Go GMP 调度和内存管理

大明

主要内容

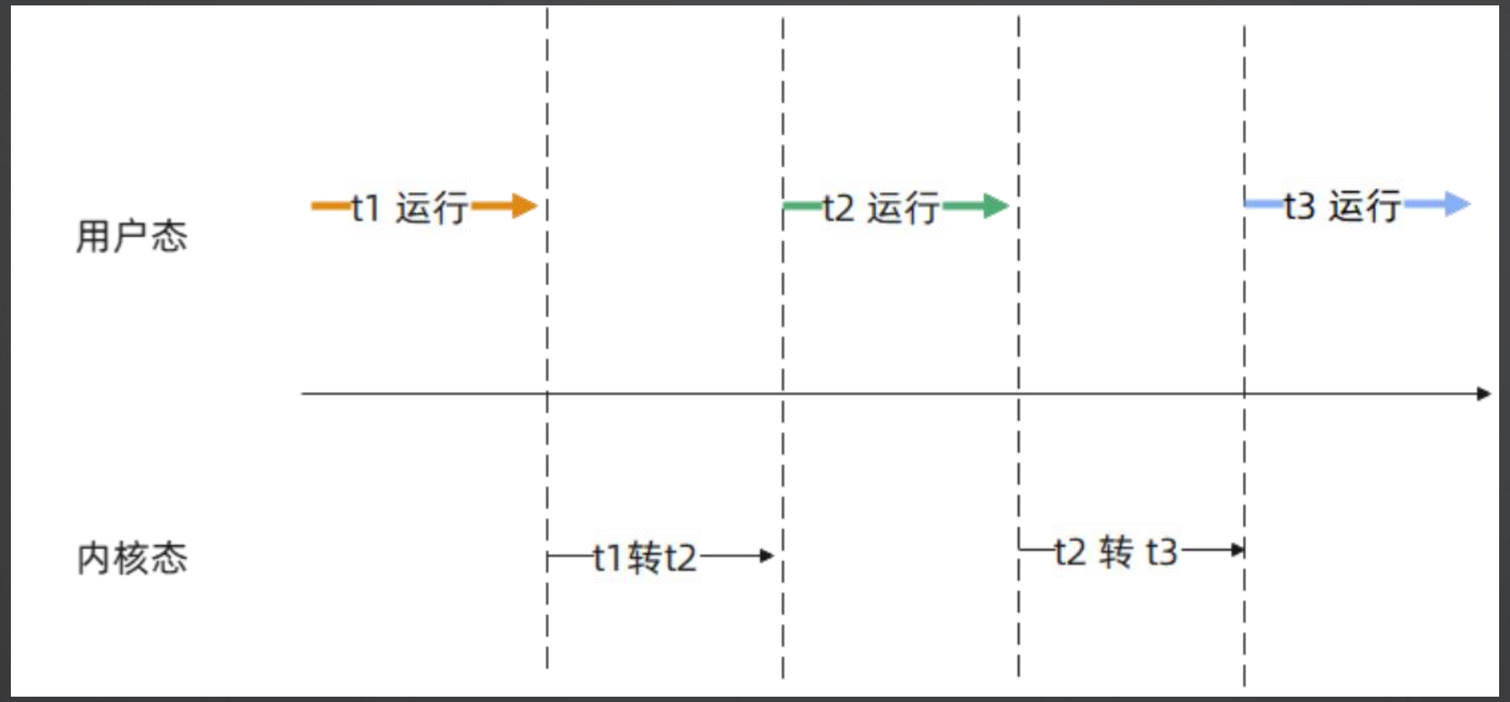
- GMP 调度
- 内存管理
  - 内存分配
  - 垃圾回收

GMP 调度

GMP 调度 —— 什么是协程

goroutine 我们一般称呼为协程（虽然它的作者不承认），要理解协程就得先理解线程。

线程是 CPU 调度的基本单位。线程的创建、销毁、阻塞、唤醒（所有和线程本身有关的操作），都是由操作系统内核来完成的。



GMP 调度 —— 什么是协程

用户态切内核态有什么代价？

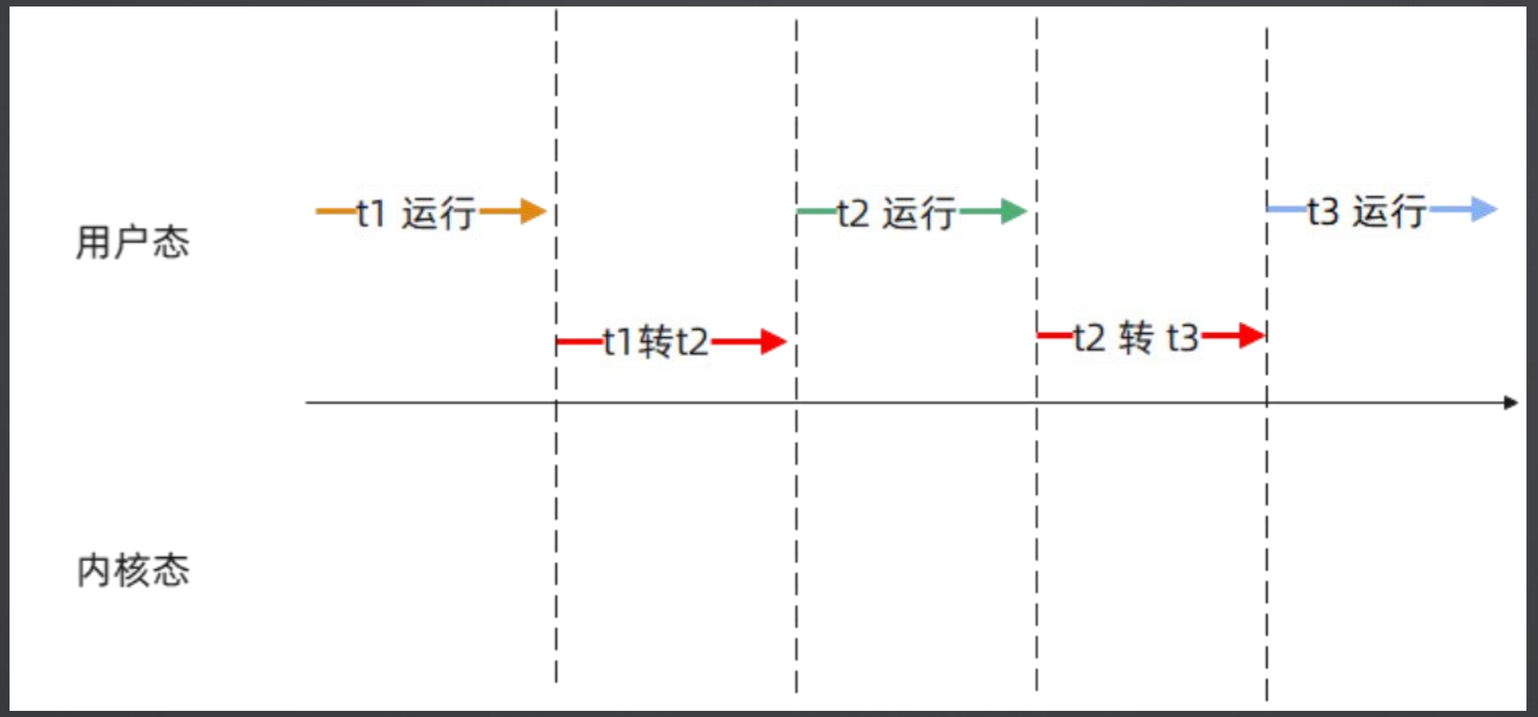
- 保存上下文（寄存器和栈）
  - TLB 刷新
- 将程序计数器设置为内核指令地址
- 执行内核态操作

