GO语言

除了OOP外，近年出现了一些小众的编程哲学，Go语言对这些思想亦有所吸收。例如，Go语言接受了函数式编程的一些想法，支持匿名函数与闭包。再如，Go语言接受了以Erlang语言为代表的面向消息编程思想，支持goroutine和通道，并推荐使用消息而不是共享内存来进行并发编程。总体来说，Go语言是一个非常现代化的语言，精小但非常强大。

Go 语言最主要的特性：

- 自动垃圾回收
- 更丰富的内置类型
- 函数多返回值
- 错误处理
- 匿名函数和闭包
- 类型和接口
- 并发编程
- 反射
- 语言交互性

动态语言因为没有从编译代码到执行代码的中间过程，用动态语言写程序可以快速看到输出。代价是，动态语言不提供静态语言提供的类型安全特性。

类型推导和接口

Go语言支持`var a = 7`语法，这让Go语言有点像动态类型语言，但它实际上是强类型语言，只是变量a定义时会被自动推导出是整数类型。Go语言在代码风格上像动态语言，在运行效率上则像静态编译型语言。

包

使用包来封装不同语义单元的功能。
每个包都在一个单独的目录里。不能把多个包放到同一个目录中，也不能把同一个包的文件分拆到多个不同的目录中。这意味着，同一个目录下的所有.go文件必须声明为同一个包名。在Go语言中，命名为main的包具有特殊的含义。Go语言的编译程序会试图把这种名字的包编译为二进制可执行文件。所有用Go语言编一的可执行程序都必须有一个叫main的包。
import . 这个`.`的含义是，在调用包的函数时，可以省略包名。`_`只用于导入包，而并不需要这个包的其他函数、常量等资源，而是调用了该包的`init`函数。

init函数

每个包可以包含任意多个init函数，这些函数都会在程序执行开始的时候被调用。 所有被编译器发现的init函数都会安排在main函数之前执行。init函数用在设置包、初始化变量或者其他要在程序运行之前优先完成的引导工作。
函数init和main不能有任何的参数和返回值。init函数只能由Go程序自动调用，不可以被外部引用。
init函数可以在任意包中定义，并且可以重复定义多个。main函数只能用于main包中，且只能定义一个。

数据类型

字符串类型、数值类型和布尔类型在Go中是值类型，切片、channel、接口、函数和map属于引用类型。结构体类型可以描述一组不同类型的值，这一组值本质上既可以是引用类型也可以是值类型。

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。**

类型的本质：使用值接收者还是指针接收者，不应该由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。非原始的类型，应该用指针。这条规则的一个例外就是，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。

Go语言中数组、字符串和切片三者是密切相关的数据结构。这3种数据类型，在底层原始数据有着相同的内存结构，在上层，因为语法的限制而有着不同的行为表现。首先，Go语言的数组是一种值类型，虽然数组的元素可以被修改，但是数组本身的赋值和函数传参都是以整体复制的方式处理的。Go语言字符串底层数据也是对应的字节数组，但是字符串的只读属性禁止了在程序中对底层字节数组的元素的修改。字符串赋值只是复制了数据地址和对应的长度，而不会导致底层数据的复制。切片的行为更为灵活，切片的结构和字符串结构类似，但是解除了只读限制。切片的底层数据虽然也是对应数据类型的数组，但是每个切片还有独立的长度和容量信息，切片赋值和函数传参时也是将切片头信息部分按传值方式处理。因为切片头含有底层数据的指针，所以它的赋值也不会导致底层数据的复制。其实Go语言的赋值和函数传参规则很简单，除闭包函数以引用的方式对外部变量访问之外，其他赋值和函数传参都是以传值的方式处理。要理解数组、字符串和切片这3种不同的处理方式的原因，需要详细了解它们的底层数据结构。　　——《Go语言高级编程》

