课程安排，请关注**微信公众平台**或者**官方微博**

编程语言： Golang 与 html5

编程工具： Goland 和 HBuilder

预计平均一周左右更新一或二节课程

授人以鱼，不如授人以渔业。

大家好，

欢迎来到 **字节教育** 课程的学习

字节教育官网：www.ByteEdu.Com

腾讯课堂地址：Gopher.ke.qq.Com

技术交流群 ： 221 273 219

微信公众号 ：Golang 语言社区

微信服务号 ：Golang 技术社区

目录：

第一季　Golang 语言社区-综合面试题

第一节 课程简介

一、公众账号：

回复关键字：客服　　获取课程助教的微信

二、核心内容讲解：

Golang 精编 100 题

能力模型

级别

模型

初级

primary

熟悉基本语法，能够看懂代码的意图；

在他人指导下能够完成用户故事的开发，编写的代码符合 CleanCode 规范；

中级

intermediate

能够独立完成用户故事的开发和测试；

能够嗅出代码的坏味道，并知道如何重构达成目标；

高级

senior

能够开发出高质量高性能的代码；

能够熟练使用高级特性，开发编程框架或测试框架；

选择题

1. 【初级】下面属于关键字的是（）

A. func

B. def

C. struct

D. class

参考答案：AC

2. 【初级】定义一个包内全局字符串变量，下面语法正确的是（）

A. var str string

B. str := ""

C. str = ""

D. var str = ""

参考答案：AD

3. 【初级】通过指针变量 p 访问其成员变量 name，下面语法正确的是（）

A. p.name

B. (*p).name

C. (&p).name

D. p->name

参考答案：AB

4. 【初级】关于接口和类的说法，下面说法正确的是（）

A. 一个类只需要实现了接口要求的所有函数，我们就说这个类实现了该接口

B. 实现类的时候，只需要关心自己应该提供哪些方法，不用再纠结接口需要拆得多细才合理

C. 类实现接口时，需要导入接口所在的包

D. 接口由使用方按自身需求来定义，使用方无需关心是否有其他模块定义过类似的接口

参考答案：ABD

5. 【初级】关于字符串连接，下面语法正确的是（）

A. str := ‘abc’ + ‘123’

B. str := "abc" + "123"

C. str ： = '123' + "abc"

D. fmt.Sprintf("abc%d", 123)

参考答案：BD

6. 【初级】关于协程，下面说法正确是（）

A. 协程和线程都可以实现程序的并发执行

B. 线程比协程更轻量级

C. 协程不存在死锁问题

D. 通过 channel 来进行协程间的通信

参考答案：AD

7. 【中级】关于 init 函数，下面说法正确的是（）

A. 一个包中，可以包含多个 init 函数

B. 程序编译时，先执行导入包的 init 函数，再执行本包内的 init 函数

C. main 包中，不能有 init 函数

D. init 函数可以被其他函数调用

参考答案：AB

8. 【初级】关于循环语句，下面说法正确的有（）

A. 循环语句既支持 for 关键字，也支持 while 和 do-while

B. 关键字 for 的基本使用方法与 C/C++中没有任何差异

C. for 循环支持 continue 和 break 来控制循环，但是它提供了一个更高级的 break，可以选择中断哪一个循环

D. for 循环不支持以逗号为间隔的多个赋值语句，必须使用平行赋值的方式来初始化多个变量

参考答案：CD

9. 【中级】对于函数定义：

```
10. func add(args ...int) int {

11.  sum :=0

12.  for _,arg := range args {

13.     sum += arg

14.  }
```

15. returnsum

}

下面对 add 函数调用正确的是（）

A. add(1, 2)

B. add(1, 3, 7)

C. add([]int{1, 2})

D. add([]int{1, 3, 7}...)

参考答案：ABD

16. 【初级】关于类型转化，下面语法正确的是（）

A.

17. type MyInt int

18. var i int = 1

var jMyInt = i

B.

type MyIntint

var i int= 1

```
var jMyInt = (MyInt)i
```

C.

```
type MyIntint

var i int= 1

var jMyInt = MyInt(i)
```

D.

```
type MyIntint

var i int= 1

var jMyInt = i.(MyInt)
```

参考答案：C

19.【初级】关于局部变量的初始化，下面正确的使用方式是（）

A. var i int = 10

B. var i = 10

C. i := 10

D. i = 10

参考答案：ABC

20.【初级】关于 const 常量定义，下面正确的使用方式是（）

A.

21. const Pi float64 = 3.14159265358979323846

const zero= 0.0

B.

const (

size int64= 1024

eof = -1

)

C.

const (

ERR_ELEM_EXISTerror = errors.New("element already exists")

ERR_ELEM_NT_EXISTerror = errors.New("element not exists")

)

D.

const u, vfloat32 = 0, 3

const a,b, c = 3, 4, "foo"

参考答案：ABD

22.【初级】关于布尔变量 b 的赋值，下面错误的用法是（）

A. b = true

B. b = 1

C. b = bool(1)

D. b = (1 == 2)

参考答案：BC

23.【中级】下面的程序的运行结果是（）

```
24. func main() {

25. if (true) {

26.    defer fmt.Printf("1")
```

```
27. } else {

28.    defer fmt.Printf("2")

29. }

30. fmt.Printf("3")

}
```

