Go语言基础之结构体

2017年6月23日 | Golang |15223 阅读

Go语言中没有“类”的概念，也不支持“类”的继承等面向对象的概念。Go语言中通过结构体的内嵌再配合接口比面向对象具有更高的扩展性和灵活性。

类型别名和自定义类型

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。

自定义类型是定义了一个全新的类型。我们可以基于内置的基本类型定义，也可以通过struct定义。例如：

```
//将MyInt定义为int类型
type MyInt int
```

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。

类型别名

类型别名是 `Go1.9` 版本添加的新功能。

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小名、乳名，上学后用学名，英语老师又会给他起英文名，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，他们的定义如下：

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

类型定义和类型别名的区别

类型别名与类型定义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。

```
//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` 。 `MyInt` 类型只会在代码中存在，编译完成时并不会有 `MyInt` 类型。

结构体

Go语言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明显就无法满足需求了，Go语言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。

Go语言中通过 `struct` 来实现面向对象。

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：标识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 person1 struct {
    name, city string
    age        int8
}
```

这样我们就拥有了一个 `person` 的自定义类型，它有 `name` 、 `city` 、 `age` 三个字段，分别表示姓名、城市和年龄。这样我们使用这个 `person` 结构体就能够很方便的在程序中表示和存储人信息了。

语言内置的基础数据类型是用来描述一个值的，而结构体是用来描述一组值的。比如一个人有名字、年龄和居住城市等，本质上是一种聚合型的数据类型

结构体实例化

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
```

基本实例化

举个例子：

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

func main() {
    var p1 person
    p1.name = "沙河娜扎"
    p1.city = "北京"
    p1.age = 18
    fmt.Printf("p1=%v\n", p1)  //p1={沙河娜扎 北京 18}
    fmt.Printf("p1=%#v\n", p1) //p1=main.person{name:"沙河娜扎", city:"北京", age:18}
}
```

我们通过 `.` 来访问结构体的字段（成员变量）,例如 `p1.name` 和 `p1.age` 等。

匿名结构体

在定义一些临时数据结构等场景下还可以使用匿名结构体。

```
package main

import (
    "fmt"
)

func main() {
    var user struct{Name string; Age int}
    user.Name = "小王子"
    user.Age = 18
    fmt.Printf("%#v\n", user)
}
```

创建指针类型结构体

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：

```
var p2 = new(person)
fmt.Printf("%T\n", p2)     //*main.person
fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"", city:"", age:0}
```

从打印的结果中我们可以看出 `p2` 是一个结构体指针。

需要注意的是在Go语言中支持对结构体指针直接使用 `.` 来访问结构体的成员。

```
var p2 = new(person)
p2.name = "小王子"
p2.age = 28
p2.city = "上海"
fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"小王子", city:"上海", age:28}
```

取结构体的地址实例化

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。

```
p3 := &person{}
fmt.Printf("%T\n", p3)     //*main.person
fmt.Printf("p3=%#v\n", p3) //p3=&main.person{name:"", city:"", age:0}
p3.name = "七米"
p3.age = 30
p3.city = "成都"
fmt.Printf("p3=%#v\n", p3) //p3=&main.person{name:"七米", city:"成都", age:30}
```

`p3.name = "七米"` 其实在底层是 `(*p3).name = "七米"` ，这是Go语言帮我们实现的语法糖。

结构体初始化

没有初始化的结构体，其成员变量都是对应其类型的零值。

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

func main() {
    var p4 person
    fmt.Printf("p4=%#v\n", p4) //p4=main.person{name:"", city:"", age:0}
}
```

使用键值对初始化

使用键值对对结构体进行初始化时，键对应结构体的字段，值对应该字段的初始值。

```
p5 := person{
    name: "小王子",
    city: "北京",
    age:  18,
}
fmt.Printf("p5=%#v\n", p5) //p5=main.person{name:"小王子", city:"北京", age:18}
```

也可以对结构体指针进行键值对初始化，例如：

```
p6 := &person{
    name: "小王子",
    city: "北京",
    age:  18,
}
fmt.Printf("p6=%#v\n", p6) //p6=&main.person{name:"小王子", city:"北京", age:18}
```

当某些字段没有初始值的时候，该字段可以不写。此时，没有指定初始值的字段的值就是该字段类型的零值。

```
p7 := &person{
    city: "北京",
}
fmt.Printf("p7=%#v\n", p7) //p7=&main.person{name:"", city:"北京", age:0}
```

使用值的列表初始化

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：

