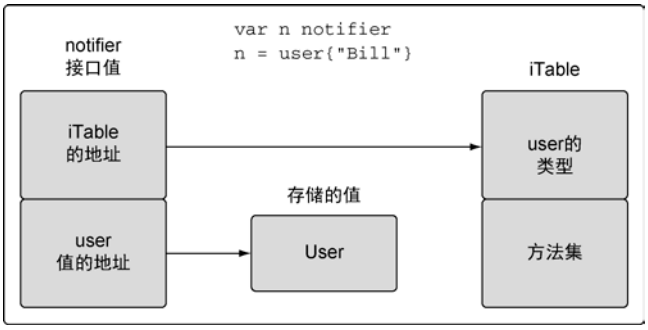

图 5-1　实体值赋值后接口值的简图

图 5-2 展示了一个指针赋值给接口之后发生的变化。在这种情况里，类型信息会存储一个指向保存的类型的指针，而接口值第二个字依旧保存指向实体值的指针。

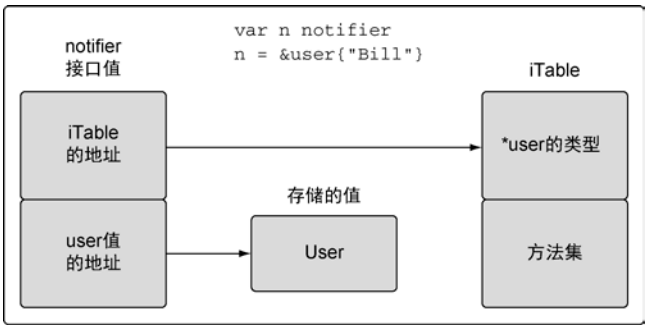

图 5-2　实体指针赋值后接口值的简图

5.4.3　方法集

方法集定义了接口的接受规则。看一下代码清单 5-36 所示的代码，有助于理解方法集在接口中的重要角色。

代码清单 5-36　listing36.go

```
01 // 这个示例程序展示 Go 语言里如何使用接口
02 package main
03
04 import (
05     "fmt"
06 )
07
08 // notifier 是一个定义了
09 // 通知类行为的接口
10 type notifier interface {
11     notify()
12 }
13
14 // user 在程序里定义一个用户类型
15 type user struct {
16     name  string
17     email string
18 }
19
20 // notify 是使用指针接收者实现的方法
21 func (u *user) notify() {
22     fmt.Printf("Sending user email to %s<%s>\n",
23         u.name,
24         u.email)
25 }
26
27 // main 是应用程序的入口
28 func main() {
29     // 创建一个 user 类型的值，并发送通知
```

```
30      u := user{"Bill", "bill@email.com"}
31
32      sendNotification(u)
33
34      // ./listing36.go:32: 不能将 u（类型是 user）作为
35      //                    sendNotification 的参数类型 notifier：
36      //   user 类型并没有实现 notifier
37      //                              （notify 方法使用指针接收者声明）
38  }
39
40  // sendNotification 接受一个实现了 notifier 接口的值
41  // 并发送通知
42  func sendNotification(n notifier) {
43      n.notify()
44  }
```

代码清单 5-36 中的程序虽然看起来没问题，但实际上却无法通过编译。在第 10 行中，声明了一个名为 `notifier` 的接口，包含一个名为 `notify` 的方法。第 15 行中，声明了名为 `user` 的实体类型，并通过第 21 行中的方法声明实现了 `notifier` 接口。这个方法是使用 `user` 类型的指针接收者实现的。

代码清单 5-37　listing36.go：第 40 行到第 44 行

```
40  // sendNotification 接受一个实现了 notifier 接口的值
41  // 并发送通知
42  func sendNotification(n notifier) {
43      n.notify()
44  }
```

在代码清单 5-37 的第 42 行，声明了一个名为 `sendNotification` 的函数。这个函数接收一个 `notifier` 接口类型的值。之后，使用这个接口值来调用 `notify` 方法。任何一个实现了 `notifier` 接口的值都可以传入 `sendNotification` 函数。现在让我们来看一下 `main` 函数，如代码清单 5-38 所示。

代码清单 5-38　listing36.go：第 28 行到第 38 行

```
28  func main() {
29      // 创建一个 user 类型的值，并发送通知
30      u := user{"Bill", "bill@email.com"}
31
32      sendNotification(u)
33
34      // ./listing36.go:32: 不能将 u（类型是 user）作为
35      //                    sendNotification 的参数类型 notifier：
36      //   user 类型并没有实现 notifier
37      //                              （notify 方法使用指针接收者声明）
38  }
```

在 `main` 函数里，代码清单 5-38 的第 30 行，创建了一个 `user` 实体类型的值，并将其赋值给变量 u。之后在第 32 行将 u 的值传入 `sendNotification` 函数。不过，调用 `sendNotification`

