# Day3 函数和结构体

tony

# 目录

1. Map数据类型

2. 函数详解

3. 结构体

4. 课后作业

# map声明和定义

1. map类型是一个key-value的数据结构。

```
//var a map[key的类型]value类型
var a map[string]int
var b map[int]string
var c map[float32]string
```

***注意：map必须初始化才能使用，否则panic***

# map声明和定义

2. map类型的变量默认初始化为nil，需要使用make分配map内存

```
package main

import (
    "fmt"
)

func main() {
    var a map[string]int
    if a == nil {
        fmt.Println("map is nil. Going to make one.")
        A = make(map[string]int)
    }
}
```

# map基本操作

3. map插入操作

```
package main

import (
    "fmt"
)

func main() {
    a := make(map[string]int)
    a["steve"] = 12000
    a["jamie"] = 15000
    a["mike"] = 9000
    fmt.Println(“a map contents:", a)
}
```

# map基本操作

4. 声明时进行初始化

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int {
        "steve": 12000,
        "jamie": 15000,
    }
    a[“mike"] = 9000
    fmt.Println(“a map contents:", a)
}
```

# map基本操作

5. 通过key访问map中的元素

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    b := "jamie"
    fmt.Println("Salary of", b, "is", a[b])
}
```

# map基本操作

5. 通过key访问map中的元素

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    b := “123”
    fmt.Println("Salary of", b, "is", a[b])
}
```

# map基本操作

6. 如何判断map指定的key是否存在？ value, ok := map[key]

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    b := "joe"
    value, ok := a[b]
    if ok == true {
        fmt.Println("Salary of", b, "is", value)
    } else {
        fmt.Println(b,”not found")
    }
}
```

# map基本操作

## 7. map遍历操作

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    fmt.Println("All items of a map")
    for key, value := range a {
        fmt.Printf("personSalary[%s] = %d\n", key, value)
    }

}
```

# map基本操作

8. map删除元素

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    fmt.Println("map before deletion", a)
    delete(a, "steve")
    fmt.Println("map after deletion", a)

}
```

# map基本操作

9. map的长度

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    fmt.Println("length is", len(a))

}
```

# map基本操作

10.map是引用类型

```
package main

import (
    "fmt"
)

func main() {
    a := map[string]int{
        "steve": 12000,
        "jamie": 15000,
    }
    a["mike"] = 9000
    fmt.Println(“origin map", a)
    b := a
    b["mike"] = 18000
    fmt.Println(“a map changed", a)

}
```

# map 进行排序

11.默认情况下，map并不是按照key有序进行遍历的

```
package main
import (
	"fmt"
)

func main() {
	var a map[string]int = make(map[string]int, 10)
	for i := 0; i < 10; i++ {
		key := fmt.Sprintf("key%d", i)
		a[key] = i
	}

	for key, value := range a {
		fmt.Printf("key:%s = %d\n", key, value)
	}
}
```

# map 进行排序

12. map按照key进行排序，遍历

```
package main
import (
    "fmt"
    "sort"
)
func main() {
    var a map[string]int = make(map[string]int, 10)
    for i := 0; i < 10; i++ {
        key := fmt.Sprintf("key%d", i)
        a[key] = i
    }
    var keys []string
    for key, _ := range a {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    for _, key := range keys {
        fmt.Printf("key:%s=%d\n", key, a[key])
    }
}
```

# map 进行排序

## 13. map类型的切片

```
package main

import (
	"fmt"
)

func main() {
	var mapSlice []map[string]int
	mapSlice = make([]map[string]int, 5)
	fmt.Println("before map init")
	for index, value := range mapSlice {
		fmt.Printf("index:%d value:%v\n", index, value)
	}

	fmt.Println()
	mapSlice[0] = make(map[string]int, 10)
	mapSlice[0]["a"] = 1000
	mapSlice[0]["b"] = 2000
	mapSlice[0]["c"] = 3000
	mapSlice[0]["d"] = 4000
	mapSlice[0]["e"] = 5000

	fmt.Println("after map init")
	for index, value := range mapSlice {
		fmt.Printf("index:%d value:%v\n", index, value)
	}
}
```

# 函数介绍

1. 定义：有输入、有输出，用来执行一个指定任务的代码块。

```
func functionname([parametername type]) [returntype] {
 //function body
}

    //其中参数列表和返回值列表是可选
```

# 函数介绍

## 2. 无参数和返回值的函数

```
func functionname() {

 //function body

}
```

# 函数介绍

3. 练习，实现两个数相加

```
func add(a int, b int) int {

    Return a + b

}
```

# 函数介绍

4. 如何连续的一系列参数的类型是一样，前面的类型可以不写，例如：

```
func add(a, b int) int {

    Return a + b

}
```

# 函数介绍

## 5. 函数调用

```
func add(a, b int) int {

        Return a + b

}


      func main() {
          sum := add(2, 3)
}
```

# 多返回值和可变参数

## 1. 多返回值

```
func calc(a, b int) (int, int) {
        sum := a + b
        sub := a - b
        return sum, sub
}


        func main() {
            sum, sub := add(2, 3)
}
```

# 多返回值和可变参数

## 2. 对返回值进行命名

