# 概述

Go 是一门用于并发编程的命令式编程语言，它主要由创造者 Google 进行开发，最初主要由 Robert Griesemer、Rob Pike 和 Ken Thompson 开发。这门语言的设计起始于 2007 年，并在 2009 年推出最初版本；而第一个稳定版本是 2012 年发布的 1.0 版本。

Go 有 C 风格的语法（没有预处理器）、垃圾回收机制，而且类似它在贝尔实验室里被开发出来的前辈们：Newsqueak（Rob Pike）、Alef（Phil Winterbottom）和 Inferno（Pike、Ritchie 等人），使用所谓的 goroutines（go协程）和channels（信道，一种基于 Hoare 的“通信顺序进程”理论的协程）提供内建的并发支持。

Go 程序以包的形式组织。包本质是一个包含 Go 文件的文件夹。包内的所有文件共享相同的命名空间，而包内的符号有两种可见性：以大写字母开头的符号对于其他包是可见，而其他符号则是该包私有的：

func PublicFunction() {

   fmt.Println("Hello world")

}

func privateFunction() {

   fmt.Println("Hello package")

}

# 类型

Go 有一个相当简单的类型系统：没有子类型（但有类型转换），没有泛型，没有多态函数，只有一些基本的类型：

* 基本类型int、int64、int8、uint、float32、float64
* struct结构体
* interface：一组方法的集合
* map[K, V]：一个从键类型到值类型的映射
* [number]Type：一些 Type 类型的元素组成的数组
* []Type：某种类型的切片（具有长度和功能的数组的指针）
* chan Type：一个线程安全的队列
* 指针 \*T 指向其他类型
* 函数
* 具名类型：可能具有关联方法的其他类型的别名（LCTT 译注：这里的别名并非指 Go 1.9 中的新特性“类型别名”）：

type T struct { foo int }

type T \*T

type T OtherNamedType

具名类型完全不同于它们的底层类型，所以你不能让它们互相赋值，但一些操作符，例如 +，能够处理同一底层数值类型的具名类型对象们（所以你可以在上面的示例中把两个 T加起来）。

映射、切片和信道是类似于引用的类型——它们实际上是包含指针的结构。包括数组（具有固定长度并可被拷贝）在内的其他类型则是值传递（拷贝）。

## 类型转换

类型转换类似于 C 或其他语言中的类型转换。它们写成这样子：

TypeName(value)

## 常量

Go 有“无类型”字面量和常量。

1 // 无类型整数字面量

const foo = 1 // 无类型整数常量

const foo int = 1 // int 类型常量

无类型值可以分为以下几类：UntypedBool、UntypedInt、UntypedRune、UntypedFloat、UntypedComplex、UntypedString 以及 UntypedNil（Go 称它们为基础类型，其他基础种类可用于具体类型，如 uint8）。

一个无类型值可以赋值给一个从基础类型中派生的具名类型；例如：

type someType int

const untyped = 2 // UntypedInt

const bar someType = untyped // OK: untyped 可以被赋值给 someType

const typed int = 2 // int

const bar2 someType = typed // error: int 不能被赋值给 someType

## 接口和对象

正如上面所说的，接口是一组方法的集合。Go 本身不是一种面向对象的语言，但它支持将方法关联到具名类型上：当声明一个函数时，可以提供一个接收者。接收者是函数的一个额外参数，可以在函数之前传递并参与函数查找，就像这样：

type SomeType struct { ... }

type SomeType struct { ... }

func (s \*SomeType) MyMethod() {

}

func main() {

   var s SomeType

   s.MyMethod()

}

如果对象实现了所有方法，那么它就实现了接口；例如，\*SomeType（注意指针）实现了下面的接口 MyMethoder，因此 \*SomeType 类型的值就能作为 MyMethoder 类型的值使用。最基本的接口类型是 interface{}，它是一个带空方法集的接口 —— 任何对象都满足该接口。

type MyMethoder interface {

   MyMethod()

}

合法的接收者类型是有些限制的；例如，具名类型可以是指针类型（例如，type MyIntPointer \*int），但这种类型不是合法的接收者类型。

## method 接收者使用值类型和指针类型的区别

在 Golang 语言中，function 的参数和 method 的接收者都可以选择使用值传递和指针传递（“引用传递”），需要注意的是，其中指针传递是传递的指针值的副本，而不是指针指向的数据的副本。也就是说 Golang 语言和 C 系的所有语言相同，一切传递都是值传递。

### method 接收者的类型选择

在使用关键字 type 定义的类型上定义 method，method 的接收者也可以作为 method 的参数，类似于 function 的参数，所以 method 的接收者和 function 参数一样，我们也需要考虑选择使用值类型和指针类型。

关于这个问题，我们通常会从两方面去考虑，一是如果该 method 需要修改接收者，那么接收者必须使用指针类型；二是如果接收者占用的内存大小较大，出于性能考虑，我们也会选择使用指针类型的接收者。

除此之外，我们还需考虑一致性。也就是说，如果该类型的某些 method 必须使用指针类型的接收者，其他 method 也应该使用指针类型的接收者。因此无论如何使用该类型，它的方法集都是一致的。

最后，如果接收者是基本类型，切片和小结构体，他们的值类型的内存占用较低，并且易读。所以，该情况下除非 method 的语义需要必须使用指针类型的接收者，否则，我们可以选择使用值类型的接收者。

### 复合类型

map 和 slice 值类似于指针：它们是包含指向底层 map 或 slice 数据的指针的描述符。复制 map 或 slice 值不会复制它指向的数据。需要注意的是，如果超过 slice 的容量，运行时会重新分配一个新内存地址。

关于接口类型，复制接口值将复制存储在接口值中的对象。如果接口值持有一个结构体，则复制接口值会复制该结构体。如果接口值持有指针，则复制接口值会复制指针，但不会复制它指向的数据。

### 值类型怎么避免拷贝副本

阅读到这里，读者朋友可能会简单认为使用值类型会拷贝副本，使用指针类型不会拷贝副本。实际上，我们可以通过优化代码，在不改变语义的前提下，实现使用值类型也不会拷贝副本。

type User struct {  
 name string  
}  
  
func (u User) SetNameValueType(str string) {  
 fmt.Printf("SetNameValueType() pointer:%p\n", &u) // SetNameValueType() pointer:0xc000096240  
 u.name = str  
}  
  
func (u User) ValueSetName(str string) User {  
 u.name = str  
 return u  
}  
  
func main () {  
 user2 := &User{}  
 fmt.Printf("user2 pointer:%p\n", user2) //user2 pointer:0xc000010290  
 user2.SetNameValueType("tom") // SetNameValueType() pointer:0xc0000102a0  
  
 user3 := &User{}  
 fmt.Printf("user3 pointer:%p\n", user3) //user3 pointer:0xc0000102b0  
 user3.ValueSetName("bob")  
 fmt.Printf("pointer:%p\n", user3) // pointer:0xc0000102b0  
}

阅读上面这段代码，我们发现 User 的 SetNameValueType 方法和 ValueSetName 方法，二者都是值传递，但是 SetNameValueType 方法会拷贝副本，ValueSetName 方法不会拷贝副本。原因是我们给 ValueSetName 方法定义了一个 User 类型的返回值，从而避免了 ValueSetName 方法拷贝副本。

### 总结

本文我们主要介绍了 method 的接收者使用值传递和指针传递的区别，并且讲述了选择使用值传递和指针传递需要考虑的决定因素，也指出了复合类型与值类型的区别。最后，使用一个简单示例演示了通过优化代码，在不改变语义的前提下，怎么实现使用值类型也不会拷贝副本。

# 控制流

Go 提供了三个主要的控制了语句：if、switch 和 for。这些语句同其他 C 风格语言内的语句非常类似，但有一些不同：

* 条件语句没有括号，所以条件语句是 if a == b {} 而不是 if (a == b) {}。大括号是必须的。
* 所有的语句都可以有初始化，比如这个 if result, err := someFunction(); err == nil { // use result }
* switch 语句在分支里可以使用任何表达式◈
* switch 语句可以处理空的表达式（等于 true）◈
* 默认情况下，Go 不会从一个分支进入下一个分支（不需要 break 语句），在程序块的末尾使用 fallthrough 则会进入下一个分支。
* 循环语句 for 不仅能循环值域：for key, val := range map { do something }

# 字符串

在go中，string是值类型，在用作返回或传参的时候会发生拷贝构造。优化string的方案也就是使用指针。

func GetName(id string)string

等价于

std::string GetName(string id)

## 高效地拼接字符串

Go 语言中，字符串是只读的，也就意味着每次修改操作都会创建一个新的字符串。如果需要拼接多次，应使用 strings.Builder，最小化内存拷贝次数。

var str strings.Builder  
for i := 0; i < 1000; i++ {  
 str.WriteString("a")  
}  
fmt.Println(str.String())

# 数组和切片

正如前边说的，数组是值类型，而切片是指向数组的指针。切片可以由现有的数组切片产生，也可以使用 make() 创建切片，这会创建一个匿名数组以保存元素。

slice1 := make([]int, 2, 5) // 分配 5 个元素，其中 2 个初始化为0

slice2 := array[:] // 整个数组的切片

slice3 := array[1:] // 除了首元素的切片

除了上述例子，还有更多可行的切片运算组合，但需要明了直观。

使用 append() 函数，切片可以作为一个变长数组使用。

slice = append(slice, value1, value2)

slice = append(slice, arrayOrSlice...)

切片也可以用于函数的变长参数。

## 返回切片

返回切片实际上是返回切片的指针，是浅拷贝。

深拷贝使用copy函数。

## 数组切片slice的容量问题

数组切片slice的容量问题带来的bug

import (

"fmt"

)

func main(){

array := [4]int{10, 20, 30, 40}

slice := array[0:2]

newSlice := append(slice, 50)

newSlice[1] += 1

fmt.Println(slice)

}

输出[10 21]————没有重新分配内存，返回之前的指针。

如果稍作修改，将以上newSlice改为扩容三次，newSlice := append(append(append(slice, 50), 100), 150)如下:

import (

"fmt"

)

func main(){

array := [4]int{10, 20, 30, 40}

slice := array[0:2]

newSlice := append(append(append(slice, 50), 100), 150)

newSlice[1] += 1

fmt.Println(slice)

}

输出：[10 20]

这就要从Golang切片的扩容说起了；切片的扩容，就是当切片添加元素时，切片容量不够了，就会扩容，扩容的大小遵循下面的原则：

### 扩容因子

如果切片的容量小于1024个元素，那么扩容的时候slice的cap就翻番，乘以2。

一旦元素个数超过1024个元素，增长因子就变成1.25，即每次增加原来容量的四分之一。

### 重新分配内存

如果扩容之后，还没有触及原数组的容量，那么，切片中的指针指向的位置，就还是原数组（这就是产生bug的原因）；

如果扩容之后，超过了原数组的容量，那么，Go就会开辟一块新的内存，把原来的值拷贝过来，这种情况丝毫不会影响到原数组。

## for range

用for range来遍历数组或者map的时候，被遍历的指针是不变的，每次遍历仅执行struct值的拷贝。

import "fmt"

type student struct{

Name string

Age int

}

func main(){

var stus []student

stus = []student{

{Name:"one", Age: 18},

{Name:"two", Age: 19},

}

data := make(map[int]\*student)

for i, v := range stus{

data[i] = &v //应该改为：data[i] = &stus[i]

}

for i, v := range data{

fmt.Printf("key=%d, value=%v \n", i,v)

}

}

# map

maps是简单的键值对储存容器，并支持索引和分配。但它们不是线程安全的。

someValue := someMap[someKey]

// 如果键值不在 someMap 中，变量 ok 会赋值为 `false`

someValue, ok := someMap[someKey]

someMap[someKey] = someValue

## map引用不存在的key

map引用不存在的key，不报错

import (

"fmt"

)

func main(){

newMap := make(map[string]int)

fmt.Println(newMap["a"])

}

不报错。不同于PHP，Golang的map和Java的HashMap类似，Java引用不存在的会返回null，而Golang会返回初始值。

## map使用range遍历

map使用range遍历顺序问题，并不是录入的顺序，而是随机顺序

import (

"fmt"

)

func main(){

newMap := make(map[int]int)

for i := 0; i < 10; i++{

newMap[i] = i

}

for key, value := range newMap{

fmt.Printf("key is %d, value is %d\n", key, value)

}

}

输出：

key is 1, value is 1

key is 3, value is 3

key is 5, value is 5

key is 7, value is 7

key is 9, value is 9

key is 0, value is 0

key is 2, value is 2

key is 4, value is 4

key is 6, value is 6

key is 8, value is 8

是杂乱无章的顺序。map的遍历顺序不固定，这种设计是有意为之的，能为能防止程序依赖特定遍历顺序。

## 数据结构

在 $GOROOT/src/pkg/runtime/hashmap.goc 找到它的实现。

本质是哈希表，等同于std::std::unordered\_map。

struct Hmap  
{  
    uint8   B;    // 可以容纳2^B个项  
    uint16  bucketsize;   // 每个桶的大小  
  
    byte    \*buckets;     // 2^B个Buckets的数组  
    byte    \*oldbuckets;  // 前一个buckets，只有当正在扩容时才不为空  
};

上面给出的结构体只是Hmap的部分的域。需要注意到的是，这里直接使用的是Bucket的数组，而不是Bucket\*指针的数组。这意味着，第一个Bucket和后面溢出链的Bucket分配有些不同。第一个Bucket是用的一段连续的内存空间，而后面溢出链的Bucket的空间是使用mallocgc分配的。

这个hash结构使用的是一个可扩展哈希的算法，由hash值mod当前hash表大小决定某一个值属于哪个桶，而hash表大小是2的指数，即上面结构体中的2^B。每次扩容，会增大到上次大小的两倍。结构体中有一个buckets和一个oldbuckets是用来实现增量扩容的。正常情况下直接使用buckets，而oldbuckets为空。如果当前哈希表正在扩容中，则oldbuckets不为空，并且buckets大小是oldbuckets大小的两倍。

具体的Bucket结构如下所示：

struct Bucket  
{  
    uint8  tophash[BUCKETSIZE]; // hash值的高8位....低位从bucket的array定位到bucket  
    Bucket \*overflow;           // 溢出桶链表，如果有  
    byte   data[1];             // BUCKETSIZE keys followed by BUCKETSIZE values  
};

其中BUCKETSIZE是用宏定义的8，每个bucket中存放最多8个key/value对, 如果多于8个，那么会申请一个新的bucket，并将它与之前的bucket链起来。

按key的类型采用相应的hash算法得到key的hash值。将hash值的低位当作Hmap结构体中buckets数组的index，找到key所在的bucket。将hash的高8位存储在了bucket的tophash中。注意，这里高8位不是用来当作key/value在bucket内部的offset的，而是作为一个主键，在查找时对tophash数组的每一项进行顺序匹配的。先比较hash值高位与bucket的tophash是否相等，如果相等则再比较bucket的第i个的key与所给的key是否相等。如果相等，则返回其对应的value，反之，在overflow buckets中按照上述方法继续寻找。

## 增量扩容

大家都知道哈希表就是以空间换时间，访问速度是直接跟填充因子相关的，所以当哈希表太满之后就需要进行扩容。

如果扩容前的哈希表大小为2^B，扩容之后的大小为2^(B+1)，每次扩容都变为原来大小的两倍，哈希表大小始终为2的指数倍，则有(hash mod 2^B)等价于(hash & (2^B-1))。这样可以简化运算，避免了取余操作。

假设扩容之前容量为X，扩容之后容量为Y，对于某个哈希值hash，一般情况下(hash mod X)不等于(hash mod Y)，所以扩容之后要重新计算每一项在哈希表中的新位置。当hash表扩容之后，需要将那些旧的pair重新哈希到新的table上(源代码中称之为evacuate)， 这个工作并没有在扩容之后一次性完成，而是逐步的完成（在insert和remove时每次搬移1-2个pair），Go语言使用的是增量扩容。

为什么会增量扩容呢？主要是缩短map容器的响应时间。假如我们直接将map用作某个响应实时性要求非常高的web应用存储，如果不采用增量扩容，当map里面存储的元素很多之后，扩容时系统就会卡住，导致较长一段时间内无法响应请求。不过增量扩容本质上还是将总的扩容时间分摊到了每一次哈希操作上面。

扩容会建立一个大小是原来2倍的新的表，将旧的bucket搬到新的表中之后，并不会将旧的bucket从oldbucket中删除，而是加上一个已删除的标记。

正是由于这个工作是逐渐完成的，这样就会导致一部分数据在old table中，一部分在new table中， 所以对于hash table的insert, remove, lookup操作的处理逻辑产生影响。只有当所有的bucket都从旧表移到新表之后，才会将oldbucket释放掉。

扩容的填充因子是多少呢？如果grow的太频繁，会造成空间的利用率很低， 如果很久才grow，会形成很多的overflow buckets，查找的效率也会下降。这个平衡点如何选取呢(在go中，这个平衡点是有一个宏控制的(#define LOAD 6.5), 它的意思是这样的，如果table中元素的个数大于table中能容纳的元素的个数， 那么就触发一次grow动作。那么这个6.5是怎么得到的呢？原来这个值来源于作者的一个测试程序，遗憾的是没能找到相关的源码，不过作者给出了测试的结果：

    LOAD    %overflow  bytes/entry     hitprobe    missprobe  
        4.00         2.13        20.77         3.00         4.00  
        4.50         4.05        17.30         3.25         4.50  
        5.00         6.85        14.77         3.50         5.00  
        5.50        10.55        12.94         3.75         5.50  
        6.00        15.27        11.67         4.00         6.00  
        6.50        20.90        10.79         4.25         6.50  
        7.00        27.14        10.15         4.50         7.00  
        7.50        34.03         9.73         4.75         7.50  
        8.00        41.10         9.40         5.00         8.00  
  
 %overflow   = percentage of buckets which have an overflow bucket  
 bytes/entry = overhead bytes used per key/value pair  
 hitprobe    = # of entries to check when looking up a present key  
 missprobe   = # of entries to check when looking up an absent key

可以看出作者取了一个相对适中的值。

## 查找过程

1. 根据key计算出hash值。
2. 如果存在old table, 首先在old table中查找，如果找到的bucket已经evacuated，转到步骤3。反之，返回其对应的value。
3. 在new table中查找对应的value。

这里一个细节需要注意一下。不认真看可能会以为低位用于定位bucket在数组的index，那么高位就是用于key/valule在bucket内部的offset。事实上高8位不是用作offset的，而是用于加快key的比较的。

do { //对每个桶b  
    //依次比较桶内的每一项存放的tophash与所求的hash值高位是否相等  
    for(i = 0, k = b->data, v = k + h->keysize \* BUCKETSIZE; i < BUCKETSIZE; i++, k += h->keysize, v += h->valuesize) {  
        if(b->tophash[i] == top) {   
            k2 = IK(h, k);  
            t->key->alg->equal(&eq, t->key->size, key, k2);  
            if(eq) { //相等的情况下再去做key比较...  
                \*keyp = k2;  
                return IV(h, v);  
            }  
        }  
    }  
    b = b->overflow; //b设置为它的下一下溢出链  
} while(b != nil);

### 插入过程分析

1. 根据key算出hash值，进而得出对应的bucket。
2. 如果bucket在old table中，将其重新散列到new table中。
3. 在bucket中，查找空闲的位置，如果已经存在需要插入的key，更新其对应的value。
4. 根据table中元素的个数，判断是否grow table。
5. 如果对应的bucket已经full，重新申请新的bucket作为overbucket。
6. 将key/value pair插入到bucket中。

这里也有几个细节需要注意一下。

在扩容过程中，oldbucket是被冻结的，查找时会在oldbucket中查找，但不会在oldbucket中插入数据。如果在oldbucket是找到了相应的key，做法是将它迁移到新bucket后加入evalucated标记。并且还会额外的迁移另一个pair。

然后就是只要在某个bucket中找到第一个空位，就会将key/value插入到这个位置。也就是位置位于bucket前面的会覆盖后面的(类似于存储系统设计中做删除时的常用的技巧之一，直接用新数据追加方式写，新版本数据覆盖老版本数据)。找到了相同的key或者找到第一个空位就可以结束遍历了。不过这也意味着做删除时必须完全的遍历bucket所有溢出链，将所有的相同key数据都删除。所以目前map的设计是为插入而优化的，删除效率会比插入低一些。

## map设计中的性能优化

读完map源代码发现作者还是做了很多设计上的选择的。本人水平有限，谈不上优劣的点评，这里只是拿出来与读者分享。

HMap中是Bucket的数组，而不是Bucket指针的数组。好的方面是可以一次分配较大内存，减少了分配次数，避免多次调用mallocgc。但相应的缺点，其一是可扩展哈希的算法并没有发生作用，扩容时会造成对整个数组的值拷贝(如果实现上用Bucket指针的数组就是指针拷贝了，代价小很多)。其二是首个bucket与后面产生了不一致性。这个会使删除逻辑变得复杂一点。比如删除后面的溢出链可以直接删除，而对于首个bucket，要等到evalucated完毕后，整个oldbucket删除时进行。

没有重用设freelist重用删除的结点。作者把这个加了一个TODO的注释，不过想了一下觉得这个做的意义不大。因为一方面，bucket大小并不一致，重用比较麻烦。另一方面，下层存储已经做过内存池的实现了，所以这里不做重用也会在内存分配那一层被重用的，

bucket直接key/value和间接key/value优化。这个优化做得蛮好的。注意看代码会发现，如果key或value小于128字节，则它们的值是直接使用的bucket作为存储的。否则bucket中存储的是指向实际key/value数据的指针，

bucket存8个key/value对。查找时进行顺序比较。第一次发现高位居然不是用作offset，而是用于加快比较的。定位到bucket之后，居然是一个顺序比较的查找过程。后面仔细想了想，觉得还行。由于bucket只有8个，顺序比较下来也不算过分。仍然是O(1)只不过前面系数大一点点罢了。相当于hash到一个小范围之后，在这个小范围内顺序查找。

插入删除的优化。前面已经提过了，插入只要找到相同的key或者第一个空位，bucket中如果存在一个以上的相同key，前面覆盖后面的(只是如果，实际上不会发生)。而删除就需要遍历完所有bucket溢出链了。这样map的设计就是为插入优化的。考虑到一般的应用场景，这个应该算是很合理的。

作者还列了另个2个TODO：将多个几乎要empty的bucket合并；如果table中元素很少，考虑shrink table。(毕竟现在的实现只是单纯的grow)。

# Go协程

关键词 go 会产生一个新的 goroutinue（Go协程），这是一个可以并行执行的函数。它可以用于任何函数调用，甚至一个匿名函数：

func main() {

 ...

   go func() {

       ...