参数传递

Go语言中的函数参数都是按值传递的。

零值

如果声明一个变量而没有给它赋值，该变量将包含其类型的零值。（也就是初始化？）
与变量一样，当创建一个数组时，它所包含的所有的值都初始化为该数组所保存类型的零值。
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。

nil是预定义的一种标识符，代表指针、信道、函数、接口、映射或切片的零值。nil不能赋值给字符串、数值和布尔类型，否则会引发panic类型的错误。

- 字符串：""
- 数值：0
- 布尔：false
- 指针：nil
- 数组：每个数组元素类型对应的零值
- 信道、函数、切片、接口、映射：nil

布尔

布尔类型无法参与数值运算，也无法与其他类型进行转换。

数组

复制指针数组，只会复制指针的值，而不会复制指针所指向的值。复制之后，两个数组指向同一组字符串。Go语言实战Page 57。
Go语言中数组是值语义。一个数组变量即表示整个数组，它并不是隐式地指向第一个元素的指针（例如C语言的数组），而是一个完整的值。当一个数组变量被赋值或者被传递的时候，实际上会复制整个数

组。如果数组较大的话，数组的赋值也会有较大的开销。为了避免复制数组带来的开销，可以传递一个指向数组的指针，但是数组指针并不是数组。　　——《Go语言高级编程》

空数组 && 通道

我们还可以定义一个空的数组：

```
var d [0]int        //定义一个长度为0的数组
var e = [0]int{}    //定义一个长度为0的数组
var f = [...]int{}  //定义一个长度为0的数组
```

长度为0的数组（空数组）在内存中并不占用空间。空数组虽然很少直接使用，但是可以用于强调某种特有类型的操作时避免分配额外的内存空间，例如用于通道的同步操作：

```
c1 := make(chan [0]int)
go func() {
    fmt.Println("c1")
    c1 <- [0]int{}
}()
<-c1
```

在这里，我们并不关心通道中传输数据的真实类型，其中通道接收和发送操作只是用于消息的同步。对于这种场景，我们用空数组作为通道类型可以减少通道元素赋值时的开销。当然，一般更倾向于用无类型的匿名结构体代替空数组：

```
c2 := make(chan struct{})
go func() {
    fmt.Println("c2")
    c2 <- struct{}{}    //struct{}部分是类型，{}表示对应的结构体值
}()
<-c2
```

多维数组：

```
array := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}
array := [4][2]int{1:{20, 21}, 3:{40,41}}
array := [4][2]int{1:{0:20}, 3:{1:41}}
```

在函数间传递数组

在函数间传递数组是一个开销很大的操作。在函数之间传递变量时，总是以值的方式传递的。如果这个变量是一个数组，意味着整个数组，不管有多长，都会完整复制，并传递给函数。

字符串

字符串是**只读**的字符片段。用单引号（'）括起来的是单个字符，用int32表示。反单引号（`）括起来的字符串不会被转义，而是按照原语输出。
string类型的零值为空字符串""，获取字符串中某个字符的地址是非法的，比如&msg[0]。

和数组不同的是，字符串的元素不可修改，是一个只读的字节数组。每个字符串的长度虽然也是固定的，但是字符串的长度并不是字符串类型的一部分。

Go语言字符串的底层结构在reflect.StringHeader中定义：

```
type StringHeader struct {
    Data uintptr
    Len int
}
```

字符串结构由**两个信息组成：第一个是字符串指向的底层字节数组；第二个是字符串的字节的长度。**字符串其实是一个结构体，因此字符串的赋值操作也就是reflect.StringHeader结构体的复制过程，并不会涉及底层字节数组的复制。

切片

```
var myArray [3]int  //数组
var mySlice []int   //切片
```

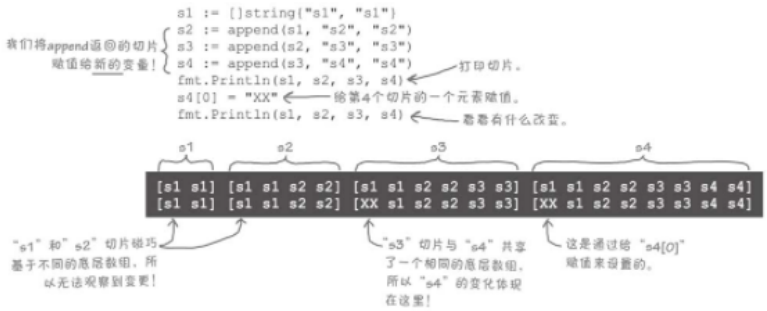
不像数组变量，声明切片变量并不会自动创建一个切片。使用make或字面量来创建和初始化切片。

```
var notes []string
notes = make([]string, 7) //创建7个字符的切片
//或者使用短变量声明，长度和容量都为5
primes := make([]int, 5)
//长度为3，容量为5
slice := make([]int, 3, 5)
```