A. 321

B. 32

C. 31

D. 13

参考答案：C

31. 【初级】关于 switch 语句，下面说法正确的有（）

A. 条件表达式必须为常量或者整数

B. 单个 case 中，可以出现多个结果选项

C. 需要用 break 来明确退出一个 case

D. 只有在 case 中明确添加 fallthrough 关键字，才会继续执行紧跟的下一个 case

参考答案：BD

32.【中级】golang 中没有隐藏的 this 指针，这句话的含义是（）

A. 方法施加的对象显式传递，没有被隐藏起来

B. golang 沿袭了传统面向对象编程中的诸多概念，比如继承、虚函数和构造函数

C. golang 的面向对象表达更直观，对于面向过程只是换了一种语法形式来表达

D. 方法施加的对象不需要非得是指针，也不用非得叫 this

参考答案：ACD

33.【中级】golang 中的引用类型包括（）

A. 数组切片

B. map

C. channel

D. interface

参考答案：ABCD

34.【中级】golang 中的指针运算包括（）

A. 可以对指针进行自增或自减运算

B. 可以通过“&”取指针的地址

C. 可以通过“*”取指针指向的数据

D. 可以对指针进行下标运算

参考答案：BC

35.【初级】关于 main 函数（可执行程序的执行起点），下面说法正确的是（）

A. main 函数不能带参数

B. main 函数不能定义返回值

C. main 函数所在的包必须为 main 包

D. main 函数中可以使用 flag 包来获取和解析命令行参数

参考答案：ABCD

36.【中级】下面赋值正确的是（）

A. var x = nil

B. var x interface{} = nil

C. var x string = nil

D. var x error = nil

参考答案：BD

37.【中级】关于整型切片的初始化，下面正确的是（）

A. s := make([]int)

B. s := make([]int, 0)

C. s := make([]int, 5, 10)

D. s := []int{1, 2, 3, 4, 5}

参考答案：BCD

38. 【中级】从切片中删除一个元素，下面的算法实现正确的是（）

A.

```
39. func (s *Slice)Remove(value interface{})error {

40. for i, v := range *s {

41.    if isEqual(value, v) {

42.        if i== len(*s) - 1 {

43.            *s = (*s)[:i]

44.        }else {

45.            *s = append((*s)[:i],(*s)[i + 2:]...)

46.        }

47.        return nil

48.    }

49. }

50. return ERR_ELEM_NT_EXIST
```

```
}
```

B.

```
func (s*Slice)Remove(value interface{}) error {

for i, v:= range *s {

    if isEqual(value, v) {

        *s =append((*s)[:i],(*s)[i + 1:])

        return nil

    }

}

returnERR_ELEM_NT_EXIST

}
```

C.

```
func (s*Slice)Remove(value interface{}) error {

for i, v:= range *s {

    if isEqual(value, v) {
```

```
        delete(*s, v)

        return nil

    }

}

returnERR_ELEM_NT_EXIST

}
```

D.

```
func (s*Slice)Remove(value interface{}) error {

for i, v:= range *s {

    if isEqual(value, v) {

        *s =append((*s)[:i],(*s)[i + 1:]...)

        return nil

    }

}

returnERR_ELEM_NT_EXIST
```

}

参考答案：D

51.【初级】对于局部变量整型切片 x 的赋值，下面定义正确的是（）

A.

52. x := []int{

53. 1, 2, 3,

54. 4, 5, 6,

}

B.

x :=[]int{

1, 2, 3,

4, 5, 6

}

C.

x :=[]int{

1, 2, 3,

4, 5, 6}

D.

x :=[]int{1, 2, 3, 4, 5, 6,}

参考答案：ACD

55. 【初级】关于变量的自增和自减操作，下面语句正确的是（）

A.

56. i := 1

i++

B.

i := 1

j = i++

C.

i := 1

++i

D.

i := 1

i--

参考答案：AD

57.【中级】关于函数声明，下面语法错误的是（）

A. func f(a, b int) (value int, err error)

B. func f(a int, b int) (value int, err error)

C. func f(a, b int) (value int, error)

D. func f(a int, b int) (int, int, error)

参考答案：C

58.【中级】如果 Add 函数的调用代码为：

59. func main() {

60. var a Integer = 1

```
61. var b Integer = 2

62. var i interface{} = &a

63. sum := i.(*Integer).Add(b)

64. fmt.Println(sum)

}
```

则 Add 函数定义正确的是（）

A.

```
typeInteger int

func (aInteger) Add(b Integer) Integer {

 return a + b

}
```

B.

```
typeInteger int

func (aInteger) Add(b *Integer) Integer {

 return a + *b
```

```
}
```

C.

```
typeInteger int

func (a*Integer) Add(b Integer) Integer {

 return *a + b

}
```

D.

```
typeInteger int

func (a*Integer) Add(b *Integer) Integer {

 return *a + *b

}
```

参考答案：AC

65. 【中级】如果 Add 函数的调用代码为：

```
66. func main() {

67. var a Integer = 1

68. var b Integer = 2

69. var i interface{} = a

70. sum := i.(Integer).Add(b)

71. fmt.Println(sum)

}
```

则 Add 函数定义正确的是（）

A.

```
typeInteger int

func (a Integer)Add(b Integer) Integer {

 return a + b

}
```

B.

```
typeInteger int
```

```
func (aInteger) Add(b *Integer) Integer {

 return a + *b

}
```

C.

```
typeInteger int

func (a*Integer) Add(b Integer) Integer {

 return *a + b

}
```

D.

```
typeInteger int

func (a*Integer) Add(b *Integer) Integer {

 return *a + *b

}
```

参考答案：A

72. 【中级】关于 GetPodAction 定义，下面赋值正确的是（）

```
73. type Fragment interface {

74. Exec(transInfo *TransInfo) error

75. }

76. type GetPodAction struct {

77. }

78. func (g GetPodAction) Exec(transInfo*TransInfo) error {

79. ...

80. return nil

}
```

