## ██基础语法

### go语言的优势

**1并发：**

Golang 针对并发进行了优化，并且在规模上运行良好

2**垃圾回收：**

自动垃圾收集明显比 Java 或 Python 更有效，因为它与程序同时执行

### 包

每个 Go 源文件都属于一个包

Go 语言的入口 main() 函数所在的包（package）叫 main，main包 想要引用别的代码，必须同样以包的方式进行引用

Golang 的标准包是 fmt，其中包含格式化和打印功能，如 Println().

"\_"(下划线)，可以简单理解为赋值但以后不再使用

引入包，会先调用包中的初始化函数，这种使用方式仅让导入的包做初始化，而不使用包中其他功能

### 常用数据类型

**数据类型：**

布尔1，字符串4，数组，切片，map8，channel，结构，结构体，

指针，函数

**Cap：**

Cap作用于数组，切片，通道

**缺省：**

Interface，指针的缺省值是nil，

bool类型的缺省值是false

其他的都是对应类型的零值，

**指针运算：**

可以通过“&”取地址

可以通过“\*”取指针指向的数据

结构体type A struct{}和结构体type A=struct{}区别就是type A=struct{}定义了一个新的数据类型，这个数据类型为A

### 堆和栈

堆（Heap）和栈的区别：**管理，内存，速度**

**管理：**

堆一般来讲是人为手动进行管理，手动申请、分配、释放。

栈由编译器进行管理，自动申请、分配、释放。

**对象大小：**

堆内存大小并不定，一般会存放较大的对象

栈内存大小并固定，一般不会放太大的对象

**速度：**

堆分配相对慢，涉及到的指令动作也相对多

栈分配速度快（用的是一级缓存）

**变量存放堆还是栈：**

由编译器决定

如果函数外部没有引用，则优先放到栈中；

如果函数外部存在引用，则必定放到堆中；

### 闭包

**全局变量的特点：**

1.常驻内存

2. 污染全局

**局部变量的特点：**

1.不常驻内存

2.不污染全局

而Go语言的闭包可以做到

1.可以让变量常驻内存

2.可以让变量不污染全局

所以闭包主要是为了避免全局变量的滥用。

**概念：**

引用了外部变量的匿名函数  
  创建闭包的常见方式就是在一个函数内部创建另一个函数，内函数可以 访问外函数的变量

闭包有权访问另一个函数作用域中的变量的函数

**优缺点：**

1让外部访问函数内部变量成为可能；

2局部变量会常驻在内存中；问速度会特别快，始终操作的都是这个变 量的本身，而不是其副本，修改时也特别方便

3可以避免使用全局变量，防止全局变量污染；

**缺点：**

有一块内存空间被长期占用，而不被释放，会造成内存泄漏

**常见形式：**

一个外部函数，一个内部函数

内部函数使用了外部函数的变量（外部函数的局部变量的操作）

外部函数返回的是内部函数

### 回调函数

**将一个函数作为另一个函数的参数**  
  fun1(),fun2()  
    将fun1函数作为fun2这个函数的参数。  
  fun2函数：高阶函数  
    接受了一个函数作为参数的函数，高阶函数  
  fun1函数：回调函数  
    作为另一个函数的参数的函数，叫做回调函数

### 深拷贝浅拷贝

**深拷贝：**

拷贝的是数据本身，创造一个新对象，新创建的对象与原对象不共享内 存，新创建的对象在内存中开辟一个新的内存地址，新对象值修改时不会 影 响原对象值

**实现深拷贝的方式：**

1copy(slice2, slice1)

2遍历append赋值

**浅拷贝：**

拷贝的是数据地址，只复制指向的对象的指针，此时新对象和老对象指 向的内存地址是一样的，新对象值修改时老对象也会变化

**实现浅拷贝的方式：**

1引用类型的变量，默认赋值操作就是浅拷贝slice2 := slice1

### 接口

**概念：**

1interface 是方法声明的集合

2任何类型的对象实现了在interface 接口中声明的全部方法，则表明 该 类型实现了该接口。

3interface可以作为一种数据类型，实现了该接口的任何对象都可以给 对应的接口类型变量赋值。

**意义：**

**1实现多态:**就是我们可以根据interface类 型来设计API接口，

**2隐藏具体实现：**只限定方法名称，具体的实现不用管。它分离方法和 具体代码实现逻辑

**注意：**

a. interface 可以被任意对象实现，一个类型/对象也可以实现多个 interface

b. 方法**不能重载**，如 eat(), eat(s string) 不能同时存在

## 实现set

type Set map[interface{}]struct{}

func (s \*Set)Add(k interface{}) {

(\*s)[k] = struct{}{}

}

### 单引号，双引号

单引号，表示byte类型或rune类型，对应 uint8和int32类型，默认是 rune 类型。

双引号，才是字符串，实际上是字符数组。可以用索引号访问某字节，也可以用len()函数来获取字符串所占的字节长度。

### 值类型、引用类型

**值类型：**

int、 string、 struct，传递进函数，在函数中就无法修改原内容数据

**引用类型**：

指针、 map、slice、 chan，传递进函数，在函数中就可以修改原内容数据

### 值传递和引用传递

**值传递：**

将**实参的值**传递给形参，形参是实参的一份拷贝，实参和形参的内存地址不同。

函数内对形参值内容的修改，是否会影响实参的值内容，取决于参数是否是**引用类型**

**引用传递：**

将**实参的地址**传递给形参

函数内对形参值内容的修改，将会影响实参的值内容

**Go只有值传递：**

Go 语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。

参数如果是非引用类型（int、string、struct等这些），这样就在函数中就无法修改原内容数据；

参数如果是引用类型（指针、map、slice、chan等这些），这样就可以修改原内容数据。

### 继承，封装，多态

**继承：（代码简洁）**

通过结构体嵌套来实现继承，如果一个struct嵌套了另一个匿名结构体，那么这个结构体可以直接访问匿名结构体的字段和方法，从而实现继承特性。

**封装：（安全）**

封装主要通过访问**权限的控制**来实现

go语言定义变量或方法为小写字母开头，则只能所在包能够访问，定义为大写字母开头，则可以跨包调用

**多态：（解耦接口和实现类之间的联系）**

执行的时候，能够**根据子类的类型去执行子类当中的方法**。

也就是说实现我们用相同的调用方式调出不同结果或者是功能的情况，这种情况就叫做多态。

通过实现interface定义的方法就是struct实现了接口

猫和狗和人都是哺乳动物。这三个类都有一个say方法，大家都知道猫、狗以及人类的say是不一样的，猫可能是喵喵叫，狗是汪汪叫，人类则是说话。

## █内置函数

### sync.Once

**单例初始化：**

Once 可以用来执行且仅仅执行一次动作，常常用于单例对象的初始化场景。

**只运行一次：**

init 函数是在文件包首次被加载的时候执行，且只执行一次

### %v 和 %+v 的区别

“%v”只会打印字段值value信息

“%+v”会以字段键值对的形式key-value格式打印

### make和new的区别：

**相同点：**

均分配内存空间，

**不同点：**

**使用场景区别：**

Make一般用于内置引用类型（ 切片、 map 和管道）

new 一般用于值类型和用户定义的类型

**返回值区别：**

make函数原型如下，返回的是slice、map、chan类型本身

new函数原型如下，返回一个指向该类型内存地址的指针

### Printf()，Sprintf()， FprintF()的异同

**相同点:**

都是格式化输出

**不同点：**

**输出的目标不一样**

Printf 是标准输出，一般是**屏幕**，也可以重定向。  
 Sprintf()是把格式化字符串输出到指定的**字符串**中。  
 Fprintf()是把格式化字符串**输出到文件**中。

### main函数和init函数

**相同点：**

main函数和init函数没有参数和返回值

**不同点：**

init可以应用于任意包中，且可以重复定义**多个**。

main函数只能用于main包中，且只能定义**一个**

init函数先于main函数自动执行，不能被其他函数调用

### tag的作用

**定义属性：**

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。

## ██数据类型实现原理

## Uint相减问题

当uint为正数时，没有问题，为负数时，系统会把负数的1的正负位当做最高进制来算，造成数值很大

## rune 类型

相当int32

golang中的字符串底层实现是通过byte数组的，

中文字符在unicode下占2个字节，在utf-8编码下占3个字节，

而golang默认编码正好是utf-8

byte等同于int8，常用来处理ascii字符

rune 等同于int32,常用来处理unicode或utf-8字符

## Struct可以比较吗

需要具体情况具体分析，

**1不同类型的struct**

不能进行比较，编译期就会报错（GoLand 会直接提示）

**2同类型的struct**

也分为两种情况，

**（1）**struct的所有成员都是可以比较的，则该 strcut 的不同实例可以比较

**（2**）struct中含有不可比较的成员（如 Slice），则该 struct 不可以比较

**不可被比较的类型:**  
① slice，因为slice是引用类型，除非是和nil比较  
② map，和slice同理，如果要比较两个map只能通过循环遍历实现  
③ 函数类型

**以上引用类型不支持比较的原因**

引用类型，如果你要比较的话，是想去比较他的值还是说他的地址？这就相当于在比较的时候会有歧义产生，因此Go从语言层面上直接杜绝了引用类型的比较；  
 就像两个slice，如果要去实现他们的 == 比较，你是想去比较len是否相等？还是cap是否相同？还是底层数组的地址是否相同？还是每个元素的值是否相同？  
 所以说歧义很大的，就干脆不支持比较了。

## Srting

**string数据结构：**

**str：**字符串的首地址；

**len：**字符串的长度；

**字符串不允许修改：**

像C++语言中的string，其本身拥有内存空间，修改string是支持的。 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转换成string不一定会拷贝内存：**

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）；

**string和[]byte使用场景：**

string 擅长的场景：

需要字符串比较的场景；

不需要nil字符串的场景；

[]byte擅长的场景：

修改字符串的场景，尤其是修改粒度为1个字节；

需要切片操作的场景；

## Channel

## Channel实现原理

channel主要用于进程内各goroutine间通信

**数据结构：**

它主要由**环形队列**、、**等待队列**组成等

**环形队列：**

环形队列作为其缓冲区，队列的长度是创建chan时指定的

**等待队列：**

**被阻塞的goroutine将会挂在channel的等待队列中：；**

从channel**读数据**，如果channel缓冲区为空或者没有缓冲区，当前goroutine会被阻塞等待。

向channel**写数据**，如果channel缓冲区已满或者没有缓冲区，当前goroutine会被阻塞等待。

**读写**

**向channel写数据：**

如果缓冲区中有空余位置，则将数据写入缓冲区，结束发送

如果缓冲区内没有空余位置，则将当前协程加入**sendq（等待写消息队列）队列**，进入睡眠并等待被读协程唤醒

**向channel读数据**

如果缓冲区中有数据，则从缓冲区中取出数据，结束读取过程

如果缓冲区中有数据，则从将当前协程加入**recvq（等待读消息队列**）队列，进入睡眠并等待被写协程唤醒

**关闭channel：**

关闭channel时会把recvq中的G全部唤醒，本该写入G的数据位置为nil。

关闭channel会把sendq中的G全部唤醒，但这些G会panic。

## ****Channel缓冲区特点****

**同步与非同步：**

无缓冲的 channel 是同步的，

有缓冲的 channel 是非同步的，缓冲满时发送阻塞

 channel无缓冲时，发送阻塞直到数据被接收，接收阻塞直到读到数据；

channel有缓冲时，当缓冲满时发送阻塞，当缓冲空时接收阻塞。

**nil：**

如果给一个nil的 channel 发送数据，会造成永远阻塞。

如果从一个nil的 channel 中接收数据，会造成永久阻塞。

**panic：**

关闭值为nil的channel

关闭已经被关闭的channel

向已经关闭的channel写数据

给一个已经关闭的channel发送数据，引起panic

**特点：**

关闭的管道读数据仍然可以读数据

从一个已经关闭的 channel 接收数据，如果缓冲区中为空，则返回一个零值

## Channel线程安全

**原因：**

使用场景就是多线程，为了保证数据的一致性，必须实现线程安全

**原理：**

channel的底层实现中，hchan结构体中采用Mutex锁来保证数据读写安全。在对数据进行入队和出队操作时，必须先获取互斥锁，才能操作channel数据

## channel死锁

**概念：**

协程永久阻塞

两个以上的协程的执行过程中，由于竞争资源或由于彼此通信而造成的一种阻塞的现象。

**场景：**

无缓冲channel只写不读

无缓冲channel读在写后面

多个协程互相等待

## Channel应用场景

channel适用于数据在多个协程中流动的场景

**① 任务定时**

比如超时处理：

select {

case <-time.After(time.Second):