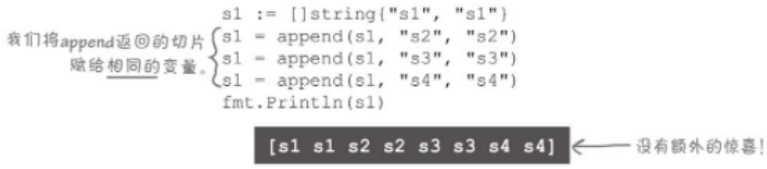
切片并不会自己保存任何数据，它仅仅是底层数组的元素的**视图**。由于切片只是底层数组内容的视图，如果你修改底层数组，这些变化也会反映到切片。给切片的一个元素赋一个新值也会修改底层数组相应的元素。

切片的底层数组并不能增长大小。如果数组没有足够的空间来保存新的元素，所有的元素会被拷贝至一个新的更大的数组，并且切片会被更新为引用这个新的数组。但是由于这些场景都发生在append函数内部，无法知道返回的切片与传入append函数的切片是否具有相同的底层数组。如果你保留了两个切片，会导致一些非预期的错误。

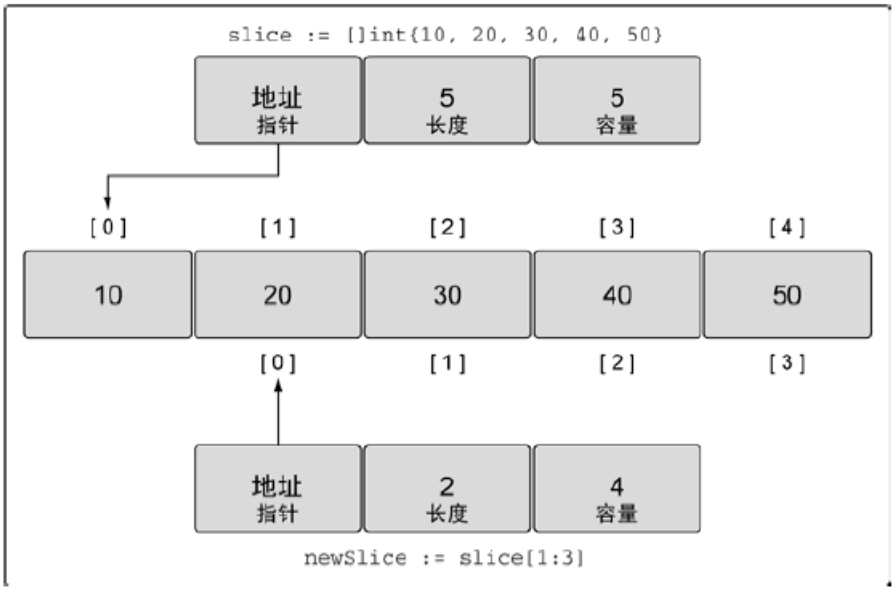
例如我们有4个切片，后三个是通过append调用生成的。我们并没有遵循惯例将append函数的返回值赋给传入的变量。当我们给切片s4的一个元素赋值的时候，我们能看到s3中的体现。因为s4和s3碰巧都共享相同的底层数组。但是改变并没有在s2或者s1中体现，因为它们都有不同的底层数组。



