回顾

map 字典 key--value 存放是无序的

结构体

有时候我们需要将不同类型的数据组合在一起成为一个有机的整体，一个学生：学号，有姓名 性别年龄 地址属性

var id int

var name string

var sex byte

var age int

var addr string

结构体

type Student **struct**{

var id int

var name string

var sex byte

var age int

var addr string

}

var s Student

s.id = 1

s.name = "zhangsan"

面向对象编程

java 编程里面一切皆对象

go语言没有沿袭传统的面向对象的编程的概念，比如继承（不支持继承，尽管匿名字段的内存分布和行为类似继承）

- 封装：通过方法实现
- 继承 ：通过匿名字段实现

- 多态：通过接口方式实现

匿名组合

一般情况下，定义结构体时候字段名和类型是一一对应的，实际中，go支持只提供类型，而不写字段的方式，这就是匿名字段，也称为嵌入字段

当匿名字段也是一个结构体的时候，那么这个结构体所有的全部字段都被隐式的方式引入到当前定义的这个结构体中。

```
type Person struct{
    name string
    sex byte
    age int
}

type Student Struct{

   Person //匿名字段，student默认包含name sex age 这些字段
   id  int
   addr string

}
```

匿名字段初始化案例

```
package main

import "fmt"

type Person struct {
   name string //名字
   sex  byte  // 姓名
   age int  // 年龄
}

type Student struct {
   Person //只有类型没有字段，匿名字段 基础person所有的成员
   id  int
   addr string
}

func main()  {
   //结构体顺序初始化，把所有字段都要初始化
   var s1 Student =Student{Person{"mike",'m',18},1,"bj"}
   fmt.Println("s1=",s1)

   //自动类型推导
   s2 := Student{Person{"mike",'m',18},1,"bj"}
```

```
    //%+v 显示更详细的信息
    fmt.Printf("s2=%+v\n",s2)

    //指定类型初始化，没有初始化的常用类型按照默认值赋值 int 0 string 为空
    s3:=Student{id:1}
    fmt.Printf("s3=%+v\n",s3)

    s4 := Student{Person:Person{name:"zhangsan"},id:1}
    fmt.Printf("s4=%+v",s4)

    //s5 := Student{"mike",'m',18,1,"bj"} //err


}
```

注意

在同一个包下面，定义的结构体 或者常量，在任何文件下面都可以使用

比如我们在文件1定义

```
type Person struct {
    name string //名字
    sex  byte  // 姓名
    age int  // 年龄
}

type Student struct {
    Person //只有类型没有字段，匿名字段 基础person所有的成员
    id  int
    addr string
}
```

在文件2里面可以直接使用

Person和Student 方法也类似

同一个包下面对外公布的方法不能重名

匿名字段成员的操作

```
package main

import "fmt"

type Person1 struct {
    name string //名字
    sex  byte  // 姓名
```

```
    age int  // 年龄
}

type Student1 struct {
    Person1 //只有类型没有字段，匿名字段 基础person所有的成员
    id  int
    addr string
}


func main(){
    s1 := Student1{Person1{"mike",'m',18},1,"bj"}

    //对象成员的操作方式一
    s1.name = "yoyo"
    s1.sex = 'f'
    s1.age = 22
    s1.id = 7777
    s1.addr="sh"

    //对象操作匿名字段方式二
    s1.Person1=Person1{"tom",'m',19}


    fmt.Println(s1.Person1.name)
    fmt.Println(s1.name) // s1.Person1.name == s1.name
    fmt.Println("s1=",s1)

}
```

同名字段的操作

就近原则：如果能在本作用域找到此成员，就操作此成员，如果找不到，就找到继承的字段

```
package main

import "fmt"

type Person2 struct {
    name string //名字
    sex  byte  // 姓名
    age int  // 年龄
}

type Student2 struct {
    Person2 //只有类型没有字段，匿名字段 基础person所有的成员
    id  int
    addr string
    name string //和person同名了
}
```

```
func main(){
   //声明（定义一个变量）
   var s Student2
  //就近原则：如果能在本作用域找到此成员，就操作此成员，如果找不到，就找到继承的字段
   s.name ="zhangsan" //Student的name
   s.sex='m'
   s.age = 18
   s.addr="bj"
   fmt.Printf("s=%+v\n",s)

}
```

