Golang

2026-04-01

八股

Channel

CSP模型：Go中不通过共享内存通信，而是通过通信实现内存共享

注意

- 给一个 `nil` 的通道发送数据，会造成**永久阻塞**
- 从一个 `nil` 的通道获取数据，会造成**永久阻塞**
- 关闭一个为 `nil` 的通道，会**引起** `panic`
- 重复关闭通道，会**引起** `panic`
- 给一个关闭的通道发送数据，会**引起** `panic`
- 从一个关闭的通道接收数据，**如果缓冲区为空，返回零值**

如何优雅关闭chan

- 统一由发送者关闭
- 如果有多个发送者，创建一个 `stopCh` ，关闭时使用 `stopCh` 通知所有发送者

hchan结构体

`chan` 的底层实现，本质上是**带锁的环形队列**

- 核心结构
  - `qcount` ：当前队列中剩余的元素个数
  - `dataqsiz` ：环形队列的长度
  - `closed` ：标识通道是否关闭
  - `sendq` ：**发送等待队列**，因缓冲区满而阻塞的 `Goroutine` ，双向链表
  - `recvq` ：**接收等待队列**，因缓冲区空而阻塞的 `Goroutine` ，双向链表
  - `lock` ：**互斥锁**，保证 `Channel` 操作的原子性，实现 `Channel` 线程安全

Ring Buffer（环形队列）

- 环形缓冲区，缓存写入的数据
- 维护两个指针， `recvx` 与 `sendx`
  - `recvx` ：指向缓冲区中未被读取的第一个数据
  - `sendx` ：指向缓冲区中新数据插入的位置

sendq/recvq

- 双向链表
- 存储的不是原始协程G，是封装后的结构体 `sudog`

gopark/goready

- `gopark` ：让协程休眠
  - 将当前协程的状态从 `_Grunning` 改为 `_Gwaiting`

    - 将当前协程从M上剥离下来
    - 让M去执行其它协程
- `goready`：让协程复活
    - 让被阻塞的协程的状态从 `_Gwaiting` 改为 `_Grunnable`
    - 让这个协程重新回到P的本地队列

调度逻辑

- 发送数据
    - **阻塞挂起**：缓冲区已满且无接收者
        - 协程G1被包装成 `sudog` 结构体，存入 `sendq` 双向链表
        - 执行 `gopark` 方法，G1进入休眠，告诉M去执行其它G
    - **直接发送**：有接收者等待
        - 此时 `recvq` 不为空，G2在等待
        - 协程G1直接绕过缓冲区，将数据直接拷贝到G2的栈空间
        - 对G2调用 `goready` 方法，唤醒G2
    - **存入缓冲区**：缓冲区未满
        - 协程G1直接把数据拷贝到缓冲区的 `sendx` 位置
- 接收数据
    - **阻塞挂起**：缓冲区为空且无发送者
        - 协程G2被包装成 `sudog` 结构体，存入 `recvq` 双向链表
        - 调用 `gopark` 方法，G2 进入等待状态，通知M去执行P里的其它协程
    - **直接接收**：缓冲区为空且有发送者**（无缓冲通道）**
        - 接收者G2发现 `sendq` 有节点，直接把数据拷贝到自己的栈空间中
        - 对等待者G1调用 `goready` 方法
    - **缓冲满时接收**：缓冲区已满且无发送者
        - 接收者从 `buf` 取走 `recvx` 位置上的数据
        - 接收者发现 `sendq` 有G1阻塞，顺手把G1的数据放到 `buf` 里面
        - 对G1调用 `goready` 方法
    - **从缓冲区接收**：缓冲区未满且无发送者
        - 接收者从 `buf` 取走 `recvx` 位置上的数据
        - 接收者继续执行代码

make与new

new

- 创建一个**类型**的实例，并返回新分配地址的**指针**
- 使用 `new` 来分配空间，传入的参数是**类型**

make

- 给 `slice` 、 `map` 、 `chan` 进行初始化，然后返回**初始化后的值**

Slice

底层实现

- 指向底层数据的指针
- `len`：切片长度
- `cap`：切片容量

扩容机制

- 如果新申请的容量大于旧容量的两倍，**新容量直接等于新申请的容量**
- 当切片元素小于256个时，**新切片容量直接翻倍**
- 当切片元素大于等于256个时，每次增加$(\text{oldcap} + 3 \times 256)/4$，**直到满足新申请的容量**