A. var fragment Fragment =new(GetPodAction)

B. var fragment Fragment = GetPodAction

C. var fragment Fragment = &GetPodAction{}

D. var fragment Fragment = GetPodAction{}

参考答案：ACD

81.【中级】关于 GoMock，下面说法正确的是（）

A. GoMock 可以对 interface 打桩

B. GoMock 可以对类的成员函数打桩

C. GoMock 可以对函数打桩

D. GoMock 打桩后的依赖注入可以通过 GoStub 完成

参考答案：AD

82.【中级】关于接口，下面说法正确的是（）

A. 只要两个接口拥有相同的方法列表（次序不同不要紧），那么它们就是等价的，可以相互赋值

B. 如果接口 A 的方法列表是接口 B 的方法列表的子集，那么接口 B 可以赋值给接口 A

C. 接口查询是否成功，要在运行期才能够确定

D. 接口赋值是否可行，要在运行期才能够确定

参考答案：ABC

83.【初级】关于 channel，下面语法正确的是（）

A. var ch chan int

B. ch := make(chan int)

C. <- ch

D. ch <-

参考答案：ABC

84.【初级】关于同步锁，下面说法正确的是（）

A. 当一个 goroutine 获得了 Mutex 后，其他 goroutine 就只能乖乖的等待，除非该 goroutine 释放这个 Mutex

B. RWMutex 在读锁占用的情况下，会阻止写，但不阻止读

C. RWMutex 在写锁占用情况下，会阻止任何其他 goroutine（无论读和写）进来，整个锁相当于由该 goroutine 独占

D. Lock()操作需要保证有 Unlock()或 RUnlock()调用与之对应

参考答案：ABC

85.【中级】 golang 中大多数数据类型都可以转化为有效的 JSON 文本，下面几种类型除外（）

A. 指针

B. channel

C. complex

D. 函数

参考答案：BCD

86.【中级】关于 go vendor，下面说法正确的是（）

A. 基本思路是将引用的外部包的源代码放在当前工程的 vendor 目录下面

B. 编译 go 代码会优先从 vendor 目录先寻找依赖包

C. 可以指定引用某个特定版本的外部包

D. 有了 vendor 目录后，打包当前的工程代码到其他机器的$GOPATH/src 下都可以通过编译

参考答案：ABD

87.【初级】 flag 是 bool 型变量，下面 if 表达式符合编码规范的是（）

A. if flag == 1

B. if flag

C. if flag == false

D. if !flag

参考答案：BD

88.【初级】 value 是整型变量，下面 if 表达式符合编码规范的是（）

A. if value == 0

B. if value

C. if value != 0

D. if !value

参考答案：AC

89.【中级】关于函数返回值的错误设计，下面说法正确的是（）

A. 如果失败原因只有一个，则返回 bool

B. 如果失败原因超过一个，则返回 error

C. 如果没有失败原因，则不返回 bool 或 error

D. 如果重试几次可以避免失败，则不要立即返回 bool 或 error

参考答案：ABCD

90. 【中级】关于异常设计，下面说法正确的是（）

A. 在程序开发阶段，坚持速错，让程序异常崩溃

B. 在程序部署后，应恢复异常避免程序终止

C. 一切皆错误，不用进行异常设计

D. 对于不应该出现的分支，使用异常处理

参考答案：ABD

91. 【中级】关于 slice 或 map 操作，下面正确的是（）

A.

92. var s []int

s =append(s,1)

B.

