Go 语言切片(Slice)

Go 语言切片是对数组的抽象，是一个拥有相同类型元素的可变长度的序列。

Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go 中提供了一种灵活，功能强悍的内置类型切片("动态数组")，与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

可以把切片理解为，可变长度的数组，其实它底层就是使用数组实现的，增加了**自动扩容**功能。

切片是可索引的，并且可以由 `len()` 函数获取长度。

切片提供了计算容量的函数 `cap()` 可以测量切片最长可以达到多少：它等于切片的长度 + 数组除切片之外的长度。如果 s 是一个切片，`cap(s)` 就是从 `s[0]` 到数组末尾的数组长度。切片的长度永远不会超过它的容量，所以对于 切片 s 来说该不等式永远成立：`0 <= len(s) <= cap(s)`。

多个切片如果表示同一个数组的片段，它们可以共享数据；因此一个切片和相关数组的其他切片是共享存储的，相反，不同的数组总是代表不同的存储。数组实际上是切片的构建块。

优点 因为切片是引用，所以它们不需要使用额外的内存并且比使用数组更有效率，所以在 Go 代码中 切片比数组更常用。

定义切片

你可以声明一个未指定大小的数组来定义切片：

```
var identifier []type
```

`identifier`：切片名称

`type`：切片保存元素的类型

切片不需要说明长度。

或使用 **make()** 函数来创建切片:

```
var slice1 []type = make([]type, len)

也可以简写为

slice1 := make([]type, len)
```

也可以指定容量，其中 **capacity** 为可选参数。

```
make([]T, length, capacity)
```

这里 len 是数组的长度并且也是切片的初始长度。

示例：

```go
package main

import "fmt"

func main() {
   //1.声明切片
   var s1 []int
   // 2.:=
   s2 := []int{}
   // 3.make()
   var s3 []int = make([]int, 0)
   fmt.Println(s1, s2, s3)
}
```

1. 切片：切片是数组的一个引用，因此切片是引用类型。但自身是结构体，值拷贝传递。
2. 切片的长度可以改变，因此，切片是一个可变的数组。
3. 切片遍历方式和数组一样，可以用len()求长度。表示可用元素数量，读写操作不能超过该限制。
4. cap可以求出slice最大扩张容量，不能超出数组限制。0 <= len(slice) <= len(array)，其中array是slice引用的数组。
5. 切片的定义：var 变量名 []类型，比如 var str []string var arr []int。
6. 如果 slice == nil，那么 len、cap 结果都等于 0。
7. **注意** 绝对不要用指针指向 slice。切片本身已经是一个引用类型，所以它本身就是一个指针!!

切片初始化

```go
s :=[] int {1,2,3 } //[]表示是切片类型，{1,2,3} 初始化值依次是 1,2,3
```

直接初始化切片，**[]** 表示是切片类型，**{1,2,3}** 初始化值依次是 **1,2,3**，其 **cap=len=3。**

```go
s := arr[:]
```

初始化切片 **s**，是数组 arr 的引用。

```go
s := arr[startIndex:endIndex]
```

将 arr 中从下标 startIndex 到 endIndex-1 下的元素创建为一个新的切片。

```go
s := arr[startIndex:]
```

默认 endIndex 时将表示一直到arr的最后一个元素。

```go
s := arr[:endIndex]
```

默认 startIndex 时将表示从 arr 的第一个元素开始。

```go
s1 := s[startIndex:endIndex]
```

通过切片 s 初始化切片 s1。

```go
s :=make([]int,len,cap)
```

通过内置函数 **make()** 初始化切片**s**，**[]int** 标识为其元素类型为 int 的切片。

示例：

```
package main

import "fmt"

func main() {
    //1.初始化赋值
    s1 := []int{1, 2, 3}
    fmt.Printf("s1: %v\n", s1)
    //2.make初始化赋值
    var s2 []int = make([]int, 10, 10)
    fmt.Printf("s2: %v\n", s2)
    for i := 0; i < len(s2); i++ {
        s2[i] = 5 * i
    }
    fmt.Printf("s2: %v\n", s2)
    // 3.从数组切片
    arr := [5]int{1, 2, 3, 4, 5}
    var s3 []int
    // 前包后不包
    s3 = arr[1:4]
    fmt.Printf("s3: %v\n", s3)
}
#结果
s1: [1 2 3]
s2: [0 0 0 0 0 0 0 0 0 0]
s2: [0 5 10 15 20 25 30 35 40 45]
s3: [2 3 4]
```

len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

```
len 长度是当前元素个数 cap 长度是底层元素个数
```

```
s := []int{2, 3, 5, 7, 11, 13}  printSlice(s) len: 6 cap:6 正确
```

```
s = s[:0]   len: 0 cap: 6 目前就只有0个元素，底层数组还是6个 printSlice(s)
```

```
s = s[:4]   printSlice(s) len: 4 cap: 6 目前是4个元素，底层还是6个
```

```
s = s[2:]   printSlice(s) len: 2 cap: 4 目前是2个元素，底层元素是从第一个元素往后数的，目前是4
```