内存对齐

- 计算出最终要扩容的容量后，Go并不会直接按这个数字分配内存
- Go会根据预设的内存块大小**进行向上取整**

注意

- 由于扩容会指向**新的底层数组**，扩容后对新切片的修改不会影响旧切片

数组

编译时确定类型，**数组的长度是类型的一部分**

- 函数传递时拷贝整个数组，开销大
- 数组分配在连续内存上

Map

`map` 并不是线程安全的，如果两个协程同时写一个 `map` ，程序会直接 `panic`

底层实现

`map` 本质上是一个**哈希表**，使用**链地址法（数组+链表）**解决哈希冲突

- `hmap` ：是 `map` 的大脑，包括
  - `count` ：成员数量
  - `B` ：桶的数量（2^B），**不包括溢出桶**
  - `buckets` ：指向桶数组的指针
  - `extra` ：溢出桶信息
- `bmap` ：桶，每个桶固定存储**8个键值对**
  - `tophash` 数组：存储每个key哈希值的高8位，用于快速定位key是否在桶内
  - `Key&Values` ：**内存对齐，防止内存浪费**，紧凑排列，key放在一起，value放在一起
  - `Overflow Pointer` ：指向下一个溢出桶的指针
- 溢出桶：桶内位置不够用，就会申请溢出桶
  - 和 `bmap` 结构一样，最后会形成**溢出桶单向链表**
  - **预分配机制**
    - 动态分配：使用 `mallocgc` 向系统申请内存
    - 预留池：初始化大 `map` 时向系统**额外申请一些内存**作为备用溢出桶

定位机制

- **定位桶**：使用哈希值的**最后B位（低位）**定位到具体的桶
- **定位桶内位置**：使用哈希值的**高8位（高位）**在桶内进行快速线性匹配
- **插入**时找到桶内的空插槽，存入Key和Value

扩容机制

根据**装载因子**决定如何进行扩容

$$\text{LocalFactor} = \frac{\text{count}}{2^B}$$

- **翻倍扩容**：装载因子超过6.5，桶快占满了，哈希碰撞严重，**溢出桶多**
  - `B` 变为 `B+1`，旧桶的数据被分流到两个新桶中，通过 `Key` **哈希值的第B位**决定去向
    - 该位置是0：留在**原序号**的新桶中
    - 该位是1：留在**原序号+**2^B的新桶中
- **等量扩容**：装载因子低于6.5，使用**过多的溢出桶**，空间利用率低，导致**桶内数据稀疏**
  - 通常发生在频繁删除哈希表中的数据的场景下，因为删除操作只把数据删除，**不会回收溢出桶占用的内存**
  - 新开辟一块和旧桶一样大的空间
  - 将旧桶里的数据重新排列，紧凑放入新桶中，**消除内存碎片**

渐进式扩容

核心：数据搬迁被拆分到了每一次的**写和删**中

- 新开辟内存空间
- 标记旧桶：将原有的 `buckets` 挂载到 `hmap.oldbuckets` 指针上
- 初始化进度条：`hmap.nevacuate` 置为0
- 当尝试修改或删除某个Key时，先定位到这个Key所在的旧桶，然后把这个旧桶及其挂载的所有数据迁移到新桶

扩容期间的读写逻辑

- **读操作**
  - 先去旧桶找，如果发现旧桶已经被搬迁，去新桶找
  - **读操作本身不触发搬迁**
- **写操作**
  - 先触发搬迁
  - 搬迁完成后，在新桶写数据

值传递

Go里面只有值传递，没有引用传递（小心面试官钓鱼）

即使是 `map`、`chan`、`slice`，在进行传递时也只是拷贝了他们的底层结构体**（包含指向底层数据的指针）**

- 如果在函数内部修改了 `slice` 元素，**外部可见**
- 如果使用 `append`，**外部不可见**，因为外部的 `len` 和 `cap` 变量的值没有改变

三色标记

白色：不可达对象，需要进行GC

灰色：中间状态对象，它内部引用的其他对象还没被检查完

黑色：存活对象，已经被检查过，并且它引用的对象已经被标记成黑色或灰色

原理

- 首先将所有的对象放到白色集合中

- 从**根节点**开始遍历对象，遍历到的白色对象放到灰色集合中
- 遍历**灰色集合**中的对象，将它引用的对象标记成灰色，同时把**它自己标记成黑色**
- 循环上述步骤，直到**无灰色对象为止**

