- slice添加元素

```
arr := [...]int{0, 1, 2, 3, 4, 5, 6, 7}
s1 := arr[2:6]
s2 := s1[3:5]

s3 := append(s2, 10)
// s4, s5不再是对arr的赋值
s4 := append(s3, 11)
s5 := append(s4, 12)

fmt.Println("s3, s4, s5 = ", s3, s4, s5) // s3, s4, s5 =  [5 6 10] [5 6 10 11] [5 6 10 11 12]
fmt.Println("arr = ", arr) // arr =  [0 1 2 3 4 5 6 10]
```

> - 添加元素如果超越cap，系统会重新分配更大的底层数组
> - 由于值传递的关系，必须接收append的返回值
> - s = append(s, v)

- 创建slice

```
// create slice
var s []int // 为空时，默认值nil
for i := 0; i < 100; i++ {
  s = append(s, 2*i+1)
}
fmt.Println(s)
// 有初始化值的slice
s6 := []int{2, 4, 6, 8}
fmt.Println(s6)

// 创建已知长度的slice
s7 := make([]int, 16)
fmt.Println(s7, cap(s7))
s8 := make([]int, 10, 32)
fmt.Println(s8, cap(s8))
```

- 拷贝slice

```
copy(s7, s6)
fmt.Println(s7, cap(s7)) // [2 4 6 8 0 0 0 0 0 0 0 0 0 0 0 0] 16
```

- 删除

```
s7 = append(s7[:3], s7[4:]...)
fmt.Println(s7, len(s7), cap(s7)) // [2 4 6 0 0 0 0 0 0 0 0 0 0 0 0] 15 16

// 删除头部元素
head := s7[0]
s7 = s7[1:]
fmt.Println(head) // 2
fmt.Println(s7, len(s7), cap(s7)) // [4 6 0 0 0 0 0 0 0 0 0 0 0 0] 14 15
// 删除尾部元素
tail := s7[len(s7) - 1]
s7 = s7[:len(s7) - 1]
fmt.Println(tail) // 0
fmt.Println(s7, len(s7), cap(s7)) // [4 6 0 0 0 0 0 0 0 0 0 0 0] 13 15
```

三）Map

```
m := map[string]string {
  "name": "jack",
  "age": "22",
  "sex": "male",
}
```

- 复合map

map[K]V, map[K1]map[K3]V

```
m1 := make(map[string]int)
fmt.Println(m1) // map[]
```

- 遍历map

```
m2 := map[string]string{
    "name": "jack",
    "age":  "22",
    "sex":  "male",
}
// 遍历map
for k, v := range m2 {
  fmt.Println(k, v)
}
// map遍历是无序的，遍历出来的顺序是变化的
```

- 获取map的值

```go
fmt.Println(m2["name"]) // "jack"
fmt.Println(m2["neme"]) // key错误，打印为空

// name, ok := m3["name"];ok来判断元素是否存在
if name, ok := m2["name"]; ok {
  fmt.Println(name)
} else {
  fmt.Println("key不存在")
}
```

- 删除map元素

```go
delete(m2, "name")
fmt.Println(m2) // map[age:22 sex:male]
```

> - map使用哈希表，必须可以比较相等
> - 除了slice、map、function的内建类型都可以作为map的key
> - struct类型不包含上述字段也可以作为key