所以正常情况下，我们都会尽量避免用户态切换

如何可以对内存进行精细的规划，其实不需要操作系统

GMP 调度 —— 什么是协程

很自然的想法就是能不能线程本身也在用户态管理？



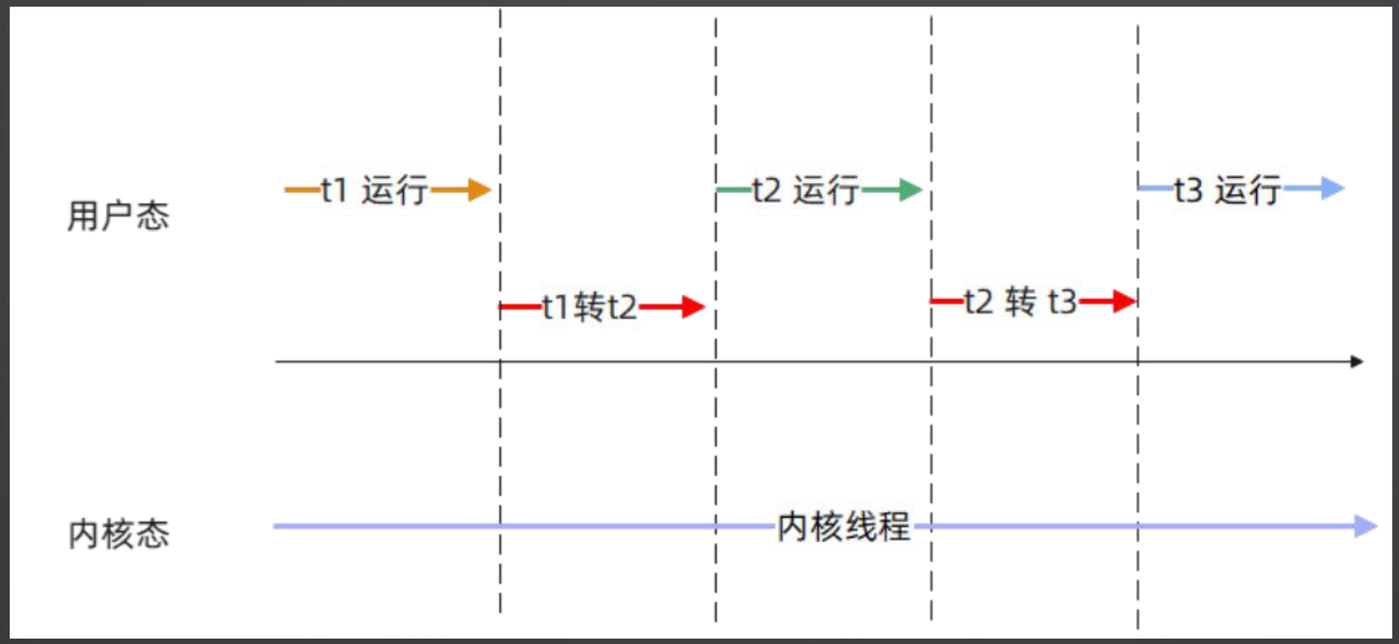
出于

答案是不太行，因为内核这个线程处于安全性考虑是只能由内核态来管的。

GMP 调度 —— 什么是协程

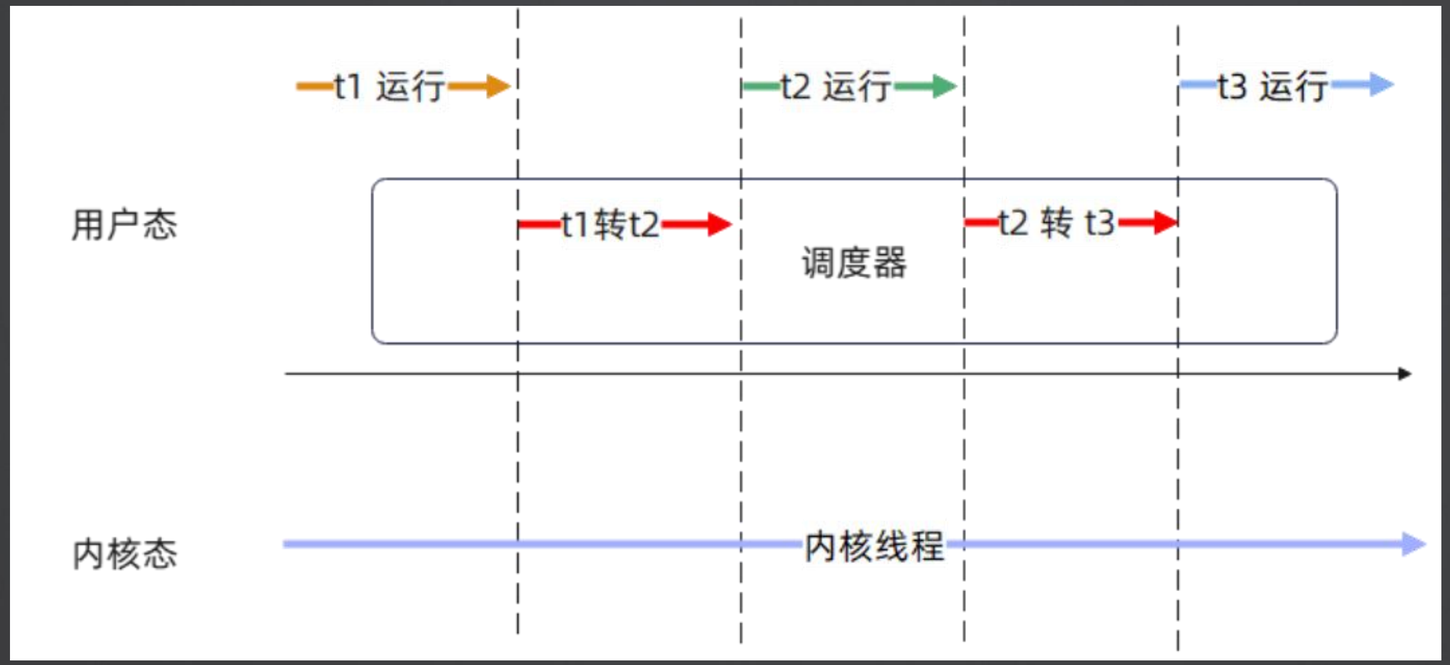
那么我们能不能搞一个用户态的线程，我们自己管？

这个时候, t1, t2, t3 就是我们自己在用户态维护的线程了。



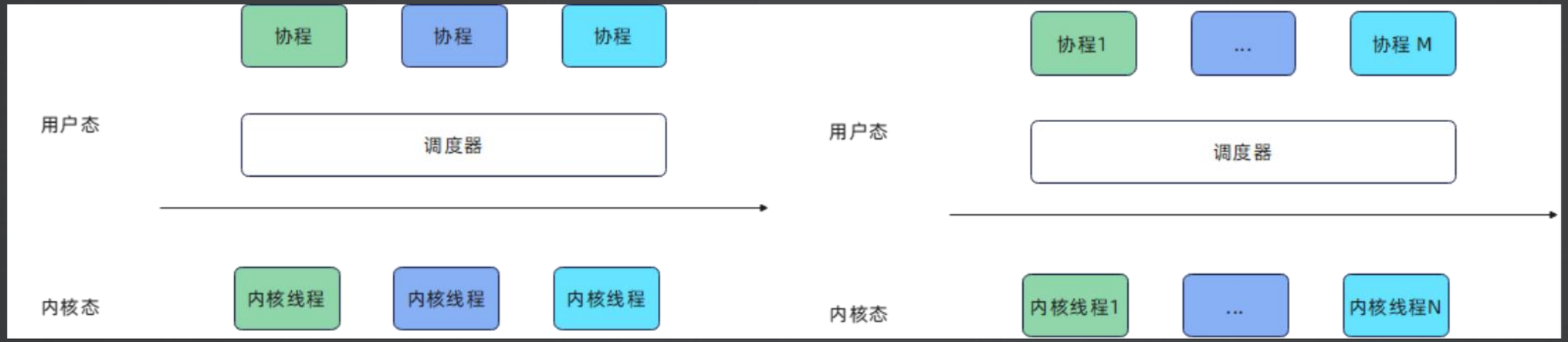
GMP 调度 —— 什么是协程

这个东西就是协程。而后调度逻辑本身，我们称为调度器。



GMP 调度 —— 什么是协程

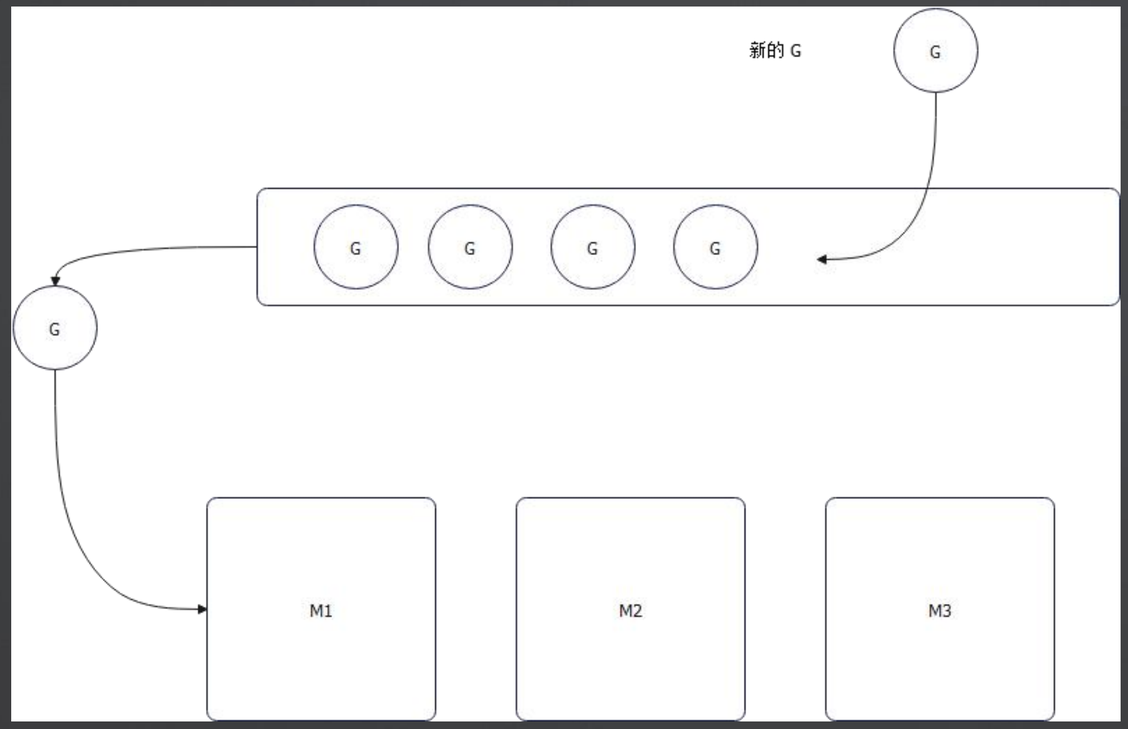
