面向对象编程——结构体和方法

目录：

一、面向对象思想的概述

（一）、OOP概述

- 面向对象的程序设计（Object Oriented Programming）
- 面向对象：关注的是对象。传统面向对象两大核心概念是类（class）和对象（object）。
- 面向过程：关注的是过程。

（二）、以吃饭为例，对比面向过程与面向对象思想的不同。

1、面向过程思路：

2、面向对象思路：

面向过程						
穿衣	→	穿正装	→	穿便装	→	化
出行	→	自驾	→	骑车	→	步
找饭店	→	高档西餐厅	→	中档湘菜馆	→	低
点菜	→	海鲜大餐	→	家常菜系	→	街
上菜	→	饭店加工饭菜，上菜	→	饭店加工饭菜，上菜	→	加
用餐	→	客人用刀叉用餐	→	客人用消毒碗筷用餐	→	用
结账	→	付费、会员打折、返券	→	非会员、结账	→	非

面向对象	衣服	交通出行
穿衣	1、衣服种类（正装、便装、休闲装） 2、穿衣者	1、交通方式（步行、骑车、自驾、打车） 2、乘客 3、司机
出行	穿衣	行驶
找饭店		
点菜		
上菜		
用餐		
结账		

3、为什么要面向对象?

接地气的说法，最大的感受是：

- 好起名字。不同的类中方法名字可以相同，但是函数则不同，必须使用不同的名字，光起函数名字就需要大费力气；
- 代码管理方便，易于模块化开发。
  - 类中包含方法和属性，属性用来记录数据，方法表示行为，类实例化后为对象；函数也表示行为，但是与函数配合的数据却散落摆放，缺乏类这样的结构来统一管理。
  - 对象既表示行为，又记录数据，行为和数据由对象来统一管理。写出来的代码方法与属性各归各类，代码逻辑清晰，阅读方便，方便也管理，利于扩展，易于模块化开发。
- 代码重用性高。面向对象的代码在使用时，通过调用各个对象中的方法和属性，不同的排列组合就能适应各种不同的业务场景。代码冗余量小，重用性高。

4、传统面向对象三大特性：

- 封装
- 继承
- 多态

（二）、面向对象的思想：

1、在生活中，随处可见的一种事物就是对象，如人、动植物、建筑等。这些对象都具备了属性和行为两大特征。

- 属性，如一个人，有年龄、性别、爱好、职业等属性特征；
- 行为即动态特征，比如教师有讲课行为，保安有负责安保的具体行为。
- 基于这两个特征，对象实现了记录数据及通过行为操作数据的结合，于是构成了多样的世界。

2、在软件领域，编程初期是面向过程编程，项目一旦庞大就变的不可控，最大的原因就是代码不可复用。

- 例如一个软件需求，需要创建游戏角色，这些角色需要有自己的名称、积分、血值、装备等属性，同时需要有攻击，复活等游戏行为，对于这些游戏属性可以用程序中的数据变量表示，对于游戏行为使用函数可以解决。但是数据和函数的代码散落摆放，一旦复用完全不可控。
- 于是软件有了面向对象的编程思想，游戏角色即被看成一个对象，这个角色封装了它应有的属性和具体的行为。

3、在其他编程语言中大多都使用class关键字来定义封装对象，表示该类的具体特征，而在Go语言中则使用struct关键字单独来完成各种数据的封装，行为操作则指引外部的具体函数执行，这种方式不但完成了其他语言面向对象所实现的效果，而且更为灵活。

（三）、Go语言面向对象

1、Go并不是一个纯面向对象的编程语言。在Go中的面向对象，结构体替代了类。

- Go并没有提供类class，但是它提供了结构体struct，方法method，可以在结构体上添加。提供了捆绑数据和方法的行为，这些数据和方法与类类似。

2、Go语言设计的非常简洁优雅，Go没有沿袭传统面向对象编程中的诸多概念，比如继承、虚方法、构造方法和析构方法、this等。

- Go不支持继承，尽管匿名字段的内存和行为类似继承，但是它不是继承。

3、尽管Go语言没有继承和多态。但是通过别的方式实现：

- 继承：通过匿名字段实现
- 多态：通过接口实现。

4、Go语言中学习面向对象，主要学习结构体struct、方法method、接口interface。

二、结构体

（一）、定义结构体

1、什么是结构体

Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。 结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。

