Golang 中的结构体

主讲教师：（大地）
合作网站：www.itying.com （IT 营）
我的专栏：https://www.itying.com/category-79-b0.html

一、关于 Golang 结构体

Golang 中没有“类”的概念，Golang 中的结构体和其他语言中的类有点相似。和其他面向对象语言中的类相比，Golang 中的结构体具有更高的扩展性和灵活性。

Golang 中的基础数据类型可以表示一些事物的基本属性，但是当我们想表达一个事物的全部或部分属性时，这时候再用单一的基本数据类型就无法满足需求了，Golang 提供了一种自定义数据类型，可以封装多个基本数据类型，这种数据类型叫结构体，英文名称 struct。也就是我们可以通过 struct 来定义自己的类型了。

二、Golang type 关键词自定义类型和类型别名

Golang 中通过 type 关键词定义一个结构体，在讲解结构体之前，我们首先给大家看看通过 type 自定义类型以及定义类型别名。

1、自定义类型

在 Go 语言中有一些基本的数据类型，如 string、整型、浮点型、布尔等数据类型， Go 语言中可以使用 type 关键字来定义自定义类型。

```
type  myInt  int
```

上面代码表示：将 myInt 定义为 int 类型，通过 type 关键字的定义，myInt 就是一种新的类型，它具有 int 的特性。

2、类型别名

Golang1.9 版本以后添加的新功能。

类型别名规定：TypeAlias 只是 Type 的别名，本质上 TypeAlias 与 Type 是同一个类型。就像一个孩子小时候有大名、小名、英文名，但这些名字都指的是他本人。

```
type TypeAlias = Type
```

我们之前见过的 rune 和 byte 就是类型别名，他们的底层定义如下：

```
type byte = uint8
type rune = int32
```

3、自定义类型和类型别名的区别

类型别名与自定义类型表面上看只有一个等号的差异，我们通过下面的这段代码来理解它们之间的区别。

```
package main
import "fmt"
//类型定义
type newInt int
//类型别名
type myInt = int
func main() {
	var a newInt
	var b myInt
	fmt.Printf("type of a:%T\n", a) //type of a:main.newInt
	fmt.Printf("type of b:%T\n", b) //type of b:int
}
```

结果显示 a 的类型是 main.newInt，表示 main 包下定义的 newInt 类型。b 的类型是 int 类型。

三、结构体定义初始化的几种方法

1、结构体的定义

使用 type 和 struct 关键字来定义结构体，具体代码格式如下：

```
type 类型名 struct {
    字段名 字段类型
    字段名 字段类型
    …
}
```

其中：

- **类型名**：表示自定义结构体的名称，在同一个包内不能重复。
- **字段名**：表示结构体字段名。结构体中的字段名必须唯一。
- **字段类型**：表示结构体字段的具体类型。

举个例子，我们定义一个 person（人）结构体，代码如下：

```
type person struct {
    name string
    city string
    age  int8
}
```

同样类型的字段也可以写在一行，

```
type person struct {
    name, city string
    age        int8
}
```

注意：结构体首字母可以大写也可以小写，大写表示这个结构体是公有的，在其他的包里面可以使用。小写表示这个结构体是私有的，只有这个包里面才能使用。

2、结构体实例化（第一种方法）

只有当结构体实例化时，才会真正地分配内存。也就是必须实例化后才能使用结构体的字段。

结构体本身也是一种类型，我们可以像声明内置类型一样使用 var 关键字声明结构体类型。

```
var 结构体实例 结构体类型
```

```
package main
import "fmt"
type person struct {
	name string
	city string
	age  int
}
func main() {
	var p1 person
	p1.name = "张三"
	p1.city = "北京"
	p1.age = 18
	fmt.Printf("p1=%v\n", p1)  //p1={张三 北京 18}
	fmt.Printf("p1=%#v\n", p1) //p1=main.person{name:"张三", city:"北京", age:18}
}
```

3、结构体实例化（第二种方法）

我们还可以通过使用 new 关键字对结构体进行实例化，得到的是结构体的地址。 格式如下：

```
package main

import "fmt"

type person struct {
	name string
	city string
	age  int
}

func main() {
	var p2 = new(person)
	p2.name = "张三"
	p2.age = 20
	p2.city = "北京"
	fmt.Printf("%T\n", p2)      //*main.person
	fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"张三", city:"北京", age:20}
}
```

从打印的结果中我们可以看出 p2 是一个结构体指针。

注意：在 Golang 中支持对**结构体指针**直接使用.来访问结构体的成员。p2.name = "张三" 其实在底层是(*p2).name = "张三"