那么协程和线程的关系是什么样的？显然，可以是 1:1 的，也可以是 M:N



GMP 调度 —— 协程和线程怎么合作（GM调度）

最开始，Go 的调度是只有 GM 的，也就是图中的协程（G）和线程（M），还没有那个 P。

你觉得要是你来设计，你会怎么设计？注意，它这边必然是一个 M:N 模型

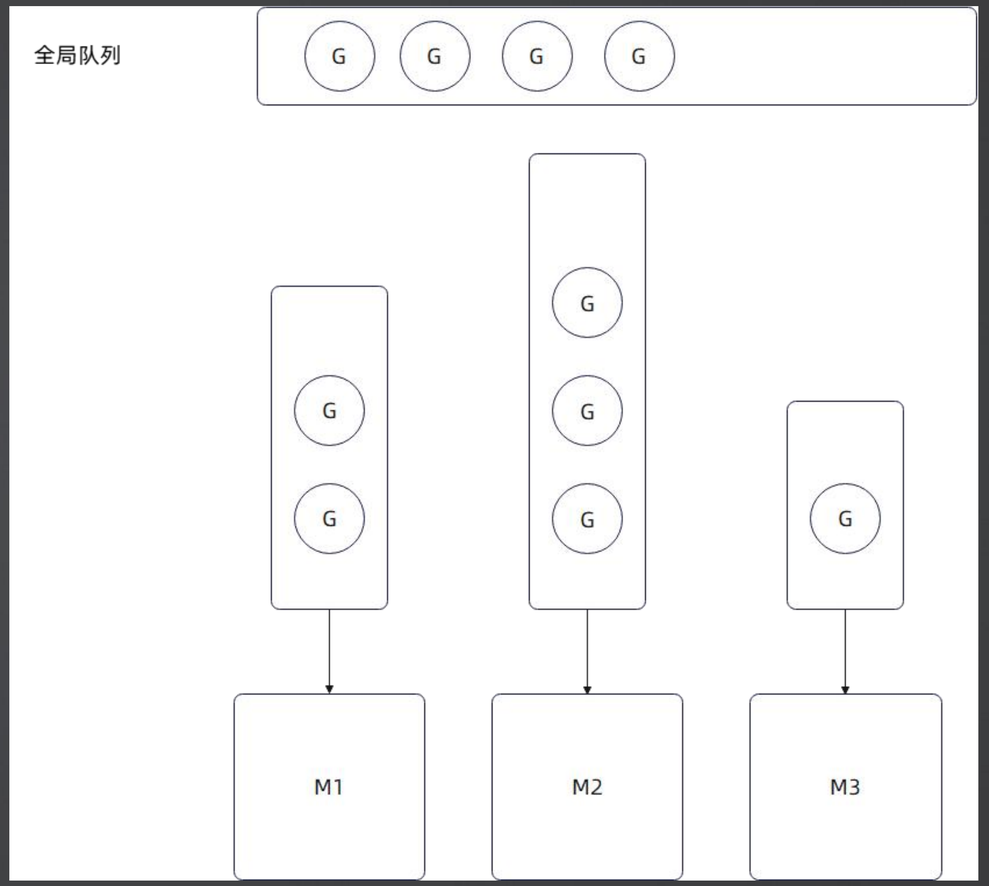


缺点：存在一个全局队列，那么就存在全局竞争。

GMP 调度 —— 协程和线程怎么合作

你肯定会想改进这个东西，比如说能不能每个 M 有一个自己的 G 的队列，来规避全局竞争。那么你的设计看起来就是这样：

M 每次都从全局队列拿一大批（这个时候是全局竞争），然后自己慢慢执行



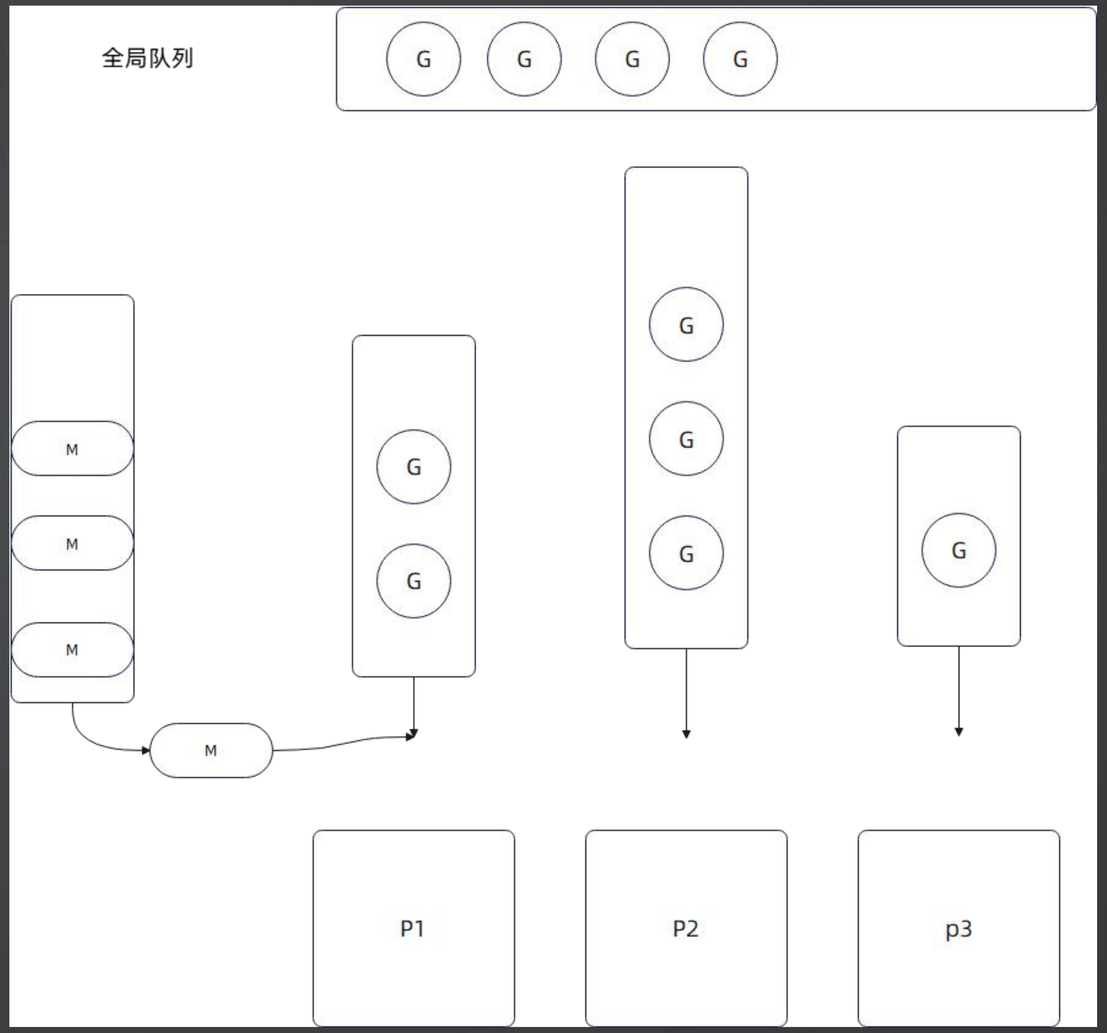
GMP 调度 —— 协程和线程怎么合作

这时候你会想到，M 会在不同的CPU上调度，万一 M 要被阻塞了，它的队列怎么办？

于是你想到，干嘛要绑定到 M 上呢？我绑定到 CPU 上，就不存在切换之类的问题了。

