**GO语言数组和切片实例详解**

**一、数组**

与其他大多数语言类似，Go语言的数组也是一个元素类型相同的定长的序列。

**（1）数组的创建。**

数组有3种创建方式：[length]Type　、[N]Type{value1, value2, ... , valueN}、[...]Type{value1, value2, ... , valueN}　如下：

复制代码代码如下:

func test5() {  
    var iarray1 [5]int32  
    var iarray2 [5]int32 = [5]int32{1, 2, 3, 4, 5}  
    iarray3 := [5]int32{1, 2, 3, 4, 5}  
    iarray4 := [5]int32{6, 7, 8, 9, 10}  
    iarray5 := [...]int32{11, 12, 13, 14, 15}  
    iarray6 := [4][4]int32{{1}, {1, 2}, {1, 2, 3}}  
    fmt.Println(iarray1)  
    fmt.Println(iarray2)  
    fmt.Println(iarray3)  
    fmt.Println(iarray4)  
    fmt.Println(iarray5)  
    fmt.Println(iarray6)  
}

结果：

[0 0 0 0 0]  
[1 2 3 4 5]  
[1 2 3 4 5]  
[6 7 8 9 10]  
[11 12 13 14 15]  
[[1 0 0 0] [1 2 0 0] [1 2 3 0] [0 0 0 0]]

我们看数组 iarray1，只声明，并未赋值，Go语言帮我们自动赋值为0。再看 iarray2 和 iarray3 ，我们可以看到，Go语言的声明，可以表明类型，也可以不表明类型，var iarray3 = [5]int32{1, 2, 3, 4, 5} 也是完全没问题的。

**（2）数组的容量和长度是一样的。cap() 函数和 len() 函数均输出数组的容量（即长度）。**如：

复制代码代码如下:

func test6() {  
    iarray4 := [5]int32{6, 7, 8, 9, 10}  
    fmt.Println(len(iarray4))  
    fmt.Println(cap(iarray4))  
}

输出都是5。

**（3）使用：**

复制代码代码如下:

func test7() {  
    iarray7 := [5]string{"aaa", `bb`, "可以啦", "叫我说什么好", "()"}  
    fmt.Println(iarray7)  
    for i := range iarray7 {  
        fmt.Println(iarray7[i])  
    }  
}

**二、切片**

Go语言中，切片是长度可变、容量固定的相同的元素序列。Go语言的切片本质是一个数组。容量固定是因为数组的长度是固定的，切片的容量即隐藏数组的长度。长度可变指的是在数组长度的范围内可变。

**（1）切片的创建。**

切片的创建有4种方式：

1）make ( []Type ,length, capacity )

2)  make ( []Type, length)

3) []Type{}

4) []Type{value1 , value2 , ... , valueN }

从3)、4)可见，创建切片跟创建数组唯一的区别在于 Type 前的“ [] ”中是否有数字，为空，则代表切片，否则则代表数组。因为切片是长度可变的。如下是创建切片的示例：

复制代码代码如下:

func test8() {  
    slice1 := make([]int32, 5, 8)  
    slice2 := make([]int32, 9)  
    slice3 := []int32{}  
    slice4 := []int32{1, 2, 3, 4, 5}  
    fmt.Println(slice1)  
    fmt.Println(slice2)  
    fmt.Println(slice3)  
    fmt.Println(slice4)  
}

输出为：

[0 0 0 0 0]  
[0 0 0 0 0 0 0 0 0]  
[]  
[1 2 3 4 5]

如上，创造了4个切片，3个空切片，一个有值的切片。

**（2）切片与隐藏数组：**

一个切片是一个隐藏数组的引用，并且对于该切片的切片也引用同一个数组。如下示例，创建了一个切片 slice0，并根据这个切片创建了2个切片 slice1 和 slice2：

复制代码代码如下:

func test9() {  
    slice0 := []string{"a", "b", "c", "d", "e"}  
    slice1 := slice0[2 : len(slice0)-1]  
    slice2 := slice0[:3]  
    fmt.Println(slice0, slice1, slice2)  
    slice2[2] = "8"  
    fmt.Println(slice0, slice1, slice2)  
}

