第二节课 结构体 指针 接口

类型别名和自定义类型

自定义类型

有了上节课的学习，我们了解了Go语言的基本的数据类型string、整型、浮点型、布尔等。除了这些，我们也可以通过**type关键词**自定义类型。

```
type Date string


var today Date="2024-11-03"
```

通过`type`定义的`Date`就是一种**新的类型**，它具有string的特性,但**不是string类型。**所以不能互相赋值或比较。

自定义类型帮助更好地封装数据、组织代码，提高程序的可读性和可维护性

类型别名

类型别名是一个类型的**别称**，本质上还是同一个类型

```
type MyInt=int
```

常见的`rune`,`byte`,`any`就是**类型别名。**

```
// byte is an alias for uint8 and is equivalent to uint8 in all ways. It is
// used, by convention, to distinguish byte values from 8-bit unsigned
// integer values.
type byte = uint8

// rune is an alias for int32 and is equivalent to int32 in all ways. It is
// used, by convention, to distinguish character values from integer values.
type rune = int32
```

自定义类型VS类型别名

最主要的区别是 自定义类型是新类型，类型别名依然是旧类型，只不过有了一个别名新的名称 。

```go
package main

import (
    "fmt"
    "reflect"
)

type NewInt int

type MyInt= int

func main() {
    var a NewInt
    var b MyInt

    fmt.Printf("type of a:%T\n", a)
    fmt.Printf("type of b:%T\n", b)

}
```

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

结构体的定义

使用`type`和`struct`关键字来定义结构体：

```
    type 类型名 struct {    //结构体的名称同一个包内不能重复
        字段名1 字段类型     //字段名必须唯一
        字段名2 字段类型
        ...
    }

//举例

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

嵌套结构体

我们把一个结构体中可以嵌套包含另一个结构体或结构体指针 叫做 **嵌套结构体**或者是**结构体内嵌**

```
type Scores struct {
    Math int
    English  int
}

type Student struct {
    Name string
    score Scores
}
```

结构体实例

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。

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

func main() {
    var p1 person
    p1.name = "DYT"
    p1.city = "重庆"
    p1.age = 18
    fmt.Printf("p1=%v\n", p1)
    fmt.Printf("p1=%#v\n", p1)
}
```

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 = "DYT"
    user.Age = 18
    fmt.Printf("%#v\n", user)
}
```

创建指针类型结构体

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

p2.name = "DYT"
p2.age = 18
p2.city = "CQ"
fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"DYT", city:"CQ", age:18}
```

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 = "DYT"
p3.age = 18
p3.city = "CQ"
fmt.Printf("p3=%#v\n", p3)
}
```

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  int
}

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: "DYT",
    city: "CQ",
    age:  18,
}
fmt.Printf("p5=%#v\n", p5)
```

对结构体指针进行键值对初始化

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

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

```
p8 := &person{
    "DYT",
    "CQ",
    18,
}
fmt.Printf("p8=%#v\n", p8)
```

此种方法需要注意： 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**。

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）。 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
}
```

example:

```go
type People struct {
    Name  string
    Age   int
    Books []Book
}

type Book struct {
    Name string
}

func (w People) PrintName() {
    fmt.Println(w.Name)
}
func (w People) PrintAge() {
    fmt.Println(w.Age)
}

func (w People) PrintBook() {
    fmt.Println(w.Books)
}

func (b Book) PrintBookName() {
    fmt.Println(b.Name)
}
```

方法只是个带接收者参数的函数。

指针

区别于C/C++中的指针，Go语言中的指针不能进行偏移和运算，是**安全指针**。

要搞明白Go语言中的指针需要先知道3个概念：`指针地址`、`指针类型`和`指针取值`。

Go语言中的指针

任何程序数据载入内存后，在内存都有他们的地址，这就是指针。而为了保存一个数据在内存中的地址，我们就需要指针变量。

go语言中的函数传参都是**值传递**，当我们想要**修改某个变量**的时候，我们可以创建一个指向该变量地址的指针变量。传递数据使用指针，而无须拷贝数据。类型指针不能进行偏移和运算。Go语言中的指针操作非常简单，只需要记住两个符号：`&`（取地址）和`*`（根据地址取值）。

值传递：实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

指针地址和指针类型

每个变量在运行时都拥有一个地址，这个地址代表变量在内存中的位置。Go语言中使用&字符放在变量前面对变量进行“取地址”操作。

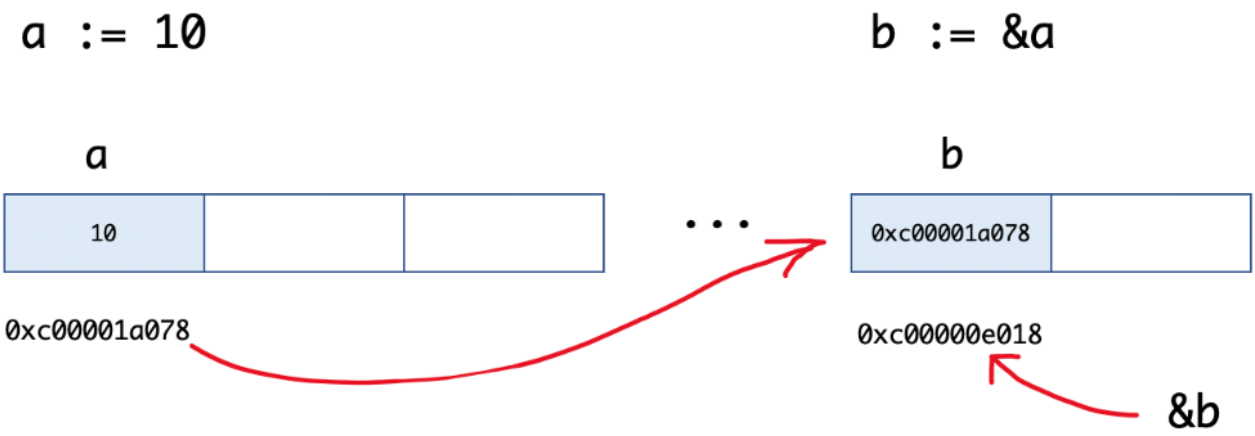
Go语言中的值类型`（int、float、bool、string、array、struct）`都有对应的指针类型，如：`*int、*int64、*string`等

我们可以通过 & 来获取地址