不过操作系统层面上，M 才是调度单位，所以你始终需要一个 M。

于是逼不得已，你只能有一个 GMP 调度了。

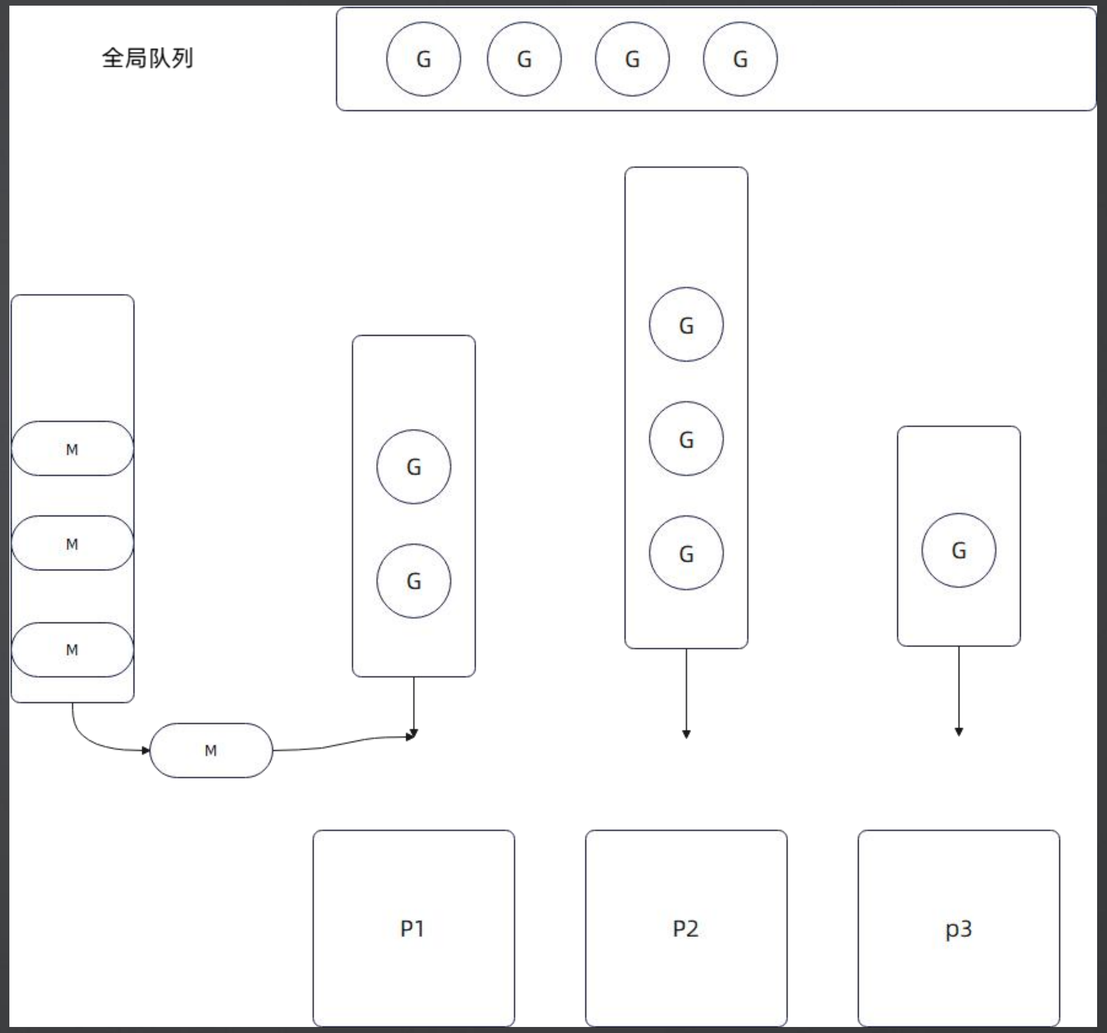


GMP 调度 —— 协程和线程怎么合作

所以基本上，GMP 调度就是指：

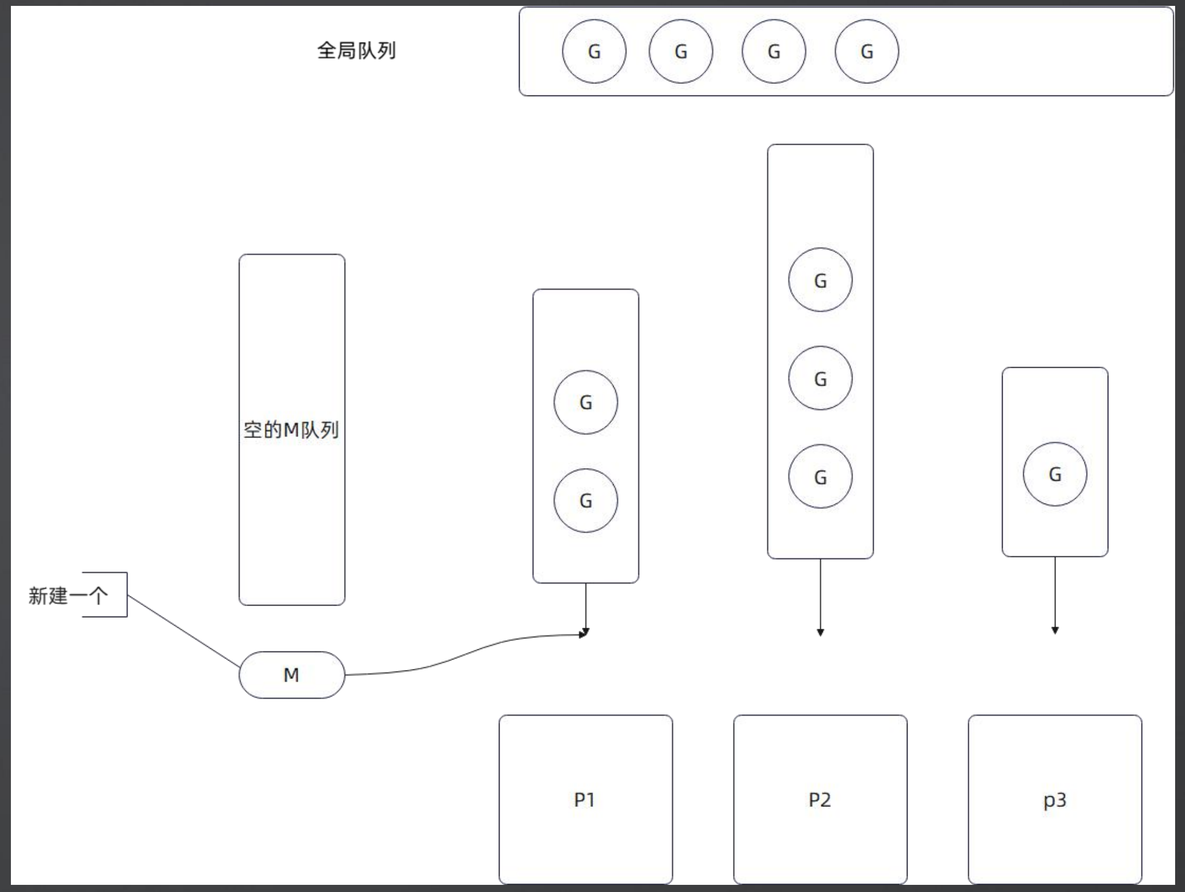
- 你最开始会创建出来多个 P，这个是由 GOMAXPROCS 指定的
- 当 P 想要执行 G 的时候，就从自己的队列里面拿到一个 G
- P 再找一个 M，将 G 绑定到 M 上面，开始执行

记住，G 必然是需要和 M 结合之后才能运行



GMP 调度 —— 协程和线程怎么合作

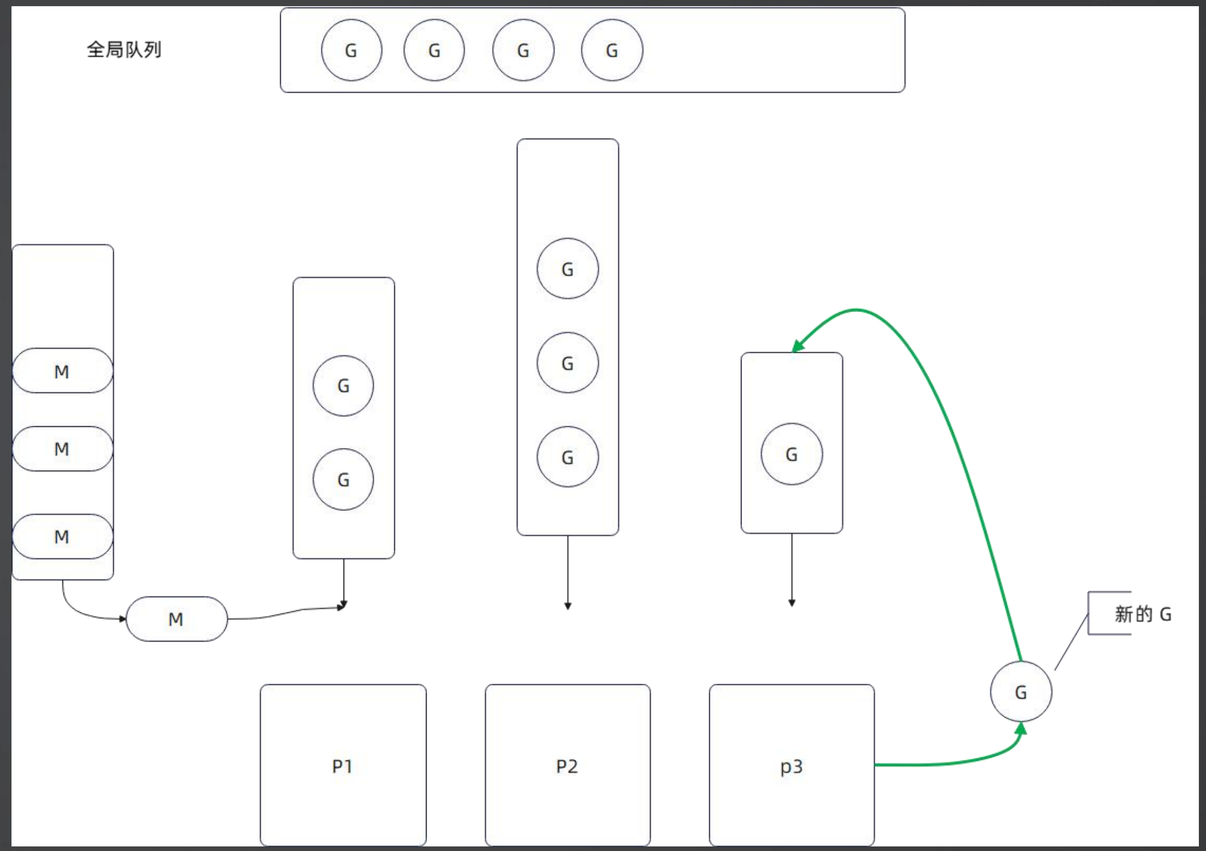
- 如果 M 不够了怎么办？比如说所有的 M 都因为上面绑定的 G 发起系统调用而被阻塞了，那么这时候你只能创建一个新的 M



