## 接口

### 接口的继承

* 当一个类型包含（内嵌）另一个类型（实现了一个或多个接口）的指针时，这个类型就可以使用（另一个类型）所有的接口方法。例如：

|  |
| --- |
| type Task struct {     Command string     \*log.Logger } |

* 这个类型的工厂方法像这样：

|  |
| --- |
| func NewTask(command string, logger \*log.Logger) \*Task {     return &Task{command, logger} } |

* 当 log.Logger 实现了 Log() 方法后，Task 的实例 task 就可以调用该方法：**task.Log()**
* 类型可以通过继承多个接口来提供像 多重继承 一样的特性：

|  |
| --- |
| type ReaderWriter struct {     \*io.Reader     \*io.Writer } |

* 上面概述的原理被应用于整个 Go 包，多态用得越多，代码就相对越少。这被认为是 Go 编程中的重要的最佳实践。
* 有用的接口可以在开发的过程中被归纳出来。添加新接口非常容易，因为已有的类型不用变动（仅仅需要实现新接口的方法）。已有的函数可以扩展为使用接口类型的约束性参数：通常只有函数签名需要改变。对比基于类的 OO 类型的语言在这种情况下则需要适应整个类层次结构的变化。

### Go中的面向对象

* Go 没有类，而是松耦合的类型、方法对接口的实现;
* OO 语言最重要的三个方面分别是：封装，继承和多态，在 Go 中它们是怎样表现的呢？
* 封装(数据隐藏)：和别的 OO 语言有 4 个或更多的访问层次相比，Go 把它简化为了 2 层
* 包范围内的：通过标识符首字母小写，对象 只在它所在的包内可见
* 可导出的：通过标识符首字母大写，对象 对所在包以外也可见
* 继承：用组合实现：内嵌一个（或多个）包含想要的行为（字段和方法）的类型；多重继承可以通过内嵌多个类型实现
* 多态：用接口实现：某个类型的实例可以赋给它所实现的任意接口类型的变量。类型和接口是松耦合的，并且多重继承可以通过实现多个接口实现。Go 接口不是 Java 和 C# 接口的变体，而且：接口间是不相关的，并且是大规模编程和可适应的演进型设计的关键

### 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 |

## 反射

### 基础概念

* 反射可以从接口值反射到对象，也可以从对象反射回接口值

### 方法和类型的反射

* Kind()
* kind总是返回底层类型(以下方法v.kind()返回reflect.Int)

|  |
| --- |
| func main() {     type MyInt int     var m MyInt = 5     v:=reflect.ValueOf(m)     fmt.Println(v.Kind()) } |

* Interface()
* 变量 v 的 Interface() 方法可以得到还原（接口）值，所以可以这样打印 v 的值：fmt.Println(v.Interface())
* 示例

|  |
| --- |
| func main() {     var x float64 = 3.4     fmt.Println("type:",reflect.TypeOf(x))     v:=reflect.ValueOf(x)     fmt.Println("value:",v)     fmt.Println("type:",v.Type())     fmt.Println("kind:",v.Kind())     //x 是一个 float64 类型的值，reflect.ValueOf(x).Float() 返回这个 float64 类型的实际值；同样的适用于 Int(), Bool(), Complex(), String()     fmt.Println("value:",v.Float())     fmt.Println(v.Interface())     fmt.Printf("value is %5.2e\n",v.Interface())     y:=v.Interface().(float64)     fmt.Println(y) } /\*\*  \* 运行结果  \*　type: float64  \* value: 3.4  \* type: float64  \* kind: float64  \* value: 3.4  \* 3.4  \* value is 3.40e+00  \* 3.4  \*/ |

