golang

基础（函数/语法/细节）

go语言特色

go的优势

- **语法简单**：语法简单，代码易于阅读和理解，学习成本很低。
- **高性能-协程**：实现简单，对比进程和线程，协程占用资源少，使用起来也简单，能够轻松地处理高并发问题。
- **内存管理友好**：Go语言的自动垃圾回收机制减少了内存泄漏的风险，使得开发者无需手动管理内存(c/c++需手动管理)。这降低了开发难度，并提高了代码的可靠性。

go有哪些很不错的地方（举例/具体）

函数可以有多个返回值，返回多个结果避免了使用复杂的数据结构封装结果。

defer使用起来很方便，比如文件的关闭，网络连接的关闭，或者结合recover进行错误处理

并发编程很方便，通过goroutines和channels等机制，开发者可以轻松地编写高效的并发程序。

GC自动内存管理，不需要关注内存的申请和释放

go的缺点

右大括号不允许换行，否则编译报错。不允许有未使用的包或变量

错误处理繁琐，需要显示的返回错误，根据错误是否为空判断是否执行成功，代码很冗余。

没有构造函数，对象创建比较麻烦

golang数据类型

基本数据类型：int/byte/rune/float64/complex6，string，boolean

结构体，数组

切片slice，map，channel，函数function，指针pointer，Interface

简述一下rune类型

`rune`是`int32`的别名，go语言中string类型底层是byte字节数组，每个字符占1个字节，但是对于中文字符占多个字节，所以byte无法表示中文，而rune占用4个字节，可以处理中文。

uint 类型溢出问题

var a uint8 =255
var b uint8 =1
a+b = 0总之类型溢出会出现难以意料的事

各数据类型的比较操作

除了基本数据类型外的其他类型最多只能使用==和!=进行比较

基本数据类型

整数、浮点数、复数支持==和!=比较，除了复数外，其余的数值类型还可以使用 < 、> 、<= 和 >= 进行大小比较。

字符串是基于字典序进行比较的，因此也可以使用 < 、> 、<= 和 >= 来比较大小。

布尔值可以使用 == 和 != 进行比较

数组

数组长度相同的时候，这两个数组才是可比较==和!=的。数组间的比较是逐个元素进行的，一旦遇到不相等的元素则停止比较并返回false

结构体

如果结构体的**所有字段都是可比较的**，则该结构体类型也是可比较的

注意不同结构体是不同的类型，即使字段一样，也不可以比较。

指针

可以比较，比较的是存储的内存地址是否相同，即两个指针是否指向同一个变量

interface

接口的动态值为可比较类型并且具体类型一致时，才可进行比较

切片/映射/通道/函数

切片（slice）、映射（map）、函数（func）、通道（chan），不可比较，除了与 nil 进行比较之外

深拷贝和浅拷贝/值传递与引用传递

值传递/深拷贝：传递的是数据的值，两个变量不共享内存，各占了一块内存空间。

引用传递/浅拷贝：传递的是数据地址，新老对象指向同一个内存，比如传递指针、切片这些引用数据类型的传递。

字符串

“”和``

- 双引号 "" 通常用于定义简单的、不需要包含换行符或特殊转义字符的字符串。
- 反引号``则常用于定义**包含换行符、特殊字符**或需要避免转义字符的复杂字符串，如正则表达式、HTML模板、SQL查询等。

如何高效拼接字符串

使用 `strings.Builder`，可以避免内存分配和复制，提高性能。

```
func main() {
    var builder strings.Builder

    for i := 0; i < 10; i++ {
        builder.WriteString("Go ")
    }

    result := builder.String() // Get the concatenated string
    fmt.Println(result)
}
```

字符串转成byte切片，会发生内存拷贝吗?

会发生内存拷贝。这是因为在Go中，字符串是不可变的，而字节切片是可变的。为了防止通过修改字节切片而间接修改字符串，Go在将字符串转换为字节切片时，会创建一个新的字节数组来存储数据。

```
s := "afsdf"
b := []byte(s)
```

make和new的区别

两个内置函数都用来为变量分配内存

- 类型：make 只能用来分配及初始化类型为slice、map、chan 的数据。new 可以分配任意类型的数据；
- 返回值：new 返回为对应类型的指针。make返回的是引用类型本身。
- 初始化：new 分配内存的时候，将内存直接置为零值。make分配空间后，会初始化引用数据类型，比如给切片底层数组设置长度和容量，初始化映射的哈希表结构等；

结构体

空结构体

使用unsafe.SizeOf()知道空结构体不占任何空间，所有空结构体都是同一个地址

使用场景：

- **实现集合类型set**，`type set map[string]struct{}` 节省空间
- **空通道chan struct{}**，用于协调Goroutine的运行
- 作为方法接收者

只用占位不用实际含义，那么我们就都可以使用空结构体，可以极大的节省不必要的内存开销。

interface

空接口类型，go中所有类型都实现了这个接口，因此interface{}可以用于存储任何类型的值

底层实现

底层有两个指针，一个指向具体类型的信息，一个指向接口类型原始数据

interface可以比较吗?

- 如果两个`interface`变量都是`nil`，则它们相等。
- 如果只有一个`interface`变量是`nil`，则它们不相等。
- 如果两个`interface`变量都非`nil`，则Go会比较它们的类型信息和指向的数据。如果底层类型信息和指向的数据都相等，则interface相等，否则不相等。

nil interface和空interface

nil interface没有初始化，类型和值都为nil

空interface，比如将一个空结构体变量赋给了接口变量，那么接口变量的类型不为空，但值为nil

类型断言

也就是向下转型，**只有接口类型**的变量可以使用类型断言。类型断言是一种机制，允许你**从接口类型中提取其底层具体类型的值**。目的就是访问这个变量所持有的具体类型的值时，你就会使用类型断言。

defer

defer概念

defer 执行的时候会把它后面的函数压入一个栈里面。当外部函数准备返回的时候，go就会按照先进先出的顺序调用这个栈中的函数。

应用场景

比如文件的关闭，网络连接的关闭，锁的释放，或者结合recover进行错误处理

defer特性

defer是闭包的一种，defer函数同样捕获变量而不是值。

后进先出（LIFO）：当在一个函数中存在多个 `defer` 语句时，它们将会以后进先出的顺序执行。也就是说，最后一个 `defer` 语句最先被执行，第一个 `defer` 语句最后被执行。