非结构体的匿名字段

```
package main

import "fmt"

type mystr string //自定义类型，给一个类型改名

type Person3 struct {
   name string //名字
   sex  byte  // 姓名
   age int  // 年龄
}

type Student3 struct {
   Person3 //只有类型没有字段，匿名字段 基础person所有的成员
   int    //基础类型的匿名字段
   mystr
}

func main(){
   s := Student3{Person3{"mike",'m',18},666,"zhangsan"}
   fmt.Printf("s=%+v\n",s)
   fmt.Println(s.name,s.age,s.sex,s.int,s.mystr)
   fmt.Println(s.Person3,s.int,s.mystr)

}
```

结构体指针类型字段

```
package main

import "fmt"
```

```
type Person5 struct {
   name string //名字
   sex  byte  // 姓名
   age int  // 年龄
}

type Student5 struct {
   *Person5 //指针类型 Person5地址
   id  int
   addr string
}

func main()  {
  s1 := Student5{&Person5{"mike",'m',18},777,"bj"}
  fmt.Printf("s1=%+v\n",s1)
  fmt.Println(s1.Person5.name)
  fmt.Println(s1.name)

  //先定义变量
  var s2 Student5
  s2.Person5 =new(Person5) //分配空间地址
  s2.name  = "tom"
  s2.sex = 'm'
  s2.age = 18
  s2.id = 2222
  s2.addr="bj"
  fmt.Println(s2.name,s2.sex,s2.age,s2.id,s2.addr)


}
```

方法

给一个类型绑定一个函数叫方法

```
//面向对象 方法:  给一个类型绑定一个函数
type  long int

func (a long)add01(b long) long{
    return  a+b
}
```

面向过程和面向对象的方法

```
package main
```

```
import (
   "fmt"
)

//面向过程
func Add(a,b int) int{
   return  a +b
}

//面向对象 方法： 给一个类型绑定一个函数
type  long int

func (a long)add01( b long) long{
   return  a+b
}

func main(){
    //考驾照 ---->买车--->自己开车 ----亲自操作
    // 出租车    出租车.开车方法（目的地）

    var result int
    result = Add(1,2)
    fmt.Println("result=",result)

    //定义一个变量
    var a  long = 2
    //调用方式 变量.函数（所需要参数）
    result1 := a.add01(3)
    fmt.Println(result1)
    //add01(a,3)
   // result1 := a.add01(3)
    //fmt.Println(result1)
}
```