的结果是产生了一个编译错误，如代码清单 5-39 所示。

代码清单 5-39　将 user 类型的值存入接口值时产生的编译错误

```
./listing36.go:32: 不能将 u（类型是 user）作为 sendNotification 的参数类型 notifier：
  user 类型并没有实现 notifier（notify 方法使用指针接收者声明）
```

既然 user 类型已经在第 21 行实现了 notify 方法，为什么这里还是产生了编译错误呢？让我们再来看一下那段代码，如代码清单 5-40 所示。

代码清单 5-40　listing36.go：第 08 行到第 12 行，第 21 行到第 25 行

```
08 // notifier 是一个定义了
09 // 通知类行为的接口
10 type notifier interface {
11     notify()
12 }

21 func (u *user) notify() {
22     fmt.Printf("Sending user email to %s<%s>\n",
23         u.name,
24         u.email)
25 }
```

代码清单 5-40 展示了接口是如何实现的，而编译器告诉我们 user 类型的值并没有实现这个接口。如果仔细看一下编译器输出的消息，其实编译器已经说明了原因，如代码清单 5-41 所示。

代码清单 5-41　进一步查看编译器错误

```
(notify method has pointer receiver)
```

要了解用指针接收者来实现接口时为什么 user 类型的值无法实现该接口，需要先了解*方法集*。方法集定义了一组关联到给定类型的值或者指针的方法。定义方法时使用的接收者的类型决定了这个方法是关联到值，还是关联到指针，还是两个都关联。

让我们先解释一下 Go 语言规范里定义的方法集的规则，如代码清单 5-42 所示。

代码清单 5-42　规范里描述的方法集

```
Values                Methods Receivers
-----------------------------------------------
    T                     (t T)
   *T                     (t T) and (t *T)
```

代码清单 5-42 展示了规范里对方法集的描述。描述中说到，T 类型的值的方法集只包含值接收者声明的方法。而指向 T 类型的指针的方法集既包含值接收者声明的方法，也包含指针接收者声明的方法。从值的角度看这些规则，会显得很复杂。让我们从接收者的角度来看一下这些规则，如代码清单 5-43 所示。

代码清单 5-43　从接收者类型的角度来看方法集

```
Methods Receivers     Values
-----------------------------------------------
   (t T)                  T and *T
   (t *T)                 *T
```

代码清单 5-43 展示了同样的规则，只不过换成了接收者的视角。这个规则说，如果使用指针接收者来实现一个接口，那么只有指向那个类型的指针才能够实现对应的接口。如果使用值接收者来实现一个接口，那么那个类型的值和指针都能够实现对应的接口。现在再看一下代码清单 5-36 所示的代码，就能理解出现编译错误的原因了，如代码清单 5-44 所示。

代码清单 5-44　listing36.go：第 28 行到第 38 行

```
28 func main() {
29     // 使用 user 类型创建一个值，并发送通知
30     u := user{"Bill", "bill@email.com"}
31
32     sendNotification(u)
33
34     // ./listing36.go:32: 不能将 u（类型是 user）作为
35     //                    sendNotification 的参数类型 notifier：
36     //   user 类型并没有实现 notifier
37     //                              （notify 方法使用指针接收者声明）
38 }
```

我们使用指针接收者实现了接口，但是试图将 `user` 类型的值传给 `sendNotification` 方法。代码清单 5-44 的第 30 行和第 32 行清晰地展示了这个问题。但是，如果传递的是 `user` 值的地址，整个程序就能通过编译，并且能够工作了，如代码清单 5-45 所示。

代码清单 5-45　listing36.go：第 28 行到第 35 行

```
28 func main() {
29     // 使用 user 类型创建一个值，并发送通知
30     u := user{"Bill", "bill@email.com"}
31
32     sendNotification(&u)
33
34     // 传入地址，不再有错误
35 }
```

在代码清单 5-45 里，这个程序终于可以编译并且运行。因为使用指针接收者实现的接口，只有 `user` 类型的指针可以传给 `sendNotification` 函数。

现在的问题是，为什么会有这种限制？事实上，编译器并不是总能自动获得一个值的地址，如代码清单 5-46 所示。

代码清单 5-46　listing46.go

```
01 // 这个示例程序展示不是总能
02 // 获取值的地址
```

```
03 package main
04
05 import "fmt"
06
07 // duration 是一个基于 int 类型的类型
08 type duration int
09
10 // 使用更可读的方式格式化 duration 值
11 func (d *duration) pretty() string {
12     return fmt.Sprintf("Duration: %d", *d)
13 }
14
15 // main 是应用程序的入口
16 func main() {
17     duration(42).pretty()
18
19     // ./listing46.go:17: 不能通过指针调用 duration(42)的方法
20     // ./listing46.go:17: 不能获取 duration(42)的地址
21 }
```

代码清单 5-46 所示的代码试图获取 duration 类型的值的地址，但是获取不到。这展示了不能总是获得值的地址的一种情况。让我们再看一下方法集的规则，如代码清单 5-47 所示。

