**目录**

**Golang**

**数据类型**

**GO的数据类型?**

1. 整数: int
2. 浮点float
3. 布尔: bool
4. 切片: slice
5. 字典: map
6. 通道: channel
7. 字符串: string
8. 字节: byte
9. 结构体: struct
10. 自定义类型: type
11. 数组: array
12. 接口类型: interface{}

**GO有哪些引用数据类型？**

slice(切片)，map(字典)，channel(通道),func (函数)

GO的结构体是否可以比较?

GO是强类型语言，不同的结构体是不同的类型无法进行比较，但是可以比较结构体的数值来进行判断。

结构体中包含了引用数据也无法通过表达式 == 判断结构体是否相同

**切片**

**切片底层原理?**

切片由数组，长度，容量组成，底层结构是数组，可以通过append函数向切片内追加元素，底层容量不够时，会生成一个更大的数组来存放原数组和被追加的元素。append每次会返回一个新的切片。在不声明 cap 的情况下，默认 cap=len

**切片和数组的区别?**

**切片截取陷阱?**

切片的截取，在原切片不发生扩容的情况下，修改原切片数据或者截取切片数据，能够直接相互影响，切片截取的元素是和原切片引用的同一个地址。

append:

append,操作切片追加，切片初始化带大小的切片是按照尾部添加，并不会从第0个下标开(可能存在的误区)

append，通过截取切片进行中间插入数据需要特别注意，一定需要通过原切片构建一个新的切片才能够插入成功，否则应为源地址的关系切片会进行扩容导致数据重复

func main() { a1 := []int{1, 2, 4, 5, 6} a2 := a1[:2] a3 := a1[2:] fmt.Println(a1) fmt.Println(a2) a2 = append(a2, 3) fmt.Println(a2) a2 = append(a2, a3...) fmt.Println(a2)}out:[1 2 4 5 6][1 2]

[1 2 3]

[1 2 3 3 5 6]

该代码可能是初学者常犯的错误，截取的本质上是引用，问题将出现在被截取的下标上，原来的切片即使在中途被扩容了，引用还是没变

**切片扩容机制?**

* 当需要的容量超过原切片容量的两倍时，会使用需要的容量作为新容量。
* 当原切片长度小于1024时，新切片的容量会直接翻倍。而当原切片的容量大于等于1024时，会反复地增加25%，直到新容量超过所需要的容量。 切片截取：
* 切片的截取，截取到的新切片，在原切片没有发生过扩容的情况下(如果发生扩容，append将返回新的切片地址)，修改原切片或者被截取的切片元素的value对双方都会产生影响。说明元素地址是相同的，切片截取并没有生成新的元素(或者说没有新分配地址)。
* 字符串的截取，字符串的截取结果还是个字符串，保留字符串的特性。同样不可修改，这是不是意味着截取字符串的时候生成了新的字符串变量。 func main() { s := “abcdefg” s2 := s[:2] fmt.Println(s, ” “, s2) if &s == &s2 { fmt.Println(”==“) } a := []int{1, 2, 3, 4} a2 := a[:2] fmt.Println(a,” “, a2) if &a == &a2 { fmt.Println(”==“) } a2[0] = 0 fmt.Println(a,” “, a2) } out: abcdefg ab [1 2 3 4] [1 2] [0 2 3 4] [0 2] 通过上面的代码可以看出来，切片和字符串截取产生变量，都是新的变量，字符串的修改只能通过byte切片来进行。而其他类型，生成新的变量，但其元素是原来切片中的引用。

**Map**

底层实现:

遍历：

没有key序列情况下只能通过range方式进行无须遍历，map遍历的kv是无序的。

**MAP是否能够进行并发读写?**

引用类型的 map 是无法进行并发读写的,如果并发读写中没有锁的控制,必定会panic 一个并发读写的错误,想要对map进行并发读写,需要借助锁来加以控制或者使用官方包中提供的 sync.Map 来实现安全的读写操作,其底层也是通过map + 锁来实现的.

**字符串**

**GC**

**GO垃圾回收是如何实现的?**

[搞懂Go垃圾回收 - 掘金](https://juejin.im/post/5d56b47a5188250541792ede)

标记清除，复制清除，引用计数，分代回收

当前Golang使用的垃圾回收机制是**三色标记发**配合**写屏障**和**辅助GC**，**三色标记法**是**标记-清除法**的一种增强版本。

**三色标记-清除法 原理:**

1. 初始化所有变量都是白色
2. 找到程序中的root变量(root变量,是程序当前时刻运行到的栈和全局数据区域,比如进入了某个函数,在函数内部声明的数据就是这个栈的root数据),将root所引用的变量标记为灰色
3. 分析灰色变量是否引用了其他变量,分析阶段灰色变量不论是否引用过其他变量都将标记为黑色,如果有被引用的变量,则将被引用的变量标记为灰色
4. 重复执行第3步骤,当灰色变量个数为0时,剩余白色的变量意味着一轮GC分析结束就可以进行清除.