为结构体添加类型方法

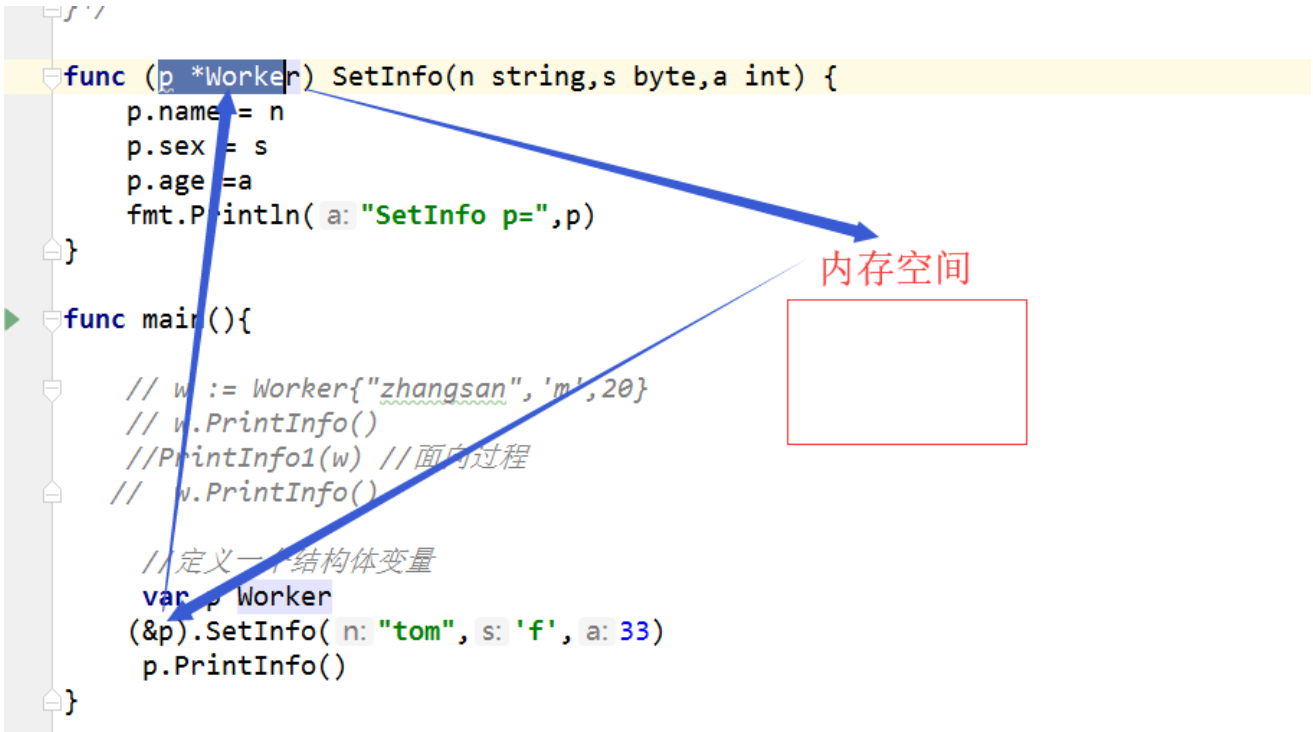
```
package main

import "fmt"

type Worker struct {
    name string //名字
    sex byte   //性别
    age int    //年龄
}

func PrintInfo1(a Worker)  {
   fmt.Println("a=",a)
}
```

```
func (a Worker) PrintInfo(){
   fmt.Println("a=",a)
}

func (p Worker) SetInfo(n string,s byte,a int) {
   p.name = n
   p.sex = s
   p.age =a
   fmt.Println("SetInfo p=",p)
}

func main(){

   // w := Worker{"zhangsan",'m',20}
   // w.PrintInfo()
   //PrintInfo1(w) //面向过程
   //  w.PrintInfo()

     //定义一个结构体变量
     var p Worker
     p.SetInfo("tom",'f',33)
     p.PrintInfo()
}
```

打印结果：

SetInfo p= {tom 102 33} a= { 0 0}

```
}

func (p Worker) SetInfo(n string,s byte,a int) {
    p.name = n
    p.sex = s
    p.age =a
    fmt.Println("SetInfo p=",p)
}

func main(){

    // w := Worker{"zhangsan",'m',20}
    // w.PrintInfo()
    //PrintInfo1(w) //面向过程
    //  w.PrintInfo()

      //定义一个结构体变量
      var p Worker
      p.SetInfo("tom",'f',33)
      p.PrintInfo()
}
```



指针作为参数

```
package main
```

```
import "fmt"

type Worker struct {
    name string //名字
    sex byte   //性别
    age int    //年龄
}

func PrintInfo1(a Worker)  {
   fmt.Println("a=",a)
}

func (a Worker) PrintInfo(){
   fmt.Println("a=",a)
}

/*func (p Worker) SetInfo(n string,s byte,a int) {
   p.name = n
   p.sex = s
   p.age =a
   fmt.Println("SetInfo p=",p)
}*/

func (p *Worker) SetInfo(n string,s byte,a int) {
   p.name = n
   p.sex = s
   p.age =a
   fmt.Println("SetInfo p=",p)
}

func main(){

   // w := Worker{"zhangsan",'m',20}
   // w.PrintInfo()
   //PrintInfo1(w) //面向过程
   //  w.PrintInfo()

     //定义一个结构体变量
     var p Worker
   //(&p).SetInfo("tom",'f',33)
    p.SetInfo("tom",'f',33)
     p.PrintInfo()
}
```

方法

在面向对象编程中，一个对象其实就是一个简单的值或者变量，在这个对象中包含一些函数，这种带有**接收者的函数我们成为方法**，本质上，一个方法则是一个和特殊类型关联的函数，以后所有的操作我们都是借助对象的方法进行操作。

