

Golang 中的切片

主讲教师：（大地）

合作网站：www.itying.com （IT 营）

我的专栏：<https://www.itying.com/category-79-b0.html>

1、为什么要使用切片.....	1
2、切片的定义.....	2
3、关于 nil 的认识.....	3
4、切片的循环遍历.....	3
5、基于数组定义切片.....	4
6、切片再切片.....	4
7、关于切片的长度和容量.....	5
8、切片的本质.....	6
9、使用 make()函数构造切片.....	7
10、切片不能直接比较.....	7
11、切片是引用数据类型--注意切片的赋值拷贝.....	8
12、append()方法为切片添加元素.....	8
13、切片的扩容策略.....	10
14、使用 copy()函数复制切片.....	11
15、从切片中删除元素.....	12
17、练习题.....	12

1、为什么要使用切片

因为数组的长度是固定的并且数组长度属于类型的一部分，所以数组有很多的局限性。 例如：

```
package main
func arraySum(x [4]int) int {
    sum := 0
```

```

    for _, v := range x {
        sum = sum + v
    }
    return sum
}

func main() {
    a := [4]int{1, 2, 3, 4}
    println(arraySum(a))

    b := [5]int{1, 2, 3, 4, 5}
    println(arraySum(b)) //错误
}

```

这个求和函数只能接受[4]int 类型，其他的都不支持。所以传入长度为 5 的数组的时候就会报错。

2、切片的定义

切片 (Slice) 是一个拥有相同类型元素的可变长度的序列。它是基于数组类型做的一层封装。它非常灵活，支持自动扩容。

切片是一个引用类型，它的内部结构包含地址、长度和容量。

声明切片类型的基本语法如下：

```
var name []T
```

其中：

1. name:表示变量名
2. T:表示切片中的元素类型

举个例子：

```

package main
import "fmt"
func main() {
    // 声明切片类型
    var a []string           //声明一个字符串切片
    var b = []int{}          //声明一个整型切片并初始化
    var c = []bool{false, true} //声明一个布尔切片并初始化
    var d = []bool{false, true} //声明一个布尔切片并初始化
}

```

```
fmt.Println(a)           //[]
fmt.Println(b)           //[]
fmt.Println(c)           //[false true]
fmt.Println(a == nil)    //true
fmt.Println(b == nil)    //false
fmt.Println(c == nil)    //false
fmt.Println(c == d)      //切片是引用类型，不支持直接比较，只能和 nil 比较
}
```

3、关于 nil 的认识

当你声明了一个变量，但却还没有赋值时，golang 中会自动给你的变量赋值一个默认零值。这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
```

4、切片的循环遍历

切片的循环遍历和数组的循环遍历是一样的

```
var a = []string{"北京", "上海", "深圳"}
// 方法 1: for 循环遍历
for i := 0; i < len(a); i++ {
    fmt.Println(a[i])
}
```

```
// 方法 2: for range 遍历
for index, value := range a {
    fmt.Println(index, value)
}
```

5、基于数组定义切片

由于切片的底层就是一个数组，所以我们可以基于数组定义切片。

```
func main() {
    // 基于数组定义切片
    a := [5]int{55, 56, 57, 58, 59}
    b := a[1:4]                //基于数组 a 创建切片，包括元素 a[1],a[2],a[3]
    fmt.Println(b)             //[56 57 58]
    fmt.Printf("type of b:%T\n", b) //type of b:[]int
}
还支持如下方式:
c := a[1:] //[56 57 58 59]
d := a[:4] //[55 56 57 58]
e := a[:]  //[55 56 57 58 59]
```

6、切片再切片

除了基于数组得到切片，我们还可以通过切片来得到切片。

