## 接口

### 使用方法集与接口

* 作用于变量上的方法实际上是不区分变量到底是指针还是值的。当碰到接口类型值时，这会变得有点复杂，原因是接口变量中存储的具体值是不可寻址的，幸运的是，如果使用不当编译器会给出错误。考虑下面的程序：

|  |
| --- |
| type List []int func (l List) Len() int {     return len(l) } func (l \*List) Append(val int) {     \*l = append(\*l, val) } type Appender interface {     Append(int) } func CountInto(a Appender, start, end int) {     for i := start; i <= end; i++ {         a.Append(i)     } } type Lener interface {     Len() int } func LongEnough(l Lener) bool {     return l.Len()\*10 > 42 } func main() {     // A bare value     var lst List     // compiler error:     // cannot use lst (type List) as type Appender in argument to CountInto:     //       List does not implement Appender (Append method has pointer receiver)     // CountInto(lst, 1, 10)     if LongEnough(lst) { // VALID:Identical receiver type         fmt.Printf("- lst is long enough\n")     }     // A pointer value     plst := new(List)     CountInto(plst, 1, 10) //VALID:Identical receiver type     if LongEnough(plst) {         // VALID: a \*List can be dereferenced for the receiver         fmt.Printf("- plst is long enough\n")     } } |

* 在 lst 上调用 CountInto 时会导致一个编译器错误，因为 CountInto 需要一个 Appender，而它的方法 Append 只定义在指针上。 在 lst 上调用 LongEnough 是可以的因为 'Len' 定义在值上。
* 在 plst 上调用 CountInto 是可以的，因为 CountInto 需要一个 Appender，并且它的方法 Append 定义在指针上。 在 plst 上调用 LongEnough 也是可以的，因为指针会被自动解引用。
* 在**接口**上调用方法时，必须有和方法定义时相同的接收者类型或者是可以从具体类型 P 直接可以辨识的：
* 指针方法可以通过指针调用
* 值方法可以通过值调用
* 接收者是值的方法可以通过指针调用，因为指针会首先被解引用
* 接收者是指针的方法不可以通过值调用，因为存储在接口中的值没有地址
* 将一个值赋值给一个接口时，编译器会确保所有可能的接口方法都可以在此值上被调用，因此不正确的赋值在编译期就会失败。
* Go 语言规范定义了接口方法集的调用规则:
* 类型 \*T 的可调用方法集包含接受者为 \*T 或 T 的所有方法集
* 类型 T 的可调用方法集包含接受者为 T 的所有方法
* 类型 T 的可调用方法集不包含接受者为 \*T 的方法

### 空接口

* 每个 interface {} 变量在内存中占据两个字长：一个用来存储它包含的类型，另一个用来存储它包含的数据或者指向数据的指针。

### 复制数据切片至空接口切片

* 假设你有一个 myType 类型的数据切片，你想将切片中的数据复制到一个空接口切片中，类似：

|  |
| --- |
| var dataSlice []myType = FuncReturnSlice() var interfaceSlice []interface{} = dataSlice |

* 可惜不能这么做，编译时会出错：cannot use dataSlice (type []myType) as type []interface { } in assignment。原因是它们俩在内存中的布局是不一样的（参考 官方说明）。必须使用 for-range 语句来一个一个显式地复制：

|  |
| --- |
| var dataSlice []myType = FuncReturnSlice() var interfaceSlice []interface{} = make([]interface{}, len(dataSlice)) for ix, d := range dataSlice {     interfaceSlice[ix] = d } |

### 接口与动态类型

* Go的动态类型
* 和其它语言相比，Go 是唯一结合了接口值，静态类型检查（是否该类型实现了某个接口），运行时动态转换的语言，并且不需要显式地声明类型是否满足某个接口。该特性允许我们在不改变已有的代码的情况下定义和使用新接口。
* 接收一个（或多个）接口类型作为参数的函数，其实参可以是任何实现了该接口的类型。 实现了某个接口的类型可以被传给任何以此接口为参数的函数 。
* 类似于 Python 和 Ruby 这类动态语言中的 动态类型（duck typing）；这意味着对象可以根据提供的方法被处理（例如，作为参数传递给函数），而忽略它们的实际类型：它们能做什么比它们是什么更重要。
* 示例

|  |
| --- |
| type IDuck interface {     Quack()     Walk() } func DuckDance(duck IDuck) {     duck.Quack()     duck.Walk() } type Bird struct {     Name string } func (b \*Bird) Quack() {     fmt.Println("I am quacking by",b.Name) } func (b \*Bird) Walk() {     fmt.Println("I am walking by",b.Name) } func main() {     b := new(Bird)     b.Name="bird"     DuckDance(b) } /\*\*  \* 运行结果  \* I am quacking by bird  \* I am walking by bird  \*/ |