在go语言中 我们可以给自定义的类型添加对应的方法

方法语法格式

recevier Receviertype 隐式接收将作为第一个实参

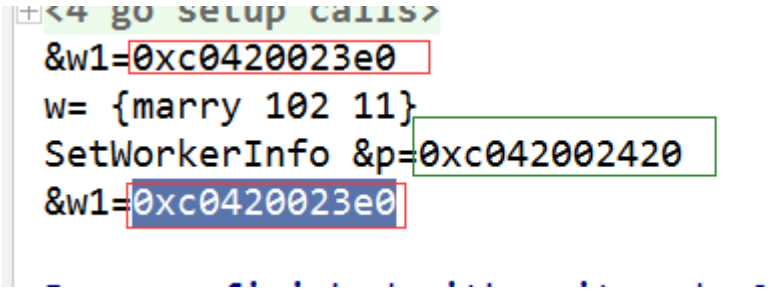
func (recevier Receviertype) funcName(对应的参数) (返回值)

recevier : 可以任意命名

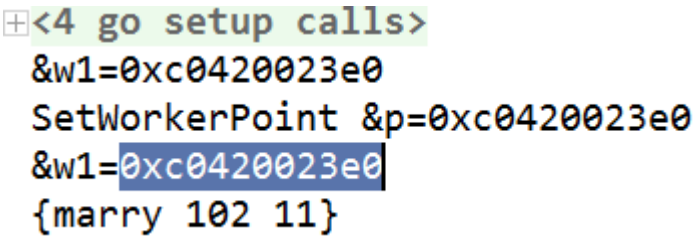
Receviertype 类型可以是T 可以是*T

值语义和引用语义

```
package main

import (
   "fmt"
)

type Worker1 struct {
   name string
   sex byte
   age int
}


//接收者为普通变量，非指针，值语义，一份拷贝
func (w Worker1) SetWorkerInfo(n string,s byte,a int)  {
    w.name = n
    w.sex = s
    w.age =a
    fmt.Println("w=",w)
    fmt.Printf("SetWorkerInfo &p=%p\n",&w)
}


func main(){

   w1 := Worker1{"tom",'m',22}
   fmt.Printf("&w1=%p\n",&w1)
   w1.SetWorkerInfo("marry",'f',11)
   fmt.Printf("&w1=%p\n",&w1)


}
```




```
//引用传递
func (w *Worker1) SetWorkerPoint(n string,s byte,a int)  {
   w.name = n
   w.sex = s
   w.age =a
   fmt.Printf("SetWorkerPoint &p=%p\n",&w)
}
```



指针变量的方法集

```
package main

import "fmt"

type Worker2 struct {
   name string
   sex byte
   age int
}


//接收者为普通变量，非指针，值语义，一份拷贝
func (w Worker2) SetWorkerInfo()  {
   fmt.Printf("SetWorkerInfo &p=%p\n",&w)
}


//引用传递
func (w *Worker2) SetWorkerPoint()  {
   fmt.Printf("SetWorkerPoint &p=%p\n",w)
}
```

```
func main(){
    //结构体变量是一个指针类型的变量，它能够调用方法，这些可以调用的方法 简称方法集
    w := &Worker2{"mike",'m',19}
    w.SetWorkerPoint()
    (*w).SetWorkerPoint() // 先把（*w）转换成w在调用 定价与上面
    //先把指针w，转换成*w在调用
    //(*w).SetWorkerInfo()
    w.SetWorkerInfo()


}
```

普通变量的方法集

```
package main

import "fmt"

type Worker2 struct {
    name string
    sex byte
    age int
}


//接收者为普通变量，非指针，值语义，一份拷贝
func (w Worker2) SetWorkerInfo()  {
    fmt.Printf("SetWorkerInfo &p=%p\n",&w)
}


//引用传递
func (w *Worker2) SetWorkerPoint()  {
    fmt.Printf("SetWorkerPoint &p=%p\n",w)
}

func main(){
    //结构体变量是一个指针类型的变量，它能够调用方法，这些可以调用的方法 简称方法集
    /*w := &Worker2{"mike",'m',19}
    w.SetWorkerPoint()
    (*w).SetWorkerPoint() // 先把（*w）转换成w在调用 定价与上面
    //先把指针w，转换成*w在调用
    //(*w).SetWorkerInfo()
    w.SetWorkerInfo()*/

    w := Worker2{"mike",'m',19}
    w.SetWorkerPoint() //w--->(&p)
    w.SetWorkerInfo()
```

```
}
```

