代码随想录知识星球精华（最强八股文）第四版（Go篇））

代码随想录知识星球精华（最强八股文）第四版为九份PDF，分别是：

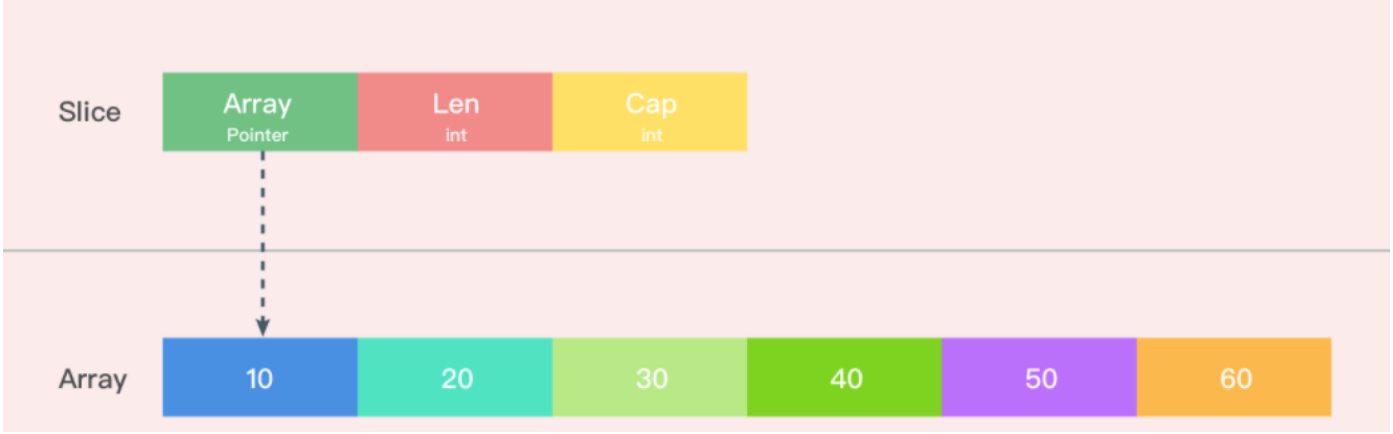
- 代码随想录知识星球八股文概述
- C++篇
- go篇
- Java篇
- 前端篇
- 算法题篇
- 计算机基础篇
- 问答精华篇
- 面经篇

本篇为最强八股文之Go篇。

简述slice的底层原理，和数组的区别是什么

slice内部的是通过指针引用一个底层数组，是对数组一个连续片段的引用，另外还有长度和容量两个变量

其数据结构如下：



Slice是可变长度的数组，其长度是基于底层数组的长度的，如果底层数组的长度不足以满足需求，可以进行扩容，扩容其实就是另外开一个数组把当前切片的内容copy过去， 扩容策略简单来说是以下规则：

如果新申请的容量大于两倍的旧容量，那么最终容量就是新申请的容量大小

如果不大于两倍旧容量，则判断旧容量是否小于1024，如果是则最终容量是旧容量的两倍

如果旧容量大于等于1024，就从旧容量大小开始每次增加现有旧容量的四分之一，直到满足需求为止

slice和数组的区别

- 声明时数组要指定长度或者是写 ...， 而slice方括号中为空
- 数组声明后可以不进行初始化，其内元素已经为默认零值，而slice需要初始化才能使用，不初始化时为空
- 函数传参时参数为数组是传的一个数组的copy，改变形参不会对原数组产生影响；而参数为slice时因为slice本身结构中包含了数组指针，因此改变slice形参可以改变底层的数组，同时在传slice时要注意slice的扩容问题

channel的发送和接收操作有哪些基本特性?