代码清单 5-47　再看一下方法集的规则

```
Values                Methods Receivers
-----------------------------------------------
    T                    (t T)
   *T                    (t T) and (t *T)

  Methods Receivers     Values
-----------------------------------------------
   (t T)                 T and *T
   (t *T)                *T
```

因为不是总能获取一个值的地址，所以值的方法集只包括了使用值接收者实现的方法。

5.4.4　多态

现在了解了接口和方法集背后的机制，最后来看一个展示接口的多态行为的例子，如代码清单 5-48 所示。

代码清单 5-48　listing48.go

```
01 // 这个示例程序使用接口展示多态行为
02 package main
03
04 import (
05     "fmt"
06 )
07
```

```
08 // notifier 是一个定义了
09 // 通知类行为的接口
10 type notifier interface {
11     notify()
12 }
13
14 // user 在程序里定义一个用户类型
15 type user struct {
16     name  string
17     email string
18 }
19
20 // notify 使用指针接收者实现了 notifier 接口
21 func (u *user) notify() {
22     fmt.Printf("Sending user email to %s<%s>\n",
23         u.name,
24         u.email)
25 }
26
27 // admin 定义了程序里的管理员
28 type admin struct {
29     name  string
30     email string
31 }
32
33 // notify 使用指针接收者实现了 notifier 接口
34 func (a *admin) notify() {
35     fmt.Printf("Sending admin email to %s<%s>\n",
36         a.name,
37         a.email)
38 }
39
40 // main 是应用程序的入口
41 func main() {
42     // 创建一个 user 值并传给 sendNotification
43     bill := user{"Bill", "bill@email.com"}
44     sendNotification(&bill)
45
46     // 创建一个 admin 值并传给 sendNotification
47     lisa := admin{"Lisa", "lisa@email.com"}
48     sendNotification(&lisa)
49 }
50
51 // sendNotification 接受一个实现了 notifier 接口的值
52 // 并发送通知
53 func sendNotification(n notifier) {
54     n.notify()
55 }
```

在代码清单 5-48 中，我们有了一个展示接口的多态行为的例子。在第 10 行，我们声明了和之前代码清单中一样的 `notifier` 接口。之后第 15 行到第 25 行，我们声明了一个名为 `user` 的结构，并使用指针接收者实现了 `notifier` 接口。在第 28 行到第 38 行，我们声明了一个名

为 admin 的结构，用同样的形式实现了 notifier 接口。现在，有两个实体类型实现了 notifier 接口。

在第 53 行中，我们再次声明了多态函数 sendNotification，这个函数接受一个实现了 notifier 接口的值作为参数。既然任意一个实体类型都能实现该接口，那么这个函数可以针对任意实体类型的值来执行 notifier 方法。因此，这个函数就能提供多态的行为，如代码清单 5-49 所示。

代码清单 5-49　listing48.go：第 40 行到第 49 行

```
40 // main 是应用程序的入口
41 func main() {
42     // 创建一个 user 值并传给 sendNotification
43     bill := user{"Bill", "bill@email.com"}
44     sendNotification(&bill)
45
46     // 创建一个 admin 值并传给 sendNotification
47     lisa := admin{"Lisa", "lisa@email.com"}
48     sendNotification(&lisa)
49 }
```

最后，可以在代码清单 5-49 中看到这种多态的行为。main 函数的第 43 行创建了一个 user 类型的值，并在第 44 行将该值的地址传给了 sendNotification 函数。这最终会导致执行 user 类型声明的 notify 方法。之后，在第 47 行和第 48 行，我们对 admin 类型的值做了同样的事情。最终，因为 sendNotification 接受 notifier 类型的接口值，所以这个函数可以同时执行 user 和 admin 实现的行为。

5.5　嵌入类型

Go 语言允许用户扩展或者修改已有类型的行为。这个功能对代码复用很重要，在修改已有类型以符合新类型的时候也很重要。这个功能是通过嵌入类型（type embedding）完成的。嵌入类型是将已有的类型直接声明在新的结构类型里。被嵌入的类型被称为新的外部类型的内部类型。

通过嵌入类型，与内部类型相关的标识符会提升到外部类型上。这些被提升的标识符就像直接声明在外部类型里的标识符一样，也是外部类型的一部分。这样外部类型就组合了内部类型包含的所有属性，并且可以添加新的字段和方法。外部类型也可以通过声明与内部类型标识符同名的标识符来覆盖内部标识符的字段或者方法。这就是扩展或者修改已有类型的方法。

让我们通过一个示例程序来演示嵌入类型的基本用法，如代码清单 5-50 所示。

代码清单 5-50　listing50.go

```
01 // 这个示例程序展示如何将一个类型嵌入另一个类型，以及
02 // 内部类型和外部类型之间的关系
03 package main
```