### 动态方法调用

* 像 Python，Ruby 这类语言，动态类型是延迟绑定的（在运行时进行）：方法只是用参数和变量简单地调用，然后在运行时才解析（它们很可能有像 responds\_to 这样的方法来检查对象是否可以响应某个方法，但是这也意味着更大的编码量和更多的测试工作）
* Go 的实现与此相反，通常需要编译器静态检查的支持：当变量被赋值给一个接口类型的变量时，编译器会检查其是否实现了该接口的所有函数。
* 因此 Go 提供了动态语言的优点，却没有其他动态语言在运行时可能发生错误的缺点。
* Go 的接口提高了代码的分离度，改善了代码的复用性，使得代码开发过程中的设计模式更容易实现。用 Go 接口还能实现 依赖注入模式。

|  |
| --- |
| type xmlWriter interface {     WriteXML(w io.Writer) error } // Exported XML streaming function. func StreamXML(v interface{}, w io.Writer) error {     if xw, ok := v.(xmlWriter); ok {         // It’s an  xmlWriter, use method of asserted type.         return xw.WriteXML(w)     }     // No implementation, so we have to use our own function (with perhaps reflection):     return encodeToXML(v, w) } // Internal XML encoding function. func encodeToXML(v interface{}, w io.Writer) error {     // ... } |

### 接口的提取

* 提取接口 是非常有用的设计模式，可以减少需要的类型和方法数量，而且不需要像传统的基于类的面向对象语言那样维护整个的类层次结构。
* Go 接口可以让开发者找出自己写的程序中的类型。假设有一些拥有共同行为的对象，并且开发者想要抽象出这些行为，这时就可以创建一个接口来使用。 假设我们需要一个新的接口 TopologicalGenus，用来给 shape 排序（这里简单地实现为返回 int）。我们需要做的是给想要满足接口的类型实现 Rank() 方法：

|  |
| --- |
| type Shaper interface {     Area() float32 } type TopologicalGenus interface {     Rank() int } type Square struct {     side float32 } func (sq \*Square) Area() float32 {     return sq.side \* sq.side } func (sq \*Square) Rank() int {     return 1 } type Rectangle struct {     length, width float32 } func (r Rectangle) Area() float32 {     return r.length \* r.width } func (r Rectangle) Rank() int {     return 2 } func main() {     r := Rectangle{5, 3} // Area() of Rectangle needs a value     q := &Square{5}      // Area() of Square needs a pointer     shapes := []Shaper{r, q}     fmt.Println("Looping through shapes for area ...")     for n, \_ := range shapes {         fmt.Println("Shape details: ", shapes[n])         fmt.Println("Area of this shape is: ", shapes[n].Area())     }     topgen := []TopologicalGenus{r, q}     fmt.Println("Looping through topgen for rank ...")     for n, \_ := range topgen {         fmt.Println("Shape details: ", topgen[n])         fmt.Println("Topological Genus of this shape is: ", topgen[n].Rank())     } } /\*\*  \* 运行结果  \* Looping through shapes for area ...  \* Shape details:  {5 3}  \* Area of this shape is:  15  \* Shape details:  &{5}  \* Area of this shape is:  25  \* Looping through topgen for rank ...  \* Shape details:  {5 3}  \* Topological Genus of this shape is:  2  \* Shape details:  &{5}  \* Topological Genus of this shape is:  1  \*/ |

* 所以你不用提前设计出所有的接口；整个设计可以持续演进，而不用废弃之前的决定。类型要实现某个接口，它本身不用改变，你只需要在这个类型上实现新的方法

### 显示地指明类型实现了某个接口

* 如果你希望满足某个接口的类型显式地声明它们实现了这个接口，你可以向接口的方法集中添加一个具有描述性名字的方法。例如：

|  |
| --- |
| type Fooer interface {     Foo()     ImplementsFooer() } |

* 类型 Bar 必须实现 ImplementsFooer 方法来满足 Footer 接口，以清楚地记录这个事实。

|  |
| --- |
| type Bar struct{} func (b Bar) ImplementsFooer() {}  func (b Bar) Foo() {} |

* 大部分代码并不使用这样的约束，因为它限制了接口的实用性。但是有些时候，这样的约束在大量相似的接口中被用来解决歧义。

### 空接口和函数重载

* 我们看到函数重载是不被允许的。在 Go 语言中函数重载可以用可变参数 ...T 作为函数最后一个参数来实现。如果我们把 T 换为空接口，那么可以知道任何类型的变量都是满足 T (空接口）类型的，这样就允许我们传递任何数量任何类型的参数给函数，即重载的实际含义。函数 fmt.Printf 就是这样做的：

|  |
| --- |
| fmt.Printf(format string, a ...interface{}) (n int, errno error) |

* 这个函数通过枚举 slice 类型的实参动态确定所有参数的类型。并查看每个类型是否实现了 String() 方法，如果是就用于产生输出信息。

### 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)         }     }() } |

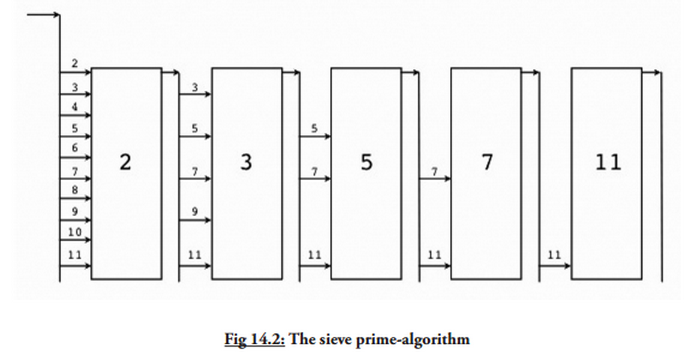
* Go多核计算例子
* 示例一

|  |
| --- |
| 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) } |