   }()

   go some\_function(some\_argument)

}

# 信道

Go 协程通常和信道channels结合，用来提供一种通信顺序进程的扩展。信道是一个并发安全的队列，而且可以选择是否缓冲数据：

var unbuffered = make(chan int) // 直到数据被读取时完成数据块发送

var buffered = make(chan int, 5) // 最多有 5 个未读取的数据块

## 运算符 <- 用于和单个信道进行通信。

valueReadFromChannel := <- channel

otherChannel <- valueToSend

## make(chan int) 和 make(chan int, 1)

make(chan int) 和 make(chan int, 1)是不一样的

chan一旦被写入数据后，当前goruntine就会被阻塞，直到有人接收才可以（即 “ <- ch”），如果没人接收，它就会一直阻塞着。而如果chan带一个缓冲，就会把数据放到缓冲区中，直到缓冲区满了，才会阻塞。

import "fmt"

func main(){

ch := make(chan int) //改为 ch := make(chan int, 1) 就好了

ch <- 1

fmt.Println("success")

}

## select

golang 的 select 的功能和 select, poll, epoll 相似， 就是监听 IO 操作，当 IO 操作发生时，触发相应的动作。

select 的代码形式和 switch 非常相似， 不过 select 的 case 里的操作语句只能是”IO操作”（不仅仅是取值<-channel，赋值channel<-也可以）， select 会一直等待等到某个 case 语句完成，也就是等到成功从channel中读到数据。 则 select 语句结束。

import "fmt"

func main(){

ch := make(chan int, 1)

ch <- 1

select {

case msg :=<-ch:

fmt.Println(msg)

default:

fmt.Println("default")

}

fmt.Println("success")

}

default可以判断chan是否已经满了

import "fmt"

func main(){

ch := make(chan int, 1)

select {

case msg :=<-ch:

fmt.Println(msg)

default:

fmt.Println("default")

}

fmt.Println("success")

}

此时因为ch中没有写入数据，为空，所以 case不会读取成功。 则 select 执行 default 语句。

### 语句 select 允许多个信道进行通信

select {

   case incoming := <- inboundChannel:

   // 一条新消息

   case outgoingChannel <- outgoing:

   // 可以发送消息

}

## 只读只写chan

channel作为函数参数传递，可以声明为只取(<- chan)或者只发送(chan <-)

一个函数在将channel作为一个类型的参数来声明的时候，可以将channl声明为只可以取值(<- chan)或者只可以发送值(chan <-)，不特殊说明，则既可以取值，也可以发送值。

只可以发送值

func setData(ch chan <- string){

//TODO

}

只可以取值:

func setData(ch <- chan string){

//TODO

}

## 使用channel时，注意goroutine之间的执行流程问题

package main

import (

"fmt"

)

func main(){

ch := make(chan string)

go setData(ch)

fmt.Println(<-ch)

fmt.Println(<-ch)

fmt.Println(<-ch)

fmt.Println(<-ch)

fmt.Println(<-ch)

}

func setData(ch chan string){

ch <- "test"

ch <- "hello wolrd"

ch <- "123"

ch <- "456"

ch <- "789"

}

以上代码的执行流程是怎样的呢？  
一个基于无缓存channel的发送或者取值操作，会导致当前goroutine阻塞，一直等待到另外的一个goroutine做相反的取值或者发送操作以后，才会正常跑。  
 以上例子中的流程是这样的：

主goroutine等待接收，另外的那一个goroutine发送了“test”并等待处理；完成通信后，打印出”test”；两个goroutine各自继续跑自己的。  
 主goroutine等待接收，另外的那一个goroutine发送了“hello world”并等待处理；完成通信后，打印出”hello world”；两个goroutine各自继续跑自己的。  
 主goroutine等待接收，另外的那一个goroutine发送了“123”并等待处理；完成通信后，打印出”123”；两个goroutine各自继续跑自己的。  
 主goroutine等待接收，另外的那一个goroutine发送了“456”并等待处理；完成通信后，打印出”456”；两个goroutine各自继续跑自己的。  
 主goroutine等待接收，另外的那一个goroutine发送了“789”并等待处理；完成通信后，打印出”789”；两个goroutine各自继续跑自己的。

记住：Golang的channel是用来goroutine之间通信的，且通信过程中会阻塞。

# defer 声明

Go 提供语句 defer 允许函数退出时调用执行预定的函数。它可以用于进行资源释放操作，例如：

func myFunc(someFile io.ReadCloser) {

   defer someFile.close()

   /\* 文件相关操作 \*/

}

当然，它允许使用匿名函数作为被调函数，而且编写被调函数时可以像平常一样使用任何变量。

## 多个defer

多个defer出现的时候，多个defer之间按照LIFO（后进先出）的顺序执行。

package main

import "fmt"

func main(){

defer func(){

fmt.Println("1")

}()

defer func(){

fmt.Println("2")

}()

defer func(){

fmt.Println("3")

}()

}

# 错误处理

Go 没有提供异常类或者结构化的错误处理。然而，它通过第二个及后续的返回值来返回错误从而处理错误：

func Read(p []byte) (n int, err error)

// 内建类型：

type error interface {

   Error() string

}

必须在代码中检查错误或者赋值给 \_：

n0, \_ := Read(Buffer) // 忽略错误

n, err := Read(buffer)

if err != nil {

   return err

}

有两个函数可以快速跳出和恢复调用栈：panic() 和 recover()。当 panic() 被调用时，调用栈开始弹出，同时每个 defer 函数都会正常运行。当一个 defer 函数调用 recover()时，调用栈停止弹出，同时返回函数 panic() 给出的值。如果我们让调用栈正常弹出而不是由于调用 panic() 函数，recover() 将只返回 nil。在下面的例子中，defer 函数将捕获 panic() 抛出的任何 error 类型的值并储存在错误返回值中。第三方库中有时会使用这个方法增强递归代码的可读性，如解析器，同时保持公有函数仍使用普通错误返回值。

## panic中可以传任何值

panic中可以传任何值不仅仅可以传string

package main

import "fmt"

func main(){

defer func(){

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

fmt.Println(r)

}

}()

panic([]int{12312})

}

func Function() (err error) {

   defer func() {

       s := recover()

       switch s := s.(type) {  // type switch

           case error:

               err = s         // s has type error now

           default:

               panic(s)

       }

   }

}

# 同步

## sync.Mutex

互斥锁，相当于linux C的pthread\_mutex\_t，std::mutex。

## sync.RWLock

读写锁，相当于Linux C的pthread\_rwlock\_t，std::shared\_lock。

## sync.Cond

条件变量，相当于pthread\_cond\_t，std::condition\_variable。Wait()，Signal()，Broadcast()。

## atomic.Value

在 Go 语言标准库中，sync/atomic包将底层硬件提供的原子操作封装成了 Go 的函数。但这些操作只支持几种基本数据类型，因此为了扩大原子操作的适用范围，Go 语言在 1.4 版本的时候向sync/atomic包中添加了一个新的类型Value。此类型的值相当于一个容器，可以被用来“原子地"存储（Store）和加载（Load）任意类型的值。

### 锁和原子操作

有用户报告了encoding/gob包在多核机器上（80-core）上的性能问题，认为encoding/gob之所以不能完全利用到多核的特性是因为它里面使用了大量的互斥锁（mutex），如果把这些互斥锁换成用atomic.LoadPointer/StorePointer来做并发控制，那性能将能提升20倍。

针对这个问题，有人提议在已有的atomic包的基础上封装出一个atomic.Value类型，这样用户就可以在不依赖 Go 内部类型unsafe.Pointer的情况下使用到atomic提供的原子操作。所以我们现在看到的atomic包中除了atomic.Value外，其余都是早期由汇编写成的，并且atomic.Value类型的底层实现也是建立在已有的atomic包的基础上。

Mutex由操作系统实现，而atomic包中的原子操作则由底层硬件直接提供支持。在 CPU 实现的指令集里，有一些指令被封装进了atomic包，这些指令在执行的过程中是不允许中断（interrupt）的，因此原子操作可以在lock-free的情况下保证并发安全，并且它的性能也能做到随 CPU 个数的增多而线性扩展。

### 原子性

一个或者多个操作在 CPU 执行的过程中不被中断的特性，称为原子性（atomicity） 。这些操作对外表现成一个不可分割的整体，他们要么都执行，要么都不执行，外界不会看到他们只执行到一半的状态。而在现实世界中，CPU 不可能不中断的执行一系列操作，但如果我们在执行多个操作时，能让他们的中间状态对外不可见，那我们就可以宣称他们拥有了"不可分割”的原子性。

有些朋友可能不知道，在 Go（甚至是大部分语言）中，一条普通的赋值语句其实不是一个原子操作。例如，在32位机器上写int64类型的变量就会有中间状态，因为它会被拆成两次写操作（MOV）——写低 32 位和写高 32 位。

如果一个线程刚写完低32位，还没来得及写高32位时，另一个线程读取了这个变量，那它得到的就是一个毫无逻辑的中间变量，这很有可能使我们的程序出现诡异的 Bug。

这还只是一个基础类型，如果我们对一个结构体进行赋值，那它出现并发问题的概率就更高了。很可能写线程刚写完一小半的字段，读线程就来读取这个变量，那么就只能读到仅修改了一部分的值。这显然破坏了变量的完整性，读出来的值也是完全错误的。

面对这种多线程下变量的读写问题，我们的主角——atomic.Value登场了，它使得我们可以不依赖于不保证兼容性的unsafe.Pointer类型，同时又能将任意数据类型的读写操作封装成原子性操作（让中间状态对外不可见）。

### 使用姿势

atomic.Value类型对外暴露的方法就两个：

v.Store(c) - 写操作，将原始的变量c存放到一个atomic.Value类型的v里。

c = v.Load() - 读操作，从线程安全的v中读取上一步存放的内容。

简洁的接口使得它的使用也很简单，只需将需要作并发保护的变量读取和赋值操作用Load()和Store()代替就行了。

### 数据结构

atomic.Value被设计用来存储任意类型的数据，所以它内部的字段是一个interface{}类型，非常的简单粗暴。

type Value struct {  
  v interface{}  
}

除了Value外，这个文件里还定义了一个ifaceWords类型，这其实是一个空interface (interface{}）的内部表示格式（参见runtime/runtime2.go中eface的定义）。它的作用是将interface{}类型分解，得到其中的两个字段。

type ifaceWords struct {  
  typ  unsafe.Pointer  
  data unsafe.Pointer  
}

#### 写入（Store）操作

func (v \*Value) Store(x interface{}) {  
  if x == nil {  
    panic("sync/atomic: store of nil value into Value")  
  }  
  vp := (\*ifaceWords)(unsafe.Pointer(v))  // Old value  
  xp := (\*ifaceWords)(unsafe.Pointer(&x)) // New value  
  for {  
    typ := LoadPointer(&vp.typ)  
    if typ == nil {  
      // Attempt to start first store.  
      // Disable preemption so that other goroutines can use  
      // active spin wait to wait for completion; and so that  
      // GC does not see the fake type accidentally.  
      runtime\_procPin()  
      if !CompareAndSwapPointer(&vp.typ, nil, unsafe.Pointer(^uintptr(0))) {  
        runtime\_procUnpin()  
        continue  
      }  
      // Complete first store.  
      StorePointer(&vp.data, xp.data)  
      StorePointer(&vp.typ, xp.typ)  
      runtime\_procUnpin()  
      return  
    }  
    if uintptr(typ) == ^uintptr(0) {  
      // First store in progress. Wait.  
      // Since we disable preemption around the first store,  
      // we can wait with active spinning.  
      continue  
    }  
    // First store completed. Check type and overwrite data.  
    if typ != xp.typ {  
      panic("sync/atomic: store of inconsistently typed value into Value")  
    }  
    StorePointer(&vp.data, xp.data)  
    return  
  }  
}

大概的逻辑：

* 第5~6行 - 通过unsafe.Pointer将现有的和要写入的值分别转成ifaceWords类型，这样我们下一步就可以得到这两个interface{}的原始类型（typ）和真正的值（data）。
* 从第7行开始就是一个无限 for 循环。配合CompareAndSwap食用，可以达到乐观锁的功效。
* 第8行，我们可以通过LoadPointer这个原子操作拿到当前Value中存储的类型。下面根据这个类型的不同，分3种情况处理。

1. 第一次写入（第9~24行） - 一个Value实例被初始化后，它的typ字段会被设置为指针的零值 nil，所以第9行先判断如果typ是 nil 那就证明这个Value还未被写入过数据。那之后就是一段初始写入的操作：

* runtime\_procPin()这是runtime中的一段函数，具体的功能我不是特别清楚，也没有找到相关的文档。这里猜测一下，一方面它禁止了调度器对当前 goroutine 的抢占（preemption），使得它在执行当前逻辑的时候不被打断，以便可以尽快地完成工作，因为别人一直在等待它。另一方面，在禁止抢占期间，GC 线程也无法被启用，这样可以防止 GC 线程看到一个莫名其妙的指向^uintptr(0)的类型（这是赋值过程中的中间状态）。
* 使用CAS操作，先尝试将typ设置为^uintptr(0)这个中间状态。如果失败，则证明已经有别的线程抢先完成了赋值操作，那它就解除抢占锁，然后重新回到 for 循环第一步。
* 如果设置成功，那证明当前线程抢到了这个"乐观锁”，它可以安全的把v设为传入的新值了（19~23行）。注意，这里是先写data字段，然后再写typ字段。因为我们是以typ字段的值作为写入完成与否的判断依据的。

1. 第一次写入还未完成（第25~30行）- 如果看到 typ字段还是^uintptr(0)这个中间类型，证明刚刚的第一次写入还没有完成，所以它会继续循环，“忙等"到第一次写入完成。
2. 第一次写入已完成（第31行及之后） - 首先检查上一次写入的类型与这一次要写入的类型是否一致，如果不一致则抛出异常。反之，则直接把这一次要写入的值写入到data字段。

这个逻辑的主要思想就是，为了完成多个字段的原子性写入，我们可以抓住其中的一个字段，以它的状态来标志整个原子写入的状态。这个想法我在 TiDB 的事务实现中看到过类似的，他们那边叫Percolator模型，主要思想也是先选出一个primaryRow，然后所有的操作也是以primaryRow的成功与否作为标志。嗯，果然是太阳底下没有新东西。

#### 读取（Load）操作

先上代码：

func (v \*Value) Load() (x interface{}) {  
 vp := (\*ifaceWords)(unsafe.Pointer(v))  
 typ := LoadPointer(&vp.typ)  
 if typ == nil || uintptr(typ) == ^uintptr(0) {  
   // First store not yet completed.  
   return nil  
 }  
 data := LoadPointer(&vp.data)  
 xp := (\*ifaceWords)(unsafe.Pointer(&x))  
 xp.typ = typ  
 xp.data = data  
 return  
}

读取相对就简单很多了，它有两个分支：

如果当前的typ是 nil 或者^uintptr(0)，那就证明第一次写入还没有开始，或者还没完成，那就直接返回 nil （不对外暴露中间状态）。

否则，根据当前看到的typ和data构造出一个新的interface{}返回出去。

### 总结

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。锁应当用来保护一段逻辑，对于一个变量更新的保护，原子操作通常会更有效率，并且更能利用计算机多核的优势，如果要更新的是一个复合对象，则应当使用atomic.Value封装好的实现。

# 反射

## Type

Type表示Go语言的一个类型，它是一个有很多方法的接口，这些方法可以用来识别类型以及透视类型的组成部分。

## Value

Value可以包含一个任意类型的值。

## reflect.Value和interface{}

reflect.Value和interface{}都可以包含任意的值。二者的区别是空接口隐藏了值的布局信息、内置操作和相关方法。

# unsafe.Pointer

出于安全考虑，Go 语言并不支持直接操作内存，但它的标准库中又提供一种不安全（不保证向后兼容性） 的指针类型unsafe.Pointer，让程序可以灵活的操作内存。

unsafe.Pointer的特别之处在于，它可以绕过 Go 语言类型系统的检查，与任意的指针类型互相转换。也就是说，如果两种类型具有相同的内存结构（layout），我们可以将unsafe.Pointer当做桥梁，让这两种类型的指针相互转换，从而实现同一份内存拥有两种不同的解读方式。

比如说，[]byte和string其实内部的存储结构都是一样的，但 Go 语言的类型系统禁止他俩互换。如果借助unsafe.Pointer，我们就可以实现在零拷贝的情况下，将[]byte数组直接转换成string类型。

## 介绍

Golang 语言中的 unsafe 包中包含的操作会绕过 Golang 程序的类型安全检查，直接操作内存，从而达到提升性能的目的。导入 unsafe 包可能是不可移植（non-portable）的（随着 Golang 的版本迭代，可能会失效），并且不受 Go 1 兼容性准则的保护，所以我们应该谨慎使用。

本文主要介绍 unsafe 包的 unsafe.Pointer，它表示任意类型的指针，它类似于 C 语言中的无类型指针 void\*，可以作为指针类型 \*T 和 uintptr 类型值之间互相转换的中转站。

我们知道 Golang 语言中的指针类型 \*T，表示一个指向 T 类型变量的指针，因为 Golang 语言是强类型的静态语言，为了安全考虑，规定两个不同的指针类型之间不可以互相转换，比如 \*int 不能与 \*float64 互相转换。但是，实际上是可以使用 unsafe.Pointer 进行转换。

Golang 语言中的内置数据类型 uintptr 也可以表示任何指针，它实际是数值类型，可以用于存储内存地址。它和 unsafe.Pointer 最大的区别是 unsafe.Pointer 不支持指针运算，比如 + 运算符，但 uintptr 可以支持。以下是 uintptr 的源码：

// uintptr is an integer type that is large enough to hold the bit pattern of  
// any pointer.  
type uintptr uintptr

## unsafe.Ponter 类型

有了前面内容的铺垫，我们开始介绍 unsafe.Ponter，它表示指向任意类型的指针。以下是 unsafe 包的源码：

// ArbitraryType is here for the purposes of documentation only and is not actually  
// part of the unsafe package. It represents the type of an arbitrary Go expression.  
type ArbitraryType int  
type Pointer \*ArbitraryType

unsafe.Ponter 类型有四个转换规则：

* 任何类型的指针值 \*T 都可以转换为 unsafe.Pointer。
* unsafe.Pointer 可以转换为任何类型的指针值 \*T。
* uintptr 可以转换为 unsafe.Pointer。
* unsafe.Pointer 可以转换为 uintptr。

unsafe.Pointer 允许程序绕过类型安全检查读写任意内存，所以使用时应格外小心。

unsafe.Pointer 包含 6 个使用模式：

* 使用 unsafe.Pointer 作为中转，将一个指针类型 \*T 转换为另外一个指针类型 \*T。
* 将 unsafe.Pointer 转换为 uintptr（但不返回给 unsafe.Pointer），然后使用 uintptr 值。
* 将 unsafe.Pointer 转换为 uintptr，然后使用 uintptr 值进行算术运算，最后将运算结果 uintptr 值再转换为 unsafe.Pointer。
* 调用 syscall.Syscall 时，将 unsafe.Pointer 转换为 uintptr 值，作为参数传递。
* 将 reflect.Value.Pointer 或 reflect.Value.UnsafeAddr 的返回结果 uintptr 值，从 uintptr 转换为 unsafe.Pointer。
* 将 reflect.SliceHeader 或 reflect.StringHeader 值的 Data 字段与 unsafe.Pointer 进行转换。

# 基础

## Q1 = 和 := 的区别？

:= 声明+赋值

= 仅赋值

var foo int  
foo = 10  
// 等价于  
foo := 10

## Q2 指针的作用？

指针用来保存变量的地址。

例如

var x = 5  
var p \*int = &x  
fmt.Printf("x = %d", \*p) // x 可以用 \*p 访问

\* 运算符，也称为解引用运算符，用于访问地址中的值。

＆运算符，也称为地址运算符，用于返回变量的地址

## Q3 Go 允许多个返回值吗？

允许

func swap(x, y string) (string, string) {  
 return y, x  
}  
  
func main() {  
 a, b := swap("A", "B")  
 fmt.Println(a, b) // B A  
}

## Q4 Go 有异常类型吗？

Go 没有异常类型，只有错误类型（Error），通常使用返回值来表示异常状态。

f, err := os.Open("test.txt")  
if err != nil {  
 log.Fatal(err)  
}

## Q5 什么是协程（Goroutine）

Goroutine 是与其他函数或方法同时运行的函数或方法。

Goroutines 可以被认为是轻量级的线程。 与线程相比，创建 Goroutine 的开销很小。 Go应用程序同时运行数千个 Goroutine 是非常常见的做法。

Linux系统线程完全由内核调度，一个用户线程被映射为一个内核线程。

协程的实现原理即n:m多路复用系统线程。

## Q6 如何高效地拼接字符串

Go 语言中，字符串是只读的，也就意味着每次修改操作都会创建一个新的字符串。如果需要拼接多次，应使用 strings.Builder，最小化内存拷贝次数。

var str strings.Builder  
for i := 0; i < 1000; i++ {  
 str.WriteString("a")  
}  
fmt.Println(str.String())

## Q7 什么是 rune 类型

ASCII 码只需要 7 bit 就可以完整地表示，但只能表示英文字母在内的128个字符，为了表示世界上大部分的文字系统，发明了 Unicode， 它是ASCII的超集，包含世界上书写系统中存在的所有字符，并为每个代码分配一个标准编号（称为Unicode CodePoint），在 Go 语言中称之为 rune，是 int32 类型的别名。

Go 语言中，字符串的底层表示是 byte (8 bit) 序列，而非 rune (32 bit) 序列。例如下面的例子中 语 和 言 使用 UTF-8 编码后各占 3 个 byte，因此 len("Go语言") 等于 8，当然我们也可以将字符串转换为 rune 序列。

fmt.Println(len("Go语言")) // 8  
fmt.Println(len([]rune("Go语言"))) // 4

## Q8 如何判断 map 中是否包含某个 key ？

if val, ok := dict["foo"]; ok {  
}

dict["foo"] 有 2 个返回值，val 和 ok，如果 ok 等于 true，则说明 dict 包含 key "foo"，val 将被赋予 "foo" 对应的值。

## Q9 Go 支持默认参数或可选参数吗？

Go 语言不支持可选参数（python 支持），也不支持方法重载（java支持）。

## Q10 defer 的执行顺序

多个 defer 语句，遵从后进先出(Last In First Out，LIFO)的原则，最后声明的 defer 语句，最先得到执行。

defer 在 return 语句之后执行，但在函数退出之前，defer 可以修改返回值。

func test() int {  
 i := 0  
 defer func() {  
 fmt.Println("defer1")  
 }()  
 defer func() {  
 i += 1  
 fmt.Println("defer2")  
 }()  
 return i  
}  
  
func main() {  
 fmt.Println("return", test())  
}  
// defer2  
// defer1  
// return 0