```
p8 := &person{
    "沙河娜扎",
    "北京",
    28,
}
fmt.Printf("p8=%#v\n", p8) //p8=&main.person{name:"沙河娜扎", 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。

结构体内存布局

结构体占用一块连续的内存。

```
type test struct {
    a int8
    b int8
    c int8
    d int8
}
n := test{
    1, 2, 3, 4,
}
fmt.Printf("n.a %p\n", &n.a)
fmt.Printf("n.b %p\n", &n.b)
fmt.Printf("n.c %p\n", &n.c)
fmt.Printf("n.d %p\n", &n.d)
```

输出：

```
n.a 0xc0000a0060
n.b 0xc0000a0061
n.c 0xc0000a0062
n.d 0xc0000a0063
```

【进阶知识点】关于Go语言中的内存对齐推荐阅读:在 Go 中恰到好处的内存对齐

空结构体

空结构体是不占用空间的。

```
var v struct{}
fmt.Println(unsafe.Sizeof(v))  // 0
```

面试题

请问下面代码的执行结果是什么？

```
type student struct {
    name string
    age  int
}

func main() {
    m := make(map[string]*student)
    stus := []student{
        {name: "小王子", age: 18},
        {name: "娜扎", age: 23},
        {name: "大王八", age: 9000},
    }

    for _, stu := range stus {
        m[stu.name] = &stu
    }
    for k, v := range m {
        fmt.Println(k, "=>", v.name)
    }
}
```

构造函数

Go语言的结构体没有构造函数，我们可以自己实现。 例如，下方的代码就实现了一个 `person` 的构造函数。 因为 `struct` 是值类型，如果结构体比较复杂的话，值拷贝性能开销会比较大，所以该构造函数返回的是结构体指针类型。

```
func newPerson(name, city string, age int8) *person {
    return &person{
        name: name,
        city: city,
        age:  age,
    }
}
```

调用构造函数

```
p9 := newPerson("张三", "沙河", 90)
fmt.Printf("%#v\n", p9) //&main.person{name:"张三", city:"沙河", age:90}
```

方法和接收者

Go语言中的 `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 `接收者（Receiver）` 。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称首字母的小写，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。

举个例子：

```
//Person 结构体
type Person struct {
    name string
    age  int8
}

//NewPerson 构造函数
func NewPerson(name string, age int8) *Person {
    return &Person{
        name: name,
        age:  age,
    }
}

//Dream Person做梦的方法
func (p Person) Dream() {
    fmt.Printf("%s的梦想是学好Go语言！\n", p.name)
}

func main() {
    p1 := NewPerson("小王子", 25)
    p1.Dream()
}
```

方法与函数的区别是，函数不属于任何类型，方法属于特定的类型。

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` 。 例如我们为 `Person` 添加一个 `SetAge` 方法，来修改实例变量的年龄。

```
// SetAge 设置p的年龄
// 使用指针接收者
func (p *Person) SetAge(newAge int8) {
    p.age = newAge
}
```

调用该方法：

```
func main() {
    p1 := NewPerson("小王子", 25)
    fmt.Println(p1.age) // 25
    p1.SetAge(30)
    fmt.Println(p1.age) // 30
}
```

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。

```
// SetAge2 设置p的年龄
// 使用值接收者
func (p Person) SetAge2(newAge int8) {
    p.age = newAge
}

func main() {
    p1 := NewPerson("小王子", 25)
    p1.Dream()
    fmt.Println(p1.age) // 25
    p1.SetAge2(30) // (*p1).SetAge2(30)
    fmt.Println(p1.age) // 25
}
```

什么时候应该使用指针类型接收者

1. 需要修改接收者中的值
2. 接收者是拷贝代价比较大的大对象
3. 保证一致性，如果有某个方法使用了指针接收者，那么其他的方法也应该使用指针接收者。

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。

```go
//MyInt 将int定义为自定义MyInt类型
type MyInt int

//SayHello 为MyInt添加一个SayHello的方法
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。

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。

```go
//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。

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。

```go
//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:   "小王子",
        Gender: "男",
        Address: Address{
            Province: "山东",
            City:     "威海",
        },
    }
    fmt.Printf("user1=%#v\n", user1)//user1=main.User{Name:"小王子", Gender:"男", Address:main.Addr
}
```

嵌套匿名结构体

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

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。

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。

```
//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 = "沙河娜扎"
    user3.Gender = "男"
    // user3.CreateTime = "2019" //ambiguous selector user3.CreateTime
    user3.Address.CreateTime = "2000" //指定Address结构体中的CreateTime
    user3.Email.CreateTime = "2000"   //指定Email结构体中的CreateTime
}
```

