Golang 开发工程师笔试题

一、 选择题

1. 下面属于关键字的是：

A. func

B. def

C. struct

D. class

2. 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 = ""

3. 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

4. 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

5. 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)

6. 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

7. 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

8. 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

9. 对于函数定义：

```
func add(args ...int) int {
    sum := 0
    for _, arg := range args {
       sum += arg
    }
    return sum
}
```

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}...)

10. 关于类型转化，下面语法正确的是：

A.

```
type MyInt int
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)
```

11. 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

12. 关于 const 常量定义，下面正确的使用方式是：

A.

```
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"
```

13. 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)

14. 下面的程序可能的运行结果是：

```
func main() {
    if true {
       defer fmt.Printf("1")
    } else {
       defer fmt.Printf("2")
    }
    fmt.Printf("3")
}
```

A. 321

B. 32

C. 31

D. 13

15. 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

16. 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

17. **golang 中的引用类型包括：**

A. 数组切片

B. map

C. channel

D. interface

18. 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

19. 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

20. 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
```

21. 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}
```

22. 从切片中删除一个元素，下面的算法实现正确的是：

A.

```
func (s *Slice) Remove(value interface{}) error {
    for i, v := range *s {
       if isEqual(value, v) {
           if i== len(*s) - 1 {
               *s = (*s)[:i]
           } else {
               *s = append((*s)[:i], (*s)[i + 2:]...)
           }
           return nil
       }
    }
    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
        }
    }
    return ERR_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
        }
    }
    return ERR_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
        }
    }
    return ERR_ELEM_NT_EXIST
}
```

23. 对于局部变量整型切片 x 的赋值，下面定义正确的是：

A.

```
x := []int{
    1, 2, 3,
    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,}
```

24. 关于变量的自增和自减操作，下面语句正确的是：

A.

```
i := 1
i++
```

B.

```
i := 1
j = i++
```

C.

```
i := 1
++i
```

D.

```
i := 1
i--
```

25. 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)`

26. 如果 Add 函数的调用代码为：

```
func main() {
    var a Integer = 1
    var b Integer = 2
    var i interface{} = &a
    sum := i.(*Integer).Add(b)
```

```
    fmt.Println(sum)
}
```

则 Add 函数定义正确的是：

A.

```
type Integer int
func (a Integer) Add(b Integer) Integer {
    return a + b
}
```

B.

```
type Integer int
func (a Integer) Add(b *Integer) Integer {
    return a + *b
}
```

C.

```
type Integer int
func (a *Integer) Add(b Integer) Integer {
    return *a + b
}
```

D.

```
type Integer int
func (a *Integer) Add(b *Integer) Integer {
    return *a + *b
}
```

27. 如果 Add 函数的调用代码为：

```
func main() {
    var a Integer = 1
    var b Integer = 2
    var i interface{} = a
    sum := i.(Integer).Add(b)
    fmt.Println(sum)
}
```

则 Add 函数定义正确的是：

A.

```
type Integer int
func (a Integer) Add(b Integer) Integer {
    return a + b
}
```

B.

```
type Integer int
func (a Integer) Add(b *Integer) Integer {
    return a + *b
}
```

C.

```
type Integer int
func (a *Integer) Add(b Integer) Integer {
    return *a + b
}
```

D.

```
type Integer int
func (a *Integer) Add(b *Integer) Integer {
    return *a + *b
}
```

28. 关于 GetPodAction 定义，下面赋值正确的是：

```
type Fragment interface {
    Exec(transInfo *TransInfo) error
}
type GetPodAction struct {
}
func (g GetPodAction) Exec(transInfo*TransInfo) error {
    ...
    return nil
}
```

