Go 语言结构体

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。

类型定义语法：

```
//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。

示例：

```
package main

import "fmt"

func main() {
    // 类型定义
    type MyInt int
    // i 为MyInt类型
    var i MyInt
    i = 100
    fmt.Printf("i: %v i: %T\n", i, i)
}
#结果
i: 100 i: main.MyInt
```

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。

类型别名定义语法：

```
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
```

示例：

```
package main

import "fmt"

func main() {
    // 类型别名定义
    type MyInt2 int
    // i 其实还是 int 类型
    var i MyInt2
    i = 100
    fmt.Printf("i: %v i: %T\n", i, i)
}
#结果
i: 100 i: main.MyInt2
```

类型定义和类型别名的区别

- 类型定义相当于定义了一个全新类型，与之前的类型不同；但是类型别名并没有定义一个新的类型，而是使用了一个别名来替换之前的类型。
- 类型别名只会在代码中存在，在编译完成之后并不会存在该别名。
- 因为类型别名和原来的类型是一致的，所以原来类型拥有的方法类型别名中也是可以调用的，但是重新定义了一个类型，那么不可以调用之前类型的方法。

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。

```
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("a类型:%T\n", a) //a类型:main.NewInt
    fmt.Printf("b类型:%T\n", b) //b类型:int
}
#结果
a类型:main.NewInt
b类型: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。

在大规模重构项目代码的时候，尤其是将一个类型从一个包移动到另一个包中的时候，有些代码会使用新包中的类型，有些代码使用旧包中的类型， 最典型的是 context 包。最开始，context 包名是 golang.org/x/net/context，1.7 开始，引入标准库，这样一来，存在两份。Go 1.9 开始采用别名重构了它。

类型别名这个功能非常有用，鉴于go中有些类型写起来非常繁琐，比如json相关的操作中，经常用到map[string]interface {}这种类型，写起来是不是很繁琐，没关系，给它起个简单的别名!这样用起来爽多了。type strMap2Any = map[string]interface {}。

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。

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。

结构体是复合类型（composite types），当需要定义一个类型，它由一系列属性组成，每个属性都有自己的类型和值的时候，就应该使用结构体，它把数据聚集在一起。然后可以访问这些数据，就好像它是一个独立实体的一部分。结构体也是值类型，因此可以通过 **new** 函数来创建。

组成结构体类型的那些数据称为 **字段（fields）**。每个字段都有一个类型和一个名字；在一个结构体中，字段名字必须是唯一的。

定义结构体

结构体定义需要使用 type 和 struct 语句。struct 语句定义一个新的数据类型，结构体中有一个或多个成员。type 语句设定了结构体的名称。结构体的格式如下：

```
type struct_variable_type struct {
   member definition
   member definition
   ...
   member definition
}
```

`type struct_variable_type struct {member1, member2 int}` 也是合法的语法，它更适用于简单的结构体。

type：结构体定义关键字

struct_variable_type：结构体类型名称

struct：结构体定义关键字

member definition：字段定义 ，member字段名称， definition字段类型

简单来说就是：

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

- 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。
- 同样类型的字段也可以写在一行。
- 如果字段在代码中从来也不会被用到，那么可以命名它为 _。

一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

```
variable_name := structure_variable_type {value1, value2...valuen}
或
variable_name := structure_variable_type { key1: value1, key2: value2..., keyn: valuen}
```

Go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。

示例：

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

结构体中属性的首字母大小写问题

- 首字母大写相当于 public。
- 首字母小写相当于 private。

结构体实例化

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。和声明普通变量相同。

```
var 结构体实例 结构体类型
或者
结构体实例 := 结构体类型{}
```

示例：

```
package main

import "fmt"

type Person struct {
    name, city string
    age        int8
```

```
}

func main() {
    var baozi Person
    fmt.Printf("baozi: %v\n", baozi)
    roubaozi := Person{}
    fmt.Printf("roubaozi: %v\n", roubaozi)
}
#结果
baozi: {  0}
roubaozi: {  0}
```