参数在 defer 语句中立即求值：在 `defer` 语句中，函数的参数变量会立即被计算并保存，即时外部修改了这个变量也不会对defer函数有影响。

defer与return：对于有返回值的函数，先给返回值赋值，在执行defer，最后再返回

介绍一下panic和recover

panic：会抛出一个异常，然后**中断当前函数的执行，沿着函数的调用栈向上搜寻，执行所有的defer函数。**

recover：**可以捕获panic的错误并恢复当前程序的执行**，只能在defer的函数中使用

```
func throwErr() {
    defer func() {
        if err := recover(); err != nil {
            fmt.Println(err)
        }
    }()
    panic("111")
}

func main() {
    throwErr()
    fmt.Println("recover后，可以继续执行")
}
```

select原理

select使用起来类似switch语句，每个case语句绑定一个channel的读写操作，select会挑选一个没有阻塞的操作执行。这样一个goroutine就可以监听多个channel操作，只有所有的case都阻塞了，这个goroutine才会阻塞。

init函数

在包导入的时候由runtime执行，用于初始化包，如果一个包被导入多次，init()函数只会执行一次

闭包

介绍

闭包引用外部变量的问题

闭包引用外部变量的时候，引用的是地址，外部变量的改变，同样会影响闭包中的行为

```
func Test1(t *testing.T) {
    var a = 1
    f := func() {
        fmt.Println(a)
    }
    a++
    f() //输出2
}
```

应用场景和缺点

闭包捕获的是变量，而非值（循环中需谨慎）

避免闭包长期持有大对象，防止内存泄漏。

错误处理

errors包

CSP并发模型

CSP（Communicating Sequential Processes），通信顺序进程，强调通过通信来共享内存，而不是通过共享内存来通信。go语言是通过channel实现的。

- **避免了复杂的锁机制和共享内存问题。**
- 使用channel可以清晰地表达goroutine之间的通信流程，**使程序更加易于理解和维护**

goroutine内存泄露有哪些情况?

gorutine泄露，Goroutine没有正确管理和销毁，导致这些Goroutine无法被垃圾回收器回收。

channel未关闭，导致接收该channel的goroutine阻塞。

死锁，当多个Goroutine相互等待对方的资源，形成死锁状态时，这些Goroutine将无法继续执行，也无法被垃圾回收器回收。

内存对齐

使用空间换时间，提高读取内存数据时的效率，数据结构应该尽可能地在自然边界上对齐，如果访问未对齐的内存，处理器可能需要做两次内存访问，而对齐的内存访问仅需要一次访问

切片

底层实现

切片它不是动态数组或者数组指针，它底层就是一个slice结构体，通过一个指针指向了底层数组，如果数组满了就会通过扩容机制创建新的且更大的底层数组，从而实现切片长度的动态增长。

```
type slice struct {
    array unsafe.Pointer    //指向底层数组的指针
    len int                 //切片长度
    cap int                 //切片容量
}
```

- 指针：指向 slice 可以访问到的第一个元素。
- 长度：slice 中元素个数。
- 容量（底层数组长度）： slice 起始元素到**底层数组**最后一个元素间的元素个数。

数组和切片的区别

- 数组是值类型，而切片是引用类型，其底层数据结构包含一个数组，切片也可以被看作对数组某一连续片段的引用
- 切片的容量会动态变化，而数组没有容量的概念，只有一个固定的长度。

切片扩容机制

在执行append操作追加若干元素的时候，如果切片没有足够的容量来容纳这些元素，那么就会扩容，首先要根据当前容量和预期想要的容量来确定一个新的容量，如果预期容量大于当前容量的两倍，那么新的容量就是预期容量，否则的话，如果当前容量小于阈值，当前容量翻倍增长，如果当前容量大于阈值，那么当前容量大概按一个1/4的比率循环增长，直到大于预期容量。

确定了容量后，底层创建一个这个容量大小的一个数组，然后把旧的数据复制过来，切片底层指针再指向这个数组，扩容完成。旧的数组会被垃圾回收掉。

具体流程为

- 根据**扩容原理**确定新切片的容量，
- 创建一个新的切片
- 将旧切片的元素和追加的元素复制到新的切片中，
- 切片的引用会指向新的底层数组，原数组会被垃圾回收。