定时任务

select {

case <- time.Tick(time.Second)

**② 解耦生产者和消费者**

可以将生产者和消费者解耦出来，生产者只需要往channel发送数据，而消费者只管从channel中获取数据。

**③ 控制并发数**

一般要配合sync使用，在我来-刷题-协程

## **退出程序怎么防止无缓冲channel没有消费完（channel还留有数据没有**）

退出时将生产者关闭，不会产生多余的数据给消费者

## 怎么控制上游生产速度过快

channel长度可以与上下游的速度比例成线性关系

## Slice

## Slice实现原理

**原理：**

从切片数据结构看，切片是基于数组实现的,

array指针指向底层数组，len表示切片长度，cap表示底层数组容量。

**创建方式：**

直接声明

使用make创建

使用数组创建，使用数组来创建Slice时，Slice将与原数组共用一部分内存

## slice的扩容

**概念：**

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

**扩容规则：**

如果期望容量大于当前容量的两倍，就会使用期望容量

如果小当前容量的两倍的话，分为以下几种情况：

如果原Slice容量小于1024，则新Slice容量将扩大为原来的2倍；

如果原Slice容量大于等于1024，则新Slice容量将扩大为原来的1.25倍；直到新容量大于或等于期望容量

**原因：**

当切片比较小时（容量小于 1024），则采用较大的扩容倍速进行扩容（新的扩容会是原来的 2 倍）， **避免频繁扩容**，从而减少内存分配的次数和数据拷贝的代价。

当切片较大的时（原来的 slice 的容量大于或者等于 1024）， 采用较小的扩容倍速（新的扩容将扩大大于或者等于原来 1.25 倍），**主要避免空间浪费**

## slice扩容前后是否相同

**情况一：**

原数组还有容量可以扩容（实际容量没有填充完），这种情况下，扩容以后的数组还是指向原来的数组，对一个切片的操作可能影响多个指针指向相同地址的 Slice。

**情况二：**

原来数组的容量已经达到了最大值，再想扩容， Go 默认会先开一片内存区域，把原来的值拷贝过来，然后再执行 append() 操作。这种情况丝毫不影响原数组。 要复制一个 Slice，最好使用 Copy 函数。

**这里的切片扩容指的是超过len，而不是超过cap**



## slice线程不安全

**概念：**

多个线程访问同一个对象时，调用这个对象的行为都可以获得正确的结果，那么这个对象就是线程安全的。

若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

原理：

slice底层结构并**没有使用加锁**等方式，不支持**并发读写**，所以并不是线程安全的，使用多个 goroutine 对类型为 slice 的变量进行操作，每次输出的值大概率都不会一样，与预期值不一致; slice在并发执行中不会报错，但是数据会丢失

## 数组和切片的区别及应用场景

**相同点：**

只能保存一组相同类型的数据结构

都是通过下标来访问

**不同点**

**类型：**

数组是值类型，切片是引用

**长度可变性：**

数组长度不可变，切片长度可变

**使用场景：**

固定长度或者已知长度使用数组性能更优

不定长度，需要append…大部分情况下使用slice

## Map

## Map的key

如果能比较就能作为map的key，不能比较就不能作为map的key

**可以比较：**布尔，整数，字符串，通道，接口，如果结构体的所有字段可比较，那也行

**不可比较：**切片，函数，map

## Map的实现原理

Golang的map使用哈希表作为底层实现

一个哈希表里可以有多个哈希表节点，也即bucket，

每个bucket就保存了map中的一个或一组键值对，每个bucket最多可以存储8个键值对

**Hmap（哈希表）的数据结构:**

**Count, B, overflow, Buckets, oldbuckets**

**Count：**

当前保存元素的个数

**B：**

buckeS数组的长度2^B

**overflow:**

溢出桶的数量

**Buckets:**

指向buckets数组的指针，数组大小为2^B，如果元素个数为0，它为nil。

**oldbuckets :**

如果发生扩容，oldbuckets是指向老的buckets数组的指针

**Bmap（bucket）的数据结构：**

**Tophash，Data，Overflow**

**Tophash：**

长度为8的数组，哈希值相同的键（准确的说是哈希值低位相同的键）存入当前bucket时会将哈希值的高位存储在该数组中，以方便后续匹配。

（低位找到数据存在哪个桶，高位找到在桶中的哪个位置）

**Data：**

存放的是key-value数据，存放顺序是key/key/key/…value/value/value，如此存放是为了节省字节对齐带来的空间浪费。

**Overflow：**

指针指向的是下一个bucket，据此将所有冲突的键连接起来。

## 哈希冲突

**概念：**

当有两个或以上数量的键被哈希到了同一个bucket时，我们称这些键发生了冲突。

**解决办法：**

Go使用链地址法来解决键冲突。由于每个bucket可以存放8个键值对，所以同一个bucket存放超过8个键值对时就会再创建一个键值对，用类似链表的方式将bucket连接起来。

bucket数据结构指示下一个bucket的指针称为overflow bucket，意为当前bucket盛不下而溢出的部分。事实上哈希冲突并不是好事情，它**降低了存取效率，**

## Map扩容及其负载因子

**负载因子：**

负载因子 = 键数量/bucket数量

即每个 bucket 桶存储的平均元素个数。

**场景：**

负载因子过小，说明空间利用率低

负载因子过大，说明冲突严重，存取效率低

Go则在在负载因子达到6.5时才会触发rehash，也就是申请bucket，对所有的键值对进行重新组织，使得这些键值均匀的分布在bucket中。

**扩容条件：**

1负载因子 > 6.5：

即平均每个bucket存储的键值对达到6.5个。触发**增量扩容**

2overflow数量 > 2^15：

即overflow数量超过32768时。触发**等量扩容**

**扩容规则：**

**增量扩容，等量扩容**

**增量扩容：**

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

**搬迁策略：**

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。

这个时候hmap中的oldbucket指向原老旧的bucket，而buckets指向了新申请的bucket

**等量扩容**

等量扩容，实际上并不是扩大容量，buckets数量不变，重新做一遍类似增量扩容的搬迁动作，把松散的键值对重新排列一次，以使bucket的使用率更高，进而保证更快的存取（冲突严重的话，最差变成链表O（n））

**原因：**

在极端场景下，比如不断的增删，而键值对正好集中在一小部分的bucket，这样会造成overflow的bucket数量增多，但负载因子又不高，从而无法执行增量搬迁的情况

此时进行一次等量扩容，即buckets数量不变，经过重新组织后overflow的bucket数量会减少，即节省了空间又会提高**访问效率**

## Map查找和插入

**查找：**

1跟据key值算出哈希值

2取哈希值低位与hmpa.B取模确定bucket位置

3取哈希值高位在tophash数组中查询，挨个和桶中tophash比较

4如果tophash[i]中存储值也哈希值相等，则去找到该bucket中的key值进行比较

5当前bucket没有找到，则继续从下个overflow的bucket中查找。

6如果当前处于搬迁过程，则优先从oldbuckets查找

**注**：如果查找不到，也不会返回空值，而是返回相应类型的0值。

**插入：**

1跟据key值算出哈希值

2取哈希值低位与hmap.B取模确定bucket位置

3查找该key是否已经存在，如果存在则直接更新值

4如果没找到将key，将key插入

## Map删除

**1定位：**

首先通过计算key找到指定的桶和位置，如果找不到就查找溢出桶，直到找到溢出桶链表的末尾。

**2清空数据：**

如果找到了key，对 key 或者 value 进行“清零”操作，将对应位置的 tophash 值置成 Empty

**3内存是否释放：**

**如果删除的元素是值类型，如int，float，bool，string以及数组和struct，map的内存不会自动释放**

**如果删除的元素是引用类型，如指针，slice，map，chan等，map的内存会自动释放，但释放的内存是子元素应用类型的内存占用**

## Map遍历为无序

**原因：**

1map在遍历时，并不是从固定的0号bucket开始遍历的，每次遍历，都会从一个随机值序号的bucket，再从其中随机的cell开始遍历

2map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，从这个角度看，遍历map的结果就不可能是按照原来的顺序了

**如何顺序遍历：**

1取出key放在切片中，

2对key进行排序

3然后通过切片中有序key去访问map

## Map是非线程安全

map默认是并发不安全的，同时对map进行并发读写时，程序会panic。

Map在查找，赋值，遍历，删除操作都会检测写标志，一旦发现写标志位为1，则直接panic。

**原因：**

Go 官方在经过了长时间的讨论后，认为 Go map 更应适配典型使用场景（不需要从多个 goroutine 中进行安全访问），而不是为了小部分情况（并发访问），导致大部分程序付出加锁代价（性能），决定了不支持。

## Sync.map

**sync.map原理：**

使用read和dirty两个map来进行读写分离，通过空间换时间的方式，降低锁时间来提高效率。

1通过 read 和 dirty 两个字段将读写分离，读的数据存在只读字段 read 上，将最新写入的数据则存在 dirty 字段上

1读取时会先查询read，不存在再查询dirty，写入时则只写入dirty

2读取read并不需要加锁，而读或写dirty都需要加锁

3另外有 misses 字段来统计read被穿透的次数（被穿透指需要读 dirty 的情况），超过一定次数则将dirty数据同步到 read 上

4对于删除数据则直接通过标记来延迟删除

**使用场景优点：**

适合读多写少的场景

**缺点：**

写多的场景，会导致read map缓存失效，需要加锁，冲突变多，性能急剧下降

**sync.map特点：**  
 sync.map 是线程安全的，读取，插入，删除也都保持着常数级的时间复杂度。

**原因：**

Go语言原生map并不是线程安全的，对它进行并发读写操作的时候，需要加锁。

**原生map并发读取缺点：**

解决并发读写 map 的思路是锁，但是效率低。

所以引入：sync.map  
 sync.map则是一种并发安全的map，在Go1.9引入。

## 并行和并发

**并行**

并行的关键是有同时处理多个任务的能力

指在同一时刻，有多条指令在多个处理器上(cpu)同时执行

多个线程时被多个cpu执行，那就是并行；

**并发**

并发的关键是有处理多个任务的能力，不一定要同时、

指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行。

多个线程同时被一个cpu 轮流切换着执行，那就是并发。

## 协程

## 协程实现原理

**Goroutine的数据结构：**

goid 唯一的goroutine的ID

sched goroutine 切换时，用于保存g的上下文

stack 栈空间

**最后会生成一个runtime.g 对象放入调度队列**

**goroutine状态**“

|  |  |
| --- | --- |
| 空闲中\_idle | G刚刚新建, 仍未初始化 |
| 待运行\_runnable | 就绪状态，G在运行队列中, 等待M取出并运行 |
| 运行中\_running | M正在运行这个G, 这时候M会拥有一个P |
| 等待中\_waiting | 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) |
| 已中止\_dead | G未被使用, 可能已执行完毕 |

**1创建：**

（1）**go关键字：**通过go关键字创建一个goroutine，goroutine会被设置成**runnable状态**

（2）**栈空间：**创建好的这个goroutine会新建一个自己的**栈空间**，同时在G的sched中维护栈地址与程序计数器这些信息。

（3）**放在队列：**每个G在被创建之后，都会被优先放入到本地队列中，如果本地队列已经满了，就会被放入到全局队列中

**2运行：**

goroutine 本身只是一个数据结构，真正让 goroutine 运行起来的是调度器。

**1running状态**：每M开始执行P的本地队列中的G时，goroutine会被设置成**running状态**

**2本地与全局：**如果某个 M 把本地队列中的G都执行完成之后，然后就会去全局队列中拿 G，这里需要注意，每次去全局队列拿 G 的时候，都需要上锁，避免同样的任务被多次拿。

**3偷取任务**：如果全局队列都被拿完了，而当前 M 也没有更多的 G 可以执行的时候，它就会去其他 P 的本地队列中拿任务，这个机制被称之为 work stealing 机制，每次会拿走一半的任务，向下取整，比如另一个 P 中有 3 个任务，那一半就是一个任务。

**4进入自旋**：当全局队列为空，M 也没办法从其他的 P 中拿任务的时候，就会让自身进入自旋状态，等待有新的 G 进来。最多只会有 GOMAXPROCS 个 M 在自旋状态，过多 M 的自旋会浪费 CPU 资源。

**3阻塞：**

channel的读写操作、等待锁、等待网络数据、系统调用等都有可能发生阻塞，会调用底层函数runtime.gopark()，会让出CPU时间片，让调度器安排其它等待的任务运行，并在下次某个时候从该位置恢复执行。

当调用该函数之后，goroutine会被设置成**waiting状态**

**4唤醒：**

处于waiting状态的goroutine，在调用runtime.goready()函数之后会被唤醒，唤醒的goroutine会被重新放到M对应的上下文P对应的runqueue中，等待被调度。

