1. = 和 := 的区别?

```
:= 声明+赋值

= 仅赋值
```

2. 指针的作用?

```
指针用来保存变量的地址。
```

3. Go 允许多个返回值吗?

允许

```
func swap(x, y string) (string, string) {
return y, x
}
```

4. Go 有异常类型吗?

Go 没有异常类型，只有错误类型（Error），通常使用返回值来表示异常状态。

5. 什么是协程（Goroutine）

Goroutine是与其他函数或方法同时运行的函数或方法。 Goroutines可以被认为是轻量级的线程。与线程相比，创建Goroutine的开销很小。 Go应用程序同时运行数千个Goroutine是非常常见的做法。

6. 如何高效地拼接字符串?

Go 语言中，字符串是只读的，也就意味着每次修改操作都会创建一个新的字符串。如果需要拼接多次，应使用 strings.Builder，最小化内存拷贝次数。

7. 什么是 rune 类型?

ASCII 码只需要 7 bit 就可以完整地表示，但只能表示英文字母在内的128个字符，为了表示世界上大部分的文字系统，发明了 Unicode，它是ASCII的超集，包含世界上书写系统中存在的所有字符，并为每个代码分配一个标准编号（称为Unicode CodePoint），在 Go 语言中称之为 rune，是 int32 类型的别名。

Go 语言中，字符串的底层表示是 byte (8 bit) 序列，而非 rune (32 bit) 序列。例如下面的例子中语和言使用 UTF-8编码后各占3个 byte，因此 len("Go语言") 等于 8，当然我们也可以将字符串转换为 rune 序列。

8. 如何判断 map 中是否包含某个 key ?

```
if val, ok := dict["foo"]; ok {
//do something here
}
```

9. Go 支持默认参数或可选参数吗?

Go 语言不支持可选参数（python 支持），也不支持方法重载（java支持）。

10. defer 的执行顺序

多个 defer 语句，遵从后进先出(Last In First Out，LIFO)的原则，最后声明的 defer 语句，最先得到执行。 defer 在 return 语句之后执行，但在函数退出之前，defer 可以修改返回值。

11. 如何交换 2 个变量的值?

```
a, b := "A", "B"
a, b = b, a
fmt.Println(a, b) // B A
```

12. Go 语言 tag 的用处?

tag 可以理解为 struct 字段的注解，可以用来定义字段的一个或多个属性。框架/工具可以通过反射获取到某个字段定义的属性，采取相应的处理方式。tag 丰富了代码的语义，增强了灵活性。

```
package main

import "fmt"
import "encoding/json"

type Stu struct {
  Name string `json:"stu_name"`
  ID string `json:"stu_id"`
  Age int `json:"-"`
}

func main() {
  buf, \_ := json.Marshal(Stu{"Tom", "t001", 18})
  fmt.Printf("%s\n", buf)
}
```

13. 如何判断 2 个字符串切片（slice) 是相等的?

go 语言中可以使用反射 reflect.DeepEqual(a, b) 判断 a、b 两个切片是否相等，但是通常不推荐这么做，使用反射非常影响性能。 通常采用的方式如下，遍历比较切片中的每一个元素.

14. 字符串打印时，%v 和 %+v 的区别

%v 和 %+v 都可以用来打印 struct 的值，区别在于 %v 仅打印各个字段的值，%+v 还会打印各个字段的名称。

15. Go 语言中如何表示枚举值(enums)

通常使用常量(const) 来表示枚举值。

```
type StuType int32
```

```
const (
  Type1 StuType = iota
  Type2
  Type3
  Type4
)

func main() {
  fmt.Println(Type1, Type2, Type3, Type4) // 0, 1, 2, 3
}
```

16. 空 struct{} 的用途

使用空结构体 `struct{}` 可以节省内存，一般作为占位符使用，表明这里并不需要一个值。

17. init() 函数是什么时候执行的?

`init()` 函数是 `Go` 程序初始化的一部分。`Go` 程序初始化先于 `main` 函数，由 `runtime` 初始化每个导入的包，初始化顺序不是按照从上到下的导入顺序，而是按照解析的依赖关系，没有依赖的包最先初始化。每个包首先初始化包作用域的常量和变量（常量优先于变量），然后执行包的 `init()` 函数。同一个包，甚至是同一个源文件可以有多个 `init()` 函数。`init()` 函数没有入参和返回值，不能被其他函数调用，同一个包内多个 `init()` 函数的执行顺序不作保证。

> 一句话总结： `import -> const -> var -> init() -> main()`

18. Go 语言的局部变量分配在栈上还是堆上? 由编译器决定。`Go` 语言编译器会自动决定把一个变量放在栈还是放在堆，编译器会做逃逸分析(`escape analysis`)，当发现变量的作用域没有超出函数范围，就可以在栈上，反之则必须分配在堆上。

```
func foo() \*int {
  v := 11
  return &v
}

func main() {
  m := foo()
  println(*m) // 11
}
```

`foo()` 函数中，如果 `v` 分配在栈上，`foo` 函数返回时，`&v` 就不存在了，但是这段函数是能够正常运行的。`Go` 编译器发现 `v` 的引用脱离了 `foo` 的作用域，会将其分在堆上。因此，`main` 函数中仍能够正常访问该值。

19. 2 个 interface 可以比较吗?