2、结构体的定义格式

```
type 类型名 struct {
        成员属性1  类型1
        成员属性2  类型2
        成员属性3 , 成员属性4  类型3
         ...
}
```

- 类型名：标识结构体的名称，在同一个包内不能重复。
- 结构体中属性，也叫字段必须唯一。
- 同类型的成员属性可以写在一行。

【回顾】：

- var 变量名 数据类型
- const 变量名 数据类型=赋值
- func 函数名 函数体{}
- type 结构体 struct{}
  - type用于给类型起别名
  - 给整个函数起type，简化代码
  - 定义结构体
  - 定义自定义类型

3、实例代码

```
//定义一个结构体
type Teacher struct {
  name string
  age  int8
  sex  byte
}
```

（二）、实例化结构体——为结构体分配内存并初始化

- 什么是实例化?
  - 结构体的定义只是一种内存布局的描述，只有当结构体实例化时，才会真正分配内存。因此必须在定义结构体并实例化后才能使用结构体；
  - 实例化就是根据结构体定义的格式创建一份与格式一致的内存区域。结构体实例之间的内存是完全独立的。

1、var声明方式实例化结构体，初始化方式为：对象.属性=值

```
var p1 Teacher
p1.name = "Steven"
p1.age = 35
p1.sex = 1
```

2、变量简短声明格式实例化结构体，初始化方式为：对象.属性=值

```
p2 := Teacher{}
p2.name = "David"
```

```
p2.age = 33
p2.sex = 1
```

3、变量简短声明格式实例化结构体，声明时初始化。初始化方式为：属性:值 。属性:值可以同行，也可以换行。（类似map的用法）

```
p3 := Teacher{
    name: "Josh",
    age:  28,
    sex:  1,
}
```

或者： p3 = Teacher{name: "Josh2", age: 28, sex: 1}

4、变量简短声明格式实例化结构体，声明时初始化，不写属性名，按属性顺序只写属性值

```
p4 := Teacher{"Ruby", 30, 0}
```

5、创建指针类型的结构体

- 使用内置函数new()对结构体进行实例化，结构体实例化后形成指针类型的结构体。
- new内置函数会分配内存。第一个参数是类型，而不是值，返回的值是指向该类型新分配的零值的指针。该函数用于创建某个类型的指针。

```
p5 := new(Teacher)
(*p5).name = "Running"
(*p5).age = 31
p5.sex = 0 //语法简写形式，语法糖
```

（三）、结构体中的语法糖

1、语法糖的概念（Syntactic sugar）

- 语法糖，也译为糖衣语法，是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。
- 通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

- 结构体和数组中都含有语法糖。

2、示例代码

```
package main

import (
  "fmt"
)

//定义一个结构体
type Emp struct {
  name string
  age  int8
  sex  byte
}

func main() {
  //new内置函数声明结构体。
  emp1 := new(Emp)
  fmt.Printf("emp1: %T,  %v , %p \n", emp1, emp1, emp1)

  (*emp1).name = "David"
  (*emp1).age = 30
  (*emp1).sex = 1

  //语法简写形式，语法糖
  emp1.name = "Steven"
  emp1.age = 35
  emp1.sex = 1

  fmt.Println(emp1)
  fmt.Println("----------------------")

  SyntacticSugar()
```

```
}

//数组中语法糖
func SyntacticSugar() {
  arr := [4]int{1, 2, 3, 4}
  arr2 := &arr
  fmt.Println((*arr2)[3])
  fmt.Println(arr2[3])

  //切片中有语法糖吗?
  arr3 := []int{10, 20, 30, 40}
  arr4 := &arr3
  fmt.Println((*arr4)[3])
  //fmt.Println(arr4[3])
}
```