- len 计算的是当前数组元素的个数，
- cap计算的是当前**切片开始位**到数组最后一个元素的个数

cap 限制

超出原 slice.cap 限制，就会重新分配底层数组，即便原数组并未填满。

```
package main

import (
    "fmt"
)

func main() {
    data := [...]int{0, 1, 2, 3, 4, 10: 0}
    s := data[:2:3]
    fmt.Printf("len=%d cap=%d slice=%v\n", len(s), cap(s), s)

    s = append(s, 100, 200) // 一次 append 两个值，超出 s.cap 限制。
    fmt.Printf("len=%d cap=%d slice=%v\n", len(s), cap(s), s)

    fmt.Println(&s[0], &data[0]) // 比对底层数组起始指针。
}
#结果
len=2 cap=3 slice=[0 1]
len=4 cap=6 slice=[0 1 100 200]
0xc000144030 0xc000102120
```

从输出结果可以看出，append 后的 s 重新分配了底层数组，并复制数据。如果只追加一个值，则不会超过 s.cap 限制，也就不会重新分配。 通常以 2 倍容量重新分配底层数组。在大批量添加数据时，建议一次性分配足够大的空间，以减少内存分配和数据复制开销。或初始化足够长的 len 属性，改用索引号进行操作。及时释放不再使用的 slice 对象，避免持有过期数组，造成 GC 无法回收。

空(nil)切片

一个切片在未初始化之前默认为 nil，长度为 0。

示例：

```
package main

import "fmt"

func main() {
    var numbers []int
    if numbers == nil {
        fmt.Printf("切片是空的\n")
    }
    fmt.Printf("len=%d cap=%d slice=%v\n", len(numbers), cap(numbers), numbers)
}
#结果
切片是空的
len=0 cap=0 slice=[]
```

切片的遍历

切片的遍历和数组的遍历非常类似，可以使用for循环索引遍历，或者for range循环。

```
package main

import "fmt"

func main() {
    s := []int{1, 2, 3, 4, 5}
    //1.for循环
    for i := 0; i < len(s); i++ {
        fmt.Printf("s[%d]: %v\n", i, s[i])
    }
    //2.for range循环
    for k, v := range s {
        fmt.Printf("s[%d]: %v\n", k, v)
    }
}
#结果都是
s[0]: 1
s[1]: 2
s[2]: 3
s[3]: 4
s[4]: 5
```

切片截取

可以通过设置下限及上限来设置截取切片 *[lower-bound:upper-bound]*，实例如下：

```
package main

import "fmt"

func main() {
   /* 创建切片 */
   numbers := []int{0,1,2,3,4,5,6,7,8}
   printSlice(numbers)

   /* 打印原始切片 */
   fmt.Println("numbers ==", numbers)

   /* 打印子切片从索引1(包含) 到索引4(不包含)*/
   fmt.Println("numbers[1:4] ==", numbers[1:4])

   /* 默认下限为 0*/
   fmt.Println("numbers[:3] ==", numbers[:3])

   /* 默认上限为 len(s)*/
   fmt.Println("numbers[4:] ==", numbers[4:])

   numbers1 := make([]int,0,5)
   printSlice(numbers1)

   /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
   number2 := numbers[:2]
   printSlice(number2)
```

```
   /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
   number3 := numbers[2:5]
   printSlice(number3)

}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
#结果
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
```

切片元素的添加append()、删除、拷贝copy()

切片是一个动态数组，可以使用`append()`函数添加元素，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。由于，切片是引用类型，通过赋值的方式，会修改原有内容，go提供了`copy()`函数来拷贝切片

添加元素append()

```
package main

import "fmt"

func main() {
    var numbers []int
    printSlice(numbers)

    /* 允许追加空切片 */
    numbers = append(numbers, 0)
    printSlice(numbers)

    /* 向切片添加一个元素 */
    numbers = append(numbers, 1)
    printSlice(numbers)

    /* 同时添加多个元素 */
    numbers = append(numbers, 2, 3, 4)
    printSlice(numbers)

    /* 添加另外一个切片 */
    var a = []int{1, 2, 3}
    var b = []int{4, 5, 6}
    c := append(a, b...)
    printSlice(c)
}
```

```
func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
#结果
len=0 cap=0 slice=[]
len=1 cap=1 slice=[0]
len=2 cap=2 slice=[0 1]
len=5 cap=6 slice=[0 1 2 3 4]
len=6 cap=6 slice=[1 2 3 4 5 6]
```

append： 向 slice 尾部添加数据，返回新的 slice 对象。

```
package main

import (
    "fmt"
)

func main() {
    s1 := make([]int, 0, 5)
    fmt.Printf("%p\n", &s1)

    s2 := append(s1, 1)
    fmt.Printf("%p\n", &s2)

    fmt.Println(s1, s2)
}
#结果
0xc42000a060
0xc42000a080
[] [1]
```

