# 1.Go语言学习

**1.1nil 是什么？**

列子：

If err := a.function(); err != nil {

//do something

}

当出现不等于nil的时候，说明出现了某些错误，这时候需要对这个错误进行一些处理。反之，如果等于nil说明程序运行正常。

nil 是无或者零值的意思。

在Go语言中，当你声明了一个变量，但并没有对它进行赋值，就会有相应类型的默认的零值，以下是每种类型对应的零值：

bool -> false

numbers -> 0

string -> ""

pointers -> nil

slices -> nil

maps -> nil

channels -> nil

functions -> nil

interfaces -> nil

nil是预定义的标识符，代表指针、通道、函数、接口、映射或切片的零值，也就是预定义好的一个变量：

type Type int

var nil Type

所以,nil 并不是GO 语言的关键字。可以自己改变nil的值：比如

var nil = errors.New("hi")//但是不建议这样做。

**1.2 nil 的作用**

**1.3 Pointer**

指针是指向内存的地址，如果对为nil的指针进行解引用，就会导致panic. 所以在使用nil指针时，判断一下：

If p == nil {

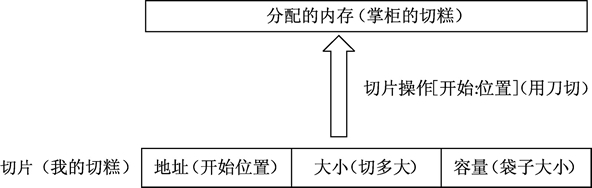
return

}

**1.4 Slice**

**1.4.1 切片的生成和声明**

切片（slice）是对数组的一个连续片段的引用，所以切片是一个引用类型（因此更类似于 C/[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 中的数组类型，或者 [Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank) 中的 list 类型），这个片段可以是整个数组，也可以是由起始和终止索引标识的一些项的子集，需要注意的是，终止索引标识的项不包括在切片内。  
  
Go语言中切片的内部结构包含地址、大小和容量，切片一般用于快速地操作一块数据集合，如果将数据集合比作切糕的话，切片就是你要的“那一块”，切的过程包含从哪里开始（切片的起始位置）及切多大（切片的大小），容量可以理解为装切片的口袋大小，如下图所示。

  
图：切片结构和内存分配

#### **1.4.1.1从数组或切片生成新的切片**

切片默认指向一段连续内存区域，可以是数组，也可以是切片本身。  
从连续内存区域生成切片是常见的操作，格式如下：

slice [开始位置 : 结束位置]

* 语法说明如下：
* slice：表示目标切片对象；
* 开始位置：对应目标切片对象的索引；
* 结束位置：对应目标切片的结束索引。

从数组生成切片，代码如下：

var a = [3]int{1, 2, 3}

fmt.Println(a, a[1:2])

其中 a 是一个拥有 3 个整型元素的数组，被初始化为数值 1 到 3，使用 a[1:2] 可以生成一个新的切片，代码运行结果如下：

[1 2 3]  [2]

其中 [2] 就是 a[1:2] 切片操作的结果。  
从数组或切片生成新的切片拥有如下特性：

* 取出的元素数量为：结束位置 - 开始位置；
* 取出元素不包含结束位置对应的索引，切片最后一个元素使用 slice[len(slice)] 获取；
* 当缺省开始位置时，表示从连续区域开头到结束位置；
* 当缺省结束位置时，表示从开始位置到整个连续区域末尾；
* 两者同时缺省时，与切片本身等效；
* 两者同时为 0 时，等效于空切片，一般用于切片复位。

根据索引位置取切片 slice 元素值时，取值范围是（0～len(slice)-1），超界会报运行时错误，生成切片时，结束位置可以填写 len(slice) 但不会报错。  
下面通过实例来熟悉切片的特性。

#### **1.4.1.2从指定范围中生成切片**

切片和数组密不可分，如果将数组理解为一栋办公楼，那么切片就是把不同的连续楼层出租给使用者，出租的过程需要选择开始楼层和结束楼层，这个过程就会生成切片，示例代码如下：

var highRiseBuilding [30]int

for i := 0; i < 30; i++ {

highRiseBuilding[i] = i + 1

}

// 区间

fmt.Println(highRiseBuilding[10:15])

