## 接口

### 结构体,集合和高阶函数

<https://github.com/Unknwon/the-way-to-go_ZH_CN/blob/master/eBook/11.14.md>

|  |
| --- |
| 通常你在应用中定义了一个结构体，那么你也可能需要这个结构体的（指针）对象集合，比如：  type Any interface{} type Car struct {     Model        string     Manufacturer string     BuildYear    int     // ... }  type Cars []\*Car 在定义所需功能时我们可以利用函数可以作为（其它函数的）参数的事实来使用高阶函数，例如：  1）定义一个通用的 Process() 函数，它接收一个作用于每一辆 car 的 f 函数作参数：  // Process all cars with the given function f: func (cs Cars) Process(f func(car \*Car)) {     for \_, c := range cs {         f(c)     } } 2）在上面的基础上，实现一个查找函数来获取子集合，并在 Process() 中传入一个闭包执行（这样就可以访问局部切片 cars）：  // Find all cars matching a given criteria. func (cs Cars) FindAll(f func(car \*Car) bool) Cars {      cars := make([]\*Car, 0)     cs.Process(func(c \*Car) {             if f(c) {                 append(cars,c)             }     )     return cars } 3）实现 Map 功能，产出除 car 对象以外的东西：  // Process cars and create new data. func (cs Cars) Map(f func(car \*Car) Any) []Any {         result := make([]Any, 0)         ix := 0         cs.Process(func(c \*Car) {                 result[ix] = f(c)                 ix++         })         return result } 现在我们可以定义下面这样的具体查询：  allNewBMWs := allCars.FindAll(func(car \*Car) bool {        return (car.Manufacturer == “BMW”) && (car.BuildYear > 2010) }) 4）我们也可以根据入参返回不同的函数。也许我们想根据不同的厂商添加汽车到不同的集合，但是这可能会是多变的。所以我们可以定义一个函数来产生特定的添加函数和 map 集：  func MakeSortedAppender(manufacturers[]string)(func(car\*Car),map[string]Cars) {     // Prepare maps of sorted cars.     sortedCars := make(map[string]Cars)     for \_, m := range manufacturers {         sortedCars[m] = make([]\*Car, 0)     }     sortedCars[“Default”] = make([]\*Car, 0)     // Prepare appender function:     appender := func(c \*Car) {         if \_, ok := sortedCars[c.Manufacturer]; ok {             sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)         } else {             sortedCars[“Default”] = append(sortedCars[“Default”], c)         }      }     return appender, sortedCars } 现在我们可以用它把汽车分类为独立的集合，像这样：  manufacturers := []string{“Ford”, “Aston Martin”, “Land Rover”, “BMW”, “Jaguar”} sortedAppender, sortedCars := MakeSortedAppender(manufacturers) allUnsortedCars.Process(sortedAppender) BMWCount := len(sortedCars[“BMW”]) 我们让这些代码在下面的程序 cars.go 中执行：  示例 11.18 cars.go：  // cars.go package main  import (     "fmt" )  type Any interface{} type Car struct {     Model       string     Manufacturer    string     BuildYear   int     // ... } type Cars []\*Car  func main() {     // make some cars:     ford := &Car{"Fiesta","Ford", 2008}     bmw  := &Car{"XL 450", "BMW", 2011}     merc := &Car{"D600", "Mercedes", 2009}     bmw2 := &Car{"X 800", "BMW", 2008}     // query:     allCars := Cars([]\*Car{ford, bmw, merc, bmw2})     allNewBMWs := allCars.FindAll(func(car \*Car) bool {       return (car.Manufacturer == "BMW") && (car.BuildYear > 2010)     })     fmt.Println("AllCars: ", allCars)     fmt.Println("New BMWs: ", allNewBMWs)     //     manufacturers := []string{"Ford", "Aston Martin", "Land Rover", "BMW", "Jaguar"}     sortedAppender, sortedCars := MakeSortedAppender(manufacturers)     allCars.Process(sortedAppender)     fmt.Println("Map sortedCars: ", sortedCars)     BMWCount := len(sortedCars["BMW"])     fmt.Println("We have ", BMWCount, " BMWs") }  // Process all cars with the given function f: func (cs Cars) Process(f func(car \*Car)) {      for \_, c := range cs {          f(c)      } }  // Find all cars matching a given criteria. func (cs Cars) FindAll(f func(car \*Car) bool) Cars {     cars := make([]\*Car, 0)      cs.Process(func(c \*Car) {         if f(c) {             cars = append(cars, c)         }     })     return cars }  // Process cars and create new data. func (cs Cars) Map(f func(car \*Car) Any) []Any {        result := make([]Any, len(cs))        ix := 0        cs.Process(func(c \*Car) {            result[ix] = f(c)            ix++        })        return result }  func MakeSortedAppender(manufacturers []string) (func(car \*Car), map[string]Cars) {      // Prepare maps of sorted cars.        sortedCars := make(map[string]Cars)         for \_, m := range manufacturers {            sortedCars[m] = make([]\*Car, 0)        }        sortedCars["Default"] = make([]\*Car, 0)         // Prepare appender function:        appender := func(c \*Car) {            if \_, ok := sortedCars[c.Manufacturer]; ok {                sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)            } else {                sortedCars["Default"] = append(sortedCars["Default"], c)            }        }        return appender, sortedCars } 输出：  AllCars:  [0xf8400038a0 0xf840003bd0 0xf840003ba0 0xf840003b70] New BMWs:  [0xf840003bd0] Map sortedCars:  map[Default:[0xf840003ba0] Jaguar:[] Land Rover:[] BMW:[0xf840003bd0 0xf840003b70] Aston Martin:[] Ford:[0xf8400038a0]] We have  2  BMWs |