拷贝copy()

```
package main

import (
    "fmt"
)

func main() {
    s1 := []int{1, 2, 3, 4, 5}
    fmt.Printf("slice s1 : %v\n", s1)
    s2 := make([]int, 10)
    fmt.Printf("slice s2 : %v\n", s2)
    copy(s2, s1)
    fmt.Printf("copied slice s1 : %v\n", s1)
    fmt.Printf("copied slice s2 : %v\n", s2)
    s3 := []int{1, 2, 3}
    fmt.Printf("slice s3 : %v\n", s3)
    s3 = append(s3, s2...)
    fmt.Printf("appended slice s3 : %v\n", s3)
    s3 = append(s3, 4, 5, 6)
    fmt.Printf("last slice s3 : %v\n", s3)
```

```
}
#结果:
slice s1 : [1 2 3 4 5]
slice s2 : [0 0 0 0 0 0 0 0 0 0]
copied slice s1 : [1 2 3 4 5]
copied slice s2 : [1 2 3 4 5 0 0 0 0 0]
slice s3 : [1 2 3]
appended slice s3 : [1 2 3 1 2 3 4 5 0 0 0 0 0]
last slice s3 : [1 2 3 1 2 3 4 5 0 0 0 0 0 4 5 6]
```

copy： 函数 copy 在两个 slice 间复制数据，复制长度以 len 小的为准。两个 slice 可指向同一底层数组，允许元素区间重叠。

```go
package main

import (
    "fmt"
)

func main() {
    data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    fmt.Println("array data : ", data)
    s1 := data[8:]
    s2 := data[:5]
    fmt.Printf("slice s1 : %v\n", s1)
    fmt.Printf("slice s2 : %v\n", s2)
    copy(s2, s1)
    fmt.Printf("copied slice s1 : %v\n", s1)
    fmt.Printf("copied slice s2 : %v\n", s2)
    fmt.Println("last array data : ", data)
}
#结果
array data :  [0 1 2 3 4 5 6 7 8 9]
slice s1 : [8 9]
slice s2 : [0 1 2 3 4]
copied slice s1 : [8 9]
copied slice s2 : [8 9 2 3 4]
last array data :  [8 9 2 3 4 5 6 7 8 9]
```

应及时将所需数据 copy 到较小的 slice，以便释放超大号底层数组内存。

删除元素

Go 并没有提供删除切片元素专用的语法或函数，需要使用切片本身的特性来删除元素。

截取法

利用对`slice`的截取删除指定元素。注意删除时，后面的元素会前移，所以下标 i 应该左移一位。

```
package main

import "fmt"

func main() {
    //删除3这个元素
    numbers := []int{1, 2, 3, 4, 5}
    numbers = append(numbers[:2], numbers[2+1:]...)

    fmt.Printf("len=%d cap=%d slice=%v\n", len(numbers), cap(numbers), numbers)
}
#结果
len=4 cap=5 slice=[1 2 4 5]
```

拷贝法

```
package main

import "fmt"

func main() {
    //删除3这个元素
    numbers := []int{1, 2, 3, 4, 5}
    tmp := make([]int, 0, len(numbers))
    for _, v := range numbers {
        if v != 3 {
            tmp = append(tmp, v)
        }
    }

    fmt.Printf("len=%d cap=%d slice=%v\n", len(numbers), cap(numbers), numbers)
    fmt.Printf("len=%d cap=%d slice=%v\n", len(tmp), cap(tmp), tmp)
}
#结果
len=5 cap=5 slice=[1 2 3 4 5]
len=4 cap=5 slice=[1 2 4 5]
```

将切片传递给函数

如果你有一个函数需要对数组做操作，你可能总是需要把参数声明为切片。当你调用该函数时，把数组分片，创建为一个 切片引用并传递给该函数。

示例：计算数组元素和的方法

```
package main

import "fmt"

func sum(a []int) int {
    s := 0
    for i := 0; i < len(a); i++ {
        s += a[i]
    }
    return s
```

```
}

func main() {
    //定义数组
    var arr = [5]int{0, 1, 2, 3, 4}
    //调用函数并将数组切片
    r := sum(arr[:])
    fmt.Printf("r: %v\n", r)
}
#结果
r: 10
```

多维切片

和数组一样，切片通常也是一维的，但是也可以由一维组合成高维。通过分片的分片（或者切片的数组），长度可以任意动态变化，所以 Go 语言的多维切片可以任意切分。而且，内层的切片必须单独分配（通过 make 函数）。

多维切片定义

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

`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[ ]。

示例：二维切片

```
//声明一个二维切片
var slice [][]int
//为二维切片赋值
slice = [][]int{{10}, {100, 200}}
//或者
// 声明一个二维整型切片并赋值
slice := [][]int{{10}, {100, 200}}
```

```
package main

import "fmt"

func main() {
    // 声明一个二维整型切片并赋值
    slice := [][]int{{10}, {100, 200}}
    fmt.Printf("len=%d cap=%d slice=%v\n", len(slice), cap(slice), slice)
    // 为第一个切片追加值为 20 的元素
    slice[0] = append(slice[0], 20)
    fmt.Printf("len=%d cap=%d slice=%v\n", len(slice), cap(slice), slice)
}
#结果
len=2 cap=2 slice=[[10] [100 200]]
len=2 cap=2 slice=[[10 20] [100 200]]
```