// 中间到尾部的所有元素

fmt.Println(highRiseBuilding[20:])

// 开头到中间指定位置的所有元素

fmt.Println(highRiseBuilding[:2])

代码输出如下：

[11 12 13 14 15]  
[21 22 23 24 25 26 27 28 29 30]  
[1 2]

代码中构建了一个 30 层的高层建筑，数组的元素值从 1 到 30，分别代表不同的独立楼层，输出的结果是不同的租售方案。  
代码说明如下：

* 第 8 行，尝试出租一个区间楼层。
* 第 11 行，出租 20 层以上。
* 第 14 行，出租 2 层以下，一般是商用铺面。

切片有点像C语言里的指针，指针可以做运算，但代价是内存操作越界，切片在指针的基础上增加了大小，约束了切片对应的内存区域，切片使用中无法对切片内部的地址和大小进行手动调整，因此切片比指针更安全、强大。

#### **1.4.1.3表示原有的切片**

生成切片的格式中，当开始和结束位置都被忽略时，生成的切片将表示和原切片一致的切片，并且生成的切片与原切片在数据内容上也是一致的，代码如下：

a := []int{1, 2, 3}

fmt.Println(a[:])

a 是一个拥有 3 个元素的切片，将 a 切片使用 a[:] 进行操作后，得到的切片与 a 切片一致，代码输出如下：

[1 2 3]

#### **1.4.1.4重置切片，清空拥有的元素**

把切片的开始和结束位置都设为 0 时，生成的切片将变空，代码如下：

a := []int{1, 2, 3}

fmt.Println(a[0:0])

代码输出如下：

[]

#### **1.4.1.5直接声明新的切片**

除了可以从原有的数组或者切片中生成切片外，也可以声明一个新的切片，每一种类型都可以拥有其切片类型，表示多个相同类型元素的连续集合，因此切片类型也可以被声明，切片类型声明格式如下：

var name []Type

其中 name 表示切片的变量名，Type 表示切片对应的元素类型。  
下面代码展示了切片声明的使用过程：

// 声明字符串切片

var strList []string

// 声明整型切片

var numList []int

// 声明一个空切片

var numListEmpty = []int{}

// 输出3个切片

fmt.Println(strList, numList, numListEmpty)

// 输出3个切片大小

fmt.Println(len(strList), len(numList), len(numListEmpty))

// 切片判定空的结果

fmt.Println(strList == nil)

fmt.Println(numList == nil)

fmt.Println(numListEmpty == nil)

代码输出结果：

[] [] []  
0 0 0  
true  
true  
false

代码说明如下：

* 第 2 行，声明一个字符串切片，切片中拥有多个字符串。
* 第 5 行，声明一个整型切片，切片中拥有多个整型数值。
* 第 8 行，将 numListEmpty 声明为一个整型切片，本来会在{}中填充切片的初始化元素，这里没有填充，所以切片是空的，但是此时的 numListEmpty 已经被分配了内存，只是还没有元素。
* 第 11 行，切片均没有任何元素，3 个切片输出元素内容均为空。
* 第 14 行，没有对切片进行任何操作，strList 和 numList 没有指向任何数组或者其他切片。
* 第 17 行和第 18 行，声明但未使用的切片的默认值是 nil，strList 和 numList 也是 nil，所以和 nil 比较的结果是 true。
* 第 19 行，numListEmpty 已经被分配到了内存，但没有元素，因此和 nil 比较时是 false。

切片是动态结构，只能与 nil 判定相等，不能互相判定相等。声明新的切片后，可以使用 [append()](http://c.biancheng.net/view/28.html" \t "http://c.biancheng.net/view/_blank) 函数向切片中添加元素。

#### **1.4.1.6使用 make() 函数构造切片**

如果需要动态地创建一个切片，可以使用 make() 内建函数，格式如下：

make( []Type, size, cap )

其中 Type 是指切片的元素类型，size 指的是为这个类型分配多少个元素，cap 为预分配的元素数量，这个值设定后不影响 size，只是能提前分配空间，降低多次分配空间造成的性能问题。  
示例如下：

a := make([]int, 2)

b := make([]int, 2, 10)

fmt.Println(a, b)

fmt.Println(len(a), len(b))