1. **对同一个通道，发送操作之间是互斥的，接收操作之间也是互斥的。**

即同一时刻，go的runtime只会执行对同一个通道的任意个发送操作中的某一个，直到这个发送的元素值被完全复制进该通道之后，其他发送操作才可能被执行。接收操作类似。

2. **发送操作和接收操作中对元素值的处理都是不可分割的**

即发送操作和接收操作都是原子的。例如接收操作时元素值从通道移动到外界，这个移动操作包含了两步，第一步是生成正在通道中的这个元素值的副本，并准备给到接收方，第二步是删除在通道中的这个元素值，这两个操作会一起完成。类似于innodb的事务机制。

这样既是为了保证通道中元素值的完整性，也是为了保证通道操作的唯一性

3. **发送操作在完全完成之前会被阻塞，接收操作也是。**

类似于接收操作，发送操作也是包括了复制元素值和放置副本到通道内部两个步骤。在这两个步骤完全完成之前，发起这个操作的那句代码会一直阻塞在那里，在通道完成发送操作之后，runtime系统会通知这句代码所在的goroutine，解除阻塞，以使它去争取继续运行代码的机会。如此阻塞代码也是为了实现操作的互斥和元素值的完整。

扩展

1. 发送操作和接收操作在什么时候可能会被长时间阻塞?

对于缓冲通道来说，如果通道已满，则对它的所有发送操作都将被阻塞，直到通道中有元素值被接收走，此时通道会通知阻塞队列的首个goroutine，通知顺序是公平的

相对的，如果通道已空，那么对它的所有接收操作会被阻塞，直到通道中有新的元素值出现。

对于非缓冲通道，无论是发送还是接收操作，一开始执行就会被阻塞，直到配对的操作也开始执行，才会继续。可以说非缓冲通道就是在用同步的方式传递数据。且用非缓冲通道时，数据并不会用通道作中转。

如果错误操作也会造成长时间阻塞，最典型的就是对值为nil（即未初始化）的通道进行操作。

2. 发送操作和接收操作什么时候会引发panic?

对一个已经关闭的通道做发送操作会引发panic，但对已经关闭的通道可以进行接收操作

对一个已经关闭的通道进行关闭操作会引发panic。

defer底层原理

1、每次defer语句在执行的时候，都会将函数进行"压栈"，函数参数会被拷贝下来。当外层函数退出时，defer函数会按照定义的顺序逆序执行 。如果defer执行的函数为nil，那么会在最终调用函数中产生panic。

2、为什么defer要按照定义的顺序逆序执行

后面定义的函数可能会依赖前面的资源，所以要先执行。如果前面先执行，释放掉这个依赖，那后面的函数就找不到它的依赖了。

3、defer函数定义时，对外部变量的引用方式有两种

分别是函数参数以及作为闭包引用。

在作为函数参数的时候，在defer定义时就把值传递给defer，并被缓存起来。

如果是作为闭包引用，则会在defer真正调用的时候，根据整个上下文去确定当前的值。

4、defer后面的语句在执行的时候，函数调用的参数会被保存起来，也就是复制一份。

在真正执行的时候，实际上用到的是复制的变量，也就是说，如果这个变量是一个"值类型"，那他就和定义的时候是一致的，如果是一个"引用"，那么就可能和定义的时候的值不一致

defer配合recover

recover(异常捕获)可以让程序在引发panic的时候不会崩溃退出。

在引发panic的时候，panic会停掉当前正在执行的程序，但是，在这之前，它会有序的执行完当前goroutine的defer列表中的语句。

所以我们通常在defer里面挂一个recover，防止程序直接挂掉，类似于try...catch，但绝对不能像try...catch这样使用，因为panic的作用不是为了抓异常。recover函数只在defer的上下文中才有效，如果直接调用recover，会返回nil

interface常见问题：

接口就是一种约定

接口分为侵入式和非侵入式，类必须明确表示自己实现了某个接口

侵入式和非侵入式的区别

1、侵入式：

你的代码里已经嵌入了别的代码，这些代码可能是你引入过的框架，也可能是你通过接口继承得来的，比如：java中的继承，必须显示的表明我要继承那个接口，这样你就可以拥有侵入代码的一些功能。所以我们就称这段代码是侵入式代码。

优点：

通过侵入代码与你的代码结合可以更好的利用侵入代码提供给的功能。

缺点：

框架外代码就不能使用了，不利于代码复用。依赖太多重构代码太痛苦了。

2、非侵入式（没有依赖，自主研发）：

正好与侵入式相反，你的代码没有引入别的包或框架，完完全全是自主开发。比如go中的接口，不需要显示的继承接口，只需要实现接口的所有方法就叫实现了该接口，即便该接口删掉了，也不会影响我，所有go语言的接口数非侵入式接口；再如Python所崇尚的鸭子类型。