STW

STW是为了捕获即时的 `GC Roots` 指针，从而捕获到最后的指针状态，防止回收正在使用的内存

- 在并发标记期间，栈的状态在第一次扫栈后会一直变化，如果最后不进行STW，就会导致GC回收掉正在使用的内存

本质： `sysmon` 让所有的处理器（P）都停止运行用户代码，进入一个**安全点**（Safe Point），确保内存状态不再发生变化

写屏障

> 若在执行GC时，程序把白色对象挂到了黑色对象下面，怎么办？

写屏障就是专门解决这个问题的：当程序试图修改对象的引用关系时，写屏障会强行把被引用的对象涂成**灰色**

GC回收流程

- 扫描栈和堆，获取初始的 `GC Roots` 集合
- 进行并发三色标记
- GC结束前进行**栈重扫**
    - 在混合写屏障诞生前会产生较长的STW

插入写屏障

概念：当一个**黑色对象**引用一个白色对象时，立刻把这个白色对象涂成**灰色**

- 在栈上不开启，因此GC结束前必须**暂停程序（STW）**重新扫描一遍栈，会造成卡顿

删除写屏障

概念：一个**灰色对象**引用了一个白色对象，程序执行时白色对象**被取消引用**，此时立刻把白色对象涂成**灰色**

- 当**黑色对象持有白色对象，且灰色对象取消这个白色对象的引用时**，就需要删除写屏障
- 因为如果不把白色对象涂灰，由于**黑色对象不会再扫描**，这个被取消引用的白色对象就会被回收，造成程序崩溃

混合写屏障

融合了**插入写屏障和删除写屏障**的优点，但是是保守的策略，会导致内存占用较大

核心：

- 初始进行一次短暂的STW，**将栈上的对象全部标黑**
- 在堆上把旧引用和新引用的对象**涂灰**
- GC期间把**栈上新创建的对象涂黑**

内存逃逸

基本概念：Go编译器在**编译阶段**通过静态分析，决定一个变量应该分配在**栈**上还是**堆**上的过程

- **栈**：随函数调用创建，结束销毁，**无需GC介入**，持有空间小
- **堆**：持有的内存空间大，但需要GC介入进行内存回收

常见逃逸场景

- **指针逃逸**：函数返回变量地址，需要进行逃逸
- **接口类型逃逸**：编译阶段类型不确定
- **变量内存太大**：栈无法承载，逃逸给内存更大的堆
- **动态长度的切片**：切片长度在编译期间不确定

注意

- 在实际开发时，传递指针会**增大GC压力**，因为对象逃逸到堆，增加扫描时长
- `new` 出来的对象不一定在堆上，如果对象没被外部引用，**编译器会对其进行优化分配到栈上**

Go Runtime

负责四大核心工作

- **协程调度**：GMP模型
- **垃圾回收**：三色标记、混合写屏障
- **内存管理与分配**
- **运行时检查与支撑**

sysmon

`System Monitor`

- 独立于GMP模型之外
- 运行在物理线程之上
- 可以看作是一个**后台监控线程**

四大基本任务

- **抢占长任务**：强制中断长时间占用CPU的协程
- **Hand Off**：当G遇到 `syscall` ，M会阻塞，**此时** `sysmon` **强行把与这个M绑定的P进行解绑**，交给其他空闲的M
- **强制GC**：系统长时间未进行GC， `sysmon` 会进行强制GC
- **释放物理内存**

抢占式调度

出现的场景

- STW
- 计算密集型无限循环任务

协作式的抢占式调度

Go1.14版本之前使用的调度方法

具体流程如下

- 后台监控线程sysmon巡检发现某个协程运行超过了10ms，把这个G的 `stackguard0` 标志位设为一个特殊值
  - 这是因为Go编译器在编译阶段会在每个函数的入口处插入一段指令，**用于检查当前协程的栈空间是否足够**
  - 检查过程需要检查标志位 `stackguard0`
- G发现自己的标志位 `stackguard0` 被设为特殊值，主动保存自己的寄存器状态，让出CPU给其它协程