代码输出如下：

[0 0] [0 0]  
2 2

其中 a 和 b 均是预分配 2 个元素的切片，只是 b 的内部存储空间已经分配了 10 个，但实际使用了 2 个元素。  
  
容量不会影响当前的元素个数，因此 a 和 b 取 len 都是 2。

***温馨提示***:

使用 make() 函数生成的切片一定发生了内存分配操作，但给定开始与结束位置（包括切片复位）的切片只是将新的切片结构指向已经分配好的内存区域，设定开始与结束位置，不会发生内存分配操作。

**1.4.2 使用append()为切片添加元素**

#### **1.4.2.1 在切片尾部添加**

Go语言的内建函数 append() 可以为切片动态添加元素，代码如下所示：

var a []int

a = append(a, 1) // 追加1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

不过需要注意的是，在使用 append() 函数为切片动态添加元素时，如果空间不足以容纳足够多的元素，切片就会进行“扩容”，此时新切片的长度会发生改变。  
  
切片在扩容时，容量的扩展规律是按容量的 2 倍数进行扩充，例如 1、2、4、8、16……，代码如下：

var numbers []int

for i := 0; i < 10; i++ {

numbers = append(numbers, i)

fmt.Printf("len: %d cap: %d pointer: %p\n", len(numbers), cap(numbers), numbers)

}

代码输出如下：

len: 1  cap: 1 pointer: 0xc0420080e8  
len: 2  cap: 2 pointer: 0xc042008150  
len: 3  cap: 4 pointer: 0xc04200e320  
len: 4  cap: 4 pointer: 0xc04200e320  
len: 5  cap: 8 pointer: 0xc04200c200  
len: 6  cap: 8 pointer: 0xc04200c200  
len: 7  cap: 8 pointer: 0xc04200c200  
len: 8  cap: 8 pointer: 0xc04200c200  
len: 9  cap: 16 pointer: 0xc042074000  
len: 10  cap: 16 pointer: 0xc042074000

代码说明如下：

* 第 1 行，声明一个整型切片。
* 第 4 行，循环向 numbers 切片中添加 10 个数。
* 第 5 行，打印输出切片的长度、容量和指针变化，使用函数 len() 查看切片拥有的元素个数，使用函数 cap() 查看切片的容量情况。

通过查看代码输出，可以发现一个有意思的规律：切片长度 len 并不等于切片的容量 cap。  
  
往一个切片中不断添加元素的过程，类似于公司搬家，公司发展初期，资金紧张，人员很少，所以只需要很小的房间即可容纳所有的员工，随着业务的拓展和收入的增加就需要扩充工位，但是办公地的大小是固定的，无法改变，因此公司只能选择搬家，每次搬家就需要将所有的人员转移到新的办公点。

* 员工和工位就是切片中的元素。
* 办公地就是分配好的内存。
* 搬家就是重新分配内存。
* 无论搬多少次家，公司名称始终不会变，代表外部使用切片的变量名不会修改。
* 由于搬家后地址发生变化，因此内存“地址”也会有修改。

#### **1.4.2.2 在切片开头添加**

除了在切片的尾部追加，我们还可以在切片的开头添加元素：

var a = []int{1,2,3}

a = append([]int{0}, a...) // 在开头添加1个元素

a = append([]int{-3,-2,-1}, a...) // 在开头添加1个切片

在切片开头添加元素一般都会导致内存的重新分配，而且会导致已有元素全部被复制 1 次，因此，从切片的开头添加元素的性能要比从尾部追加元素的性能差很多。  
  
因为 append 函数返回新切片的特性，所以切片也支持链式操作，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

每个添加操作中的第二个 append 调用都会创建一个临时切片，并将 a[i:] 的内容复制到新创建的切片中，然后将临时创建的切片再追加到 a[:i] 中。

**1.4.3 复制切片**

Go语言的内置函数 copy() 可以将一个数组切片复制到另一个数组切片中，如果加入的两个数组切片不一样大，就会按照其中较小的那个数组切片的元素个数进行复制。  
  
copy() 函数的使用格式如下：

copy( destSlice, srcSlice []T) int

其中 srcSlice 为数据来源切片，destSlice 为复制的目标（也就是将 srcSlice 复制到 destSlice），目标切片必须分配过空间且足够承载复制的元素个数，并且来源和目标的类型必须一致，copy() 函数的返回值表示实际发生复制的元素个数。  
  