GMP 调度 —— 协程和线程怎么合作

- 如果我正在运行的 G 又创建了一个 G，那么怎么办？放全局队列还是放当前 P 的队列？

这肯定放当前 P 的局部队列，因为没有任何竞争



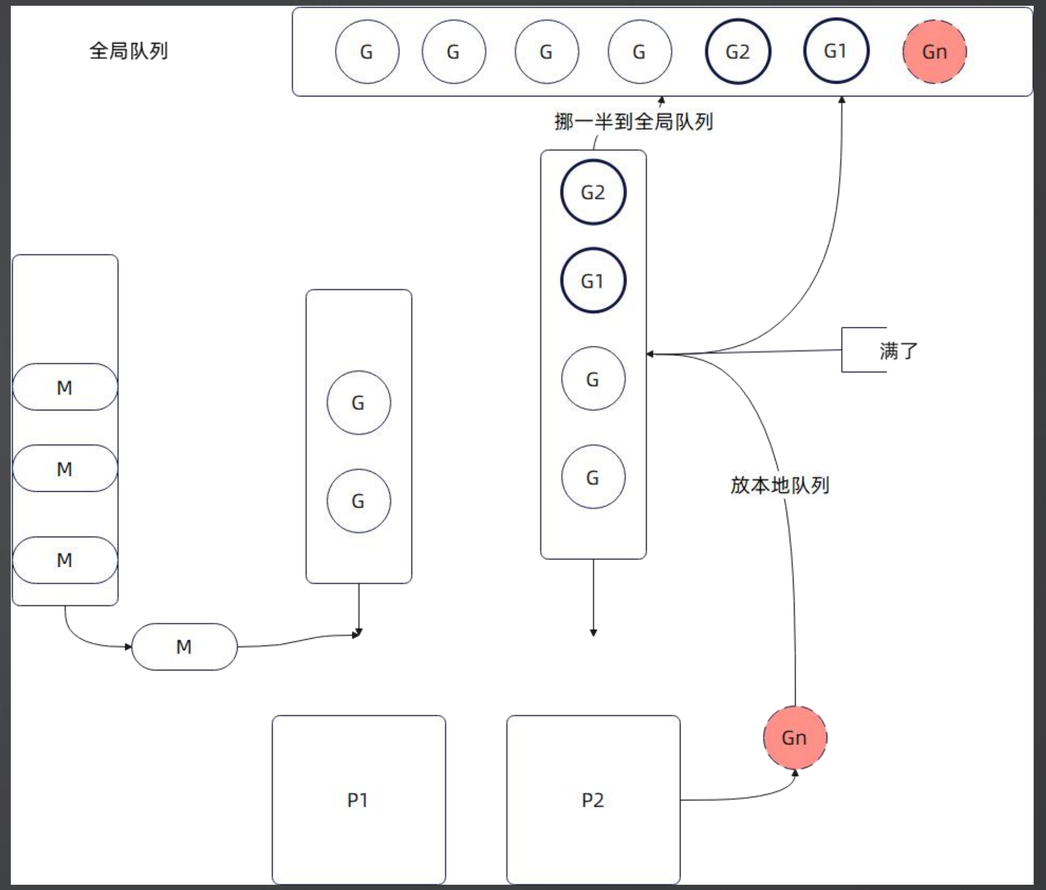
GMP 调度 —— 协程和线程怎么合作

- 那万一 P 自己的队列满了呢？

那就没得选，只能放回去全局队列。

在这个过程，它会把 P 队列里面的一半顺便挪走。
这一半会被重新打乱顺序。

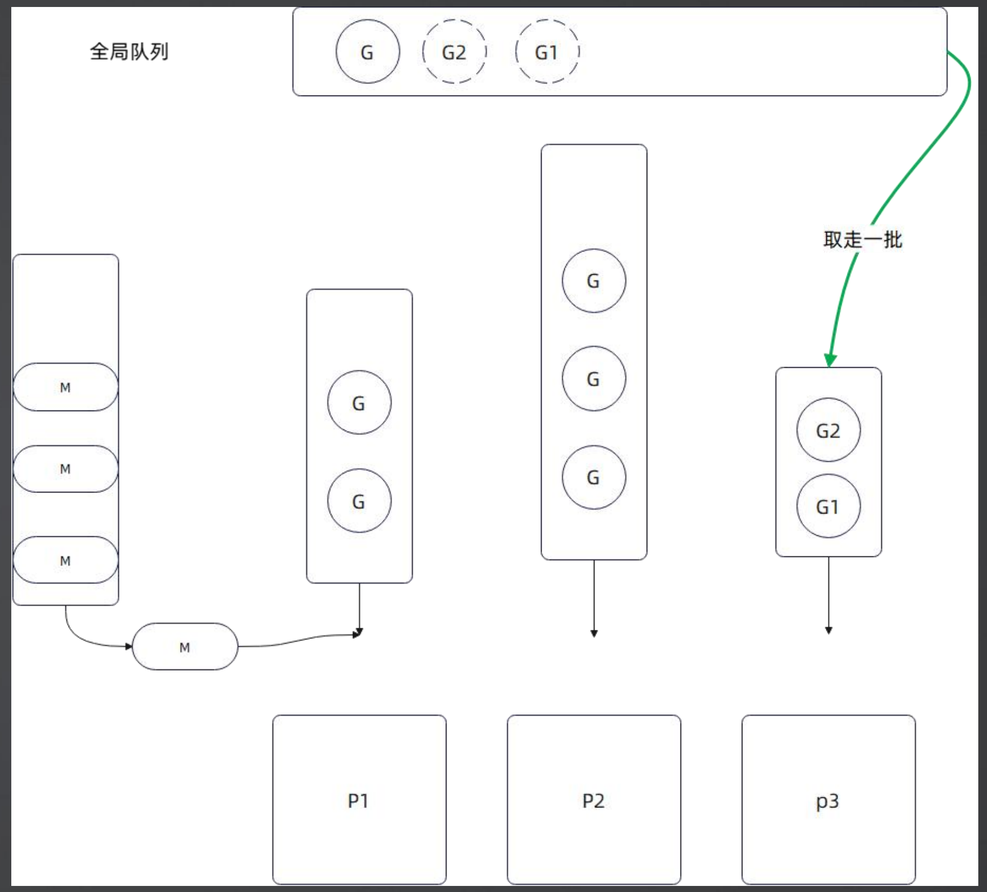
为什么？我猜测是为了避免饥饿。



GMP 调度 —— 协程和线程怎么合作

- 万一我 P3 执行得非常快，本地的 G 都执行完了，怎么办？

自然是从全局队列里面拿一批，这个是全局竞争的。

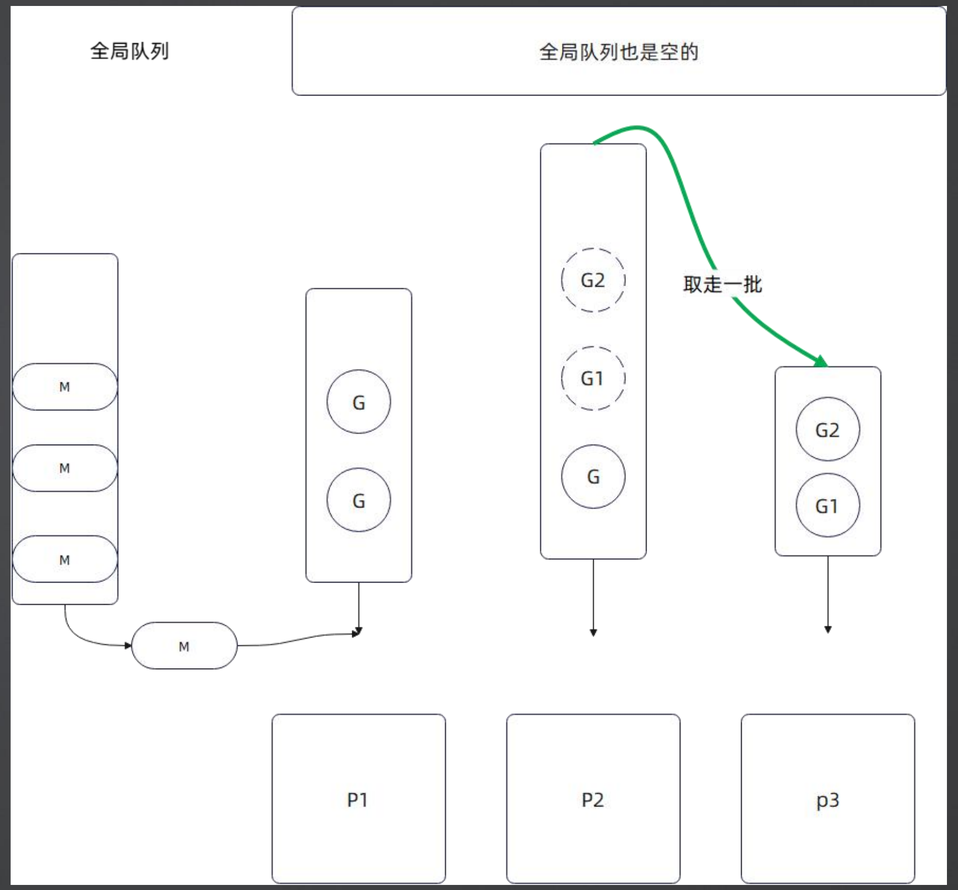


GMP 调度 —— 协程和线程怎么合作

- 万一连全局队列都没了怎么办？