弊端

- 若协程执行的是死循环，由于无函数调用，协程不会检查标志位也就无法退出，此时协程占用CPU资源无法释放

基于信号的抢占式调度

Go1.14版本及之后使用的调度方法

具体流程如下

- 后台监控线程sysmon巡检发现某个协程运行超过了10ms，sysmon调用系统调用，向运行该协程的线程M发送一个 `SIGURG` 信号
- 内核接受到信号后，暂停M的指令流，强制将 CPU 的控制权交给预先注册好的**信号处理函数（sighandler）**
- `sighandler` 会把当前协程的寄存器状态进行保存，并修改协程的程序计数器，让他指向一个特殊的 `Runtime` 函数
- 这个 `Runtime` 函数把原本被中断的代码地址压栈，然后调用调度逻辑，将协程放回全局队列，让M运行其它的协程

Go的GM模型

- **G**： `goroutine` **协程**
- **M**：操作系统内核的 `thread` **线程**

运行流程

- 维护一个全局队列，全局队列存储G
- 系统中所有的M空闲时，就去全局队列取出G执行

系统调用（System Call）

- 简称 `syscall` ，**应用程序向操作系统内核请求服务**的唯一入口
- 一次系统调用的代价非常昂贵，往往需要**几百纳秒**

弊端：

- 任何关于全局队列的操作都需要**加锁**，锁竞争导致严重的性能瓶颈
- 系统调用引起**线程爆炸**，易导致OOM
  - 当协程G1需要进行 `syscall` 时，CPU会把G1踢出CPU，并且把M1挂起（此时M1无法进行任何操作），如果此时全局队列还有很多G未被执行时，系统会创建更多的线程执行G，导致线程爆炸
- **局部性灾难**，缓存失效
  - 当G1执行 `syscall` 时，M1会被系统挂起进入休眠，当系统调用结束后，G1会被送入全局队列，此时可能由M2拿到G1，但是M2并没有G1中的数据，需要从内存重新加载，造成缓存失效