```
func main() {
    //切片再切片
    a := [...]string{"北京", "上海", "广州", "深圳", "成都", "重庆"}
    fmt.Printf("a:%v type:%T len:%d cap:%d\n", a, a, len(a), cap(a))
    b := a[1:3]
    fmt.Printf("b:%v type:%T len:%d cap:%d\n", b, b, len(b), cap(b))
    c := b[1:5]
    fmt.Printf("c:%v type:%T len:%d cap:%d\n", c, c, len(c), cap(c))
}
输出:
a:[北京 上海 广州 深圳 成都 重庆] type:[6]string len:6 cap:6
b:[上海 广州] type:[]string len:2 cap:5
c:[广州 深圳 成都 重庆] type:[]string len:4 cap:4
```

注意： 对切片进行再切片时，索引不能超过原数组的长度，否则会出现索引越界的错误。

7、关于切片的长度和容量

切片拥有自己的长度和容量，我们可以通过使用内置的 `len()` 函数求长度，使用内置的 `cap()` 函数求切片的容量。

切片的长度就是它所包含的元素个数。

切片的容量是从它的第一个元素开始数，到其底层数组元素末尾的个数。

切片 `s` 的长度和容量可通过表达式 `len(s)` 和 `cap(s)` 来获取。

```
s := []int{2, 3, 5, 7, 11, 13}

fmt.Println(s)

fmt.Printf("长度:%v 容量 %v\n", len(s), cap(s))

c := s[:2]
fmt.Println(c)

fmt.Printf("长度:%v 容量 %v\n", len(c), cap(c))

d := s[1:3]
fmt.Println(d)
fmt.Printf("长度:%v 容量 %v", len(d), cap(d))
```

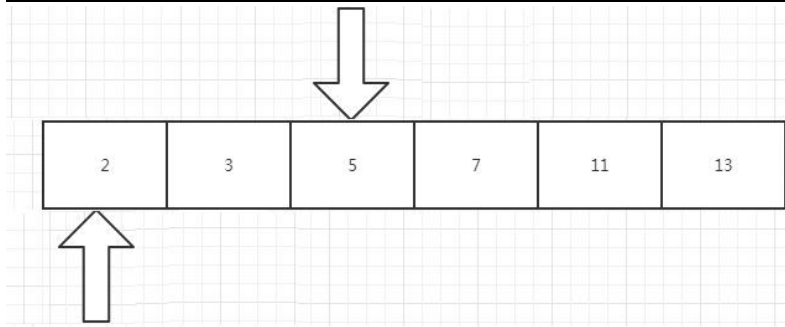
输出：