A. `var fragment Fragment = new(GetPodAction)`

B. `var fragment Fragment = GetPodAction`

C. `var fragment Fragment = &GetPodAction{}`

D. `var fragment Fragment = GetPodAction{}`

29. 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

30. 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 <-`

31. 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

32. 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

33. 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

34. 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

35. 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

36. 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

37. 关于 slice 或 map 操作，下面正确的是：

A.

```
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
```

38. 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

39. 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

40. 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

41. 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

42. 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

43. 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

44. 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

二、 填空题

1. 声明一个整型变量 i _____

2. 声明一个含有 10 个元素的整型数组 a _____

3. 声明一个整型数组切片 s _____

4. 声明一个整型指针变量 p _____

5. 声明一个 key 为字符串型 value 为整型的 map 变量 m _____

6. 声明一个入参和返回值均为整型的函数变量 f _____

7. 声明一个只用于读取 int 数据的单向 channel 变量 ch _____

8. 假设源文件的命名为 slice.go，则测试文件的命名为_____

9. go test 要求测试函数的前缀必须命名为_____

10. 下面的程序的运行结果是_____

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

11. 下面的程序的运行结果是_____

```
func main() {
    x := 1
    {
       x := 2
       fmt.Print(x)
    }
    fmt.Println(x)
}
```

12. 下面的程序的运行结果是_____

```
func main() {
    strs := []string{"one","two", "three"}

    for _, s := range strs {
       go func() {
```

```
            time.Sleep(1 * time.Second)
            fmt.Printf("%s ", s)
        }()
    }
    time.Sleep(3 * time.Second)
}
```

13. 下面的程序的运行结果是_____

```
func main() {
    x := []string{"a", "b","c"}
    for v := range x {
        fmt.Print(v)
    }
}
```

14. 下面的程序的运行结果是_____

```
func main() {
    x := []string{"a", "b","c"}
    for _, v := range x {
        fmt.Print(v)
    }
}
```

15. 下面的程序的运行结果是_____

```
func main() {
    i := 1
    j := 2
    i, j = j, i
    fmt.Printf("%d%d\n", i, j)
}
```

16. 下面的程序的运行结果是_____

```
func incr(p *int) int {
    *p++
    return *p
}
func main() {
    v := 1
    incr(&v)
    fmt.Println(v)
}
```

17. 启动一个 goroutine 的关键字是_____

18．下面的程序的运行结果是_____

```
type Slice []int
func NewSlice() Slice {
    return make(Slice, 0)
}
func (s* Slice) Add(elem int) *Slice {
    *s = append(*s, elem)
    fmt.Print(elem)
    return s
}
func main() {
    s := NewSlice()
    defer s.Add(1).Add(2)
    s.Add(3)
}
```

三、 判断题

1． 数组是一个值类型：

2． 使用 map 不需要引入任何库：

3． 内置函数 delete 可以删除数组切片内的元素：

4． 指针是基础类型：

5． interface{}是可以指向任意对象的 Any 类型：

6． 下面关于文件操作的代码可能触发异常：

```
file, err := os.Open("test.go")
defer file.Close()
if err != nil {
    fmt.Println("open file failed:", err)
    return
}
...
```

7． Golang 不支持自动垃圾回收：

8． Golang 支持反射，反射最常见的使用场景是做对象的序列化：

9． Golang 可以复用 C/C++的模块，这个功能叫 Cgo：

10. 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：

```
type Position struct {
    X int `json:"x"`
    Y int `json:"y"`
    Z int `json:"z"`
}
```

11. 通过成员变量或函数首字母的大小写来决定其作用域：

12. 对于常量定义 zero(const zero = 0.0)，zero 是浮点型常量：

13. 对变量 x 的取反操作是~x：

14. 下面的程序的运行结果是 xello：

```
func main() {
    str := "hello"
    str[0] = 'x'
    fmt.Println(str)
}
```

15. golang 支持 goto 语句：

16. 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：

```
type TimesMatcher struct {
    base int
}
func NewTimesMatcher(base int) *TimesMatcher{
    return &TimesMatcher{base:base}
}
func main() {
    p := NewTimesMatcher(3)
    ...
}
```