输出为：

[a b c d e] [c d] [a b c]  
[a b 8 d e] [8 d] [a b 8]  
可见，切片slice0 、 slice1 和 slice2是同一个底层数组的引用，所以slice2改变了，其他两个都会变。

**（3）遍历、修改切片：**

复制代码代码如下:

func test10() {  
    slice0 := []string{"a", "b", "c", "d", "e"}  
    fmt.Println("\n~~~~~~元素遍历~~~~~~")  
    for \_, ele := range slice0 {  
        fmt.Print(ele, " ")  
        ele = "7"  
    }  
    fmt.Println("\n~~~~~~索引遍历~~~~~~")  
    for index := range slice0 {  
        fmt.Print(slice0[index], " ")  
    }  
    fmt.Println("\n~~~~~~元素索引共同使用~~~~~~")  
    for index, ele := range slice0 {  
        fmt.Print(ele, slice0[index], " ")  
    }  
    fmt.Println("\n~~~~~~修改~~~~~~")  
    for index := range slice0 {  
        slice0[index] = "9"  
    }  
    fmt.Println(slice0)  
}

如上，前三种循环使用了不同的for range循环，当for后面，range前面有2个元素时，第一个元素代表索引，第二个元素代表元素值，使用 “\_” 则表示忽略，因为go语言中，未使用的值会导致编译错误。

只有一个元素时，该元素代表索引。

只有用索引才能修改元素。如在第一个遍历中，赋值ele为7，结果没有作用。因为在元素遍历中，ele是值传递，ele是该切片元素的副本，修改它不会影响原本值，而在第四个遍历——索引遍历中，修改的是该切片元素引用的值，所以可以修改。

结果为：

~~~~~~元素遍历~~~~~~  
a b c d e   
~~~~~~索引遍历~~~~~~  
a b c d e   
~~~~~~元素索引共同使用~~~~~~  
aa bb cc dd ee   
~~~~~~修改~~~~~~  
[9 9 9 9 9]

**（4）、追加、复制切片：**

复制代码代码如下:

func test11() {  
    slice := []int32{}  
    fmt.Printf("slice的长度为：%d,slice为：%v\n", len(slice), slice)  
    slice = append(slice, 12, 11, 10, 9)  
    fmt.Printf("追加后，slice的长度为：%d,slice为：%v\n", len(slice), slice)  
    slicecp := make([]int32, (len(slice)))  
    fmt.Printf("slicecp的长度为：%d,slicecp为：%v\n", len(slicecp), slicecp)  
    copy(slicecp, slice)  
    fmt.Printf("复制赋值后，slicecp的长度为：%d,slicecp为：%v\n", len(slicecp), slicecp)  
}

追加、复制切片，用的是内置函数append和copy，copy函数返回的是最后所复制的元素的数量。

**（5）、内置函数append**

内置函数append可以向一个切片后追加一个或多个同类型的其他值。如果追加的元素数量超过了原切片容量，那么最后返回的是一个全新数组中的全新切片。如果没有超过，那么最后返回的是原数组中的全新切片。无论如何，append对原切片无任何影响。如下示例：

复制代码代码如下:

func test12() {  
    slice := []int32{1, 2, 3, 4, 5, 6}  
    slice2 := slice[:2]  
    \_ = append(slice2, 50, 60, 70, 80, 90)  
    fmt.Printf("slice为：%v\n", slice)  
    fmt.Printf("操作的切片：%v\n", slice2)  
    \_ = append(slice2, 50, 60)  
    fmt.Printf("slice为：%v\n", slice)  
    fmt.Printf("操作的切片：%v\n", slice2)  
}

如上，append方法用了2次，结果返回的结果完全不同，原因是第二次append方法追加的元素数量没有超过 slice 的容量。而无论怎样，原切片slice2都无影响。结果：

slice为：[1 2 3 4 5 6]  
操作的切片：[1 2]  
slice为：[1 2 50 60 5 6]  
操作的切片：[1 2]