当调用该函数之后，goroutine会被设置成**runnable状态**

**5退出：**

当goroutine执行完成后，会调用底层函数runtime.Goexit()

当调用该函数之后，goroutine会被设置成dead状态

## 协程和线程的区别

**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：**为轻量级线程，与线程相比，协程不受操作系统的调度，协程的调度器由用户应用程序提供，协程调度器按照调度策略把协程调度到线程中运行

**1内存2创建和销毁的开销3切换的开销**

**1内存：**

Goroutine一个的栈内存消耗为 2 KB，实际运行过程中，如果栈空间不够用，会自动进行扩容

线程一个的栈内存消耗为 1 MB

**2创建和销毁的开销：**

goroutine 因为是由Goruntime负责管理的，创建和销毁的消耗非常小，是用户级。

线程创建和销毀都会有巨大的消耗，因为要和操作系统打交道，是内核级的，通常解决的办法就是线程池

**3切换的开销：**  
 goroutine切换约为 200 ns

只需要保存PC（程序计数器）、SP（堆栈指针寄存器，即栈顶的位置）、BP（栈帧指针，当前栈帧位置），这三个也是协程的上下文

完全在用户空间进行  
 线程切换会消耗 1000-1500 ns

需要保存线程Id、线程状态、[堆栈](https://so.csdn.net/so/search?q=%E5%A0%86%E6%A0%88&spm=1001.2101.3001.7020" \t "_blank)、寄存器状态

涉及特权模式切换，需要在内核空间完成

## 协程泄露

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。就是 goroutine 没有被关闭，或者没有添加超时控制，让 goroutine 一直处于阻塞状态，不能被GC

**原因：**

**阻塞：**Goroutine内进行channel/mutex 等**读写操作**被一直阻塞。

Goroutine内的业务逻辑进入长时间等待，有不断新增的 Goroutine 进入等待

**死循环：**Goroutine内的业务逻辑进入**死循环**，**资源一直无法释放。**

**泄露场景：**

如果输出的 goroutines 数量是在不断增加的，就说明存在泄漏

**1channel忘记初始化**

那么无论你是读，还是写操作，都会造成阻塞

**2发送不接收**

channel 发送数量 超过 channel接收数量，就会造成阻塞

**3接收不发送**

channel 接收数量 超过 channel发送数量，也会造成阻塞

**4sync.WaitGroup使用不当**

 wg.Add 的数量与 wg.Done 数量并不匹配，因此在调 用 wg.Wait 方法后一直阻塞等待

**排查：**

**单个函数：**

调用 runtime.**NumGoroutine** 方法来打印 执行代码前后Goroutine 的运行数量，进行前后比较，就能知道有没有泄露了。

**生产/测试环境：**

使用PProf实时监测Goroutine的数量，以清晰的看到goroutine的数量以及调用关系

**处理：**

goroutine泄漏有没有处理，设置timeout，select加定时器

## 协程控制并发数

**原因：**

如果不对goroutine加以控制而进行滥用的话，会造成耗尽系统资源导致程序崩溃，或者CPU使用率过高导致系统忙不过来

**方法：**

**1有缓冲channel：**

利用缓冲满时发送阻塞的特性

**2无缓冲channel：**

任务发送和执行分离，指定消费者并发协程数

## 协程控制并发顺序（主协程如何等其余协程完再操作）

WiteGroup，channel，context，锁

## 使用协程交叉打印数组

在我来-刷题-协程里面

## ██控制结构实现原理

## Select原理

go的 select 为 golang 提供了多路 IO 复用机制，和其他 IO 复用一样，用于检测是否有读写事件是否 ready。linux 的系统 IO 模型有 select，poll，epoll，go 的 select 和 linux 系统 select 非常相似。

select 结构组成主要是由 case 语句和执行的函数组成 select 实现的多路复用是：每个线程或者进程都先到注册和接受的 channel（装置）注册，然后阻塞，然后只有一个线程在运输，当注册的线程和进程准备好数据后，装置会根据注册的信息得到相应的数据。

**数据结构：**

**c：**表示case语句操作的管道，由于select数据结构中仅能存放一个管道，所以case语句只能处理一个管道

**kind：**类型kind表示该case的类型，分为读channel、写channel和default

**elem：**数据数据存放的地址

**实现逻辑：**

通过源码包里面的selectgo()函数

selectgo（）函数会从一组case语句中挑选一个case，然后返回命中case的下标。

**总结：**

Select语句中当所有的case被阻塞的时候，就执行的是default

select语句中除default外，每个case操作一个channel，要么读要么写 select语句中除default外，各case执行顺序是随机的

select语句中如果没有default语句，则会阻塞等待任一case

select语句中读操作要判断是否成功读取，关闭的channel也可以读取

## select应用场景

1永久阻塞

不希望主函数退出（协程处理任务的时候），使用select{}

3限时等待

### range

### range实现原理

实现原理

for init;cond;post{

iter\_init

index=index\_temp

value=value\_temp

original statements

}

1编译器会从for-range语句中提取出**初始化语句init**，**条件语句cond**，**和迭代语句post**。

2然后**通过一个for循环实现**

循环体中有两个赋值语句，分别对应的是for-range的两个返回值。

**range for slice**

循环开始前循环次数就已经确定了，所以循环过程中新添加的元素是没办法遍历到的

**range for map**

遍历map时没有指定循环次数，循环体与遍历slice类似。由于map底层实现与slice不同，map底层使用hash表实现，插入数据位置是随机的，所以遍历过程中新插入的数据不能保证遍历到

**range for channel**

channel遍历是依次从channel中读取数据,读取前是不知道里面有多少个元素的。

如果channel中没有元素，则会阻塞等待，

如果channel已被关闭，则会解除阻塞并退出循环。

### range性能优化

**切片遍历优化：**

**通过key去获得value，而不是直接在range接受value**

func RangeSlice(slice []int) {

for index, value := range slice {

\_, \_ = index, value

}

}

遍历过程中每次迭代会对index和value进行赋值，如果数据量大或者value类型为string时，对value的赋值操作可能是多余的，

可以在for-range中忽略value值，使用slice[index]引用value值

**Map遍历优化：**

**直接获取value可能比通过可以获取value更为高效**

func RangeMap(myMap map[int]string) {

for key, \_ := range myMap {

\_, \_ = key, myMap[key]

}

}

函数中for-range语句中只获取key值，然后跟据key值获取value值，虽然看似减少了一次赋值，但通过key值查找value值的性能消耗可能高于赋值消耗。能否优化取决于map所存储数据结构特征、结合实际情况进行

## ██调度模型

## 线程实现模型

**背景：**

**1单进程时代**

**概念：**

每个程序就是一个进程，直到一个程序运行完，才能进行下一个进程

**缺点**

无法并发，只能串行

进程阻塞所带来的 CPU 时间浪费

**解决办法:**

多线程/多进程

**2多进程/多线程时代**

**概念：**

一个线程阻塞，cpu 可以立刻切换到其他线程中去执行，这样从宏观来看，似乎多个进程是在同时被运行。

**缺点：**

占用内存高

上下文切换成本高

**解决办法：**

协程

**3协程时代：**

**概念：**

协程（用户态线程）绑定线程（内核态线程），cpu调度线程执行

**缺点：**

实现起来较复杂，协程和线程的绑定依赖调度器算法

**优点**

占用内存更小（几kb）

调度更灵活(runtime调度)

减小上下文开销，因为工作在用户态

**线程（协程）模型：**

**（1:1）；（N:1）；（M:N）**

**1:1（java）**

一个用户线程运行在一个内核线程中

**优点：**

能够利用多核充分的利用了cpu的算力

**缺点：**

上下文切换慢

**N:1(python)**

N个用户线程运行在一个内核线程中，

**优点：**

上下文切换快，在用户态即可完成切换

**缺点：**

无法充分利用CPU多核的算力

**M:N**

**优点：**

充分利用算力且上下文切换快

**缺点**

调度算法复杂

## GMP模型和GM模型

**调度器要求：**

能在适当的时机将合适的协程分配到合适的位置，保证公平和效率

**GMP模型**

**G（Goroutine）:**

**概念：**协程**，**存储了 Goroutine 的执行栈信息、Goroutine 状态以 及 Goroutine 的任务函数。

**数量**：数量受内存限制。

**创建时机：**由go 关键字创建

**M（Machine）：**

**概念：**Go对操作系统线程（OS thread）的封装，可以看作操作系统内 核线程

**数量：**在默认情况下，调度器最多可以创建 10000 个线程，但是其中 大多数的线程都不会执行用户代码（可能陷入系统调用），最多只会 有 GOMAXPROCS 个活跃线程能够正常运行。

运行时会将 GOMAXPROCS 设置成当前机器的核数，我们也可以在程序 中使用 [runtime.GOMAXPROCS](https://draveness.me/golang/tree/runtime.GOMAXPROCS) 来改变最大的活跃线程数

**创建时机：**没有足够的M来关联P并运行其中的可运行的G。比如所有的M此时都阻塞住了，而P中还有很多就绪任务，就会去寻找空闲的M，而没有空闲的，就会去创建新的M。

**P（Processor）：**

**概念：**虚拟处理器，M执行G所需要的**资源和上下文**，只有将P和M绑定，才能让的runq中的G真正运行起来。

**数量：**P的数量受CPU核数影响，通过环境变量$GOMAXPROCS来设置。P 的数量决定了系统内最大可并行的 G 的数量，

**创建时机：**在确定了P的最大数量n后，运行时系统会根据这个数量创建个P。

一般默认**GOMAXPROCS设置为CPU核数**，能充分利用CPU

设置的大一些，在io密集型应用中，可能性能会提升，Go调度器检测到M被阻塞是有一定延迟的，也即旧的M被阻塞和新的M得到运行之间是有一定间隔的

也有可能Go调度器不停地进行OS线程切换，性能下降

IO密集型：文件读写，网络请求等（需要切换）

CPU密集型：大量的计算，消耗cpu资源

**GM模型**

**缺点**

**1锁竞争：**

全局队列的锁竞争，创建、销毁、调度G都M需要获取队列锁，导致激烈的锁竞争

**2M转移G会造成延迟和额外的系统负载**：

当G中包含创建新协程的时候，M1创建了G2，为了继续执行G1，需要把G2交给M2执行，也造成了很差的局部性，因为G1和G2是相关的，最好放在M上执行，而不是其他M'

**3线程使用效率不能最大化（复用线程）**：

没有work-stealing 和hand-off 机制

**解决办法：**

计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决，所以加入p层

**work stealing机制**

当本线程无可运行的G时，尝试从其他线程绑定的P偷取G，而不是销毁线程。避免了G创建销毁的性能损耗

**hand off机制**

如果 M在执行 G 的过程发生系统调用阻塞（同步），会阻塞G和M（操作系统限制），此时P会和当前M解绑，并寻找新的M，如果没有空闲的M就会新建一个M ，接管正在阻塞G所属的P，接着继续执行 P中其余的G，这种阻塞后释放P的方式称之为handoff

