# go专家编程

## Chan



一个chan同时只能被一个协程读或者写。

1. func makechan(t \*chantype, size int) \*hchan {

2. var c \*hchan

3. c = new(hchan)

4. c.buf = malloc(元素类型大小\*size)

5. c.elemsize = 元素类型大小

6. c.elemtype = 元素类型

7. c.dataqsiz = size

8.

9. return c

10. }

可以看出chan里面是有两个队列的，分别存着阻塞着的队列，就可以互相唤醒咯，反正就是这样了，反正那个unsafe的指针是可以存任意类型的指针的。

## Slice

7. func main() {

8. var array [10]int

9.

10. var slice = array[5:6]

11.

12. fmt.Println("lenth of slice: ", len(slice))

13. fmt.Println("capacity of slice: ", cap(slice))

14. fmt.Println(&slice[0] == &array[5])

15. }

这个len是1， 但是cap是5，后面的比较是true

注意，第二个是cap是5， 这个cap就是从起始位置，到真正的空间的结尾的位置。

append如果扩容了地址才会变的。

使用append向Slice追加元素时，如果Slice空间不足，将会触发Slice扩容，扩容实际上重新一配一块更大的内存，将原Slice数据拷贝进新Slice，然后返回新Slice，扩容后再将数据追加进去。

例如长度为10的切片拷贝到长度为5的切片时，将会拷贝5个元素。

也就是说，copy过程中不会发生扩容。

slice := array[start:end]

1. type slice struct {

2. array unsafe.Pointer

3. len int

4. cap int

5. }

使用make来创建Slice时，可以同时指定长度和容量，创建时底层会分配一个数组，数组的长度即容量。

## Map

map的实现是hash表。

1. type hmap struct {

2. count int // 当前保存的元素个数

3. ...

4. B uint8 // 指示bucket数组的大小

5. ...

6. buckets unsafe.Pointer // bucket数组指针，数组的大小为2^B

7. ...

8. }

buckets就是个bucket的数组，所以这里也是用的unsafe的指针。

1. type bmap struct {

2. tophash [8]uint8 //存储哈希值的高8位

3. data byte[1] //key value数据:key/key/key/.../value/value/value...

4. overflow \*bmap //溢出bucket的地址

5. }

这是其中一个的bucket的结构

可以看到，一个bucket并不是只存一个k-v，而是存8个，后面也是bucket做的拉链表。

注意放kv的时候也是key和value分开放，这样可以防止字节对不齐。

负载因子 = 键数量/bucket数量

哈希表需要将负载因子控制在合适的大小，超过其阀值需要进行rehash，也即键值对重新组织：

哈希因子过小，说明空间利用率低

哈希因子过大，说明冲突严重，存取效率低

每个哈希表的实现对负载因子容忍程度不同，比如Redis实现中负载因子大于1时就会触发rehash，而Go则在在负载因子达到6.5时才会触发rehash，因为Redis的每个bucket只能存1个键值对，而Go的bucket可能存8个键值对，所以Go可以容忍更高的负载子。

当负载因子过大时，就新建一个bucket，新的bucket长度是原来的2倍，然后旧bucket数据搬迁到新的bucket。

考虑到如果map存储了数以亿计的key-value，一次性搬迁将会造成比较大的延时，Go采用逐步搬迁策略，即每次访问map时都会触发一次搬迁，每次搬迁2个键值对。

如果查找不到，也不会返回空值，而是返回相应类型的0值。

如果当前处于搬迁过程，则优先从oldbuckets查找

## String

1. type stringStruct struct {

2. str unsafe.Pointer

3. len int

4. }

string数据结构跟切片有些类似，只不过切片还有一个表示容量的成员，事实上string和切片，准确的说是byte切片经常发生转换。

字符串构建过程是先跟据字符串构建stringStruct，再转换成string

string在runtime包中就是stringStruct，对外呈现叫做string。

1. func gostringnocopy(str \*byte) string { // 跟据字符串地址构建string

2. ss := stringStruct{str: unsafe.Pointer(str), len: findnull(str)} // 先构造stringStruct

3. s := \*(\*string)(unsafe.Pointer(&ss)) // 再将stringStruct转换成string

4. return s

5. }

这个unsafe.pointer是可以转换成各种类型的指针的。

字符串和[]byte 互相转换其实都是内存拷贝的。

频繁的字符串+拼接这样效率是很低的，如果要是正经拼接的话的是strings.builder。

像C++语言中的string，其本身拥有内存空间，修改string是支持的。但Go的实现中，string不包含内存空间，只有一个内存的指针，这样做的好处是string变得非常轻量，可以很方便的进行传递而不用担心内存拷贝。

因为string通常指向字符串字面量，而字符串字面量存储位置是只读段，而不是堆或栈上，所以才有了string不可修改的约定。

byte切片转换成string的场景很多，为了性能上的考虑，有时候只是临时需要字符串的场景下，byte切片转换成string时并不会拷贝内存，而是直接返回一个string，这个string的指针(string.str)指向切片的内存。

比如，编译器会识别如下临时场景：

使用m[string(b)]来查找map（map是string为key，临时把切片b转成string）；

字符串拼接，如”<” + “string(b)” + “>”；

字符串比较：string(b) == “foo”

因为是临时把byte切片转换成string，也就避免了因byte切片同容改成而导致string引用失败的情况

## Defer

defer语句用于延迟函数的调用，每次defer都会把一个函数压入栈中，函数返回前再把延迟的函数取出并执行。 为了方便描述，我们把创建defer的函数称为主函数，defer语句后面的函数称为延迟函数。

1. func deferFuncParameter() {

2. var aInt = 1

3.