所以我们调用append函数，惯例是将函数的返回值赋给你传入的那个切片变量。如果你只保存一个切片，你就无须考虑两个切片是否共享了同一个底层数组。



切片有3个字段，分别是指向底层数组的指针、切片访问的元素的个数（即长度）和切片允许增长到的元素的个数（即容量）。



在对切片本身进行赋值或参数传递时，和数组指针的操作方式类似，但是只复制切片头信息（reflect.SliceHeader），而不会复制底层的数据。对于类型，和数组的最大不同是，切片的类型和长度信息无关，只要是相同类型元素构成的切片均对应相同的切片类型。

nil切片和空切片

在需要描述一个不存在的切片时，nil切片会很好用。`var a []int`nil切片，和nil相等。切片可以和nil进行比较，只有当切片底层数据指针为空时切片本身才为nil，这时候切片的长度和容量信息将是无效的。如果有切片的底层数据指针为空，但是长度和容量不为0的情况，那么说明切片本身已经被损坏了

空切片：

```
// 空切片的两种方法
slice := make([]int, 0)
slice := []int{}
```

不管是使用nil切片还是空切片，对其调用内置的append、len和cap的效果都是一样的。

append

切片增长：在切片容量小于1000时，成倍增长；在元素个数超过1000时，增长因子为1.25。

```
s1 := []int{1, 2}
s2 := []int{3, 4}
fmt.Printf("%v\n", append(s1, s2...)) //s2中的所有元素都添加到了s1之后。
```

以下来自《Go语言高级编程》

末尾添加：内置的泛型函数append()可以在切片的尾部追加N个元素：注意【解包】

```
var a []int
a = append(a, 1)                //追加一个元素
a = append(a, 1, 2, 3)          //追加多个元素，手写解包方式
a = append(a, []int{1,2,3}...)  //追加一个切片，切片需要解包
```

开头添加：除了在切片的尾部追加，还可以在切片的开头添加元素：

```
var a = []int{1,2,3}
a = append([]int{0}, a...)
a = append([]int{-3, -2, -1}, a...)
```

在开头一般都会导致内存的重新分配，而且会导致已有的元素全部复制一次。因此，从切片的开头添加元素的性能一般要比从尾部追加元素的性能差很多。

中间添加：由于append()函数返回新的切片，也就是它支持链式操作，因此我们可以将多个append ()操作组合起来，实现在切片中间插入元素：

```
var a []int
a = append(a[:i], append([]int{x}, a[i:]...)...)        //在第i个位置插入x
a = append(a[:i], append([]int{1,2,3}, a[i:]...)...)    //在第i个位置插入切片
```

可以用copy和append结合的方式避免使用中间切片来进行在中间添加元素，具体看书1.3.3节。

在本节开头的数组部分我们提到过有类似[0]int的空数组，空数组一般很少用到。但是对于切片来说，len为0但是cap容量不为0的切片则是非常有用的特性。当然，如果len和cap都为0的话，则变成一个真正的空切片，虽然它并不是一个nil的切片。**在判断一个切片是否为空时，一般通过len获取切片的长度来判断，一般很少将切片和nil做直接的比较。**