## 协程调度原理

设计思想：

线程复用（work stealing 机制和hand off 机制）

利用并行（利用多核CPU）

抢占调度（解决公平性问题）

## 协程调度对象来源

**G的来源：**

P的runnext（只有1个G，局部性原理，永远会被最先调度执行）

P的本地队列（数组，最多256个G）

全局G队列（链表，无限制）

网络轮询器network poller（存放网络调用被阻塞的G）

**M的来源：**

休眠线程队列（未绑定P，长时间休眠会等待GC回收销毁）

运行线程（绑定P，指向P中的G）

自旋线程（绑定P，指向M的G0）

**P的来源：**

全局P队列（数组，GOMAXPROCS个P）

## 协程调度流程

**1创建G：**

通过go关键字来创建一个goroutine；

**2保存G：**

新创建的G会先保存在P的本地队列中，如果P的本地队列已经满 了就会保存在全局的队列中。

**3唤醒或者新建M：**

m执行任务，进入调度循环**（步骤4,5,6)**

**4 M获取G:**

M会从P的本地队列获取G来执行，如果P的本地队列为空，

则从全局队列获取 G，如果全局队列也为空，

则从另一个本地队列偷取一半数量的 G（负载均衡），

这种从其它P偷的方式称之为 **work stealing**

**5 M调度和执行G:**

M调用 G.func() 函数执行G：

**1**

如果M在执行 G 的过程发生系统调用阻塞（同步），

会阻塞G和M（操作系统限制），

P会和当前M解绑，并寻找新的M，

如果没有空闲的M就会新建一个M ，接管正在阻塞G所属的P， 接继续执行 P中其余的G，这种阻塞后释放P的方式称之为 **hand off.**

当M系统调用结束时候

G会尝试获取一个空闲的P执行，放入到这个P的本地队列。

G如果获取不到P，M变成休眠状态，加入到空闲线程中，

G会被放入全局队列中

**2**

如果M在执行G的过程发生网络IO等操作阻塞时（异步）， 阻塞G，不会阻塞M

M会寻找P中其它可执行的G继续执行，

G会被网络轮询器network poller 接手，

当阻塞的G恢复后

G1从network poller被移回到P的LRQ （本地运行队列）中

重新进入可执行状态

异步情况下，通过调度，Go scheduler 成功地将 I/O 的任务转变成了 CPU 任务，

或者说将内核级别的线程切换转变成了用户级别的 goroutine 切换，大大提高了效率。

**6 清理现场:**

M执行完G后清理现场重新进入调度循环（将M上运⾏的goroutine切换为G0，G0负责调度时协程的切换）

## 协程调度时机

**主动调度：**

**新起一个协程和协程执行完毕：**

触发调度循环

**垃圾回收之后：**

由于进行GC 的goroutine也需要在M上运行，因此肯定会发生调 度。

GC 不管栈上的内存，只会回收堆上的内存，栈上的内存由系统管 理。

**被动调度：**

**系统调用阻塞（比如文件IO）（同步）：**

阻塞G和M，P与M分离，

将P交给其它M绑定，

其它M执行P的剩余G

**网络IO调用阻塞（异步）：**

阻塞G，

G移动到网络轮询器NetPoller，

M执行P的剩余G

**抢占式调度：**

检测到协程运行过久（比如sleep，死循环）

切换到g0，进入调度循环

## 协程调度策略

**1队列轮转**

除了每个P维护的G本地队列以外，还有一个全局的队列，每个P会周期性的查看全局队列中是否有G待运行并将期调度到M中执行

为了防止全局队列中的G被饿死。

**2工作量窃取**

多个P中维护的G队列有可能是不均衡的，如果不加以控制，则可能出现部分处理器P非常繁忙，部分处理器P怠工的情况。

因此go调度器提供了工作量窃取策略，即当某个处理器P没有需要调度的协程时，且全局队列中没有协程时，将从其他处理器中偷取协程。

**3抢占式调度**

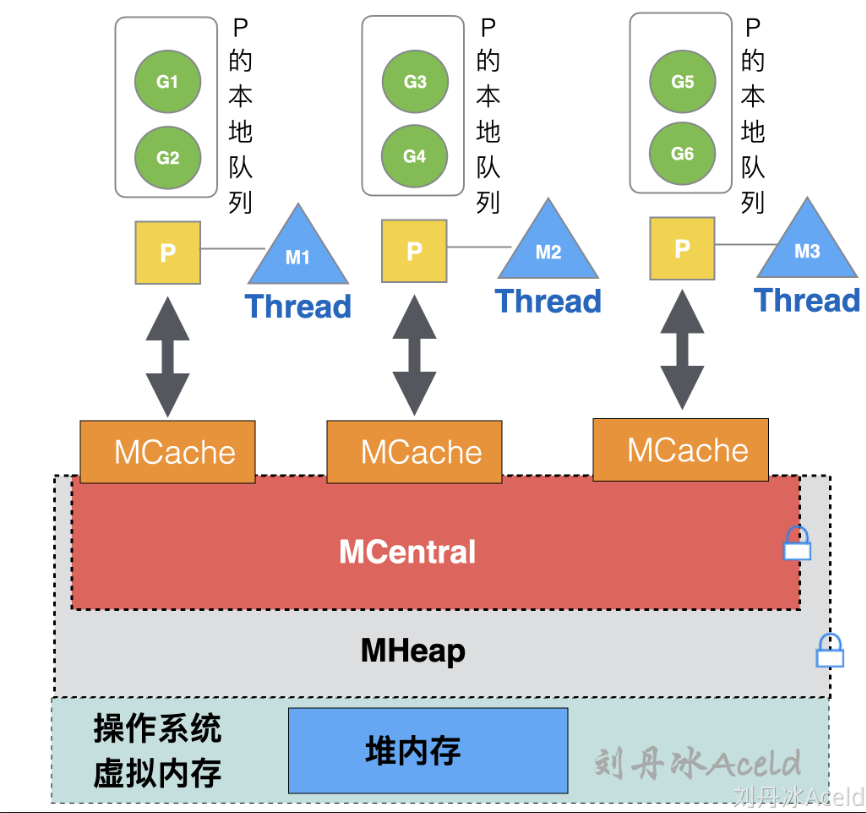
避免某个协程长时间执行，而阻碍其他协程被调度的机制。

调度器会监控每个协程的执行时间，一旦执行时间太长且有其他协程在等待时，会把协程暂定，转而调度等待的协程，已达到类似于时间片轮转的效果。

在go1.14之前，go协程调度器抢占式调度机制有一定的局限性，在改设计中，在函数调用间隙检查协程是否可被抢占，如果协程没有函数调用，则会无限期的占用执行权。

在go1.14之后，引入了基于信号的抢占机制，这个问题被解决。

## ██内存管理



## 内存管理

**1性能：**

空间换时间，同一管理内存会提前分配一大块内存，进而减少与操作系统沟通所造成的开销，进而提高程序运行的性能

**2垃圾回收：**

更好的支持垃圾回收，回收对象内存时，并没有将其真正释放掉，只是放回预先分配的大块内存中，以便复用。只有内存闲置过多的时候，才会尝试归还部分内存给操作系统，降低整体开销

## 内存逃逸

**背景：**

**堆**

分配速度慢，且会形成内存碎片

适合不可预知的内存大小分配

**栈**

分配速度快

内存大小固定

**区别：**

性能上，栈内存的使用和回收更迅速一些；尽管Golang 的 GC 很高效，但也不可避免的会带来一些性能损耗。因此，Go 优先使用栈内存进行内存分配。在不得不将对象分配到堆上时，才将特定的对象放到堆中。

**概念**

在一段程序中，每一个函数都会有自己的内存区域存放自己的局部变量、返回地址等，

这些内存会由**编译器在栈**中进行分配，每一个函数都会分配一个栈桢，在函数运行结束后进行销毁（在编译阶段完成）

但是有些变量我们想在函数运行结束后仍然使用它，那么就需要把这个变量在**堆**上分配，

这种从"栈"上逃逸到"堆"上的现象就成为内存逃逸

**目的：**

逃逸分析目的是决定内分配地址是栈还是堆

**机制：**

如果函数外部没有引用，则优先放到栈中；

如果函数外部存在引用，则必定放到堆中;

如果栈上放不下，则必定放到堆上;

## 逃逸分析作用

**能够做到真正的按需分配：**

如果这块内存退出函数没有使用，就直接分配到栈上

如果在函数之外还有引用，就分配到堆上

**减小垃圾回收的压力：**

如果全部分配在堆上，堆上的内存会不想栈上自动清理，就会造成go频繁的垃 圾回收，垃圾回收会占用比较多的系统开销。

## 逃逸场景

**1指针逃逸：**

Go可以返回局部变量指针

**2栈空间不足逃逸：**

是否逃逸取决于栈空间是否足够大

Slice()函数中分配了一个1000个长度的切片，此处并没有发生逃逸

Slice()函数中分配了一个10000个长度的切片，就会逃逸

**传指针不一定比传值效率高：**

我们知道传递指针可以减少底层值的拷贝，可以提高效率，但是如果拷贝的数据量小，由于指针传递会产生逃逸，可能会使用堆，也可能会增加GC的负担，所以传递指针不一定是高效的。

小变量还是使用传值，大变量传指针。

## 内存对齐

**概念：**

内存对齐，是指内存地址是所存储数据大小（按字节为单位）的整数倍，以便CPU可以一次将该数据从内存中读取出来。

编译器通过在结构体的各个字段之间填充一些空白已达到对齐的目的。

**对齐系数：**

32位系统对齐系数是4，64位系统对齐系数是8

**优点：**

1、提高可移植性;

不是所有的硬件平台都能访问任意地址上的任意数据的；

2、提高内存访问效率：

CPU一次可以读取1个字长的数据到内存中（64位为8个字节 的数据）

如果所需要读取的数据正好跨了1个字长，那就得花两个CPU 周期的时间去读取了。因此在内存中存放数据时进行对齐，可以提高内 存访问效率

**缺点：**

存在内存空间的浪费，实际上是空间换时间

**规则：**

1数据成员对齐规则：第一个数据成员放在offset为0的地方，以后每 个数据成员存储的起始位置要从该成员大小的整数倍开始

2结构体成员对齐规则:如果一个结构里有某些结构体成员,则结构体成员 要从其内部最大元素大小的整数倍地址开始存储

3收尾工作：结构体的总大小,也就是sizeof的结果,必须是其内部最大成 员的整数倍或者是编译器默认对齐长度（#pragma pack（n）32位为4,64 位为8）的整数倍，取最大数的最小整数倍作为对齐值

**需要两次的原因：**

内存并不是像我们逻辑上认为的那样，一个挨着一个，如果是这样的话，那就可以访问任意地址，并且把它输出到总线，但是为了实现更高的访问效率

典型的内存布局是这样的，一个内存条的一面是个rank，rank一片存储单元为一个chip。

一个chip包括8个bank。到bank这里就可以通过选择行和列来定位地址。这就不是我们逻辑上认为的存在，他们公用同一个地址，各自选择同一个位置的一个字节，在组合起来作为我们逻辑上认为的连续8个字节 ，通过这样的并行操作， 提高了内存的访问效率。但是如果使用这种设计话，地址一定是8的倍数，如果不是的话，那就不能在一次操作中被同一个地址选中。这个时候你就要操作两次。分成两组，分两次取，最后组合在一起。

## 内存分配

设计思想

内存分配算法采用Google的TCMalloc算法

**内存池：避免锁竞争**

每个线程都会自行维护一个独立的内存池，进行内存分配时优先从该内存池中分配，当内存池不足时才会向加锁向全局内存池申请，减少系统调用并且避免不同线程对全局内存池的锁竞争

**粒度小：多级管理，按需分配**

把内存切分的非常的细小，分为多级管理，以降低锁的粒度

**没有真正释放：**

回收对象内存时，并没有将其真正释放掉，只是放回预先分配的大块内存中，以便复用。只有内存闲置过多的时候，才会尝试归还部分内存给操作系统，降低整体开销

## 内存分配流程

**对象大小：**

(0, 16B) 且不包含指针的对象： Tiny分配（tiny对象）

(0, 16B) 包含指针的对象：正常分配（小对象）

[16B, 32KB] : 正常分配（小对象）

(32KB, -) : 大对象分配（大对象）

**tiny对象分配流程：**

**大小判断，指针判断，放入tiny区**

（1）P向**MCache**申请微小对象（如一个Bool变量）。

如果申请的Object在Tiny对象的大小范围则进入Tiny对象申请流程，否则进入小对象或大对象申请流程。

（2）判断申请的Tiny对象是否包含指针，

如果包含则进入小对象申请流程（不会放在Tiny缓冲区，因为需要GC走扫描等流程）。

（3）如果**Tiny空间**的16B没有多余的存储容量，则从Size Class = 2（即Span Class = 4或5）的Span中获取一个16B的Object放置Tiny缓冲区。

（4）将1B的Bool类型放置在16B的Tiny空间中，以字节对齐的方式。

**小对象分配流程：**

**Mcache，mcentral，mheap**

（1）获取当前线程的私有缓存mcache，跟据size计算出适合的**class**的ID（**Mcache不加锁，其他的需要加锁**）

（2）从mcache的alloc[class]链表中查询可用的span

（3）如果mcache没有可用的**span**则从mcentral申请一个新的span加入mcache中

（4）如果mcentral中也没有可用的**span**则从mheap中申请一个新的span加入mcentral

（5）从该span中获取到空**闲对象地**址并返回

**大对象分配流程：**

**Mheap，arenas，虚拟内存**

（1）申请如果超过32KB，则直接绕过MCache和MCentral直接向MHeap申请。

（2）MHeap根据对象所需的空间计算得到需要多少个**Page**。将会根据需求进行切分，以返回用户所需**的页数**。剩余的页构成一个新的**span**放回 mheap 的空闲列表。