```
D:\golang\src\demo01>go run main.go
[2 3 5 7 11 13]
长度:6 容量 6
[2 3]
长度:2 容量 6
[3 5]
长度:2 容量 5
```

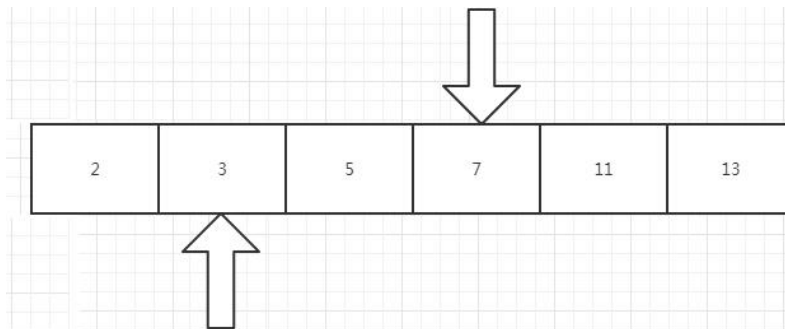
1、第一个输出为[2,3,5,7,11,13]，长度为6，容量为6

2	3	5	7	11	13
---	---	---	---	----	----

3、`c:=s[:2]`后输出：[2 3]，左指针 `s[0]`，右指针 `s[2]`，所以长度为2，容量为6



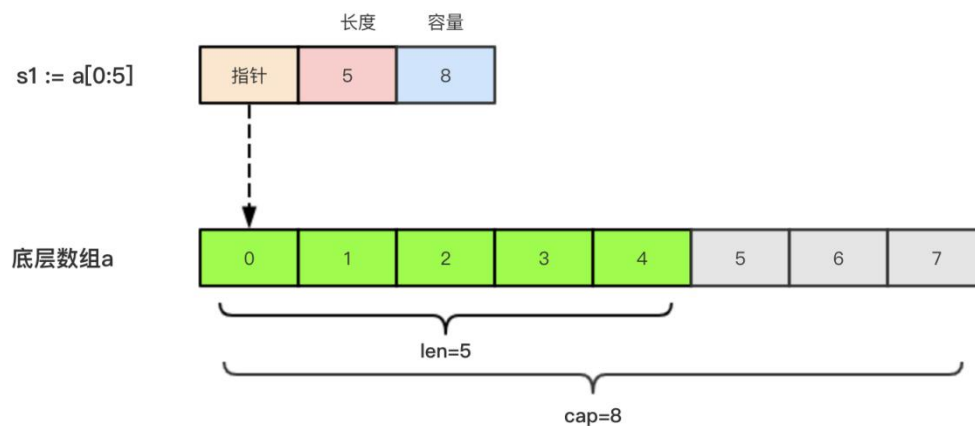
4、`d := s[1:3]`后输出：[3 5]，左指针 `s[1]`，右指针 `s[3]`，所以长度为 2，容量为 5



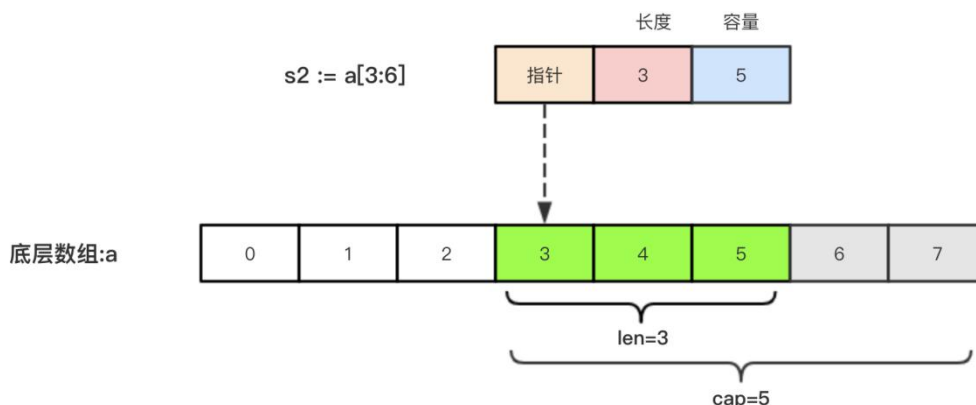
8、切片的本质

切片的本质就是对底层数组的封装，它包含了三个信息：底层数组的指针、切片的长度（`len`）和切片的容量（`cap`）。

举个例子，现在有一个数组 `a := [8]int{0, 1, 2, 3, 4, 5, 6, 7}`，切片 `s1 := a[:5]`，相应示意图如下。



切片 `s2 := a[3:6]`，相应示意图如下：



9、使用 make()函数构造切片

我们上面都是基于数组来创建的切片，如果需要动态的创建一个切片，我们就需要使用内置的 `make()` 函数，格式如下：

```
make([]T, size, cap)
```

其中：

3. T:切片的元素类型
4. size:切片中元素的数量
5. cap:切片的容量

举个例子：

```
func main() {
    a := make([]int, 2, 10)
    fmt.Println(a)      //[0 0]
    fmt.Println(len(a)) //2
    fmt.Println(cap(a)) //10
}
```

上面代码中 `a` 的内部存储空间已经分配了 10 个，但实际上只用了 2 个。容量并不会影响当前元素的个数，所以 `len(a)` 返回 2，`cap(a)` 则返回该切片的容量。

10、切片不能直接比较

切片之间是不能比较的，我们不能使用 `==` 操作符来判断两个切片是否含有全部相等元素。切片唯一合法的比较操作是和 `nil` 比较。一个 `nil` 值的切片并没有底层数组，一个 `nil` 值的切片

的长度和容量都是 0。但是我们不能说一个长度和容量都是 0 的切片一定是 nil，例如下面的示例：