这个例子中，可以看到 defer 的执行顺序：后进先出。但是返回值并没有被修改，这是由于 Go 的返回机制决定的，执行 return 语句后，Go 会创建一个临时变量保存返回值，因此，defer 语句修改了局部变量 i，并没有修改返回值。那如果是有名的返回值呢？

## Q11 如何交换 2 个变量的值？

a, b := "A", "B"  
a, b = b, a  
fmt.Println(a, b) // B A

## Q12 Go 语言 tag 的用处？

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。

package main  
  
import "fmt"  
import "encoding/json"  
  
type Stu struct {  
 Name string `json:"stu\_name"`  
 ID string `json:"stu\_id"`  
 Age int `json:"-"`  
}  
  
func main() {  
 buf, \_ := json.Marshal(Stu{"Tom", "t001", 18})  
 fmt.Printf("%s\n", buf)  
}

这个例子使用 tag 定义了结构体字段与 json 字段的转换关系，Name -> stu\_name, ID -> stu\_id，忽略 Age 字段。很方便地实现了 Go 结构体与不同规范的 json 文本之间的转换。

## Q13 如何判断 2 个字符串切片（slice) 是相等的？

go 语言中可以使用反射 reflect.DeepEqual(a, b) 判断 a、b 两个切片是否相等，但是通常不推荐这么做，使用反射非常影响性能。

通常采用的方式如下，遍历比较切片中的每一个元素（注意处理越界的情况）。

func StringSliceEqualBCE(a, b []string) bool {  
 if len(a) != len(b) {  
 return false  
 }  
  
 if (a == nil) != (b == nil) {  
 return false  
 }  
  
 b = b[:len(a)]  
 for i, v := range a {  
 if v != b[i] {  
 return false  
 }  
 }  
  
 return true  
}

## Q14 字符串打印时，%v 和 %+v 的区别

%v 和 %+v 都可以用来打印 struct 的值，区别在于 %v 仅打印各个字段的值，%+v 还会打印各个字段的名称。

type Stu struct {  
 Name string  
}  
  
func main() {  
 fmt.Printf("%v\n", Stu{"Tom"}) // {Tom}  
 fmt.Printf("%+v\n", Stu{"Tom"}) // {Name:Tom}  
}

但如果结构体定义了 String() 方法，%v 和 %+v 都会调用 String() 覆盖默认值。

## Q15 Go 语言中如何表示枚举值(enums)

通常使用常量(const) 来表示枚举值。

type StuType int32  
  
const (  
 Type1 StuType = iota  
 Type2  
 Type3  
 Type4  
)  
  
func main() {  
 fmt.Println(Type1, Type2, Type3, Type4) // 0, 1, 2, 3  
}

## Q16 空 struct{} 的用途

使用空结构体 struct{} 可以节省内存，一般作为占位符使用，表明这里并不需要一个值。

fmt.Println(unsafe.Sizeof(struct{}{})) // 0

比如使用 map 表示集合时，只关注 key，value 可以使用 struct{} 作为占位符。如果使用其他类型作为占位符，例如 int，bool，不仅浪费了内存，而且容易引起歧义。

type Set map[string]struct{}  
  
func main() {  
 set := make(Set)  
  
 for \_, item := range []string{"A", "A", "B", "C"} {  
 set[item] = struct{}{}  
 }  
 fmt.Println(len(set)) // 3  
 if \_, ok := set["A"]; ok {  
 fmt.Println("A exists") // A exists  
 }  
}

再比如，使用信道(channel)控制并发时，我们只是需要一个信号，但并不需要传递值，这个时候，也可以使用 struct{} 代替。

func main() {  
 ch := make(chan struct{}, 1)  
 go func() {  
 <-ch  
 // do something  
 }()  
 ch <- struct{}{}  
 // ...  
}

再比如，声明只包含方法的结构体。

type Lamp struct{}  
  
func (l Lamp) On() {  
 println("On")  
  
}  
func (l Lamp) Off() {  
 println("Off")  
}

# 实现原理

## Q1 init() 函数是什么时候执行的？

init() 函数是 Go 程序初始化的一部分。Go 程序初始化先于 main 函数，由 runtime 初始化每个导入的包，初始化顺序不是按照从上到下的导入顺序，而是按照解析的依赖关系，没有依赖的包最先初始化。

每个包首先初始化包作用域的常量和变量（常量优先于变量），然后执行包的 init() 函数。同一个包，甚至是同一个源文件可以有多个 init() 函数。init() 函数没有入参和返回值，不能被其他函数调用，同一个包内多个 init() 函数的执行顺序不作保证。

一句话总结： import –> const –> var –> init() –> main()

package main  
  
import "fmt"  
  
func init() {  
 fmt.Println("init1:", a)  
}  
  
func init() {  
 fmt.Println("init2:", a)  
}  
  
var a = 10  
const b = 100  
  
func main() {  
 fmt.Println("main:", a)  
}  
// 执行结果  
// init1: 10  
// init2: 10  
// main: 10

## Q2 Go 语言的局部变量分配在栈上还是堆上？

由编译器决定。Go 语言编译器会自动决定把一个变量放在栈还是放在堆，编译器会做逃逸分析(escape analysis)，当发现变量的作用域没有超出函数范围，就可以在栈上，反之则必须分配在堆上。

func foo() \*int {  
 v := 11  
 return &v  
}  
  
func main() {  
 m := foo()  
 println(\*m) // 11  
}

foo() 函数中，如果 v 分配在栈上，foo 函数返回时，&v 就不存在了，但是这段函数是能够正常运行的。Go 编译器发现 v 的引用脱离了 foo 的作用域，会将其分配在堆上。因此，main 函数中仍能够正常访问该值。

## Q3 2 个 interface 可以比较吗？

如果两个接口值的动态类型一致，但对应的动态值是不可比较的（比如slice），那么程序会崩溃。

Go 语言中，interface 的内部实现包含了 2 个字段，类型 T 和 值 V，interface 可以使用 == 或 != 比较。2 个 interface 相等有以下 2 种情况：

* 两个 interface 均等于 nil（此时 V 和 T 都处于 unset 状态）
* 类型 T 相同，且对应的值 V 相等。

看下面的例子：

type Stu struct {  
 Name string  
}  
  
type StuInt interface{}  
  
func main() {  
 var stu1, stu2 StuInt = &Stu{"Tom"}, &Stu{"Tom"}  
 var stu3, stu4 StuInt = Stu{"Tom"}, Stu{"Tom"}  
 fmt.Println(stu1 == stu2) // false  
 fmt.Println(stu3 == stu4) // true  
}

stu1 和 stu2 对应的类型是 \*Stu，值是 Stu 结构体的地址，两个地址不同，因此结果为 false。  
 stu3 和 stu4 对应的类型是 Stu，值是 Stu 结构体，且各字段相等，因此结果为 true。

## Q4 两个 nil 可能不相等吗？

可能。

接口(interface) 是对非接口值(例如指针，struct等)的封装，内部实现包含 2 个字段，类型 T 和 值 V。一个接口等于 nil，当且仅当 T 和 V 处于 unset 状态（T=nil，V is unset）。

* 两个接口值比较时，会先比较 T，再比较 V。
* 接口值与非接口值比较时，会先将非接口值尝试转换为接口值，再比较。

func main() {  
 var p \*int = nil  
 var i interface{} = p  
 fmt.Println(i == p) // true  
 fmt.Println(p == nil) // true  
 fmt.Println(i == nil) // false  
}

上面这个例子中，将一个 nil 非接口值 p 赋值给接口 i，此时，i 的内部字段为(T=\*int, V=nil)，i 与 p 作比较时，将 p 转换为接口后再比较，因此 i == p，p 与 nil 比较，直接比较值，所以 p == nil。

但是当 i 与 nil 比较时，会将 nil 转换为接口 (T=nil, V=nil)，与i (T=\*int, V=nil) 不相等，因此 i != nil。因此 V 为 nil ，但 T 不为 nil 的接口不等于 nil。

## Q5 简述 Go 语言GC(垃圾回收)的工作原理

最常见的垃圾回收算法有标记清除(Mark-Sweep) 和引用计数(Reference Count)，Go 语言采用的是标记清除算法。并在此基础上使用了三色标记法和写屏障技术，提高了效率。

标记清除收集器是跟踪式垃圾收集器，其执行过程可以分成标记（Mark）和清除（Sweep）两个阶段：

* 标记阶段 — 从根对象出发查找并标记堆中所有存活的对象；
* 清除阶段 — 遍历堆中的全部对象，回收未被标记的垃圾对象并将回收的内存加入空闲链表。

标记清除算法的一大问题是在标记期间，需要暂停程序（Stop the world，STW），标记结束之后，用户程序才可以继续执行。为了能够异步执行，减少 STW 的时间，Go 语言采用了三色标记法。

三色标记算法将程序中的对象分成白色、黑色和灰色三类。

* 白色：不确定对象。
* 灰色：存活对象，子对象待处理。
* 黑色：存活对象。

标记开始时，所有对象加入白色集合（这一步需 STW ）。首先将根对象标记为灰色，加入灰色集合，垃圾搜集器取出一个灰色对象，将其标记为黑色，并将其指向的对象标记为灰色，加入灰色集合。重复这个过程，直到灰色集合为空为止，标记阶段结束。那么白色对象即可需要清理的对象，而黑色对象均为根可达的对象，不能被清理。

三色标记法因为多了一个白色的状态来存放不确定对象，所以后续的标记阶段可以并发地执行。当然并发执行的代价是可能会造成一些遗漏，因为那些早先被标记为黑色的对象可能目前已经是不可达的了。所以三色标记法是一个 false negative（假阴性）的算法。

三色标记法并发执行仍存在一个问题，即在 GC 过程中，对象指针发生了改变。比如下面的例子：

A(黑) -> B (灰) -> C (白) -> D (白)

正常情况下，D 对象最终会被标记为黑色，不应被回收。但在标记和用户程序并发执行过程中，用户程序删除了 C 对 D 的引用，而 A 获得了 D 的引用。标记继续进行，D 就没有机会被标记为黑色了（A 已经处理过，这一轮不会再被处理）。

A (黑) -> B (灰) -> C (白)   
 ↓  
 D (白)

为了解决这个问题，Go 使用了内存屏障技术，它是在用户程序读取对象、创建新对象以及更新对象指针时执行的一段代码，类似于一个钩子。垃圾收集器使用了写屏障（Write Barrier）技术，当对象新增或更新时，会将其着色为灰色。这样即使与用户程序并发执行，对象的引用发生改变时，垃圾收集器也能正确处理了。

一次完整的 GC 分为四个阶段：

1）标记准备(Mark Setup，需 STW)，打开写屏障(Write Barrier)

2）使用三色标记法标记（Marking, 并发）

3）标记结束(Mark Termination，需 STW)，关闭写屏障。

4）清理(Sweeping, 并发)

## Q6 函数返回局部变量的指针是否安全？

这在 Go 中是安全的，Go 编译器将会对每个局部变量进行逃逸分析。如果发现局部变量的作用域超出该函数，则不会将内存分配在栈上，而是分配在堆上。

## Q7 非接口的任意类型 T() 都能够调用 \*T 的方法吗？反过来呢？

* 一个T类型的值可以调用为\*T类型声明的方法，但是仅当此T的值是可寻址(addressable) 的情况下。编译器在调用指针属主方法前，会自动取此T值的地址。因为不是任何T值都是可寻址的，所以并非任何T值都能够调用为类型\*T声明的方法。
* 反过来，一个\*T类型的值可以调用为类型T声明的方法，这是因为解引用指针总是合法的。事实上，你可以认为对于每一个为类型 T 声明的方法，编译器都会为类型\*T自动隐式声明一个同名和同签名的方法。

哪些值是不可寻址的呢？

* 字符串中的字节；
* map 对象中的元素（slice 对象中的元素是可寻址的，slice的底层是数组）；
* 常量；
* 包级别的函数等。

举一个例子，定义类型 T，并为类型 \*T 声明一个方法 hello()，变量 t1 可以调用该方法，但是常量 t2 调用该方法时，会产生编译错误。

type T string  
  
func (t \*T) hello() {  
 fmt.Println("hello")  
}  
  
func main() {  
 var t1 T = "ABC"  
 t1.hello() // hello  
 const t2 T = "ABC"  
 t2.hello() // error: cannot call pointer method on t  
}

# 并发编程

## Q1 无缓冲的 channel 和 有缓冲的 channel 的区别？

对于无缓冲的 channel，发送方将阻塞该信道，直到接收方从该信道接收到数据为止，而接收方也将阻塞该信道，直到发送方将数据发送到该信道中为止。

对于有缓存的 channel，发送方在没有空插槽（缓冲区使用完）的情况下阻塞，而接收方在信道为空的情况下阻塞。

func main() {  
 st := time.Now()  
 ch := make(chan bool)  
 go func () {  
 time.Sleep(time.Second \* 2)  
 <-ch  
 }()  
 ch <- true // 无缓冲，发送方阻塞直到接收方接收到数据。  
 fmt.Printf("cost %.1f s\n", time.Now().Sub(st).Seconds())  
 time.Sleep(time.Second \* 5)  
}

func main() {  
 st := time.Now()  
 ch := make(chan bool, 2)  
 go func () {  
 time.Sleep(time.Second \* 2)  
 <-ch  
 }()  
 ch <- true  
 ch <- true // 缓冲区为 2，发送方不阻塞，继续往下执行  
 fmt.Printf("cost %.1f s\n", time.Now().Sub(st).Seconds()) // cost 0.0 s  
 ch <- true // 缓冲区使用完，发送方阻塞，2s 后接收方接收到数据，释放一个插槽，继续往下执行  
 fmt.Printf("cost %.1f s\n", time.Now().Sub(st).Seconds()) // cost 2.0 s  
 time.Sleep(time.Second \* 5)  
}

## Q2 什么是协程泄露(Goroutine Leak)？

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。常见的导致协程泄露的场景有以下几种：

* 缺少接收器，导致发送阻塞

这个例子中，每执行一次 query，则启动1000个协程向信道 ch 发送数字 0，但只接收了一次，导致 999 个协程被阻塞，不能退出。

func query() int {  
 ch := make(chan int)  
 for i := 0; i < 1000; i++ {  
 go func() { ch <- 0 }()  
 }  
 return <-ch  
}  
  
func main() {  
 for i := 0; i < 4; i++ {  
 query()  
 fmt.Printf("goroutines: %d\n", runtime.NumGoroutine())  
 }  
}  
// goroutines: 1001  
// goroutines: 2000  
// goroutines: 2999  
// goroutines: 3998

* 缺少发送器，导致接收阻塞

那同样的，如果启动 1000 个协程接收信道的信息，但信道并不会发送那么多次的信息，也会导致接收协程被阻塞，不能退出。

* 死锁(dead lock)

两个或两个以上的协程在执行过程中，由于竞争资源或者由于彼此通信而造成阻塞，这种情况下，也会导致协程被阻塞，不能退出。

* 无限循环(infinite loops)

这个例子中，为了避免网络等问题，采用了无限重试的方式，发送 HTTP 请求，直到获取到数据。那如果 HTTP 服务宕机，永远不可达，导致协程不能退出，发生泄漏。

func request(url string, wg sync.WaitGroup) {  
 for {  
 if \_, err := http.Get(url); err == nil {  
 // write to db  
 break  
 }  
 time.Sleep(time.Second)  
 }  
 wg.Done()  
}  
  
func main() {  
 var wg sync.WaitGroup  
 for i := 0; i < 1000; i++ {  
 wg.Add(1)  
 go request(fmt.Sprintf("exampe.com/%d", i), wg)  
 }  
 wg.Wait()  
}

## Q3 Go 可以限制运行时操作系统线程的数量吗？

可以使用环境变量 GOMAXPROCS 或 runtime.GOMAXPROCS(num int) 设置，例如：

runtime.GOMAXPROCS(1) // 限制同时执行Go代码的操作系统线程数为 1

从官方文档的解释可以看到，GOMAXPROCS 限制的是同时执行用户态 Go 代码的操作系统线程的数量，但是对于被系统调用阻塞的线程数量是没有限制的。GOMAXPROCS 的默认值等于 CPU 的逻辑核数，同一时间，一个核只能绑定一个线程，然后运行被调度的协程。因此对于 CPU 密集型的任务，若该值过大，例如设置为 CPU 逻辑核数的 2 倍，会增加线程切换的开销，降低性能。对于 I/O 密集型应用，适当地调大该值，可以提高 I/O 吞吐率。

# Go语言代码编写规范

## 注意事项

### 文件名字不要轻易以\_\_test.go为结尾

Golang的source文件的命名和其他语言本无差别，但是Golang自带Unit test，它的unit test有个小规范：所有unit test文件都要以\_\_test.go为结尾！  
所以，当你命名一个非unit test文件为XXX\_test.go，而且执意要编译时，就会报错：no buildable Go source files in XXXXXX(你的文件路径)。  
 所以，切记，以\_\_test.go为结尾的都是unit test的文件，且切记不要把unit test文件和普通Go文件放到一起，一定要把unit test文件集体放到一个目录中，否则会编译不过的。

### Go中没有继承

Go中没有继承！没有继承！Go中是叫组合！是组合！

import "fmt"

type student struct{

Name string

Age int

}

func (p \*student) love(){

fmt.Println("love")

}

func (p \*student) like(){

fmt.Println("like first")

p.love()

}

type boy struct {

student

}

func (b \* boy) love(){

fmt.Println("hate")

}

func main(){

b := boy{}

b.like()

}

### 参数为函数

不管运行顺序如何，当参数为函数的时候，要先计算参数的值。

func main(){

a := 1

defer print(function(a))

a = 2;

}

func function(num int) int{

return num

}

func print(num int){

fmt.Println(num)

}

//输出1（在将defer推入栈时，已经将function(a)执行了）

### 注意是struct的函数，还是\* struct的函数

import "fmt"

type people interface {

speak()

}

type student struct{

name string

age int

}

func (stu \*student) speak(){

fmt.Println("I am a student, I am ", stu.age)

}

func main(){

var p people

p = student{name:"RyuGou", age:12} //应该改为 p = &student{name:"RyuGou", age:12}

p.speak()

}

编译报错：

cannot use student literal (type student) as type people in assignment:

student does not implement people (speak method has pointer receiver)

### Go语言中不存在未初始化的变量

var 变量名字 类型 = 表达式

其中类型和表达式均可省略，如果初始化表达式被省略，将用零值初始化该变量。

* 数值变量对应的是0值
* 布尔变量对应的是false
* 字符串对应的零值是空字符串
* 接口或者引用类型（包括slice，map，chan）变量对应的是nil
* 数组或者结构体等聚合类型对应的零值是每个元素或字段对应该类型的零值。

### :=注意的问题

使用:=定义的变量，仅能使用在函数内部。

在定义多个变量的时候:=周围不一定是全部都是刚刚声明的，有些可能只是赋值，例如下面的err变量

in, err := os.Open(infile)

// TODO

out, err := os.Create(outfile)

### new

new在Go语言中只是一个预定义的函数，它并不是一个关键字，我们可以将new作为变量或者其他。（不建议）

func delta(old, new int) int {

return new - old

}

### 并不是使用new就一定会在堆上分配内存

编译器会自动选择在栈上还是在堆上分配存储空间，但可能令人惊讶的是，这个选择并不是由用var还是new声明变量的方式决定的。

请看例子:

var global \*int

func f() {

var x int x=1

global = &x

}

func g() {

y := new(int)

\*y = 1

}

f()函数中的x就是在堆上分配内存，而g()函数中的y就是分配在栈上。

### init函数在同一个文件中可以包含多个

在同一个包文件中，可以包含有多个init函数，多个init函数的执行顺序和定义顺序一致。

### Golang中没有“对象”

package main

import (

"fmt"

)

type test struct {

name string

}

func (t \*test) getName(){

fmt.Println("hello world")

}

func main() {

var t \*test

t = nil

t.getName()

}

可以正常输出。Go本质上不是面向对象的语言，Go中是不存在object的含义的，Go语言书籍中的对象也和Java、PHP中的对象有区别，不是真正的”对象”，是Go中struct的实体。（C++中，这里也可以正常输出）

调用getName方法，在Go中还可以转换，转换为：Type.method(t Type, arguments)  
所以，以上代码main函数中还可以写成：

func main() {

(\*test).getName(nil)

}

### 指针\*符号

&的意思大家都明白的，取地址，假如你想获得一个变量的地址，只需在变量前加上&即可。

a := 1

b := &a

现在，我拿到a的地址了，但是我想取得a指针指向的值，该如何操作呢？用\*号，\*b即可。  
\*的意思是对指针取值。

下面对a的值加一：

a := 1

b := &a

\*b++

\*和&可以相互抵消，同时注意，\*&可以抵消，但是&\*不可以；所以a和\*&a是一样的，和\*&\*&\*&a也是一样的。

### os.Args

os.Args获取命令行指令参数，应该从数组的1坐标开始。

os.Args的第一个元素，os.Args[0], 是命令本身的名字。

package main

import (

"fmt"

"os"

)

func main() {

fmt.Println(os.Args[0])

}

以上代码，经过go build之后，打包成一个可执行文件main，然后运行指令./main 123

输出：./main

## 代码规范

* 模块名全部使用小写，不得包含下划线 \_
* 非必须的导出的标识符，全部小写开头
* 线上非启动过程中的代码不能有fmt.Printf系列的函数调用
* 函数的参数个数不超过5个
* 返回值超过2个以上的，尽量建议使用命名的返回值
* If 条件变量里面超过3个以上条件，需要每行换行 并且把 || &&这些运算符放在每一行的最后

## 命名规范

命名是代码规范中很重要的一部分，统一的命名规则有利于提高的代码的可读性，好的命名仅仅通过命名就可以获取到足够多的信息

* 当命名（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）
* 命名如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 private）

### 1. 包命名：package

保持package的名字和目录保持一致，尽量采取有意义的包名，简短，有意义，尽量和标准库不要冲突。包名应该为小写单词，不要使用下划线或者混合大小写

**包命名请不要使用golang内置包名，以及可能涉及的知名第三方包的包名**

package comet

package main

// 不使用内置包名，如：log/time/sort/context...

// 不适用知名第三方包名，特别是我们项目可能引用的，如：redis/protobuf/gin/beego/mysql/...