var mmap[string]int

m["one"]= 1

C.

```
var s[]int

s =make([]int, 0)

s =append(s,1)
```

D.

```
var mmap[string]int

m =make(map[string]int)

m["one"]= 1
```

参考答案：ACD

93. 【中级】关于 channel 的特性，下面说法正确的是（）

A. 给一个 nil channel 发送数据，造成永远阻塞

B. 从一个 nil channel 接收数据，造成永远阻塞

C. 给一个已经关闭的 channel 发送数据，引起 panic

D. 从一个已经关闭的 channel 接收数据，如果缓冲区中为空，则返回一个零值

参考答案：ABCD

94. 【中级】关于无缓冲和有冲突的 channel，下面说法正确的是（）

A. 无缓冲的 channel 是默认的缓冲为 1 的 channel

B. 无缓冲的 channel 和有缓冲的 channel 都是同步的

C. 无缓冲的 channel 和有缓冲的 channel 都是非同步的

D. 无缓冲的 channel 是同步的，而有缓冲的 channel 是非同步的

参考答案：D

95. 【中级】关于异常的触发，下面说法正确的是（）

A. 空指针解析

B. 下标越界

C. 除数为 0

D. 调用 panic 函数

参考答案：ABCD

96. 【中级】关于 cap 函数的适用类型，下面说法正确的是（）

A. array

B. slice

C. map

D. channel

参考答案：ABD

97.【中级】关于 beego 框架，下面说法正确的是（）

A. beego 是一个 golang 实现的轻量级 HTTP 框架

B. beego 可以通过注释路由、正则路由等多种方式完成 url 路由注入

C. 可以使用 bee new 工具生成空工程，然后使用 bee run 命令自动热编译

D. beego 框架只提供了对 url 路由的处理，而对于 MVC 架构中的数据库部分未提供框架支持

参考答案：ABC

98.【中级】关于 goconvey，下面说法正确的是（）

A. goconvey 是一个支持 golang 的单元测试框架

B. goconvey 能够自动监控文件修改并启动测试，并可以将测试结果实时输出到 web 界面

C. goconvey 提供了丰富的断言简化测试用例的编写

D. goconvey 无法与 go test 集成

参考答案：ABC

99.【中级】关于 go vet，下面说法正确的是（）

A. go vet 是 golang 自带工具 go tool vet 的封装

B. 当执行 go vet database 时，可以对 database 所在目录下的所有子文件夹进行递归检测

C. go vet 可以使用绝对路径、相对路径或相对 GOPATH 的路径指定待检测的包

D. go vet 可以检测出死代码

参考答案：ACD

100. 【中级】关于 map，下面说法正确的是（）

A. map 反序列化时 json.unmarshal 的入参必须为 map 的地址

B. 在函数调用中传递 map，则子函数中对 map 元素的增加不会导致父函数中 map 的修改

C. 在函数调用中传递 map，则子函数中对 map 元素的修改不会导致父函数中 map 的修改

D. 不能使用内置函数 delete 删除 map 的元素

参考答案：A

101. 【中级】关于 GoStub，下面说法正确的是（）

A. GoStub 可以对全局变量打桩

B. GoStub 可以对函数打桩

C. GoStub 可以对类的成员方法打桩

D. GoStub 可以打动态桩，比如对一个函数打桩后，多次调用该函数会有不同的行为

参考答案：ABD

102. 【初级】关于 select 机制，下面说法正确的是（）

A. select 机制用来处理异步 IO 问题

B. select 机制最大的一条限制就是每个 case 语句里必须是一个 IO 操作

C. golang 在语言级别支持 select 关键字

D. select 关键字的用法与 switch 语句非常类似，后面要带判断条件

参考答案：ABC

103. 【初级】关于内存泄露，下面说法正确的是（）

A. golang 有自动垃圾回收，不存在内存泄露

B. golang 中检测内存泄露主要依靠的是 pprof 包

C. 内存泄露可以在编译阶段发现

D. 应定期使用浏览器来查看系统的实时内存信息，及时发现内存泄露问题

参考答案：BD

填空题

1. 【初级】声明一个整型变量 i_____

参考答案：var i int

2. 【初级】声明一个含有 10 个元素的整型数组 a_____

参考答案：var a [10]int

3. 【初级】声明一个整型数组切片 s_____

参考答案：var s []int

4. 【初级】声明一个整型指针变量 p_____

参考答案：var p *int

5. 【初级】声明一个 key 为字符串型 value 为整型的 map 变量 m_____

参考答案：var m map[string]int

6. 【初级】声明一个入参和返回值均为整型的函数变量 f_____

参考答案：var f func(a int) int

7. 【初级】声明一个只用于读取 int 数据的单向 channel 变量 ch_____

参考答案：var ch <-chan int

8. 【初级】假设源文件的命名为 slice.go，则测试文件的命名为_____

参考答案：slice_test.go

9. 【初级】 go test 要求测试函数的前缀必须命名为_____

参考答案：Test

10. 【中级】下面的程序的运行结果是_____

```
11. for i := 0; i < 5; i++ {

12. defer fmt.Printf("%d ", i)

}
```

参考答案：4 3 2 1 0

13. 【中级】下面的程序的运行结果是_____

```
14. func main() {

15. x := 1

16. {
```

```
17.    x := 2

18.    fmt.Print(x)

19. }

20. fmt.Println(x)

}
```

参考答案：21

21.【中级】下面的程序的运行结果是_____

```
22. func main() {

23. strs := []string{"one","two", "three"}

24.

25. for _, s := range strs {

26.    go func() {

27.        time.Sleep(1 * time.Second)
```

```
28.       fmt.Printf("%s ", s)

29.   }()

30. }

31. time.Sleep(3 * time.Second)

}
```

参考答案：three threethree

32. 【中级】下面的程序的运行结果是_____

```
33. func main() {

34. x := []string{"a", "b","c"}

35. for v := range x {

36.   fmt.Print(v)

37. }

}
```

参考答案：012

38. 【中级】下面的程序的运行结果是_____

```
39. func main() {

40. x := []string{"a", "b","c"}

41. for _, v := range x {

42.    fmt.Print(v)

43. }

}
```

参考答案：abc

44. 【初级】下面的程序的运行结果是_____

```
45. func main() {

46. i := 1

47. j := 2
```

48. i, j = j, i

49. fmt.Printf("%d%d\n", i, j)

}

参考答案：21

50. 【初级】下面的程序的运行结果是_____

51. func incr(p *int) int {

52. *p++

53. return *p

54. }

55. func main() {

56. v := 1

57. incr(&v)

58. fmt.Println(v)

}

参考答案：2

59. 【初级】启动一个 goroutine 的关键字是_____

参考答案：go

60. 【中级】下面的程序的运行结果是_____

```
61. type Slice []int

62. func NewSlice() Slice {

63. return make(Slice, 0)

64. }

65. func (s* Slice) Add(elem int) *Slice {

66. *s = append(*s, elem)

67. fmt.Print(elem)

68. return s
```

```
69. }

70. func main() {

71. s := NewSlice()

72. defer s.Add(1).Add(2)

73. s.Add(3)

}
```

参考答案：132

判断题

1. 【初级】数组是一个值类型（）

参考答案：T

2. 【初级】使用 map 不需要引入任何库（）

参考答案：T

3. 【中级】内置函数 delete 可以删除数组切片内的元素（）

参考答案：F

4. 【初级】指针是基础类型（）

参考答案：F

5. 【初级】 interface{}是可以指向任意对象的 Any 类型（）

参考答案：T

6. 【中级】下面关于文件操作的代码可能触发异常（）

```
7. file, err := os.Open("test.go")

8. defer file.Close()

9. if err != nil {

10. fmt.Println("open file failed:",err)
```

11. return

12. }

...

参考答案：T

13. 【初级】 Golang 不支持自动垃圾回收（）

参考答案：F

14. 【初级】 Golang 支持反射，反射最常见的使用场景是做对象的序列化（）

参考答案：T

15. 【初级】 Golang 可以复用 C/C++的模块，这个功能叫 Cgo（）

参考答案：F

16. 【初级】下面代码中两个斜点之间的代码，比如 json:"x"，作用是 X 字段在从结构体实例编码到 JSON 数据格式的时候，使用 x 作为名字，这可以看作是一种重命名的方式（）

17. type Position struct {

18. X int `json:"x"`

19. Y int `json:"y"`

20. Z int `json:"z"`

}

参考答案：T

21. 【初级】通过成员变量或函数首字母的大小写来决定其作用域（）

参考答案：T

22. 【初级】对于常量定义 zero(const zero = 0.0)，zero 是浮点型常量（）

参考答案：F

23. 【初级】对变量 x 的取反操作是~x（）

参考答案：F

24. 【初级】下面的程序的运行结果是 xello（）

25. func main() {

26. str := "hello"

27. str[0] = 'x'

28. fmt.Println(str)

}

参考答案：F

29. 【初级】 golang 支持 goto 语句（）

参考答案：T

30. 【初级】下面代码中的指针 p 为野指针，因为返回的栈内存在函数结束时会被释放（）

31. type TimesMatcher struct {

```
32. base int

33. }

34. func NewTimesMatcher(base int) *TimesMatcher{

35. return &TimesMatcher{base:base}

36. }

37. func main() {

38. p := NewTimesMatcher(3)

39. ...

}
```