4、结构体实例化（第三种方法）

使用&对结构体进行取地址操作相当于对该结构体类型进行了一次 new 实例化操作。

```
package  main

import  "fmt"

type  person  struct  {
      name  string
      city  string
      age    int
}

func  main()  {
      p3  :=  &person{}
      fmt.Printf("%T\n",  p3)        //*main.person
      fmt.Printf("p3=%#v\n",  p3)  //p3=&main.person{name:"",  city:"",  age:0}
      p3.name  =  "zhangsan"
      p3.age  =  30
      p3.city  =  "深圳"
      (*p3).age  =  40   //这样也是可以的
      fmt.Printf("p3=%#v\n",  p3)  //p3=&main.person{name:"zhangsan",  city:"深圳",  age:30}
}
```

5、结构体实例化（第四种方法） 键值对初始化

```
package main

import "fmt"

type person struct {

      name string

      city string

      age    int

}

func main() {
```

```
    p4 := person{
        name: "zhangsan",
        city: "北京",
        age:  18,
    }
    fmt.Printf("p4=%#v\n", p4) //p4=main.person{name:"zhangsan", city:"北京", age:18}
}
```

注意：最后一个属性的,要加上

6、结构体实例化（第五种方法） 结构体指针进行键值对初始化

```
package main

import "fmt"

type person struct {
    name string
    city string
    age  int
}

func main() {
    p5 := &person{
        name: "zhangsan",
        city: "上海",
        age:  28,
    }
    fmt.Printf("p5=%#v\n", p5) //p5=&main.person{name:"zhangsan", city:"上海", age:28}
}
```

当某些字段没有初始值的时候，这个字段可以不写。此时，没有指定初始值的字段的

```
package main

import "fmt"
```

```
type person struct {
	name string
	city string
	age  int
}
func main() {
	p6 := &person{
		city: "北京",
	}
	fmt.Printf("p6=%#v\n", p6) //p6=&main.person{name:"", city:"北京", age:0}
}
```

7、结构体实例化（第六种方法） 使用值的列表初始化

```
package main
import "fmt"
type person struct {
	name string
	city string
	age  int
}
func main() {
	// 初始化结构体的时候可以简写，也就是初始化的时候不写键，直接写值：
	p7 := &person{
		"zhangsan",
		"北京",
		28,
```

```
    }

    fmt.Printf("p7=%#v\n", p7) //p7=&main.person{name:"zhangsan", city:"北京", age:28}

}
```

使用这种格式初始化时，需要注意：

1.必须初始化结构体的所有字段。

2.初始值的填充顺序必须与字段在结构体中的声明顺序一致。

3.该方式不能和键值初始化方式混用。

四、结构体方法和接收者

在 go 语言中，没有类的概念但是可以给类型（结构体，自定义类型）定义方法。所谓方法就是定义了接收者的函数。接收者的概念就类似于其他语言中的 this 或者 self。

方法的定义格式如下：

```
func (接收者变量 接收者类型) 方法名(参数列表) (返回参数) {
    函数体
}
```

其中

- **接收者变量：**接收者中的参数变量名在命名时，官方建议使用接收者类型名的第一个小写字母，而不是 self、this 之类的命名。例如，Person 类型的接收者变量应该命名为 p，Connector 类型的接收者变量应该命名为 c 等。
- **接收者类型：**接收者类型和参数类似，可以是指针类型和非指针类型。
- **方法名、参数列表、返回参数：**具体格式与函数定义相同。

实例 1：给结构体 Person 定义一个方法打印 Person 的信息

```
package main
import "fmt"
type Person struct {
    name string
    age  int8
}
func (p Person) printInfo() {
    fmt.Printf("姓名:%v 年龄:%v", p.name, p.age)
```

```
}
func main() {
    p1 := Person{
        name: "小王子",
        age:  25,
    }
    p1.printInfo()
}
```

1、值类型的接收者

当方法作用于值类型接收者时，Go 语言会在代码运行时将接收者的值复制一份。在值类型接收者的方法中可以获取接收者的成员值，但修改操作只是针对副本，无法修改接收者变量本身。

2、指针类型的接收者

指针类型的接收者由一个结构体的指针组成，由于指针的特性，调用方法时修改接收者指针的任意成员变量，在方法结束后，修改都是有效的。这种方式就十分接近于其他语言中面向对象中的 this 或者 self。

```
package main
import "fmt"
type Person struct {
    name string
    age  int
}
//值类型接受者
func (p Person) printInfo() {
    fmt.Printf("姓名:%v 年龄:%v\n", p.name, p.age)
}
//指针类型接收者
func (p *Person) setInfo(name string, age int) {
    p.name = name
    p.age = age
}
func main() {
    p1 := Person{
        name: "小王子",
        age:  25,
    }
```