下面的代码展示了使用 copy() 函数将一个切片复制到另一个切片的过程：

slice1 := []int{1, 2, 3, 4, 5}

slice2 := []int{5, 4, 3}

copy(slice2, slice1) // 只会复制slice1的前3个元素到slice2中

copy(slice1, slice2) // 只会复制slice2的3个元素到slice1的前3个位置

虽然通过循环复制切片元素更直接，不过内置的 copy() 函数使用起来更加方便，copy() 函数的第一个参数是要复制的目标 slice，第二个参数是源 slice，两个 slice 可以共享同一个底层数组，甚至有重叠也没有问题。

【示例】通过代码演示对切片的引用和复制操作后对切片元素的影响。

package main

import "fmt"

func main() {

// 设置元素数量为1000

const elementCount = 1000

// 预分配足够多的元素切片

srcData := make([]int, elementCount)

// 将切片赋值

for i := 0; i < elementCount; i++ {

srcData[i] = i

}

// 引用切片数据

refData := srcData

// 预分配足够多的元素切片

copyData := make([]int, elementCount)

// 将数据复制到新的切片空间中

copy(copyData, srcData)

// 修改原始数据的第一个元素

srcData[0] = 999

// 打印引用切片的第一个元素

fmt.Println(refData[0])

// 打印复制切片的第一个和最后一个元素

fmt.Println(copyData[0], copyData[elementCount-1])

// 复制原始数据从4到6(不包含)

copy(copyData, srcData[4:6])

for i := 0; i < 5; i++ {

fmt.Printf("%d ", copyData[i])

}

}

代码说明如下：

* 第 8 行，定义元素总量为 1000。
* 第 11 行，预分配拥有 1000 个元素的整型切片，这个切片将作为原始数据。
* 第 14～16 行，将 srcData 填充 0～999 的整型值。
* 第 19 行，将 refData 引用 srcData，切片不会因为等号操作进行元素的复制。
* 第 22 行，预分配与 srcData 等大（大小相等）、同类型的切片 copyData。
* 第 24 行，使用 copy() 函数将原始数据复制到 copyData 切片空间中。
* 第 27 行，修改原始数据的第一个元素为 999。
* 第 30 行，引用数据的第一个元素将会发生变化。
* 第 33 行，打印复制数据的首位数据，由于数据是复制的，因此不会发生变化。
* 第 36 行，将 srcData 的局部数据复制到 copyData 中。
* 第 38～40 行，打印复制局部数据后的 copyData 元素。

**1.4.4 删除切片**

Go语言并没有对删除切片元素提供专用的语法或者接口，需要使用切片本身的特性来删除元素，根据要删除元素的位置有三种情况，分别是从开头位置删除、从中间位置删除和从尾部删除，其中删除切片尾部的元素速度最快。

#### **1.4.4.1从开头位置删除**

删除开头的元素可以直接移动数据指针：

a = []int{1, 2, 3}

a = a[1:] // 删除开头1个元素

a = a[N:] // 删除开头N个元素

也可以不移动数据指针，但是将后面的数据向开头移动，可以用 append 原地完成（所谓原地完成是指在原有的切片数据对应的内存区间内完成，不会导致内存空间结构的变化）：

a = []int{1, 2, 3}

a = append(a[:0], a[1:]...) // 删除开头1个元素

a = append(a[:0], a[N:]...) // 删除开头N个元素

还可以用 copy() 函数来删除开头的元素：

a = []int{1, 2, 3}

a = a[:copy(a, a[1:])] // 删除开头1个元素

a = a[:copy(a, a[N:])] // 删除开头N个元素

#### **1.4.4.2从中间位置删除**

对于删除中间的元素，需要对剩余的元素进行一次整体挪动，同样可以用 append 或 copy 原地完成：

a = []int{1, 2, 3, ...}

a = append(a[:i], a[i+1:]...) // 删除中间1个元素

a = append(a[:i], a[i+N:]...) // 删除中间N个元素

a = a[:i+copy(a[i:], a[i+1:])] // 删除中间1个元素

a = a[:i+copy(a[i:], a[i+N:])] // 删除中间N个元素

#### **1.4.4.3从尾部删除**

a = []int{1, 2, 3}

a = a[:len(a)-1] // 删除尾部1个元素

a = a[:len(a)-N] // 删除尾部N个元素  
 删除开头的元素和删除尾部的元素都可以认为是删除中间元素操作的特殊情况，下面来看一 个示例。  
 【示例】删除切片指定位置的元素。

package main

import "fmt"

func main() {

seq := []string{"a", "b", "c", "d", "e"}

// 指定删除位置

index := 2

// 查看删除位置之前的元素和之后的元素

fmt.Println(seq[:index], seq[index+1:])