Go 1.18版本之前容量确定

1. 如果期望容量大于当前容量的两倍就会使用期望容量；

2. 否则如果当前切片的长度小于 1024 就会将容量翻倍；

3. 如果当前切片的长度大于等于 1024 就会循环增加 25% 的容量，直到新容量大于期望容量；

Go 1.18版本之后容量确定

1. 如果期望容量大于当前容量的两倍就会使用期望容量；

2. 如果当前切片的长度小于阈值（默认 256）就会将容量翻倍；

3. 如果当前切片的长度大于等于阈值（默认 256），就会循环增加 25% 的容量，基准是 newcap + 3*256，直到新容量大于期望容量；

Go的设计者不断优化切片扩容的机制，其目的只有一个：就是控制让小的切片容量增长速度快一点，减少内存分配次数，而让大切片容量增长率小一点，更好地节省内存。

切片有什么坑？原理是什么

多个切片指向同一个地址，当某一个切片执行append操作发生了扩容，那么它指向的地址会改变

切片内存泄漏场景

为什么切片没有缩容机制

map

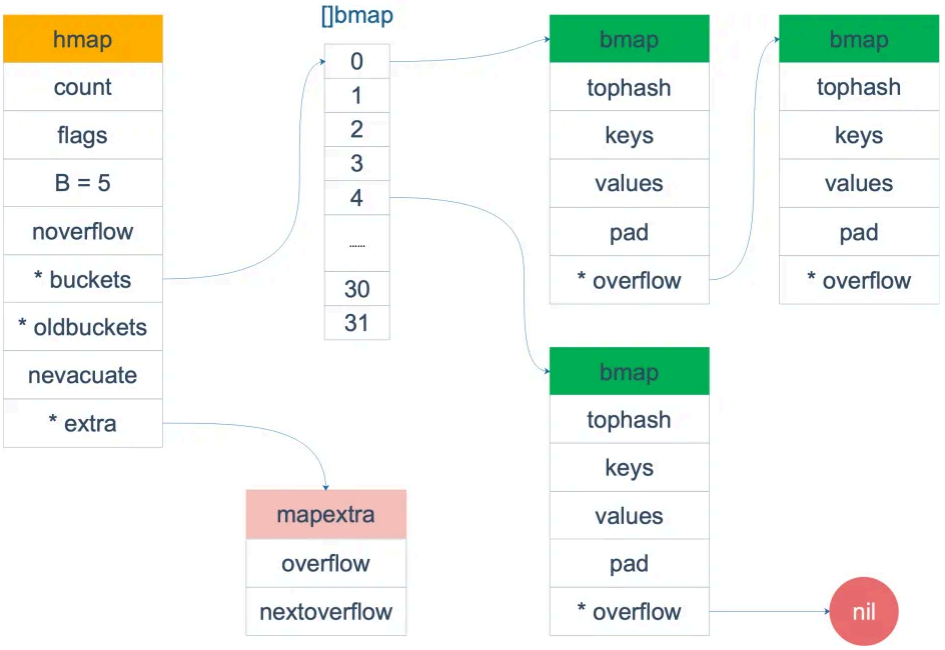
底层实现

概括

map的底层是一个hmap结构体。里面包含了一个**桶数组**来存储键值对，其实就是一个哈希表。每个桶都是一个链表，其实就是通过拉链法来解决哈希冲突，每个桶底层都是bmap结构体，可以存8个键值对。读写数据的时候，首先通过哈希函数计算出哈希值，对桶的大小取模（哈希值低B位）找到对应的桶，在这个桶和溢出桶链表中（通过tophash即哈希值高8位）找到key的位置，读写元素。

```
// go 1.17
type hmap struct {
    count      int            //元素个数，调用len(map)时直接返回
    flags      uint8          //标志是否正在写map，并发操作会返回fatalerror
    B          uint8          //桶(buckets)的对数 B=5表示能容纳32个元素
    noverflow  uint16        //桶(buckets)溢出数量，如果一个单元能存8个key，此时存储了
9个，溢出了，就需要再增加一个桶
    hash0      uint32         //哈希种子
    buckets    unsafe.Pointer //指向桶(buckets)数组,大小为2^B，可以为nil
    oldbuckets unsafe.Pointer //扩容的时候，buckets长度会是oldbuckets的两倍
    nevacute   uintptr        //指示扩容进度，小于此buckets迁移完成
    extra      *mapextra      //与gc相关 可选字段
}
```

```
// A bucket for a Go map.
type bmap struct {
    tophash [bucketCnt]uint8
}
//实际上编译期间会生成一个新的数据结构
type bmap struct {
    topbits  [8]uint8     //哈希值高8位
    keys     [8]keytype   //key
    values   [8]valuetype //value
    pad      uintptr
    overflow uintptr      //下一个桶的地址
}
```

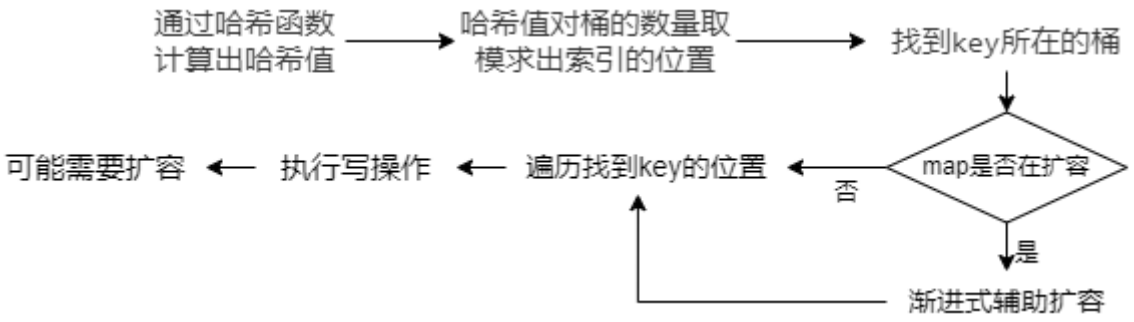


读流程

通过哈希函数计算出哈希值->哈希值对桶的数量取模求出索引的位置->遍历桶和溢出桶链表和其中的8个槽位找到key所在的位置->没找到返回零值，找到返回val

桶中包含8个键值对和8个tophash（hash值高8位）用于读的时候判断是不是这个位置。

写流程



扩容机制

map有两种扩容机制，翻倍扩容与等量扩容

- **翻倍扩容**：在**装载因子**也就是map元素与桶数量之比超过6.5时，表明空闲位置少，冲突概率高，性能低，从而触发翻倍扩容分配更大的内存空间。创建一个长度为原来的两倍的桶数组，从而减少哈希冲突，旧bucket数据搬迁到新的bucket，然后底层指针指向新的桶数组。
- **等量扩容**：在溢出桶的数量超过了普通桶数量时触发，只迁移数据而不扩大容量，把松散的键值对重新排列一次，使数据更加的紧凑，**进而保证更快的存取，也提高了空间利用率**

渐进式扩容：扩容的时候，对map进行操作时才进行部分数据的迁移，避免一次性全部数据迁移引发性能抖动

key可以是哪些类型？可以嵌套map吗？

map的key**可以是任何可以比较的类型**。这包括所有的基本类型，如**整数、浮点数、字符串和布尔值，以及结构体和数组**

key不能是：map，切片，函数

可以嵌套map

map是并发安全的吗？为什么？

并发安全：多个线程进行并发读写时，程序可以正常运行并得到预期结果

Map不是并发安全的：因为map没有加锁访问，多个线程可以同时访问桶数组键值对，并发读写时程序会panic。

更深入：因为写操作不是原子操作，分为了多个步骤，从内存读取，赋值给临时变量，写入内存。如果多个线程同时读取，就会出现覆盖的情况。

线程安全的map实现：

- 使用sync包加互斥锁或读写锁
- 使用官方的线程安全的map，sync.Map

map的遍历为什么是无序的？

map可以通过`for range map`进行遍历，但是每一次遍历的顺序是不一致的，原因如下：

- 每次遍历不是从0号桶开始遍历，遍历时底层会生成一个随机数来决定从哪里开始遍历
- map在扩容后，会发生key的搬迁，这造成原来落在一个bucket中的key，搬迁后，有可能会落到其他bucket中了

sync.Map的原理以及与普通map的性能比较

使用了**读写分离**的方法，冗余了两个map分别读写数据，当要读取数据时，先从read无锁读取，没获取到或者数据已经过时，就加锁读取dirty map，并且未命中次数加一，如果未命中次数到达了dirty的长度，就把dirty中的数据复制到read中，相当与回写缓存。写操作就直接加锁访问dirty。

每次写操作时先将read中的数据标记为过时，然后加锁更新dirty map

- sync.map 分离读写操作，减少了锁的使用，相比普通map**大部分读操作都是无锁的**，提高了性能。但只适合读多写少的场景
- 普通map读写都要加锁，性能比不上sync.map，但相比于sync.map维护两个map**内存开销更大**

如果在单线程或低并发场景下，普通 `map`（带手动加锁）通常比 `sync.Map` 性能更好。
在高并发场景下， `sync.Map` 性能更好，优化了读多写少的并发性能。

```
type Map struct {
    //该锁用来保护dirty
    mu Mutex
    //只读的map，不需要加锁
    read atomic.Value // readOnly
    //可写的map，用于存储所有通过Store方法添加的键值对。在read map中没有找到对应的键时，会尝试从dirty map中查找。
    dirty map[interface{}]*entry
    //记录从read map中未命中（即未找到对应键）的次数，当等于len（dirty）的时候，会将dirty拷贝到read中（从而提升读的性能）。
    misses int
}
```

sync.Map使用**空间换时间、读写分离**的方法冗余了两个map进行操作。**对read map的操作可以不用加锁**，执行读操作时会优先操作read map，数据不在read中才加锁访问dirty map。相比直接加锁，提高了性能，但只适用于读多写少的场景，写操作较多，频繁的加锁访问。