优点：

代码可复用，方便移植。非侵入式也体现了代码的设计原则：高内聚，低耦合

缺点：

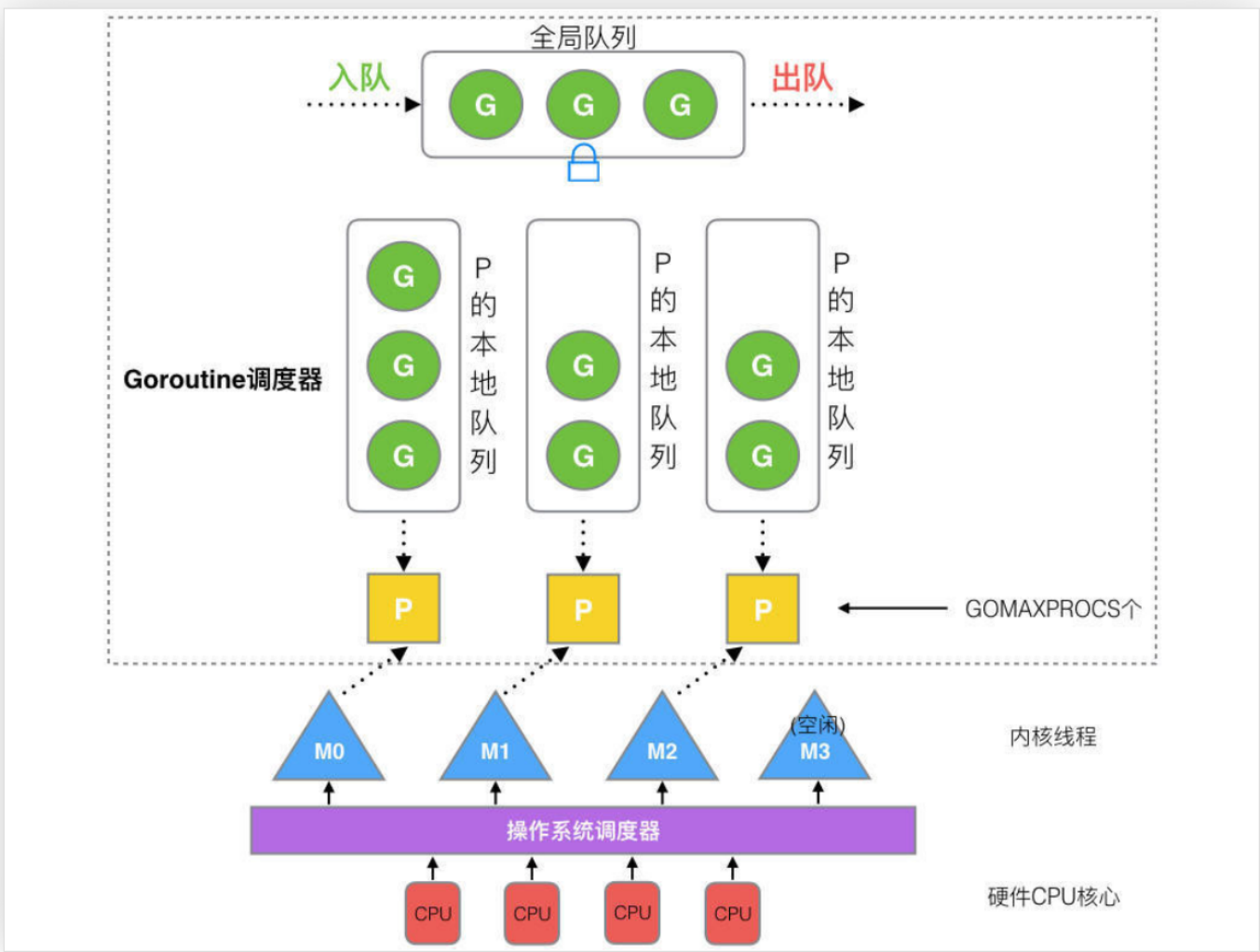
无法复用框架提供的代码和功能

简述GMP模型和它的优点

GMP模型中：

- G表示goroutine
- M表示Thread

- P表示Processor：Processor包含了运行goroutine所需要的资源，如果线程想要运行goroutine则必须要先获取processor，P中还包含了可运行的G队列，其中线程是执行goroutine的实体，processor的功能是把goroutine调度到工作线程上去执行