那就是偷一部分别的 P 的队列上的 G，这也就是所谓的工作窃取（work stealing）。

工作窃取在”抢占式”设计里面就非常常见。

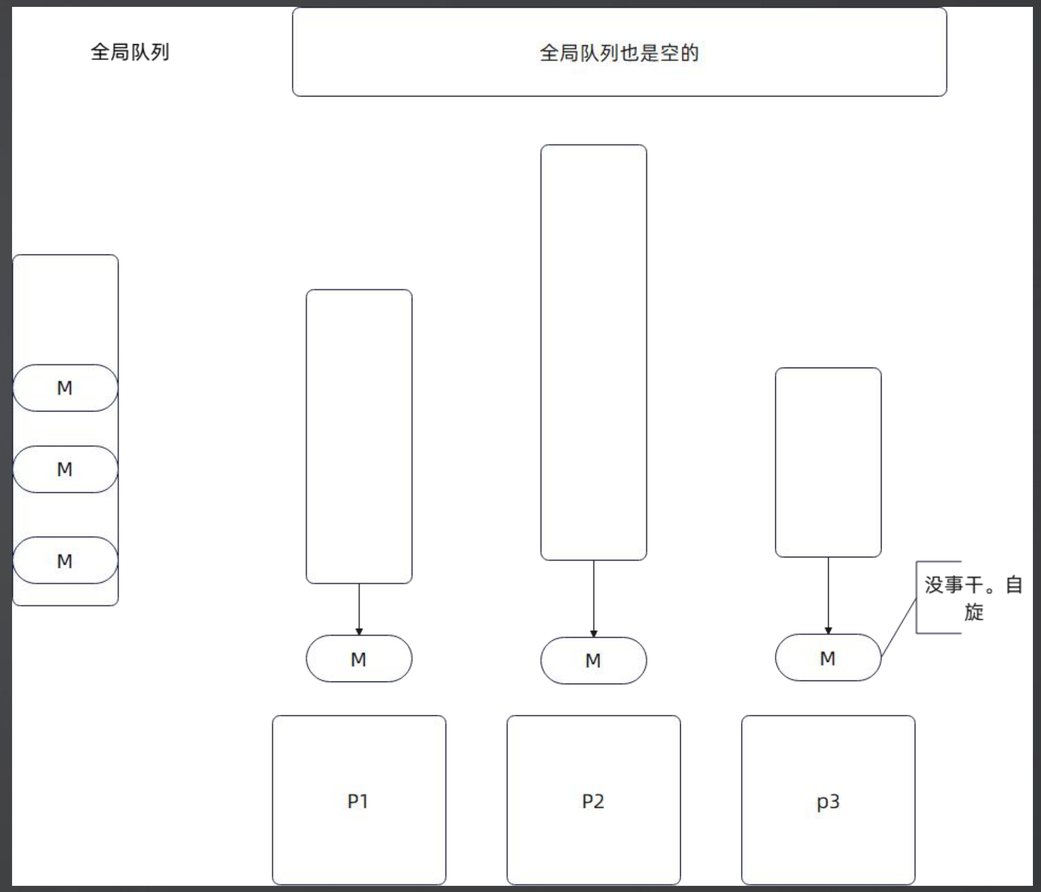


GMP 调度 —— 协程和线程怎么合作

- 如果 G 全没了怎么办？

那么已经绑定了 P 的 M 会自旋等新的 G；没有绑定 P 的 M 会休眠。

因为可以预期短时间内会有新的 G，这样自旋一小会就避免了线程的调度



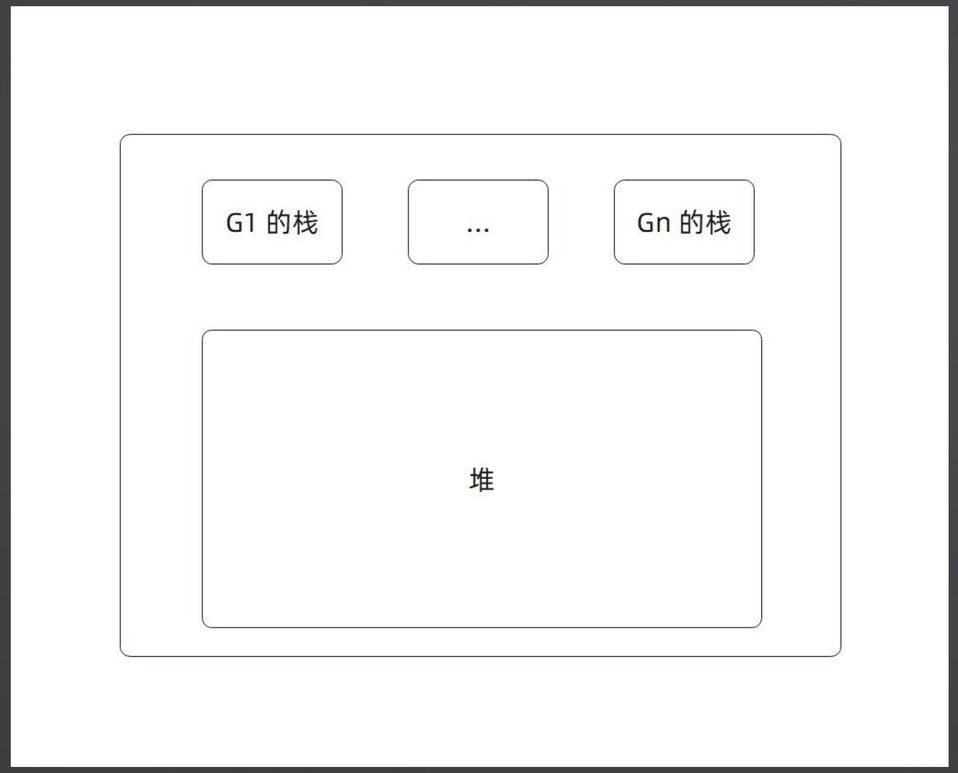
GMP 调度 —— 协程和线程怎么合作

- 什么是 GMP 调度？解释清楚 GMP 的含义，全局队列，本地队列
- 什么是工作窃取？
- 创建一个新的 G 会发生什么？
- 如果一个 P 本地的队列已经空了，会发生什么？
- 如果一个 P 本地队列空了，全局队列也空了，会发生什么？
- 如果一个 P 本地队列空了，全局队列也空了，其它 P 的队列也空了，会发生什么？
- 为什么要从 GM 演进到 GMP？