```
ptr := &v
```

example:

```
func main() {
    a := 10
    b := &a
    fmt.Printf("a:%d ptr:%p\n", a, &a)
    fmt.Printf("b:%p type:%T\n", b, b)
    fmt.Println(&b)
}
```



空指针

当一个指针被定义后没有分配到任何变量时，它的值为 nil

```go
package main

import "fmt"

func main() {
   var p *int
   fmt.Println(p)
   fmt.Printf("p的值是%v\n", p)
   if p != nil {
       fmt.Println("非空")
   } else {
       fmt.Println("空指针")
   }
}
```

new和make

看一个小例子，想想这段程序的运行结果

```go
func main() {
    var a *int
    *a = 10
    fmt.Println(*a)

    var b map[string]string
    b["name"] ="代雨婷"
    fmt.Println(b)
}
```

答案是执行上面的代码会引发panic，为什么呢?

在Go语言中对于**引用类型**的变量，我们在使用的时候不仅要声明它，还要为它分配内存空间，否则我们的值就没办法存储。

而对于**值类型**的声明不需要分配内存空间，是因为它们在声明的时候已经默认分配好了内存空间。

要分配内存，就引出来今天的new和make。 Go语言中new和make是内建的两个函数，主要用来分配内存。

new

new是一个内置的函数，它的函数签名如下：

```
func new(Type) *Type
```

上面的例子：

```
var a *int    //指针—>引用类型
a=new(int)
  *a = 10
  fmt.Println(*a)
```

make

make也是用于内存分配的，区别于new，它**只用于`slice`、`map`以及`chan`的内存创建**，而且它返回的类型就是这三个**类型本身**，而不是他们的指针类型，因为这三种类型就是引用类型，所以就没有必要返回他们的指针了。make函数的函数签名如下：

```
func make(t Type, size ...IntegerType) Type
```

对于上面的例子：

```
  var b map[string]string
b=make(map[string]string)
  b["name"] = "DYT"
  fmt.Println(b)
```

你是不是会疑惑，这里没有利用map的指针，为什么修改成功了？

这里涉及到make函数创建map时会调用runtime.makemap函数,该函数返回的是*hmap类型，也就是说我们创建的map本质上是一个 *hmap类型...

接口

在Go语言中接口（interface）是一种类型，一种抽象的类型。

相较于之前章节中讲到的那些具体类型（字符串、切片、结构体等）更注重“我是什么”，接口类型更注重“我能做什么”的问题。

接口类型

接口是一种由程序员来定义的类型，**一个接口类型就是一组方法的集合**，它规定了需要实现的所有方法。

接口的定义

每个接口类型由**任意个方法签名**组成，接口的定义格式如下：

```
type 接口类型名 interface{
        方法名1( 参数列表1 ) 返回值列表1
        方法名2( 参数列表2 ) 返回值列表2
    }
```

接口的实现条件

接口就是规定了一个**需要实现的方法列表**，在 Go 语言中一个类型只要**实现了接口中规定的所有方法**，那么我们就称它**实现了这个接口**。

我们定义的`Sayer`接口类型，它包含一个`Say`方法。

```
type Sayer interface {
    Say()
}

type cat struct {}

//Cat类型的Say方法
func (c cat) Say() {
    fmt.Println("喵喵喵")
}

func main() {
    var x Sayer=cat{}
    x.Say()
}
```

这样就称Cat实现了Sayer接口。

空接口

空接口是指没有定义**任何方法**的接口。因此任何类型都实现了空接口。

空接口被用来处理未知类型的值。例如：`fmt.Print`可接受类型为`interface{}`的任意数量的参数。

