练识课堂 -- 资深Go语言工程师实践课程

1 数据类型

01 声明

```
package main

import "fmt"

func main() {
   b := []byte("Hello")
   r := []rune("Seattle")
   i := []int("Gophers")
   fmt.Println(b, r, i)
}
```

- 1，2有效
- byte，rune有效

02 数据类型

```
package main

func A(string string) string {
return string + string
}

func B(len int) int {
return len+len
}

func C(val, default string) string {
 if val == "" {
         return default
     }
 return val
}
```

- A B 能运行
- default 是关键字，不可以使用做变量名，string 和 len 是事先声明的标识符，可以使用，但是不建议使用

03 类型别名

```
package main

import "fmt"

type P *int
type Q *int
```

```
func main() {
   var p P = new(int)
   *p += 8
   var x *int = p
   var q Q = x
   *q++
   fmt.Println(*p, *q)
}
```

- 9 9
- 引用同一内存块

04 硬编码格式

```
package main
import "fmt"
func main() {
  var x uint32 = 100.5 + 100.5 + 2147483447.0
  fmt.Println(x)
}
```

- 2147483648
- float在运算时候依然按照float类型进行运算，在进行赋值时会进行类型转换。Go不允许在变量之间进行隐式转换，但是对于常量来说规则是不同的。在任何情况下，一个常数x可以转换为T类型。

05 编码

```
package main

import "fmt"

func main() {
   fmt.Println("Seattle, WA 🌳")
}
```

- 可以有效运行
- utf-8编码中可以正常打印图片

06 字符串处理

```
package main
import (
  "fmt"
  "strings"
)
func main() {
  s := strings.TrimRight("abcdefedcba", "fedabc")
  fmt.Printf("%q\n", s)
}
```

- ""

- TrimRight 从右侧开始依次扫描是否包含右侧所存在字符，存在则消除，直到碰到第一个不存在字符为终结，每个字符消除都会依次检测是否在第二个参数中存在，只想删除特定顺序时候使用。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    s := strings.TrimSuffix("abcdeabcdef", "abcdef")
    fmt.Printf("%q\n", s)
}
```

07 位运算符

```
package main

func main() {
   a := 2 ^ 15
   b := 4 ^ 15
   if a > b {
       println("a")
   } else {
       println("b")
   }
}
```

- a
- ^ 异或符号

```
a := 2 ^ 15
2 = 0010
15 = 1111
     XOR
a := 1101
b := 4 ^ 15
4 = 0100
15 = 1111
     XOR
b := 1011
a > b == true
```

08 切片拷贝

```go
package main

import "fmt"

func f(a int, b uint) {
  val := copy(make([]struct{}, a), make([]struct{}, b))
  fmt.Printf("%d\n",  val)
}

func main() {
  f(90, 314)
}
```

- 90
- copy 返回的是成功拷贝的个数，在第一个结构体数组空间用完或者第二个结构体数组完全复制后会停止，较小的数字则为实际copy次数

09 切片比较运算

```go
package main

type Hash [32]byte

func MustNotBeZero(h Hash) {
 if h == Hash {
   fmt.Println(1)
 }
}

func main() {
 MustNotBeZero(Hash{})
}
```

- 编译错误
- type Hash is not an expression

```go
  package main

  import "fmt"

  type Hash [32]byte

  func MustNotBeZero(h Hash) {
    if h == [32]byte{} {
      fmt.Println(1)
    }
  }

  func main() {
    MustNotBeZero(Hash{})
  }
```

- Hash是一个32byte声明的数组，不可以直接使用Hash{}进行判断，但是可以直接使用[32]byte{} 进行判断

10 切片操作

```
package main

import "fmt"

 func main() {
     v := []int{1, 2, 3}
     for i := range v {
         v = append(v, i)
     }
     fmt.Println(v)
 }
```

- [1,2,3,0,1,2]
- range 在运行时候会对源数据进行一次拷贝，所以不会影响到数据输出

11 切片截取

```
package main

import "fmt"

func main() {
  a := []int{0,1,2,3,4,5,6,7}
  b := a[:3]
  b = append(b,4)
  fmt.Println(a)
  fmt.Println(b)
}
```

- 输出为 a [0 1 2 4 4 5 6 7] b [0 1 2 4]
- 切片下标是一个[a,b)空间，包含起始节点但是包含结束节点
- append在对切片进行添加元素时候会检测切片所指向的数组是否还有足够的空余空间，如果有责直接在当前切片结尾处继续添加，会覆盖原数组内的数据，如果不足则会开辟新的内存空间，不会对原数据进行修改

12 切片追加

```
package main
import "fmt"
func main() {
  s1 := []int{1, 2, 3}
  s2 := []int{4, 5}
  s1 = append(s1, s2)
  fmt.Println(s1)
}
```

- 程序编译错误
- append第二个到第N个参数为元素，使用数组或者切片需要加... ， s1 = append(s1,s2...)

13 map的值