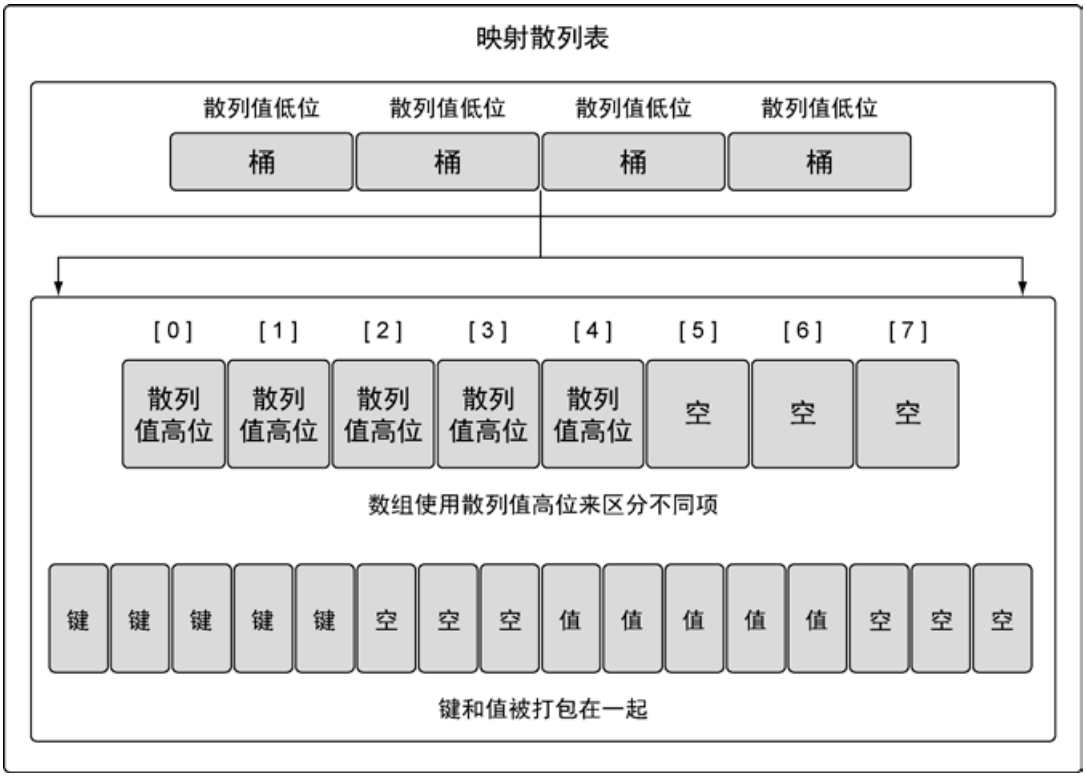
在函数间传递切片

在函数见传递切片就是要在函数间以值的方式传递切片。由于切片尺寸很小，在函数间复制和传递切片成本也很低。在64位架构的机器上，一个切片需要24字节的内存：指针字段需要8个字节，长度和容量分别需要8字节。由于与切片关联的数据包含在底层数组里，不属于切片本身，所以将切片复制到任意函数的时候，对底层数组大小都不会有影响。复制时只会复制切片本身，不会涉及底层数组。

Go语言中，如果以切片为参数调用函数，有时候会给人一种参数采用了传引用的方式的假象：因为在被调用函数内部可以修改传入的切片的元素。其实，任何可以通过函数参数修改调用参数的情形，都是因为函数参数中显式或隐式传入了指针参数。函数参数传值的规范更准确说是只针对数据结构中固定的部分传值，例如字符串或切片对应结构体中的指针和字符串长度结构体传值，但是并不包含指针间接指向的内容。　因为切片中的底层数组部分通过隐式指针传递（指针本身依然是传值的，但是指针指向的却是同一份的数据），所以被调用函数可以通过指针修改调用参数切片中的数据。除数据之外，切片结构还包含了切片长度和切片容量信息，这两个信息也是传值的。如果被调用函数中修改了Len或Cap信息，就无法反映到调用参数的切片中，这时候我们一般会通过返回修改后的切片来更新之前的切片。这也是内置的append ()必须要返回一个切片的原因。　——《Go语言高级编程》

映射

映射一个存储键值对的无序集合。



在我们的例子里，键是字符串，代表颜色。这些字符串会转换为一个数值（散列值）。这个数值落在映射已有桶的序号范围内表示一个可以用于存储的桶的序号。之后，这个数值就被用于选择桶，用于存储或者查找指定的键值对。对Go 语言的映射来说，**生成的散列键的一部分，具体来说是低位（LOB），被用来选择桶。**
如果再仔细看看图4-24，就能看出桶的内部实现。映射使用两个数据结构来存储数据。第一个数据结构是一个数组，内部存储的是用于选择桶的散列键的高八位值。这个数组用于区分每个键值对要存在哪个桶里。第二个数据结构是一个字节数组，用于存储键值对。该字节数组先依次存储了这个桶里所有的键，之后依次存储了这个桶里所有的值。实现这种键值对的存储方式目的在于减少每个桶所需的内存。