17. 匿名函数可以直接赋值给一个变量或者直接执行：

18. 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：

19. 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：

20. 错误是业务过程的一部分，而异常不是：

21. 函数执行时，如果由于 panic 导致了异常，则延迟函数不会执行：

22. 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：

23. 同级目录下文件的包名不允许有多个：

24. 可以给任意类型添加相应的方法：

25. 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：

26. 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：

27. switch 后面可以不跟表达式：

28. 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：

29. 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：

30. 当函数 deferDemo 返回失败时，并不能 destroy 已 create 成功的资源：

```
func deferDemo() error {
    err := createResource1()
    if err != nil {
        return ERR_CREATE_RESOURCE1_FAILED
    }
    defer func() {
       if err != nil {
           destroyResource1()
       }
    }()

    err = createResource2()
    if err != nil {
        return ERR_CREATE_RESOURCE2_FAILED
    }
    defer func() {
       if err != nil {
           destroyResource2()
       }
```

```
    }()

    err = createResource3()
    if err != nil {
        return ERR_CREATE_RESOURCE3_FAILED
    }
    return nil
}
```

31. channel 本身必然是同时支持读写的，所以不存在单向 channel：

32. import 后面的最后一个元素是包名：

四、 简答题

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")
}
```

2. 以下代码有什么问题，说明原因

```
type student struct {
    Name string
    Age  int
}
func paseStudent() {
    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
    }
}
```

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++ {
         go func() {
             fmt.Println("A: ", i)
             wg.Done()
         }()
    }
    for i:= 0; i < 10; i++ {
           go func(i int) {
               fmt.Println("B: ", i)
               wg.Done()
           }(i)
    }
    wg.Wait()
}
```

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")
}
type Teacher struct {
    People
}
func (t *Teacher) ShowB() {
    fmt.Println("teachershowB")
}
func main() {
    t := Teacher{}
    t.ShowA()
}
```

5. 下面代码会触发异常吗？请详细说明

```
package main

import (
   "fmt"
   "runtime"
)

func main() {
   runtime.GOMAXPROCS(1)
   var (
      intChan    = make(chan int, 1)
      stringChan = make(chan string, 1)
   )
   intChan <- 1
   stringChan <- "hello"
   select {
   case value := <-intChan:
      fmt.Println(value)

   case value := <-stringChan:
      panic(value)
   }
}
```

6. 下面代码输出什么？

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

7. 请写出以下输入内容

```
func main() {
    s := make([]int, 5)
```

```
    s = append(s, 1, 2, 3)
    fmt.Println(s)
}
```

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()
    defer ua.Unlock()
    ua.ages[name] = age
}
func (ua *UserAges) Get(name string) int {
    if age, 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() <-chan interface{} {
    ch := make(chan interface{})
    go func() {
        set.RLock()
        for elem := range set.s {
           ch <- elem
        }
        close(ch)
        set.RUnlock()
    }()
    return ch
}
```

10. 以下代码能编译过去吗？为什么？

```
package main
import ("fmt")
type People interface {
    Speak(string) string
}
type Stduent struct{}
func (stu *Stduent) Speak(think string) (talk string) {
     if think == "golang" {
```

```
        talk = "You are a good boy"
    } else {
        talk = "hi"
    }
    return
}
func main() {
    var peo People = Stduent{}
    think := "golang"
    fmt.Println(peo.Speak(think))
}
```

11. 以下代码打印出来什么内容，说出为什么。

```
package main
import ("fmt")
type People interface {
    Show()
}
type Student struct{}
func (stu *Student) Show() {
}
func live() People {
    var stu *Student
    return stu
}
func main() {
     if live() == nil {
         fmt.Println("A")
     } else {
         fmt.Println("B")
     }
}
```

12. 是否可以编译通过？如果通过，输出什么？

```
func GetValue() int {
   return 1
}

func main() {
   i := GetValue()
   switch i.(type) {
   case int:
      println("int")
   case string:
```

```
      println("string")
   case interface{}:
      println("interface")
   default:
      println("unknown")
   }
}
```

13. 下面函数有什么问题？

```
func Mui(x, y int) (sum int, error) {
    return x+y, nil
}
```