// 将删除点前后的元素连接起来

seq = append(seq[:index], seq[index+1:]...)

fmt.Println(seq)

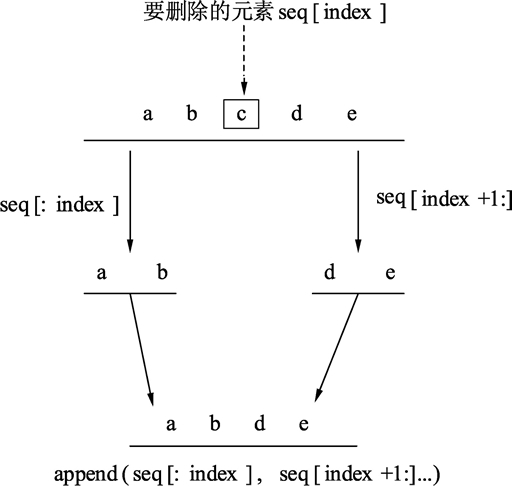
}

代码输出结果：

[a b] [d e]  
[a b d e]

代码说明如下：

* 第 1 行，声明一个整型切片，保存含有从 a 到 e 的字符串。
* 第 4 行，为了演示和讲解方便，使用 index 变量保存需要删除的元素位置。
* 第 7 行，seq[:index] 表示的就是被删除元素的前半部分，值为 [1 2]，seq[index+1:] 表示的是被删除元素的后半部分，值为 [4 5]。
* 第 10 行，使用 append() 函数将两个切片连接起来。
* 第 12 行，输出连接好的新切片，此时，索引为 2 的元素已经被删除。  
  代码的删除过程可以使用下图来描述。

  
图：切片删除元素的操作过程

Go语言中删除切片元素的本质是，以被删除元素为分界点，将前后两个部分的内存重新连接起来。

***提示***：

连续容器的元素删除无论在任何语言中，都要将删除点前后的元素移动到新的位置，随着元素的增加，这个过程将会变得极为耗时，因此，当业务需要大量、频繁地从一个切片中删除元素时，如果对性能要求较高的话，就需要考虑更换其他的容器了（如双链表等能快速从删除点删除元素）。

**1.4.5 利用关键字range和for迭代切片中的元素**

通过前面的学习我们了解到切片其实就是多个相同类型元素的连续集合，既然切片是一个集合，那么我们就可以迭代其中的元素，Go语言有个特殊的关键字 range，它可以配合关键字 for 来迭代切片里的每一个元素，如下所示：

// 创建一个整型切片，并赋值

slice := []int{10, 20, 30, 40}

// 迭代每一个元素，并显示其值

for index, value := range slice {

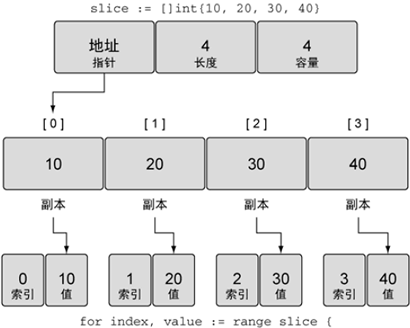
fmt.Printf("Index: %d Value: %d\n", index, value)

}

第 4 行中的 index 和 value 分别用来接收 range 关键字返回的切片中每个元素的索引和值，这里的 index 和 value 不是固定的，读者也可以定义成其它的名字。  
  
上面代码的输出结果为：

Index: 0 Value: 10  
Index: 1 Value: 20  
Index: 2 Value: 30  
Index: 3 Value: 40