### 2. 文件命名

尽量采取有意义的文件名，简短，有意义，应该为小写单词，使用下划线分隔各个单词

my\_test.go

### 3. 结构体命名

采用驼峰命名法，首字母根据访问控制大写或者小写。struct 申明和初始化格式采用多行，比如：

// 多行声明

type User struct{

Username string

Email string

}

// 多行初始化

u := User{

Username: "tom",

Email: "tom@gmail.com",

}

### 4. 接口命名

命名规则基本和上面的结构体类型。单个函数的结构名以 “er” 作为后缀，例如 Reader, Writer

type Reader interface {

Read(p []byte) (n int, err error)

}

### 5. 变量命名

* 和结构体类似，变量名称一般遵循驼峰法，首字母根据访问控制原则大写或者小写，但遇到特有名词时，需要遵循以下规则：
* 如果变量为私有，且特有名词为首个单词，则使用小写，如 apiClient
* 其它情况都应当使用该名词原有的写法，如 APIClient、repoID、UserID
* 错误示例：UrlArray，应该写成 urlArray 或者 URLArray
* 若变量类型为 bool 类型，则名称应以 Has, Is, Can 或 Allow 开头

var isExist bool

var hasConflict bool

var canManage bool

var allowGitHook bool

### 6. 常量命名

如果是枚举类型的常量，需要先创建相应类型：

type Scheme string

const (

HTTP Scheme = "http"

HTTPS Scheme = "https"

)

## 注释规范

### 1. 注释规范

* 全部使用单行注释且//后面需要一个空格（通过设置IDE实现），即 // xxxx
* 统一使用中文注释，对于中英文字符之间严格使用空格分隔，如:

// 从 Redis 中批量读取属性，对于没有读取到的 id ， 记录到一个数组里面，准备从 DB 中读取

### 2. 包注释

模块的最小单位是包，所以只需要包注释，不需要文件注释。

每个包都应该有一个包注释，一个位于package子句之前的行注释。包如果有多个go文件，只需要出现在一个go文件中（一般是和包同名的文件）即可。 包注释应该包含下面基本信息：

// @Title 简单描述

// @Description 详细描述

// @Author 创建者 创建时间

// @Update 修改者 修改时间

### 3. 结构体（接口）注释

每个自定义的结构体或者接口都应该有注释说明，该注释对结构进行简要介绍，放在结构体定义的前一行，格式为： 结构体名， 结构体说明。同时结构体内的每个成员变量都要有说明，该说明放在成员变量的后面（注意对齐），实例如下：

// User，用户对象，定义了用户的基础信息

type User struct{

UserName string `description:"用户名称"`

Email string `description:"邮箱"`

}

### 4. 函数（方法）注释

函数名及参数命名需要做到词要达意，让人一看就知道含义

// TestDemo, 函数介绍

// [可选] 逻辑介绍/返回值介绍

// ...

func TestDemo(a, b string) (int64, err) {

// todo

}

### 5. 代码逻辑注释

对于一些关键位置的代码逻辑，或者局部较为复杂的逻辑，需要有相应的逻辑说明，方便其他开发者阅读该段代码，实例如下：

// 从 Redis 中批量读取属性，对于没有读取到的 id ， 记录到一个数组里面，准备从 DB 中读取

xxxxx

xxxxxxx

xxxxxxx

## 代码风格

### 1. 缩进，折行与换行

* 缩进使用tab键缩进
* 不允许折行，即不允许超出IDE设置的编码区间
* 一行只允许一条数据
* 方法（函数）之间换一行，代码模块之间换一行

### 2. 括号与空格

括号和空格方面，也可以直接使用 gofmt 工具格式化（go 会强制左大括号不换行，换行会报语法错误），所有的运算符和操作数之间要留空格

// 正确的方式

if a > 0 {

}

// 错误的方式

if a>0 // a ，0 和 > 之间应该空格

{ // 左大括号不可以换行，会报语法错误

}

### 3. import规范

* import在多行的情况下，如果引入了三种类型的包（标准库包，程序内部包，第三方包），有顺序的引入包，不同的类型采用空格分离，第一种实标准库，第二是项目包，第三是第三方包：

import (

"encoding/json"

"strings"

"myproject/models"

"myproject/controller"

"myproject/utils"

"github.com/astaxie/beego"

"github.com/go-sql-driver/mysql"

)

在项目中不要使用相对路径引入包：

// 这是不好的导入

import “../net”

// 这是正确的做法

import “github.com/repo/proj/src/net”

### 4. 错误处理

* 错误处理的原则就是不能丢弃任何有返回err的调用，不要使用 \_ 丢弃，必须全部处理。接收到错误，要么返回err，或者使用log记录下来
* 尽早return：一旦有错误发生，马上返回
* 尽量不要使用panic，除非你知道你在做什么

// 错误写法

if err != nil {

// error handling

} else {

// normal code

}

// 正确写法

if err != nil {

// error handling

return // or continue, etc.

}

// normal code

### 5. 减少嵌套，去掉不必要的else

代码应通过尽可能先处理错误情况/特殊情况并尽早返回或继续循环来减少嵌套。减少嵌套多个级别的代码的代码量

// Bad

for \_, v := range data {

if v.F1 == 1 {

v = process(v)

if err := v.Call(); err == nil {

v.Send()

} else {

return err

}

} else {

log.Printf("Invalid v: %v", v)

}

}

-------------------------------------------------------------------

// Good

for \_, v := range data {

if v.F1 != 1 {

log.Printf("Invalid v: %v", v)

continue

}

v = process(v)

if err := v.Call(); err != nil {

return err

}

v.Send()

}

### 6. 缩小变量作用域

如果有可能，尽量缩小变量作用范围。除非它与<减少嵌套>的规则冲突，优先考虑代码的简洁性

// Bad

err := ioutil.WriteFile(name, data, 0644)

if err != nil {

return err

}

-------------------------------------------------------------------

// Good

if err := ioutil.WriteFile(name, data, 0644); err != nil {

return err

}

## 指导原则

### 1. 在边界处拷贝 Slices 和 Maps

slices 和 maps 包含了指向底层数据的指针，因此在需要复制它们时要特别注意，特别是读取的配置需要根据用户动态修改时，需要提前做一次拷贝

**特别注意配置中心使用 proto 定义的配置，如有写操作，请务必检查有没有做深拷贝，proto 生成的结构体成员皆为指针，浅拷贝无法完全避免配置值全局篡改问题！！！**

// Bad

func (d \*Driver) SetTrips(trips []Trip) {

d.trips = trips

}

trips := ...

d1.SetTrips(trips)

// 你是要修改 d1.trips 吗？

trips[0] = ...

-------------------------------------------------------------------

// Good

func (d \*Driver) SetTrips(trips []Trip) {

d.trips = make([]Trip, len(trips))

copy(d.trips, trips)

}

trips := ...

d1.SetTrips(trips)

// 这里我们修改 trips[0]，但不会影响到 d1.trips

trips[0] = ...

### 2. 使用 defer 释放资源

使用 defer 释放资源，诸如文件、锁及DB类资源

// Bad

p.Lock()

if p.count < 10 {

p.Unlock()

return p.count

}

p.count++

newCount := p.count

p.Unlock()

return newCount

// 当有多个 return 分支时，很容易遗忘 unlock

-------------------------------------------------------------------

// Good

p.Lock()

defer p.Unlock()

if p.count < 10 {

return p.count

}

p.count++

return p.count

// 更可读

### 3. 枚举从 1 开始

在 Go 中引入枚举的标准方法是声明一个自定义类型和一个使用了 iota 的 const 组。由于变量的默认值为 0，因此通常应以非零值开头枚举

// Bad

type Operation int

const (

Add Operation = iota

Subtract

Multiply

)

// Add=0, Subtract=1, Multiply=2

-------------------------------------------------------------------

// Good

type Operation int

const (

Add Operation = iota + 1

Subtract

Multiply

)

// Add=1, Subtract=2, Multiply=3

### 4. 使用 time 处理时间

* 使用 time.Time 表达瞬时时间
* 在处理时间的瞬间时使用 time.time，在比较、添加或减去时间时使用 time.Time 中的方法

// Bad

func isActive(now, start, stop int) bool {

return start <= now && now < stop

}

-------------------------------------------------------------------

// Good

func isActive(now, start, stop time.Time) bool {

return (start.Before(now) || start.Equal(now)) && now.Before(stop)

}

使用 time.Duration 表达时间段

在处理时间段时使用 time.Duration

// Bad

func poll(delay int) {

for {

// ...

time.Sleep(time.Duration(delay) \* time.Millisecond)

}

}

poll(10) // 是几秒钟还是几毫秒?

-------------------------------------------------------------------

// Good

func poll(delay time.Duration) {

for {

// ...

time.Sleep(delay)

}

}

poll(10\*time.Second)

### 5. 避免可变全局变量

使用选择依赖注入方式避免改变全局变量。 既适用于函数指针又适用于其他值类型

// Bad

// sign.go

var \_timeNow = time.Now

func sign(msg string) string {

now := \_timeNow()

return signWithTime(msg, now)

}

// sign\_test.go

func TestSign(t \*testing.T) {

oldTimeNow := \_timeNow

\_timeNow = func() time.Time {

return someFixedTime

}

defer func() { \_timeNow = oldTimeNow }()

assert.Equal(t, want, sign(give))

}

-------------------------------------------------------------------

// Good

// sign.go

type signer struct {

now func() time.Time

}

func newSigner() \*signer {

return &signer{

now: time.Now,

}

}

func (s \*signer) Sign(msg string) string {

now := s.now()

return signWithTime(msg, now)

}

// sign\_test.go

func TestSigner(t \*testing.T) {

s := newSigner()

s.now = func() time.Time {

return someFixedTime

}

assert.Equal(t, want, s.Sign(give))

}

### 6. 避免使用 init()

尽可能避免使用init()。当init()是不可避免或可取的，代码应先尝试：

* 无论程序环境或调用如何，都要完全确定
* 避免依赖于其他init()函数的顺序或副作用。虽然init()顺序是明确的，但代码可以更改， 因此init()函数之间的关系可能会使代码变得脆弱和容易出错
* 避免访问或操作全局或环境状态，如机器信息、环境变量、工作目录、程序参数/输入等
* 避免I/O，包括文件系统、网络和系统调用

## 最佳实践

### 1. 优先使用 strconv 而不是 fmt

将原语转换为字符串或从字符串转换时，strconv速度比fmt快

// Bad

for i := 0; i < b.N; i++ {

s := fmt.Sprint(rand.Int())

}

-------------------------------------------------------------------

// Good

for i := 0; i < b.N; i++ {

s := strconv.Itoa(rand.Int())

}

### 2. 不要在循环里重复相同的操作

eg: 不要反复从固定字符串创建字节 slice。相反，请执行一次转换并捕获结果

// Bad

for i := 0; i < b.N; i++ {

w.Write([]byte("Hello world"))

}

-------------------------------------------------------------------

// Good

data := []byte("Hello world")

for i := 0; i < b.N; i++ {

w.Write(data)

}

### 3. 指定容器(map，slice)容量，减少内存重新分配频率

尽可能指定容器容量，以便为容器预先分配内存。这将在添加元素时最小化后续分配（通过复制和调整容器大小）

// Bad

for n := 0; n < b.N; n++ {

data := make([]int, 0)

for k := 0; k < size; k++{

data = append(data, k)

}

}

-------------------------------------------------------------------

// Good

for n := 0; n < b.N; n++ {

data := make([]int, 0, size)

for k := 0; k < size; k++{

data = append(data, k)

}

}

# goroutine 调度器

说到“调度”，首先会想到操作系统对进程、线程的调度。操作系统调度器会将系统中的多个线程按照一定算法调度到物理 CPU 上去运行。传统的编程语言比如 C、C++ 等的并发实现实际上就是基于操作系统调度的，即程序负责创建线程，操作系统负责调度。

线程的调度方式相对于进程来说，线程运行所需要资源比较少，在同一进程中进行线程切换效率会高很多。

线程是操作系统调度时的最基本单元，而 Linux 在调度器并不区分进程和线程的调度，只是说线程调度因为资源少，所以切换的效率比较高。

使用多线程编程会遇到以下问题：

* 并发单元间通信困难，易错：多个 thread 之间的通信虽然有多种机制可选，但用起来是相当复杂；并且一旦涉及到共享内存，就会用到各种 lock，一不小心就会出现死锁的情况。
* 对于线程池的大小不好确认，在请求量大的时候容易导致 OOM 的情况
* 虽然线程比较轻量，但是在调度时也有比较大的额外开销。每个线程会都占用 1 兆以上的内存空间，在对线程进行切换时不仅会消耗较多的内存，恢复寄存器中的内容还需要向操作系统申请或者销毁对应的资源，每一次线程上下文的切换仍然需要一定的时间（us 级别）
* 对于很多网络服务程序，由于不能大量创建 thread，就要在少量 thread 里做网络多路复用，例如 JAVA 的Netty 框架，写起这样的程序也不容易。

这便有了“协程”，线程分为内核态线程和用户态线程，用户态线程需要绑定内核态线程，CPU 并不能感知用户态线程的存在，它只知道它在运行1个线程，这个线程实际是内核态线程。

用户态线程实际有个名字叫协程（co-routine），为了容易区分，使用协程指用户态线程，使用线程指内核态线程。

协程跟线程是有区别的，线程由CPU调度是抢占式的，协程由用户态调度是协作式的，一个协程让出 CPU 后，才执行下一个协程。

Go中，协程被称为 goroutine（但其实并不完全是协程，还做了其他方面的优化），它非常轻量，一个 goroutine 只占几 KB，并且这几 KB 就足够 goroutine 运行完，这就能在有限的内存空间内支持大量 goroutine，支持了更多的并发。虽然一个 goroutine 的栈只占几 KB，但实际是可伸缩的，如果需要更多内容，runtime会自动为 goroutine 分配。

而将所有的 goroutines 按照一定算法放到 CPU 上执行的程序就称为 goroutine 调度器或 goroutine scheduler。

不过，一个 Go 程序对于操作系统来说只是一个用户层程序，对于操作系统而言，它的眼中只有 thread，它并不知道有什么叫 Goroutine 的东西的存在。goroutine 的调度全要靠 Go 自己完成，所以就需要 goroutine 调度器来实现 Go 程序内 goroutine 之间的 CPU 资源调度。

在操作系统层面，Thread 竞争的 CPU 资源是真实的物理 CPU，但对于 Go 程序来说，它是一个用户层程序，它本身整体是运行在一个或多个操作系统线程上的，因此 goroutine 们要竞争的所谓 “CPU” 资源就是操作系统线程。这样 Go scheduler 的要做的就是：将 goroutines 按照一定算法放到不同的操作系统线程中去执行。这种在语言层面自带调度器的，称之为原生支持并发。

## goroutine 调度器

调度器的任务是在用户态完成 goroutine 的调度，而调度器的实现好坏，对并发实际有很大的影响。

## M模型

### G——go routinue

理解为一个函数指针，即需要被执行的函数

### M——machine物理线程

理解为线程池物理线程，即实际执行函数的人。

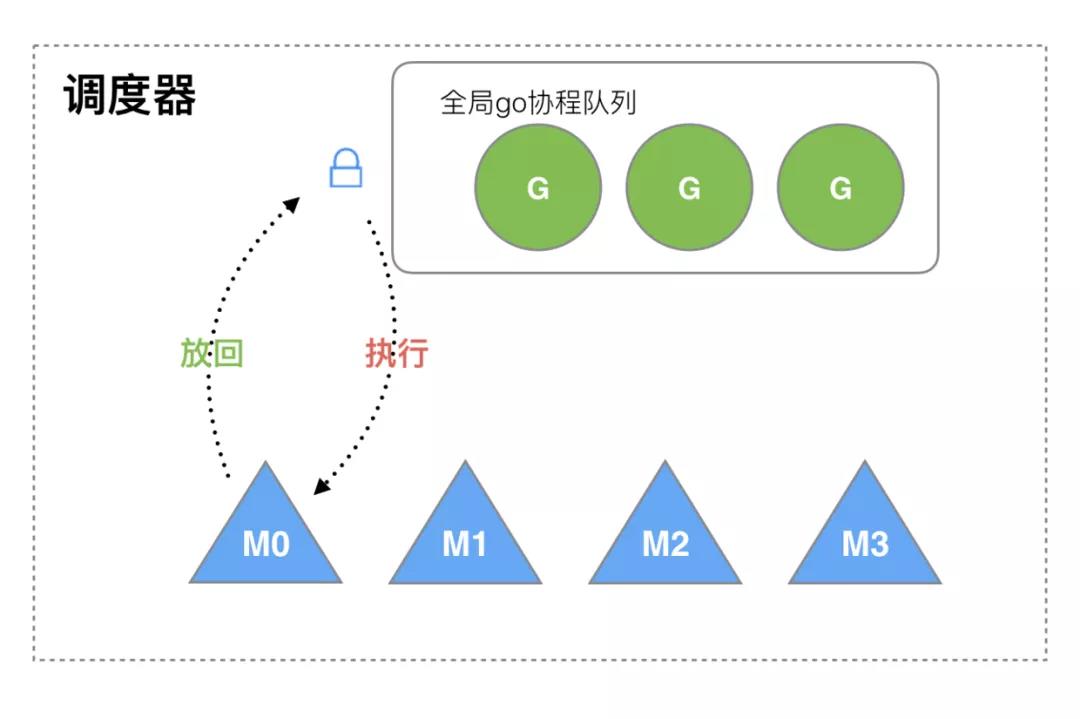
### 模型

**像线程池调度模型，routinue代表的函数交给machine执行。**

现在的 Go语言调度器是 2012 年重新设计的，在这之前的调度器称为老调度器，老调度器采用的是 G-M 模型，在这个调度器中，每个 goroutine 对应于 runtime 中的一个抽象结构：G。

而 os thread 作为物理 CPU 的存在而被抽象为一个结构：M(machine)。

M 想要执行 G、放回 G 都必须访问全局 G 队列，并且 M 有多个，即多线程访问同一资源需要加锁进行保证互斥/同步，所以全局 G 队列是有互斥锁进行保护的。



这个结构虽然简单，但是却存在着许多问题。它限制了 Go 并发程序的伸缩性，尤其是对那些有高吞吐或并行计算需求的服务程序。主要体现在如下几个方面：

* 单一全局互斥锁(Sched.Lock)和集中状态存储的存在导致所有 goroutine 相关操作，比如：创建、重新调度等都要上锁，这会造成激烈的锁竞争
* goroutine 传递问题：M 经常在 M 之间传递可运行的 goroutine，这导致调度延迟增大以及额外的性能损耗
* 每个 M 做内存缓存，导致内存占用过高，数据局部性较差。
* 系统调用导致频繁的线程阻塞和取消阻塞操作增加了系统开销。

所以用了 4 年左右就被替换掉了。

## G-M-P 模型

### P——给M分配

将**全局任务队列拆分到局部，局部包含他自己的任务队列、线程队列**。减小锁的粒度，降低竞争。像个CPU，包含自己的核心数，也有一些投入到这里等待执行的任务。

### 模型

面对之前调度器的问题，Go 设计了新的调度器，并在其中引入了 P（Processor），另外还引入了任务窃取调度的方式（work stealing）。

P：Processor处理器，它包含了运行 goroutine 的资源，如果线程想运行 goroutine，必须先获取 P（CPU），P 中还包含了可运行的 G（go routinue） 队列。

P会绑定一个G队列，一个M队列。G表示等待执行的routinue，M表示可调度的物理线程。P将全局G队列拆分为若干个P内部的G队列。

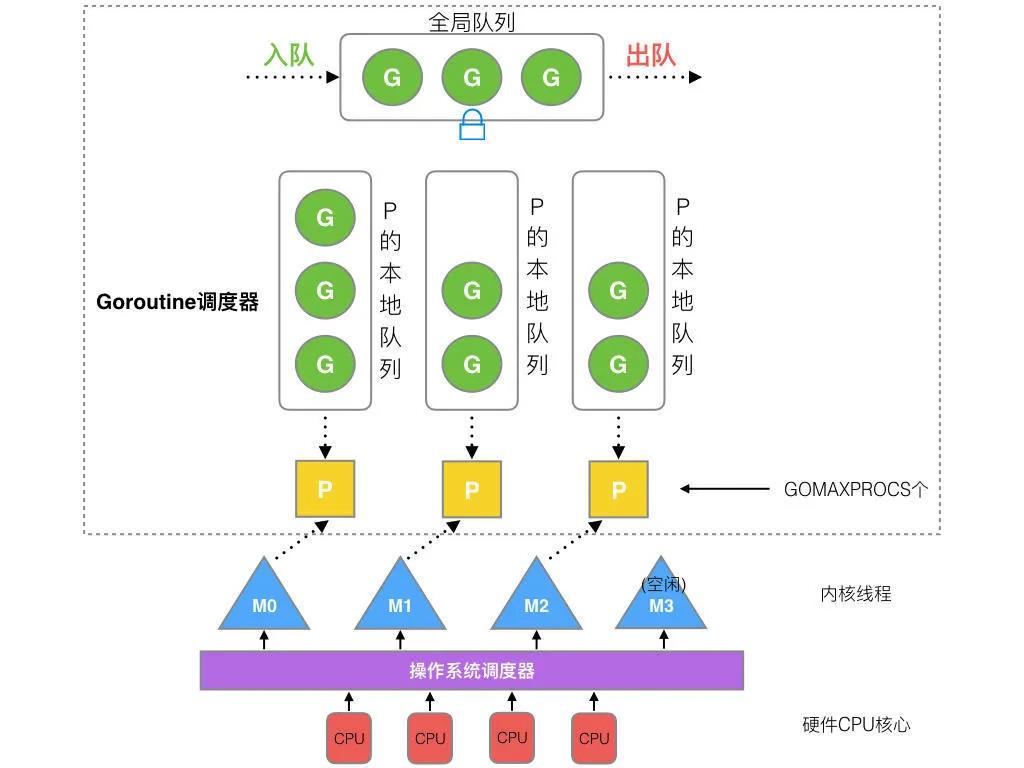
#### work stealing

任务窃取调度。当 M 绑定的 P 没有可运行的 G 时，它可以从其他运行的 M 那里偷取G。

CPU有4核，当它的核全都被调用时，可以去其他CPU偷核。

#### 结构

G-P-M 模型的结构如下：



从上往下是调度器的4个部分：

全局队列（Global Queue）：存放等待运行的 G。

P 的本地队列：同全局队列类似，存放的也是等待运行的G，存的数量有限，不超过256个。新建 G 时，G 优先加入到 P 的本地队列，如果队列满了，则会把本地队列中一半的 G 移动到全局队列。

P列表：所有的P都在程序启动时创建，并保存在数组中，最多有GOMAXPROCS 个。

M：线程想运行任务就得获取 P，从 P 的本地队列获取 G，P 队列为空时，M 也会尝试从全局队列拿一批 G放到 P 的本地队列，或从其他 P 的本地队列偷一半放到自己 P 的本地队列。M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。Goroutine 调度器和 OS 调度器是通过 M 结合起来的，每个 M 都代表了1个内核线程，OS 调度器负责把内核线程分配到 CPU 的核上执行。

### P 的数量

由启动时环境变量 $GOMAXPROCS 或者是由 runtime 的方法 GOMAXPROCS() 决定。这意味着在程序执行的任意时刻都只有 $GOMAXPROCS 个 goroutine 在同时运行。