```
func Print(a ...any) (n int, err error)
```

通常我们在使用空接口类型时不必使用`type`关键字声明，可以像下面的代码一样直接使用`interface{}`。

```
    var x interface{}
     s := "hello"
     x = s

func NilInterface(x  interface{}){
}
```

类型断言

上面我们提到了 空接口可以存储任意类型，但是我们怎么才能知道他的类型呢?

类型断言 提供了访问接口值底层具体值的方式

语句断言接口值`i`保存了具体类型`T`，并将其底层类型为`T`的值赋予变量`t`

```
t := i.(T)
```

为了 **判断** 一个接口值是否保存了一个特定的类型，类型断言可返回两个值：其**底层值**以及一个报告断言是否成功的**布尔值**。

通过.(type)返回value和ok，若断言失败则ok为false，value为空值，若成功则返回对应类型的value，ok为true

```
func main() {
    var x interface{}
    s := "RedRock"
    x = s
    v, ok := x.(string)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("类型断言失败")
    }
}
```

为什么要使用接口

在 Golang 语言中，我们使用结构体和方法可以很完美的实现需求。为什么还要使用接口呢?

接口的意义在于：将约定和实现分离，或者说，就是让调用者无需关心底层的实现，只需要遵照接口方法里的定义调用。

相关解释&扩展

https://www.zhihu.com/question/318138275

https://blog.csdn.net/qq_34556414/article/details/123395653

作业

LV0：

复习理解上课所讲的内容，自己敲一遍代码。

LV1：

任务描述：

1. 定义一个表示 **学生(Student)** 的结构体，包含以下字段：
    - 姓名 (Name)
    - 年龄 (Age)
    - 分数 (Score)
    - ……
2. 创建多个学生实例，存入一个 **切片(Slice)** 中。
3. 遍历该切片，打印每个学生的信息。

LV2：

任务描述：(注意嵌套结构体)

1. 定义两个结构体：

    Student：包含姓名 (Name)、年龄 (Age) 和 **成绩切片(Scores)**（一个 `[]int`）

    Classroom：包含班级名 (ClassName) 和多个学生（`[]*Student`，即指针切片）.

    2.编写以下函数：

- `AddStudent(c *Classroom, s *Student)`：向班级中添加学生。

- `UpdateScore(s *Student, score int)`：向学生的 `Scores` 切片中追加新的成绩。
- `CalculateAverage(s *Student) float64`：计算并返回学生的平均成绩。
- 在 `main` 函数中，完成以下任务：
    - 创建一个班级，并添加若干学生。
    - 给每个学生添加不同的成绩，并打印他们的平均成绩

LV3:

实现一个任务调度器

任务描述：

你需要实现一个简单的任务调度器，能够管理和执行不同类型的任务。每种任务都有不同的执行逻辑，但它们都应该遵循相同的接口。

步骤

1. **定义接口**：定义一个名为 `Task` 的接口，包含以下方法：
    - `Execute() error`：执行任务，返回错误信息（如果有）。
2. **实现任务结构体**：
    - 创建一个 `PrintTask` 结构体，包含一个字符串字段 `Message`，用于打印消息。
    - 创建一个 `CalculationTask` 结构体，包含两个字段 `A` 和 `B`，用于计算它们的和，并打印结果。
    - 创建一个 `SleepTask` 结构体，包含一个整数字段 `Duration`，用于使程序休眠指定的秒数。
3. **实现接口方法**：
    - 为 `PrintTask` 结构体实现 `Execute` 方法，打印 `Message`。
    - 为 `CalculationTask` 结构体实现 `Execute` 方法，计算并打印 `A + B` 的结果。
    - 为 `SleepTask` 结构体实现 `Execute` 方法，使用 `time.Sleep` 使程序休眠指定的时间。
4. **任务调度器**：
    - 创建一个 `Scheduler` 结构体，包含一个切片字段 `Tasks`，用于存储所有任务。
    - 为Scheduler添加一个方法 `AddTask(task Task)`，将任务添加到调度器中。
    - 为Scheduler添加一个方法 `RunAll()`，遍历所有任务并调用它们的 `Execute` 方法。

预习作业

使用你已经学过的知识，尝试用Go编写一个计时器程序

当用户：

- 输入0时，重置计时器
- 输入1时，开始计时
- 输入2时，暂停或继续计时

如果暂时做不出来也没关系，分析一下可能的问题

提交

将作业项目打包压缩成压缩包，并作为附件邮箱发到*be@redrock.team* 中，邮箱的主题为 学号-姓名-后端第几次作业-*lv*几 （做到哪一级发几） *example*：*2024213xxx*-卷娘-后端第二次作业-*lv2*