切片、函数以及包含切片的结构类型这些类型由于具有引用语义，不能作为映射的键，使用这些类型会造成编译错误。

在函数间传递映射时，并不会制造出该映射的一个副本。

make与new

new一般用来获取类型对应的指针类型，而make只用来分配管道、字典和切片的创建等。make返回的是传入的类型，而不是指针。

```
// The new built-in function allocates memory. The first argument is a type,
// not a value, and the value returned is a pointer to a newly
// allocated zero value of that type.
func new(Type) *Type
```

```
// The make built-in function allocates and initializes an object of type
// slice, map, or chan (only). Like new, the first argument is a type, not a
// value. Unlike new, make's return type is the same as the type of its
// argument, not a pointer to it. The specification of the result depends on
// the type:
// Slice: The size specifies the length. The capacity of the slice is
// equal to its length. A second integer argument may be provided to
// specify a different capacity; it must be no smaller than the
// length. For example, make([]int, 0, 10) allocates an underlying array
// of size 10 and returns a slice of length 0 and capacity 10 that is
// backed by this underlying array.
// Map: An empty map is allocated with enough space to hold the
// specified number of elements. The size may be omitted, in which case
// a small starting size is allocated.
// Channel: The channel's buffer is initialized with the specified
// buffer capacity. If zero, or the size is omitted, the channel is
// unbuffered.
func make(t Type, size ...IntegerType) Type
```

函数

函数还可以返回一个函数。

函数重载

Go简单地不支持函数重载。但可以用方法实现。

函数中的变量存储（堆栈）

栈是计算机内存中的一个区域，主要用于存储由函数创建的局部变量。当函数调用完成后，栈中存储的局部变量的内存会被自动清空，**操作系统可有效管理栈内内存空间，因此内存不会碎片化。**由于是内存连续的结构，因此存取数据也比较快。栈的内存大小限制取决于操作系统本身，且无法动态调整变量的内存大小。栈的内存是非常有限的，在栈上创建太多变量可能会增加栈溢出的风险。如果递归调用太多，就可能会导致栈内存溢出的情况。

堆是主要用来存储全局变量或大对象的地方。一般来说，所有全局变量都存储在堆内存空间中，它支持动态内存分配。堆中的变量一般会由垃圾回收机制来定期清理，但是如果语言本身没有自动垃圾回收，就需要程序员自行清理内存，否则比较容易造成内存泄漏。堆中的内存结构往往不是连续的，因此读取数据的速度相对于栈来说慢一些。堆内存管理比栈内存管理更加复杂，执行的时间也比栈更长。但是，堆可以进行全局变量操作，且能使用操作系可以提供的最大内存来存取变量。

递归调用

Go语言中，函数还可以直接或间接地调用自己，也就是支持递归调用。Go语言函数的递归调用深度在逻辑上没有限制，**函数调用的栈是不会出现溢出错误的，因为Go语言运行时会根据需要动态地调整函数栈的大小。**

封装

Go中使用未导出的变量、struct字段、函数或方法，把数据封装在包中。

defer

延迟调用。defer语句经常用于对资源进行释放的场景，比如释放数据库链接、解锁和关闭文件等。因此，它在一些需要回收资源的场景非常有用，可以方便地在函数推出前做一些清理工作。

嵌入

一个类型使用匿名字段的方式保存到另一个struct类型中，被称为嵌入了struct。嵌入类型的方法会提升到外部类型。它们可以被调用，就像它们是在外部类型上定义的一样。

Go开发者使用组合设计模式，只需简单地将一个类型嵌入到另一个类型，就能复用所有的功能。

方法

Go使用接收器参数来代替self和this。两者有着巨大的不同，self和this是隐含的，而Go中是显式地声明一个接收器参数。
你只能为定义在当前包的类型定义方法。为一个像int一样全局定义的类型定义方法会导致编译错误。

当你用一个非指针的变量调用一个需要指针的接收器的方法的时候，Go会自动为你将非指针类型转换为指针类型。同样指针类型也会自动转换为非指针类型，如果你调用一个要求值类型的接收器，Go会自动帮你获取指针指向的值，然后传递给方法。

```
//Package, imports, type omitted
func (n *Number) Double() {
    *n *= 2
}
func main() {
    number := Number(4)
    fmt.Println("Original value of number:", number)
    number.Double() //不需要改方法的调用
}
```