```
var s1 []int //len(s1)=0;cap(s1)=0;s1==nil
s2 := []int{} //len(s2)=0;cap(s2)=0;s2!=nil
s3 := make([]int, 0) //len(s3)=0;cap(s3)=0;s3!=nil
```

所以要判断一个切片是否是空的，要是用 `len(s) == 0` 来判断，不应该使用 `s == nil` 来判断。

11、切片是引用数据类型--注意切片的赋值拷贝

下面的代码中演示了拷贝前后两个变量共享底层数组，对一个切片的修改会影响另一个切片的内容，这点需要特别注意。

```
func main() {
    s1 := make([]int, 3) //[0 0 0]

    s2 := s1 //将 s1 直接赋值给 s2，s1 和 s2 共用一个底层数组

    s2[0] = 100

    fmt.Println(s1) //[100 0 0]

    fmt.Println(s2) //[100 0 0]
}
```

12、append()方法为切片添加元素

Go 语言的内建函数 `append()` 可以为切片动态添加元素，每个切片会指向一个底层数组，这个数组的容量够用就添加新增元素。当底层数组不能容纳新增的元素时，切片就会自动按照一定的策略进行“扩容”，此时该切片指向的底层数组就会更换。“扩容”操作往往发生在 `append()` 函数调用时，所以我们通常都需要用原变量接收 `append` 函数的返回值。

给切片追加元素的错误写法：

```
s3 := []int{1, 2, 3, 5, 6, 7}
s3[6] = 8
fmt.Println(s3) //index out of range [6] with length 6
```


append()方法为切片追加元素:

```
func main() {  
    //append()添加元素和切片扩容  
    var numSlice []int  
    for i := 0; i < 10; i++ {  
        numSlice = append(numSlice, i)  
        fmt.Printf("%v len:%d cap:%d ptr:%p\n", numSlice, len(numSlice), cap(numSlice), numSlice)  
    }  
}
```

输出:

```
[0] len:1 cap:1 ptr:0xc0000a8000  
[0 1] len:2 cap:2 ptr:0xc0000a8040  
[0 1 2] len:3 cap:4 ptr:0xc0000b2020  
[0 1 2 3] len:4 cap:4 ptr:0xc0000b2020  
[0 1 2 3 4] len:5 cap:8 ptr:0xc0000b6000  
[0 1 2 3 4 5] len:6 cap:8 ptr:0xc0000b6000  
[0 1 2 3 4 5 6] len:7 cap:8 ptr:0xc0000b6000  
[0 1 2 3 4 5 6 7] len:8 cap:8 ptr:0xc0000b6000  
[0 1 2 3 4 5 6 7 8] len:9 cap:16 ptr:0xc0000b8000  
[0 1 2 3 4 5 6 7 8 9] len:10 cap:16 ptr:0xc0000b8000
```

从上面的结果可以看出:

1. **append()**函数将元素追加到切片的最后并返回该切片。
2. 切片 **numSlice** 的容量按照 1, 2, 4, 8, 16 这样的规则自动进行扩容, 每次扩容后都是扩容前的 2 倍。

append()函数还支持一次性追加多个元素。 例如:

```
var citySlice []string  
// 追加一个元素  
citySlice = append(citySlice, "北京")  
// 追加多个元素  
citySlice = append(citySlice, "上海", "广州", "深圳")  
// 追加切片  
a := []string{"成都", "重庆"}  
citySlice = append(citySlice, a...)
```

```
fmt.Println(citySlice) //[北京 上海 广州 深圳 成都 重庆]
```

切片的追加切片

```
s1 := []int{100, 200, 300}
s2 := []int{400, 500, 600}
s3 := append(s1, s2...)
fmt.Println(s3)
```

13、切片的扩容策略

可以通过查看\$GOROOT/src/runtime/slice.go 源码，其中扩容相关代码如下：