```
func calc(a, b int) (sum int, sub int) {
        sum = a + b
        sub = a - b
        return
}


    func main() {
        sum, sub := add(2, 3)
}
```

# 多返回值和可变参数

3. 对返回值进行命名

```
func calc(a, b int) (sum int, sub int) {
        sum = a + b
        sub = a - b
        return
}


        func main() {
            sum, sub := add(2, 3)
}
```

# 多返回值和可变参数

4. _标识符

```
func calc(a, b int) (sum int, sub int) {
        sum = a + b
        sub = a - b
        return
}


    func main() {
        sum, _ := add(2, 3)
}
```

# 多返回值和可变参数

## 5. 可变参数

```
func calc_v1(b ...int) (sum int, sub int) {

	return
}

func calc_v2(a int, b ...int) (sum int, sub int) {

	return
}

func calc_v3(a int, b int, c ...int) (sum int, sub int) {

	return
}
```

# defer语句

1. defer

```
func calc_v1(b ...int) (sum int, sub int) {

    defer fmt.Println(“defer”)

    return

}
```

# defer语句

2. 多个defer语句，遵循栈的特性：先进后出

```
func calc_v1(b ...int) (sum int, sub int) {

    defer fmt.Println(“defer1”)

    defer fmt.Println(“defer2”)

    return

}
```

# 内置函数

1. close：主要用来关闭channel

2. len：用来求长度，比如string、array、slice、map、channel

3. new：用来分配内存，主要用来分配值类型，比如int、struct。返回的是指针

4. make：用来分配内存，主要用来分配引用类型，比如chan、map、slice

5. append：用来追加元素到数组、slice中

6. panic和recover：用来做错误处理

# 变量作用域

1. 全局变量，在程序整个生命周期有效。

```
var a int = 100
```

# 变量作用域

2. 局部变量，分为两种：1）函数内定义，2）语句块内定义。

```
func add(a int, b int) int {
    var sum int = 0
    //sum是局部变量
    if a > 0 {
      var c int = 100
      //c是布局变量，尽在if语句块有效
    }
}
```

# 变量作用域

3. 可见性，包内任何变量或函数都是能访问的。包外的话，首字母大写是可导出的能够被其他包访问或调用。小写表示是私有的，不能被外部的包访问。

```
func add(a int, b int) int {

}
//add这个函数只能在包内部调用，是私有的，不能被外部的包调用
```

# 匿名函数

1. 函数也是一种类型，因此可以定义一个函数类型的变量

# 匿名函数

2. 匿名函数，即没有名字的函数

# 匿名函数

3. defer中使用匿名函数

# 匿名函数

4. 函数作为一个参数

# 闭包

1. 闭包：一个函数和与其相关的引用环境组合而成的实体

```
package main

import “fmt”

func main() {
        var f = Adder()
        fmt.Print(f(1),” - “)
        fmt.Print(f(20),” - “)
        fmt.Print(f(300))
}

func Adder() func(int) int {
        var x int
        return func(d int) int {
                x += d
                 return x
        }
}
```

# 闭包

## 2. 闭包的例子1

```
package main

func add(base int) func(int) int {
    return func(i int) int {
        base += i
        return base
    }
}

func main() {
    tmp1 := add(10)
    fmt.Println(tmp(1), tmp(2))
    tmp2 := add(100)
    fmt.Println(tmp2(1), tmp2(2))
}
```

# 闭包

3. 闭包的例子2

```
package main

import (
	"fmt"
	"strings"
)
func makeSuffixFunc(suffix string) func(string) string {
	return func(name string) string {
		if !strings.HasSuffix(name, suffix) {
			return name + suffix
		}
		return name
	}
}
func main() {
	func1 := makeSuffixFunc(".bmp")
	func2 := makeSuffixFunc(".jpg")
	fmt.Println(func1("test"))
	fmt.Println(func2("test"))
}
```

# 闭包

4. 闭包的例子3

```
func calc(base int) (func(int) int, func(int) int) {

    add := func(i int) int {
        base += i
        return base
    }

    sub := func(i int) int {
        base -= i
        return base
    }

    return add, sub
}

func main() {
    f1, f2 := calc(10)
    fmt.Println(f1(1), f2(2))
    fmt.Println(f1(3), f2(4))
    fmt.Println(f1(5), f2(6))
    fmt.Println(f1(7), f2(8))
}
```

# 闭包

5. 闭包的例子4

```
func calc(base int) (func(int) int, func(int) int) {

    add := func(i int) int {
        base += i
        return base
    }

    sub := func(i int) int {
        base -= i
        return base
    }

    return add, sub
}

func main() {
    f1, f2 := calc(10)
    fmt.Println(f1(1), f2(2))
    fmt.Println(f1(3), f2(4))
    fmt.Println(f1(5), f2(6))
    fmt.Println(f1(7), f2(8))
}
```

# struct声明和定义

1. Go中面向对象是通过struct来实现的, struct是用户自定义的类型

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
}
```