值接收者使用值的副本来调用方法，而指针接收者使用实际值来调用方法。

在C++语言中方法对应一个类对象的成员函数，是关联到具体对象上的虚表中的。但是Go语言的方法却是关联到类型的，这样可以在编译阶段完成方法的静态绑定。

Go语言中方法是编译时静态绑定的。如果需要虚函数的多态特性，我们需要借助Go语言接口来实现。

接口

在Go中，一个接口被定义为特定值预期具有的**一组方法。**你可以把接口看作需要类型实现的一组行为。接口类型并不描述是哪个值：它们不说它的基础类型是什么，或者数据是如何存储的。它们仅仅描述了这个值能做什么：它有哪些方法。

如果一个类型声明了指针接收器方法，你就只能将那个类型的指针传递给接口变量。

Go语言的多态通过接口来实现。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。

接口类型值的形式

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。

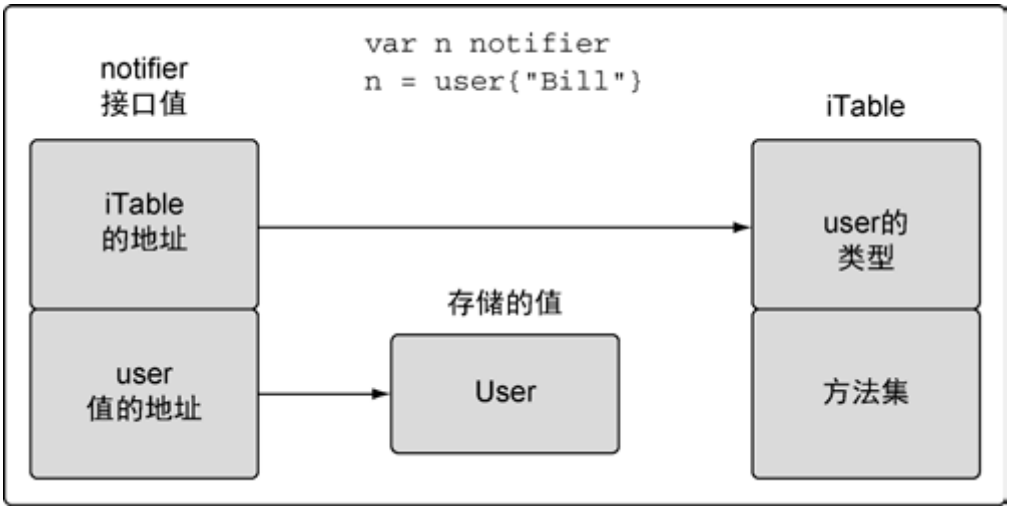
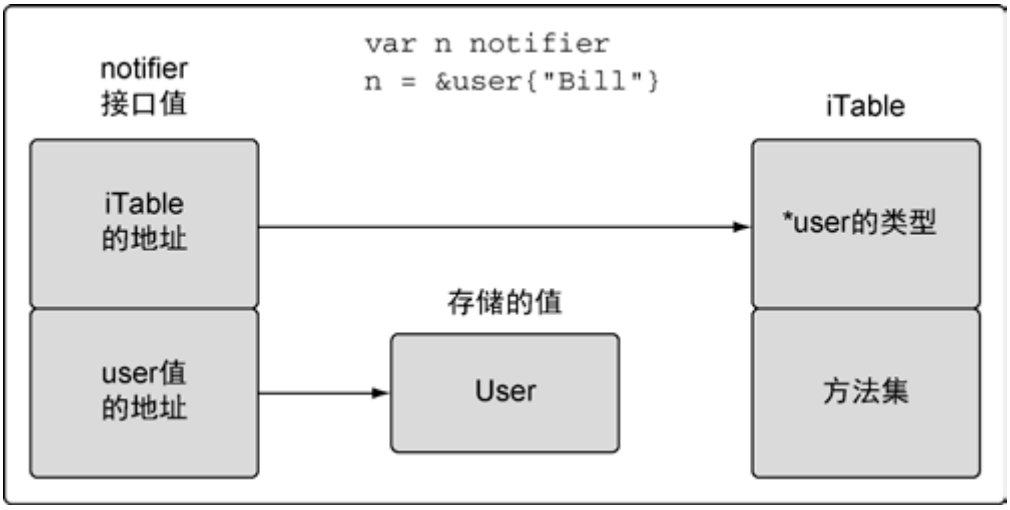