必须是指针类型的

```
    p1.printInfo()
    p1.setInfo("张三", 20)
    p1.printInfo()
}
```

五、给任意类型添加方法

在 Go 语言中，接收者的类型可以是任何类型，不仅仅是结构体，任何类型都可以拥有方法。举个例子，我们基于内置的 int 类型使用 type 关键字可以定义新的自定义类型，然后为我们的自定义类型添加方法。

```
package main
import "fmt"
type myInt int
func (m myInt) SayHello() {
    fmt.Println("Hello, 我是一个 int。")
}
func main() {
    var m1 myInt
    m1.SayHello() //Hello, 我是一个 int。
    m1 = 100
    fmt.Printf("%#v  %T\n", m1, m1) //100  main.MyInt
}
```

注意事项： 非本地类型不能定义方法，也就是说我们不能给别的包的类型定义方法。

六、结构体的匿名字段

结构体允许其成员字段在声明时没有字段名而只有类型，这种没有名字的字段就称为匿名字段。

```
//Person 结构体 Person 类型
type Person struct {
    string
    int
}
func main() {
    p1 := Person{
        "小王子",
        18,
```

```
    }
    fmt.Printf("%#v\n", p1)              //main.Person{string:"北京", int:18}
    fmt.Println(p1.string, p1.int) //北京 18
}
```

匿名字段默认采用类型名作为字段名，结构体要求字段名称必须唯一，因此一个结构体中同种类型的匿名字段只能有一个。

七、嵌套结构体

一个结构体中可以嵌套包含另一个结构体或结构体指针。

```
package main

import "fmt"

//Address 地址结构体

type Address struct {

    Province string

    City     string

}

//User 用户结构体

type User struct {

    Name    string

    Gender  string

    Address Address

}


func main() {

    user1 := User{

        Name:    "张三",
```

```
        Gender: "男",
        Address: Address{
            Province: "广东",
            City:     "深圳",
        },
    }
    fmt.Printf("user1=%#v\n", user1) //user1=main.User{Name:"张三", Gender:"男", Address:main.Address{Province:"广东", City:"深圳"}}
}
```

八、嵌套匿名结构体

```
package main
import "fmt"
//Address 地址结构体
type Address struct {
    Province string
    City     string
}
//User 用户结构体
type User struct {
    Name    string
    Gender  string
    Address //匿名结构体
}
func main() {
    var user2 User
    user2.Name = "张三"
```

```
    user2.Gender = "男"
    user2.Address.Province = "广东"      //通过匿名结构体.字段名访问
    user2.City = "深圳"                  //直接访问匿名结构体的字段名
    fmt.Printf("user2=%#v\n", user2) //user2=main.User{Name:"张三", Gender:"男", Address:main.Address{Province:"广东", City:"深圳"}}
}
```

注意：当访问结构体成员时会先在结构体中查找该字段，找不到再去匿名结构体中查找。

九、关于嵌套结构体的字段名冲突

嵌套结构体内部可能存在相同的字段名。这个时候为了避免歧义需要指定具体的内嵌结构体的字段。

```
package main
//Address 地址结构体
type Address struct {
    Province    string
    City        string
    CreateTime string
}
//Email 邮箱结构体
type Email struct {
    Account     string
    CreateTime string
}
//User 用户结构体
type User struct {
    Name    string
    Gender string
```

```
    Address

    Email

}

func main() {

    var user3 User

    user3.Name = "张三"

    user3.Gender = "男"

    // user3.CreateTime = "2020"              // ambiguous selector user3.CreateTime

    user3.Address.CreateTime = "2020" //指定 Address 结构体中的 CreateTime

    user3.Email.CreateTime = "2021"     //指定 Email 结构体中的 CreateTime

}
```

十、结构体的继承

Go 语言中使用结构体也可以实现其他编程语言中的继承。

```
package main

import "fmt"

//Animal  动物

type Animal struct {

    name string

}

func (a *Animal) run() {

    fmt.Printf("%s 会运动！ \n", a.name)

}

//Dog  狗

type Dog struct {

    Age        int8

    *Animal //通过嵌套匿名结构体实现继承
```

```
}

func (d *Dog) wang() {

	fmt.Printf("%s 会汪汪汪~\n", d.name)

}

func main() {

	d1 := &Dog{

		Age: 4,

		Animal: &Animal{ //注意嵌套的是结构体指针

			name: "阿奇",

		},

	}

	d1.wang() //乐乐会汪汪汪~

	d1.run()    //乐乐会动！

}
```