### 通过反射修改(设置)值)

* 反射中有些内容是需要用地址去改变它的状态的
* 示例

|  |
| --- |
| func main() {     var x float64 = 3.4     fmt.Printf("&x = %p\n", &x)     v := reflect.ValueOf(x)     //v.SetFloat(3.1415) // Error: will panic: reflect.Value.SetFloat using unaddressable value     fmt.Println("settability of v:", v.CanSet())     v = reflect.ValueOf(&x)     fmt.Println(v)     fmt.Println("type of v:", v.Type())     v = v.Elem()     fmt.Println("The Elem of is:", v)     fmt.Println("settability of v:", v.CanSet())     v.SetFloat(3.1415)     fmt.Println(v.Interface())     fmt.Println(v)  } /\*\*  \* 运行结果  \* &x = 0xc0420381d0  \* settability of v: false  \* 0xc0420381d0  \* type of v: \*float64  \* The Elem of is: 3.4  \* settability of v: true  \* 3.1415  \* 3.1415  \*/ |

### 反射结构

* 有些时候需要反射一个结构类型。NumField() 方法返回结构内的字段数量；通过一个 for 循环用索引取得每个字段的值 Field(i)。
* 我们同样能够调用签名在结构上的方法，例如，使用索引 n 来调用：Method(n).Call(nil)。

|  |
| --- |
| type NotknownType struct {     s1, s2, s3 string } func (n NotknownType) String() string {     return n.s1 + " - " + n.s2 + " - " + n.s3 } var secret interface{} = NotknownType{"Ada", "Go", "Oberon"} func main() {     value:=reflect.ValueOf(secret)     typ:=reflect.TypeOf(secret)     fmt.Println(typ)     knd:=value.Kind()     fmt.Println(knd)     for i:=0;i<value.NumField();i++{         fmt.Printf("Field %d:%v\n",i,value.Field(i))     }     results :=value.Method(0).Call(nil)     fmt.Println(results) } /\*\*  \* 运行结果  \*　main.NotknownType  \* struct  \* Field 0:Ada  \* Field 1:Go  \* Field 2:Oberon  \* [Ada - Go - Oberon]  \*/ |

* 结构中只有被导出字段(首字母大写)才是可设置的

|  |
| --- |
| type T struct {     A int     B string } func main() {     t := T{23, "skidoo"}     s := reflect.ValueOf(&t).Elem()     typeOfT := s.Type()     for i := 0; i < s.NumField(); i++ {         f := s.Field(i)         fmt.Printf("%d: %s %s=%v\n", i, typeOfT.Field(i).Name, f.Type(), f.Interface())     }     s.Field(0).SetInt(77)     s.Field(1).SetString("Sunset Strip")     fmt.Println("t is now", t) } /\*\*  \* 运行结果  \* 0: A int=23  \* 1: B string=skidoo  \* t is now {77 Sunset Strip}  \*/ |

### Printf和反射

* Printf 中的 ... 参数为空接口类型。Printf 使用反射包来解析这个参数列表。所以，Printf 能够知道它每个参数的类型。因此格式化字符串中只有%d而没有 %u 和 %ld，因为它知道这个参数是 unsigned 还是 long。这也是为什么 Print 和 Println 在没有格式字符串的情况下还能如此漂亮地输出。

|  |
| --- |
| type Stringer interface {     String() string } type Celsius float64  func (c Celsius) String() string {     return strconv.FormatFloat(float64(c), 'f', 1, 64) + "°C" } type Day int var dayName = []string{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"} func (day Day) String() string {     return dayName[day] } func main() {     print(Day(1), "was", Celsius(18.36)) } func print(args ...interface{}) {     for i, arg := range args {         if i > 0 { os.Stdout.WriteString(" ") }         switch a := arg.(type) {         case Stringer:             os.Stdout.WriteString(a.String())         case int:             os.Stdout.WriteString(strconv.Itoa(a))         case string:             os.Stdout.WriteString(a)         default:             os.Stdout.WriteString("???")         }     } } /\*\*  \* 运行结果  \* Tuesday was 18.4°C  \*/ |

## Goroutine和Channel

### 基础概念

|  |
| --- |
| /\*\*  \* 只要进程还活者,即便携程的创建者生命周期结束了,它创建的携程依然能够继续存活.  \*/ func main() {     go go1()     time.Sleep(time.Second \* 30) } func go1() {     defer fmt.Println("exit go1")     go func() {         for i := 0; i < 10; i++ {             fmt.Println(i)             time.Sleep(time.Second)         }     }() } |