GMP模型中有如下几种结构：

- 全局队列：存放等待运行的G
- P的本地队列：每个P所拥有的队列，也是存放等待运行的G，有最大个数的限制，新创建一个G优先加入到P的本地队列中，如果本地队列已满就放入全局队列中顺便将本地队列中一半的P放到全局队列
- P：所有的P都在程序启动时即创建，并保存在一个数组中，最多有GOMAXPROCS个
- M：线程想要运行G就得获取P，优先从P的本地队列中获取，如果P的本地队列为空，就会尝试从全局队列中拿一批G放入本地队列，或者从其他的P中偷一半放到自己P的本地队列中

GMP模型的优点

- 复用线程：避免频繁的创建销毁线程
  - work stealing策略：当本线程绑定的P本地队列中无可运行的G时，会尝试从其他P那里偷G来运行，而不是销毁本线程
  - hand off机制：当本线程因为执行某个G发生系统调用而阻塞时，会将绑定的P释放，将P转移给其他空闲的M去执行
- 多核并行：GOMAXPROCS设置P的数量，因此最多可有这么多个线程分布在多个cpu上同时执行
- 抢占：在其他协程中要等待一个协程主动让出cpu才会让下一个协程执行，而go中一个goroutine最多占用cpu10ms，防止其他goroutine被饿死
- 全局队列：当work stealing策略失效时，会从全局队列中获取G来执行

goroutine与线程的区别

1、使用方面：

（1）goroutine比线程更加轻量级，可以轻松创建十万、百万，不用担心资源问题

（2）goroutine与channel搭配使用，能够更加方便的实现高并发

2、实现方面：

（1）从资源上讲

1. 线程栈的内存大小一般固定为2MB
2. goroutine栈内存是可变的，初始的时候一般为2KB，最大可以扩大到1GB

（2）从调度上讲

1. 线程的调度由OS的内核完成
2. goroutine调度由自身的调度器完成

goroutine与线程的联系：

（1）多个goroutine绑定在同一个线程上面，按照一定的调度算法执行

goroutine调度机制

三个基本概念：MPG

1、M

代表一个线程，所有的G(goroutine)任务最终都会在M上执行

2、P（Processor）

1. 代表一个处理器，每个运行的M都必须绑定一个P。P的个数是GOMAXPOCS，最大为256，在程序启动时固定，一般不去修改。
2. GOMAXPOCS默认值是当前电脑的核心数，单核CPU就只能设置为1，如果设置>1，在GOMAXPOCS函数中也会被修改为1。
3. M和P的个数不一定一样多，M>=P，每一个P都会保存本地的G任务队列，另外还有一个全局的G任务队列。G任务队列可以认为线程池中的线程队列。

3、G（Goroutine）

1. 代表一个goroutine对象，每次go调用的时候都会创建一个G对象

goroutine调度流程

带了张图，便于理解

1、启动一个goroutine

也就是创建一个G对象，然后加入到本地队列或者全局队列中

2、查找是否有空闲的P

如果没有就直接返回
如果有，就用系统API创建一个M（线程）

3、由这个刚创建的M循环执行能找到的G任务

4、G任务执行的循序

- 先从本地队列找，本地没有找到
- 就从全局队列找，如果还没有找到
- 就去其他P中找

5、所有的G任务的执行是按照go的调用顺序执行的

6、如果一个系统调用或者G任务执行的时间太长，就会一直占用这个线程

（1）在启动的时候，会专门创建一个线程sysmon，用来监控和管理，在内部挨个循环

（2）sysmon主要执行任务（中断G任务）

1. 记录所有P的G任务并用schedtick变量计数，该变量在每执行一个G任务之后递增
2. 如果schedtick一直没有递增，说明这个P一直在执行同一个任务
3. 如果持续超过10ms，就在这个G任务的栈信息加一个标记
4. G任务在执行的时候，会检查这个标记，然后中断自己，把自己添加到队列的末尾，执行下一个G

