golang

数组与切片

切片的初始化和追加

```
s := make([]int, 3) //初始化均为0
s = append(s, 1, 2, 3) //切片追加
```

slice拼接

```
s1 := []int{1, 2, 3}
s2 := []int{4, 5, 6}
//s3 := append(s1, s2)  两个slice拼接时，需要将第二个slice ...打散
s3 := append(s1, s2...)
```

new和make区别

- 两者都是内存的分配(堆上)
- make只能用于slice、map以及channel的初始化(非零值)
- new用于类型的内存分配，并且内置为零
- make返回的还是这三个引用类型本身
- new返回的是指向类型的指针

map

创建map两种方式

```
mp1 := make(map[string]int)
mp1["a"] = 1
```

```
mp2 := map[string]int{"a": 1}
```

错误: var mp3 map[string]int //var只声明了map没有进行初始化

```
mp3["a"] = 1
```

map结构体成员不能修改

```
type Person struct {
    Id   int
    Name string
}
mp4["student"] = Person{
    Id:   1,
    Name: "xiaoming",
} //加上指针就可以&Person{}
mp4["student"].Id = 1 //不能赋值
```

因为golang中map的value本身是不可寻址的

多个协程并发访问一个map，有可能会导致程序退出，并打印下面的错误信息： fatal error: concurrent map read and map write。 当并发的协程数比较大时，遇到的概率较大，可以是sync.map

map的遍历赋值

```
m := make(map[string]*student)
stus := []student{
    {Name: "zhang", Age: 12},
    {Name: "li", Age: 13},
    {Name: "GAI", Age: 14},
}
for _, stu := range stus {
    fmt.Printf("%p\n", &stu)
    m[stu.Name] = &stu
}
for k, v := range m {
    fmt.Println(k, "=>", v.Name)
}
```

```
0xc00000c078
0xc00000c078
0xc00000c078
li => GAI
GAI => GAI
zhang => GAI
```

For each中，stu是结构体拷贝的一个副本(固定地址的临时变量)，所以m[stu.Name]=&stu实际上一致指向同一个指针，最终该指针为遍历的最后一个struct的值拷贝

```
// 遍历结构体数组，依次赋值给map
for i := 0; i < len(stus); i++  {
    m[stus[i].Name] = &stus[i]
}
```

interface

interface赋值问题

发生多态的要素(满足三个条件，父类指针可以调用子类方法)

- 有interface接口，并且有接口定义的方法
- 有子类去重新interface接口
- 有父类指针指向子类的具体对象

```
type People interface {
    Speak(string) string
}

type Student struct {
}

func (stu *Student) Speak(think string) (talk string) {
    if "love" == think {
        talk = "you are good boy"
    } else {
        talk = "hi"
    }
    return
}
func main() {
    var people People = &Student{} //不能使用var people People = Student{}
    fmt.Println(people.Speak("love"))
}
```

interface两种表现形式

empty interface

var MyInterface interface{}

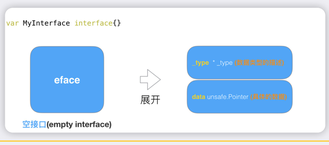
```
type eface struct {     //空接口
    _type *_type        //类型信息
    data  unsafe.Pointer //指向数据的指针(go语言中特殊的指针类型unsafe.Pointer类似于c语言中的void*)
}
```

```
type _type struct {
    size       uintptr  //类型大小
    ptrdata    uintptr  //前缀持有所有指针的内存大小
    hash       uint32   //数据hash值
    tflag      tflag
    align      uint8    //对齐
    fieldalign uint8    //嵌入结构体时的对齐
    kind       uint8    //kind 有些枚举值kind等于0是无效的
    alg        *typeAlg //函数指针数组，类型实现的所有方法
    gcdata     *byte
    str        nameOff
    ptrToThis typeOff
}
```

空接口eface

_type：是Go语言中所有类型的公共描述，Go语言中几乎所有的数据结构都可以抽象成_type，是所有类型公共描述，type负责决定data应该如何解释和操作

data属性：表示指向具体实例数据的指针



Non-empty interface

```
type MyInterface interface {
    function()
}
```

iface

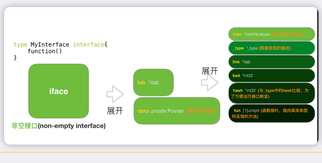
```
type iface struct {
  tab  *itab
  data unsafe.Pointer
}
```

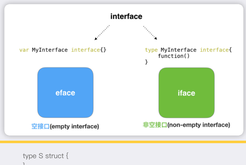
```
type itab struct {
  inter  *interfacetype   // 接口自身的元信息
  _type  *_type           // 具体类型的元信息
  link   *itab
  bad    int32
  hash   int32         // _type里也有一个同样的hash，此处多放一个是为了方便运行接口断言
  fun    [1]uintptr    // 函数指针，指向具体类型所实现的方法
}
```

interfacetype

包含一些关于interface本身的信息，比如package path，包含的method

type表示具体化的类型，与eface的type类型相同





interface与*interface区别

```
type S struct {
}
func f(x interface{}) {
}
func g(x *interface{}) {
}
func main() {
    s := S{}
    p := &s
    f(s) //A
    g(s) //B
    f(p) //C
    g(p) //D
}
```

B、D两行错误

B错误为： cannot use s (type S) as type *interface {} in argument to g:
*interface {} is pointer to interface, not interface

D错误为：cannot use p (type *S) as type *interface {} in argument to g:
*interface {} is pointer to interface, not interface

golang是强类型语言,interface是所有golang类型的父类，函数 f(x interface{})的interface{}可以支持传入golang的任何类型，包括指针

g(x *interface{}) *interface只能接受 *interface{}

channel

- 给一个nil channe发送数据，造成永远阻塞
- 从一个nil channel接收数据，造成永远阻塞
- 给一个关闭的channel发送数据，引起pannic
- 从一个已经关闭的channel接收数据，如果缓存区为空，返回一个零值
- 无缓冲channel是同步的，而缓冲的channel是非同步的