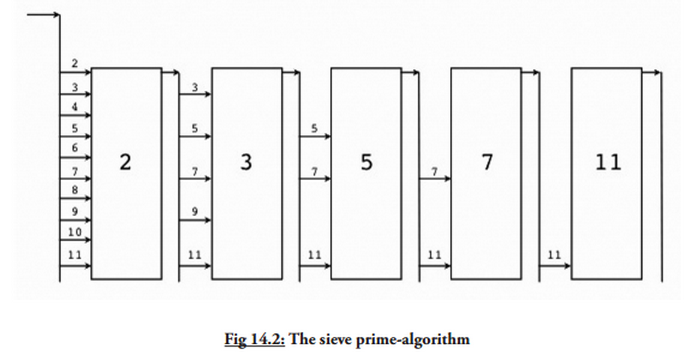
* Go多核计算例子
* 示例一

|  |
| --- |
| func main() {     runtime.GOMAXPROCS(4)     start := time.Now().UnixNano() / 1000000     c := make(chan bool, 10)     for i := 0; i < 10; i++ {         go Go(c, i)     }     for i := 0; i < 10; i++ {         <-c     }     end := time.Now().UnixNano() / 1000000     fmt.Println("use:", end-start, "ms") } func Go(c chan bool, index int) {     a := 1     for i := 0; i < 900000000; i++ {         a += i     }     fmt.Println(index, a)     c <- true } |

* 示例二

|  |
| --- |
| func main() {     wg:=sync.WaitGroup{}     wg.Add(10)     runtime.GOMAXPROCS(runtime.NumCPU())     for i := 0; i < 10; i++ {         go Go(&wg, i)     }     wg.Wait() } func Go(wg \*sync.WaitGroup, index int) {     a := 1     for i := 0; i < 1000000000; i++ {         a += i     }     fmt.Println(index, a)     wg.Done() } |

### 筛选素数算法

* 这里有一个来自 Go 指导的很赞的例子，打印了输出的素数，使用选择器（‘筛’）作为它的算法。每个 prime 都有一个选择器，如下图：
* 
* 版本一:协程 filter(in, out chan int, prime int) 拷贝整数到输出通道，丢弃掉可以被 prime 整除的数字。然后每个 prime 又开启了一个新的协程，生成器和选择器并发请求。

|  |
| --- |
| func main() {     ch := make(chan int)     go generate(ch)     for {         prime := <-ch         fmt.Print(prime, " ")         ch1 := make(chan int)         go filter(ch, ch1, prime)         ch = ch1     } } func generate(ch chan int) {     for i := 2; ; i++ {         ch <- i     } } func filter(in, out chan int, prime int) {     for {         i := <-in         if i%prime != 0 {             out <- i         }     } } |

* 版本二:引入了上边的习惯用法：函数 sieve、generate 和 filter 都是工厂；它们创建通道并返回，而且使用了协程的 lambda 函数。main 函数现在短小清晰：它调用 sieve() 返回了包含素数的通道，然后通过 fmt.Println(<-primes) 打印出来。

|  |
| --- |
| func main() {     primes := sieve()     for {         fmt.Println(<-primes)     } } func generate() chan int {     ch := make(chan int)     go func() {         for i := 2; ; i++ {             ch <- i         }     }()     return ch } func filter(in chan int, prime int) chan int {     out := make(chan int)     go func() {         for {             if i := <-in; i%prime != 0 {                 out <- i             }         }     }()     return out } func sieve() chan int {     out := make(chan int)     go func() {         ch := generate()         for {             prime := <-ch             ch = filter(ch, prime)             out <- prime         }     }()     return out } |

版本三,今日头条 Go 建千亿级微服务的实践

|  |
| --- |
| func main() {     origin, wait := make(chan int), make(chan struct{})     Processor(origin, wait)     for num := 2; num < 10000; num++ {         origin <- num     }     close(origin)     <-wait } func Processor(seq chan int, wait chan struct{}) {     go func() {         prime, ok := <-seq         if !ok {             close(wait)             return         }         fmt.Println(prime)         out := make(chan int)         Processor(out, wait)         for num := range seq {             if num%prime != 0 {                 out <- num             }         }         close(out)     }() } |

## 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) } |