### M 的数量

* go 语言本身的限制：go 程序启动时，会设置 M 的最大数量，默认 10000。但是内核很难支持这么多的线程数，所以这个限制可以忽略。
* runtime/debug 中的 SetMaxThreads 函数，可以设置 M 的最大数量
* 一个 M 阻塞了，会创建新的 M。

M 与 P 的数量没有绝对关系，一个 M 阻塞，P 就会去创建或者切换另一个 M，所以，即使 P 的默认数量是 1，也有可能会创建很多个 M 出来。

## 抢占式调度

1. P-M 模型中还实现了抢占式调度，所谓抢占式调度指的是在 coroutine 中要等待一个协程主动让出 CPU 才执行下一个协程。
2. 在 Go 中，一个 goroutine 最多占用CPU 10ms，防止其他 goroutine 被饿死，这也是 goroutine 不同于 coroutine 的一个地方。在 goroutine 中先后实现了两种抢占式调度算法，分别是基于协作的方式和基于信号的方式。

### 基于协作的抢占式调度

G-P-M 模型的实现是 Go scheduler 的一大进步，但此时的调度器仍然有一个问题，那就是不支持抢占式调度，导致一旦某个 G 中出现死循环或永久循环的代码逻辑，那么 G 将永久占用分配给它的 P 和 M，位于同一个 P 中的其他 G 将得不到调度，出现“饿死”的情况。当只有一个 P 时(GOMAXPROCS=1)时，整个 Go 程序中的其他 G 都会被饿死。所以后面 Go 设计团队在 Go 1.2 中实现了基于协作的抢占式调度。

这种抢占式调度的原理则是在每个函数或方法的入口，加上一段额外的代码，让 runtime 有机会检查是否需要执行抢占调度。

#### 工作原理

基于协作的抢占式调度的工作原理大致如下：

1. 编译器会在调用函数前插入一个 runtime.morestack 函数
2. Go 语言运行时会在垃圾回收暂停程序、系统监控发现 Goroutine 运行超过 10ms 时发出抢占请求，此时会设置一个 StackPreempt 字段值为 StackPreempt ，标示当前 Goroutine 发出了抢占请求。
3. 当发生函数调用时，可能会执行编译器插入的 runtime.morestack 函数，它调用的 runtime.newstack 会检查 Goroutine 的 stackguard0 字段是否为 StackPreempt
4. 如果 stackguard0 是 StackPreempt，就会触发抢占让出当前线程

这种实现方式虽然增加了运行时的复杂度，但是实现相对简单，也没有带来过多的额外开销，所以在 Go 语言中使用了 10 几个版本。因为这里的抢占是通过编译器插入函数实现的，还是需要函数调用作为入口才能触发抢占，所以这是一种协作式的抢占式调度。这种解决方案只能说局部解决了“饿死”问题，对于没有函数调用，纯算法循环计算的 G，scheduler 依然无法抢占。

### 基于信号的抢占式调度

Go 语言在 1.14 版本中实现了非协作的抢占式调度，在实现的过程中重构已有的逻辑并为 Goroutine 增加新的状态和字段来支持抢占。

#### 工作原理

基于信号的抢占式调度的工作原理大致如下：

1. 程序启动时，在 runtime.sighandler 函数中注册一个 SIGURG 信号的处理函数 runtime.doSigPreempt
2. 在触发垃圾回收的栈扫描时会调用函数 runtime.suspendG 挂起 Goroutine，此时会执行下面的逻辑：

* 将处于运行状态（\_Grunning）的 Goroutine 标记成可以被抢占，即将 Goroutine 的字段 preemptStop 设置成 true；
* 调用 runtime.preemptM函数， 它可以通过 SIGURG 信号向线程发送抢占请求触发抢占；

1. runtime.preemptM 会调用 runtime.signalM 向线程发送信号SIGURG；
2. 操作系统收到信号后会中断正在运行的线程并执行预先在第 1 步注册的信号处理函数 runtime.doSigPreempt；
3. runtime.doSigPreempt 函数会处理抢占信号，获取当前的 SP 和 PC 寄存器并调用 runtime.sigctxt.pushCall；
4. runtime.sigctxt.pushCall 会修改寄存器并在程序回到用户态时执行 runtime.asyncPreempt；
5. 汇编指令 runtime.asyncPreempt 会调用运行时函数 runtime.asyncPreempt2；
6. runtime.asyncPreempt2 会调用 runtime.preemptPark；
7. runtime.preemptPark 会修改当前 Goroutine 的状态到 \_Gpreempted 并调用 runtime.schedule 让当前函数陷入休眠并让出线程，调度器会选择其它的 Goroutine 继续执行

\_Gpreempted 状态表示当前 groutine 由于抢占而被阻塞，没有执行用户代码并且不在运行队列上，等待唤醒。

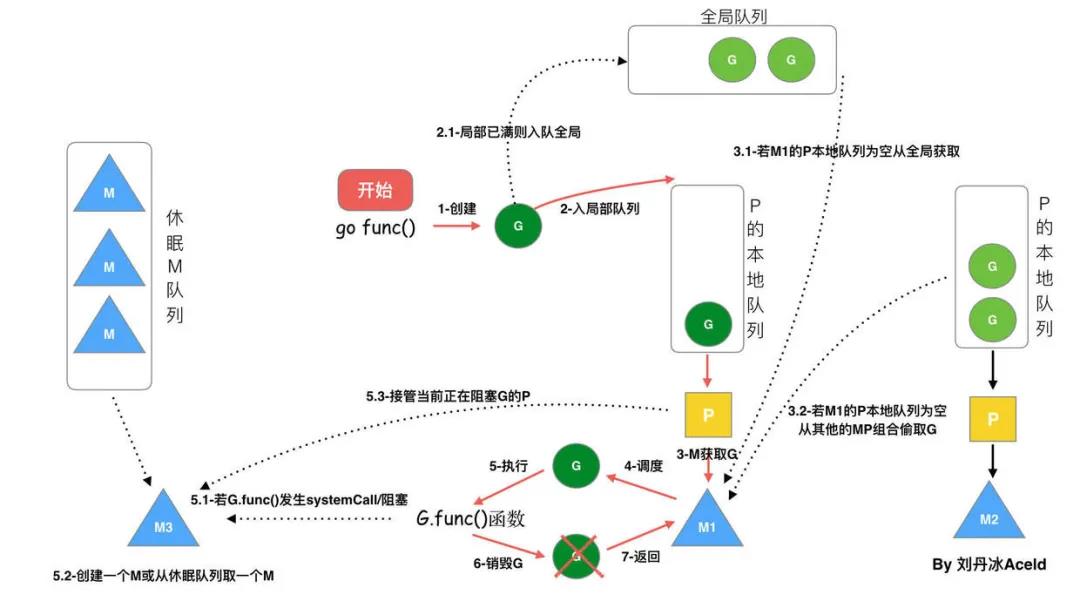
在上面的选择 SIGURG 作为触发异步抢占的信号：

* 该信号需要被调试器透传；
* 该信号不会被内部的 libc 库使用并拦截；
* 该信号可以随意出现并且不触发任何后果；
* 需要处理多个平台上的不同信号；

垃圾回收过程中需要暂停整个程序（Stop the world，STW），有时候可能需要几分钟的时间，这会导致整个程序无法工作。所以 STW 和栈扫描是一个可以抢占的安全点（Safe-points）， Go 语言在这里先加入抢占功能。基于信号的抢占式调度只解决了垃圾回收和栈扫描时存在的问题，它到目前为止没有解决全部问题。

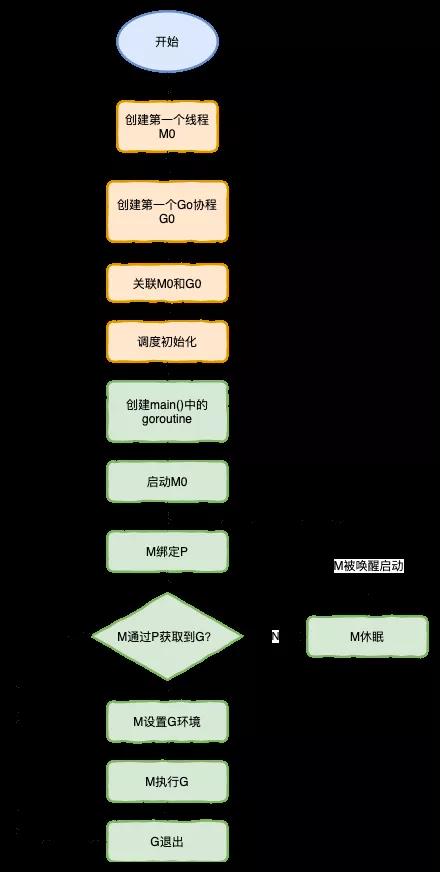
## go func() 调度流程

基于上面的模型，当我们使用 go func() 创建一个新的 goroutine 的时候，其调度流程如下：



1. 通过 go func () 来创建一个 goroutine；
2. 有两个存储 G 的队列，一个是局部调度器 P 的本地队列、一个是全局 G 队列。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. G 只能运行在 M 中，一个 M 必须持有一个 P，M 与 P 是 1：1 的关系。M 会从 P 的本地队列弹出一个可执行状态的 G 来执行，如果 P 的本地队列为空，就会想其他的 MP 组合偷取一个可执行的 G 来执行；
4. 一个 M 调度 G 执行的过程是一个循环机制；
5. 当 M 执行某一个 G 时候如果发生了 syscall 或则其余阻塞操作，M 会阻塞，如果当前有一些 G 在执行，runtime 会把这个线程 M 从 P 中摘除 (detach)，然后再创建一个新的操作系统的线程 (如果有空闲的线程可用就复用空闲线程) 来服务于这个 P；
6. 当 M 系统调用结束时候，这个 G 会尝试获取一个空闲的 P 执行，并放入到这个 P 的本地队列。如果获取不到 P，那么这个线程 M 变成休眠状态， 加入到空闲线程中，然后这个 G 会被放入全局队列中。

## Goroutine 生命周期



在这里有一个线程和一个 groutine 比较特殊，那就是 M0 和 G0：

M0：M0 是启动程序后的编号为 0 的主线程，这个 M 对应的实例会在全局变量 runtime.m0 中，不需要在 heap 上分配，M0 负责执行初始化操作和启动第一个 G， 在之后 M0 就和其他的 M 一样了。

G0 ：G0 是每次启动一个 M 都会第一个创建的 gourtine，G0 仅用于负责调度的 G，G0 不指向任何可执行的函数，每个 M 都会有一个自己的 G0。在调度或系统调用时会使用 G0 的栈空间，全局变量的 G0 是 M0 的 G0。

package main  
import "fmt"  
// main.main  
func main() {  
   fmt.Println("Hello scheduler")  
}

其运行时所经历的过程跟上面的生命周期相对应：

1. runtime 创建最初的线程 m0 和 goroutine g0，并把 2 者关联。
2. 调度器初始化：初始化 m0、栈、垃圾回收，以及创建和初始化由 GOMAXPROCS 个 P 构成的 P 列表。
3. 示例代码中的 main 函数是 main.main，runtime 中也有 1 个 main 函数——runtime.main，代码经过编译后，runtime.main会调用 main.main，程序启动时会为 runtime.main 创建 goroutine，称它为main goroutine，然后把 main goroutine 加入到P的本地队列。

启动 m0，m0 已经绑定了 P，会从 P 的本地队列获取 G，获取到 main goroutine。

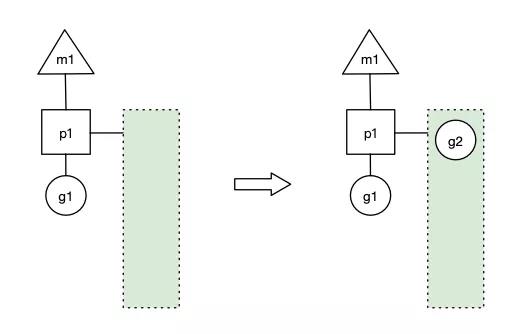
1. G 拥有栈，M 根据 G 中的栈信息和调度信息设置运行环境
2. M 运行 G
3. G 退出，再次回到 M 获取可运行的 G，这样重复下去，直到 main.main 退出，runtime.main执行 Defer 和 Panic 处理，或调用 runtime.exit 退出程序。

调度器的生命周期几乎占满了一个Go程序的一生，runtime.main 的 goroutine 执行之前都是为调度器做准备工作，runtime.main 的 goroutine 运行，才是调度器的真正开始，直到 runtime.main 结束而结束。

## Goroutine 调度器场景分析

### 场景一 本地队列

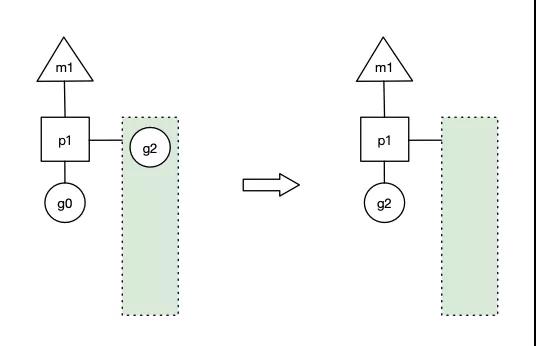
p1 拥有 g1，m1 获取 p1 后开始运行g1，g1 使用 go func() 创建了 g2，为了局部性 g2 优先加入到 p1 的本地队列：



### 场景二 协程切换

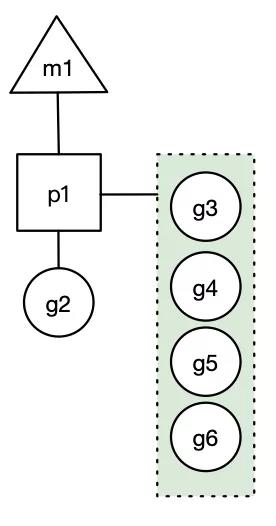
g1运行完成后(函数：goexit)，m 上运行的 goroutine 切换为 g0，g0 负责调度时协程的切换（函数：schedule）。

从 p1 的本地队列取 g2，从 g0 切换到 g2，并开始运行 g2 (函数：execute)。实现了线程 m1 的复用。



### 场景三 本地队列满，扔到全局

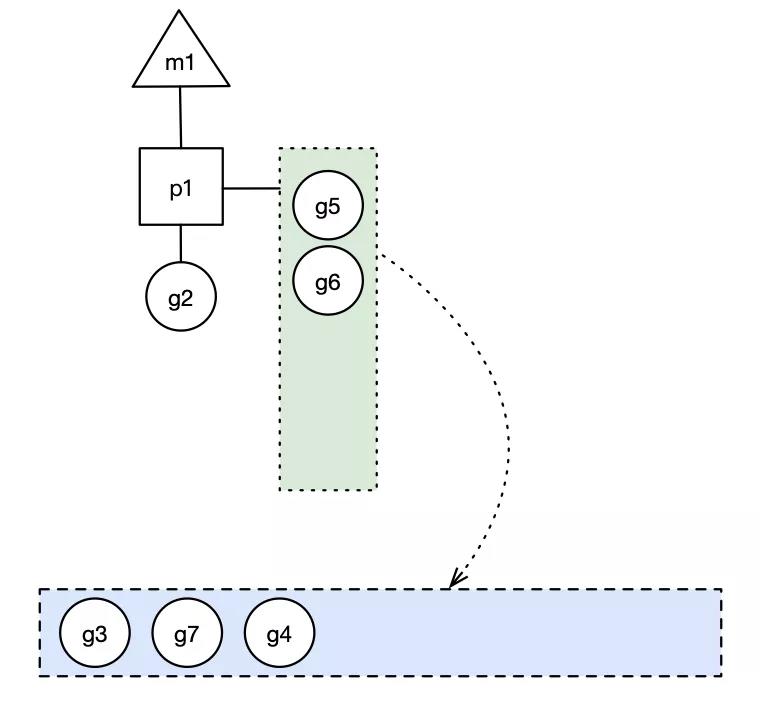
假设每个 p 的本地队列只能存 4 个 g。g2 要创建 6 个 g，前 4 个g（g3, g4, g5, g6）已经加入 p1 的本地队列，p1 本地队列满了。



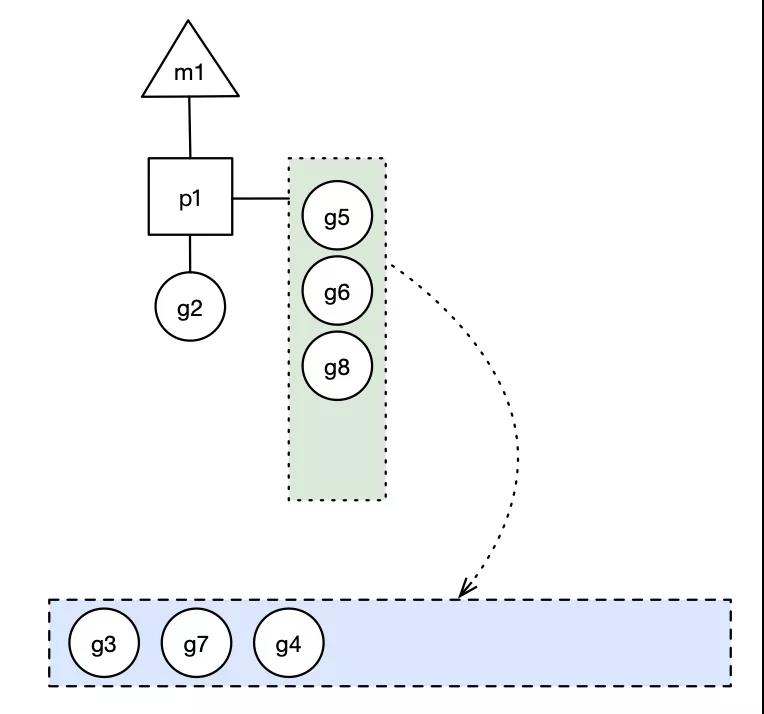
g2 在创建 g7 的时候，发现 p1 的本地队列已满，需要执行负载均衡，把 p1 中本地队列中前一半的 g，还有新创建的 g 转移到全局队列。

实现中并不一定是新的 g，如果 g 是 g2 之后就执行的，会被保存在本地队列，利用某个老的 g 替换新 g 加入全局队列），这些 g 被转移到全局队列时，会被打乱顺序。

所以 g3,g4,g7 被转移到全局队列。

蓝色长方形代表全局队列。

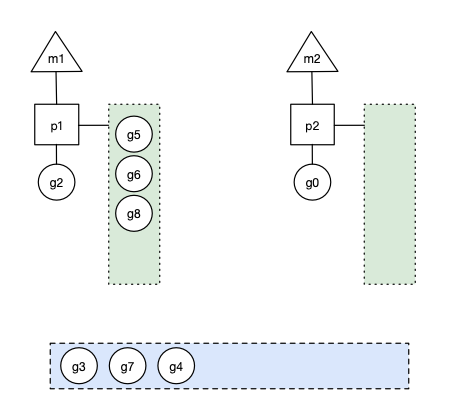
如果此时 G2 创建 G8 时，P1 的本地队列未满，所以 G8 会被加入到 P1 的本地队列：



### 场景四 本地队列空，从全局取

在创建 g 时，运行的 g 会尝试唤醒其他空闲的 p 和 m 组合去执行。

假定 g2 唤醒了 m2，m2 绑定了 p2，并运行 g0，但 p2 本地队列没有 g，m2 此时为自旋线程（没有 G 但为运行状态的线程，不断寻找 g）。

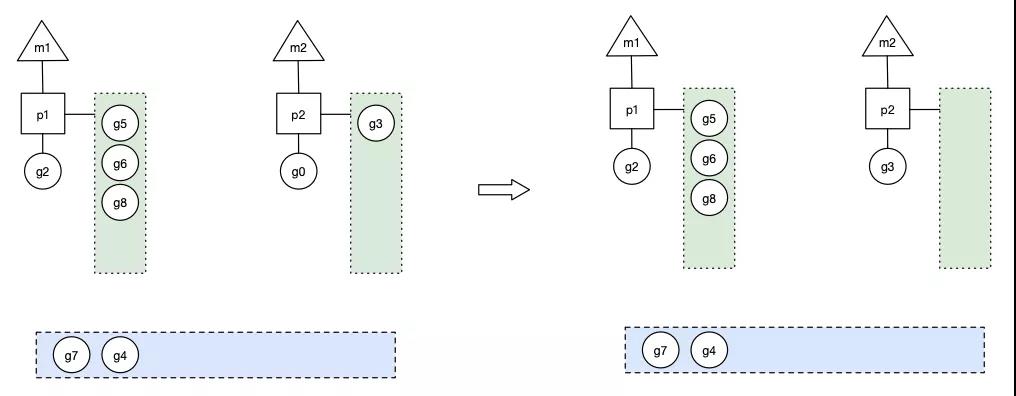


m2 接下来会尝试从全局队列 (GQ) 取一批 g 放到 p2 的本地队列（函数：findrunnable）。m2 从全局队列取的 g 数量符合下面的公式：

n = min(len(GQ)/GOMAXPROCS + 1, len(GQ/2))

公式的含义是，至少从全局队列取 1 个 g，但每次不要从全局队列移动太多的 g 到 p 本地队列，给其他 p 留点。这是从全局队列到 P 本地队列的负载均衡。

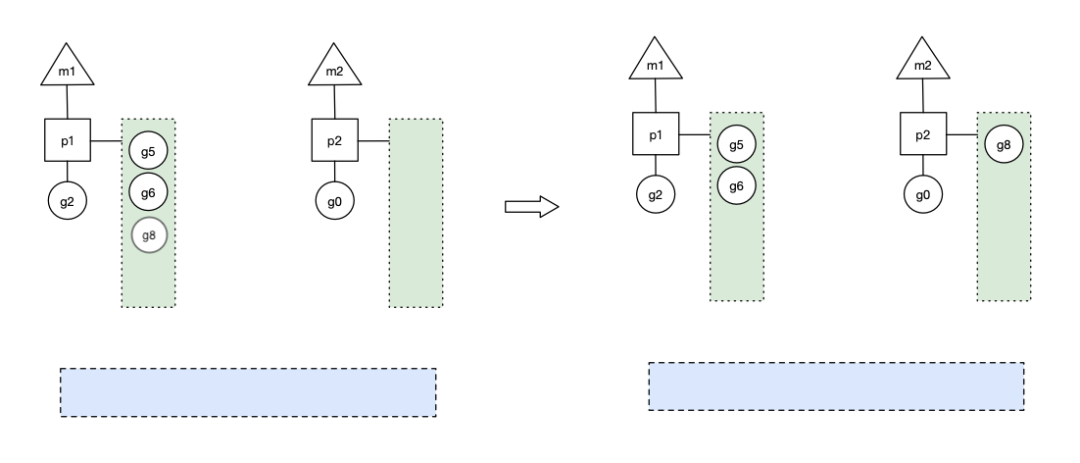
假定场景中一共有 4 个 P（GOMAXPROCS=4），所以 m2 只从能从全局队列取 1 个 g（即 g3）移动 p2 本地队列，然后完成从 g0 到 g3 的切换，运行 g3：