（四）、结构体是值类型

1、示例代码

```
package main

import (
  "fmt"
)

type Human struct {
  name string
  age  int8
  sex  byte
}

func main() {
  //1、初始化Human
  h1 := Human{"Steven", 35, 1}
```

```
    fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)
    fmt.Println("----------------------")

    //将结构体对象进行拷贝
    h2 := h1
    h2.name = "David"
    h2.age = 30
    fmt.Printf("h2修改后=%T , %v , %p \n", h2, h2, &h2)
    fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)
    fmt.Println("----------------------")


    //将结构体对象作为参数传递
    changeName(h1)
    fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)
    fmt.Println("----------------------")

    //changeName2(&h1)
    //fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)
}

//传对象
func changeName(h Human) {
    h.name = "Daniel"
    h.age = 13
    fmt.Printf("函数内h修改后=%T , %v , %p \n", h, h, &h)
}
```

（五）、结构体的深拷贝和浅拷贝

- 值类型是深拷贝
- 引用类型是浅拷贝

1、示例代码

```go
package main

import (
  "fmt"
)

type Dog struct {
  name  string
  color string
  age   int8
  kind  string //品种
}

func main() {

  //1、实现结构体的深拷贝
  //struct的数据类型：值类型，所以默认的复制就是深拷贝
  d1 := Dog{"豆豆", "黑色", 2, "二哈"} //Dog
  fmt.Printf("d1：%T , %v , %p \n", d1, d1, &d1)
  d2 := d1 //深拷贝  dog
  fmt.Printf("d2：%T , %v , %p \n", d2, d2, &d2)

  //修改d2，d1是否也发生变化?
  d2.name = "毛毛"
  fmt.Println("d2修改后=", d2)
  fmt.Println("d1=", d1)
  fmt.Println("----------------------")

  //2、实现结构体的浅拷贝：直接拷贝指针地址实现浅拷贝
  d3 := &d1
  fmt.Printf("d3：%T , %v , %p \n", d3, d3, d3)
  d3.kind = "萨摩耶"
  d3.color = "白色"
  d3.name = "球球"
```

```
    fmt.Println("d3修改后=", d3)
    fmt.Println("d1=", d1)
    fmt.Println("----------------------")

    //3、实现结构体的浅拷贝
    //拷贝通过new函数实例化的对象
    d4 := new(Dog) //*Dog
    d4.name = "多多"
    d4.color = "棕色"
    d4.age = 1
    d4.kind = "巴哥犬"
    d5 := d4 //*Dog
    fmt.Printf("d4：%T , %v , %p \n", d4, d4, d4)
    fmt.Printf("d5：%T , %v , %p \n", d5, d5, d5)

    //修改d2，d1是否也发生变化?
    d5.color = "金色"
    d5.kind = "金毛"
    fmt.Println("d5修改后=", d5)
    fmt.Println("d4=", d4)
    fmt.Println("----------------------")
}
```

（六）、结构体对象或指针作为函数的参数及函数返回值

- 结构体对象作为函数参数与结构体指针作为函数参数的不同?
- 结构体对象作为函数返回值与结构体指针作为函数返回值的不同?