（3）如果Mheap中没有内存，则向Arenas中的HeapArena申请相对应的**Pages**。

（4）如果Arenas中没有HeapA可提供合适的Pages内存，则向操作系统的**虚拟内存**申请，且填充至Arenas中。

（5）MHeap返回大对象的内存空间。

（6）**协程逻辑层**P得到内存，流程结束。

## tiny对象

**概念：**

MCache中不仅保存着各个Span Class级别的内存块空间，还有一个比较特殊的Tiny存储空间，

Tiny空间是从Size Class = 2（对应Span Class = 4 或5）中获取一个16B的Object，作为Tiny对象的分配空间

**为什么要有tiny对象：内存浪费**

如果协程逻辑层申请的内存空间小于等于8B，如int32（4b）、 bool(1b）等微小对象

根据正常的Size Class匹配会匹配到Size Class = 1（对应Span Class = 2或3）也就是8B

int32（4b）、 bool(1b）各自独享这8B的空间，进而导致有一定的内存空间浪费。

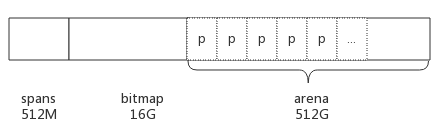
**解决办法：**

所以Golang内存管理决定尽量不使用Size Class = 1的Span，而是将申请的Object小于16B（不包含指针）的申请统一归类为Tiny对象申请。

## 内存组件/粒度

**内存划分：**

Golang程序启动时会向系统申请的内存如下图所示：



**三部分：**

预申请的内存划分为spans、bitmap、arena。

**Arena：**

即为所谓的堆区，应用中需要的内存从这里分配。arena的大小为512G，为了方便管理把arena区域划分成一个个的page，每个page为8KB,一共有512GB/8KB个页；

**Spans：**

spans区域存放span的指针，每个指针对应一个page，所以span区域的大小为(512GB/8KB)\*指针大小8byte = 512M

**bitmap：**

bitmap区域大小也是通过arena计算出来，不过主要用于GC。

**1）page**

大小：一个Page的大小依然是8KB。

最小单元：Page表示内存管理与虚拟内存交互内存的最小单元。

数量：一共有512GB/8KB个页

**2）span**

定义：多个连续的Page称之为是一个Span，

allocBit：allocBits指向一个位图，每位代表一个块是否被分配，本例中有两个块已经被分配，其allocCount也为2

**3）size class**

表示一块内存的所属规格或者刻度

**4）Span Class**

定义：Golang内存管理额外定义的规格属性，是针对Span来进行划分的，是Span大小的级别。

一个Size Class会对应两个Span Class，

其中一个Span为存放需要GC扫描的对象（包含指针的对象），

另一个Span为存放不需要GC扫描的对象（不包含指针的对象）

**5）mcache（线程缓存）**

定义：MCache是与GPM中的P所绑定，而不是和线程绑定

数量：数量和P的数量一致，即GOMAXPROCS个，

不加锁：MCache与P进行绑定，不需要加锁就可以获取到内存

**6）mcentarl（中心缓存）**

**定义：**当MCache中某个Size Class对应的Span被一次次Object被上层取走后，如果出现当前Size Class的Span空缺情况，MCache则会向MCentral申请对应的Span。

加锁：向MCentral申请Span是同样是需要加锁的。

**7）mheap**

定义： MHeap的内存不够时会向操作系统的虚拟内存空间申请。访

加锁：向MHeap获取内存依然是需要加锁的。

## ██垃圾回收

**概念：**

程序申请的内存不在需要的时候，将其回收至操作系统，以便供其他代码进行内存申请。

**类型：**

在应用程序中会使用到两种内存，分别为堆（Heap）和栈（Stack），GC负责回收堆内存，而不负责回收栈中的内存：

## 根对象

**是垃圾回收器在标记过程时最先检查的对象，**包括：

**全局变量：**

程序在编译期就能确定那些存在于程序整个生命周期的变量

**执行栈：**

每个goroutine都包含自己的执行栈，这些执行栈上包含栈上的变量及指向分配的堆内存区块的指针。

**寄存器：**

寄存器的值可能表示一个指针，参与计算的这些指针可能指向某些赋值器分配的堆内存块

## 常用垃圾回收算法

**引用计数：**

对每个对象维护一个引用计数，当引用该对象的对象被销毁时，引用计数减1，当引用计数器为0是回收该对象。

**优点：**

对象可以很快的被回收，不会出现内存耗尽或达到某个阀值时才回收。

**缺点：**

不能很好的处理循环引用，而且实时维护引用计数，有也一定的代价。

**代表语言**：

Python、PHP

**标记-清除：**

从根变量开始遍历所有引用的对象，引用的对象标记为”被引用”，没有被标记的进行回收。

**优点：**

解决了引用计数的缺点。

**缺点：**

需要STW，即要暂时停掉程序运行。

**代表语言**：

Golang(其采用三色标记法)

## STW

STW 可以是 Stop the World 的缩写，也可以是 Start the World 的缩写。

指代从 Stop the World 这一动作发生时到 Start the World 这一动作发生时这一段时间间隔，即万物静止。

## v1.3之前的标记清除法

**流程**

**第一步：**

**暂停**程序业务逻辑, 分类出**可达**和**不可达**的对象，然后做上标记

**第二步：**

开始标记，程序找出它**所有可达的对象**，并做**上标记**

**第三步:**

:标记完了之后，然后开始**清除未标记的对象**

**第四步:**

**停止暂停**，让程序继续跑。

**优点:**

**简单**

**缺点:**

需要STW，让程序暂停，程序出现卡顿 (重要问题)，效率极低

## v1.5的三色标记法

**流程**

**第一步：（新建白色）**

每次新创建的对象，默认的颜色都是标记为“白色”

**第二步：（根对象为灰色）**

每次GC回收开始, 会从根节点开始遍历所有对象，把遍历到的对象从白色集合放入“灰色”集合（非递归形式，是从程序抽次可抵达的对象遍历一层）

**第三步：（根对象黑色，引用的为灰色）**

遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合

**第四步：（重复）**

重复第三步, 直到灰色中无任何对象

**第五步: （回收）**

回收所有的白色标记表的对象. 也就是回收垃圾

**缺点：**

还是需要**STW**，程序出现卡顿 (重要问题)

## 无STW的三色标记法

**问题：对象丢失**

**条件1:** 一个白色对象被黑色对象引用(白色被挂在黑色下)

**条件2:** 灰色对象与它之间的可达关系的白色对象遭到破坏(灰色同时丢了该白色)

如果当以上两个条件同时满足时，就会出现对象丢失现象!

**解决办法：**

**方法1：**

STW，但是STW的过程有明显的资源浪费，对所有的用户程序都有很大影响

**方法2：**

屏障机制，尝试去破坏上面的两个必要条件就可以了

## 屏障机制（v1.8）

**概念**  
 额外的对对象进行判断

对象被引用时触发的机制是插入屏障

对象被删除时触发的机制是删除屏障

**原理**

强弱三色不变式，满足“强三色不变式”和“弱三色不变式”之一时，即可保对象不丢失。

**强三色不变式**

不存在黑色对象引用到白色对象的指针（插入屏障）

**弱三色不变式**

所有被黑色对象引用的白色对象都处于灰色保护状态（删除屏障）

**1插入屏障**

对象被引用时触发的机制

**具体操作:**

在A对象引用B对象的时候，B对象被标记为灰色。(将B挂在A下游，B必须被标记为灰色)

**满足:**

强三色不变式. (不存在黑色对象引用白色对象的情况了， 因为白色会强制变成灰色)

**优点：**

相比无没有屏障机制的纯stw的垃圾回收，插入写屏障只需要在结束的时候stw对栈进行扫描。而不是stw对栈和堆进行扫描，相比之下，性能有提升。

**缺点：**

结束时需要STW来重新扫描栈，还是存在性能上的问题

插入屏障机制只在堆空间使用：

栈空间的特点是容量小,但是要求相应速度快,因为函数调用弹出频繁使用, 所以“插入屏障”机制,在栈空间的对象操作中不使用（如果在栈上使用，会影响栈的运行效率）

但是如果栈不添加,当全部三色标记扫描之后,栈上有可能依然存在白色对象被引用的情况(.所以要对栈重新进行三色标记扫描, 但这次为了对象不丢失, 要对本次标记扫描启动STW暂停. 直到栈空间的三色标记结束.

结束时需要STW来重新扫描栈，还是存在性能上的问题

**删除写屏障：**

对象被删除时触发的机制

**具体操作:**

被删除的对象，如果自身为灰色或者白色，那么被标记为灰色。

**满足:**

弱三色不变式. (保护灰色对象到白色对象的路径不会断)

**缺点：**

**1**回收精度低：一个对象即使被删除了最后一个指向它的指针也依旧可以活过这一轮，在下一轮GC中被清理掉

**2**需要STW：删除写屏障也叫基于快照的写屏障方案，必须在起始时，STW 扫描整个栈，保证所有堆上在用的对象都处于灰色保护下，保证的是弱三色不变式；

删除写屏障不适用于栈特别大的场景，栈越大，STW 扫描时间越长，对于现代服务器上的程序来说，栈地址空间都很大，所以删除写屏障都不适用，一般适用于很小的栈内存，比如嵌入式，物联网的一些程序；

**混合写屏障：**

**插入写屏障和删除写屏障的缺点：**

插入写屏障：结束时需要STW来重新扫描栈，标记栈上引用的白色对象的存活；

删除写屏障：回收精度低，GC开始时STW扫描堆栈来记录初始快照，这个过程会保护开始时刻的所有存活对象。

**具体操作:**

1、GC开始将栈上的对象全部扫描并标记为黑色

2、GC期间，任何在栈上创建的新对象，均为黑色。

3、被删除的对象标记为灰色。

4、被添加的对象标记为灰色。

**满足:**

弱三色不变式. （栈不启用屏障，堆启动屏障，因为第一次将栈上的对象全部扫描并标记为黑，）

**优点：**

1无需快照：起始无需 STW 打快照，直接并发扫描垃圾即可；

2无需结尾stw：赋值器是黑色赋值器，扫描过一次就不需要扫描了，这样就消除了插入写屏障时期最后 STW 的重新扫描栈；

3精度：混合写屏障扫描精度继承了删除写屏障，比插入写屏障更低，随着带来的是 GC 过程全程无 STW；

**注意：**

混合写屏障扫描栈虽然没有 STW，但是扫描某一个具体的栈的时候，还是要停止这个 goroutine 赋值器的工作的哈（针对一个 goroutine 栈来说，是暂停扫的，要么全灰，要么全黑哈，原子状态切换）

**总结**

GoV1.3- 普通标记清除法，整体过程需要启动STW，效率极低。

GoV1.5- 三色标记法， 堆空间启动写屏障，栈空间不启动，全部扫描之后，需要重新扫描一次栈(需要STW)，效率普通

GoV1.8-三色标记法，混合写屏障机制， 栈空间不启动，堆空间启动。整个过程几乎不需要STW（在标记准备每个栈单独暂停），效率较高

## 触发 GC 的时机

**主动触发：**

通过调用 runtime.GC 来触发 GC，此调用阻塞式地等待当前 GC 运行完毕。

**被动触发：系统监控和步调算法**

**系统监控：**

当超过两分钟没有产生任何 GC 时，强制触发 GC。

**步调（Pacing）算法：**

其核心思想是控制内存增长的比例。每次内存分配的时候都会检查当前内存分配量是否已达到阈值，如果达到阈值则立即启动GC。（阈值=上次GC内存分配量\*内存增长率）（内存增长率由环境变量控制，默认为100）

## GC调优

**GC调优的场景：**

**对停顿敏感：**GC 过程中产生的长时间停顿、或由于需要执行 GC 而没有执行用户代码，导致需要立即执行的用户代码执行滞后。

**对资源消耗敏感：**对于频繁分配内存的应用而言，频繁分配内存增加 GC 的工作量，原本可以充分利用 CPU 的应用不得不频繁地执行垃圾回收，影响用户代码对 CPU 的利用率，进而影响用户代码的执行效率。

**GC调优的方式：**

**1控制内存分配的速度：**

限制 goroutine 的数量，从而提高赋值器对 CPU 的利用率。

在两个嵌套的 for 循环中一口气创建了 800 个 goroutine。在 main 函数中，启动了一个 goroutine 并在程序结束前不断的触发 GC，并尝试输出 GC 的平均执行时间

goroutine 的执行时间占其生命周期总时间非常短的一部分，但大部分时间都花费在调度器的等待上了（蓝色的部分）

同时创建大量 goroutine 对调度器产生的压力确实不小，我们不妨将这一产生速率减慢，一批一批地创建 goroutine：

**2复用内存：**

例如使用 sync.Pool 来复用需要频繁创建临时对象，使用全局变量等

**3增大GOGC的值：**

降低 GC 的运行频率。

GC 的触发原则是由步调算法来控制的，其关键在于估计下一次需要触发 GC 时，堆的大小。可想而知，如果我们在遇到海量请求的时，为了避免 GC 频繁触发，是否可以通过将 GOGC 的值设置得更大，让 GC 触发的时间变得更晚，从而减少其触发频率。