内存分配

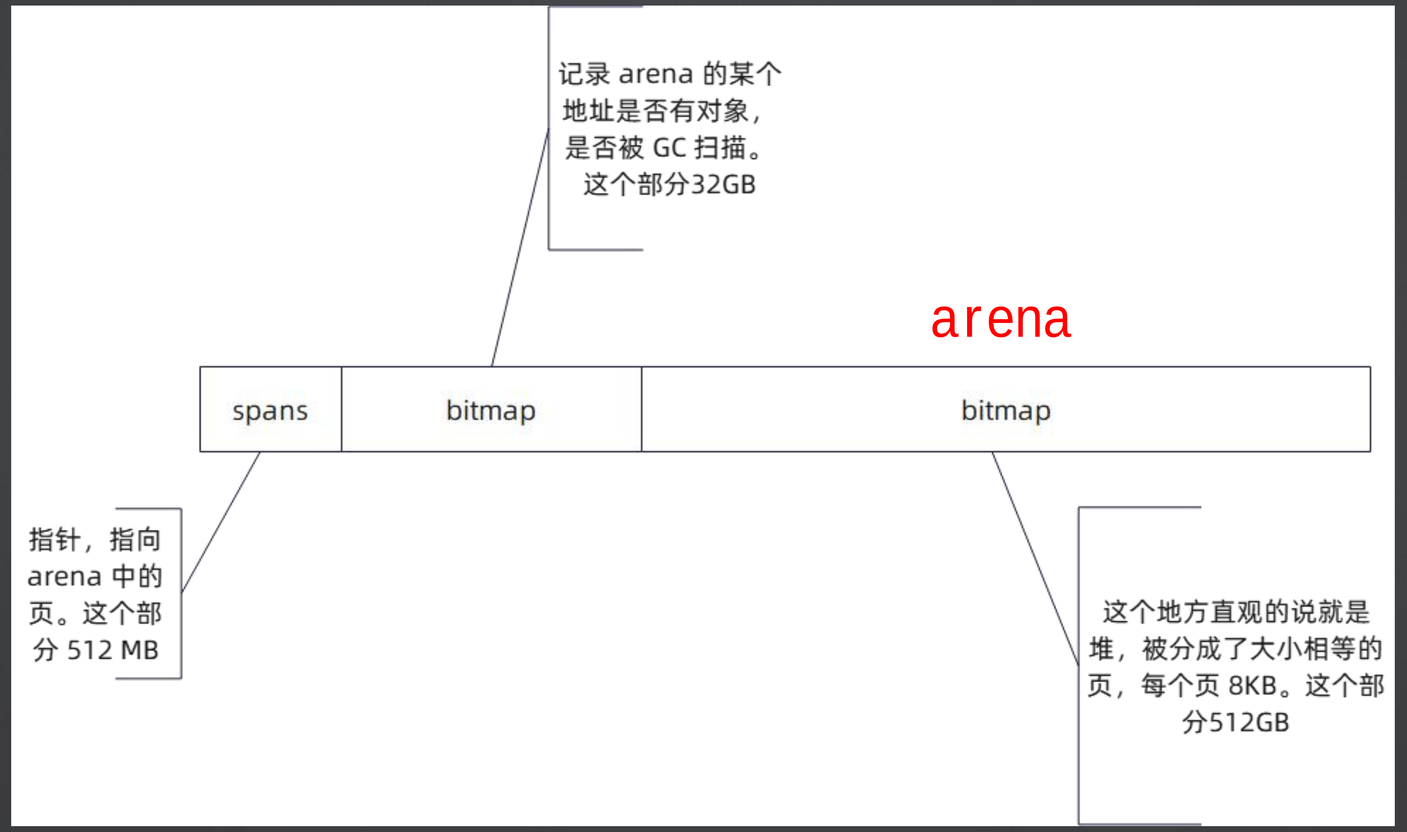
Go 内存分配

即便 Goroutine 再怎么神奇，但是它本质上都是参考了线程来设计的，所以线程有堆栈的概念，那么 goroutine 也是有的。



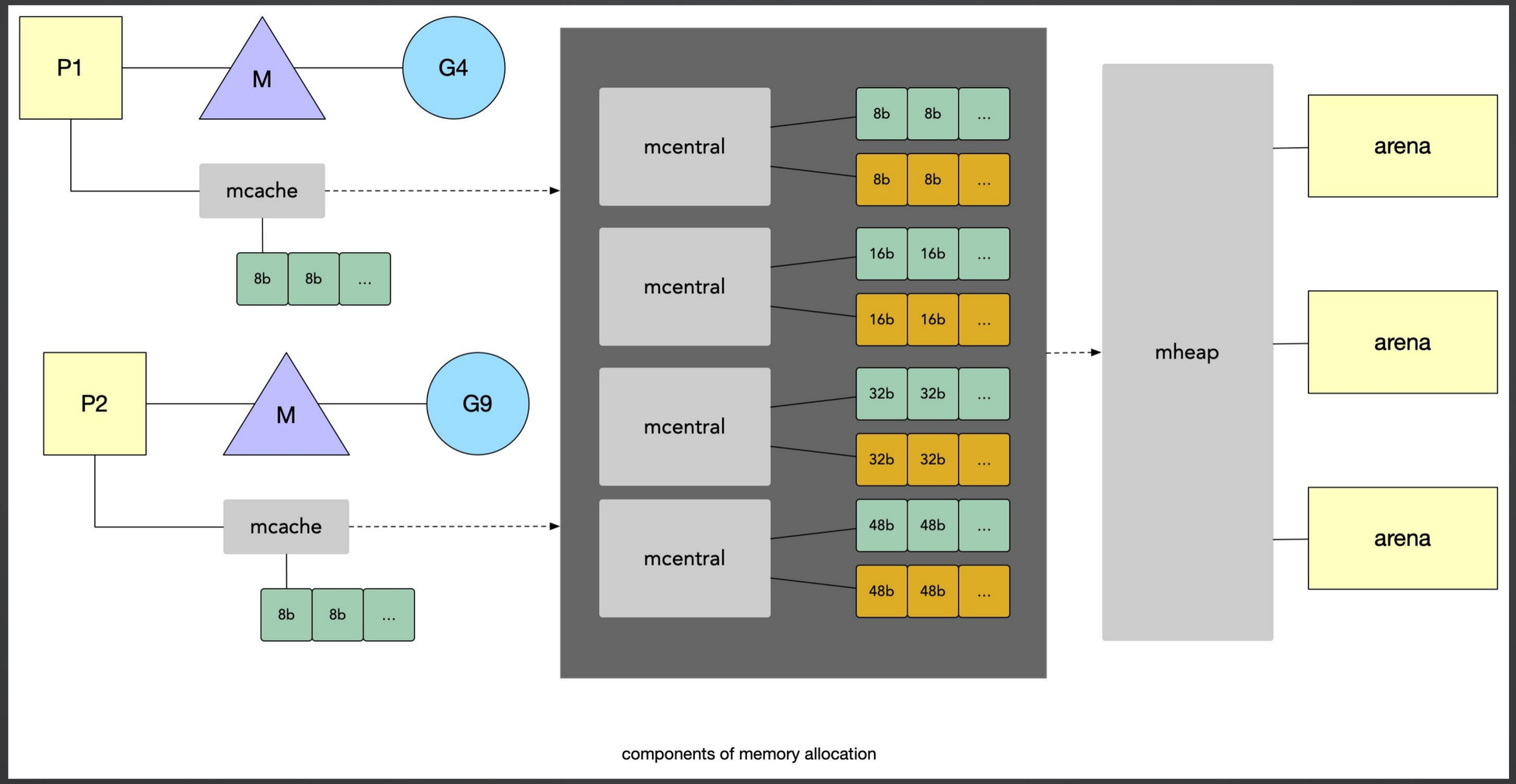
Go 内存分配

在 Go 启动的时候，就会直接申请一块内存，然后分成三个部分：spans, bitmap 和 arena



Go 内存分配

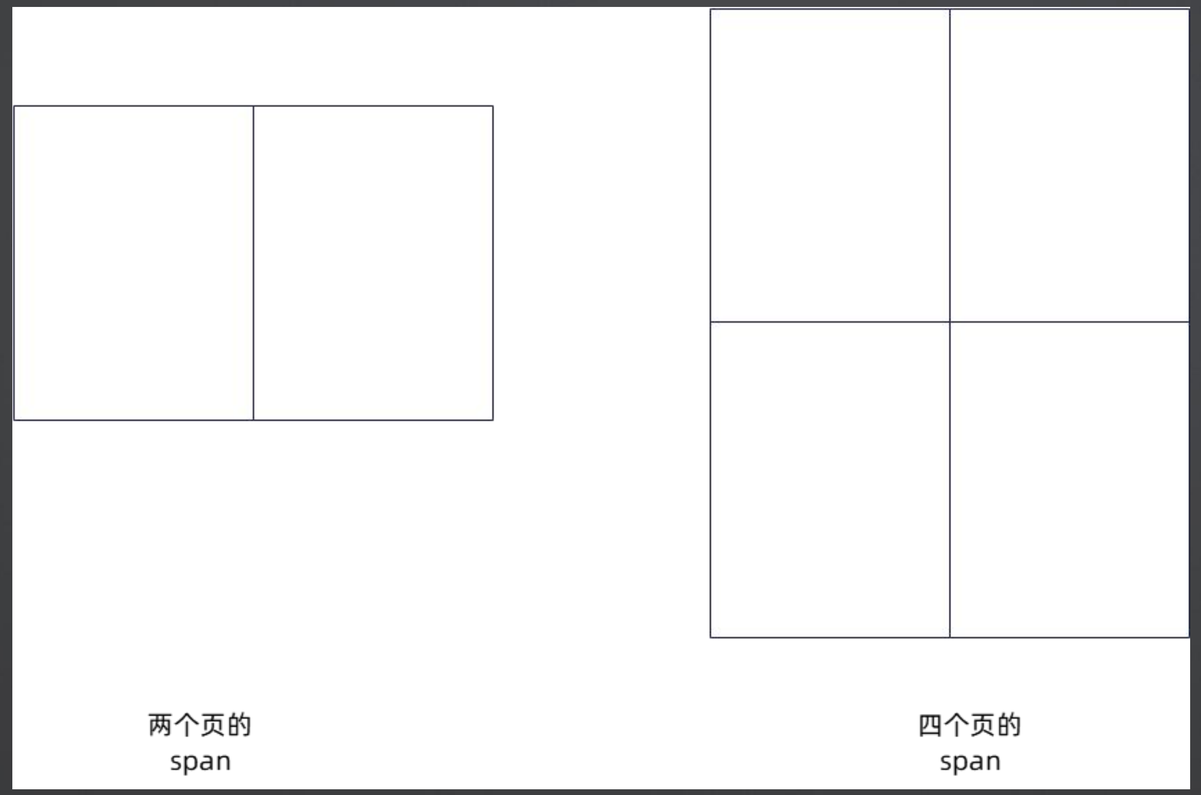
整个和内存分配有关的组件。



components of memory allocation

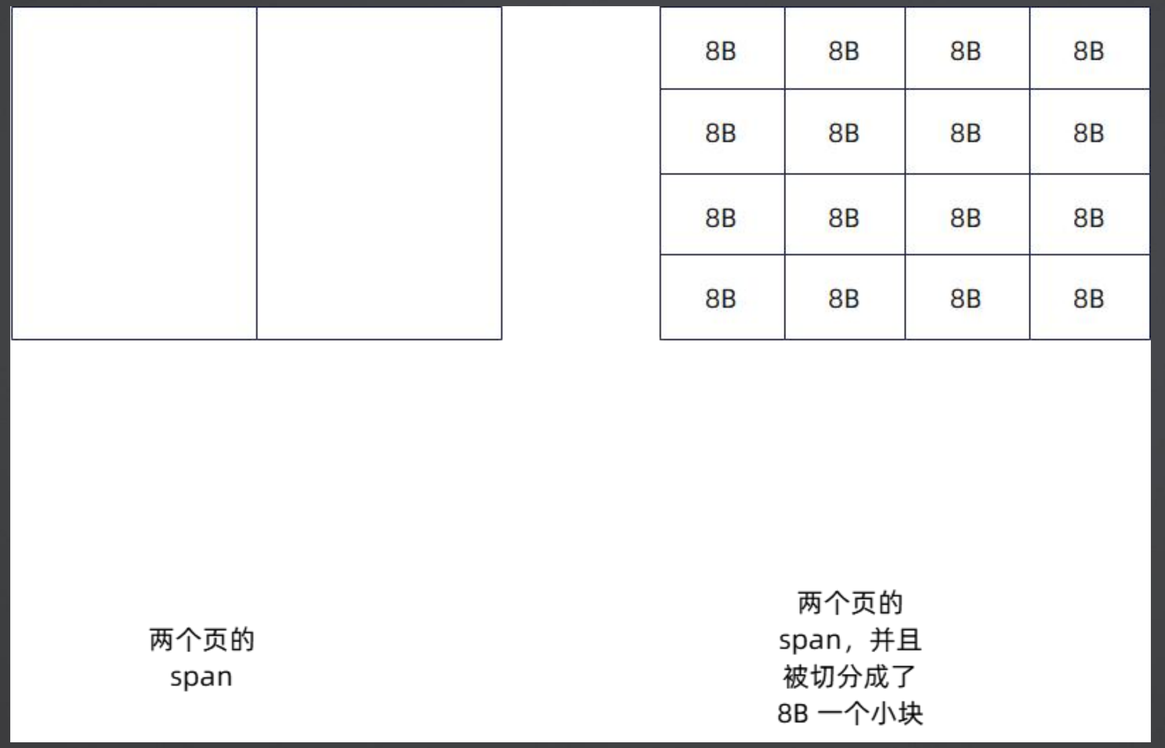
Go 内存分配 —— mspan

首先，span 是指一个或者多个连续的页组成的内存块。同时，一个 span 内部，被切成等大小的小块（slot），比如说 8B, 16B 大小的块



Go 内存分配 —— mspan