1、示例代码

```
package main
import (
  "fmt"
)
type Flower struct {
  name  string
```

```
    color string
}

func main() {
    //1、测试结构体作为参数
    f1 := Flower{"玫瑰", "红"}
    fmt.Printf("f1：%T , %v , %p \n", f1, f1, &f1)
    fmt.Println("----------------------")

    //将结构体对象作为参数传递
    changeInfo1(f1)
    fmt.Printf("f1：%T , %v , %p \n", f1, f1, &f1)
    fmt.Println("----------------------")
    //将结构体指针作为参数传递
    changeInfo2(&f1)
    fmt.Printf("f1：%T , %v , %p \n", f1, f1, &f1)
    fmt.Println("----------------------")

    //2、测试结构体作为返回值
   //结构体对象作为返回值
    f2 := getFlower1()
    f3 := getFlower1()
    fmt.Println(f2, f3)
    f2.name = "杏花"
    fmt.Println(f2, f3)
    fmt.Printf("f2：%T , %v , %p \n", f2, f2, &f2)
    fmt.Printf("f3：%T , %v , %p \n", f3, f3, &f3)
    fmt.Println("----------------------")
   //结构体指针作为返回值
    f4 := getFlower2()
    f5 := getFlower2()
    fmt.Println(f4, f5)
    f4.name = "桃花"
    fmt.Println(f4, f5)
```

```
    fmt.Printf("f4：%T , %v , %p \n", f4, f4, f4)
    fmt.Printf("f5：%T , %v , %p \n", f5, f5, f5)
    fmt.Println("----------------------")
}

//传结构体对象
func changeInfo1(f Flower) {
    f.name = "月季"
    f.color = "粉"
    fmt.Printf("函数内f修改后=%T , %v , %p \n", f, f, &f)
}

//传对象指针
func changeInfo2(f *Flower) {
    f.name = "蔷薇"
    f.color = "紫"
    fmt.Printf("函数内f修改后=%T , %v , %p \n", f, f, f)
}
//返回结构体对象
func getFlower1() (f Flower) {
    f = Flower{"牡丹", "白"}
    return
}
//返回结构体指针
func getFlower2() (f *Flower) {
    f = &Flower{"芙蓉", "红"}
    return
}
```

（七）、匿名结构体和匿名字段

1、匿名结构体

（1）、概念

- 没有名字的结构体。无需通过type关键字定义就可以直接使用。

- 在创建匿名结构体时，同时要创建对象。
- 匿名结构体由结构体定义和键值对初始化两部分组成。
- 语法格式：

变量名 := struct {
    //定义成员属性
} {//初始化成员属性}

（2）、示例代码

```
package main

import (
  "fmt"
  "math"
)

func main() {
  //匿名函数
  res := func(a, b float64) float64 {
    return math.Pow(a, b)
  }(2, 3)
  fmt.Println(res)

  // 匿名结构体
  addr := struct {
    province,city string
  }{"陕西省" , "西安市"}
  fmt.Println(addr)

  cat := struct {
    name,color string
    age int8
  } {
    name:"绒毛",
    color:"黑白",
```

```
    age:1,
  }
  fmt.Println(cat)
}
```

2、结构体的匿名字段

（1）、概念

- 结构体中的字段没有名字，只包含一个没有字段名的类型。这些字段被称为匿名字段。
- 如果字段没有名字，那么默认使用类型作为字段名。
- 注意：同一个类型只能写一个。
- 结构体嵌套中采用匿名结构体字段可以模拟继承关系。

（2）、示例代码

```
type User struct {
  //name string
  //sex byte
  //age int8
  //height float64
  string
  byte
  int8
  float64
}

func main() {
  //实例化结构体
  user := User{"Steven", 'm',35, 177.5}
  fmt.Println(user)
  //如果想依次获得姓名、年龄、身高可以写成：
  fmt.Printf("姓名：%s , 性别：%c , 身高：%.2f ， 年龄：%d \n",
user.string, user.byte , user.float64, user.int8)
}
```

（八）、结构体嵌套

1、概念

- 将一个结构体作为另一个结构体的属性（字段），这种结构就是结构体嵌套。
- 结构体嵌套可以模拟面向对象中的两种关系：
  - 聚合关系：一个类作为另一个类的属性
  - 继承关系：一个类作为另一个类的子类。子类和父类。