**4减小小对象的分配：**

通常小对象过多会导致GC三色法消耗过多的GPU。优化思路是，减少对象分配，

## 查看GC信息

1 GODEBUG='gctrace=1'

2 go tool trace

## 小对象多了gc有压力

小于等于32k的对象就是小对象，其它都是大对象。一般小对象通过 mspan 分配内存；大对象则直接由 mheap 分配内存。

通常小对象过多会导致GC三色法消耗过多的CPU，

## ██并发控制

**并发模型**

并发模型说的是系统中的线程如何协作完成并发任务

不同的并发模型，线程以**不同的方式进行通信和协作**。

**线程间通信方式：**

两种：共享内存和消息传递

**共享内存：**

**抽象层级：**

抽象层级低，当我们遇到对资源进行更细粒度的控制或者对性能有 极高要求的场景才应该考虑抽象层级更低的方法

**耦合：**

高，线程需要在读取或者写入数据时先获取保护该资源的互斥锁

**线程竞争：**

需要加锁，才能避免线程竞争和数据冲突

**消息传递：**

抽象层级：

抽象层级高，提供了更良好的封装和与领域更相关和契合的设计，比如Go 语言中的Channel就提供了 Goroutine 之间用于传递信息的方式，它在内部实现时就广泛用到了共享内存和锁，通过对两者进行的组合提供了更高级的同步机制

耦合：

低，生产消费者模型

线程竞争：

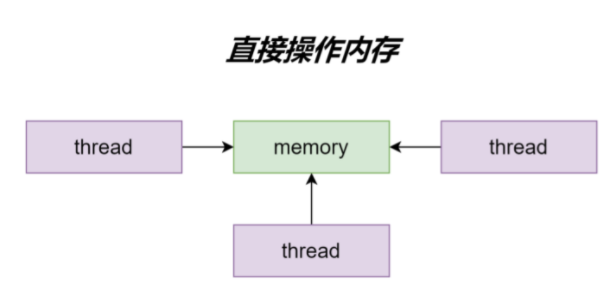
保证同一时间只有一个活跃的线程能够访问数据，channel维护所有被该chanel阻塞的协程，保证有资源的时候只唤醒一个协程，从而避免竞争

**并发模型种类：**

两种：共享内存并发模型，CSP模型。

**共享内存模型：**

通过直接共享内存 + 锁的方式同步信息，传统多线程并发



**csp模型：communicating sequential processes**

通过发送消息的方式来同步信息，

Go语言推荐使用的通信顺序进程（communicating sequential processes）并发模型，通过goroutine和channel来实现

**goroutine** 是Go语言中并发的执行单位，可以理解为”线程“

**channel**是Go语言中各个并发结构体(goroutine)之前的通信机制。

## 并发控制顺序方法

**场景：**

协程A执行过程中需要创建子协程A1、A2、A3…An，

协程A创建完子协程后就等待子协程退出。

**解决方案三种：**

Channel：使用channel控制子协程

WaitGroup：使用信号量机制控制子协程

Context：使用上下文控制子协程

**1Channel：**

**优点：**

实现简单

**缺点：**

当需要大量创建协程时就需要有相同数量的channel，而且对于子协程继续派生出来的协程不方便控制。

**2WaitGroup：**

**优点：**

子协程个数可动态调整

**缺点：**

对派生的协程不方便控制

**3Context：**

**优点：**

对派生的goroutine有更强力的控制力，可以控制多级的goroutine

**缺点：**

context可以携带值，但是没有任何限制，类型和大小都没有限制，也就是没有任何约束，这样很容易导致滥用，程序的健壮很难保证；

可以自定义context，这样风险不可控，更加会导致滥用。

## Channel

channel一般用于协程之间的通信，channel也可以用于并发控制。比如主协程启动N个子协程，主协程等待所有子协程退出后再继续后续流程，这种场景下channel也可轻易实现。

## Waitgroup

**概念：**

Wait-Goroutine-Group，即等待一组goroutine结束。

比如某个goroutine需要等待其他几个goroutine全部完成，那么使用WaitGroup可以轻松实现

**原理：**

维护了 2 个计数器：请求计数器 v，等待计数器 w，

每次 Add 执行，请求计数器 v 加 1，

Done 方法执行，等待计数器减 1，

v 为0 时通过信号量唤醒 Wait()。

**信号量：**

信号量是Unix系统提供的一种保护共享资源的机制，用于防止多个线程同时访问某个资源。

可简单理解为信号量为一个数值：

当信号量>0时，表示资源可用，获取信号量时系统自动将信号量减1；

当信号量==0时，表示资源暂不可用，

获取信号量时，当前线程会进入睡眠，当信号量为正时被唤醒；

**数据结构：**

state1是个长度为3的数组：包含了state和一个信号量，而state实际上是两个计数器：

counter： 当前还未执行结束的goroutine计数器

waiter count: 等待goroutine-group结束的goroutine数量，即有多少个等候者

semaphore: 信号量

**WaitGroup对外提供三个接口：**

Add(delta int):

第一：将delta值加到counter中

第二是当counter值变为0时，跟据waiter数值释放等量的信 号量，把等待的goroutine全部唤醒，如果counter变为负值， 则panic.

Wait()：

第一：waiter递增1，

第二：阻塞等待信号量semaphore

Done()：

第一：counter递减1，按照waiter数值释放相应次数信号量

**注意：**

Add()操作必须早于Wait(), 否则会panic

Add()设置的值必须与实际等待的goroutine个数一致，否则会panic

**WaitGroup用法：**

1调用Add()设置协程个数：

main 协程通过调用 wg.Add(delta int) 设置 worker 协程的个数

2创建

创建worker协程；

3调用Done()worker

协程执行结束以后，都要调用 wg.Done()；

3调用Wait()阻塞

main协程调用 wg.Wait()且被阻塞，直到所有 worker 协程全部执行结束后返回。

## Context

**作用：**

context 主要用来在 goroutine 之间传递上下文信息，

包括：取消信号、超时时间、截止时间、传值等

**原理：**

Go里并没有直接为我们提供一个统一的context对象，而是设计了一个接口类型的 Context。然后在这些接口上来实现了几种具体类型的 context

**context 接口定义四个方法：**

**1** Deadline() (deadline time.Time, ok bool)

**2** Done() <-chan struct{}

**3** Err() error

**4** Value(key interface{}) interface{}

**1：Deadline() ：**

表示如果有截止时间的话，得返回对应 deadline 时间；

如果没有，则 ok 的值为 false。

**2：Done()**

表示关于 channel 的数据通信，而且它的数据类型是 struct{}，一个空结构体，

因此在 Go 里都是直接通过 close channel 来进行通知的，不会涉及具体数据传输。

**3：Err()**

返回的是一个错误 error，

如果上面的 Done() 的channel 没被 close，则 error 为 nil；

如果channel已被close，则error将会返回close的原因，比如超时或手动取消。

**4：Value()**

则是用来存储具体数据的方法。

**context类型**

**1emptyCtx，2cancelCtx，3timerCtx，4valueCtx**

类型的创建方法如下：

Background()和TODO()方法创建emptyCtx。

WithCancel 方法创建的是 cancelCtx 类型的 context。

WithDeadline 方法创建的是 timerCtx 类型的 context。

WithValue 方法创建的是 valueCtx 类型的 context。

context.Background()返回的就是 emptyCtx。

上面三个方法在创建的时候都会要求传 parent context 进来，以此达到链式传递信息的目的

**1emptyCtx：**

emptyCtx 表示什么都没有的 context，一般用作最初始的 context，作为父 context 使用

是一个空的context，永远不会被 cancel，没有存储值，也没有 deadline

**创建方式：**

Background()和TODO()方法

background 通常用在 main 函数中，作为所有 context 的根节点。

todo 通常用在并不知道传递什么 context的情形。

**2cancelCtx：**

cancelCtx 的核心是可以主动取消的上下文，他取消的时候还会将所有由该上下文派生的的子上下文一并取消，

主要是通过mu，done，children，err实现，**WithCancel方法创建**

**mu：**

锁，用于保护并发 ，首先根据 cancelCtx 的核心需求，可以取消派生的所有上下文，也就意味着我们需要存储这个 context 派生的所有子 context，那我们推测这个锁的作用应该就是保护存储子上下文或者删除子上下文的结构体。（总结会含有真正作用）

**done：**

推测用于标识此结构体是否结束。

**children：**

是一个 canceler 的 map，可以发现就是用于存储上下文的结构题

err:错误信息，用于判断是否已经取消

**children的作用：**

因为 cancelContext 是父子相关联的，一个 cancelContext 取消的时候既需要干掉自己所有的子，也需要告诉自己的父。

**Mu的作用：**

context 被多个协程互相传递使用，这就要保证它一定要是并发安全的，实现过程中各种修改操作，如取消，删除子，增加子，都需要用锁保证并发安全。

**3timerCtx：**

timerCtx的核心是在到达指定时间后自动 cancel，所以相对于 cancelCtx 它只新增了两个结构体，WithDeadline方法创建

timer：计时器

deadline：截止时间

**cancelCtx 和timerCtx区别：**

**相同点**

cancelCtx 、timerCtx(用来通知用的context)

**不同点**

cancelCtx是手动调用cancel方法来触发取消通知；

timerCtxt则通过AfterFunc超时时间来自动触发cancel方法。

**4valueCtx：**

用来传值的 context，WithValue 方法创建

valueCtx 通过 key-value 形式来存储数据，当找不到 key 时，就会到 父 context 里查找，直到没有父context

和链表有点像，只是它的方向相反：Context 指向它的父节点，链表则指向下一个节点。通过 WithValue 函数，可以创建层层的 valueCtx，存储 goroutine 间可以共享的变量。

因为查找方向是往上走的，所以，父节点没法获取子节点存储的值，子节点却可以获取父节点的值。

**缺点：**

在一个处理过程中，有若干子函数、子协程。各种不同的地方会向 context 里塞入各种不同的 k-v 对，最后在某个地方使用

不清楚这些值会不会被覆盖，很难进行排查和项目的梳理

**在项目中使用：**

在jwt认证中间件中，把用户的id信息保存到上下文中c.set（key，value的形式）

然后在controller层建了一个方法就是从上下文中取出用户id，以便于在需要id信息的时候直接取出c.get（key）

## 锁

并发操作同一对象可能带来错误，需要一些机制来保证某一时刻只能有一个协程执行特定操作，通常是通过锁机制来完成

锁的作用是保证资源在使用时的独有性，不会因为并发而导致数据错乱，保证系统的稳定性。

## 互斥锁

**概念：**

互斥锁是并发程序中对共享资源进行访问控制的主要手段，

Go语言提供了非常简单易用的Mutex，Mutex为一结构体类型

**方法：**

Lock()和Unlock()分别用于加锁和解锁

**置1：**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。

**等信号量：**抢不到的话就阻塞等待Mutex.sema信号量，

**一次唤醒：**一旦持有锁的协程解锁，等待的协程会依次被唤醒。

**数据结构：sema和state**

**Mutex.sema：**

概念：表示信号量，协程阻塞等待该信号量，

作用：解锁的协程释放信号量从而唤醒等待信号量的协程。

**Mutex.state：（Locked，Woken，Starving Waiter）**

**1Locked:**

表示该Mutex是否已被锁定，0：没有锁定 1：已被锁定。

**2Woken:**

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。

**3Starving：**

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。

**4Waiter:**

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。

**加锁和解锁：**

**1简单加锁**

Locked：加锁过程会去判断Locked标志位是否为0，

置1：如果是0则把Locked位置1，代表加锁成功。

**2简单解锁**

没有协程等待：由于没有其他协程阻塞等待加锁，

置0：此时解锁时只需要把Locked位置为0即可，不需要释放信号量

**3加锁被阻塞**

加锁被阻塞：协程A加锁，当协程B对一个已被占用的锁再次加锁时，Waiter计数器增加了1，此时协程B将被阻塞，

**4解锁并唤醒协程**

协程A解锁过程分为两个步骤

lock置0检查：，一是把Locked位置0，

Waiter>0释放信号量唤醒：二是查看到Waiter>0，所以释放一个信号量，唤醒一个阻塞的协程，被唤醒的协程B把Locked位置1，于是协程B获得锁。

**自旋：**

概念：加锁时，如果当前Locked位为1，说明该锁当前由其他协程持有，尝试加锁的协程并不是马上转入阻塞，而是会持续的探测Locked位是否变为0，这个过程即为自旋过程。