总结：无论是普通的方法集 还是 指针的方法集，你主要想值靠拷贝

```
//接收者为普通变量，非指针，值语义，一份拷贝
func (w Worker2) SetWorkerInfo()  {
   fmt.Printf("SetWorkerInfo &p=%p\n",&w)
}
```

如果想改改变值

```
//引用传递
func (w *Worker2) SetWorkerPoint()  {
   fmt.Printf("SetWorkerPoint &p=%p\n",w)
}
```

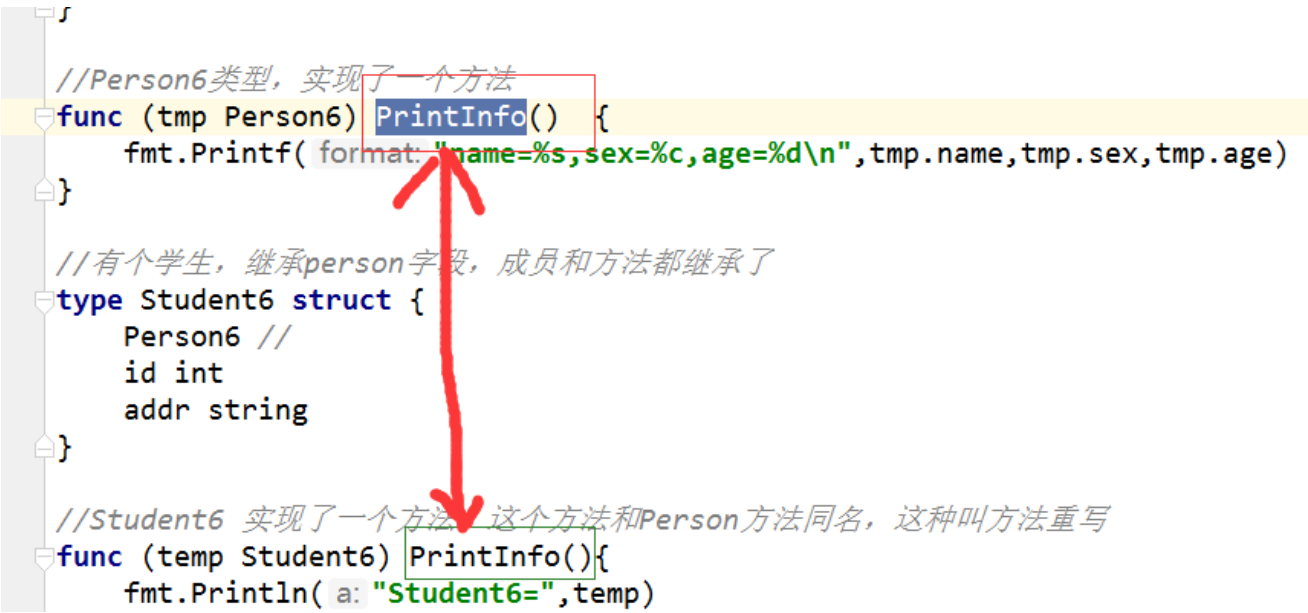
指针--普通 内部可以互相转换的

方法的继承

```
package main

import "fmt"

type Person6 struct {
   name string
   sex  byte
   age int
}

//Person6类型，实现了一个方法
func (tmp Person6) PrintInfo()  {
   fmt.Printf("name=%s,sex=%c,age=%d\n",tmp.name,tmp.sex,tmp.age)
}

//有个学生，继承person字段，成员和方法都继承了
type Student6 struct {
   Person6 //
   id int
   addr string
}


func main(){
    s := Student6{Person6{"mike",'m',18},666,"bj"}
    s.PrintInfo()
```

```
}
```