```
type student struct {
```

```
  Name string
  Age  int
}

func pase_student() {
  m := make(map[string]*student)
  stus := []student{
     {Name: "zhou", Age: 24},
     {Name: "li", Age: 23},
     {Name: "wang", Age: 22},
  }
  for _, stu := range stus {
     m[stu.Name] = &stu
  }
  fmt.Println(m)
}
```

- 不可以得到想要的结果(map中的value指向同一个内存地址)

14 map的值操作

```
package main

import "fmt"

func main() {
   m := make(map[string]int)
   m["foo"]++
   fmt.Println(m["foo"])
}
```

- 1
- 会进行初始化，m["foo"]++可以分解为

```
v := m["foo"]  //因为还不存在，int类型会给出0
v++  //v = 1
m["foo"] = v //一次map赋值
```

15 map的值操作

```
package main

import "fmt"

type Test struct {
    Name string
}

var list map[string]Test

func main() {

    list = make(map[string]Test)
    name := Test{"xiaoming"}
    list["name"] = name
```

```
    list["name"].Name = "Hello"
    fmt.Println(list["name"])
}
```

- 编译失败。
- 编程报错`cannot assign to struct field list["name"].Name in map`。
- 因为list[“name”]不是一个普通的指针值，map的value本身是不可寻址的，因为map中的值会在内存中移动，并且旧的指针地址在map改变时会变得无效。 定义的是var list map[string]Test，注意哦Test不是指针，而且map我们都知道是可以自动扩容的，那么原来的存储name的Test可能在地址A，但是如果map扩容了地址A就不是原来的Test了，所以go就不允许写数据。改为var list map[string]*Test。

16 指针

```
package main
import "fmt"
func main() {
  list := new([]int)
  list = append(list, 1)
  fmt.Println(list)
}
```

- 程序编译错误
- new出来的是指针类型，创建list使用make，list := make([]int,0)

17 结构体比较

```
package main

import "fmt"

func main() {
  sn1 := struct {
     age  int
     name string
  }{age: 11, name: "qq"}
  sn2 := struct {
     age  int
     name string
  }{age: 11, name: "qq"}

  if sn1 == sn2 {
     fmt.Println("sn1 == sn2")
  }
}
```

- 输出sn1 == sn2
- 进行结构体比较时候，只有相同类型的结构体才可以比较，结构体是否相同不但与属性类型个数有关，还与属性顺序相关。结构体相同的情况下还需要结构体属性都是可比较类型才可以使用"=="进行比较。

18 结构体比较

```
package main

import "fmt"

func main() {
sn1 := struct {
  age  int
  m   map[string]string
}{age: 11,m: map[string]string{"a": "1"}}
sn2 := struct {
  age  int
  m   map[string]string
}{age: 11, m: map[string]string{"a": "1"}}

if sn1 == sn2 {
  fmt.Println("sn1 == sn2")
}
}
```

- 编译错误
- 如果结构体包含 map slice chan 函数类型 不可以作为比较操作
- 进行结构体比较时候，只有相同类型的结构体才可以比较，结构体是否相同不但与属性类型个数有关，还与属性顺序相关。结构体相同的情况下还需要结构体属性都是可比较类型才可以使用"=="进行比较。

19 数据类型

```
package main
func main() {
  i := GetValue()
  switch i.(type) {
  case int:
     println("int")
  case string:
     println("string")
  case interface{}:
     println("interface")
  default:
     println("unknown")
  }
}
func GetValue() int {
  return 1
}
```

- 编译错误
- type 只能用于interface类型的判断

20 包名

```
1 package ಠ_ಠ
2 package 100
3 package ಠ~ಠ
4 package 你好
```

- 14声明是有效的
- 包名必须是有效的标识符，ʊ是一个有效字符，_也是也是一个有效字符，你好也是一个有效字符，100是数字，不可以直接用作包名，只能跟在有效字符后面，例如你好123，~不是一个包名声明的有效字符。

21 方法继承

```
package main
import "fmt"
type People struct{}
func (p *People) ShowA() {
  fmt.Println("showA")
  p.ShowB()
}
func (p *People) ShowB() {
  fmt.Println("showB")
}
type Teacher struct {
  People
}
func (t *Teacher) ShowB() {
  fmt.Println("teacher showB")
}
func main() {
  t := Teacher{}
  t.ShowA()
}
```

- 输出showA showB
- 这是Golang的组合模式，可以实现OOP的继承。 被组合的类型People所包含的方法虽然升级成了外部类型Teacher这个组合类型的方法（一定要是匿名字段），但它们的方法(ShowA())调用时接受者并没有发生变化。 此时People类型并不知道自己会被什么类型组合，当然也就无法调用方法时去使用未知的组合者Teacher类型的功能。

2 函数操作

01 函数名