> `Go`语言中，`interface`的内部实现包含了`2`个字段，类型`T`和值`V`，`interface`可以使用 `==` 或 `!=` 比较。`2`个`interface`相等有以下`2`种情况

- 两个interface均等于 nil（此时V和T都处于unset 状态）
- 类型T相同，且对应的值V相等。

20 两个 nil 可能不相等吗?

可能。

接口(interface)是对非接口值(例如指针，struct等)的封装，内部实现包含2个字段，类型T和值V。一个接口等于nil，当且仅当T和V处于unset状态（T=nil，V is unset）。

两个接口值比较时，会先比较T，再比较V。 接口值与非接口值比较时，会先将非接口值尝试转换为接口值，再比较。

21. 简述 Go 语言GC(垃圾回收)的工作原理

最常见的垃圾回收算法有标记清除(Mark-Sweep) 和引用计数(Reference Count)，Go语言采用的是标记清除算法。并在此基础上使用了三色标记法和写屏障技术，提高了效率。

标记清除收集器是跟踪式垃圾收集器，其执行过程可以分成标记（Mark）和清除（Sweep）两个阶段：

标记阶段 — 从根对象出发查找并标记堆中所有存活的对象；
清除阶段 — 遍历堆中的全部对象，回收未被标记的垃圾对象并将回收的内存加入空闲链表。
标记清除算法的一大问题是在标记期间，需要暂停程序（Stop the world，STW），标记结束之后，用户程序才可以继续执行。为了能够异步执行，减少 STW 的时间，Go 语言采用了三色标记法。

三色标记算法将程序中的对象分成白色、黑色和灰色三类。

白色：不确定对象。
灰色：存活对象，子对象待处理。
黑色：存活对象。
标记开始时，所有对象加入白色集合（这一步需 STW ）。首先将根对象标记为灰色，加入灰色集合，垃圾搜集器取出一个灰色对象，将其标记为黑色，并将其指向的对象标记为灰色，加入灰色集合。重复这个过程，直到灰色集合为空为止，标记阶段结束。那么白色对象即可需要清理的对象，而黑色对象均为根可达的对象，不能被清理。

三色标记法因为多了一个白色的状态来存放不确定对象，所以后续的标记阶段可以并发地执行。当然并发执行的代价是可能会造成一些遗漏，因为那些早先被标记为黑色的对象可能目前已经是不可达的了。所以三色标记法是一个 false negative（假阴性）的算法。

三色标记法并发执行仍存在一个问题，即在 GC 过程中，对象指针发生了改变。比如下面的例子：

```
1
A（黑） -> B（灰） -> C（白） -> D（白）
```

正常情况下，D 对象最终会被标记为黑色，不应被回收。但在标记和用户程序并发执行过程中，用户程序删除了 C 对 D 的引用，而 A 获得了 D 的引用。标记继续进行，D 就没有机会被标记为黑色了（A 已经处理过，这一轮不会再被处理）。

```
1
2
3
```

```
A （黑） -> B （灰） -> C （白）
  ↓
 D （白）
```

为了解决这个问题，Go 使用了内存屏障技术，它是在用户程序读取对象、创建新对象以及更新对象指针时执行的一段代码，类似于一个钩子。垃圾收集器使用了写屏障（Write Barrier）技术，当对象新增或更新时，会将其着色为灰色。这样即使与用户程序并发执行，对象的引用发生改变时，垃圾收集器也能正确处理了。

一次完整的 GC 分为四个阶段：

1）标记准备(Mark Setup，需 STW)，打开写屏障(Write Barrier)
2）使用三色标记法标记（Marking, 并发）
3）标记结束(Mark Termination，需 STW)，关闭写屏障。
4）清理(Sweeping, 并发)
参考 fullstack

23. 函数返回局部变量的指针是否安全?

这在 Go 中是安全的，Go 编译器将会对每个局部变量进行逃逸分析。如果发现局部变量的作用域超出该函数，则不会将内存分配在栈上，而是分配在堆上。

24. 非接口的任意类型 T() 都能够调用 *T 的方法吗？反过来呢?

一个T类型的值可以调用为*T类型声明的方法，但是仅当此T的值是可寻址(addressable) 的情况下。编译器在调用指针属主方法前，会自动取此T值的地址。因为不是任何T值都是可寻址的，所以并非任何T值都能够调用为类型T声明的方法。 反过来，一个*T类型的值可以调用为类型T声明的方法，这是因为解引用指针总是合法的。事实上，你可以认为对于每一个为类型 T 声明的方法，编译器都会为类型*T自动隐式声明一个同名和同签名的方法。

哪些值是不可寻址的呢?