（3）G任务的恢复

1. 中断的时候将寄存器中栈的信息保存到自己G对象里面
2. 当两次轮到自己执行的时候，将自己保存的栈信息复制到寄存器里面，这样就可以接着上一次运行

goroutine是按照抢占式进行调度的，一个goroutine最多执行10ms就会换下一个

goroutine在什么情况下会被挂起呢?

goroutine被挂起也就是调度器重新发起调度更换P来执行时

- 在channel堵塞的时候;
- 在垃圾回收的时候;
- sleep休眠;
- 锁等待;
- 抢占;
- IO阻塞;

主goroutine与其他goroutine有什么不同

类似于一个进程总会有一个主线程，每一个独立的go程序在运行时也总会有一个主goroutine，主goroutine是自动启用而不需要手动操作的，每条go语句（启用一个goroutine的语句）一般都会携带一个函数调用，这个调用的函数被称为go函数，而主goroutine的go函数就是作为程序入口的main函数

从main函数的角度来理解主goroutine，则主goroutine就是程序运行的主程序，其他goroutine执行的程序是被异步调用的，同时主goroutine的结束也就意味着整个进程的结束。

扩展

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          for i := 0; i < 10; i++ {
7                  go func() {
8                          fmt.Println(i)
9                  }()
10         }
11 }
```

问：这段代码执行后会输出什么?

这段代码在主goroutine中进行了十次go语句调用，也就是启用了十个goroutine（因为执行一条go语句时go的runtime总是会优先从存放有空闲G的队列中获取一个G，没有空闲的情况下才会创建新的G，所以叫启用goroutine更合适）来打印出i的值。

需要注意的是go函数真正被执行的时间一定总是滞后于所属的go语句被执行的时间的（因为GMP调度器需要按先进先出的顺序来执行G），而go语句本身执行完毕后如果不加干涉则不会等待其go函数的执行，会立刻去执行后面的语句，所以for循环会很快的执行完，此时的那些go函数很可能还没有执行，此时i的值已经为10

另外当for语句执行完毕后主goroutine便结束了，则那些还未被执行的go函数将不会继续执行，即不会输出内容

所以对于以上代码，绝大多数情况下不会有任何输出，也可能会输出乱序的0到9或是10个10

GC（垃圾回收）原理 1.5版本

三色标记法

1、概念

（1）白色：代表最终需要清理的对象内存块
（2）灰色：待处理的内存块
（3）黑色：活跃的内存块

2、流程

（1）起初将所有对象都置为白色

（2）扫描出所有的可达（可以搜寻到的）对象，也就是还在使用的，不需要清理的对象，标记为灰色，放入待处理队列

（3）从队列中提取灰色对象，将其引用对象标记为灰色放入队列，将自身标记为黑色