还采用了**动态调整**（数据同步）的机制，当read map未命中的次数达到dirty map的长度的时候，会将dirty中的同步到read map中，从而提高read map的命中率

删除操作采用内了**延迟删除**的策略，删除并不是真正意义上的删除，只是将被删除的数据进行了标记，直到dirty和read进行数据同步的时候，才真正删除这些标记的数据，标记这些数据可以迅速完成，提高了性能。

sync.Map与map的区别

sync.Map无需初始化，直接声明即可使用

sync.Map无需指定存储的数据类型，同一个map可以存多种类型

必须使用Store、Load、Delete方法进行操作

map注意事项

零值

如果你试图获取一个 `map` 中不存在的键的值，你将得到该类型的零值，而不是一个错误。

nil map读写

nil map可读不可写

`nil` map 是一个未初始化的 map，其值为 `nil`。你不能向 `nil` map 添加任何元素，否则会引发运行时错误（panic）。但是，你可以从 `nil` map 中获取元素，这不会引发运行时错误，但总是返回元素类型的零值。

迭代顺序

当你迭代一个 `map` 时（如使用 `for` 循环），键的顺序是不确定的。每次迭代的顺序可能都不同。如果你需要稳定的迭代顺序，你可能需要将键排序后再迭代。

并发安全

channel

介绍一下channel

- 通道是golang中实现goroutine间通信的一个数据类型，多个goroutine向其中发送和读取数据，不会发生资源竞争问题。
- channel本质上是一个队列，一个加锁的循环数组，遵循先入先出规则。
- 通道可以被定义为各种数据类型的，它可以有缓冲，也可以无缓冲；可以声明只读只写通道。

底层实现

概括

channel底层是一个hchan的结构体，通过一个环形数组来存储数据，有两个变量保存当前环形数组正在发送和接收数据的位置。还有两个读写等待队列存读写操作被阻塞的协程。通过加互斥锁解决并发安全问题。

当向channel中发送数据的时候，首先判断读等待队列是否为空，如果不为空就说明当前有被阻塞的读操作且缓冲区为空，就从中唤醒一个读协程，并直接将数据发送过去。如果读等待队列为空就去使用缓冲区，如果缓冲区没满就直接将数据放入其中，如果缓冲区已满，将要发送的数据和当前的协程封装好放入写等待队列，并阻塞当前协程。

向channel中读取数据时，也是类似的，先判断写等待队列是否为空，如果不为空就说明当前有写操作被阻塞并且缓冲区已满，从中唤醒一个goroutine并且接收它的数据。如果写等待队列为空，就只能读取缓存，缓存中没有数据同样要进行阻塞操作。

无缓存通道相比于有缓存通道就是没有初始化缓存区，只通过读写等待队列进行读写操作的数据传输，没有经过缓存区。

```
type hchan struct {
    closed uint32   //channel是否关闭，0未关闭
    elemtype *_type //channel元素类型

    buf unsafe.Pointer  //指向底层循环数组的指针
    qcount   uint       // 循环数组中的元素数量
```

```
    dataqsize uint        // 循环数组的长度
    elemsize uint16       // 每个元素的大小
    sendx    uint         // 下一次写下标的位置
    recvx    uint         // 下一次读下标的位置

    recvq    waitq  // 读等待队列
    sendq    waitq  // 写等待队列

    lock mutex //互斥锁，保证读写channel时不存在并发竞争问题
}
```

阻塞的协程队列

```
type waitq struct {
    first *sudog    //队列头部
    last  *sudog    //队列尾部
}
```

用于包装协程的节点

```
type sudog struct {
    g *g            //协程
    next *sudog     //队列中后一个节点
    prev *sudog     //队列中前一个节点
    elem unsafe.Pointer //读取/写入 channel 的数据的容器;
    isSelect bool //当前协程是否处在 select 多路复用的流程中
    c        *hchan //标识与当前 sudog 交互的 chan.
}
```

创建

在内存中实例化了一个 ***hchan*** 结构体，并返回一个chan指针

对于无缓冲通道不会初始化底层的环形数组

发送步骤

channel已初始化且未关闭

- 首先检查**读等待队列是否为空**，如果不为空就说明当前有被阻塞的读操作且缓冲区为空，就从中唤醒一个读协程，并将数据发送过去
- 如果读等待队列为空
  - **缓冲区没满**就直接将数据放入其中
  - 如果**缓冲区已满**就将要发送的数据和当前的协程封装好放入写等待队列，并阻塞当前协程

接收步骤

channel已初始化

- 首先检查**写等待队列是否为空**，如果不为空就说明当前有被阻塞的写操作且缓冲区已满，将数据读取出来后，从中唤醒一个写协程
- 如果写等待队列为空
  - 如果**缓冲区不为空**就直接读取
  - **缓冲区为空**就将当前协程放到读等待队列，并阻塞等待唤醒

非阻塞模式

在select语句组成的多路复用分支中，与 channel 的交互会变成非阻塞模式

非阻塞模式下协程不会被阻塞，要阻塞时直接返回false

channel和锁如何选择?

channel的开销更大，对于简单的操作使用锁更加高效

channel数据会流动，而锁要保护的数据是固定的。应根据具体场景进行选择

channel应用场景

- **goroutine通信**：发送一些通知，或者传递一些数据给其他goroutine
- **同步控制**：通过goroutine可以控制一些goroutine的执行顺序
- **超时处理**：配合select进行超时处理

通道操作结果表，相关特点

向空channel发送和读取数据会阻塞，**向已关闭的channel发送数据会panic**，向已关闭的channel读取数据会正常读取没有数据会读取到0值和false。**关闭已关闭的chan或为nil的chan会panic**

通道操作结果表				
状态 / 操作	nil	没值	有值	满
发送	阻塞	发送成功	发送成功	阻塞
接收	阻塞	阻塞	接收成功	接收成功
关闭	panic	关闭成功	关闭成功	关闭成功

向未初始化，为nil的管道读写会永久阻塞

nil管道关闭会panic

关闭的管道可读，但写的话会panic

关闭的管道再关闭会panic

值为nil的channel有什么用?

向其中接受和发送数据都会永久阻塞，可以用来在select语句中禁用一个case

有缓冲和无缓冲channel有什么区别?