结构体初始化

未初始化的结构体，成员都是零值 int 0 float 0.0 bool false string nil nil。

Go语言中结构体变量的初始化的方式有两种，分别为：**使用列表对字段挨个赋值**和**使用键值对赋值**的方式。

列表初始化结构体

语法

```
varName := StructName{
    Field1Value,
    Field2Value,
    Field3Value,
    ...
}
```

我们只需要定义一个该结构体类型的变量名，接着在大括号里面对结构体的每个字段挨个设置值。这里的每个字段都必须设置值，如果不设置，则程序报错。就是按照结构体**顺序一致**的挨个设置每个字段的值。

```
package main

import "fmt"

type Person struct {
    name, city string
    age        int8
}

func main() {
    var roubao = Person{
        "肉包",
        "北京",
        20,
    }

    fmt.Printf("roubao: %v\n", roubao)
}
#结果
roubao: {肉包 北京 20}
```

> 使用列表初始化的方式定义结构体时，最后一个字段也需要**加逗号结尾符。**
>
> 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。

键值对初始化结构体

使用键值对赋值的方式初始化结构体，没有被赋值的字段将使用**该字段类型的默认值**，如果使用键值对赋值的方式初始化结构体，那么我们可以有选择的指定赋值的字段。

语法

```
varName := StructName{
    Field1:Field1Value,
    Field2:Field2Value,
    ...
}
```

使用键值对的形式给结构体初始化时，我们只需要指定需要设置值的字段名，然后使用冒号的形式给字段名设置值，不需要设置值的字段名可以忽略。使用键值对的形式初始化结构体变量，键值对顺序可以任意。

```
package main

import "fmt"

type Person struct {
    name, city string
    age        int8
}

func main() {
    var roubao = Person{
        name: "肉包",
        age:  20, //这里顺序不一致  并且没有city字段
    }

    fmt.Printf("roubao: %v\n", roubao)
}
#结果
roubao: {肉包  20}
```

访问结构体成员

如果要访问结构体成员，需要使用点号 . 操作符，格式为：

```
结构体.成员名"
```

示例：

```
package main

import "fmt"
```

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

func main() {
    var baozi Person
    baozi.name = "包子"
    baozi.city = "北京"
    baozi.age = 18

    roubao := Person{
        name: "肉包",
        city: "北京",
        age:  20,
    }
    fmt.Printf("baozi: %v\n", baozi)
    fmt.Printf("roubao: %v\n", roubao)
    fmt.Printf("baozi.name: %v\n", baozi.name)
    fmt.Printf("roubao.name: %v\n", roubao.name)
}
#结果
baozi: {包子 北京 18}
roubao: {肉包 北京 20}
baozi.name: 包子
roubao.name: 肉包
```

们通过`.`来访问结构体的字段（成员变量）,例如baozi.name和baozi.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。匿名结构体没有类型名称，无须通过 type 关键字定义就可以直接使用。

```
package main

import (
    "fmt"
)

func main() {
    var baozi struct {
        Name string
        Age  int
    }
    baozi.Name = "包子"
    baozi.Age = 18
    fmt.Printf("%#v\n", baozi)
}
#结果
struct { Name string; Age int }{Name:"包子", Age:18}
```

结构体指针

取结构体的地址实例化

结构体指针和普通的变量指针相同。你可以定义指向结构体的指针类似于其他指针变量。

```
var 结构体指针变量 *结构体
```

以上定义的指针变量可以存储结构体变量的地址。

查看结构体变量地址，可以将 `&` 符号放置于结构体变量前：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。

```
结构体指针变量 = &结构体
```

使用结构体指针访问结构体成员，使用 "`.`" 操作符：

```
结构体指针变量.结构体字段
```

示例：

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age  int
}

// 添加指针接收器
func printPerson(p *Person) {
    fmt.Println("*Person Name =", p.Name, "Age =", p.Age)
}
func main() {
    var pPerson = &Person{
        Name: "包子",
        Age:  18,
    }
    fmt.Printf("pPerson: %v\n", pPerson)
    printPerson(pPerson)
}
#结果
pPerson: &{包子 18}
*Person Name = 包子 Age = 18
```