2、示例代码

（1）、结构体嵌套模拟聚合关系

- 示例代码

```
package main
import (
  "fmt"
)

type Address struct {
  province, city string
}

type Person struct {
  name    string
  age     int
  address Address
}

func main() {
  //模拟对象之间的聚合关系
  p := Person{}
  p.name = "Steven"
  p.age = 35


  //赋值方式1
  addr := Address{}
```

```
addr.province = "北京市"
addr.city = "海淀区"
p.address = addr

fmt.Println(p)
fmt.Println("姓名:", p.name)
fmt.Println("年龄:", p.age)
fmt.Println("省:", p.address.province)
fmt.Println("市:", p.address.city)
fmt.Println("----------")

//修改Person对象的数据，是否影响Address对象?
p.address.city = "昌平区"
fmt.Println("姓名:", p.name)
fmt.Println("年龄:", p.age)
fmt.Println("省:", p.address.province)
fmt.Println("市:", p.address.city)
fmt.Println("----------")
fmt.Println("市:", addr.city) //没有影响

//修改Address对象的数据，是否影响Person对象?
addr.city = "大兴区"
fmt.Println("姓名:", p.name)
fmt.Println("年龄:", p.age)
fmt.Println("省:", p.address.province)
fmt.Println("市:", p.address.city)
fmt.Println("----------")


//赋值方式2：
p.address = Address{
  province: "陕西省",
  city:    "西安市",
}
```

```
    fmt.Println(p)
    fmt.Println("姓名:", p.name)
    fmt.Println("年龄:", p.age)
    fmt.Println("省:", p.address.province)
    fmt.Println("市:", p.address.city)
    fmt.Println("----------------------")
}
```

（2）、结构体嵌套模拟继承关系

- 继承是传统面向对象编程中三大特征之一。用于描述两个类之间的关系。一个类（子类、派生类）继承于另一个类（父类、超类）。
- 子类可以有自己的属性和方法，也可以重写父类已有的方法。
- 子类可以直接访问父类所有的属性和方法。
- 提升字段：
  - 在结构体中属于匿名结构体的字段称为提升字段，因为它们可以被访问，就好像它们属于拥有匿名结构字段的结构一样。
  - 换句话说，父类中的字段就是提升字段。
- 继承的意义：
  - 避免重复代码
  - 扩展类的功能
- 采用匿名字段的形式就是模拟继承关系。而模拟聚合关系时一定要采用有名字的结构体作为字段。
-
- 示例代码

```
package main
import (
  "fmt"
)

type Person struct {
  Name string
  Age  int
  Sex  string
}
```

```go
type Student struct {
  Person //采用匿名结构体字段模拟继承关系
  SchoolName string
}

func main() {
  //1、初始化Person
  p1 := Person{"Steven", 35, "男"}
  fmt.Println(p1)
  fmt.Printf("p1: %T , %+v \n", p1, p1)
  fmt.Println("----------------------")

  //2、初始化Student
  //写法1：
  s1 := Student{p1, "北航软件学院"}
  fmt.Println(s1)
  fmt.Printf("s1: %T , %+v \n", s1, s1)
  fmt.Println("----------------------")

  //写法2：
  s2 := Student{Person{"Josh", 30, "男"}, "北外高翻学院"}
  fmt.Println(s2)
  fmt.Printf("s2: %T , %+v \n", s2, s2)
  fmt.Println("----------------------")

  //写法3：
  s3 := Student{Person: Person{
    Name: "Penn",
    Age:  19,
    Sex:  "男",
  },
    SchoolName: "北大元培学院",
  }
  fmt.Println(s3)
  fmt.Printf("s3: %T , %+v \n", s3, s3)
  fmt.Println("----------------------")
```

```
    //写法4：
    s4 := Student{}
    s4.Name = "Daniel"
    s4.Sex = "男"
    s4.Age = 12
    s4.SchoolName = "十一龙樾"
    fmt.Println(s4)
    fmt.Printf("s4: %T , %+v \n", s4, s4)
    fmt.Println("----------------------")
}
```

(3)、结构体嵌套中成员名字冲突

- 结构体嵌套时，可能拥有相同的成员名，成员重名会发生什么？
- 示例代码：

```
package main
import "fmt"
type A struct {
    a, b int
}

type B struct {
    a, d int
}

type C struct {
    A
    B
}

func main() {
    c := C{}
    c.A.a = 1
    c.B.a = 2 //如果调用c.a = 2，则会提示“引起歧义的参数。”
    c.b = 3
    c.d = 4
```

```
    fmt.Println(c)
}
```

- 当重名时，编译器会报错：Ambiguous reference。

三、方法

（一）、什么是方法?