参考答案：F

40. 【初级】匿名函数可以直接赋值给一个变量或者直接执行（）

参考答案：T

41.【初级】如果调用方调用了一个具有多返回值的方法，但是却不想关心其中的某个返回值，可以简单地用一个下划线“_”来跳过这个返回值，该下划线对应的变量叫匿名变量（）

参考答案：T

42.【初级】在函数的多返回值中，如果有 error 或 bool 类型，则一般放在最后一个（）

参考答案：T

43.【初级】错误是业务过程的一部分，而异常不是（）

参考答案：T

44.【初级】函数执行时，如果由于 panic 导致了异常，则延迟函数不会执行（）

参考答案：F

45.【中级】当程序运行时，如果遇到引用空指针、下标越界或显式调用 panic 函数等情况，则先触发 panic 函数的执行，然后调用延迟函数。调用者继续传递 panic，因此该过程一直在调用栈中重复发生：函数停止执行，调用延迟执行函数。如果一路在延迟函数中没有 recover 函数的调用，则会到达该携程的起点，该携程结束，然后终止其他所有携程，其他携程的终止过程也是重复发生：函数停止执行，调用延迟执行函数（）

参考答案：F

46.【初级】同级文件的包名不允许有多个（）

参考答案：T

47.【中级】可以给任意类型添加相应的方法（）

参考答案：F

48.【初级】 golang 虽然没有显式的提供继承语法，但是通过匿名组合实现了继承（）

参考答案：T

49.【初级】使用 for range 迭代 map 时每次迭代的顺序可能不一样，因为 map 的迭代是随机的（）

参考答案：T

50.【初级】 switch 后面可以不跟表达式（）

参考答案：T

51.【中级】结构体在序列化时非导出变量（以小写字母开头的变量名）不会被 encode，因此在 decode 时这些非导出变量的值为其类型的零值（）

参考答案：T

52.【初级】 golang 中没有构造函数的概念，对象的创建通常交由一个全局的创建函数来完成，以 NewXXX 来命名（）

参考答案：T

53.【中级】当函数 deferDemo 返回失败时，并不能 destroy 已 create 成功的资源（）

54. func deferDemo() error {

55. err := createResource1()

56. if err != nil {

57. return ERR_CREATE_RESOURCE1_FAILED

58. }

```
59. defer func() {
60.     if err != nil {
61.         destroyResource1()
62.     }
63. }()
64.
65. err = createResource2()
66. if err != nil {
67.     return ERR_CREATE_RESOURCE2_FAILED
68. }
69. defer func() {
70.     if err != nil {
71.         destroyResource2()
72.     }
73. }()
```

74.

75. err = createResource3()

76. if err != nil {

77. return ERR_CREATE_RESOURCE3_FAILED

78. }

79. return nil

}

参考答案：F

80.【中级】 channel 本身必然是同时支持读写的，所以不存在单向 channel（）

参考答案：F

81.【初级】 import 后面的最后一个元素是包名（）

参考答案：F

最近在很多地方看到了 golang 的面试题，看到了很多人对 Golang 的面试题心存恐惧，也是为了复习基础，我把解题的过程总结下来。

面试题

1 写出下面代码输出内容。

```
package main

import (

"fmt"

)

funcmain() {

    defer_call()

}

funcdefer_call() {

    deferfunc() {fmt.Println("打印前")}()

    deferfunc() {fmt.Println("打印中")}()

    deferfunc() {fmt.Println("打印后")}()

    panic("触发异常")

}
```

考点：defer 执行顺序

解答：

defer 是后进先出。

panic 需要等 defer 结束后才会向上传递。 出现 panic 恐慌时候，会先按照 defer 的后入先出的顺序执行，最后才会执行 panic。

```
打印后

打印中

打印前

panic: 触发异常
```

2 以下代码有什么问题，说明原因。

```
type student struct {

    Name string

    Age  int
```

```
}

funcpase_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

   }

}
```

考点：foreach

解答：

这样的写法初学者经常会遇到的，很危险！ 与 Java 的 foreach 一样，都是使用副本的方式。所以 m[stu.Name]=&stu 实际上一致指向同一个指针， 最终该指针的值为遍历的最后一个 struct 的值拷贝。 就像想修改切片元素的属性：

```
for _, stu := rangestus {

   stu.Age = stu.Age+10}
```

也是不可行的。 大家可以试试打印出来：

```
func pase_student() {

   m := make(map[string]*student)

   stus := []student{

      {Name: "zhou",Age: 24},

      {Name: "li",Age: 23},

      {Name: "wang",Age: 22},

   }
```

```
    // 错误写法
    for _,stu := range stus {
        m[stu.Name] =&stu
    }
     fork,v:=range m{
      println(k,"=>",v.Name)
    }
      // 正确
    for i:=0;i<len(stus);i++ {
       m[stus[i].Name] = &stus[i]
    }
     fork,v:=range m{
       println(k,"=>",v.Name)
    }
}
```

3 下面的代码会输出什么，并说明原因

```
func main() {
    runtime.GOMAXPROCS(1)
    wg := sync.WaitGroup{}
    wg.Add(20)   for i := 0; i < 10; i++ {
        gofunc() {
          fmt.Println("A: ", i)
          wg.Done()
        }()
    }
        for i:= 0; i < 10; i++ {
          gofunc(i int) {
```

```
        fmt.Println("B: ", i)
        wg.Done()
      }(i)
   }
   wg.Wait()
}
```

考点：go 执行的随机性和闭包

解答：

谁也不知道执行后打印的顺序是什么样的，所以只能说是随机数字。 但是 A:均为输出 10，B:从 0~9 输出(顺序不定)。第一个 go func 中 i 是外部 for 的一个变量，地址不变化。遍历完成后，最终 i=10。 故 go func 执行时，i 的值始终是 10。

第二个 go func 中 i 是函数参数，与外部 for 中的 i 完全是两个变量。 尾部(i)将发生值拷贝，go func 内部指向值拷贝地址。

4 下面代码会输出什么？

```
type People struct{}func (p *People)ShowA() {
   fmt.Println("showA")
   p.ShowB()
}
func(p*People)ShowB() {
   fmt.Println("showB")
}
typeTeacher struct {
   People
}
func(t*Teacher)ShowB() {
   fmt.Println("teachershowB")
}
funcmain() {
```

```
    t := Teacher{}

    t.ShowA()

}
```