Go的GMP模型

- **G**： `goroutine` **协程**
- **M**：操作系统内核的 `thread` **线程**
- **P**：**协程处理器**

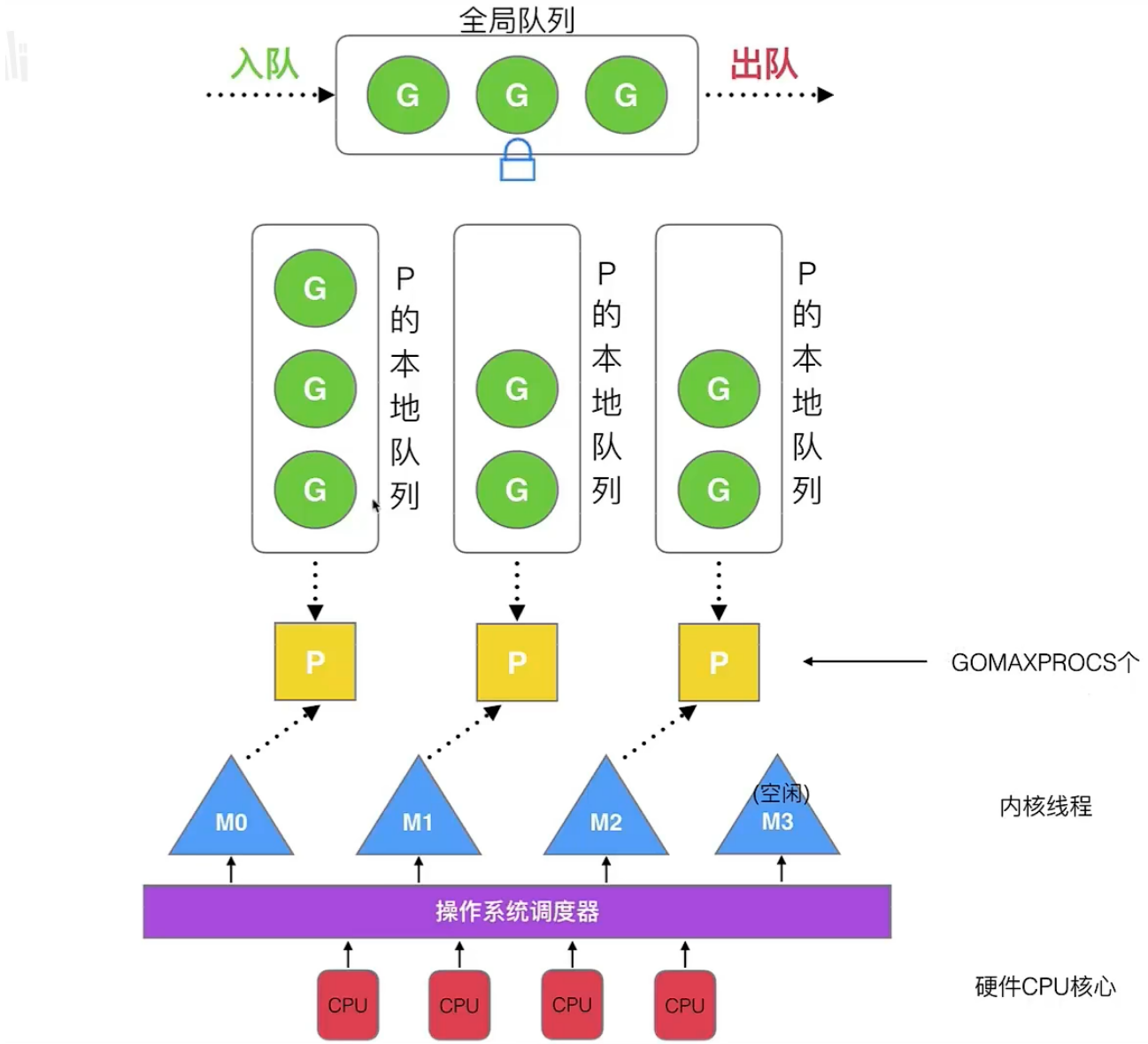

image-20251015160120180

GMP调度流程

- 每个P和一个M绑定，M是真正执行 `goroutine` 的实体，M从**绑定的P的局部队列**获取 `goroutine` 来执行
- 每个P都有一个**局部队列**，局部队列保存待执行的 `goroutine` ，当局部队列满了之后就会把 `goroutine` 放到**全局队列**
- 当M绑定的P的**局部队列为空**时，M会从**全局队列**获取到本地队列来执行G；当全局队列为空时，M会从**其它P的局部队列**偷取G来执行（ `Work Stealing` ）

解读

- 可以通过设置 `GOMAXPROCS` 来调整**协程处理器的个数**
- 每个 P 拥有一个 **本地队列（LRQ）**
- 所有 P 共享一个 **全局队列（GRQ）**
  - 当P的本地队列满了之后，**新创建的协程**会放进全局队列中

设计策略

- **复用线程**： `work stealing` 机制， `hand off` 机制

- `work stealing` 机制：当某一个 `thread` 空闲时，会去别的 `Processor` 的本地队列**偷取一批协程执行**
- `hand off` 机制：当某一个 `thread` 进行 `syscall` **遇到阻塞时**，会唤醒一个 `thread` ，将**被阻塞的** `thread` **的** `Processor` **的本地队列**交给被唤醒的 `thread`
  - 发生 `hand off` 时，需要对线程上下文的状态进行存储，以便下次调度时进行恢复。**M的栈保存在G对象上，将M所需要的寄存器保存到G对象上即可实现现场保护**