```
newcap := old.cap
doublecap := newcap + newcap
if cap > doublecap {
    newcap = cap
} else {
    if old.len < 1024 {
        newcap = doublecap
    } else {
        // Check 0 < newcap to detect overflow
        // and prevent an infinite loop.
        for 0 < newcap && newcap < cap {
            newcap += newcap / 4
        }
        // Set newcap to the requested cap when
        // the newcap calculation overflowed.
        if newcap <= 0 {
            newcap = cap
        }
    }
}
```

从上面的代码可以看出以下内容：

1、首先判断，如果新申请容量（cap）大于 2 倍的旧容量（old.cap），最终容量（newcap）就是新申请的容量（cap）。

- 2、否则判断，如果旧切片的长度小于 1024，则最终容量(newcap)就是旧容量(old.cap)的两倍，即 (newcap=doublecap)，
- 3、否则判断，如果旧切片长度大于等于 1024，则最终容量 (newcap) 从旧容量 (old.cap) 开始循环增加原来的 1/4，即 (newcap=old.cap, for {newcap += newcap/4}) 直到最终容量 (newcap) 大于等于新申请的容量(cap)，即 (newcap >= cap)
- 4、如果最终容量 (cap) 计算值溢出，则最终容量 (cap) 就是新申请容量 (cap)。

需要注意的是，切片扩容还会根据切片中元素的类型不同而做不同的处理，比如 int 和 string 类型的处理方式就不一样。

14、使用 copy()函数复制切片

首先我们来看一个问题：

```
func main() {
    a := []int{1, 2, 3, 4, 5}
    b := a
    fmt.Println(a) //[1 2 3 4 5]
    fmt.Println(b) //[1 2 3 4 5]
    b[0] = 1000
    fmt.Println(a) //[1000 2 3 4 5]
    fmt.Println(b) //[1000 2 3 4 5]
}
```

由于切片是引用类型，所以 a 和 b 其实都指向了同一块内存地址。修改 b 的同时 a 的值也会发生变化。

Go 语言内建的 copy()函数可以迅速地将一个切片的数据复制到另外一个切片空间中，copy()函数的使用格式如下：

```
copy(destSlice, srcSlice []T)
```

其中：

- srcSlice: 数据来源切片
- destSlice: 目标切片

举个例子：

```
func main() {
    // copy()复制切片
    a := []int{1, 2, 3, 4, 5}
```

```
c := make([]int, 5, 5)
copy(c, a)    //使用 copy()函数将切片 a 中的元素复制到切片 c
fmt.Println(a) //[1 2 3 4 5]
fmt.Println(c) //[1 2 3 4 5]
c[0] = 1000
fmt.Println(a) //[1 2 3 4 5]
fmt.Println(c) //[1000 2 3 4 5]
}
```

15、从切片中删除元素

Go 语言中并没有删除切片元素的专用方法，我们可以使用切片本身的特性来删除元素。代码如下：

```
func main() {
    // 从切片中删除元素
    a := []int{30, 31, 32, 33, 34, 35, 36, 37}
    // 要删除索引为 2 的元素
    a = append(a[:2], a[3:]...)
    fmt.Println(a) //[30 31 33 34 35 36 37]
}
```

总结一下就是：要从切片 a 中删除索引为 index 的元素，操作方法是 `a = append(a[:index], a[index+1:]...)`

17、练习题

1.请写出下面代码的输出结果。

```
func main() {
    var a = make([]string, 5, 10)
    for i := 0; i < 12; i++ {
        a = append(a, fmt.Sprintf("%v", i))
    }
    fmt.Println(a)
}
```

2.请使用内置的 sort 包对数组 `var a = [...]int{3, 7, 8, 9, 1}`进行排序（ ）。