**GO的 GC运行原理:**

任何的GC都会发生 STW(暂停世界线,暂时停止用户线程的运行),GO中的GC是交替进行的,STW的影响对程序就比较小.

标记部分:

1. 初始化标记: 开启写屏障,辅助GC,统计root变量,这个过程会需要 STW
2. 开始对统计好的root变量进行扫描,其中包含了全局指针,和每个 Goroutine栈 上的指针(扫描对应的G栈会暂停当前G栈的运行),将其加入标记队列(灰色队列)，并循环处理灰色队列的对象，直到灰色队列为空。**该过程后台并行执行(后台标记的操作,意味着标记工作和用户线程同时进行的)**

完成标记以后:

完成标记以后,需要重新扫描全局指针和G栈,这个操作的原因是,标记过程和用户执行过程是并行的,在运行期间很可能产生新的内存分配和指针复制,重新扫描过程可以通过写屏障记录下来.

写屏障工作原理,在标记变量的时候会对每个变量的标记状态进行记录,在标记完成以后,进行重新扫描的时候再次更新这个标记记录,避免新分配的内存被错误的回收(一个例子,在代码开头定义了一个变量没有被其他数据引用,第一次标记的时刻没有被分配内存和数据引用,但是执行到后面的某一时刻时候却初始化了,并且被一些root进行了引用,此时如果没有第二次扫描,这个数据很可能在第一次标记结果里面被清除掉,产生严重的错误)

重新扫描完成以后,就可以根据结果进行回收所有的白色状态对象,这个过程在后台处理

等待上一轮GC清理的结束,然后开启下一轮GC

大家都会想到,即使有重新标记,也难保证其中会产生新的对象,当发现GC的速度跟不上分配的速度后,GO依然会选择暂停用户线程,优先进行GC处理,这就是辅助GC的作用

**何时会触发GC？**

内存分配达到阈值，每当内存扩大到一定量来触发GC

默认情况下最长2分钟自动触发GC

手动触发runtime.GC()

**如何GC调优？**

* GO自带的tool pprof能够生成性能查看到程序运行的内存情况，可以根据该工具进行权衡GC对程序的影响。
* 减少对象的分配，合理重复利用；
* 避免string与[]byte转化；
* 少量的小字符串可以用 + 来进行连接，大量小文本采用 strings.Join方式进行连接(就是对Builder 封装了一下)
* 大型文本字符串采用bytes.Buffer来经处理;
* 使用Builder 和 Buffer两者都是对内存的重复利用避免频繁的创建分配，根据业务情况也可以使用sync.Pool来管理经常创建的变量
* GO有自动垃圾回收机制，但是GC不是万能的，最好还是养成手动释放内存的习惯：比如手动把不再使用的内存释放，把对象置成nil，也可以考虑在合适的时候调用runtime.GC()触发GC。

**生么是逃逸?**

[Go变量逃逸分析 - itbsl - 博客园](https://www.cnblogs.com/itbsl/p/10476674.html)

逃逸分析，可以尽量把那些不需要分配到堆上的变量直接分配到栈上，堆上的变量少了，会减轻分配堆内存的开销，同时也会减少gc的压力，提高程序的运行速度。

**协程**

**进程，线程，Go程？**

**进程**

是计算机中一个单一的执行单元（一个程序在计算机中的最小执行单位），每个进程有独立的地址空间，是系统进行资源分配和调度的独立单位。操作系统中用于描述进程的数据结构是PCB（进程控制块，在整个操作系统中每个PCB都是唯一的）。进程的状态：创建，就绪，运行，终止。运行期间的状态转化：就绪（运行所需要的资源得到分配），运行（得到cpu时间片），等待（cpu时间片用完或者资源分配无法满足），就绪，运行…。（在进程资源的分配上面，银行家算法能够有效避免死锁）。一个进程下面可以包含多个线程。

进程调度算法：

先来先服务

轮询调度

最短作业进程

剩余时间最短

优先权调度

**线程**

操作系统中能够进行运算的最小单位。同一进程中的多条线程共享该进程中的全部系统资源。

上下文切换：

在进程和线程中都会发生上下文切换，上下文切换的过程就是在进程或者线程，阻塞或者使用完cpu调度时间片发生的转移交接下一个进程或者线程需要执行的信息并保存被切换的信息以便下次执行。线程的上下文切换是远小于进程上下文切换（其原因在于，线程的执行在同一片进程地址内部，只用切换一小部分缓存地址等信息即可，同理Go程也是如此只是粒度更细了）