（4）有专有的锁监视对象内存修改

（5）在完成全部的扫描和标记工作之后，剩余的只有黑色和白色，分别代表活跃对象与回收对象

（6）清理所有的白色对象

简述Go的垃圾回收机制

go目前使用的垃圾回收机制是三色标记法配合写屏障和辅助GC

三色标记法是对标记回收算法的改进:

1. **初始阶段**所有对象都是**白色**
2. 从root根出发**扫描根对象**,将它们引用到的对象都标记为**灰色**,其中**root区域主要是当前程序运行到的栈和全局数据区域,是实时使用到的内存**
3. 将**灰色对象标记为黑色**,分析该灰色对象是否引用了其他对象,如果有引用其他对象,就将引用的对象标记为灰色
4. **不断分析灰色对象,直到灰色对象队列为空,此时白色对象即为垃圾,进行回收**

在内存管理中,allocBits记录了每块内存的分配情况,而gcmarkBits记录了每块内存的回收标记情况,在标记阶段会对每块内存进行标记,有对象引用的标记为1,没有的标记为0,结束标记后,将allocBits指向gcmarkBits,则有标记的才是存活的内存块,这样就完成了内存回收

进行垃圾回收需要进行STW,如果STW时间过长对于应用执行来说是灾难性的,因此为了缩短STW的时间,golang引入了**写屏障和辅助GC**

写屏障是让GC和应用程序并发执行的手段,可以有效减少STW的时间

辅助GC是为了防止GC过程中内存分配的速度过快,因此会在GC过程中让mutator线程并发执行,协助GC执行一部分回收工作

GC触发机制有:

1. **内存分配量达到阈值**:每次内存分配前都会检查当前内存分配量是否达到阈值,如果达到则触发GC, 阈值=上次GC时的内存分配量 * 内存增长率
2. **定时触发GC**:默认情况下两分钟触发一次GC,可由runtime中的参数声明
3. **手动触发GC**:可以在代码中通过使用runtime.GC()来手动触发

select实现机制

1、锁定scase中所有channel

2、按照随机顺序检测scase中的channel是否ready

（1）如果case可读，读取channel中的数据
（2）如果case可写，写入channel
（3）如果都没准备好，就直接返回

3、所有case都没有准备好，而且没有default

（1）将当前的goroutine加入到所有channel的等待队列
（2）将当前协程转入阻塞，等待被唤醒

4、唤醒之后，返回channel对应的case index

5、select总结

（1）select语句中除了default之外，每个case操作一个channel，要么读要么写

（2）除default之外，各个case执行顺序是随机的

（3）如果select中没有default，会阻塞等待任意case

（4）读操作要判断成功读取，关闭的channel也可以读取

协程优势及其通信方式

协程相当于是用户态的线程

进程切换消耗资源很大，且进程间通信较复杂，于是有了线程

每个线程都有自己的堆栈和寄存器，并共享所属进程内的其他资源，因此可以方便地实现线程切换和通信，但是也由于多个线程共享地址空间，任何一个线程出错时，同进程内的所有进程都会崩溃