### 场景五 本地全局皆空，偷取

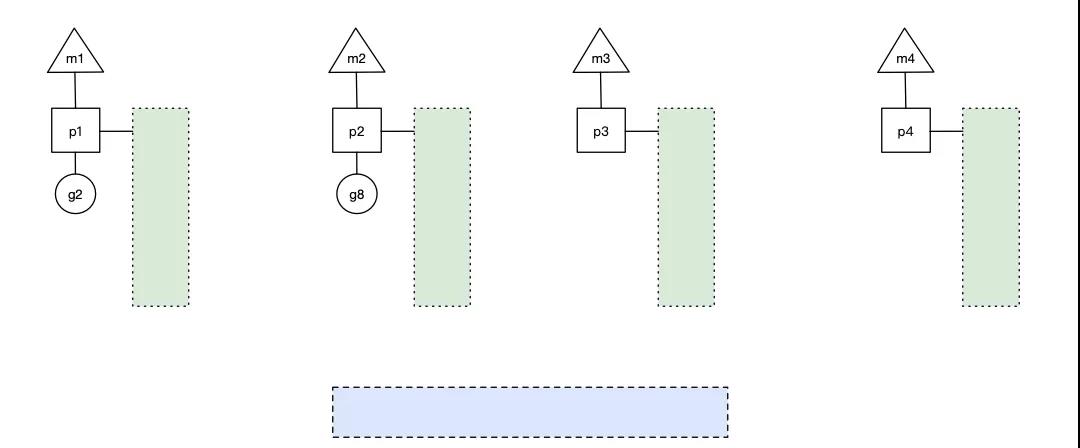
假设 g2 一直在 m1上运行，经过 2 轮后，m2 已经把 g7、g4 也挪到了p2的本地队列并完成运行，全局队列和 p2 的本地队列都空了，如下图左边所示。

全局队列已经没有 g，那 m 就要执行 work stealing：从其他有 g 的 p 哪里偷取一半 g 过来，放到自己的 P 本地队列。p2 从 p1 的本地队列尾部取一半的 g，本例中一半则只有 1 个 g8，放到 p2 的本地队列，情况如下图右边：



### 场景六 M自旋

p1 本地队列 g5、g6 已经被其他 m 偷走并运行完成，当前 m1 和 m2 分别在运行 g2 和 g8，m3 和 m4 没有goroutine 可以运行，m3 和 m4 处于自旋状态，它们不断寻找 goroutine。



这里有一个问题，为什么要让 m3 和 m4 自旋？自旋本质是在运行，线程在运行却没有执行 g，就变成了浪费CPU，销毁线程可以节约CPU资源不是更好吗？

实际上，创建和销毁CPU都是浪费时间的，我们希望当有新 goroutine 创建时，立刻能有 m 运行它，如果销毁再新建就增加了时延，降低了效率。当然也考虑了过多的自旋线程是浪费 CPU，所以系统中最多有 GOMAXPROCS 个自旋的线程，多余的没事做的线程会让他们休眠（函数：notesleep() 实现了这个思路）。

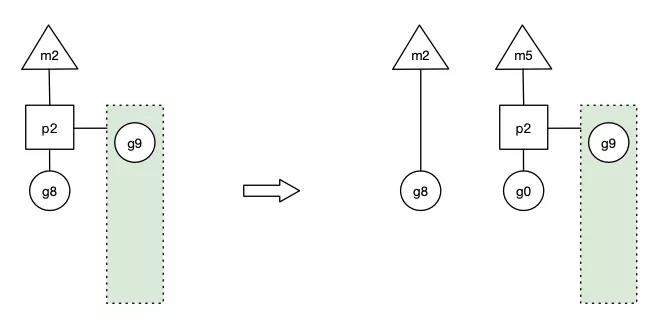
### 场景七 调用阻塞时，M和P解绑

阻塞时，线程释放绑定的 P，把 P 转移给其他空闲的线程执行。

假定当前除了 m3 和 m4 为自旋线程，还有 m5 和 m6 为自旋线程，g8 创建了 g9，g9 会放入本地队列。加入此时g8 进行了阻塞的系统调用，m2 和 p2 立即解绑，p2 会执行以下判断：

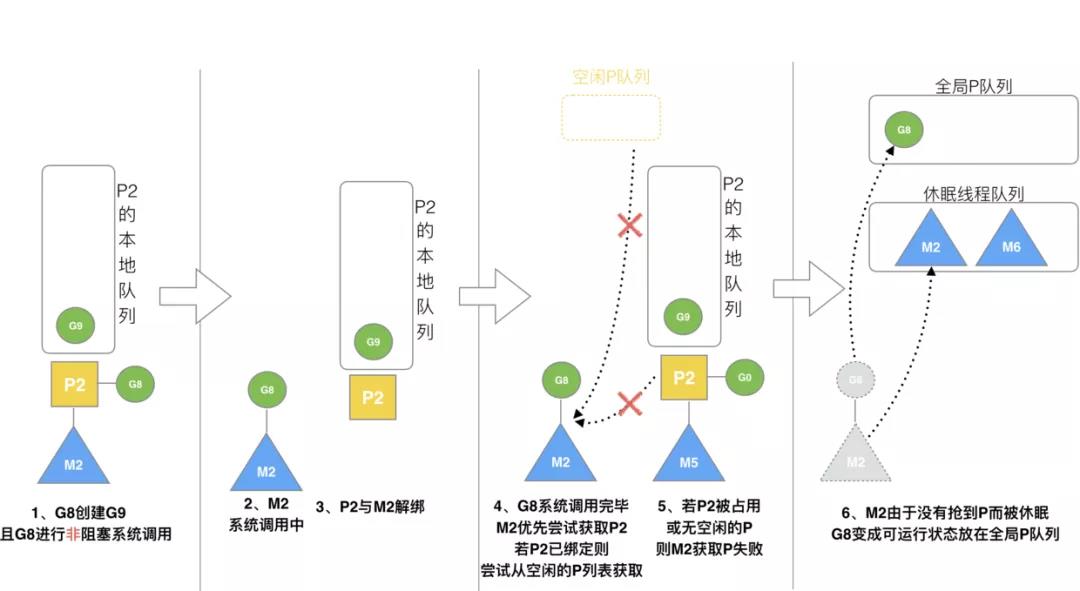
如果 p2 本地队列有 g、全局队列有 g 或有空闲的 m，p2 都会立马唤醒1个 m 和它绑定，否则 p2 则会加入到空闲 P 列表，等待 m 来获取可用的 p。

本场景中，p2 本地队列有 g，可以和其他自旋线程 m5 绑定。



### 场景八 G和M进入系统函数

假设 g8 创建了 g9，假如 g8 进行了非阻塞系统调用（CGO会是这种方式，见cgocall()），m2 和 p2 会解绑，但 m2 会记住 p，然后 g8 和 m2 进入系统调用状态。当 g8 和 m2 退出系统调用时，会尝试获取 p2，如果无法获取，则获取空闲的 p，如果依然没有，g8 会被记为可运行状态，并加入到全局队列。



### 场景九 协作的请求式抢占

前面说过，Go 调度在 go1.12 实现了抢占，应该更精确的称为基于协作的请求式抢占，那是因为 go 调度器的抢占和 OS 的线程抢占比起来很柔和，不暴力，不会说线程时间片到了，或者更高优先级的任务到了，执行抢占调度。

go 的抢占调度柔和到只给 goroutine 发送 1 个抢占请求，至于 goroutine 何时停下来，那就管不到了。

抢占请求需要满足2个条件中的1个：

* G 进行系统调用超过 20us
* G 运行超过 10ms。调度器在启动的时候会启动一个单独的线程 sysmon，它负责所有的监控工作，其中 1 项就是抢占，发现满足抢占条件的 G 时，就发出抢占请求。

## 系统调用

UNIX 系统上运行的程序最终都会通过 C 系统调用来和内核打交道。用其他语言编写程序进行系统调用，方法不外乎两个：一是自己封装，二是依赖 glibc、或者其他的运行库。Go 语言选择了前者，把系统调用都封装到了 syscall 包。封装时也同样得通过汇编实现。

Go 语言通过 Syscall 和 Rawsyscall 等使用汇编语言编写的方法封装了操作系统提供的所有系统调用。

异步系统调用 G 会和MP分离（G挂到netpoller），同步系统调用 GM 会和P分离（P另寻M），生动的说明了GPM相对GM的精妙之处。

### 阻塞

在 Go 里面阻塞主要分为以下 4 种场景：

#### 锁

由于原子、互斥量或通道操作调用导致  Goroutine  阻塞，调度器将把当前阻塞的 Goroutine 切换出去，重新调度 LRQ 上的其他 Goroutine；

#### 网络

由于网络请求和 IO 操作导致  Goroutine  阻塞。Go 程序提供了网络轮询器（NetPoller）来处理网络请求和 IO 操作的问题，其后台通过 kqueue（MacOS），epoll（Linux）或  iocp（Windows）来实现 IO 多路复用。通过使用 NetPoller 进行网络系统调用，调度器可以防止  Goroutine  在进行这些系统调用时阻塞 M。这可以让 M 执行 P 的  LRQ  中其他的  Goroutines，而不需要创建新的 M。

执行网络系统调用不需要额外的 M，网络轮询器使用系统线程，它时刻处理一个有效的事件循环，有助于减少操作系统上的调度负载。用户层眼中看到的 Goroutine 中的“block socket”，实现了 goroutine-per-connection 简单的网络编程模式。实际上是通过 Go runtime 中的 netpoller 通过 Non-block socket + I/O 多路复用机制“模拟”出来的。

#### 读文件

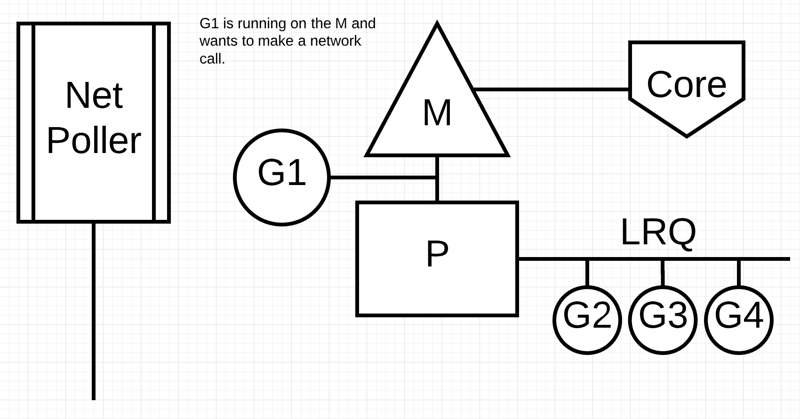
当调用一些系统方法的时候（如文件 I/O），如果系统方法调用的时候发生阻塞，这种情况下，网络轮询器（NetPoller）无法使用，而进行系统调用的  G1  将阻塞当前 M1。调度器引入 其它M 来服务 M1 的P。

#### Sleep

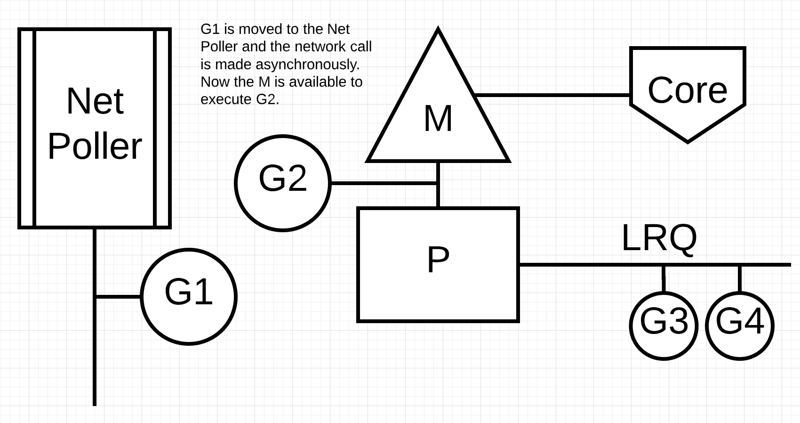
如果在 Goroutine 去执行一个 sleep 操作，导致 M 被阻塞了。Go 程序后台有一个监控线程 sysmon，它监控那些长时间运行的 G 任务然后设置可以强占的标识符，别的 Goroutine 就可以抢先进来执行。

### 异步系统调用

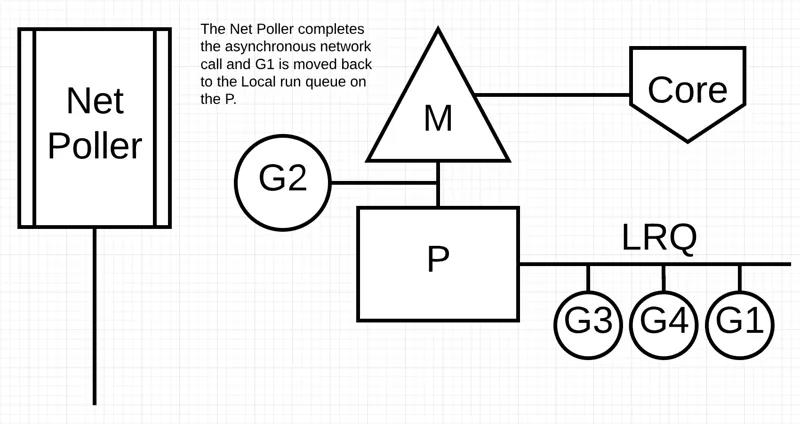
通过使用网络轮询器进行网络系统调用，调度器可以防止 Goroutine 在进行这些系统调用时阻塞M。这可以让M执行P的 LRQ 中其他的 Goroutines，而不需要创建新的M。有助于减少操作系统上的调度负载。（LRQ = local routine queue）



G1正在M上执行，还有 3 个 Goroutine 在 LRQ 上等待执行

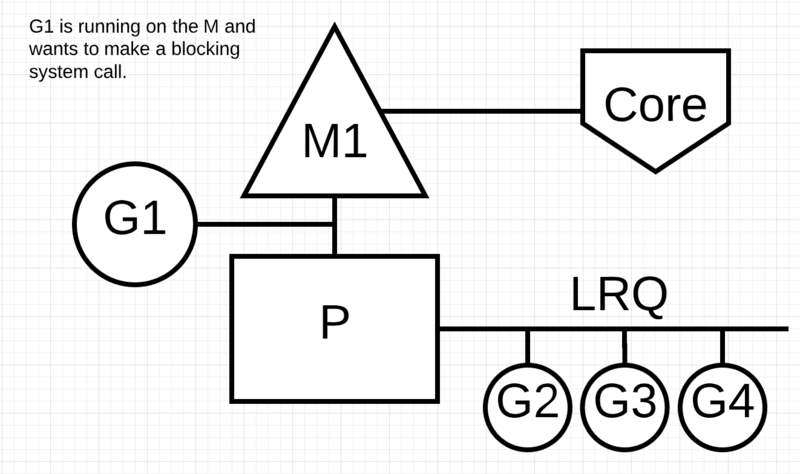


接下来，G1想要进行网络系统调用，因此它被移动到网络轮询器并且处理异步网络系统调用。然后，M可以从 LRQ 执行另外的 Goroutine。

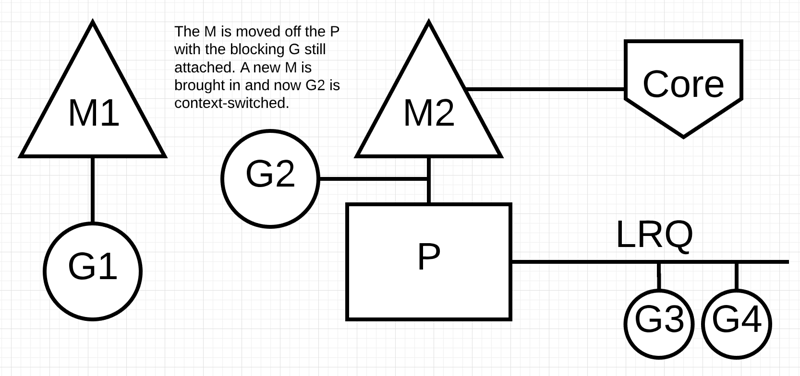


最后：异步网络系统调用由网络轮询器完成，G1被移回到P的 LRQ 中。一旦G1可以在M上进行上下文切换，它负责的 Go 相关代码就可以再次执行。

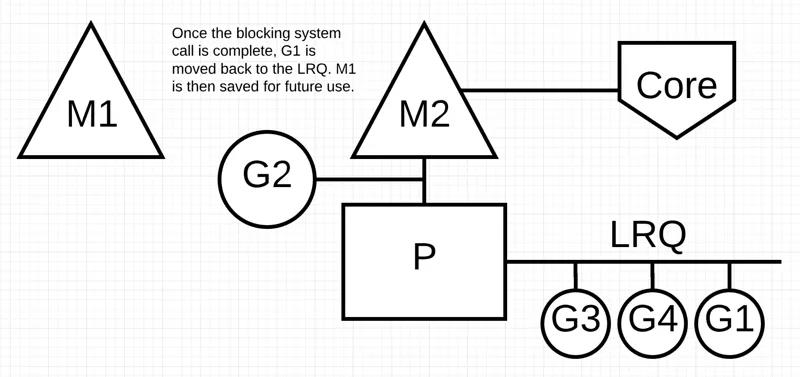
### 同步系统调用



G1将进行同步系统调用以阻塞M1

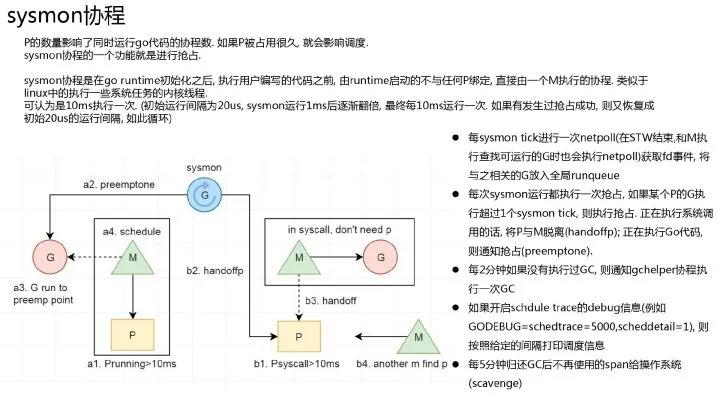


调度器介入后：识别出G1已导致M1阻塞，此时，调度器将M1与P分离，同时也将G1带走。然后调度器引入新的M2来服务P。



阻塞的系统调用完成后：G1可以移回 LRQ 并再次由P执行。如果这种情况需要再次发生，M1将被放在旁边以备将来使用。

### sysmon 协程



在 linux 内核中有一些执行定时任务的线程, 比如定时写回脏页的 pdflush, 定期回收内存的 kswapd0, 以及每个 cpu 上都有一个负责负载均衡的 migration 线程等。

在 go 运行时中也有类似的协程 sysmon。 sysmon 运行在 M，且不需要 P。它会每隔一段时间检查 Go 语言runtime，确保程序没有进入异常状态。

系统监控的触发时间就会稳定在 10ms，功能比较多:

* 检查死锁runtime.checkdead
* 运行计时器 — 获取下一个需要被触发的计时器；
* 定时从 netpoll 中获取 ready 的协程
* Go 的抢占式调度

当 sysmon 发现 M 已运行同一个 G（Goroutine）10ms 以上时，它会将该 G 的内部参数 preempt 设置为 true。然后，在函数序言中，当 G 进行函数调用时，G 会检查自己的 preempt 标志，如果它为 true，则它将自己与 M 分离并推入“全局队列”。由于它的工作方式（函数调用触发），在 for{} 的情况下并不会发生抢占，如果没有函数调用，即使设置了抢占标志，也不会进行该标志的检查。

Go1.14 引入抢占式调度（使用信号的异步抢占机制），sysmon 仍然会检测到运行了 10ms 以上的 G（goroutine）。然后，sysmon 向运行 G 的 P 发送信号（SIGURG）。Go 的信号处理程序会调用P上的一个叫作 gsignal 的 goroutine 来处理该信号，将其映射到 M 而不是 G，并使其检查该信号。gsignal 看到抢占信号，停止正在运行的 G。

* 在满足条件时触发垃圾收集回收内存；
* 打印调度信息,归还内存等定时任务.