创建方式不同，无缓冲通道make初始化的时候不指定容量或指定容量为0

- 无缓冲在有另一个协程读取数据前发送阻塞，在有另一个协程发送数据前接受阻塞，而有缓冲的情况下，发送数据只有缓冲区满时阻塞，接收数据只有缓冲区为空才阻塞。
- 无缓冲通道是发送数据和接收数据精确的同步，发送时阻塞，直到被接收。而有缓冲通道是异步的，发送时放到缓存就不阻塞

channel的优缺点

channel优势是降低了并发中的耦合，劣势是不当操作可能会永久阻塞

使用channel模拟信号量进行同步控制

```
var wg sync.WaitGroup

func print(s string, in, out chan struct{}) {
    <-in
    fmt.Println(s)
    out <- struct{}{}
    wg.Done()
}

func main() {
    ch1 := make(chan struct{}, 1)
    ch2 := make(chan struct{}, 1)
    ch3 := make(chan struct{}, 1)
    ch1 <- struct{}{}
    wg.Add(3)
    go print("111", ch1, ch2)
    go print("222", ch2, ch3)
    go print("333", ch3, ch1)
    wg.Wait()
}
```

Mutex

互斥锁实现原理

mutex通过自旋和排队阻塞相结合的实现的

正常模式下是公平竞争的，goroutine会通过自旋和CAS操作进行加锁，如果自旋4次都没有加锁成功那么就会进入等待队列阻塞。解锁时被唤醒的goroutine，不会立即拥有锁，而是会和当前正在自旋请求加锁的goroutine竞争锁，因为当前自旋中的goroutine占用了cpu时间片，所以刚唤醒的goroutine很有可能竞争不过，重新进入等待队列阻塞，从而可能导致饥饿问题。

当有goroutine等待时间超过1ms时，锁的模式就会切换饥饿模式来应对这种激烈的竞争。饥饿模式下新创建的goroutine不会自旋获取锁而是直接进入队列尾部排队等待。互斥锁会依次交给队列中的队头goroutine处理，这样就解决了饥饿等待问题。**如果等待队列已经空了或者一个goroutine的等待时间小于1ms就会切换回正常模式。**

总结：正常模式下通过自旋操作避免了goroutine频繁的阻塞和调度，有更好的性能，而饥饿模式解决因为锁竞争导致的饥饿问题。

```
type Mutex struct {
    // 状态码
    state int32
    // 信号量，用于向处于 Gwaitting 的 G 发送信号
    sema  uint32
}
```

正常模式

- 加锁时如果锁已被占用会自旋，如果自旋4次没有加锁成功就会进入等待队列被阻塞
- 解锁时，唤醒的goroutine不会直接拥有锁，而是会和当前请求加锁的goroutine竞争锁，被唤醒的goroutine竞争很有可能失败，在这种情况下，这个被唤醒的 goroutine 会重新加入到等待队列的前面。

饥饿模式

- 当一个goroutine等待锁时间超过1毫秒时，锁会切换到饥饿模式，
- 新进来的goroutine加锁时，不会参与抢锁也不会进入自旋状态，直接排队等待
- 解锁时直接将锁交给等待队列中的第一个协程
- 如果一个 Goroutine 获得了互斥锁并且它在队列的末尾或者它等待的时间少于 1ms，那么当前的互斥锁就会切换回正常模式。

允许自旋的条件

- 锁被占用且不处于饥饿模式，且自旋次数小于4
- CPU核数和P的数量大于1，否则自旋没有意义，因为此时不可能有其他协程释放锁
- 当前 Goroutine 所挂载的 P 下，等待运行队列为空，否则会先执行完其他任务再执行需要自旋的goroutine

golang实现自旋锁

```go
package main

import (
    "sync/atomic"
    "time"
)

type Spinlock struct {
    locked int32
}

func (s *Spinlock) Lock() {
    for !atomic.CompareAndSwapInt32(&s.locked, 0, 1) {
        // 这里可以添加一些退避策略，比如随机等待一段时间，以避免过多的CPU占用
        // time.Sleep(time.Nanosecond) // 注意：实际使用中可能不需要或想要这样的退避
    }
}

func (s *Spinlock) Unlock() {
    atomic.StoreInt32(&s.locked, 0)
}

func main() {
    var lock Spinlock

    // 示例：使用自旋锁
    go func() {
        lock.Lock()
        // 执行一些操作...
        lock.Unlock()
    }()

    // 在另一个goroutine中尝试获取锁
    go func() {
        lock.Lock()
        // 执行一些操作...
        lock.Unlock()
    }()

    // 等待足够的时间以确保goroutines完成
    time.Sleep(time.Second)
}
```

读写锁实现原理（小）

```
type RWMutex struct {
    w           Mutex  // 控制 writer 在 队列B 排队
    writerSem   uint32 // 写信号量，用于等待前面的 reader 完成读操作
    readerSem   uint32 // 读信号量，用于等待前面的 writer 完成写操作
    readerCount int32  // reader 的总数量，同时也指示是否有 writer 在队列A 中等待
    readerWait  int32  // 当前第一个阻塞的writer前面reader的数量
}

// 允许最大的 reader 数量
const rwmutexMaxReaders = 1 << 30
```

RWMtex是底层是基于mutex实现的，封装了一个互斥锁用于写协程和写协程互斥，readerCount变量表示当前正在读和被阻塞的所有读协程数量，readerWait表示当前第一个阻塞的writer前面reader的数量

RLock

读协程加锁，将readerCount加1，如果readerCount小于0表示当前有写协程正在写，需要阻塞

RUnLock

读协程解锁，将readerCount减1，如果readerCount小于0表示当前有阻塞的写协程，将readerWait减1，如果readerWait减为了0就将阻塞的写协程唤醒

Lock

- 写协程加锁，首先加互斥锁进行写写互斥，如果当前有写协程正在写就会被阻塞
- 令readerWait+=readerCount，readerCount减为一个很小的负数，如果当前有读协程正在读即readerWait!=0，就阻塞

UnLock

- 将readerCount变为原来的正数，将所有阻塞的读协程唤醒
- 释放互斥锁

锁的注意事项

- 读锁或写锁在 Lock() 之前使用 Unlock() 会导致 panic 异常
- 使用 Lock() 加锁后，再次 Lock() 会导致死锁（不支持重入），需Unlock()解锁后才能再加锁
- 锁定状态与 goroutine 没有关联，一个 goroutine 可以 RLock（Lock），另一个 goroutine 可以 RUnlock（Unlock）

可重入锁如何实现

可重入锁：当某个线程获取了某个锁，还可以再次获取该锁，而不会发生死锁

通过一个变量记录当前获取到锁的协程即可

除了加锁还有什么方法解决并发安全问题

channel是并发安全的，底层封装了互斥锁访问

原子操作

信号量

原子操作

由一个独立的CPU指令代表和完成，可以在**不形成临界区和创建互斥量**的情况下完成并发安全的值替换操作，避免了并发安全问题。golang的atomic包提供了简单的原子操作，增减、交换、载入、存储等。