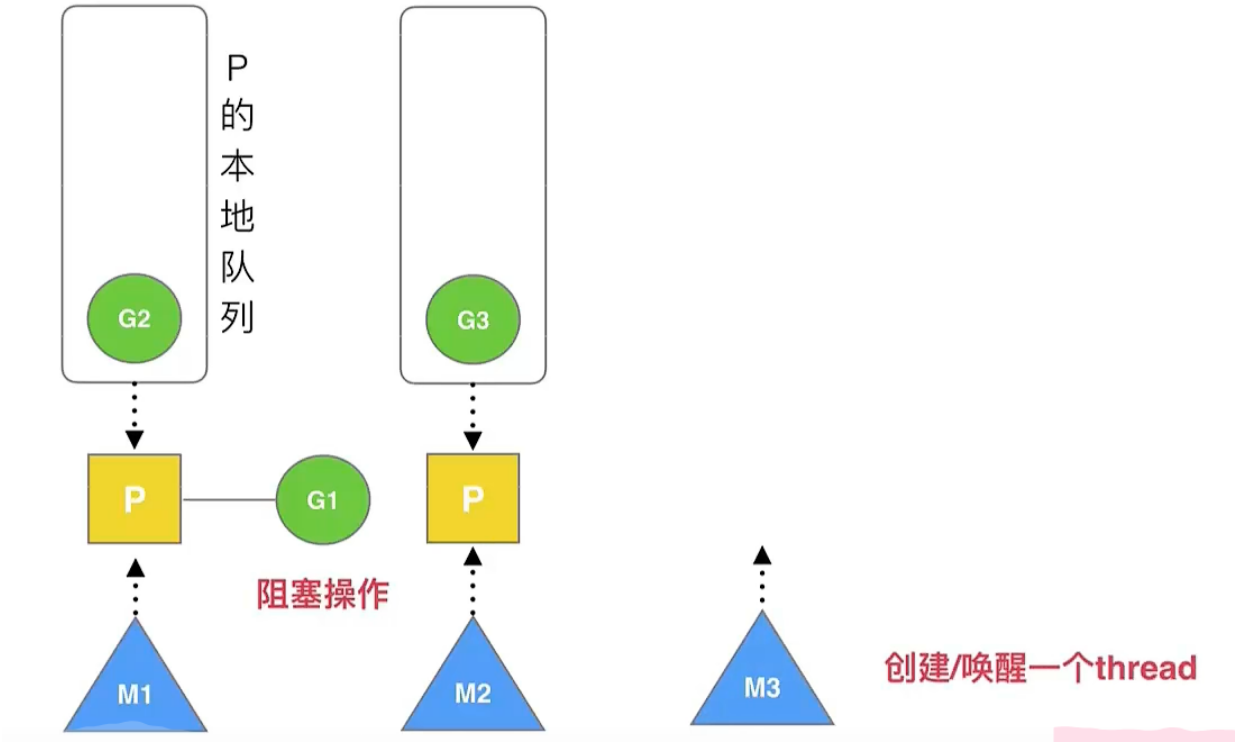

image-20251015161152465

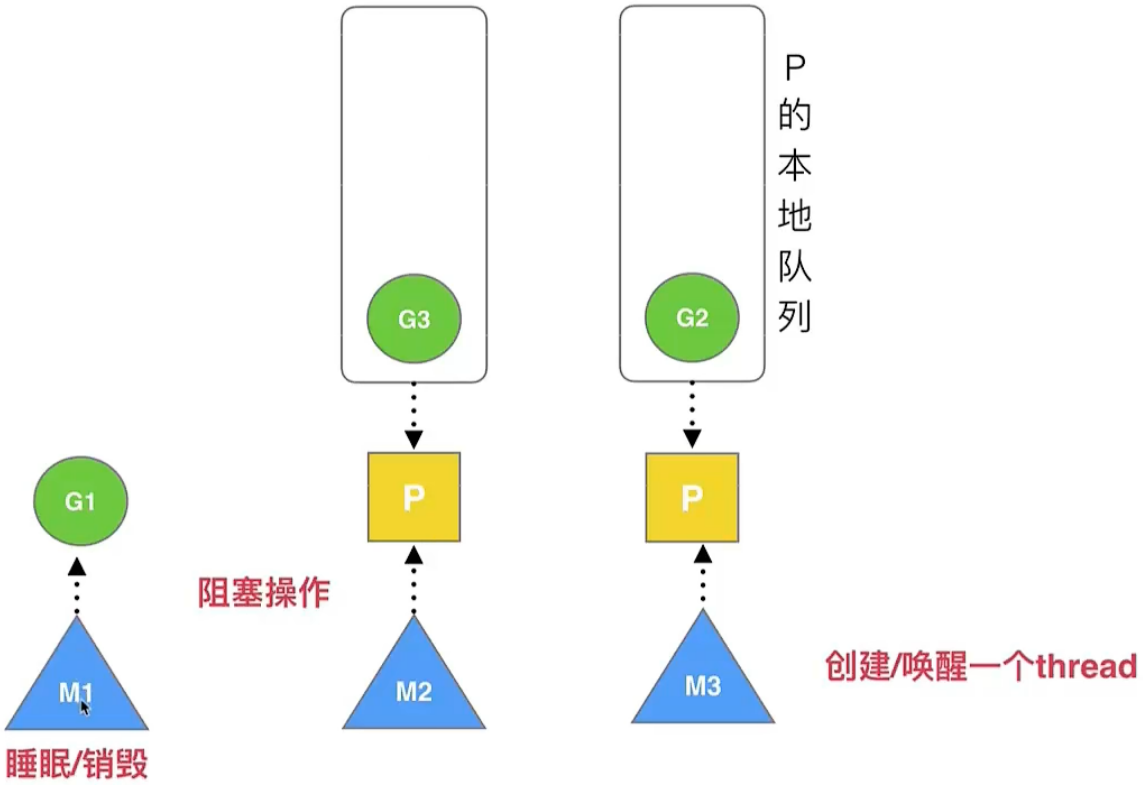

image-20251015161111010

- **利用并行**：通过 `GOMAXPROCS` 限定P的个数，一般约定为**CPU核数/2**
- **抢占**：当 `thread` 和某个 `goroutine` 绑定，且当前 `thread` 被阻塞，**此时只允许** `thread` **等待一定时间**，超过这个时间 `thread` 就会**分配给其它在等待的** `goroutine`