## 状态汇总

从上面的场景中可以总结各个模型的状态：

### G状态

G的主要几种状态：

| 状态 | 描述 |
| --- | --- |
| \_Gidle | 刚刚被分配并且还没有被初始化，值为0，为创建goroutine后的默认值 |
| \_Grunnable | 没有执行代码，没有栈的所有权，存储在运行队列中，可能在某个P的本地队列或全局队列中 |
| \_Grunning | 正在执行代码的goroutine，拥有栈的所有权 |
| \_Gsyscall | 正在执行系统调用，拥有栈的所有权，没有执行用户代码，被赋予了内核线程 M 但是不在运行队列上 |
| \_Gwaiting | 由于运行时而被阻塞，没有执行用户代码并且不在运行队列上，但是可能存在于 Channel 的等待队列上 |
| \_Gdead | 当前goroutine未被使用，没有执行代码，可能有分配的栈，分布在空闲列表 gFree，可能是一个刚刚初始化的 goroutine，也可能是执行了 goexit 退出的 goroutine |
| \_Gcopystack | 栈正在被拷贝，没有执行代码，不在运行队列上 |
| \_Gpreempted | 由于抢占而被阻塞，没有执行用户代码并且不在运行队列上，等待唤醒 |
| \_Gscan | GC 正在扫描栈空间，没有执行代码，可以与其他状态同时存在 |

### P 状态

| **状态** | **描述** |
| --- | --- |
| \_Pidle | 处理器没有运行用户代码或者调度器，被空闲队列或者改变其状态的结构持有，运行队列为空 |
| \_Prunning | 被线程 M 持有，并且正在执行用户代码或者调度器 |
| \_Psyscall | 没有执行用户代码，当前线程陷入系统调用 |
| \_Pgcstop | 被线程 M 持有，当前处理器由于垃圾回收被停止 |
| \_Pdead | 当前处理器已经不被使用 |

### M 状态

自旋线程：处于运行状态但是没有可执行 goroutine 的线程，数量最多为 GOMAXPROC，若是数量大于 GOMAXPROC 就会进入休眠。

非自旋线程：处于运行状态有可执行 goroutine 的线程。

## 调度器设计的两大思想

### 多路复用线程

协程本身就是运行在一组线程之上，所以不需要频繁的创建、销毁线程，而是对线程进行复用。在调度器中复用线程还有2个体现：

* work stealing，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。
* hand off，当本线程因为 G 进行系统调用阻塞时，线程释放绑定的 P，把 P 转移给其他空闲的线程执行。

### 利用并行

GOMAXPROCS 设置 P 的数量，当 GOMAXPROCS 大于 1 时，就最多有 GOMAXPROCS 个线程处于运行状态，这些线程可能分布在多个 CPU 核上同时运行，使得并发利用并行。另外，GOMAXPROCS 也限制了并发的程度，比如 GOMAXPROCS = 核数/2，则最多利用了一半的 CPU 核进行并行。

## 调度器设计的两小策略

### 抢占

在 coroutine 中要等待一个协程主动让出 CPU 才执行下一个协程，在 Go 中，一个 goroutine 最多占用CPU 10ms，防止其他 goroutine 被饿死，这就是 goroutine 不同于 coroutine 的一个地方。

### 全局G队列

在新的调度器中依然有全局 G 队列，但功能已经被弱化了，当 M 执行 work stealing 从其他 P 偷不到 G 时，它可以从全局 G 队列获取 G。

# GC

## 引用计数&可达性分析

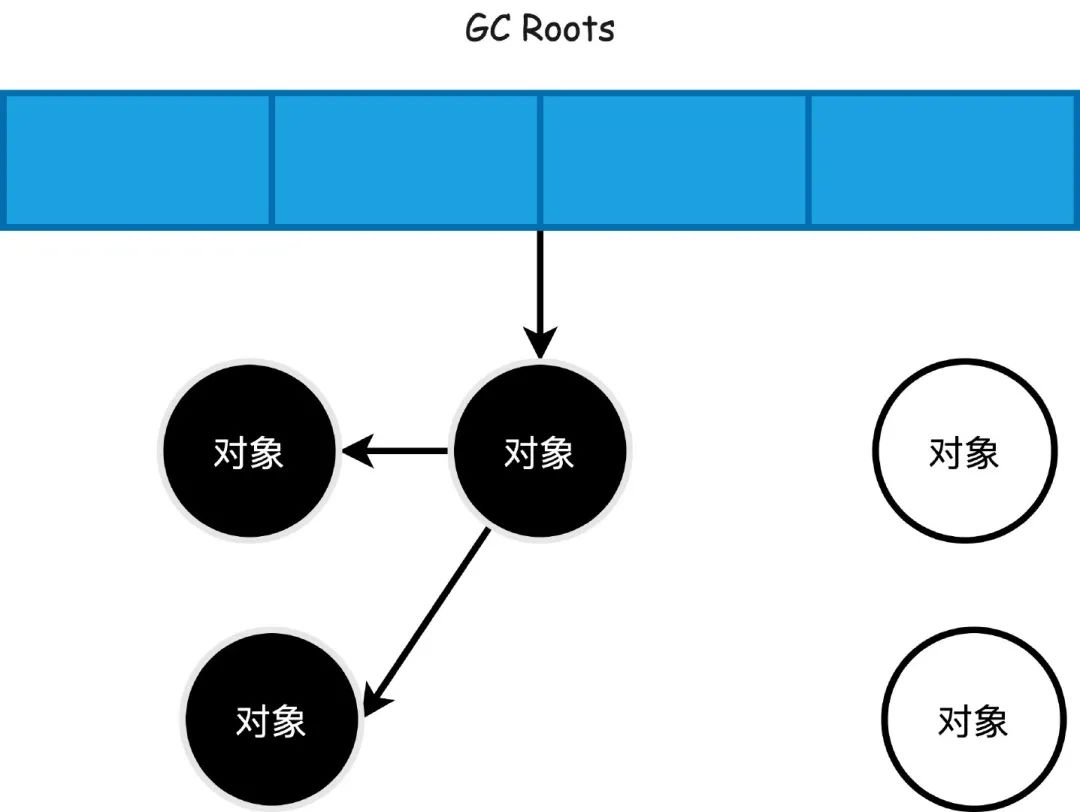
要进行垃圾回收GC，那么我们首先就要决定到底怎么判断对象是否存活？一般来说有两种方式。

引用计数，给对象添加一个计数器，每当有地方引用它计数器就+1，反之引用失效时就-1，那么计数器值为0的对象就是可以回收的对象，但是有一个问题就是循环引用的话无法解决。

对于现在的虚拟机来说，主要用的算法是可达性分析算法。

首先定义GC ROOTS根对象集合，通过GC ROOTS向下搜索，搜索的过程走过的路径称作引用链，如果某个对象到GC ROOTS没有任何引用链，那么就是对象不可达，是可以被回收的对象。

不可达对象需要进行两次标记，第一次发现没有引用链相连，会被第一次标记，如果需要执行finalize()方法，之后这个对象会被放进队列中等待执行finalize()，如果在finalize()中成功和引用链上的其他对象关联，就会被移出可回收对象集合。（但是不建议使用finalize()方法）



## 分代收集（年轻代、老年代）

——对年轻代和老年代对象建索引

有了如何判断对象存活的基础，接下来的问题就是怎么进行垃圾收集GC，现在商用的虚拟机基本上都是分代收集的实现，它的实现建立于两个假说：

* 绝大多数对象都是朝生夕死的
* 熬过越多次垃圾回收的对象越难死亡

基于这两个假说，就产生了现在我们常见的年轻代和老年代。

因为分代了，所以GC也就分代了。

年轻代用于存放那些死的快的对象，年轻代GC我们称之为MinorGC，每次年轻代内存不够我们就触发MinorGC，以后还有存活的对象我们就根据经历过MinorGC次数和动态年龄判断来决定是否晋升老年代。

老年代则存放老不死的对象，这里GC称之为OldGC，现在也有很多人把他叫做FullGC，实际上这并不准确，FullGC应该泛指年轻代和老年代的的GC。

按照我们上文所说的使用可达性分析算法来判断对象的存活，那么假如我们进行MinorGC，会不会有对象被老年代引用着？进行OldGC会不会又有对象被年轻代引用着？

如果是的话，那我们进行MinorGC的时候不光要管GC Roots，还有再去遍历老年代，这个性能问题就很大了。

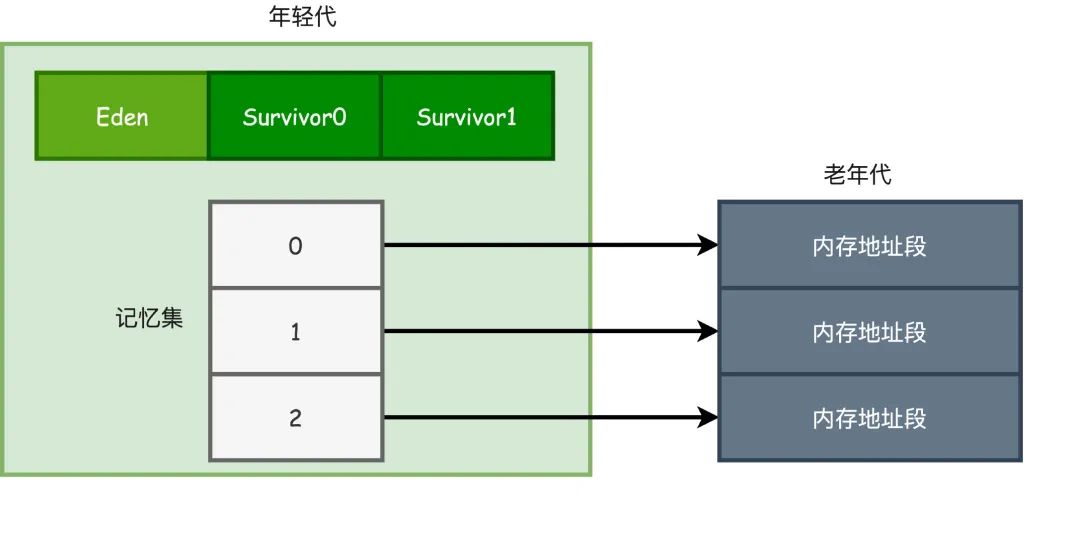
因此，又来了一个假说。。。

**跨代引用相对于同代引用来说仅占极少数。**

由此就产生了一个新的解决方案，我们不用去扫描整个老年代了，只要在年轻代建立一个数据结构，叫做**记忆集Remembered Set，他把老年代划分为N个区域，标志出哪个区域会存在跨代引用。**

以后在进行MinorGC的时候，只要把这些包含了跨代引用的内存区域加入GC Roots一起扫描就行了。

——建立索引，标记需要扫描的老年代。



## 卡表（记录需要扫描的老年代）

说完这些，才到了第一个话题：卡表。

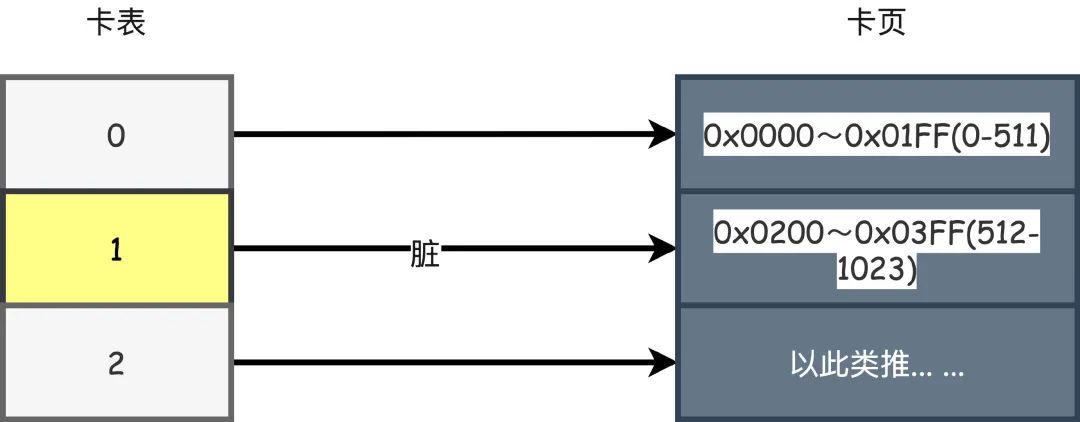
卡表实际上就是记忆集的一种实现方式，如果说记忆集是接口的话，那么卡表就是他的实现类。

对于HotSpot虚拟机来说，卡表的实现方式就是一个字节数组。

CARD\_TABLE [this address >> 9] = 0;

这段代码代表着卡表标记的的逻辑。实际上卡表就是映射了一块块的内存地址，这些内存地址块称为卡页，从代码可以看出每个卡页的大小就是2^9=512字节。

如果转换为16进制，数组的0，1号元素就映射为0x0000～0x01FF(0-511)、0x0200～0x03FF(512-1023)内存地址的卡页。



只要一个卡页内的对象存在一个或者多个跨代对象指针，就将该位置的卡表数组元素修改为1，表示这个位置为脏，没有则为0。

在GC的时候，就直接把值为1对应的卡页对象指针加入GC Roots一起扫描即可。

**有了卡表，我们就不需要去在发生MinorGC的时候扫描整个老年代了**，性能得到了极大的提升。

## 卡表的问题

### 写屏障（回调）

卡表的数组元素要修改成1，也就是脏的状态，对于HotSpot来说是通过写屏障来实现的，**实际上就是在其他分代引用了当前分代的对象时候，在对引用进行赋值的时候进行更新**，更新的方式类似AOP的切面思想。（回调）

void oop\_field\_store(oop\* field, oop new\_value) {

// 引用字段赋值操作

\*field = new\_value;

// 写后屏障，在这里完成卡表状态更新

post\_write\_barrier(field, new\_value);

}

写屏障带来的问题就是额外的性能开销，不过这个问题不大，还能接受。

### 伪共享（数据竞争）

缓存行通常来说都是64字节，一个卡表元素1个字节，占用的卡页内存大小就是64\*512=32KB的大小。

如果多线程刚好更新刚好处于这32KB范围内的对象，那么就会对性能产生影响。

怎么解决伪共享问题？

JDK7之后新增了一个参数-XX:+UseCondCardMark，他代表是否开启卡表更新的判断，没有被标记过才标记为脏。

if (CARD\_TABLE [this address >> 9] != 0)

CARD\_TABLE [this address >> 9] = 0;

## 三色标记法

卡表解决了跨代收集和根节点枚举的性能问题。而有了这些措施实际上枚举根节点这个过程造成的STW停顿已经属于可控范围。

**另外还存在一个问题就是接下来从GC Roots开始遍历，怎么才能高效的标记这些对象，这就是三色标记法的作用了。**因为如果堆内的对象越多，那么显然标记产生的停顿时间就越长。

以现在我们熟知的CMS或者G1来举例，GC的前两个步骤如下：

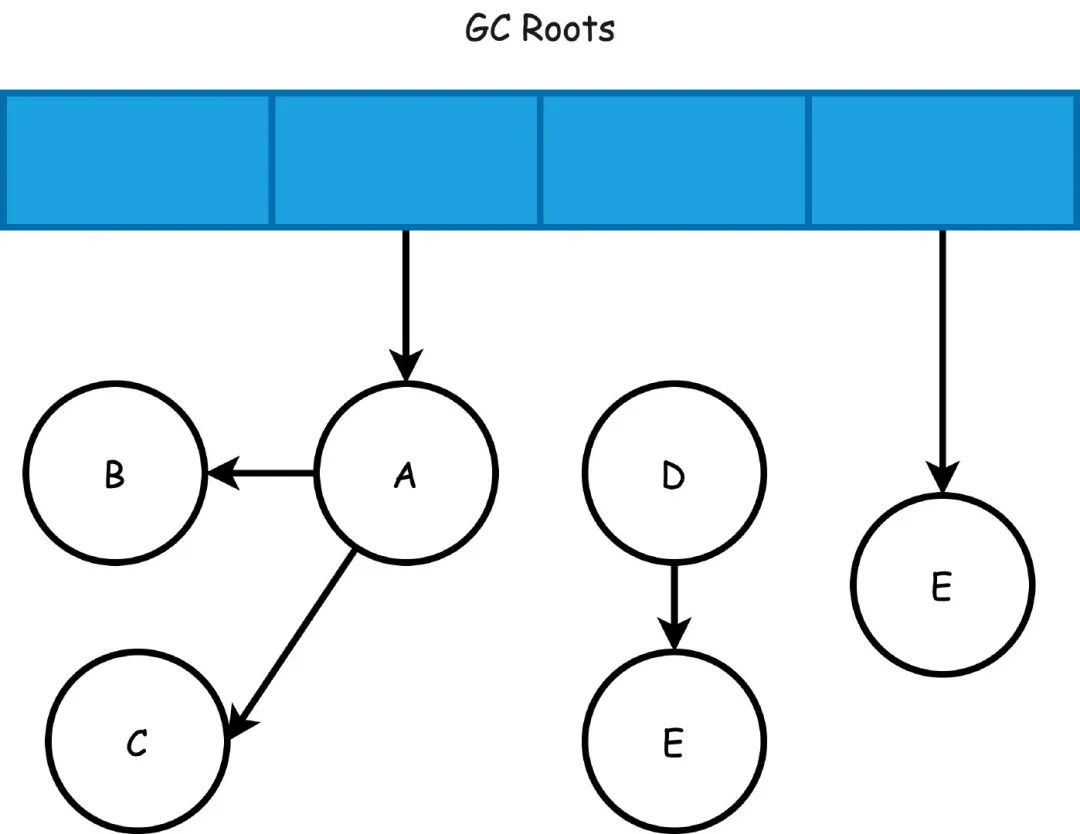
**初始标记：标记GC ROOT能关联到的对象，这一步需要STW**，但是停顿的时间很短。

**并发标记：从GCRoots的直接关联对象开始遍历整个对象图的过程**，这个时间会比较长，但是现在是可以和用户线程并发执行的，这个效率的问题就是三色标记关注的问题。

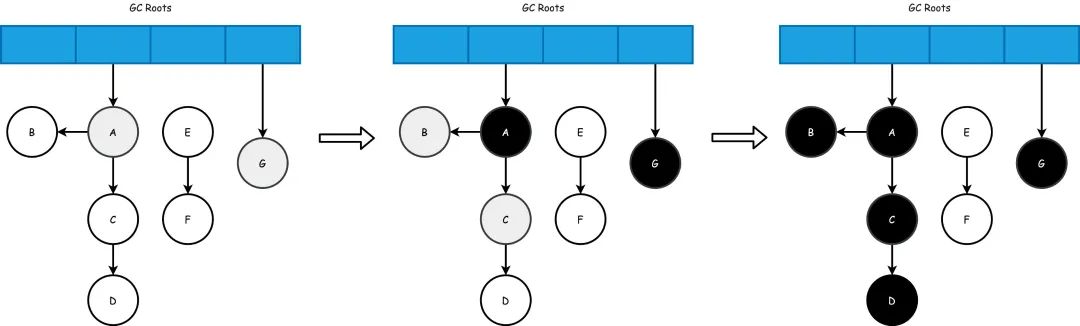
在三色标记法中，把从GC Roots开始遍历的对象标记为以下三种颜色：

* 白色，在刚开始遍历的时候，所有的对象都是白色的。**还没扫描过**
* 灰色，被垃圾回收器扫描过，但是至少还有一个引用没有被扫描。**扫了，但没扫描完。**
* 黑色，被垃圾回收器扫描过，并且这个对象的引用也全部都被扫描过，是安全存活的对象。**扫描完了，划为老年代。**

整个标记的过程如下，首先刚开始从GC Roots开始遍历的时候肯定所有的对象都是白色的。



1. 接着A\G对象被扫描到变成灰色，然后A\G对象的引用也都被扫描，A\G对象变成黑色。
2. B\C对象开始被扫描变成灰色，他们的引用也被扫描完成后自己也就都变成了黑色。
3. 而后D对象也一样会经历从灰色到黑色的过程(偷点懒，省略一张无关紧要的过程图吧)



## 三色标记的问题

虽然三色标记法很高效，但是也会引申出其他的问题。

首先我们上文说过并发标记的过程是不会STW的，就是你妈在打扫卫生，而你在旁边一直丢垃圾，这也没关系，大不了最后就是还有一些垃圾没扫干净而已。

对于三色标记来说就是把应该要清理的对象标记成存活，这样本次GC就无法清理这个对象，这个被称作为浮动垃圾，解决方案就是等下次GC的时候再清理，这次扫不干净就等你妈下次打扫卫生的时候再清理就行了。

与此相反，如果把存活对象标记成需要清理，那么就有点麻烦了，这样你的程序就该出问题了。

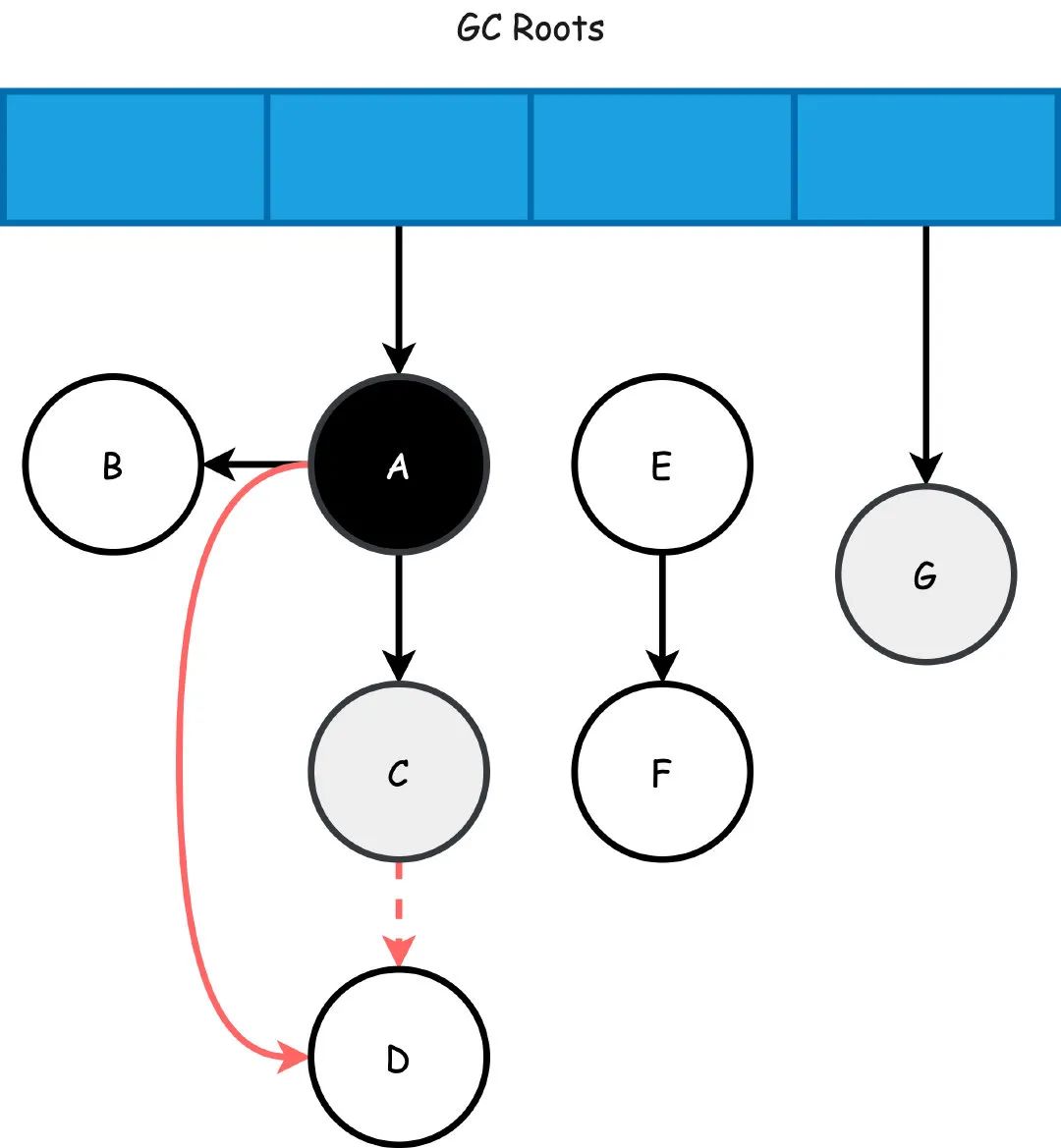
所以经过研究表明，只有同时满足两个条件才会发生这种对象消失的问题：

* **插入了一条或者多条黑色到白色对象的引用。（新产生的老引新）**
* **删除了全部从灰色到白色对象的引用。（同时灰引新被删了）**

那么，针对这个问题也有两种解决方案：增量更新和原始快照，如果对应到垃圾回收器的话，CMS使用的是增量更新，而像G1则是使用原始快照。

思路就是既然要同时满足，那么我只需要破坏其中一个条件那么不就可以了吗？

所以，先看上面我们的例子中的一个场景，假设A扫描完，刚好C成为灰色，此时C->D的引用删除，同时A->D新增了引用（同时满足两个条件了吧），这样本来按照顺序接下来D应该会变成黑色(黑色对象不应该被清理)，**但是由于C->D没有引用了，A已经成为了黑色对象，他不会再被重新扫描了，所以即便新增了A->D的引用，D也只能成为白色对象，最终被无情地清理。**



**增量更新解决方案就是，他会把这些新插入的引用记录下来，扫描结束之后，再以黑色对象为根重新扫描一次**。这样看起来不就是增量更新吗？新插入的记录再扫一次！

**原始快照则是去破坏第二个条件，他把这个要删除的引用记录下来，扫描结束之后，以灰色对象为根重新扫描一次**。所以就像是快照一样，不管你删没删，其实最终还是会按照之前的关系重新来一次。

# 工具

## go module

Go 的包管理方式是逐渐演进的， 最初是 monorepo 模式，所有的包都放在 GOPATH 里面，使用类似命名空间的包路径区分包，不过这种包管理显然是有问题，由于包依赖可能会引入破坏性更新，生产环境和测试环境会出现运行不一致的问题。

从 v1.5 开始开始引入 vendor 包模式，如果项目目录下有 vendor 目录，那么 go 工具链会优先使用 vendor 内的包进行编译、测试等，这之后第三方的包管理思路都是通过这种方式来实现，比如说由社区维护准官方包管理工具 dep。

不过官方并不认同这种方式，在 v1.11 中加入了 Go Module 作为官方包管理形式，就这样 dep 无奈的结束了使命。最初的 Go Module 提案的名称叫做 vgo，下面为了介绍简称为 gomod。不过在 v1.11 和 v1.12 的 Go 版本中 gomod 是不能直接使用的。可以通过 go env 命令返回值的 GOMOD 字段是否为空来判断是否已经开启了 gomod，如果没有开启，可以通过设置环境变量 export GO111MODULE=on 开启。

目前 gomod 在 Go v1.12 功能基本稳定，到下一个版本 v1.13 将默认开启，是时候开始在项目中使用 gomod 了。

### Hello,World

Go 维护者 Russ Cox 写一个简单的库，用于说明 gomod 的使用，下文我将使用这个库开始介绍。

首先新建一个文件夹，然后直接使用 go mod init 即可。

mkdir -p ~/gopher/gomodtest

cd ~/gopher/gomodtest

go mod init github.com/islishude/gomodtest

这时可看到目录内多了 go.mod 文件，内容很简单只有两行：

module github.com/islishude/gomodtest

go 1.12

首行为当前的模块名称，接下来是 go 的使用版本。这两行和 npm package.json 的 name 和 engine 字段的功能很类似。

然后新建一个 main.go 写入以下内容，这里我们引用了 rsc.io/quote 包，注意我们现在还没有下载这个包。

package main

import (

"fmt"

"rsc.io/quote")

func main() {

fmt.Println(quote.Hello())}

如果是默认情况下，使用 go run main.go 肯定会提示找不到这个包的错误，但是当前 gomod 模式，如果没有此依赖回先下载这个依赖。

$ go run main.go

go: finding rsc.io/quote v1.5.2

go: finding rsc.io/sampler v1.3.0

go: finding golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c

go: downloading rsc.io/quote v1.5.2

go: extracting rsc.io/quote v1.5.2

go: downloading rsc.io/sampler v1.3.0

go: extracting rsc.io/sampler v1.3.0

go: downloading golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c

go: extracting golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c

Hello, world.