考点：go 的组合继承

解答：

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。

showAshowB

5 下面代码会触发异常吗？请详细说明

```
func main() {

    runtime.GOMAXPROCS(1)

    int_chan := make(chanint, 1)

    string_chan := make(chanstring, 1)

    int_chan <- 1

    string_chan <- "hello"

    select {

          case value := <-int_chan:

       fmt.Println(value)

          casevalue := <-string_chan:

          panic(value)

    }

}
```

考点：select 随机性

解答：

select 会随机选择一个可用通用做收发操作。 所以代码是有肯触发异常，也有可能不会。 单个 chan 如果无缓冲时，将会阻塞。但结合 select 可以在多个 chan 间等待执行。有三点原则：

select 中只要有一个 case 能 return，则立刻执行。

当如果同一时间有多个 case 均能 return 则伪随机方式抽取任意一个执行。

如果没有一个 case 能 return 则可以执行”default”块。

6 下面代码输出什么？

```
funccalc(indexstring, a, bint) int {
   ret := a+ b
   fmt.Println(index,a, b, ret)
   return ret
}
funcmain() {
    a := 1
   b := 2
   defer calc("1", a,calc("10", a, b))    a = 0
   defer calc("2", a,calc("20", a, b))    b = 1
}
```

考点：defer 执行顺序

解答：

这道题类似第 1 题 需要注意到 defer 执行顺序和值传递 index:1 肯定是最后执行的，但是 index:1 的第三个参数是一个函数，所以最先被调用

calc("10",1,2)==>10,1,2,3 执行 index:2 时,与之前一样，需要先调用 calc("20",0,2)==>20,0,2,2 执行到 b=1 时候开始调用，index:2==>calc("2",0,2)==>2,0,2,2 最后执行 index:1==>calc("1",1,3)==>1,1,3,4

10 1 2 320 0 2 22 0 2 21 1 3 4

7 请写出以下输入内容

```
funcmain() {
      s := make([]int,5)
   s = append(s,1, 2, 3)
   fmt.Println(s)
}
```

考点：make 默认值和 append

解答：

make 初始化是由默认值的哦，此处默认值为 0

[00000123]

大家试试改为:

```
s := make([]int, 0)
s = append(s, 1, 2, 3)
fmt.Println(s)//[1 2 3]
```

8 下面的代码有什么问题?

```
type UserAges struct {
   ages map[string]int
   sync.Mutex
}
func(ua*UserAges)Add(name string, age int) {
   ua.Lock()
      deferua.Unlock()
   ua.ages[name] = age
}
```

```
func(ua*UserAges)Get(name string)int {
     ifage, ok := ua.ages[name]; ok {
       return age
   }
     return-1
}
```

考点：map 线程安全

解答：

可能会出现

```
fatal error: concurrent mapreadandmapwrite.
```

修改一下看看效果

```
func (ua *UserAges)Get(namestring)int {
   ua.Lock()
    deferua.Unlock()
    ifage, ok := ua.ages[name]; ok {
        return age
   }
     return-1
}
```

9. 下面的迭代会有什么问题？

```
func (set *threadSafeSet)Iter()<-chaninterface{} {
   ch := make(chaninterface{})
           gofunc() {
```

```
        set.RLock()
            for elem := range set.s {
           ch <- elem
        }
            close(ch)
        set.RUnlock()
    }()
     return ch
}
```

考点：chan 缓存池

解答：

看到这道题，我也在猜想出题者的意图在哪里。 chan?sync.RWMutex?go?chan 缓存池?迭代? 所以只能再读一次题目，就从迭代入手看看。 既然是迭代就会要求 set.s 全部可以遍历一次。但是 chan 是为缓存的，那就代表这写入一次就会阻塞。 我们把代码恢复为可以运行的方式，看看效果

```
package main
import (
     "sync"
    "fmt")//下面的迭代会有什么问题？type threadSafeSet struct {
    sync.RWMutex
    s []interface{}
}
func(set*threadSafeSet)Iter() <-chaninterface{} {
//ch := make(chan interface{}) // 解除注释看看！
    ch := make(chaninterface{},len(set.s))
gofunc() {
        set.RLock()
forelem,value := range set.s {
          ch <- elem
```

```
println("Iter:",elem,value)
        }       close(ch)
        set.RUnlock()
    }()
return ch
}
funcmain() {
    th:=threadSafeSet{
        s:[]interface{}{"1","2"},
    }
    v:=<-th.Iter()
    fmt.Sprintf("%s%v","ch",v)
}
```

10 以下代码能编译过去吗？为什么？

```
package main
import (   "fmt")
typePeople interface {
    Speak(string) string
}
typeStduent struct{}
func(stu*Stduent)Speak(think string)(talk string) {
ifthink == "bitch" {
        talk = "Youare a good boy"
    } else {
        talk = "hi"
    }
    return
```

```
}

funcmain() {

    var peoPeople = Stduent{}

    think := "bitch"

   fmt.Println(peo.Speak(think))

}
```

考点：golang 的方法集

解答：

编译不通过！ 做错了！？说明你对 golang 的方法集还有一些疑问。 一句话：golang 的方法集仅仅影响接口实现和方法表达式转化，与通过实例或者指针调用方法无关。

11 以下代码打印出来什么内容，说出为什么。

```
package main

import (   "fmt")

typePeople interface {

    Show()

}

typeStudent struct{}

func(stu*Student)Show() {

}

funclive()People {

    var stu*Student

    return stu

}

funcmain() {   if live() == nil

{

        fmt.Println("AAAAAAA")

    } else {
```

```
        fmt.Println("BBBBBBB")
    }
}
```

考点：interface 内部结构

解答：

很经典的题！ 这个考点是很多人忽略的 interface 内部结构。 go 中的接口分为两种一种是空的接口类似这样：

```
varininterface{}
```