- 字符串中的字节;
- map 对象中的元素（slice 对象中的元素是可寻址的，slice的底层是数组）;
- 常量;
- 包级别的函数等。

举一个例子，定义类型 T，并为类型 *T 声明一个方法 hello()，变量 t1 可以调用该方法，但是常量 t2 调用该方法时，会产生编译错误。

```
type T string

func (t *T) hello() {
    fmt.Println("hello")
}

func main() {
    var t1 T = "ABC"
    t1.hello() // hello
    const t2 T = "ABC"
```

```
    t2.hello() // error: cannot call pointer method on t
}
```

无缓冲的 channel 和 有缓冲的 channel 的区别

对于无缓冲的 channel，发送方将阻塞该信道，直到接收方从该信道接收到数据为止，而接收方也将阻塞该信道，直到发送方将数据发送到该信道中为止。

对于有缓存的 channel，发送方在没有空插槽（缓冲区使用完）的情况下阻塞，而接收方在信道为空的情况下阻塞。

什么是协程泄露(Goroutine Leak)? 协程泄露是指协程创建后，长时间得不到释放，并且还在不断地创建新的协程，最终导致内存耗尽，程序崩溃。常见的导致协程泄露的场景有以下几种：

- 缺少接收器，导致发送阻塞
  这个例子中，每执行一次 query，则启动1000个协程向信道 ch 发送数字 0，但只接收了一次，导致 999 个协程被阻塞，不能退出。

```
func query() int {
    ch := make(chan int)
    for i := 0; i < 1000; i++ {
        go func() { ch <- 0 }()
    }
    return <-ch
}

func main() {
    for i := 0; i < 4; i++ {
        query()
        fmt.Printf("goroutines: %d\n", runtime.NumGoroutine())
    }
}
```

- 缺少发送器，导致接收阻塞
  那同样的，如果启动 1000 个协程接收信道的信息，但信道并不会发送那么多次的信息，也会导致接收协程被阻塞，不能退出。

- 死锁(dead lock)
  两个或两个以上的协程在执行过程中，由于竞争资源或者由于彼此通信而造成阻塞，这种情况下，也会导致协程被阻塞，不能退出。

- 无限循环(infinite loops)
  这个例子中，为了避免网络等问题，采用了无限重试的方式，发送 HTTP 请求，直到获取到数据。那如果 HTTP 服务宕机，永远不可达，导致协程不能退出，发生泄漏。

Go 可以限制运行时操作系统线程的数量吗?

可以使用环境变量 GOMAXPROCS 或 runtime.GOMAXPROCS(num int) 设置. 从官方文档的解释可以看到，GOMAXPROCS 限制的是同时执行用户态 Go 代码的操作系统线程的数量，但是对于被系统调用阻塞的线程数

量是没有限制的。GOMAXPROCS 的默认值等于 CPU 的逻辑核数，同一时间，一个核只能绑定一个线程，然后运行被调度的协程。因此对于 CPU 密集型的任务，若该值过大，例如设置为 CPU 逻辑核数的 2 倍，会增加线程切换的开销，降低性能。对于 I/O 密集型应用，适当地调大该值，可以提高 I/O 吞吐率。

下列代码的输出是

```
func main() {
    const (
        a, b = "golang", 100
        d, e
        f bool = true
        g
    )
    fmt.Println(d, e, g)
}
```

golang 100 true

在同一个 const group 中，如果常量定义与前一行的定义一致，则可以省略类型和值。编译时，会按照前一行的定义自动补全

```
func main() {
    const (
        a, b = "golang", 100
        d, e = "golang", 100
        f bool = true
        g bool = true
    )
    fmt.Println(d, e, g)
}
```

下列代码的输出是

```
func main() {
    const N = 100
    var x int = N

    const M int32 = 100
    var y int = M
    fmt.Println(x, y)
}
```

编译失败：cannot use M (type int32) as type int in assignment

Go 语言中，常量分为无类型常量和有类型常量两种，const N = 100，属于无类型常量，赋值给其他变量时，如果字面量能够转换为对应类型的变量，则赋值成功，例如，var x int = N。但是对于有类型的常量 const M

int32 = 100，赋值给其他变量时，需要类型匹配才能成功，所以显示地类型转换：

```
var y int = int(M)
```

下列代码的输出是

```
func main() {
    var a int8 = -1
    var b int8 = -128 / a
    fmt.Println(b)
}
```

-128

int8 能表示的数字的范围是 [-2^7, 2^7-1]，即 [-128, 127]。-128 是无类型常量，转换为 int8，再除以变量 -1，结果为 128，常量除以变量，结果是一个变量。变量转换时允许溢出，符号位变为1，转为补码后恰好等于 -128。

对于有符号整型，最高位是是符号位，计算机用补码表示负数。补码 = 原码取反加一。