## 实战示例

### 北京UTC+8 时间问题和日志打印格式问题

|  |
| --- |
| func main() {     // 设置日期、时间、文件名+行号（打印错误信息，比较方便定位错误点，问题定位很有用）     log.SetFlags(log.LstdFlags | log.Lshortfile)     // 北京UTC+8 时间问题     time.Local = time.FixedZone("CST", 3600\*8)     log.Println("当前时间:", time.Now().Local()) } |

## 疑惑

### 切片append

|  |
| --- |
| func main() {     data := make([]int, 1, 3)     fmt.Printf("main中data地址:%p\n", data)     fmt.Println(data)     mytest(data)     fmt.Println(data) } func mytest(data []int) {     fmt.Printf("mytest中data地址:%p\n", data)     data = append(data, 33)     fmt.Printf("mytest中data地址:%p\n", data)     fmt.Println(data) } /\*\*  运行结果:  main中data地址:0xc0420026a0  [0]  mytest中data地址:0xc0420026a0  mytest中data地址:0xc0420026a0  [0 33]  [0]  \*/ |

### Defer

* 分析为什么下面程序a()放回0

|  |
| --- |
| func main() {     fmt.Println("a return:", a()) } func a() int {     var i int     fmt.Println("&i:", &i)     defer func() {         i++         fmt.Println("a defer2:", &i)         fmt.Println("a defer2:", i)     }()     defer func() {         i++         fmt.Println("a defer1:", &i)         fmt.Println("a defer1:", i)     }()     return i } |

# 官方标准库

## 详情参文档:

<http://studygolang.com/pkgdoc>

## strings

### strings.Map()

func Map(mapping func(rune) rune, s string) string

将s的每一个unicode码值r都替换为mapping(r)，返回这些新码值组成的字符串拷贝。如果mapping返回一个负值，将会丢弃该码值而不会被替换。（返回值中对应位置将没有码值）

### strings.IndexFunc()

func IndexFunc(s string, f func(rune) bool) int

s中第一个满足函数f的位置i（该处的utf-8码值r满足f(r)==true），不存在则返回-1。

## unicode/utf8

### utf8.RunCountInString

|  |  |
| --- | --- |
|  | func main() {     str:="abc中d"     fmt.Println(len(str))     fmt.Println(len([]int32(str)))     // 以下方法效率更高     fmt.Println(utf8.RuneCountInString(str)) } /\* 运行结果 7 5 5 \*/ |

## sync

### sync.Mutex

* sync.Mutex 是一个互斥锁，它的作用是守护在临界区入口来确保同一时间只能有一个线程进入临界区
* 在 sync 包中还有一个 RWMutex 锁：他能通过 RLock() 来允许同一时间多个线程对变量进行读操作，但是只能一个线程进行写操作。如果使用 Lock() 将和普通的 Mutex 作用相同。包中还有一个方便的 Once 类型变量的方法 once.Do(call)，这个方法确保被调用函数只能被调用一次。

## Crypto/md5

### Md5.Sum

* func Sum(data []byte) [Size]byte
* 返回数据data的MD5校验和

### Md5.New

* func New() hash.Hash
* 返回一个新的使用MD5校验的hash.Hash接口

## Encoding/hex

### hex.EncodToString

* func EncodeToString(src []byte) string
* 将数据src编码为字符串s
* 示例

|  |
| --- |
| func main() {     md5Str := md5.New()     md5Str.Write([]byte(`123456`))     fmt.Println(md5Str.Sum(nil))     str := hex.EncodeToString(md5Str.Sum(nil))     fmt.Println(string(str))     byteStr,err:=hex.DecodeString(str)     if err!=nil{         fmt.Println(err)     }     fmt.Println(byteStr) } |