但是线程也难以实现高并发，因为：

1. **线程消耗的内存还是很多**，在linux系统中高达8MB，同时为了解决线程申请堆内存时互相竞争的问题，每个线程预先在这个空间内预分配了64MB作为堆内存池，因此没有足够的内存去开启十几万的线程实现并发
2. **线程切换耗时**：线程的切换是由内核控制的，因此当线程繁多的时候，线程间的切换会消耗cpu很多的计算能力

而协程使本来由内核实现的切换工作，交给了用户态的代码完成

通常创建协程时，会从进程的堆上分配一段内存作为协程的栈，线程的栈有8M，而协程的栈只有几十K

每个协程都有独立的栈，在go语言中，运行时系统会帮助自动创建和销毁系统线程，而对应的用户级协程是架设在线程之上的，用户级协程的创建销毁和调度都完全由程序实现和处理，而不用经过操作系统去做，速度会很快，很容易控制和灵活

因此总结下来，协程的优势有：

1. **节省cpu,避免系统内核级的线程频繁切换造成的cpu资源浪费**

2. 节约内存
3. 稳定性
4. 开发效率,协程是合作式的,可以方便地将一些操作异步化

Go中协程的通信方式

可以通过共享内存(变量)加锁的方式来进行通信,但是维护成本较高

官方推荐**通过channel进行通信**:

channel的主要功能是:

1. 作为队列存储数据
2. 阻塞和唤醒**goroutine**

select:

select搭配channel使用,其机制是监听多个channel,每一个case都是一个事件,一旦某个事件就绪(chan没有堵塞),就会从这些就绪事件中随机选择一个去执行,default用于所有chan都堵塞的情况执行

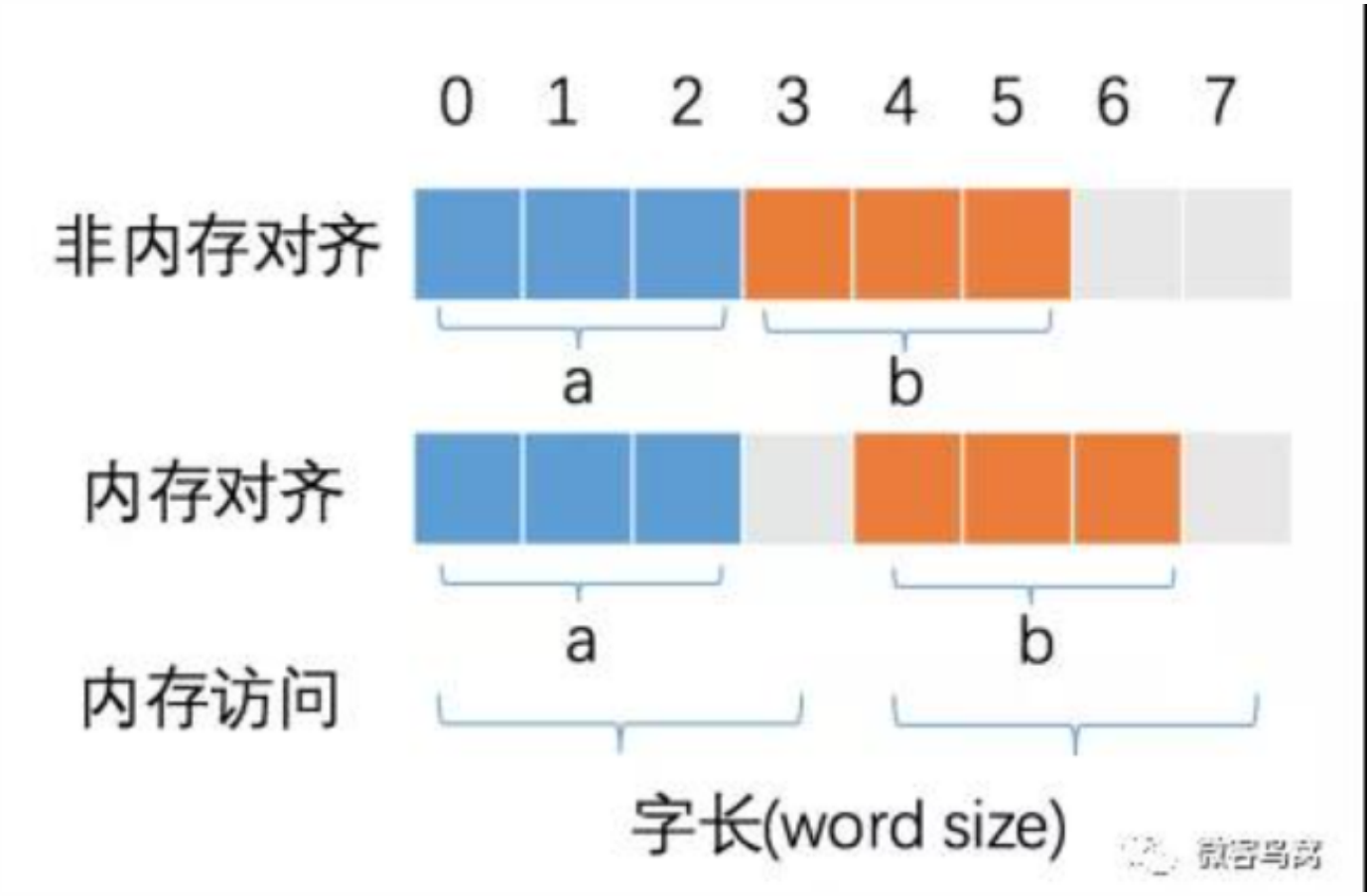
使用channel进行控制子goroutine的机制可以总结为:

循环监听一个channel,在循环中可以放select来监听channel达到通知子goroutine的效果,再配合Waitgroup,主进程可以等待所有协程结束后再自己退出;这样就通过channel实现了优雅控制goroutine并发的开始和结束

Go的内存对齐

CPU 访问内存时，并不是逐个字节访问，而是以字长（word size）为单位访问。比如 32 位的 CPU ，字长为 4 字节，那么 CPU 访问内存的单位也是 4 字节。

CPU 始终以字长访问内存，如果不进行内存对齐，很可能增加 CPU 访问内存的次数，例如：



变量 a、b 各占据 3 字节的空间，内存对齐后，a、b 占据 4 字节空间，CPU 读取 b 变量的值只需要进行一次内存访问。如果不进行内存对齐，CPU 读取 b 变量的值需要进行 2 次内存访问。第一次访问得到 b 变量的第 1 个字节，第二次访问得到 b 变量的后两个字节。

也可以看到，内存对齐对实现变量的原子性操作也是有好处的，每次内存访问是原子的，如果变量的大小不超过字长，那么内存对齐后，对该变量的访问就是原子的，这个特性在并发场景下至关重要。

简言之：合理的内存对齐可以提高内存读写的性能，并且便于实现变量操作的原子性。