另一种如题目：

```
type People interface {
    Show()
}
```

他们的底层结构如下：

```
type eface struct      //空接口
    _type *_type        //类型信息
    data  unsafe.Pointer //指向数据的指针(go 语言中特殊的指针类型 unsafe.Pointer 类似于 c 语言中的 void*)}
typeiface struct      //带有方法的接口
    tab  *itab          //存储 type 信息还有结构实现方法的集合
    data unsafe.Pointer  //指向数据的指针(go 语言中特殊的指针类型 unsafe.Pointer 类似于 c 语言中的 void*)}
type_type struct {
    size        uintptr //类型大小
    ptrdata     uintptr //前缀持有所有指针的内存大小
    hash        uint32  //数据 hash 值
    tflag      tflag
```

```
    align      uint8   //对齐

    fieldalign uint8   //嵌入结构体时的对齐

    kind       uint8   //kind 有些枚举值 kind 等于 0 是无效的

    alg       *typeAlg //函数指针数组，类型实现的所有方法

    gcdata    *byte    str       nameOff

    ptrToThis typeOff
}type itab struct {

    inter  *interfacetype //接口类型

    _type  *_type         //结构类型

    link   *itab

    bad    int32

    inhash int32

    fun    [1]uintptr     //可变大小方法集合}
```

可以看出 iface 比 eface 中间多了一层 itab 结构。 itab 存储_type 信息和[]fun 方法集，从上面的结构我们就可得出，因为 data 指向了 nil 并不代表 interface 是 nil， 所以返回值并不为空，这里的 fun(方法集)定义了接口的接收规则，在编译的过程中需要验证是否实现接口 结果：

BBBBBBB

12.是否可以编译通过？如果通过，输出什么？

```
func main() {

    i := GetValue() switch i.(type) {

        caseint:

        println("int")

        casestring:

        println("string")

        caseinterface{}:

        println("interface")

        default:
```

```
        println("unknown")
    }
}
funcGetValue()int {
return1
}
```

解析

考点：type

编译失败，因为 type 只能使用在 interface

13.下面函数有什么问题？

```
func funcMui(x,y int)(sum int,error){
returnx+y,nil
}
```

解析

考点：函数返回值命名

在函数有多个返回值时，只要有一个返回值有指定命名，其他的也必须有命名。 如果返回值有有多个返回值必须加上括号； 如果只有一个返回值并且有命名也需要加上括号； 此处函数第一个返回值有 sum 名称，第二个未命名，所以错误。

14.是否可以编译通过？如果通过，输出什么？

```
package mainfunc main() {    println(DeferFunc1(1)) println(DeferFunc2(1)) println(DeferFunc3(1))
}func DeferFunc1(i int)(t int) {
    t = i   deferfunc() {
        t += 3
```

```
    }() return t
}
funcDeferFunc2(i int)int {
    t := i  deferfunc() {
        t += 3
    }() return t
}
funcDeferFunc3(i int)(t int) {    deferfunc() {
        t += i
    }() return2}
```

解析

考点:defer 和函数返回值

需要明确一点是 defer 需要在函数结束前执行。 函数返回值名字会在函数起始处被初始化为对应类型的零值并且作用域为整个函数 DeferFunc1 有函数返回值 t 作用域为整个函数，在 return 之前 defer 会被执行，所以 t 会被修改，返回 4; DeferFunc2 函数中 t 的作用域为函数，返回 1;DeferFunc3 返回 3

15.是否可以编译通过？如果通过，输出什么？

```
funcmain() {    list := new([]int)
    list = append(list,1)
    fmt.Println(list)
}
```

解析

考点：new

```
list:=make([]int,0)
```

16.是否可以编译通过？如果通过，输出什么？

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

解析

考点：append

append 切片时候别漏了'…'

17.是否可以编译通过？如果通过，输出什么？

```
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")
   }
   sm1 := struct {
      age int
      m   map[string]string
   }{age: 11, m:map[string]string{"a": "1"}}
```

```
    sm2 := struct {
        age int
        m   map[string]string
    }{age: 11, m:map[string]string{"a": "1"}}
        if sm1 == sm2 {
        fmt.Println("sm1== sm2")
    }
}
```

解析

考点:结构体比较

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。

```
sn3:= struct {
    name string
    age  int
}
{age:11,name:"qq"}
```

sn3 与 sn1 就不是相同的结构体了，不能比较。 还有一点需要注意的是结构体是相同的，但是结构体属性中有不可以比较的类型，如 map,slice。 如果该结构属性都是可以比较的，那么就可以使用“==”进行比较操作。

可以使用 reflect.DeepEqual 进行比较

```
if reflect.DeepEqual(sn1, sm) {
    fmt.Println("sn1==sm")
}else {
    fmt.Println("sn1!=sm")
}
```

所以编译不通过： invalid operation: sm1 == sm2

18.是否可以编译通过？如果通过，输出什么？

```
func Foo(x interface{}) {    if x== nil {
       fmt.Println("emptyinterface")
         return
   }
   fmt.Println("non-emptyinterface")
}
      funcmain() {
      var x *int = nil
   Foo(x)
}
```

解析

考点：interface 内部结构

non-emptyinterface

19.是否可以编译通过？如果通过，输出什么？

```
func GetValue(m map[int]string, id int)(string, bool) {
        if _,exist := m[id]; exist {
          return"存在数据", true
   }
        returnnil, false}funcmain() {
```

```
    intmap:=map[int]string{
1:"a",
2:"bb",
3:"ccc",
    }
    v,err:=GetValue(intmap,3)
    fmt.Println(v,err)
}
```

解析

考点：函数返回值类型

nil 可以用作 interface、function、pointer、map、slice 和 channel 的“空值”。但是如果不特别指定的话，Go 语言不能识别类型，所以会报错。报:cannot use nil as type string in return argument.