```
package main

func f() {}
func f(a int) {}
func f() int { return 0}

func main() {}
```

- 编译错误
- f redeclared in this block previous declaration at prog.go:
- 一个包下不可以有两个相同名称的函数,init函数除外

02 函数名init

```go
package main

func init() {}
func init() {}

func main() {}
```

- 程序将会正常运行
- 一个包下不可以有两个相同名称的函数，init函数除外，编译期间，编译器会用唯一的后缀重写每个init函数的名称

03 函数名init

```go
func init()  {
  var pcs = make([]uintptr, 1)
  runtime.Callers(1, pcs[:])
  fn := runtime.FuncForPC(pcs[0])
  fmt.Println(fn.Name())
}

func init()  {
  var pcs = make([]uintptr, 1)
  runtime.Callers(1, pcs[:])
  fn := runtime.FuncForPC(pcs[0])
  fmt.Println(fn.Name())
}
```

- main.init.0 main.init.1
- 编译期间，编译器会用唯一的后缀重写每个init函数的名称。

04 函数名init

```go
package main

func init() {
   panic("first init")
}

func init() {
   panic("i am the second init")
}

func main() {}
```

- panic: first init

05 函数名init

```go
package main

import "fmt"

var x int
```

```
func init() {
   x++
}

func main() {
   init()
   fmt.Println(x)
}
```

- undefined: init
- init函数不可以被显示调用

06 匿名类型

```
package main

type T []int

func F(t T) {}

func main() {
var x []int
var y T
y = x
F(x)
_ = y
}
```

- 正常运行
- 匿名类型，可将类型转让。

07 匿名函数

```
package main

func test() []func()  {
var funs []func()
for i:=0;i<2 ;i++  {
     funs = append(funs, func() {
         println(&i,i)
     })
 }
 return funs
}

func main(){
funs:=test()
for _,f:=range funs{
     f()
 }
}
```

- 闭包延迟求值
- for循环复用局部变量i，每一次放入匿名函数的应用都是同一个变量。

结果：

```
0xc042046000 2
0xc042046000 2
```

如果想不一样：

```
func test() []func()  {
var funs []func()
for i:=0;i<2 ;i++  {
x:=i
funs = append(funs, func() {
   println(&x,x)
  })
  }
   return funs
  }

结果:
0xc000096000 0
0xc000096008 1
```

07 defer调用

```
package main

import "fmt"

func main()  {
  defer func() { fmt.Println("1") }()
  defer func() { fmt.Println("2") }()
  defer func() { fmt.Println("3") }()

  panic("触发异常")
}
```

- 输出321 触发异常
- 函数完成后defer按照从下到上顺序进行执行，类似栈操作

08 defer调用

```
package main

import "fmt"

func calc(index string, a, b int) int {
  ret := a + b
  fmt.Println(index, a, b, ret)
  return ret
}

func main() {
  a := 1
  b := 2
  defer calc("1", a, calc("3", a, b))
  a = 0
  defer calc("2", a, calc("4", a, b))
```

```
  b = 1
}
```

- 输出 3 1 2 3 , 4 0 2 2 , 2 0 2 2 , 1 1 3 4
- defer 会最后从下到上执行，在运行过程中defer所调用的参数的第三个参数是一个函数会顺序执行下来，所以执行顺序是3，4，2，1，并且按照执行时候的变量值进行

3 并发编程

01 死锁

```
package main

import "time"

func main() {
t := time.NewTicker(100)
for range t.C {
  t.Stop()
}
}
```

- 死锁
- Stop会关闭信号，但是不会关闭通道

02 协程

```
package main

import "fmt"

func main() {
  go func() { panic("Boom") }()
  for i := 0; i < 10000; i++ {
     fmt.Print(".")
  }
  fmt.Println("Done")
}
```

- 输出不一定多少个.后输出panic: Boom
- goroutine中抛出异常程序终止

03 select

```
package main

import (
  "time"
  "fmt"
)

func A() int {
  time.Sleep(100 * time.Millisecond)
  return 1
```

```
}

func B() int {
  time.Sleep(1000 * time.Millisecond)
  return 2
}

func main() {
  ch := make(chan int, 1)
  go func() {
     select {
     case ch <- A():
     case ch <- B():
     default:
        ch <- 3
     }
  }()
  fmt.Println(<-ch)
}
```

- 1 或 2
- 管道获取到返回值后完成了主函数，主函数在接收到第一个返回值前在进行阻塞等待

04 等待组

```
package main

import (
  "fmt"
  "runtime"
  "sync"
)

func main()  {
  runtime.GOMAXPROCS(1)
  wg := sync.WaitGroup{}
  wg.Add(10)
  for i := 0; i < 10; i++ {
     go func(i int) {
        fmt.Println(i)
        wg.Done()
     }(i)
  }
  wg.Wait()
}
```

- 如果是多核输出顺序随机
- go 执行是随机的，所以顺序不一定