- Go语言同时有函数和方法，方法的本质是函数，但是方法和函数又具有不同点。

1、含义不同

- 函数function是一段具有独立功能的代码，可以被反复多次调用，从而实现代码复用。而方法method是一个类的行为功能，只有该类的对象才能调用。

2、方法有接受者，而函数无接受者

- Go语言的方法method是一种作用于特定类型变量的函数。这种特定类型变量叫做Receiver（接受者、接收者、接收器）。
- 接受者的概念类似于传统面向对象语言中的this或self关键字。
- Go语言的接受者强调了方法具有作用对象，而函数没有作用对象。
- 一个方法就是一个包含了接受者的函数。
- Go语言中，接受者的类型可以是任何类型，不仅仅是结构体，也可以是struct类型外的其他任何类型。

3、函数不可以重名，而方法可以重名

- 只要接受者不同，则方法名可以一样。

（二）、方法的语法格式：

```
func （接受者变量  接受者类型） 方法名(参数列表)（返回值列表） {
    //方法体
}
```

- 接受者在func关键字和方法名之间编写，接受者可以是struct类型或非struct类型，可以是指针类型和非指针类型。
- 接受者中的变量在命名时，官方建议使用接受者类型的第一个小写字母。
- 实例代码：

```
package main
```

```go
import "fmt"

type Employee struct {
  name, currency string
  salary         int
}

func main() {
  emp1 := Employee{
    name:     "Daniel Wang",
    salary:   2000,
    currency: "$",
  }
  //调用方法
  emp1.displaySalary()
  //调用函数
  displaySalary(emp1)
}

//displaySalary() 方法，接受者类型为Employee
func (e Employee) displaySalary() {
  fmt.Printf("员工姓名：%s ，薪资： %s%d \n", e.name, e.currency, e.salary)
}

//displaySalary() 函数，参数为Employee类型对象
func displaySalary(e Employee) {
  fmt.Printf("员工姓名：%s ，薪资： %s%d \n", e.name, e.currency, e.salary)
}
```

（三）、方法和函数

- 既然可以用函数来写相同的程序，为什么还要使用方法?
- 有以下两个原因
  - Go不是一种纯粹面向对象的编程语言，它不支持类。因此其方法是一种实现类似于类的行为的方法。
  - 相同名称的方法可以在不同的类型上定义，而具有相同名称的函数是不允许的。假设我们有一个正方形和圆形的结构。可以在正方形和圆形上定义一个名为Area的求取面积的方法。

（四）、可以定义相同的方法名

1、虽然method的名字一模一样，但是如果接受者不一样，那么method就不一样

- method里面可以访问接受者的字段
- 调用method通过.访问，就像struct里面访问字段一样

2、示例代码

```
package main
import (
  "fmt"
  "math"
)

type Rectangle struct {
  width, height float64
}

type Circle struct {
  radius float64
}

func main() {
  r1 := Rectangle{10, 4}
  r2 := Rectangle{12, 5}
  c1 := Circle{1}
  c2 := Circle{10}
  fmt.Println("r1的面积：", r1.area())
  fmt.Println("r2的面积：", r2.area())
  fmt.Println("c1的面积：", c1.area())
  fmt.Println("c2的面积：", c2.area())
}

//该 method 属于 Rectangle类型的对象
func (r Rectangle) area() float64 {
  return r.width * r.height
}
```

```
//该 method 属于 Circle 类型的对象
func (c Circle) area() float64 {
  return c.radius * c.radius * math.Pi
}
```