20.是否可以编译通过？如果通过，输出什么？

```
const (
    x = iota
    y
    z = "zz"
    k
    p = iota)
funcmain()
{
    fmt.Println(x,y,z,k,p)
}
```

解析

考点：iota

结果:

0 1 zz zz 4

21.编译执行下面代码会出现什么?

```
package mainvar(
    size :=1024
    max_size = size*2)
funcmain() {
println(size,max_size)
}
```

解析

考点:变量简短模式

变量简短模式限制:

定义变量同时显式初始化

不能提供数据类型

只能在函数内部使用

结果:

```
syntaxerror: unexpected :=
```

22.下面函数有什么问题?

```
package main
const cl = 100
var bl   = 123
funcmain() {
```

println(&bl,bl)

println(&cl,cl)

}

解析

考点:常量

常量不同于变量的在运行期分配内存，常量通常会被编译器在预处理阶段直接展开，作为指令数据使用，

cannot take the address of cl

23.编译执行下面代码会出现什么?

package main

funcmain() {

for i:=0;i<10;i++ {

   loop:

println(i)

   }   gotoloop

}

解析

考点：goto

goto 不能跳转到其他函数或者内层代码

goto loop jumps intoblock starting at

24.编译执行下面代码会出现什么?

package main

```
import"fmt"

funcmain() {

 typeMyInt1 int

 typeMyInt2 = int

   var i int =9

   var i1MyInt1 = i

   var i2MyInt2 = i

   fmt.Println(i1,i2)

}
```

解析

考点：**Go 1.9 新特性 Type Alias **

基于一个类型创建一个新类型，称之为 defintion；基于一个类型创建一个别名，称之为 alias。 MyInt1 为称之为 defintion，虽然底层类型为 int 类型，但是不能直接赋值，需要强转； MyInt2 称之为 alias，可以直接赋值。

结果:

```
cannot use i (typeint) astype MyInt1 in assignment
```

25.编译执行下面代码会出现什么?

```
package main

import"fmt"

typeUser struct {

}

typeMyUser1 User

typeMyUser2 = User

func(iMyUser1)m1(){

   fmt.Println("MyUser1.m1")

}
```

```
func(iUser)m2(){
   fmt.Println("User.m2")
}
funcmain() {
   var i1MyUser1
   var i2MyUser2
   i1.m1()
   i2.m2()
}
```

解析

考点：**Go 1.9 新特性 Type Alias **

因为 MyUser2 完全等价于 User，所以具有其所有的方法，并且其中一个新增了方法，另外一个也会有。 但是

```
i1.m2()
```

是不能执行的，因为 MyUser1 没有定义该方法。 结果:

```
MyUser1.m1User.m2
```

26.编译执行下面代码会出现什么?

```
package main
import"fmt"
type T1 struct {
}
func(tT1)m1(){
   fmt.Println("T1.m1")
}
```

```
type T2= T1

typeMyStruct struct {

   T1

   T2

}

funcmain() {

   my:=MyStruct{}

   my.m1()

}
```

解析

考点：**Go 1.9 新特性 Type Alias **

是不能正常编译的,异常：

ambiguousselectormy.m1

结果不限于方法，字段也也一样；也不限于 type alias，type defintion 也是一样的，只要有重复的方法、字段，就会有这种提示，因为不知道该选择哪个。 改为:

```
my.T1.m1()

my.T2.m1()
```

type alias 的定义，本质上是一样的类型，只是起了一个别名，源类型怎么用，别名类型也怎么用，保留源类型的所有方法、字段等。

27.编译执行下面代码会出现什么?

```
package main

import (

    "errors"
```

```
   "fmt")
varErrDidNotWork = errors.New("did not work")
funcDoTheThing(reallyDoItbool)(errerror) {
ifreallyDoIt {
      result, err:= tryTheThing()
if err!= nil || result != "it worked" {
         err = ErrDidNotWork
      }
   }   return err
}
functryTheThing()(string,error) {
return"",ErrDidNotWork
}
funcmain() {
   fmt.Println(DoTheThing(true))
   fmt.Println(DoTheThing(false))
}
```

解析

考点：变量作用域

因为 if 语句块内的 err 变量会遮罩函数作用域内的 err 变量，结果：

改为：

```
func DoTheThing(reallyDoIt bool)(errerror) {
varresult string
   ifreallyDoIt {
      result, err =tryTheThing()
if err!= nil || result != "it worked" {
```

```
        err = ErrDidNotWork
      }
   }   return err
}
```

28.编译执行下面代码会出现什么?

```
package main
functest() []func() {
varfuns []func()
   fori:=0;i<2;i++ {
      funs = append(funs,func() {
         println(&i,i)
      })
   }   returnfuns
}
funcmain(){
   funs:=test()
     for_,f:=range funs{
      f()
   }
}
```

解析

考点：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想一个变量。 结果：

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

如果想不一样可以改为：

```
func test() []func()  {
varfuns []func()
   fori:=0;i<2;i++  {
      x:=i
      funs = append(funs,func() {
println(&x,x)
      })
   }    returnfuns
}
```

29.编译执行下面代码会出现什么?

```
package main
functest(x int)(func(),func()) {
returnfunc() {
println(x)
   x+=10
   }, func() {
     println(x)
   }
}
funcmain() {
   a,b:=test(100)
   a()
   b()
}
```

解析

考点：闭包引用相同变量*

结果：

100

110

30. 编译执行下面代码会出现什么?

```
package main
import (    "fmt"
    "reflect")
funcmain1() {
deferfunc() {
iferr:=recover();err!=nil{
        fmt.Println(err)
      }else {
        fmt.Println("fatal")
      }
   }()
deferfunc() {
panic("deferpanic")
   }()
panic("panic")
}
funcmain() {
deferfunc() {
iferr:=recover();err!=nil{
```

```
            fmt.Println("++++")
            f:=err.(func()string)
fmt.Println(err,f(),reflect.TypeOf(err).Kind().String())
        }else {
            fmt.Println("fatal")
        }
    }()
deferfunc() {
panic(func()string {
return "defer panic"
        })
    }()
panic("panic")
}
```

解析

考点：panic 仅有最后一个可以被 revover 捕获

触发 panic("panic")后顺序执行 defer，但是 defer 中还有一个 panic，所以覆盖了之前的 panic("panic")