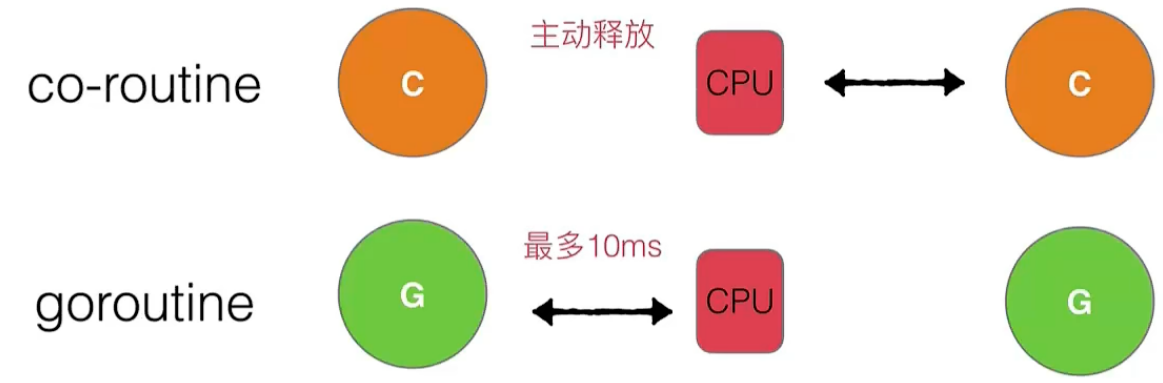

image-20251015165748812

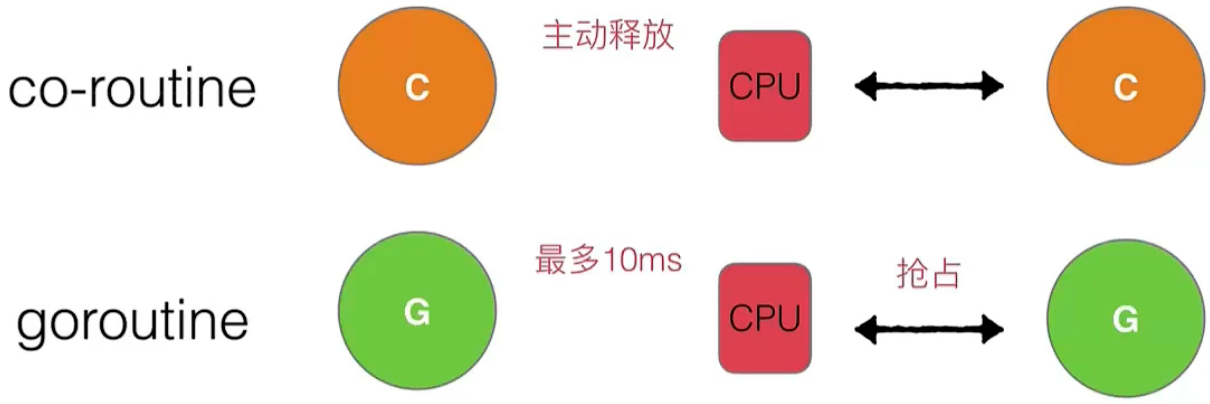

image-20251015165732289

- **全局G队列**：拥有锁的机制
  - 当 `thread` 空闲且其它 `thread` 也没有待处理的协程时， `thread` 就会去全局队列获取协程
  - **全局队列（GRQ）**需要加锁访问，频繁竞争会影响性能。因此Go优先通过P的**本地队列（LRQ）**和 `Work Stealing` 实现无锁调度，仅在 `LRQ` 不足时使用 `GRQ`

> 为什么每个P要绑定一个局部队列，而不是直接使用全局队列?

减小锁竞争，实现无锁化调度，因为每个M从局部队列获取G的过程是无锁的，极其迅速的，如果使用全局队列，就会加剧锁竞争，降低工作效率