- 增减Add
- 载入Load
- 比较并交换CompareAndSwap，实现乐观锁
- 交换Swap
- 存储Store

Goroutine与并发

进程、线程和协程

进程：是应用程序的启动实例，每个进程都有独立的内存空间，不同的进程通过进程间的通信方式来通信。

线程：从属于进程，每个进程至少包含一个线程，线程是 CPU 调度的基本单位，多个线程之间可以共享进程的资源并通过共享内存等线程间的通信方式来通信。

协程：是轻量级的线程，是**用户态“线程”**，不受操作系统的调度而是由用户自行创建和控制的，而且占用内存更小，因此协程的创建和销毁以及调度的性能更高

goroutine和线程的区别

- **管理方式**：goroutine是轻量级线程，由 Go runtime而不是操作系统调度
- **资源占用**：goroutine资源占用更小，栈内存为2kb，线程堆栈内存有几mb。
- **资源消耗**：goroutine上下文切换、创建和销毁开销更小

for循环多次执行goroutine会有什么问题

在协程中打印for的下标i或当前下标的元素,会随机打印载体中的元素(Go1.22解决了这个问题)

```
func main() {
    for i := 0; i < 5; i++ {
        go func() {
            fmt.Println(i)
        }()
    }//输出10 10 10 10 10
    time.Sleep(3*time.Second)
}
```

for循环很快就执行完了，但是创建的10个协程需要做初始化。上下文准备，堆栈，和内核态的线程映射关系的工作，是需要时间的，比for慢，等都准备好了的时候，会同时访问i。这个时候的i可能是for执行完成后的下标。

解决方法：给匿名函数增加入参，拷贝一份i传到这个协程里面去。

```
func main() {
    for i := 0; i < 5; i++ {
        go func(val int) {
            fmt.Println(val)
        }(i)
    }
    time.Sleep(3*time.Second)
}
```

什么是goroutine泄漏以及解决方法

类似于内存泄露，是指程序中创建的 Goroutine 没有正常退出，被无意义地保持存活，占用系统资源，可能最终导致资源耗尽，程序崩溃。

协程泄露的原因通常有两种：

- 有些协程在完成它们的工作后**没有被正确地停止。**
- 有些协程**因为阻塞而无法退出。**

解决方法：

- 通过`select`语句配合`time.After`进行超时控制。
- 使用context包来传递取消信号进行控制。
- 使用sync.WaitGroup等待所有的协程完成

Go 中主协程如何等待其余协程退出?

使用sync包下的WaitGroup

- Add()是协程计数，表示当前有多少个正在执行的协程
- Done()减去一个计数，表示一个携程已经执行完成
- Wait()阻塞直到所有的任务完成。

注意一个问题：

Add()操作不能放到goroutine执行函数中，因为创建goroutine后会将其放到队列中，不一定会马上调度执行。协程计数可能小于预期值，Wait()操作提前结束了。

怎么控制并发数量

有缓冲通道

```
func main() {
    count := 100
    c := make(chan struct{}, count)
    sum := 10000
    wg := sync.WaitGroup{}
    for i := 0; i < sum; i++ {
        c <- struct{}{} //任务数大于 count 时会阻塞
        wg.Add(1)
        go func() {
            defer wg.Done()
            // Simulate some work
            <-c
        }()
    }
    wg.Wait()
}
```

GMP调度模型

线程模型有哪些？为什么 Go Scheduler 需要实现 M:N 的方案?

一对一、一对多、多对多

1:1模型：每一个用户级线程绑定一个内核级线程。创建和销毁内核级线程，以及线程**上下文切换的开销都相对较大**，且内核级线程占用的资源较多。

N:1模型：多个用户级线程绑定一个内核级线程。这种模型可以支持大量用户级线程，因为用户级线程比内核级线程轻量。但**一旦一个用户级线程开始执行系统调用，整个内核级线程都会被阻塞**，导致其他所有用户级线程都无法执行。

M:N模型：多个用户级线程绑定多个内核级线程。协程切换发生在用户态，而且如果发生系统调用阻塞了当前内核级线程，还可以将它的协程移交到其他线程运行。

早期GM模型，为什么GMP要有P?

早期GM模型没有P，有很多缺点：

全局锁竞争：多个M同时访问全局队列需要加锁，频繁锁竞争导致性能下降。

M转移G：M执行G时新创建的G可能会到其他的M中执行，G的执行状态和信息需要在M之间传递，这涉及到了线程上下文切换，消耗了额外的系统资源。而引入P可以让相关的G都在同一个M上运行，遵循了局部性原理。

系统调用：当M执行G时遇到系统调用（如I/O操作）导致阻塞时，M会被挂起并等待G执行完毕。这期间，M无法执行其他G，导致资源闲置。

P在GMP模型中的作用及优势:

减少锁竞争：每个 P 有自己的本地队列，大幅度的减轻了对全局队列的直接依赖，避免了频繁的加锁，减少了锁竞争。

提高资源利用率：通过窃取机制，如果 P 的本地队列为空时，则会从全局队列或其他 P 的本地队列中窃取可运行的 G 来运行，减少空转，提高了资源利用率。

GMP原理（重要）

- G(goroutine)：是goroutine，就是Go语言中实现的协程，由go runtime系统进行创建和控制，里面存着一些goroutine的栈信息，状态以及要执行的任务函数。
- M(machine)：它是Go对操作系统线程的封装。goroutine必须要在m上执行，M必须绑定一个P才能进入一个循环调度，不断从P的本地队列和全局队列中获取G执行。
- P(Processor)：P是上下文处理器，里面有goroutine运行的资源，当P有任务的时候，就会去创建或唤醒一个线程处理它的队列中的任务。

G 代表着 goroutine，P 代表着上下文处理器，M 代表 thread 线程。

在 GPM 模型，有一个全局队列（Global Queue）：存放等待运行的 G，还有一个 P 的本地队列：也是存放等待运行的 G，但数量有限，不超过 256 个。

创建Goroutine：通过 `go func()` 创建的新的 Goroutine会先保存在一个P的本地队列，如果本地队列已满就会放到全局队列

调度执行：P收到goroutine任务后就会去创建或者唤醒一个M进行绑定。M与P进行绑定之后，M就会进入一个调度循环，从 P 的本地队列中获取一个 G 来执行。

工作窃取：如果P的本地队列为空，M 就会向全局队列偷取部分G，如果全局队列为空就从其他p的队列中偷取，**以确保Goroutine在不同处理器P上的负载均衡，充分利用资源。**

（**协程切换**：如果goroutine发生了阻塞，**在用户态下就可以完成协程的切换**，将当前的goroutine放到等待队列中等待唤醒。M会重新从队列中获取goroutine执行。）

移交机制：如果发生了系统调用，那么不仅协程会阻塞，运行协程的线程也会阻塞，处理器P不会等待系统调用完成，而是会让M与P会进行解绑，然后寻找一个空闲的线程处理接下来的任务。**防止资源的闲置。**

抢占式调度

在执行goroutine中，防止其他goroutine被饿死，`一个goroutine最多占用CPU 10ms`，这就是goroutine不同于coroutine的一个地方。