时间：自旋时间很短，但如果在自旋过程中发现锁已被释放，那么协程可以立即获取锁。此时即便有协程被唤醒也无法获取锁，只能再次阻塞。

**自旋的好处：**

更充分的利用CPU，尽量避免协程切换。

当加锁失败时不必立即转入阻塞，有一定机会获取到锁，这样可以避免协程的切换

因为当前申请加锁的协程拥有CPU，如果经过短时间的自旋可以获得锁，当前协程可以继续运行，不必进入阻塞状态。

**自旋的缺点：**

如果自旋过程中获得锁，那么之前被阻塞的协程将无法获得锁，如果加锁的协程特别多，每次都通过自旋获得锁，那么之前被阻塞的进程将很难获得锁，从而进入饥饿状态。

**自旋的条件：**

1）还没自旋超过 4 次,

2）多核处理器，

3）GOMAXPROCS > 1，

4）p 上本地 goroutine 队列为空。

**解决办法：**

starving状态：为了避免协程长时间无法获取锁，自1.8版本以来增加了一个状态，即Mutex的Starving状态。这个状态下不会自旋，一旦有协程释放锁，那么一定会唤醒一个协程并成功加锁。

**Mutex 允许自旋的条件：**

锁已被占用，并且锁不处于饥饿模式。

CPU 核数大于 1。

有空闲的 P。

**Mutex模式：**

两个模式

Normal和Starving。

**normal模式：（非公平锁）**

默认情况下，Mutex的模式为normal。

该模式下，协程如果加锁不成功不会立即转入阻塞排队，而是判断是否满足自旋的条件，如果满足则会启动自旋过程，尝试抢锁。

**问题：**

如果自旋过程中获得锁，那么之前被阻塞的协程将无法获得锁，如果加锁的协程特别多，每次都通过自旋获得锁，那么之前被阻塞的进程将很难获得锁，从而进入饥饿状态。通过**starvation模式解决**

**starvation模式：（公平锁）**

为了解决阻塞的 goroutine 队列的长尾饥饿问题

饥饿模式下，新进来的 goroutine 不会参与抢锁也不会进入自旋状态，会直接进入等待队列的尾部。

这样解决了老的 goroutine 一直抢不到锁的场景。

**饥饿模式的触发条件：**

当一个 goroutine 等待锁时间超过 1 毫秒时，或者当前队列只剩下一个 goroutine 的时候， Mutex 切换到饥饿模式。

**worken位：**

Woken状态用于加锁和解锁过程的通信

举个例子，同一时刻，两个协程一个在加锁，一个在解锁，在加锁的协程可能在自旋过程中，此时把Woken标记为1，用于通知解锁协程不必释放信号量了，好比在说：你只管解锁好了，不必释放信号量，我马上就拿到锁了。

死锁

加锁后立即使用defer对其解锁，可以有效的避免死锁。

## 读写锁

**适用场景：**

在多读少写的情况下，如果长时间没有写操作的话，那么读取的到的会是完全相同的值，则不需要通过互斥锁的方式获取。一般通过读写锁获取。

只有读锁不会阻塞读锁，其他的都会阻塞

读写锁内部仍有一个互斥锁，用于将两个写操作隔离开来，其他的几个都用于隔离读操作和写操作。

**数据结构**

**W：**

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

**writerSem**

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

**readerSem**

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

**readerCount**

记录读者个数

**readerWait**

记录写阻塞时读者个数

**加锁和解锁**

RLock()：读锁定

RUnlock()：解除读锁定

Lock(): 写锁定，与Mutex完全一致

Unlock()：解除写锁定，与Mutex完全一致

**RLock()：读锁定**

增加读操作计数，即readerCount++

阻塞等待写操作结束(如果有的话)

**RUnlock()：解除读锁定**

减少读操作计数，即readerCount—

唤醒等待写操作的协程（只有最后一个读者释放锁后才会唤醒因写锁而阻塞的协程）

**Lock():**

写锁定，与Mutex完全一致

获取互斥锁（获取互斥锁是写锁定的前提，多个写锁需要排队）

阻塞等待所有读操作结束（如果有的话）

**Unlock()：**

解除写锁定，与Mutex完全一致

唤醒因写锁定而被阻塞的协程（如果有的话）

解除互斥锁

## 原子操作

原子操作就是不可中断的操作，外界是看不到原子操作的中间状态，要么看到原子操作已经完成，要么看到原子操作已经结束。

针对某个值的原子操作在被进行的过程中，CPU 绝不会再去进行其他的针对该值的操作。

**原子操作和锁的区别**

锁应当用来保护**一段逻辑**，原子操作对于**一个变量更新**的保护。

**原子操作和锁的底层原理**

原子操作由底层硬件支持，而锁则由操作系统的调度器实现。

## CAS

CAS 的全称为 Compare And Swap，直译就是比较交换。

是一条 CPU 的原子指令，其作用是让 CPU 先进行比较两个值是否相等，然后原子地更新某个位置的值

简述过程：25假设包含 3 个参数内存位置(V)、预期原值(A)和新值(B)。 V 表示要更新变量的值， E 表示预期值， N 表示新值。

仅当 V 值等于 E 值时，才会将 V 的值设为 N，如果 V 值和 E 值不同，则说明已经有其他线程在做更新，则当前线程什么都不做，最后 CAS 返回当前 V 的真实值。 CAS 操作时抱着乐观的态度进行的，它总是认为自己可以成功完成操作。基于这样的原理， CAS 操作即使没有锁，也可以发现其他线程对于当前线程的干扰。

**乐观锁**  
 乐观锁的概念其实很简单，就是在操作一个共享变量时，我们先认为多个线程之间没有冲突

**CAS**  
 CAS是乐观锁的一种实现，CAS全称是比较和替换Compare And Swap，

**CAS的操作步骤：**  
 1.先查询原始值  
 2.操作时比较原始值是否修改  
 3.如果修改，则操作失败，禁止更新操作，如果没有发生修改，则更新为新值

**CAS的缺点**：  
CAS虽然在低并发量的情况下可以减少系统的开销，但是CAS也有一些问题：  
1.CPU开销过大问题2.ABA问题3.只能针对一个共享变量

**1.CPU开销过大**  
 在我们使用CAS时，如果并发量过大，我们的程序有可能会一直自旋，长时间占用CPU资源。

**2ABA问题**  
 假设有个共享变量J，原始值为1。  
 1.线程A读取变量J，值为1  
 2.线程B读取变量J，值为1  
 3.线程A变量J+1，CAS成功从1修改为2  
 4.线程C读取变量J，值为2  
 5.线程C将变量J-1，CAS成功从2修改为1  
 6.线程A通过CAS比较和替换，依然可以改为自己想修改的值  
上述过程，线程B和C已经将变量J的值已经改变了，但是**线程A无法发现，**依然可以修改共享变量，这就产生了ABA问题。

**解决办法：**加标志位（version）。  
 至于标志位可以是自增的数字，也可以是时间戳。通过标志位我们可以 精确的知道每次修改

**3共享变量单一**  
 CAS操作单个共享变量的时候可以保证原子的操作，无法操作多个变量。

## ██反射

**概念**

反射是一种检查interface变量的底层类型和值的机制

**静态类型**

你肯定知道Go是静态类型语言，比如”int”、”float32”、”[]byte”等等。每个变量都有一个静态类型，且在编译时就确定了。

**举例：**

type Myint int

var i int

var j Myint

i 和j类型是不同的。 二者拥有不同的静态类型，没有类型转换的话是不可以互相赋值的，尽管二者底层类型是一样的。

**特殊的静态类型interface：**

interface类型是一种特殊的类型，它代表方法集合。 它可以存放任何实现了其方法的值

任何类型，比如某struct，只要实现了其中的方法就被认为是实现了接口

接口类型的变量可以存储任何实现该接口的值。

**特殊的interface类型：**

最特殊的interface类型为空interface类型，即interface {}，前面说了，interface用来表示一组方法集合，所有实现该方法集合的类型都被认为是实现了该接口。

空interface类型的方法集合为空，也就是说所有类型都可以认为是实现了该接口。

一个空interface类型变量可以存放所有值，记住是所有值，这才是最最重要的。 这也是有些人认为Go是动态类型的原因，这是个错觉。

**反射使用场景**

不能明确接口调用哪个函数，需要根据传入的参数在运行时决定。

例如：人种接口，有个说话方法，黑人结构体，黑人结构体实现说 话方法，白人结构体，白人结构体实现方法

不能明确传入函数的参数类型，需要在运行时处理任意对象。

**反射的缺点：**

1难以阅读

2性能低下：反射对性能影响还是比较大的，比正常代码运行速度慢一到两个数量级。所以，对于一个项目中处于运行效率关键位置的代码，尽量避免使用反射特性

**反射三大定律：（可以看）**

interface类型有个(value，type)对，而反射就是检查interface的这个(value, type)对的。

具体一点说就是Go提供一组方法提取interface的value，提供另一组方法提取interface的type.

reflect.Type 提供一组接口处理interface的类型，即（value, type）中的type

reflect.Value提供一组接口处理interface的值,即(value, type)中的value

**1 反射第一定律：反射可以将interface类型变量转换成反射对象**

[reflect.TypeOf](https://draveness.me/golang/tree/reflect.TypeOf) 获取了变量 author 的类型，

[reflect.ValueOf](https://draveness.me/golang/tree/reflect.ValueOf) 获取了变量的值 draven。

如果我们知道了一个变量的类型和值，那么就意味着我们知道了这个变量的全部信息。

**2 反射第二定律：反射可以将反射对象还原成interface对象**

[reflect](https://golang.org/pkg/reflect/) 中的 [reflect.Value.Interface](https://draveness.me/golang/tree/reflect.Value.Interface)将反射对象还原成接口类型

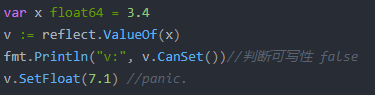
不过需要进行显示转换

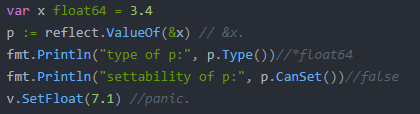
v := reflect.ValueOf(1)

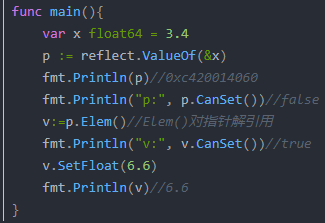
v.Interface().(int)

**3 反射第三定律：反射对象可修改，value值必须是可设置的**

**因为go是值传递，所以必须传递地址才不会出错,但是不能直接使用地址，需要用elem方法进行解引用**







## Tag原理

Go 中解析的 tag 是通过反射实现的，

反射的三大定律 反射将接口变量转换成反射对象 Type 和 Value； 反射可以通过反射对象 Value 还原成原先的接口变量； 反射可以用来修改一个变量的值，前提是这个值可以被修改；

## ██错误与异常

## defer

**使用场景**

1释放资源：关闭数据库连接,关闭文件等等

2流程控制

3异常处理

4与recover配合，消除panic，

**defer规则**

1延迟函数的参数在defer语句出现时就已经确定下来了

2延迟函数执行按**后进先出**顺序执行，即先出现的defer最后执行

**defer实现原理**

每个defer语句都对应一个defer实例，多个实例使用指针连接起来形成一个单连表，保存在 gotoutine 数据结构中，每次插入\_defer 实例，均插入到链表的头部，函数结束再一次从头部取出，从而形成后进先出的效果。

**数据结构**

type \_defer struct {

sp uintptr //函数栈指针

pc uintptr //程序计数器

fn \*funcval //函数地址

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

}

**函数封装：**每个defer实例实际上是对一个函数的封装，它拥有执行函数的必要信息比如也有栈地址、程序计数器、函数地址等等。

**指针：**与函数不同的一点是它含有一个指针，可用于指向另一个defer

**Goroutine：**每个goroutine数据结构中有一个defer指针，该指针指向一个defer的单链表

**声明：**每次声明一个defer时就将defer插入到单链表表头，

**执行：**每次执行defer时就从单链表表头取出一个defer执行。

使用链表的原因，defer数量不确定

**执行机制**

中间代码生成阶段的会负责处理程序中的 defer，该函数会根据条件的不同，使用三种不同的机制处理该关键字：

堆分配、栈分配和开放编码是处理 defer 关键字的三种方法

**1早期，堆分配，性能较差**

Go 语言会在堆上分配 [runtime.\_defer](https://draveness.me/golang/tree/runtime._defer) 结构体