4. defer fmt.Println(aInt)

5.

6. aInt = 2

7. return

8. }

这个输出的是1，因为这个是走的参数，我们以前的那种走的是闭包defer引用了外面的变量，而不是参数传进来的。

1. func deferFuncReturn() (result int) {

2. i := 1

3.

4. defer func() {

5. result++

6. }()

7.

8. return i

9. }

这个最终会返回2.

return不是原子的，会先把i赋值给result，然后在执行defer里面的内容。

注意是这个流程。

这种函数带有具名返回值的就是这个流程。

1. func foo() int {

2. var i int

3.

4. defer func() {

5. i++

6. }()

7.

8. return i

9. }

注意这里最后是0，

流程还是一样的，先把i赋个一个匿名的result，defer又去把i给++了，最后返回的是匿名result，是这个流程的。

1. type \_defer struct {

2. sp uintptr //函数栈指针

3. pc uintptr //程序计数器

4. fn \*funcval //函数地址

5. link \*\_defer //指向自身结构的指针，用于链接多个defer

6. }

这是一个defer的链表

deferproc()： 在声明defer处调用，其将defer函数存入goroutine的链表中；

deferreturn()：在return指令，准确的讲是在ret指令前调用，其将defer从goroutine链表中取出并执行。

## Select

select是Golang在语言层面提供的多路IO复用的机制

## Mutex

1. type Mutex struct {

2. state int32

3. sema uint32

4. }

Mutex.state表示互斥锁的状态，比如是否被锁定等。

Mutex.sema表示信号量，协程阻塞等待该信号量，解锁的协程释放信号量从而唤醒等待信号量的协程



Locked: 表示该Mutex是否已被锁定，0：没有锁定 1：已被锁定。

Woken: 表示是否有协程已被唤醒，0：没有协程唤醒 1：已有协程唤醒，正在加锁过程中。

Starving：表示该Mutex是否处理饥饿状态， 0：没有饥饿 1：饥饿状态，说明有协程阻塞了超过1ms。 这里就可以统计有多少饥饿的协程。

Waiter: 表示阻塞等待锁的协程个数，协程解锁时根据此值来判断是否需要释放信号量。

协程之间抢锁实际上是抢给Locked赋值的权利，能给Locked域置1，就说明抢锁成功。抢不到的话就阻塞等待Mutex.sema信号量，一旦持有锁的协程解锁，等待的协程会依次被唤醒。



自旋对应于CPU的”PAUSE”指令，CPU对该指令什么都不做，相当于CPU空转，对程序而言相当于sleep了一小段时间，时间非常短，当前实现是30个时钟周期。

自旋过程中会持续探测Locked是否变为0，连续两次探测间隔就是执行这些PAUSE指令，它不同于sleep，不需要将协程转为睡眠状态。

## Rwmutex

1. type RWMutex struct {

2. w Mutex //用于控制多个写锁，获得写锁首先要获取该锁，如果有一个写锁在进行，那么再到来的写锁将会阻塞于此

3. writerSem uint32 //写阻塞等待的信号量，最后一个读者释放锁时会释放信号量

4. readerSem uint32 //读阻塞的协程等待的信号量，持有写锁的协程释放锁后会释放信号量

5. readerCount int32 //记录读者个数

6. readerWait int32 //记录写阻塞时读者个数

7. }

可以看到，里面还是有一个mutex的。

## Gmp

空闲的P会将其他P中的G偷取一部分过来，一般每次偷取一半。

## Timer

func DelayFunction() {

2. timer := time.NewTimer(5 \* time.Second)

3.

4. select {

5. case <- timer.C:

6. log.Println("Delayed 5s, start to do something.")

7. }

8. }

延迟执行可以这样操作，不一定非得timesleep

1. func AfterDemo() {

2. log.Println(time.Now())

3. <- time.After(1 \* time.Second)

4. log.Println(time.Now())

5. }

这样也是可以的做一个。

一次性定时器Timer和周期性定时器Ticker，这两种定时器内部实现机制完全相同。创建定时器的协程并不负责计时，而是把任务交给系统协程，系统协程统一处理所有的定时器。

1. type timer struct {

2. tb \*timersBucket // the bucket the timer lives in // 当前定时器寄存于系统timer堆的地址

3. i int // heap index // 当前定时器寄存于系统timer堆的下标

4.

5. when int64 // 当前定时器下次触发时间

6.

period int64 // 当前定时器周期触发间隔（如果是Timer，间隔为0，表示不重复触

发）

7. f func(interface{}, uintptr) // 定时器触发时执行的函数

8. arg interface{} // 定时器触发时执行函数传递的参数一

9. seq uintptr // 定时器触发时执行函数传递的参数二(该参数只在网络收发场景下使用)

10. }

会有一个叫做timerbucket的结构，所有的timer都放在这里面，然后有专门的系统协程去调度这些timer。

这些timer会被存到一个小顶堆里面，然后距离执行时间最近的timer放在堆顶，这样去调度就可以了。

## TIPS

Go的struct声明允许字段附带 Tag 来对字段做一些标记。

该 Tag 不仅仅是一个字符串那么简单，因为其主要用于反射场景， reflect 包中提供了操作 Tag 的方法，所以 Tag 写法也要遵循一定的规则。