**注意：type是用来定义一种类型**

# struct声明和定义

2. struct初始化方法1

```
var user User
user.Age = 18
user.Username = “user01”
user.Sex = “男”
user.AvatarUrl = “http://my.com/xxx.jpg"
```

**注意：使用变量名+ ‘.’ + 字段名访问结构体中的字段**

# struct声明和定义

3. struct初始化方法2

```
var user User = User {
        “Username” : “user01”,
        “Age”: 18,
        “Sex”: “男”,
        “AvatarUrl”: “http://my.com/xxx.jpg”,
}
```

**注意：也可以部分初始化**

**更简单的写法：**

```
user := User {
        “Username” : “user01”,
        “Age”: 18,
        “Sex”: “男”,
        “AvatarUrl”: “http://my.com/xxx.jpg”,
}
```

# struct声明和定义

4. struct初始化的默认值

```
var user User
fmt.Printf("%#v\n", user)
```

# struct声明和定义

5. 结构体类型的指针

```
var user *User = &User{}

fmt.Printf("%p %#v\n", user)
```

```
var user *User = &User {
        "Username" : "user01",
        "Age": 18,
        "Sex": "男",
        "AvatarUrl": "http://my.com/xxx.jpg",
}
```

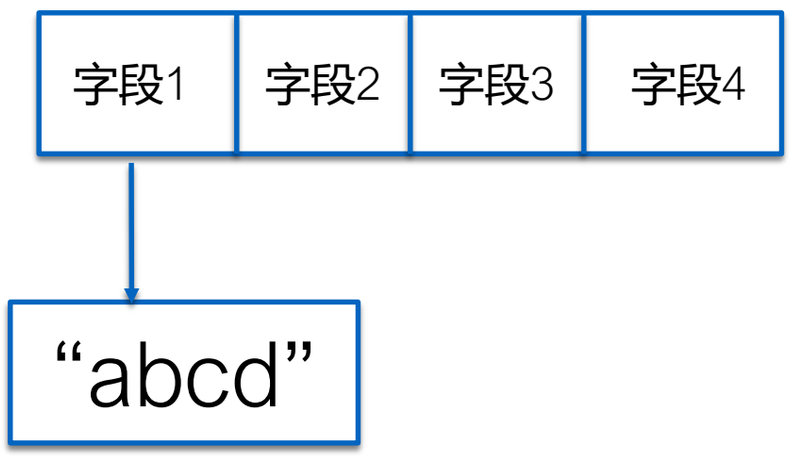
**注意：&User{}和new(User)本质上是一样的，都是返回一个结构体的地址**

```
var user User = new(User)

user.Age = 18
user.Username = "user01"
user.Sex = "男"
user.AvatarUrl = "http://my.com/xxx.jpg"
```

# struct内存布局

6. 结构体的内存布局： 占用一段连续的内存空间



# struct内存布局

7. 结构体没有构造函数， 必要时需要自己实现

# 匿名字段和嵌套

8. 匿名字段: 即没有名字的字段

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
}
```

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
    int
        string
}
```

**注意：匿名字段默认采用类型名作为字段名**

# 匿名字段和嵌套

9. 结构体嵌套

```
type Address struct {
    City        string
    Province    string
}
```

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
    address Address
}
```

# 匿名字段和嵌套

## 10.匿名结构体

```
type Address struct {
    City        string
    Province    string
}
```

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
    Address
}
```

# 匿名字段和嵌套

## 11.匿名结构体与继承

```
type Animal struct {
    City        string
    Province    string
}
```

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
    Address
}
```

# 匿名字段和嵌套

## 12. 冲突解决

```
type Address struct {
    City        string
    Province    string
    CreateTime  string
}
```

```
type Email struct {
    Account     string
    CreateTime  string
}
```

```
type User struct {
    Username  string
    Sex       string
    Age       int
    AvatarUrl string
    Address
    Email
    CreateTime string
}
```

# 结构体与tag应用

13.字段可见性，大写表示可公开访问，小写表示私有

```
type User struct {
    Username  string
    Sex       string
    Age       int
    avatarUrl string
    CreateTime string
}
```

# 结构体与tag应用

14. 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

```
type User struct {
    Username  string   `json:"username",db:"user_name"`
    Sex       string   `json:"sex"`
    Age       int      `json:"age"`
    avatarUrl string
    CreateTime string
}
```

**字段类型后面，以反引号括起来的key-value结构的字符串，多个tag以逗号隔开。**

# 课后练习

1. 实现一个简单的学生管理系统，每个学生有分数、年级、性别、名字等字段，用户可以在控制台添加学生、修改学生信息、打印所有学生列表的功能。

```
package main
import (
    “fmt”
    “flag”
)
type Student struct {
    Username string
    Score float32
    Grade string
    Sex int
}

func main() {
}
```

# QA