因为包含 [http://golang.org](https://link.zhihu.com/?target=http://golang.org" \t "/home/wby/文档\\x/_blank) 下的包，记得设置代理。这个时候当前包目录除了 go.mod 还有一个 go.sum 的文件，这个类似 npm package-lock.json。

gomod 不会在 $GOPATH/src 目录下保存 rsc.io 包的源码，而是包源码和链接库保存在 $GOPATH/pkg/mod 目录下。

$ ls $GOPATH/pkg/mod

cache golang.org rsc.io

除了 go run 命令以外，go build、go test 等命令也能自动下载相关依赖包。

### 包管理命令

当然我们平常都不会直接先写代码，写上引入的依赖名称和路径，然后在 build 的时候在下载。

### 安装依赖

如果要想先下载依赖，那么可以直接像以前那样 go get 即可，不过 gomod 下可以跟语义化版本号，比如 go get foo@v1.2.3，也可以跟 git 的分支或 tag，比如go get foo@master，当然也可以跟 git 提交哈希，比如 go get foo@e3702bed2。需要特别注意的是，gomod 除了遵循语义化版本原则外，还遵循最小版本选择原则，也就是说如果当前版本是 v1.1.0，只会下载不超过这个最大版本号。如果使用 go get foo@master，下次在下载只会和第一次的一样，无论 master 分支是否更新了代码，如下所示，使用包含当前最新提交哈希的虚拟版本号替代直接的 master 版本号。

$ go get golang.org/x/crypto/sha3@master

go: finding golang.org/x/crypto/sha3 latest

go: finding golang.org/x/crypto latest

$ cat go.mod

module github.com/adesight/test

go 1.12

require (

golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a // indirect

rsc.io/quote v1.5.2

)

如果下载所有依赖可以使用 go mod download 命令。

### 升级依赖

查看所有以升级依赖版本：

$ go list -u -m all

go: finding golang.org/x/sys latest

go: finding golang.org/x/crypto latest

github.com/adesight/test

golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a

golang.org/x/sys v0.0.0-20190215142949-d0b11bdaac8a [v0.0.0-20190316082340-a2f829d7f35f]

golang.org/x/text v0.3.0

rsc.io/quote v1.5.2

rsc.io/sampler v1.99.99

升级次级或补丁版本号：

go get -u rsc.io/quote

仅升级补丁版本号：

go get -u=patch rscio/quote

升降级版本号，可以使用比较运算符控制：

go get foo@'<v1.6.2'

### 移除依赖

当前代码中不需要了某些包，删除相关代码片段后并没有在 go.mod 文件中自动移出。

运行下面命令可以移出所有代码中不需要的包：

go mod tidy

如果仅仅修改 go.mod 配置文件的内容，那么可以运行 go mod edit --droprequire=path，比如要移出 golang.org/x/crypto 包

go mod edit --droprequire=golang.org/x/crypto

### 查看依赖包

可以直接查看 go.mod 文件，或者使用命令行：

$ go list -m all

github.com/adesight/test

golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a

golang.org/x/sys v0.0.0-20190215142949-d0b11bdaac8a

golang.org/x/text v0.3.0

rsc.io/quote v1.5.2

rsc.io/sampler v1.99.99

$ go list -m -json all # json 格式输出

{

"Path": "golang.org/x/text",

"Version": "v0.3.0",

"Time": "2017-12-14T13:08:43Z",

"Indirect": true,

"Dir": "/Users/lishude/go/pkg/mod/golang.org/x/text@v0.3.0",

"GoMod": "/Users/lishude/go/pkg/mod/cache/download/golang.org/x/text/@v/v0.3.0.mod"

}

{

"Path": "rsc.io/quote",

"Version": "v1.5.2",

"Time": "2018-02-14T15:44:20Z",

"Dir": "/Users/lishude/go/pkg/mod/rsc.io/quote@v1.5.2",

"GoMod": "/Users/lishude/go/pkg/mod/cache/download/rsc.io/quote/@v/v1.5.2.mod"

}

### 模块配置文本格式化

由于可手动修改 go.mod 文件，所以可能此文件并没有被格式化，使用下面命令进行文本格式化。

go mod edit -fmt

### 发布版本

发布包新版本和其它包管理工具基本一致，可以直接打标签，不过打标签之前需要在 go.mod 中写入相应的版本号：

$ go mod edit --module=github.com/islishude/gomodtest/v2

$ cat go.mod

module github.com/islishude/gomodtest/v2

go 1.12

require (

golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a // indirect

rsc.io/quote v1.5.2

)

官方推荐将上述过程在一个新分支来避免混淆，那么类如上述例子可以创建一个 v2 分支，但这个不是强制要求的。

还有一种方式发布新版本，那就是在主线版本种加入 v2 文件夹，相应的也需要内置 go.mod 这个文件。

比如上述我们引入的 [http://rsc.io/quote](https://link.zhihu.com/?target=http://rsc.io/quote" \t "/home/wby/文档\\x/_blank) 包，其中 v3 版本是用内置文件夹，而 v2 使用的是 tag。

$ tree .

.

├── LICENSE

├── README.md

├── buggy

│ └── buggy\_test.go

├── go.mod

├── go.sum

├── quote.go

├── quote\_test.go

└── v3

├── go.mod

├── go.sum

└── quote.go

$ git tag -a

bad

v1.0.0

v1.1.0

v1.2.0

v1.2.1

v1.3.0

v1.4.0

v1.5.0

v1.5.1

v1.5.2

v1.5.3-pre1

v2.0.0

v2.0.1

v3.0.0

v3.1.0

(END)

根据上面的说明，想必你会看到一个问题，当我们升级主版本号的时候，要更改 module 名称，也就是上面所说的加上版本号，这就存在一个问题，如果我们要更新到主版本号的依赖就没有这么简单了，因为升级的依赖包路径都需要修改，**这个在其它语言包管理以及 Go 第三方包管理工具都不存在的一点**。

如下所示，升级 rsc.io/quote 到 v3 版本。注意一点，作为例子这里包作者对函数也加上了版本，其实大部分人是不会加的。这个模式叫做 semantic import versioning，也是备受争议，大多数人认为这个没有特别大的作用，而维护者则认为这是为了 Go 下一个十年的必要条件。

package main

import (

"fmt"

"rsc.io/quote/v3"

)

func main() {

fmt.Println(quote.HelloV3())

}

对于内部开发我觉得还挺好，让大家都了解，不要随意加入破坏性更新。

不过由于这个不讨喜功能，不同版本可以存在同一个包了。补充一句，对于 v0 和 v1 版本并不需要加入到 import path 内。

package main

import (

"fmt"

q1 "rsc.io/quote"

"rsc.io/quote/v3"

)

func main() {

fmt.Println(quote.HelloV3())

fmt.Println(q1.Hello())

}

### 从老项目迁移

从很多第三方的包管理工具迁移到 gomod 特别简单，直接运行 go mod init 即可。

如果没有使用任何第三方包管理工具，除了运行 go mod init 初始化以外，还要使用 go get ./... 下载安装所有依赖包，并更新 go.mod 和 go.sum 文件。

默认情况下，go get 命令使用 @latest 版本控制符对所有依赖进行下载，如果想要更改某一个包的版本，可以使用 go mod edit --require 命令，比如要更新 rsc.io/quote 到 v3.1.0 版本。

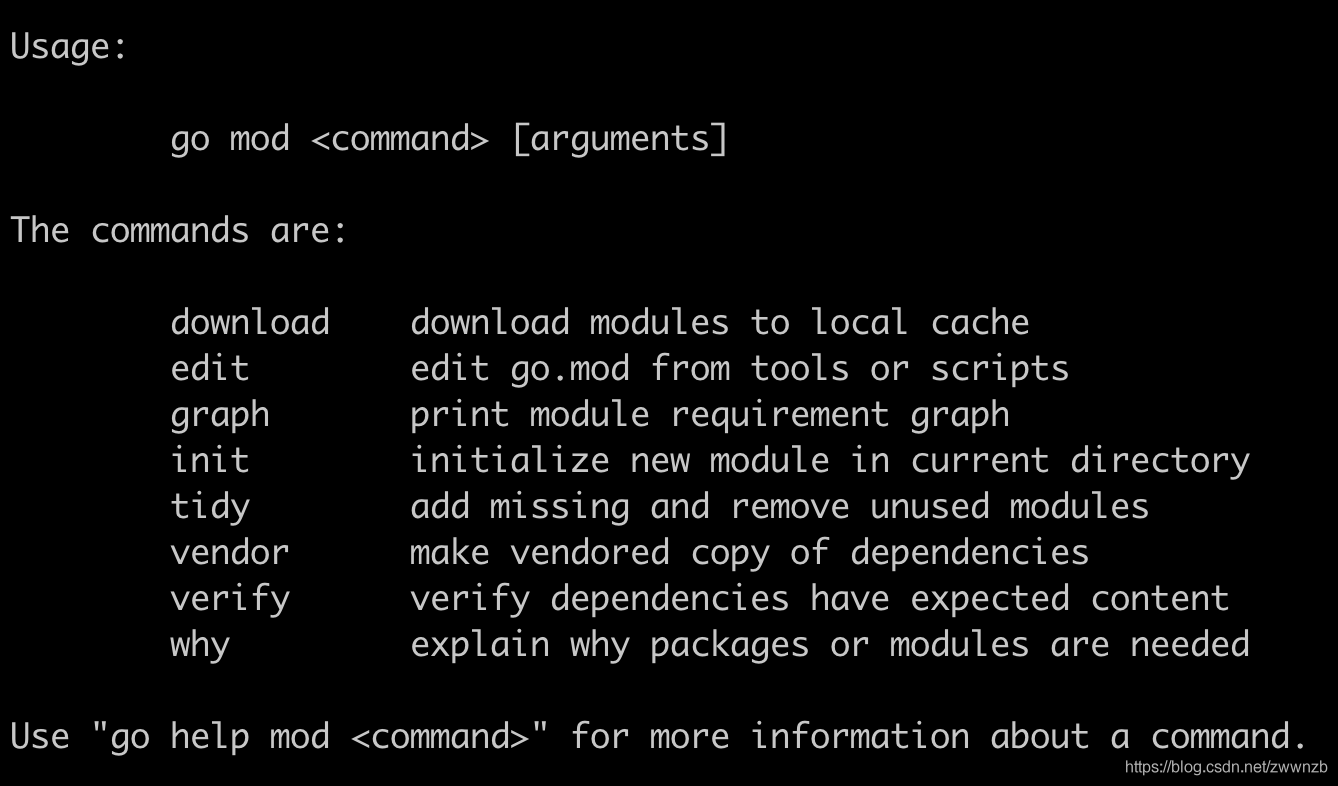
go mod edit [--require=rsc.io/quote@v3.1.0](mailto:--require=rsc.io/quote@v3.1.0)

### mod的基本操作

模块是相关Go包的集合。modules是源代码交换和版本控制的单元。 go命令直接支持使用modules，包括记录和解析对其他模块的依赖性。

modules替换旧的基于GOPATH的方法来指定在给定构建中使用哪些源文件。

go help mod 查看命令



具体实现见以下说明：

创建一个模块 mkdir gomods && cd gomods

初始化模块 模块名为项目名

go.mod文件一旦创建后，它的内容将会被go toolchain全面掌控。go toolchain会在各类命令执行时，比如go get、go build、go mod等修改和维护go.mod文件。

go mod init gomods

mod download 使用  目前所有模块版本数据均缓存在 $GOPATH/pkg/mod和 ​$GOPATH/pkg/sum 下 直接下载依赖包到本地缓存

go mod download

编辑go.mod文件 选项有-json、-require和-exclude，可以使用帮助go help mod edit

go help mod edit

go mod edit -fmt // 如果手动编辑 可以用fmt进行格式化

go mod edit -require=golang.org/x/text 添加依赖 会自动加载到.mod 文件

go mod edit -replace=golang.org/x/crypto@v0.0.0=github.com/golang/crypto@latest //替代依赖包

module语句指定包的名字（路径）

require语句指定的依赖项模块

replace语句可以替换依赖项模块

exclude语句可以忽略依赖项模块

* 以文本模式打印模块需求图

go mod graph

* 删除错误或者不使用的modules 获取依赖的和清理不依赖的或者无效包

go mod tidy

* 生成vendor目录

go mod vendor

* 验证依赖是否正确

go mod verify

* 清理moudle 缓存

go clean -modcache

* 查看某个依赖包可下载的版本

go list -m -versions github.com/gogf/gf

go list -m -json all

go list -m输出的信息被称为build list，也就是构建当前module所要构建的所有相关package（及版本）的列表。

go list -m

go mod 包依赖分为2种： 本地项目：

1:  同一个项目之间引用包

    具体事例见 moduledemo 项目 （mod 模块需要用replace）

2、不同项目之间引用包

   具体事例见 moduledemo 项目 和mypackage 项目 统一目录的项目

   注意点： 如果引用的最后的包名一样的话（无论路径是否相同），会导致错误，给包起个别名可以解决

3、远程控制事例：

  见onemod 项目 在 twomod的应用

备注： 库的 版本升级规范

vendar -》 gopath

## gofmt

Golang的开发团队制定了统一的官方代码风格，并且推出了gofmt工具（gofmt或go fmt）来帮助开发者格式化他们的代码到统一的风格。gofmt是一个cli程序，会优先读取标准输入，如果传入了文件路径的话，会格式化这个文件，如果传入一个目录，会格式化目录中所有.go文件，如果不传参数，会格式化当前目录下的所有.go文件。

gofmt默认不对代码进行简化，使用-s参数可以开启简化代码功能，具体来说会进行如下的转换：

用法：gofmt[flags][path…]

-cpuprofile字符串

将CPU配置文件写入此文件

-d 显示差异而不是重写文件

-e 报告所有错误（不只是不同行的前10个错误）

-l 列出格式与gofmt不同的文件

-R 字符串

重写规则（例如“a[b:len（a）]->a[b:''）

-S 简化代码

可以看到，gofmt命令还支持自定义的重写规则，使用-r参数，按照pattern -> replacement的格式传入规则。

## go版本的控制

查看当前 go的版本 go version

1. go多版本控制工具---gvm

安装方式如下：

**curl -s -S -L https://raw.githubusercontent.com/moovweb/gvm/master/binscripts/gvm-installer**

校验是否安装成功：

gvm version

如果不成功，可能会提示错误，注意需要配置host 文件

   2. 查看已经安装的版本有哪些？

        gvm list

   3. 查看所有的版本

      gvm listall

  4. 指定安装哪个版本

     gvm install  go1.16 -B    安装最新的1.16版本

 5. go版本的使用和切换

   gvm use go1.16

## go语言的编译

go build     最常用的编译go 文件

go install  只是将编译的中间文件放在 GOPATH 的 pkg 目录下，以及固定地将编译结果放在 GOPATH 的 bin 目录下。

go build 文件

go install 文件 本地打包编译命令

GO111MODULE 有三个值：off, on和auto（默认值）

## 获取依赖包的使用

go get github.com/tal-tech/go-zero/core/discov      下载项目依赖包

go get github.com/tal-tech/go-zero/core/disco@latest   下载最新包

go get github.com/tal-tech/go-zero/core/disco@master    拉取master分之最新的commit

go get -u-patch github.com/tal-tech/go-zero/core/disco  更新到最新的修改bug的版本

go get github.com/tal-tech/go-zero/core/disco@v1.0.0     根据tag 拉取commit

go get github.com/tal-tech/go-zero/core/disco@commit\_id     根据commit\_id 获取依赖包 或自动转为对应的tag

go get github.com/tal-tech/go-zero/core/disco/v3 指定版本拉取

go get -u            更新包

## go test 单元测试

特殊：不需要 main() 作为函数入口。所有在以\_test结尾的源码内以Test开头的函数会自动被执行

* 每一个test文件需要import一个testing
* 默认的情况下，go test命令不需要任何的参数，它会自动把你源码包下面所有 test 文件测试完毕，当然你也可以带上参数。
* 这里介绍几个常用的参数：
* -bench regexp 执行相应的 benchmarks，例如 -bench=.；
* -cover 开启测试覆盖率；
* -run regexp 只运行 regexp 匹配的函数，例如 -run=Array 那么就执行包含有 Array 开头的函数；
* -v 显示测试的详细命令

## ****代码规范检测****

 1、ide 配置插件-代码格式化 gofmt

     gofmt -代码格式化:  func def() erro{} 之间的空格等

2、golint --对代码定义规范的检查 （主要 各种语法规范校验）

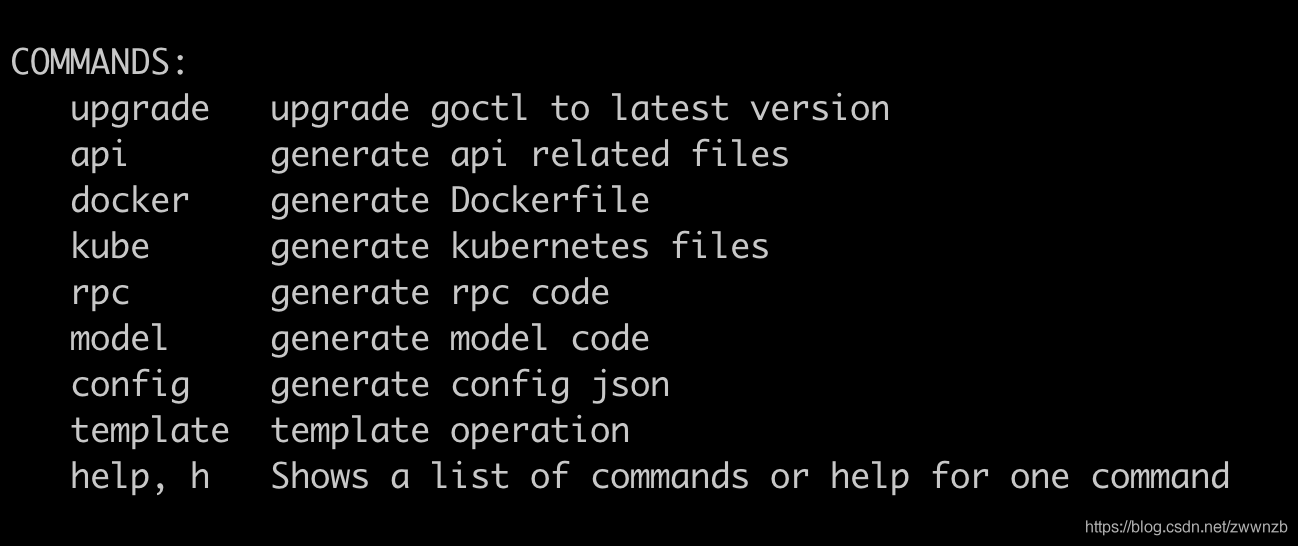
     Golint会对代码做以下几个方面检查

1. package注释 必须按照 “Package xxx 开头”
2. package命名 不能有大写字母、下划线等特殊字符
3. struct、interface等注释 必须按照指定格式开头
4. struct、interface等命名
5. 变量注释、命名
6. 函数注释、命名

golint file //检测文件

golint dir //检测目录

## goctl：代码生成工具



### goctl用途

* 定义api请求
* 根据定义的api自动生成golang(后端), java(iOS & Android), typescript(web & 小程序)，dart(flutter)
* 生成MySQL CURD

### goctl 参数说明

goctl api [go/java/ts] [-api user/user.api] [-dir ./src]

api 后面接生成的语言，现支持go/java/typescript

-api 自定义api所在路径

-dir 自定义生成目录

<https://github.com/tal-tech/go-zero/blob/master/tools/goctl/goctl.md>