14. 是否可以编译通过？如果通过，输出什么？

```
package main

func main() {
   println(DeferFunc1(1))
   println(DeferFunc2(1))
   println(DeferFunc3(1))
}
func DeferFunc1(i int) (t int) {
   t = i
   defer func() {
      t += 3
   }()
   return t
}
func DeferFunc2(i int) int {
   t := i
   defer func() {
      t += 3
   }()
   return t
}
func DeferFunc3(i int) (t int) {
   defer func() {
      t += i
   }()
   return 2
}
```

15. 是否可以编译通过？如果通过，输出什么？

```
func main() {
```

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

16. 是否可以编译通过？如果通过，输出什么？

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

17. 是否可以编译通过？如果通过，输出什么？

```
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")
   }
   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")
   }
}
```

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")
}
func main() {
    var x *int = nil
    Foo(x)
}
```

19. 是否可以编译通过？如果通过，输出什么？

```
package main

import "fmt"

func GetValue(m map[int]string, id int) (string, bool) {
   if _, exist := m[id]; exist {
      return "存在数据", true
   }
   return nil, false
}
func main() {
   intMap := map[int]string{
      1: "a",
      2: "bb",
      3: "ccc",
   }
   v, err := GetValue(intMap, 3)
   fmt.Println(v, err)
}
```

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
)
func main() {
    fmt.Println(x, y, z, k, p)
}
```

21. 编译执行下面代码会出现什么?

```
package main
var (
    size := 1024
    maxSize = size*2
)
func main() {
    println(size, maxSize)
}
```

22. 下面函数有什么问题？

```
package main
const cl = 100
var bl = 123
func main() {
    println(&bl, bl)
    println(&cl, cl)
}
```

23. 编译执行下面代码会出现什么?

```
func main() {
   for i := 0;i < 10;i++ {
loop:
      println(i)
   }
   goto loop
}
```

24. 编译执行下面代码会出现什么?

```
func main() {
   type MyInt1 int
   type MyInt2 = int
   var i int = 9
   var i1 MyInt1 = i
   var i2 MyInt2 = i
   fmt.Println(i1, i2)
}
```

25. 编译执行下面代码会出现什么?

```
package main

import "fmt"

type User struct {
}
type MyUser1 User
type MyUser2 = User

func (i MyUser1) m1() {
   fmt.Println("MyUser1.m1")
}
func (i User) m2() {
   fmt.Println("User.m2")
}
func main() {
   var i1 MyUser1
   var i2 MyUser2
   i1.m1()
   i2.m2()
}
```

26. 编译执行下面代码会出现什么?

```
package main
import"fmt"
type T1 struct {
}
func (t T1) m1() {
    fmt.Println("T1.m1")
}
type T2= T1
type MyStruct struct {
    T1
    T2
}
func main() {
    my := MyStruct{}
    my.m1()
}
```

27. 编译执行下面代码会出现什么?

```
package main

import (
   "errors"
   "fmt"
)

var ErrDidNotWork = errors.New("did not work")

func DoTheThing(reallyDoIt bool) (err error) {
   if reallyDoIt {
      result, err := tryTheThing()
      if err != nil || result != "it worked" {
         err = ErrDidNotWork
      }
   }
   return err
}
func tryTheThing() (string, error) {
   return "", ErrDidNotWork
}
func main() {
   fmt.Println(DoTheThing(true))
   fmt.Println(DoTheThing(false))
}
```

28. 编译执行下面代码会出现什么?

```
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()
   }
}
```

29. 编译执行下面代码会出现什么?

```
package main

func test(x int) (func(), func()) {
   return func() {
         println(x)
         x += 10
      }, func() {
         println(x)
      }
}
func main() {
   a, b := test(100)
   a()
   b()
}
```

30. 编译执行下面代码会出现什么?

```
package main

import (
   "fmt"
)

func main() {
   defer func() {
      if err := recover(); err != nil {
         fmt.Println(err)
      } else {
         fmt.Println("fatal")
      }
   }()
   defer func() {
      panic("defer panic")
   }()
   panic("panic")
}
```