当迭代切片时，关键字 range 会返回两个值，第一个值是当前迭代到的索引位置，第二个值是该位置对应元素值的一份副本，如下图所示。

  
图：使用 range 迭代切片会创建每个元素的副本

需要强调的是，range 返回的是每个元素的副本，而不是直接返回对该元素的引用，如下所示。  
  
【示例 1】range 提供了每个元素的副本

// 创建一个整型切片，并赋值

slice := []int{10, 20, 30, 40}

// 迭代每个元素，并显示值和地址

for index, value := range slice {

fmt.Printf("Value: %d Value-Addr: %X ElemAddr: %X\n", value, &value, &slice[index])

}

输出结果为：

Value: 10 Value-Addr: 10500168 ElemAddr: 1052E100  
Value: 20 Value-Addr: 10500168 ElemAddr: 1052E104  
Value: 30 Value-Addr: 10500168 ElemAddr: 1052E108  
Value: 40 Value-Addr: 10500168 ElemAddr: 1052E10C

因为迭代返回的变量是一个在迭代过程中根据切片依次赋值的新变量，所以 value 的地址总是相同的，要想获取每个元素的地址，需要使用切片变量和索引值（例如上面代码中的 &slice[index]）。  
  
如果不需要索引值，也可以使用下划线\_来忽略这个值，代码如下所示。  
  
【示例 2】使用空白标识符（下划线）来忽略索引值

// 创建一个整型切片，并赋值

slice := []int{10, 20, 30, 40}

// 迭代每个元素，并显示其值

for \_, value := range slice {

fmt.Printf("Value: %d\n", value)

}

输出结果为：

Value: 10  
Value: 20  
Value: 30  
Value: 40

关键字 range 总是会从切片头部开始迭代。如果想对迭代做更多的控制，则可以使用传统的 for 循环，代码如下所示。  
  
【示例 3】使用传统的 for 循环对切片进行迭代

// 创建一个整型切片，并赋值

slice := []int{10, 20, 30, 40}

// 从第三个元素开始迭代每个元素

for index := 2; index < len(slice); index++ {

fmt.Printf("Index: %d Value: %d\n", index, slice[index])

}

输出结果为：

Index: 2 Value: 30  
Index: 3 Value: 40

在前面几节的学习中我们了解了两个特殊的内置函数 len() 和 cap()，可以用于处理数组、切片和通道，对于切片，函数 len() 可以返回切片的长度，函数 cap() 可以返回切片的容量，在上面的示例中，使用到了函数 len() 来控制循环迭代的次数。  
  
当然，range 关键字不仅仅可以用来遍历切片，它还可以用来遍历数组、字符串、map 或者通道等。

**1.4.6 多维切片**

Go语言中同样允许使用多维切片，声明一个多维数组的语法格式如下：

var sliceName [][]...[]sliceType

其中，sliceName 为切片的名字，sliceType为切片的类型，每个[ ]代表着一个维度，切片有几个维度就需要几个[ ]。  
  
下面以二维切片为例，声明一个二维切片并赋值，代码如下所示。

//声明一个二维切片

var slice [][]int

//为二维切片赋值

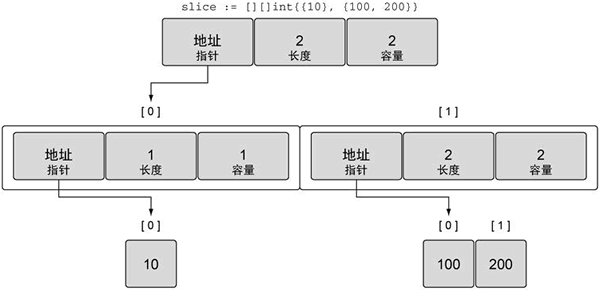
slice = [][]int{{10}, {100, 200}}

上面的代码也可以简写为下面的样子。

// 声明一个二维整型切片并赋值

slice := [][]int{{10}, {100, 200}}

上面的代码中展示了一个包含两个元素的外层切片，同时每个元素包又含一个内层的整型切片，切片 slice 的值如下图所示。

  
图：整型切片的切片的值

通过上图可以看到外层的切片包括两个元素，每个元素都是一个切片，第一个元素中的切片使用单个整数 10 来初始化，第二个元素中的切片包括两个整数，即 100 和 200。  
  
这种组合可以让用户创建非常复杂且强大的[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)，前面介绍过的关于内置函数[append()](http://c.biancheng.net/view/28.html" \t "http://c.biancheng.net/view/_blank)的规则也可以应用到组合后的切片上，如下所示。  
  
