Golang内存模型

关键字： `Golang` 内存模型 并发

> Golang并发编程中涉及的GMM内存模型备忘。

nil

nil 代表 “zero value”（空值），和Java的 null 不一样。对于不同类型，nil 具体所代表的值不同。

```
package main
import "fmt"
type A struct {
}
func main() {
    var a *A = nil
    var ai interface{} = a
    var ei interface{} = nil
    fmt.Printf("ai == nil: %v\n", ai == nil)
    fmt.Printf("ai == ei: %v\n", ai == ei)
    fmt.Printf("ei == a: %v\n", a == ei)
    fmt.Printf("ei == nil: %v\n", ei == nil)
}
// -> 输出
// ai == nil: false
// ai == ei: false
// ei == a: false
// ei == nil: true
```

a 为“*A 类型的空值”，而 ai 为“*A 类型的空值转型为 interface{} 类型的值”。struct pointer 到 interface 有隐式转换。

> An interface value is nil only if the inner value and type are both unset, (nil, nil). In particular, a nil interface will always hold a `nil type`. If we store a nil pointer of type *int inside an interface value, the inner type will be *int regardless of the value of the pointer: (*int, nil). Such an interface value will therefore be non-nil even when the pointer inside is nil.

简言之: 在Go中, interace的实现包含二个元素, interface为nil当且仅当. 当声明 `var err *MyErrorImpl` 或 `var err *MyErrorImpl = nil` 时，已经变成了MyError<*MyErrorImpl,nil>, 故err作为MyError类型返回已经不再是nil.

```
func returnsError() error {
    var p *MyError = nil
    if bad() {
```

```
            p = ErrBad
        }
        return p // Will always return a non-nil error.
    }
```

以上代码是错误的，返回值永远不会为nil。正确的做法是：

```
    func returnsError() error {
      if bad() {
        return ErrBad
      }
      return nil
    }
```

参考 https://golang.org/doc/faq#nil_error

slice 和 array

slice更类似于"其他语言中的array"，简单来说，它是一个指向一段数组的指针。

slice定义：

```
    struct slice{
        ptr *Elem
        len int
        cap int
    }
```

An array variable denotes the entire array; it is not a pointer to the first array element (as would be the case in C).

数组指针指向的是数组，而不是数组的第一个元素。slice的地址指向的数组的第一个元素。

- array 是值类型，slice 和 map 是引用类型。它们是有很大区别的，尤其是在参数传递的时候。数组在使用的过程中都是值传递，将一个数组赋值给一个新变量或作为方法参数传递时，是将源数组在内存中完全复制了一份，而不是引用源数组在内存中的地址。
- slice 和 map 的变量仅仅声明是不行的，必须还要分配空间（也就是初始化，initialization）才可以使用。
- 在创建slice的时候，不要指定slice的长度。（否则就成了数组）

```
    var intArray = [20]int{}   //数组
```

```
var intSlice []int         //slice
```

上面声明了intSlice是一个指向int数组的slice，注意中括号里为空，这区别于array的声明。另外这只是一个声明，所以intSlice会得到一个slice的默认值，即为nil。 slice只能跟nil比较，如果你想尝试下面这代码：

```
letsTry := intSlice
fmt.Printf("intSlice == letsTry? %v\n", intSlice == letsTry)
```

你会得到下面这个错误信息：

> invalid operation: intSlice == letsTry (slice can only be compared to nil)

创建一个数组并赋给intSlice： `intSlice = make([]int, 1, 3)`

通过[]string直接创建切片再进行添加操作会发生什么呢？

```
test := []string{"test"}
println(test)
test = append(test,"test1")
println(test)

[1/1]0xc820041f08
[2/2]0xc82000e400
```

可以看到,test的内存已经发生了变化.也就是说,如果我们使用append()的时候,切片的长度已经大于我们最初分配的内存,此时切片会重新分配内存,分配的规则就是将当前长度转换为二进制然后左移一位。

以上创建slice test,实际上是创建了一个数组作为底层结构,然后创建了一个slice结构体并返回。

- append()函数默认在slice的末尾添加内容,而我们用make创建slice的时候"顺便"初始化了指定长度的内容。也就是说,append()会绕过这些被初始化的内容在末尾开始添加。
- 当我们只想返回slice的某一部分的时候,譬如用test[:3]来返回slice的前三个位置,如果我们对新建的slice进行append()操作时会覆盖掉原来slice指向的array的第四个位置。

不管是append操作，还是赋值操作，都影响了源数组或者其他引用同一数组的slice的元素。slice进行数组引用的时候，其实是将指针指向了内存中具体元素的地址，如数组的内存地址，事实上是数组中第一个元素的内存地址。

```
a := [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
sa := a[2:7]
sb := sa[3:8]
fmt.Printf("%p\n", sa)          //输出：0xc084004290
```

```
    fmt.Println(&a[2], &sa[0])      //输出：0xc084004290 0xc084004290
    fmt.Printf("%p\n", sb)          //输出：0xc0840042a8
    fmt.Println(&a[5], &sb[0])      //输出：0xc0840042a8 0xc0840042a8
```

make

happens-before