**GO程**

是粒度更细的多任务调度，每个GO 关键字创建一个go程（可以看作一个执行任务），为什么说粒度更细，因为go程基于线程来调度任务。具体实现方式见MPG

**GO 线程模型(MPG)?**

参考文章：

[The Go scheduler - Morsing's blog](https://morsmachine.dk/go-scheduler)

!<https://morsmachine.dk/in-motion.jpg>

M: 操作系统调度的线程

P :表示运行时期的上下文，在模型中的角色是处理器，上下文存在的意义是，任务执行和传递（比如系统调用阻塞为保证剩余的Goroutine能够的以执行）。

G : 代表每个协程任务

MPG三者之间的关系：每个操作系统调度的线程都需要在持有一个上下文P才可以对协程进行调度，而在每个P上面都维护着一个任务队列，也就是Goroutines。除了每个上下文维护的Goroutines，还有一个全局的Goroutines队列会被周期性的检查调度。

GO进程的上下文P默认读取环境变量中的配置，也可以通过函数镜像配置。一般情况下GO上下文的最大运行上下文数量会和计算机CPU的核数相关。

处理器的执行：处理器正在调度的Goroutine每执行一条语句，就会放到被维护队列的末尾，然后调度队列中的下一个，以此往复。早期版本的MPG模型是公用一个全局队列来进行调度控制，通过锁来互斥等待，在线程和上下文处理器少的情况下体现不出问题来，当上下文处理器量一大，竞争激烈，性能就无法发挥出来。

**MPG调度中的处理策略是什么?**

循环调度：

循环调度，在一开始就已经提到了，每执行一个Goroutine的一行代码，就会切换任务，重复这样的操作。

系统调用：

!<https://morsmachine.dk/syscall.jpg>

在系统调度中，上下文处理器有重要的作用，当一个Goroutine某一时刻调度时候发生了系统调度导致阻塞。此时上下文处理器可以把当前系统调用的任务单独交给系统线程进行执行，上下文处理器将带着执行信息和维护的队列挂到其它系统线程上面进行执行，其它线程可能是需要新创建，也可能是之前处于睡眠状态的线程接管P继续进行工作，**系统调用的任务执行过后，全局任务队列这个时候就接管了此时的Goroutine。并且会周期性的进行检查调度。**

工作窃取：

!<https://morsmachine.dk/steal.jpg>

工作量窃取是，为了更加高效的执行任务，当一个上下文中的任务处理完毕，它会优先去全局队列中去寻找可执行的任务。全局队列没有待执行任务则会对正在执行任务中的上下文队列中拉取任务，拉取任务数量约为一半，保证每个调度器都在执行任务。

**锁：**

Mutx：属于悲观锁，读写冲突。需要等待读或者写完成了，才会释放。

RWMuxt：属于乐观锁，读操作不冲突，写写和读写操作冲突，适用于读多写少的场景。

**Channel**

**无缓冲Channel？**

无缓冲通道又叫通信通道，可以用于并发之间的语句排序，也可用于数据共享（go中的一句话，通过共享数据进行并发，而不是通过共享内存进行并发），无缓冲通道必须在多线程中使用，并且先接收，后发送。

**通道遍历？**

通过for range 阻塞读取通道数据，当channel被关闭时结束for range

**如何判断Channel 是否关闭？**

channel 第二个返回值 返回一个 bool ，true表示chan没有关闭。false表示关闭

**已关闭的Channel能否读取出数据？**

channel关闭状态依然可以读取到数据，当数据读取完之后，会读取到零值。

**重复关闭Channel?**

重复关闭同一个会 panic

**defer**

**panic会影响defer的执行吗？**

panic 不会影响 defer语句的执行顺序，函数遇到panic 后会离开函数，离开函数之前会调用defer 语句执行，然后输出panic信息

func main() { defer fmt.Println("1") defer fmt.Println("2") defer fmt.Println("3") defer fmt.Println("4") panic("0")}4321panic: 0

**defer 修改返回结果？**

**匿名返回**

一般值传递方式，不会改变返回结果，指针操作是可以修改返回结果的(前提返回类型是指针) **具名返回**

defer 直接修改 返回参数变量可以更改

**解析**

和defer执行时机，return返回机制有关，defer的确是在return之前执行，但return返回操作不是一个原子操作，匿名返回方式中return返回值会产生中间变量保存要返回的值所以之后的defer操作原来的值是无法影响到函数最终结果的，具名返回值return不会产生中间变量，defer操作的参数就是返回值本身所以会改变函数最终结果。