图 5-2 展示了一个**指针赋值**给接口之后发生的变化。在这种情况里，类型信息会存储一个指向保存的类型的指针，而接口值第二个字依旧保存指向实体值的指针。



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。

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 5-43　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
```

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。

> 为什么会有这种限制?

因为编译器并不能总是自动获得一个值的地址。

goroutine

在Go语言中会使用同一个线程来执行多个goroutine，它占用的内存远少于线程，使用它需要的代码更少。

Go 语言的并发同步模型来自一个叫作通信顺序进程（Communicating Sequential Processes，CSP）的范型（paradigm）。CSP 是一种消息传递模型，通过在goroutine 之间传递数据来传递消息，而不是对数据进行加锁来实现同步访问。用于在goroutine 之间同步和传递数据的关键数据类型叫作通道（channel）。

操作系统会在物理处理器上调度线程来运行，而Go 语言的运行时会在逻辑处理器上调度goroutine来运行。

channel

channel通过blocking（阻塞）——暂停当前goroutine中的所有进一步操作来实现这一点。发送操作阻塞发送goroutine，直到另一个goroutine在同一channel上执行了接收操作。 | channel可以帮用户避免其他语言里的共享内存访问的问题。

需要强调的是，通道并不提供跨goroutine的数据访问保护机制。如果通过通道传输数据的一份副本，那么每个goroutine都持有一份副本，各自对自己的副本修改是安全的。当传输的是指向数据的指针时，如果读和写是由不同的goroutine完成的，每个goroutine依旧需要额外的同步动作。

在Go 语言里，你不仅可以使用原子函数和互斥锁来保证对共享资源的安全访问以及消除竞争状态，还可以使用通道，**通过发送和接收需要共享的资源**，在goroutine 之间做同步。声明通道时，需要指定将要被共享的数据的类型。可以**通过通道共享内置类型、命名类型、结构类型和引用类型的值或者指针。**

```
unbuffered := make(chan int)
buffered := make(chan string, 10)
```

可以看到使用内置函数make 创建了两个通道，一个无缓冲的通道，一个有缓冲的通道。make 的第一个参数需要是关键字chan，之后跟着允许通道交换的数据的
类型。如果创建的是一个有缓冲的通道，之后还需要在第二个参数指定这个通道的缓冲区的大小。

无缓冲通道：无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何值的通道。这种类型的通道**要求发送goroutine 和接收goroutine 同时准备好**，才能完成发送和接收操作。如果两个goroutine没有同时准备好，通道会导致先执行发送或接收操作的goroutine **阻塞等待。**这种对通道进行发送和接收的**交互行为本身就是同步的。**其中任意一个操作都无法离开另一个操作单独存在。

有缓冲通道：有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个值的通道。这种类型的通道并不强制要求goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：**无缓冲的通道保证进行发送和接收的goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。**

当通道关闭后，goroutine 依旧可以从通道接收数据，但是不能再向通道里发送数据。能够从已经关闭的通道接收数据这一点非常重要，因为这允许通道关闭后依旧能取出其中缓冲的全部值，而不会有数据丢失。

信号

《Go并发编程实战》P73：再看os/signal代码包中的Notify和Stop函数。它们都是以signal接收通道为唯一标识来对相应的信号集合进行处理的。在signal处理程序的内部，存在一个包级私有的字典，这个字典用于存放以signal接收通道为键并以信号集合的变体为元素的键值对。

Go mod

To build Go code, there are several options:

- Set up a VPN and access google.golang.org through that.
- Without Go module support: `git clone` the repo manually:

  ```
  git clone https://github.com/grpc/grpc-go.git $GOPATH/src/google.golang.org/grpc
  ```

  You will need to do the same for all of grpc's dependencies in `golang.org`, e.g. `golang.org/x/net`.

- With Go module support: it is possible to use the `replace` feature of `go mod` to create aliases for golang.org packages. In your project's directory:

  ```
  go mod edit -replace=google.golang.org/grpc=github.com/grpc/grpc-go@latest
  go mod tidy
  go mod vendor
  go build -mod=vendor
  ```

  Again, this will need to be done for all transitive dependencies hosted on golang.org as well. For details, refer to golang/go issue #28652.