【示例】组合切片的切片

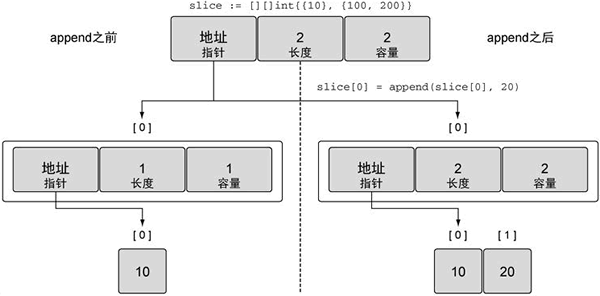
// 声明一个二维整型切片并赋值

slice := [][]int{{10}, {100, 200}}

// 为第一个切片追加值为 20 的元素

slice[0] = append(slice[0], 20)

Go语言里使用 append() 函数处理追加的方式很简明，先增长切片，再将新的整型切片赋值给外层切片的第一个元素，当上面代码中的操作完成后，再将切片复制到外层切片的索引为 0 的元素，如下图所示。

  
图：append 操作之后外层切片索引为 0 的元素的布局

即便是这么简单的多维切片，操作时也会涉及众多的布局和值，在函数间这样传递数据结构会很复杂，不过切片本身结构很简单，可以用很小的成本在函数间传递。

**1.5 map**

Go语言中 map 是一种特殊的[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)，一种元素对（pair）的无序集合，pair 对应一个 key（索引）和一个 value（值），所以这个结构也称为关联数组或字典，这是一种能够快速寻找值的理想结构，给定 key，就可以迅速找到对应的 value。  
  
map 这种数据结构在其他编程语言中也称为字典（[Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank)）、hash 和 HashTable 等。

**1.5.1 map 概念**

map 是引用类型，可以使用如下方式声明：

var mapname map[keytype]valuetype

其中：

* mapname 为 map 的变量名。
* keytype 为键类型。
* valuetype 是键对应的值类型。

提示：[keytype] 和 valuetype 之间允许有空格。

在声明的时候不需要知道 map 的长度，因为 map 是可以动态增长的，未初始化的 map 的值是 nil，使用函数 len() 可以获取 map 中 pair 的数目。  
【示例】

package main

import "fmt"

func main() {

var mapLit map[string]int

//var mapCreated map[string]float32

var mapAssigned map[string]int

mapLit = map[string]int{"one": 1, "two": 2}

mapCreated := make(map[string]float32)

mapAssigned = mapLit

mapCreated["key1"] = 4.5

mapCreated["key2"] = 3.14159

mapAssigned["two"] = 3

fmt.Printf("Map literal at \"one\" is: %d\n", mapLit["one"])

fmt.Printf("Map created at \"key2\" is: %f\n", mapCreated["key2"])

fmt.Printf("Map assigned at \"two\" is: %d\n", mapLit["two"])

fmt.Printf("Map literal at \"ten\" is: %d\n", mapLit["ten"])

}

输出结果：

Map literal at "one" is: 1  
Map created at "key2" is: 3.14159  
Map assigned at "two" is: 3  
Map literal at "ten" is: 0

示例中 mapLit 演示了使用{key1: value1, key2: value2}的格式来初始化 map ，就像数组和结构体一样。  
  
上面代码中的 mapCreated 的创建方式mapCreated := make(map[string]float)等价于mapCreated := map[string]float{} 。  
mapAssigned 是 mapList 的引用，对 mapAssigned 的修改也会影响到 mapLit 的值。  
  
注意：可以使用 make()，但不能使用 new() 来构造 map，如果错误的使用 new() 分配了一个引用对象，会获得一个空引用的指针，相当于声明了一个未初始化的变量并且取了它的地址：

mapCreated := new(map[string]float)

接下来当我们调用mapCreated["key1"] = 4.5的时候，编译器会报错：

invalid operation: mapCreated["key1"] (index of type \*map[string]float).