匿名的返回指针类型，defer操作指针类型是可以达到修改的目的。

**range**

可以通过range来对数组，切片，map以及通道进行遍历，是一种快捷的遍历方式，range返回值有两个，第一个返回值是索引值，第二个返回值是具体的值，对map来说返回值分别是k和v。channel的返回值特殊，只有一个，读取出channel的数据，channel通过range读取的时候会堵塞，直到channel关闭后才会结束range读取操作。

注意点：

* 数组，切片，map，range的第二个返回值，都会发生一次值拷贝，和通过索引或者key得到的效率不同。
* range在遍历的开始就已经确定了遍历的次数，中途对数组，切片类型的追加操作，是无法遍历到的。
* 对于map的range的遍历，不能保证是有序的。想要做到有序需要自行实现对mapkey的有序存储来达到一个有序遍历的效果。
* go1.22 版本之前的经典例子,迭代期间的 第二个返回值是一个共享变量会造成一下效果,1.22版本之后得到了改进,不会出现这样的情况
* func main() {
* arr := []int{1, 2, 3}
* myMap := make(map[int]\*int)
* for i, v := range arr {
* myMap[i] = &v
* }
* for \_, v := range myMap {
* fmt.Println(\*v)
* }
* }
* out：
* 3
* 3
* 3

**const**

itao itao的值是由 itao 所在 conts代码块中的第一个索引位置决定的，随后的未赋值的常量自增1，中间存在中断的位置将重新变化，如下

const (

a = iota // 0

b = iota // 1

)

const (

c = "menglu"

d = iota// 1

e = iota// 2

)

const (

h="c"

i="c"

j=iota// 2

k="c"

l=iota// 4

m //5

)

**panic**

**go运行期间那些行为会发生panic?**

1. 空指针
2. 数组|切片越界
3. 除以0
4. 重复关闭channel
5. 使用未初始化的map（即没有make分配）
6. 断言类型不匹配（解决方案，通过第二个参数判断是否断言成功）
7. 自定义panic **如何处理panic？** 程序中出现手动panic的地方，需要在panic之前使用defer 进行调用 recover 函数进行panic 错误捕捉，golang层面的panic同理。

**recover 处理panic注意事项?**

recover 函数必须放在panic的上一层函数中捕捉，只有在defer函数中直接调用才能够捕捉到。在defer的嵌套defer中也是如此，想要在协程中捕获panic必须在协程内部使用 **recover** ，否则无法捕捉。

func main() {

defer func() {

if err := recover(); err != nil {

fmt.Println(err)

}

}()

go func() {

panic("1")

}()

time.Sleep(time.Second \* 5)

}

正确做法

func main() {

go func() {

defer func() {

if err := recover(); err != nil {

fmt.Println(err)

}

}()

panic("1")

}()

time.Sleep(time.Second \* 5)

}

处理协程中的错误也可以借助官方的errgroup库来实现

package main

import (

"fmt"

"golang.org/x/sync/errgroup"

)

func main() {

group := errgroup.Group{}

for n := range 1000 {

group.Go(func() error {

if n == 100 {

return fmt.Errorf("err: %d", n)

}

println(n)

return nil

})

}

if err := group.Wait(); err != nil {

println(err.Error())

return

}

}

**go中的new() 和 make() 的区别？**

make函数专门用于创建和分配 go的引用类型数据 map ，silce，chan(通道) ，返回的内容是类型本身。

new 函数是用于创建一个类型并且分配内存空间，返回的是一个指向分配好内存空间的指针。

make能做的事情new不一定能替代，比如对 map和chan 的创建必须通过 make才能初始化完成使用

**import 的初始化？**

GO的导报顺序是更具 import 顺序依次初始化的,值得注意的是 特殊函数 init() 的执行在不同的包下调用的严格有序的,在同一个包下的多个 init() 函数执行没有明确的顺序,总的导报初始化顺序就是,初始化完成 import 的包以后初始化 包中的const常量部分,然后是包全局变量,接着是init()函数,最后开始执行main()函数.

**如何GO包循环依赖?**

循环依赖可能出现在编译阶段，两个不同的包相互依赖导致。出现 import A ，A import B，B import A，这样的死循环，编译阶段无法通过。

解决方案：

1. 需要避免两个包之间内部进行相互调用,例如 在 a.Func() 中调用 b.Func1() ,在b.Func1()中调用 a.Func2() 这样的情况.
2. 需要两个包都用到的函数提取为一个工具包进行调用即可

**GO中的switch 语句？**

switch 中的case选项可以有多个并列值(类型必须一致)，通过关键字fallthrough 强制执行紧跟的下一个case语句，并且不会判断case的正确性是否满足条件。

switch 也常用于对变量类型的判断 value.(type) 该语句仅在switch中可用。