使用`new`关键字创建结构体指针

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。

```
var 结构体指针变量 = new(结构体)
```

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    var p_person = new(Person)
    p_person.Name = "测试"
    p_person.Age = 18
    fmt.Printf("p_person=%v\n", p_person)  //指针值
    fmt.Printf("p_person=%T\n", p_person)  //指针类型
    fmt.Printf("p_person=%p\n", p_person)  //指针地址
    fmt.Printf("p_person=%v\n", *p_person) //取指针值
}
#结果
p_person=&{测试 18}
p_person=*main.Person
p_person=0xc000008078
p_person={测试 18}
```

结构体作为函数参数

可以像其他数据类型一样将结构体类型作为参数传递给函数。

1. 直接传递结构体，这是是一个副本（拷贝），在函数内部不会改变外面结构体内容。
2. 传递结构体指针，这时在函数内部，能够改变外部结构体内容。

示例：

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age  int
}

func showPerson(p Person) {
    fmt.Printf("p: %v\n", p)
}

func showPerson1(p *Person) {
    fmt.Printf("p: %v\n", p)
}

func main() {
```

```
    per := Person{
        Name: "包子",
        Age:  20,
    }
    showPerson(per) //值传递

    per1 := &Person{
        Name: "肉包子",
        Age:  18,
    }
    showPerson1(per1) //指针传递（引用传递）
}
#结果
p: {包子 20}
p: &{肉包子 18}
```

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。

```
package main

import (
    "fmt"
)

type Person struct {
    string //没有字段名
    int
}

func main() {
    per := Person{
        "包子", //列表实例化
        20,
    }
    fmt.Printf("per: %v\n", per)
}
```

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语言没有面向对象编程思想，也没有继承关系，但是可以通过结构体嵌套来实现这种效果。

```
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 //嵌套了地址结构体
}

func main() {
    userInfo := User{
        Name:   "包子",
        Gender: "男",
        Address: Address{
            Province: "黑龙江",
            City:     "哈尔滨",
        },
    }
    fmt.Printf("userInfo=%v\n", userInfo)
}
#结果
userInfo={包子 男 {黑龙江 哈尔滨}}
```

嵌套匿名结构体

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。

```
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 //嵌套了匿名结构体
}

func main() {
    var userInfo User
    userInfo.Name = "肉包子"
    userInfo.Gender = "男"
    userInfo.Address.Province = "黑龙江"      //通过匿名结构体.字段名访问
    userInfo.City = "哈尔滨"                  //直接访问匿名结构体的字段名
```

```
    fmt.Printf("userInfo=%v\n", userInfo) //
}
#结果
userInfo={肉包子 男 {黑龙江 哈尔滨}}
```

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

import (
    "fmt"
)

// 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 userInfo User
    userInfo.Name = "肉包子"
    userInfo.Gender = "男"
    userInfo.Address.Province = "黑龙江"
    userInfo.Address.City = "哈尔滨"
    userInfo.Email.Account = "baozi@163.com"
    userInfo.Address.CreateTime = "2023" //指定Address结构体中的CreateTime
    userInfo.Email.CreateTime = "2023"   //指定Email结构体中的CreateTime
    fmt.Printf("userInfo=%v\n", userInfo)
}
#结果
userInfo={肉包子 男 {黑龙江 哈尔滨 2023} {baozi@163.com 2023}}
```

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。

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func newPerson(name string, age int) *Person {
    return &Person{
        Name: name,
        Age:  age,
    }
}

func main() {
    userInfo := newPerson("包子", 18) //调用构造函数
    fmt.Printf("%v\n", *userInfo)
}
#结果
{包子 18}
```