**1.5.2 map 容量**

和数组不同，map 可以根据新增的 key-value 动态的伸缩，因此它不存在固定长度或者最大限制，但是也可以选择标明 map 的初始容量 capacity，格式如下：

make(map[keytype]valuetype, cap)

例如：

map2 := make(map[string]float, 100)

当 map 增长到容量上限的时候，如果再增加新的 key-value，map 的大小会自动加 1，所以出于性能的考虑，对于大的 map 或者会快速扩张的 map，即使只是大概知道容量，也最好先标明。  
这里有一个 map 的具体例子，即将音阶和对应的音频映射起来：

noteFrequecy := map[string]float32 {

"C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,

"G0": 24.50, "A0": 27.50, "B0": 30.87, "A4": 440}

**1.5.3用切片作为 map 的值**

既然一个 key 只能对应一个 value，而 value 又是一个原始类型，那么如果一个 key 要对应多个值怎么办？例如，当我们要处理 unix 机器上的所有进程，以父进程（pid 为整形）作为 key，所有的子进程（以所有子进程的 pid 组成的切片）作为 value。通过将 value 定义为 []int 类型或者其他类型的切片，就可以优雅的解决这个问题，示例代码如下所示：

mp1 := make(map[int][]int)

mp2 := make(map[int]\*[]int)

**1.5.4 遍历map**

map 的遍历过程使用 for range 循环完成，代码如下：

scene := make(map[string]int)

scene["route"] = 66

scene["brazil"] = 4

scene["china"] = 960

for k, v := range scene {

fmt.Println(k, v)

}

遍历对于Go语言的很多对象来说都是差不多的，直接使用 for range 语法即可，遍历时，可以同时获得键和值，如只遍历值，可以使用下面的形式：

for \_, v := range scene {

将不需要的键使用\_改为匿名变量形式。  
  
只遍历键时，使用下面的形式：

for k := range scene {

无须将值改为匿名变量形式，忽略值即可。

注意：遍历输出元素的顺序与填充顺序无关，不能期望 map 在遍历时返回某种期望顺序的结果。

如果需要特定顺序的遍历结果，正确的做法是先排序，代码如下：

scene := make(map[string]int)

// 准备map数据

scene["route"] = 66

scene["brazil"] = 4

scene["china"] = 960

// 声明一个切片保存map数据

var sceneList []string

// 将map数据遍历复制到切片中

for k := rane scene {

sceneList = append(sceneList, k)

}

// 对切片进行排序

sort.Strings(sceneList)

// 输出

fmt.Println(sceneList)

代码输出如下：

[brazil china route]

代码说明如下：

* 第 1 行，创建一个 map 实例，键为字符串，值为整型。
* 第 4～6 行，将 3 个键值对写入 map 中。
* 第 9 行，声明 sceneList 为字符串切片，以缓冲和排序 map 中的所有元素。
* 第 12 行，将 map 中元素的键遍历出来，并放入切片中。
* 第 17 行，对 sceneList 字符串切片进行排序，排序时，sceneList 会被修改。
* 第 20 行，输出排好序的 map 的键。

sort.Strings 的作用是对传入的字符串切片进行字符串字符的升序排列，排序接口的使用将在后面的章节中介绍。

**1.5.5 map元素的删除和清空**

Go语言提供了一个内置函数 delete()，用于删除容器内的元素，下面我们简单介绍一下如何用 delete() 函数删除 map 内的元素。

#### **1.5.5.1使用 delete() 函数从 map 中删除键值对**

使用 delete() 内建函数从 map 中删除一组键值对，delete() 函数的格式如下：

delete(map, 键)

其中 map 为要删除的 map 实例，键为要删除的 map 中键值对的键。  
  
从 map 中删除一组键值对可以通过下面的代码来完成：

scene := make(map[string]int)

// 准备map数据

scene["roue"] = 66

scene["brazil"] = 4

scene["china"] = 960

delete(scene, "brazil")

for k, v := range scene {

fmt.Println(k, v)

}

代码输出如下：

route 66  
china 960

这个例子中使用 delete() 函数将 brazil 从 scene 这个 map 中删除了。

#### **1.5.5.2清空 map 中的所有元素**

有意思的是，Go语言中并没有为 map 提供任何清空所有元素的函数、方法，清空 map 的唯一办法就是重新 make 一个新的 map，不用担心垃圾回收的效率，Go语言中的并行垃圾回收效率比写一个清空函数要高效的多。

**1.5.5 map的多建索引**