方法重写



```
//Person6类型，实现了一个方法
func (tmp Person6) PrintInfo()  {
    fmt.Printf( format: "name=%s,sex=%c,age=%d\n",tmp.name,tmp.sex,tmp.age)
}

//有个学生，继承person字段，成员和方法都继承了
type Student6 struct {
    Person6 //
    id int
    addr string
}

//Student6 实现了一个方法，这个方法和Person方法同名，这种叫方法重写
func (temp Student6) PrintInfo(){
    fmt.Println( a: "Student6=",temp)
}
```

```
package main

import "fmt"

type Person6 struct {
   name string
   sex  byte
   age int
}

//Person6类型，实现了一个方法
func (tmp Person6) PrintInfo()  {
   fmt.Printf("name=%s,sex=%c,age=%d\n",tmp.name,tmp.sex,tmp.age)
}

//有个学生，继承person字段，成员和方法都继承了
type Student6 struct {
   Person6 //
   id int
   addr string
}

//Student6 实现了一个方法，这个方法和Person方法同名，这种叫方法重写
func (temp Student6) PrintInfo(){
   fmt.Println("Student6=",temp)
}


func main(){
```

```
    s := Student6{Person6{"mike",'m',18},666,"bj"}
    //就近原则：先找本作用域的方法，找不到在去找继承的方法，如果找不到 调用报错
    s.PrintInfo()

      //显示调用继承的方法
    s.Person6.PrintInfo()
}
```

把方法复制给变量

```
package main

import "fmt"

type Person7 struct {
   name string //姓名
   sex byte  //性别，字符类型
   age int  //年龄
}

func (p Person7) SetInfoValue(){
   fmt.Printf("SetInfoValue:%p,%v\n",&p,p)
}

func (p *Person7) SetInfoPoint(){
   fmt.Printf("SetInfoPoint:%p,%v\n",p,p)
}

func main()  {
   p := Person7{"mike",'m',18}
   fmt.Printf("main:%p,%v\n",&p,p)

   p.SetInfoPoint() //传统的调用方式

   pFunc := p.SetInfoPoint //这个是方法值，调用函数时，无需传递接收者，
   pFunc()

   pSetinfo := p.SetInfoValue
   pSetinfo()     //这个就等价于p.SetInfoValue()

}
```

接口

比如跳高的猫 ：正常猫（ name color age ） eat sleep

jump（ ）

在go语言中，接口（interface）是一个自定义的类型，接口类型具体描述了一系列的方法集合

接口类型是一种抽象的类型，它不会暴露出所有的对象的内部值的机构和这个对象支持的基础操作集合，他们只展他们自己的方法。因此接口类型不能实例化。必须通过子类来实现化

go通过接口实现了鸭子类型（duck-typing）:" 当看到一只鸟走起来像鸭子，游泳起来像鸭子，叫起来像鸭子，那么这只鸟就可以被称为鸭子"，我们不关心对象是什么类型，到底是不是鸭子，我们只关心行为

接口定义：

type Humaner interface{

SayHi()

}

- 接口命名习惯以er结尾
- 接口只有方法声明，没有方法实现，没有数据字段
- 接口可以匿名嵌入其他接口，或嵌入到结构体中

注意点： 接口用来定义行为类型，这些被定为的行为不由接口直接实现，而是通过方法由用户定义的类型实现

接口的定义和实现

```
package main

import "fmt"

//定义一个接口类型
type Humaner interface {
    //方法，只有声明，没有实现，由别的类型（自定义类型）实现
    sayHi()
}

type Student9 struct {
   name string
   id  int
}

func (temp *Student9) sayHi(){
    fmt.Printf("Student9[%s,%d] sayHi\n",temp.name,temp.id)
}

type Teacher struct {
    addr string
    id  int
}

func (temp *Teacher) sayHi()  {
   fmt.Printf("Teacher[%s,%d] sayHi\n",temp.addr,temp.id)
}
```

```
type Worker4 struct {
   id int
}

func (temp *Worker4) sayHi()  {
   fmt.Printf("Worker4[%d] sayHi\n",temp.id)
}


func main()  {

   //定义一个接口的类型
   var i Humaner
   // 只有实现了此接口的方法的类型，那个这个类型的变量才能赋值给接口变量
   s := &Student9{"mike",7777}

   i =s
   i.sayHi()

   t := &Teacher{"bj",9999}
   i = t

   i.sayHi()

   w := &Worker4{8888}
   i= w
   i.sayHi()


}
```