运行结果

r1的面积： 40

r2的面积： 60

c1的面积： 3.141592653589793

c2的面积： 314.1592653589793

（五）、指针作为接受者

1、若接受者不是指针，实际只是获取了一个copy，而不能真正改变接受者中原来的数据

2、示例代码

```
package main

import (
  "fmt"
)

type Rectangle struct {
  width, height float64
}

func main() {
  r1 := Rectangle{5, 8}
  r2 := r1

  //打印内存地址
  fmt.Printf("r1的地址： %p \n", &r1)
  fmt.Printf("r2的地址： %p \n", &r2)

  r1.setVal()
  fmt.Println("r1.height=", r1.height)
  fmt.Println("r2.height=", r2.height)
```

```
    fmt.Println("--------------------")

    r1.setVal2()
    fmt.Println("r1.height=", r1.height)
    fmt.Println("r2.height=", r2.height)
}

func (r Rectangle) setVal() {
    fmt.Printf("setVal()方法中r的地址：%p \n", &r)
    r.height = 10
}

func (r *Rectangle) setVal2() {
    fmt.Printf("setVal2()方法中r的地址：%p \n", r)
    r.height = 20
}
```

运行结果：

r1的地址：0xc420014050
r2的地址：0xc420014060
setVal()方法中r的地址：0xc420014080
r1.height= 8
r2.height= 8
----
setVal2()方法中r的地址：0xc420014050
r1.height= 20
r2.height= 8

（六）、method继承

1、method是可以继承的，如果匿名字段实现了一个method，那么包含这个匿名字段的struct也能调用该匿名结构体中的method

2、示例代码

```
package main

import "fmt"

type Human struct {
```

```
    name, phone string
    age        int
}
type Student struct {
    Human //匿名字段
    school string
}
type Employee struct {
    Human //匿名字段
    company string
}

func main() {
    s1 := Student{Human{"Daniel", "15012345678", 13}, "十一中学"}
    e1 := Employee{Human{"Steven", "17812345678", 35}, "1000phone"}

    s1.SayHi()
    e1.SayHi()
}

func (h *Human) SayHi() {
    fmt.Printf("大家好!我是 %s ， 我%d岁，我的联系方式是：%s\n", h.name, h.age ,
h.phone)
}
```

运行结果

大家好!我是 Daniel ， 我13岁，我的联系方式是：15012345678
大家好!我是 Steven ， 我35岁，我的联系方式是：17812345678

（七）、method重写

1、方法是可以继承和重写的。存在继承关系时，按照就近原则，进行调用

2、示例代码

```
package main

import "fmt"

type Human struct {
  name, phone string
```

```
  age        int
}
type Student struct {
  Human //匿名字段
  school string
}
type Employee struct {
  Human //匿名字段
  company string
}

func main() {
  s1 := Student{Human{"Daniel", "15012345678", 13}, "十一中学"}
  e1 := Employee{Human{"Steven", "17812345678", 35}, "1000phone"}

  s1.SayHi()
  e1.SayHi()
}

func (h *Human) SayHi() {
  fmt.Printf("大家好! 我是 %s ， 我%d岁，我的联系方式是：%s\n", h.name, h.age,
h.phone)
}

//Student的method重写Human的method
func (s *Student) SayHi() {
  fmt.Printf("大家好! 我是 %s ， 我%d岁，我在%s上学，我的联系方式是：%s\n", s.name,
s.age, s.school, s.phone)
}

//Employee的method重写Human的method
func (e *Employee) SayHi() {
  fmt.Printf("大家好! 我是 %s ， 我%d岁，我在%s工作，我的联系方式是：%s\n", e.name,
e.age, e.company, e.phone)
}
```

运行结果

大家好! 我是 Daniel ， 我13岁，我在十一中学上学，我的联系方式是：15012345678

大家好! 我是 Steven ， 我35岁， 我在1000phone工作， 我的联系方式是: 17812345678