Golang 随记（slice）

- make创建切片，需要指定切片类型，长度，容量（可选，否则长度和容量都将会设置为一样）。
- slice可以通过[]int{}进行赋值和初始化。
- 切片slice是对标其它编程语言中通俗意义上的动态数组，切片元素存放在一块连续的内存地址，可以用索引获取指定元素，切片长度和容量是可变的，使用过程中可以根据需要进行扩容。
- slice结构体
  - array：底层数组的起点地址，进行slice传递时，内存地址是一致的，或者说是引用传递（虽然slice是值传递，但是array是指针，也就是复用的同一片内存空间，最后的效果是引用传递，但是如果在副本中对len或cap进行修改，是不会引起原本slice的变化的）
  - len：长度，slice中实际存储的元素数量
  - cap：容量，slice当前能存储的元素的最大数量
- slice截取的原理
  - 修改array，同一块地址，起点不一样
  - 修改len
  - 修改cap
- slice append操作后len会加一，cap如果不够用会进行扩容（注意如果够用则不会扩容，同一个slice容量也不会减小），array的地址也可能变。

s := make([]int, 5)（len和cap都设置为5）

for 5次循环使用append加入i

最后的结果是前面5个为0，后面五个为预期的数字

为了实现预期目的，要不就make设置len为0，不然就通过遍历去设置，而不是append

- 如果知道slice的大概容量，尽量在初始化时设置好容量值，避免在运行时多次去扩充容量。
- slice扩容（1.19），下面的都是if-else循环：
  - 预期的新容量（老容量+append的大小）大于等于老容量的两倍，直接取新容量作为扩容后的容量
  - 如果老容量小于256，扩容后的新容量为老容量的2倍
  - 如果原容量大于等于 256，在原容量 n 的基础上循环执行 n += (n+3*256)/4 的操作，直到 n 大于等于预期新容量，并取 n 作为新容量（计算过程可能越界，那么直接使用新容量）
  - memorymove将老切片元素拷贝到新切片中
- slice删除元素的原理和截取一致或者是多个截取的append（删除中间元素）。
- 快速删除slice的元素，可以用[:0]
- 将一个slice赋值给slice是浅拷贝，指向同一块地址空间，如果想深拷贝可以使用copy(slice1, slice2)。或者make然后去遍历赋值。
- slice原生不支持并发，需要加锁

```
func Test_slice(t *testing.T){
    s := make([]int,10,12)
    s1 := s[8:]
    changeSlice(s1)
    t.Logf("s: %v, len of s: %d, cap o
    t.Logf("s1: %v, len of s1: %d, cap
}


func changeSlice(s1 []int){
  s1 = append(s1, 10)
}
```

答案：

```
s: [0 0 0 0 0 0 0 0 0 0], len of s: 10
s1: [0 0], len of s1: 2, cap of s1: 4
```

虽然切片是引用传递，但是在方法调用时，传递的会是一个新的 slice header.

因此在局部方法 changeSlice 中，虽然对 s1 进行了 append 操作，但这这会在局部方法中这个独立的 slice header 中生效，不会影响到原方法 Test_slice 当中的 s 和 s1 的长度和容量.

```
func Test_slice(t *testing.T){
    s := []int{0,1,2,3,4}
    s = append(s[:2],s[3:]...)
    t.Logf("s: %v, len: %d, cap: %d", s
    v := s[4]
    // 是否会数组访问越界
}
```

答案：

输出内容为：

```
 s: [0 1 3 4], len: 4, cap: 5
```

会发生 panic

执行完上述 append 操作之后，s 的实际长度为 4，容量维持不变为 5. 此时访问 s[4]会发生数组越界的错误.