多态：只有一个函数，可以表现有不同的形式

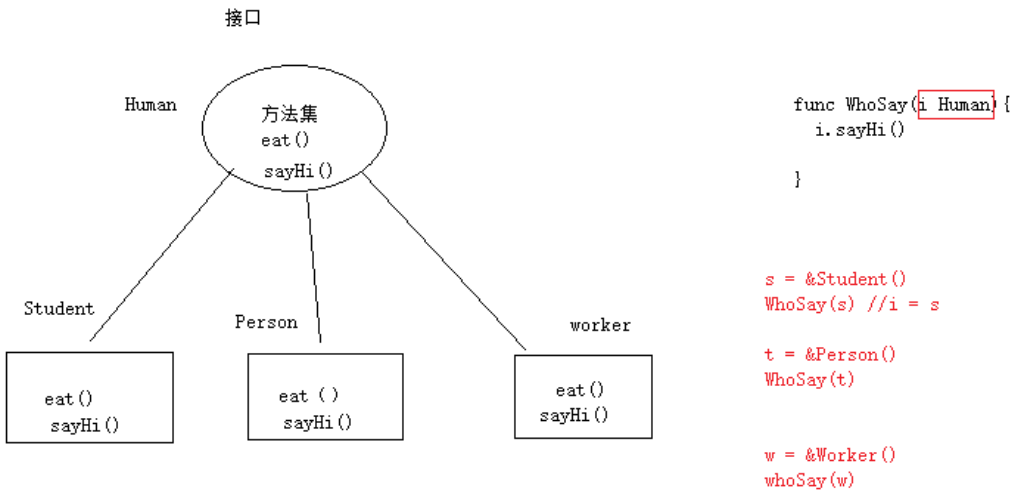
```
package main

import "fmt"

//定义一个接口类型
type Humaner interface {
    //方法，只有声明，没有实现，由别的类型（自定义类型）实现
    sayHi()
}

type Student9 struct {
   name string
   id  int
}
```

```go
func (temp *Student9) sayHi(){
    fmt.Printf("Student9[%s,%d] sayHi\n",temp.name,temp.id)
}

type Teacher struct {
    addr string
    id  int
}

func (temp *Teacher) sayHi()  {
   fmt.Printf("Teacher[%s,%d] sayHi\n",temp.addr,temp.id)
}

type Worker4 struct {
   id int
}

func (temp *Worker4) sayHi()  {
   fmt.Printf("Worker4[%d] sayHi\n",temp.id)
}

//定义一个普通函数，函数的参数为接口类型
//只有一个函数，可以表现有不同的形式，多态
func WhoSayHi(i Humaner){ //接口作为参数
   i.sayHi()
}


func main()  {

   //定义一个接口的类型
   //var i Humaner
   // 只有实现了此接口的方法的类型，那个这个类型的变量才能赋值给接口变量
   s := &Student9{"mike",7777}

   //i =s
   //i.sayHi()

   t := &Teacher{"bj",9999}
   //i = t

   //i.sayHi()

   w := &Worker4{8888}
   //i= w
   //i.sayHi()

   WhoSayHi(s)
   WhoSayHi(t)
   WhoSayHi(w)
```

```
}
```

切片接口的创建

```
//创建一个切片,切片放入是接口
 x := make([]Humaner,3)
 x[0] = s
 x[1]= t
 x[2] = w

 for _,i :=range x{
   i.sayHi()
 }
```



接口的继承

```
package main

import "fmt"

type Humaner01 interface {
   sayHi()
}

type Personer interface {
   Humaner01 //匿名字段，继承sayHi()
   sing(lrc string)
}

type Student10 struct {
   name string
   id  int
```

```
}

func (temp *Student10) sayHi(){
   fmt.Printf("Student10[%s,%d] sayHi\n",temp.name,temp.id)
}

func (temp *Student10)sing(lrc string)  {
   fmt.Println("student10在唱：",lrc)
}

func main(){
   //定义一个接口类型的变量
   var i Personer
   s := &Student10{"mike",7777}
   i = s
   i.sayHi()
   i.sing("人民解放歌")

   var h Humaner01
   h = s
   h.sayHi()

}
```