早期是**基于协作式的抢占式调度**，编译器**在有函数调用的地方插入“抢占”代码**（埋点），**通过系统监控为goroutine设置抢占标记**，通过抢占标记判断是否触发抢占。这种方法只能在有函数调用的情况下使用，对于循环计算没有函数调用的goroutine无效

基于信号的抢占式调度。这种方法的思路是**M会注册一个信号处理函数**，**通过系统监控运行的G**的执行时间，如果协程独占时间超过了10ms，就会向线程**发送抢占信号**，M收到抢占信号后，就切换goroutine执行。

G和M有哪些状态

G：与线程的状态类似，空闲、就绪、运行、阻塞、中止，还有一个系统调用状态。

- **_Gidle**：**空闲状态**，表示G刚刚新建，仍未初始化。
- **_Grunnable**：**就绪状态**，表示G在运行队列中，等待M取出并运行。
- **_Grunning**：**运行状态**，表示M正在运行这个G，这时候M会拥有一个P。
- **_Gsyscall**：**系统调用**，表示M正在运行这个G发起的系统调用，这时候M并不拥有P。
- **_Gwaiting**：**阻塞状态**，表示G在等待某些条件完成，这时候G不在运行也不在运行队列中（可能在channel的等待队列中）。
- **_Gdead**：**中止状态**，表示G未被使用，可能已执行完毕（并在freelist中等待下次复用）。
- **_Gcopystack**：**栈复制中**，表示G正在获取一个新的栈空间并把原来的内容复制过去（用于防止GC扫描）。

M：空闲、自旋、运行、系统调用阻塞

空闲状态：M发现无待运行的G时会进入休眠，并添加到空闲M链表中，这时M并不拥有P。

自旋中（spinning）：M正在从运行队列获取G，这时候M会拥有一个P。

运行状态：M正在执行go代码，这时候M会拥有一个P。

阻塞状态：系统调用阻塞，M与P可能解绑

Go什么时候发生阻塞？阻塞时调度器会怎么做

等待channel、互斥锁等待、time.sleep或者系统调用

更新goroutine的状态，将goroutine与线程解绑，然后将G放到等待队列中，M重新获取一个G执行。如果是系统调用阻塞的话，进入内核态，线程也阻塞了，那么P与M也会解绑，P会重新寻找或创建一个线程执行当前的任务。

本地队列要加锁吗？

要，因为本地队列也会被其它处理器窃取goroutine，也是共享资源。但是加的是乐观锁，通过CAS操作获取G，没获取到就自旋。

G和P和M的数量问题

- **G理论上无限制**：G的数量在理论上是没有上限的，只要系统的内存足够，就可以创建大量的goroutine
- **P的数量默认等于CPU核数**，由启动时环境变量 `$GOMAXPROCS` 或者由 `runtime` 包的方法 `GOMAXPROCS()` 决定。P的数量决定同时执行的任务的数量
- **M的数量**：
  - **动态创建和销毁**：当一个M上的所有goroutine都阻塞时，该M可能会被销毁，而当有goroutine等待执行但没有可用的M时，会创建新的M。
  - **M的最大数量默认为10000个**，`runtime/deBug` 中的 `SetMaxThreads( )` 函数可设置M的最大数量

P与M 1:1进行绑定，但M的数量会略大于P的数量，**多出来的M将会在G产生系统调用时发挥作用**，当有M因为系统调用阻塞时，P与这个M解绑，空闲的M将获取这个P继续执行其中剩下的任务

本地队列：无锁，减少全局锁竞争

work-stealing机制：本地队列为空的P可以尝试从其他P本地队列偷取一半的G补充到自身队列，防止不同P的闲忙差异过大。

调度类型：主动调度(gosched)、被动调度(gopark)、正常调度(goexit0)、抢占调度(系统调用)

草稿

schedule获取可执行的goroutine

本地队列有直接返回

从全局队列偷取

从其他本地队列偷取，需要注意，虽然本地队列是属于 p 独有的，但是由于 work-stealing 机制的存在，其他 p 可能会前来执行窃取动作，因此操作仍需加锁。但是，由于窃取动作发生的频率不会太高，因此当前 p 取得锁的成功率是很高的，因此可以说p 的本地队列是接近于无锁化，但没有达到真正意义的无锁.

execute：更新 g 的状态信息runnable->running，建立 g 与 m 之间的绑定关系，调度器调度次数加1；调用 gogo 方法，执行 goroutine 中的任务.

gosched：协作式调度主动让出，在协作式的抢占式调度中，g会调用 mcall 方法将执行权归还给 g0，并由 g0 调用 gosched_m 方法，g的状态running->runnable，解绑g和m，将g添加到全局队列，通过schedule开始新一轮调度

gopark：**被动调度**，g会调用 mcall 方法切换至 g0，g0调用 park_m 方法将 g 置为阻塞态，g的状态running->waiting，解绑g和m，将g添加到等待队列，通过schedule开始新一轮调度

goready: 当因被动调度陷入阻塞态的 g 需要被唤醒时，会由**其他协程执行 goready 方法**将 g 重新置为可执行的状态。g的状态由waiting->runnable，然后将当前 g 添加到唤醒者 p 的本地队列中，如果队列满了，会连带 g 一起将一半的元素转移到全局队列.

goexit0：**正常调度**，当 g 执行完成时，会先执行 mcall 方法切换至 g0，然后g0调用 goexit0 方法，g的状态running->dead，解绑g和m，通过schedule开始新一轮调度

retake：**抢占式调度**，抢占调度的执行者不是 g0，而是一个全局的 monitor g。执行步骤是先将当前 p 的状态更新为 idle，然后步入 handoffp 方法中，判断是否需要为 p 寻找接管的 m（因为其原本绑定的 m 正在执行系统调用）

reentersyscall：m在执行系统调用之前，g0先执行reentersyscall，保存当前 g 的执行环境，将 g 和 p 的状态更新为 syscall，将 g 和 p 的状态更新为 syscall，将 p 添加到 当前 m 的 oldP 容器当中，后续 m 恢复后，会优先寻找旧的 p 重新建立绑定关系

exitsyscall：当 m 完成了内核态的系统调用之后，此时会步入位于 exitsyscall 函数中，尝试寻找 p 重新开始运作，方法执行之初，此时的执行权是普通 g.倘若此前设置的 oldp 仍然可用，则重新和 oldP 绑定，将当前 g 重新置为 running 状态，然后开始执行后续的用户函数，old 绑定失败，则调用 mcall 方法切换到 m 的 g0，并执行 exitsyscall0 方法，将 g 由系统调用状态切换为可运行态，并解绑 g 和 m 的关系，从全局 p 队列获取可用的 p，如果获取到了，则执行 g，如若无 p 可用，则将 g 添加到全局队列，当前 m 陷入沉睡. 直到被唤醒后才会继续发起调度