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面向对象的继承。

```go
//Animal 动物
type Animal struct {
    name string
}

func (a *Animal) move() {
    fmt.Printf("%s会动！\n", a.name)
}

//Dog 狗
type Dog struct {
    Feet    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{
        Feet: 4,
        Animal: &Animal{ //注意嵌套的是结构体指针
            name: "乐乐",
        },
    }
    d1.wang() //乐乐会汪汪汪~
    d1.move() //乐乐会动！
}
```

结构体字段的可见性

结构体中字段大写开头表示可公开访问，小写表示私有（仅在定义当前结构体的包中可访问）。

结构体与JSON序列化

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。易于人阅读和编写。同时也易于机器解析和生成。JSON键值对是用来保存JS对象的一种方式，键/值对组合中的键名写在前面并用双引号 `""` 包裹，使用冒号 `:` 分隔，然后紧接着值；多个键值之间使用英文 `,` 分隔。

```go
//Student 学生
type Student struct {
    ID     int
    Gender string
    Name   string
}

//Class 班级
type Class struct {
    Title    string
    Students []*Student
}

func main() {
    c := &Class{
        Title:    "101",
        Students: make([]*Student, 0, 200),
    }
    for i := 0; i < 10; i++ {
        stu := &Student{
            Name:   fmt.Sprintf("stu%02d", i),
            Gender: "男",
            ID:     i,
        }
        c.Students = append(c.Students, stu)
    }
    //JSON序列化：结构体-->JSON格式的字符串
    data, err := json.Marshal(c)
    if err != nil {
        fmt.Println("json marshal failed")
        return
    }
    fmt.Printf("json:%s\n", data)
    //JSON反序列化：JSON格式的字符串-->结构体
    str := `{"Title":"101","Students":[{"ID":0,"Gender":"男","Name":"stu00"},{"ID":1,"Gender":"男"
    c1 := &Class{}
    err = json.Unmarshal([]byte(str), c1)
    if err != nil {
        fmt.Println("json unmarshal failed!")
        return
    }
    fmt.Printf("%#v\n", c1)
}
```

结构体标签（Tag）

`Tag` 是结构体的元信息，可以在运行的时候通过反射的机制读取出来。 `Tag` 在结构体字段的后方定义，由一对**反引号**包裹起来，具体的格式如下：

```
`key1:"value1" key2:"value2"`
```

结构体tag由一个或多个键值对组成。键与值使用冒号分隔，值用双引号括起来。同一个结构体字段可以设置多个键值对tag，不同的键值对之间使用空格分隔。

注意事项： 为结构体编写 `Tag` 时，必须严格遵守键值对的规则。结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，通过反射也无法正确取值。例如不要在key和value之间添加空格。

例如我们为 `Student` 结构体的每个字段定义json序列化时使用的Tag：

```go
//Student 学生
type Student struct {
    ID     int    `json:"id"` //通过指定tag实现json序列化该字段时的key
    Gender string //json序列化是默认使用字段名作为key
    name   string //私有不能被json包访问
}

func main() {
    s1 := Student{
        ID:     1,
        Gender: "男",
        name:   "沙河娜扎",
    }
    data, err := json.Marshal(s1)
    if err != nil {
        fmt.Println("json marshal failed!")
        return
    }
    fmt.Printf("json str:%s\n", data) //json str:{"id":1,"Gender":"男"}
}
```

结构体和方法补充知识点

因为slice和map这两种数据类型都包含了指向底层数据的指针，因此我们在需要复制它们时要特别注意。我们来看下面的例子：

```
type Person struct {
    name   string
    age    int8
    dreams []string
}

func (p *Person) SetDreams(dreams []string) {
    p.dreams = dreams
}

func main() {
    p1 := Person{name: "小王子", age: 18}
    data := []string{"吃饭", "睡觉", "打豆豆"}
    p1.SetDreams(data)

    // 你真的想要修改 p1.dreams 吗?
    data[1] = "不睡觉"
    fmt.Println(p1.dreams)  // ?
}
```

正确的做法是在方法中使用传入的slice的拷贝进行结构体赋值。

```
func (p *Person) SetDreams(dreams []string) {
    p.dreams = make([]string, len(dreams))
    copy(p.dreams, dreams)
}
```

同样的问题也存在于返回值slice和map的情况，在实际编码过程中一定要注意这个问题。

练习题

1. 使用“面向对象”的思维方式编写一个学生信息管理系统。
    1. 学生有id、姓名、年龄、分数等信息
    2. 程序提供展示学生列表、添加学生、编辑学生信息、删除学生等功能

15 comments