接口的转换

```
func main(){
   //定义一个接口类型的变量
   var i Personer
   s := &Student10{"mike",7777}
   i = s
   i.sayHi()
   i.sing("人民解放歌")

   var h Humaner01
   h = s
   h.sayHi()

   h = i //子接口可以赋值给父接口
   h.sayHi()

   //i = h //父接口定义变量不能给子接口

}
```

空接口

空接口万能类型，保存任意类型的值

```
package main

import "fmt"

func xxx(arg  ...interface{}){ //interface{} 空接口
    for i,k :=range  arg{
       fmt.Println(i,k)
   }
}

func main(){
   //空接口万能类型，保存任意类型的值
   var i interface{} = 1
    fmt.Println("i=",i)
    i = "abc"
    fmt.Println("i=",i)
    xxx(i)
}
```

类的断言

类型的断言：interface 是什么类型

```
package main

import "fmt"

type Student11 struct {
    name string
    id  int
}

func main(){
   i := make([]interface{},3)
   i[0] = 1
   i[1]="hello go"
   i[2]=Student11{"mike",7777}

   //类型的查询，类型的断言
    //index 是返回下标  data 对应的值 data【0】 data【1】 data【2】
   for index,data :=range  i{
```

```
        //第一个返回时值，ok返回时真还是假
        if value,ok := data.(int);ok==true{
            fmt.Printf("i[%d] 类型为int，内容为%d\n",index,value)
        }else if value,ok:=data.(string);ok==true{
            fmt.Printf("i[%d] 类型为string，内容为%s\n",index,value)
        }else  if value,ok:=data.(Student11);ok==true{
            fmt.Printf("i[%d]类型为student，内容为
name=%s,id=%d\n",index,value.name,value.id)
        }
    }

}
```

类型的断言 switch版本

```
for index,data := range i{
    switch value :=data.(type) {
    case int:
        fmt.Printf("i[%d] 类型为int，内容为%d\n",index,value)
    case string:
        fmt.Printf("i[%d] 类型为string，内容为%s\n",index,value)
    case Student11:
        fmt.Printf("i[%d]类型为student，内容为name=%s,id=%d\n",index,value.name,value.id)
    }
}
```

错误接口的使用

```
import (
    "fmt"
    "github.com/kataras/iris/core/errors"
)

func main()  {
     //var err1 error =fmt.Errorf("%s","this is normol err")
     err1 :=fmt.Errorf("%s","this is normol err")
     fmt.Println("err1=",err1)

     err2 := errors.New("this is normal err2")
     fmt.Println(err2.Message)
}
```

错误接口的应用

```
import (
   "github.com/kataras/iris/core/errors"
   "fmt"
)

func MyDiv(a ,b int)(result int,err error)  {
    err = nil
    if b == 0{
       err = errors.New("分母不为零")// fmt.Errorf("%s","分母不为零")
   }else{
      result  = a / b
   }
   return
}

func main(){
   result,err := MyDiv(10,0)
   if err !=nil{
        fmt.Println("err",err)
   }else {
       fmt.Println("result=",result)
   }
}
```

panic 函数会导致程序崩溃

```
package main

import "fmt"

func testa(){
   fmt.Println("aaaaaaaaaaaaaaaaaa")
}

func testb(){
   fmt.Println("bbbbbbbbbbbbbbbbb")
   //显示调用panic函数，导致程序的中断
   panic("this is a panic test")
   /* a := 10
   b :=0
   i := a/b //当分母为0的时候，产生一个panic。导致程序崩溃
   fmt.Println(i)*/
}

func testc(){
   fmt.Println("cccccccccccccccccc")
```

```
}

func main(){
   testa()
   testb()
   testc()

}
```

recover恢复程序

```
package main


import "fmt"

func testa1(){
   fmt.Println("aaaaaaaaaaaaaaaaaa")
}

func testb1(x int){
   defer func() {
      //recover() //通过revocer捕获错误信息，继续往后执行
      //fmt.Println(recover())
      if err:=recover();err !=nil{ //打印出产生的异常
         fmt.Println(err)
      }
   }() //别忘了调用此函数


   var a [10]int
   a[x] = 1111 //当x为20 时候，数组越界，产生一个panic，导致程序崩溃



}

func testc1(){
   fmt.Println("cccccccccccccccccc")
}

func main(){
   testa1()
   testb1(20)
   testc1()

}
```