**2 1.13**  **栈上分配 减少了30%的额外开销**

**3 1.14** **基于开放编码的 defer**，**该关键字的额外开销可以忽略不计**

**defer的执行顺序**

它是一个“栈”的关系，也就是先进后出。一个函数中，写在前面的defer会比写在后面的defer调用的晚。

**defer与return谁先谁后**

(1) return最先执行，return负责将结果写入返回值中；

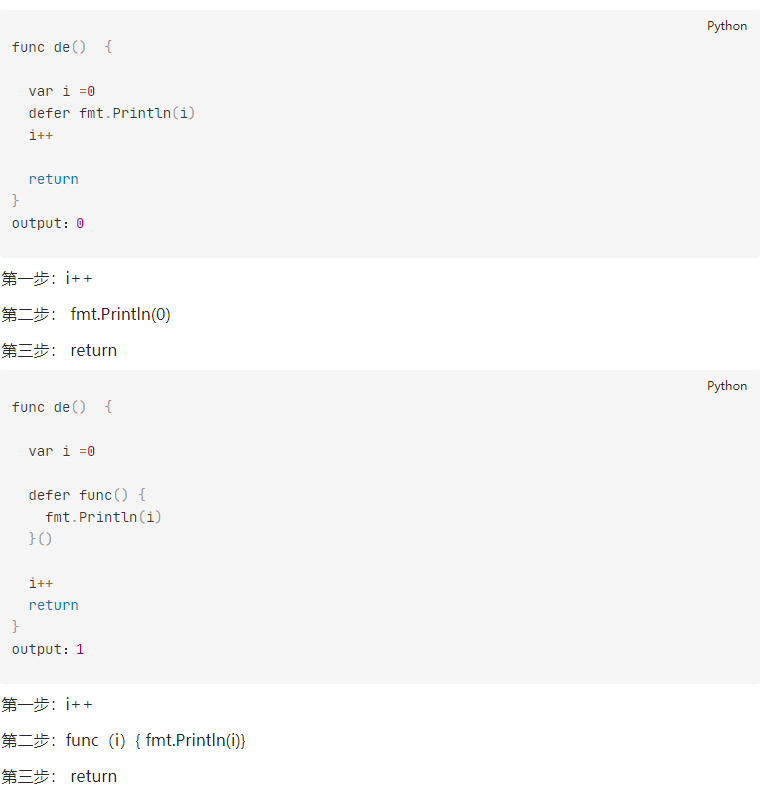
(2) 接着defer开始执行一些收尾工作；

(3) 最后函数携带当前返回值退出

return和defer的执行顺序：return是分为两步执行的，第一步赋值给返回值，第二步真正的返回到函数外部。而defer是在第一步之后执行。

**Defer场景**

**1无函数返回值的时候**，defer要注意后面跟的是函数还是语句



**2存在返回值**

（1）具名返回值

直接将return进行拆解为：将具名返回值ret赋值（赋值的是return后的常量或者变量）+return

然后将defer放在return前面进行分析

（2）匿名返回值

假定函数返回值变量为”ret”，将return进行拆解为：将具名返回值ret赋值（赋值的是return后的常量或者变量）+return

然后将defer放在return前面进行分析（注意，最后return的是ret）

## error

**1概念**

错误指的是可能出现问题的地方出现了问题。比如打开一个文件时失败，这种情况在人们的意料之中 。

异常指的是不应该出现问题的地方出现了问题。比如引用了空指针，这种情况在人们的意料之外

错误是业务过程的一部分，而异常不是 。

**2原理**

Golang 中的错误也是一种类型。错误用内置的 error 类型表示。就像其他类型，如 int，float64 等。

错误值可以存储在变量中，也可以从函数中返回

**error源码**

type error interface {

Error() string

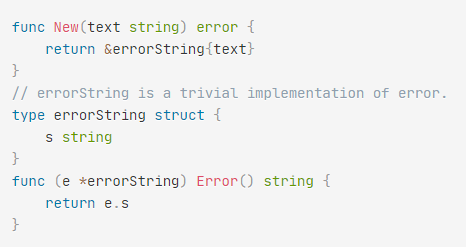
}

error 是一个接口类型，它包含一个 Error() 方法，返回值为 string。

**error创建**

**创建方式有两种：errors.New()，fmt.Errorf()**

**1 errors.New () 函数**



New () 函数返回一个错误，该错误的格式为给定的文本。

即使文本相同，每次对 New 的调用也会返回一个不同的错误值。

**缺点**

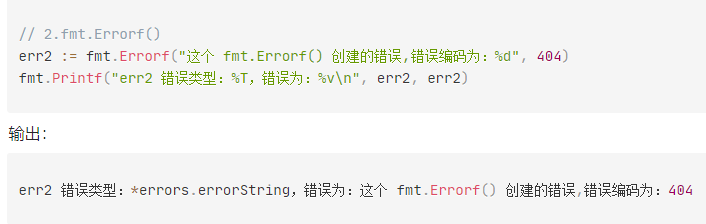
但是有些时候我们需要更加具体的信息。即需要具体的 **“上下文”** 信息，表明具体的错误值。

使用fmt.Errorf函数解决

**2 fmt.Errorf ()函数**

fmt.Errorf() 函数，它先将字符串格式化，并增加上下文的信息，更精确的描述错误。

**举例1**



err2 返回的错误类型也是 ：\*errors.errorString，

用fmt.Errorf() 创建的和errors.New(）的返回类型一样

**原因：**

当p.wrappedErr 为 nil 的时候，会调用 errors.New() 来创建错误。

所以err2 的错误类型是 \*errors.errorString

**p.wrappedErr：**

wrapErrs 字段，bool 类型 创建的错误如果包裹了其他错误，则不为空

**总结**

第一种：errors.New() 函数，其返回值类型为 \*errors.errorString。

第二种：fmt.Errorf() 函数 当使用 fmt.Errorf() 来创建错误时，核心有以下两点：

错误描述中不包含 %w 时，p.wrappedErr 为 nil，所以底层也是调用 errors.New() 创建错误。因此错误类型就是 \*errors.errorString。

错误描述中包含%w 时(包裹了其他错误)，p.wrappedErr 不为 nil，所以底层实例化 wrapError结构体指针。 因此错误类型是 \*fmt.wrapError，可以理解为包裹错误类型。

## Panic

**概念**

处理错误的方式是返回error给调用者

处理异常的方式是一般是显示的出发panic，提前结束程序运行

**优点**

比如同样是退出程序，与os.exit（）相比，panic的退出方式比较优雅 panic会做一定的善后处理使用defer函数，并且支持用recover消除panic）

**原则：**

1**前面：**defer 表达式的函数如果定义在 panic 后面，该函数在 panic 后就无法被执行到

2**递归调用：**panic会递归调用执行协程中的所有的defer

3**支持嵌套**：panic支持嵌套，defer中可以再次出发panic

4**捕获**：panic只能出发当前协程的defer函数，当前协程defer处理结束后，如果没有被recover（）捕获，则会引发程序退出。即panic不能被其他协程recover（）捕获

**panic原理**

panic 关键字在 Go 语言的源代码是由数据结构 runtime.\_panic 表示的。

argp 是指向 defer 调用时参数的指针

Panic中有link 字段指向了更早调用的 runtime.\_panic 结构；

recovered字段 表示当前 runtime.\_panic 是否被 recover 恢复；

argp 是指向 defer 调用时参数的指针

**流程：**

**1转换：**

编译器会将关键字 panic 转换成 [runtime.gopanic](https://draveness.me/golang/tree/runtime.gopanic)

**2加入panic列表**

创建新的 [runtime.\_panic](https://draveness.me/golang/tree/runtime._panic) 并添加到所在 Goroutine 的 \_panic 链表 的最前面；

**3循环调用defer链表并运行**

在循环中不断从当前 Goroutine 的 \_defer 中链表获取 [runtime.\_defer](https://draveness.me/golang/tree/runtime._defer) 并调用 [runtime.reflectcall](https://draveness.me/golang/tree/runtime.reflectcall) 运行延迟调用函数；

**4终止**

调用 [runtime.fatalpanic](https://draveness.me/golang/tree/runtime.fatalpanic) 中止整个程序；

## Recover

**概念**

内置函数recover（）用于消除panic并使得程序恢复正常

**原则：**

**1不能再嵌套中：**recover函数必须位于defer函数中，且不能出现在另一个嵌套函数中，否则无法捕获

**2无法回到原位置：**recover函数成功处理异常后，无法再次回到本函数发生panic的位置上继续执行

**3无法再次捕获：**painc被捕获后，无法在继续使用recover捕获

**recover原理**

**recover流程**

**1转换：**

编译器会将关键字 recover 转换成 [runtime.gorecover](https://draveness.me/golang/tree/runtime.gorecover)：

**2没有panic：**

当前Goroutine没有调用panic，那么该函数会直接返回 nil，这也是崩溃恢复在非 defer 中调用会失效的原因。

**3有panic：**

如果当前 Goroutine 调用了 panic，它会修改 [runtime.\_panic](https://draveness.me/golang/tree/runtime._panic) 的 recovered 字段标记成 true，

**4恢复：**

[runtime.gorecover](https://draveness.me/golang/tree/runtime.gorecover) 函数中并不包含恢复程序的逻辑，

[runtime.gopanic](https://draveness.me/golang/tree/runtime.gopanic) 函数负责程序的恢复

**恢复流程：**

**1取出栈指针和程序计数器进行恢复**：

调用 defer 关键字时，调用时的栈指针 sp 和程序计数器 pc 就已经存储到了 runtime.\_defer 结构体中

[runtime.gopanic](https://draveness.me/golang/tree/runtime.gopanic) 会从 [runtime.\_defer](https://draveness.me/golang/tree/runtime._defer) 结构体中取出程序计数器 pc 和栈指针 sp 并调用 [runtime.recovery](https://draveness.me/golang/tree/runtime.recovery) 函数进行恢复程序；

[runtime.recovery](https://draveness.me/golang/tree/runtime.recovery) 会根据传入的 pc 和 sp 跳转回 [runtime.deferproc](https://draveness.me/golang/tree/runtime.deferproc)；

runtime.recovery 在调度过程中会将函数的返回值设置成 1

编译器自动生成的代码会发现 [runtime.deferproc](https://draveness.me/golang/tree/runtime.deferproc) 的返回值不为 0，

这时会跳回 [runtime.deferreturn](https://draveness.me/golang/tree/runtime.deferreturn) 并恢复到正常的执行流程；

**2调用gogo函数：**

runtime.gogo 函数会跳回 defer 关键字调用的位置。

**总结**

**1编译器会负责做转换关键字的工作；**

将 panic 转换成 [runtime.gopanic](https://draveness.me/golang/tree/runtime.gopanic)

将 recover 转换成 [runtime.gorecover](https://draveness.me/golang/tree/runtime.gorecover)；

将 defer 转换成 [runtime.deferproc](https://draveness.me/golang/tree/runtime.deferproc)；

在调用 defer 的函数末尾调用 [runtime.deferreturn](https://draveness.me/golang/tree/runtime.deferreturn) 函数；

**2遇到gopanic，调用defer**

在运行过程中遇到 [runtime.gopanic](https://draveness.me/golang/tree/runtime.gopanic) 方法时，会从 Goroutine 的链表依次取出 [runtime.\_defer](https://draveness.me/golang/tree/runtime._defer) 结构体并执行；

**3调用defer，遇到reover**

如果调用延迟执行函数时遇到了 [runtime.gorecover](https://draveness.me/golang/tree/runtime.gorecover)

将 \_panic.recovered 标记成 true 并返回 panic 的参数；

在这次调用结束之后，[runtime.gopanic](https://draveness.me/golang/tree/runtime.gopanic) 会从 [runtime.\_defer](https://draveness.me/golang/tree/runtime._defer) 结构体中取出程序计数器 pc 和栈指针 sp 并调用 [runtime.recovery](https://draveness.me/golang/tree/runtime.recovery) 函数进行恢复程序；

[runtime.recovery](https://draveness.me/golang/tree/runtime.recovery) 会根据传入的 pc 和 sp 跳转回 [runtime.deferproc](https://draveness.me/golang/tree/runtime.deferproc)；

编译器自动生成的代码会发现 [runtime.deferproc](https://draveness.me/golang/tree/runtime.deferproc) 的返回值不为 0，

这时会跳回 [runtime.deferreturn](https://draveness.me/golang/tree/runtime.deferreturn) 并恢复到正常的执行流程；

**4调用defer，没有遇到reover**

如果没有遇到 [runtime.gorecover](https://draveness.me/golang/tree/runtime.gorecover) 就会依次遍历所有的 [runtime.\_defer](https://draveness.me/golang/tree/runtime._defer)，

并在最后调用 [runtime.fatalpanic](https://draveness.me/golang/tree/runtime.fatalpanic) 中止程序、打印 panic 的参数并返回错误码；

## 协程池