调度测略：

轮转

系统调用：

调度结束

抢占调度

垃圾回收 GC

过程和原理

标记清除法 v1.3

首先要开启STW(Stop The World)防止并发问题，从根对象出发，标记所有可达对象，然后GC会回收所有未被标记的对象的内存。

因为整个GC期间需要开启STW，将整个程序暂停，性能损耗很大。但如果不开STW，让GC和程序并发执行，又会有漏标的问题（已经被标记的对象A，引用了新的未被标记的对象B）。

漏标导致误删问题

GC与程序并发执行时，如果某个黑色对象引用白色对象并且没有灰色对象间接引用这个白色对象，**就会导致这个白色的对象无法被访问标记而被错误的回收了。**

通过破坏两个条件之一就可以解决这个问题，因此就有了

- **强三色不变式：强制性不允许黑色对象引用白色对象**
- **弱三色不变式：允许黑色对象引用白色对象，但是白色对象必须存在其他灰色对象对它的引用**

通过这两个规则就分别提出了两种实现机制：**插入写屏障和删除写屏障。**

三色标记法 v1.5

- **初始化阶段**：首先短暂的STW将所有对象初始化为白色，然后将所有根对象标记为灰色。
- **并发标记阶段**：从根对象中开始遍历所有对象，当访问一个灰色对象的时候，就将它的所有引用标记为灰色，并把这个灰色对象标记为黑色。重复执行这个步骤，直到所有的可到达的对象变成了黑色。
  - **屏障处理**：在标记过程中会通过屏障机制来处理堆区对象的引用变化，防止并发过程产生漏标问题导致对象被错误的回收。只能在堆区使用。
  - **栈区标记**：栈区没有写屏障，老版本只能在标记完所有对象后，启用STW全局暂停，重新扫描一次，防止栈区漏标。1.8新版本使用混合写屏障机制，GC刚开始的时候，会将栈区所有可达的对象直接标记为黑色，且新创建的对象直接标为黑色，不需要STW全局暂停就避免了漏标。
- **清除回收阶段**：回收所有白色对象的内存空间

插入写屏障

当一个对象引用另外一个白色对象时，就直接将白色对象标记为灰色。实现了强三色不变式，不会出现黑色对象引用白色对象的情况，解决了漏标问题。

缺点：栈区是没有写屏障的，因为栈区有大量的对象和高频的调动，施加写屏障会有较大的性能负担。因此在对所有标记完成后，为防止栈中漏标的情况，需要通过STW保护栈区，然后重新进行扫描，防止栈区有白色对象漏标。

删除写屏障

当一个灰色对象删除对一个白色对象的引用的时候，这个白色对象会被标记为灰色。实现了弱三色不变色，让灰色对象到白色对象的路径不会中断。这样即使黑色对象引用了白色对象，只要还有其他的对象引用了这个白色对象，就不会出现漏标问题。

缺点：可能出现多标的问题，删除引用导致白色对象变成灰色对象，可能这个灰色对象已经不可达本来就是要被回收的，但被标记为灰色不会被回收，只能由下一轮GC来回收。

三色标记+混合写屏障 go v1.8

1.8版本使用不仅使用了混合写屏障还对栈区标记作了优化，GC刚开始的时候，会将栈区所有可达的对象直接标记为黑色，且新创建的对象直接标为黑色，不需要STW全局暂停就避免了漏标。

插入写屏障与删除写屏障相结合使用

总结

go 1.3版本之前是简单的标记清除法，为了防止漏标需要GC全局暂停。go 1.5版本采用三色标记法引入了屏障机制让GC标记阶段可以和程序并发执行，从而降低了STW的时间，提高了GC效率。go 1.8通过混合写屏障进一步降低了STW的时间。

GC触发时机

- **阈值**：当分配的内存达到某个阈值会触发，这个阈值默认是上一次GC分配的内存的两倍。阈值 = 上次GC内存分配量 × (1 + GOGC/100)，GOGC为环境变量默认为100
- **定期**：默认2min触发一次gc，src/runtime/proc.go:forcegcperiod
- **手动**：调用runtime包中的GC()函数手动触发，runtime.gc()

GC调优

- 调整环境变量**GOGC**，从而增大因为内存分配引起GC的阈值，降低GC的频率。
- 可以通过**sync.pool**对象池复用对象，减少内存分配和回收的次数。
- 控制**goroutine**的数量，减少内存分配，降低GC负担

三色标记法STW出现时机

- 开始标记阶段之前：扫描根对象标记为灰色
- 标记结束阶段：统计存活对象
- 老版本标记结束后，还要对栈区启动STW，重新进行扫描标记。

go中的内存逃逸

逃逸分析：逃逸分析是Go编译器在编译时执行的一个过程，用于确定一个变量是否“逃逸”出了其原始的作用域。发生了逃逸，编译器会将其分配到堆上，而不是栈上。

内存逃逸（memory escape）：它指的是变量原本在栈上，但由于某些原因被分配到了堆上。这种情况通常发生在变量的生命周期超出了其原始作用域，导致编译器无法在栈上安全地管理这些变量的内存。

产生原因：

- **变量的生命周期超出作用域**：在**函数内部声明的变量，如果在函数返回后仍然被引用**，就会导致内存逃逸。这些变量将被分配到堆上，以确保它们在函数返回后仍然可用。
- **使用闭包**：在Go中，闭包（函数值）可以捕获外部变量，如果**闭包引用了函数的局部变量**，那么这些变量也会逃逸到堆上
- **大对象和大切片**：当在栈上分配的对象或切片超过了一定的大小限制时，可能会被分配到堆上，以**避免栈溢出。**

为什么小对象多了会造成GC压力

GC回收采用的是标记回收法，小对象多了会增加标记阶段的工作量，增加GC负担。

小对象的频繁分配和回收还可能导致**内存碎片化。**内存碎片化是指内存中存在许多小块的不连续空间，这些空间无法被有效利用来存储新的大对象，增加GC压力。

GC的缺点

stw

频繁的GC会影响程序性能

泛型

反射

golang中的tag如何解析?

context

context结构原理

context包提供了一个context的接口，用于goroutine之间共享数据和传递一些信号，主要方法就是：

- `Deadline`：返回当前`Context`何时会被取消。如果`Context`不会被取消，则返回ok为false。
- `Done`：返回一个通道，当`Context`被取消或超时时，该通道会被关闭。
- `Err`：返回`Context`为何被取消。
- `Value`：返回与`Context`相关的值，这些值必须是线程安全的。

context包提供了两个函数用于创建Context对象：context.Background()和context.TODO()

context包的作用

- 传递数据 WithValue
- 超时控制 WithTimeOut WithDeadline

扩展

你用的什么版本的go?

不同版本有什么区别？每个版本的特性