首先，span 是指一个或者多个连续的页组成的内存块。同时，一个 span 内部，被切成等大小的小块（slot），比如说 8B, 16B 大小的块



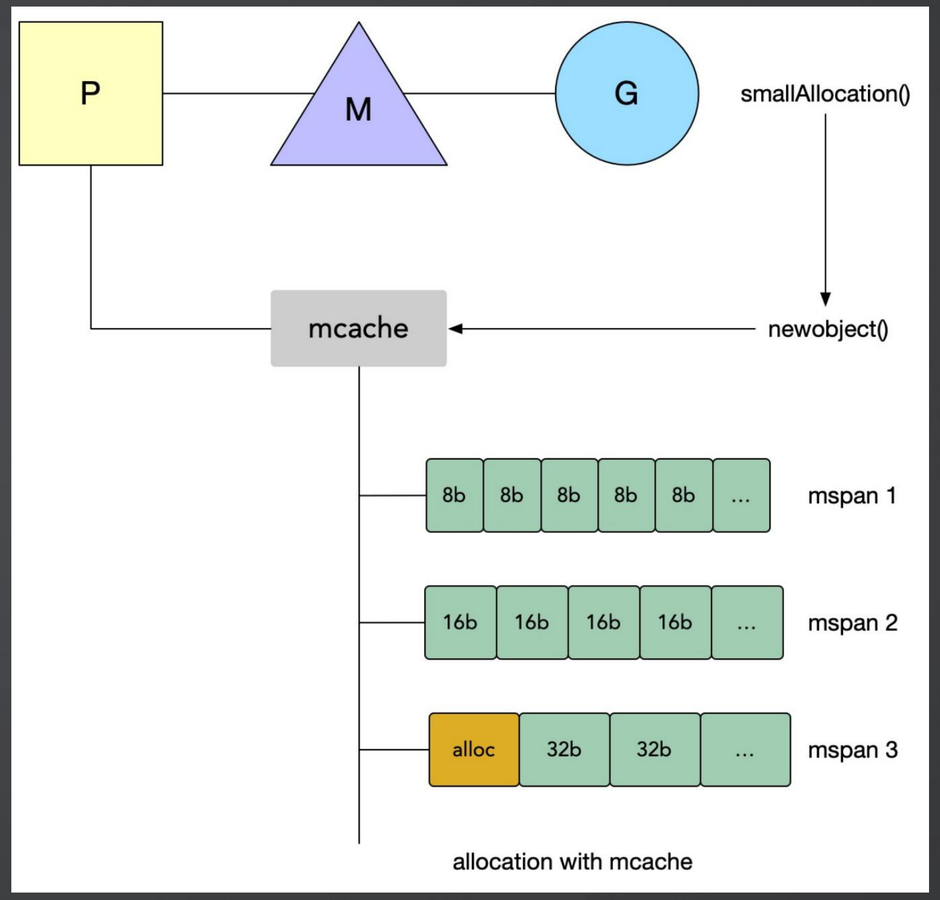
Go 内存分配 —— <= 32 KB

在 P 上面绑定了一个 mcache 的东西，你可以理解为就是一串 span。这串 span 中的 slot 大小是不同的，从 8B 开始，总共有 67 个 span。

右图中的 span 是只有一个页的，理论上来说也可以有多个页。

进一步，如果根据对象里面有没有指针字段，又分成两类。没有指针字段的意味着 GC 的时候不用进行扫描。

所以总共有 67 * 2 个 span。

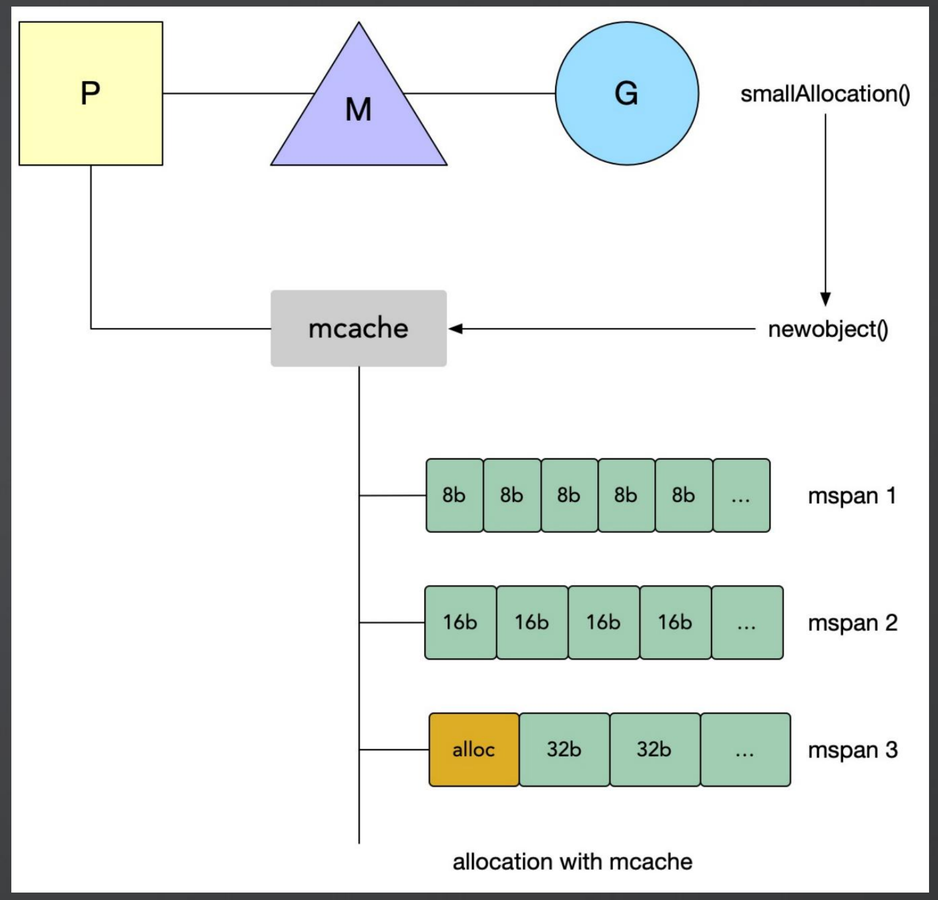


Go 内存分配 —— <= 32 KB

正常来说，就是在 mcache 里面的 span 里面去分配对象。

比如说你新对象大小是在 16 - 32 B 之间，那么就会在 mspan3 里面给你一个 slot。

记住，即便你只有 20 B ，但是也会给你 32 B了。

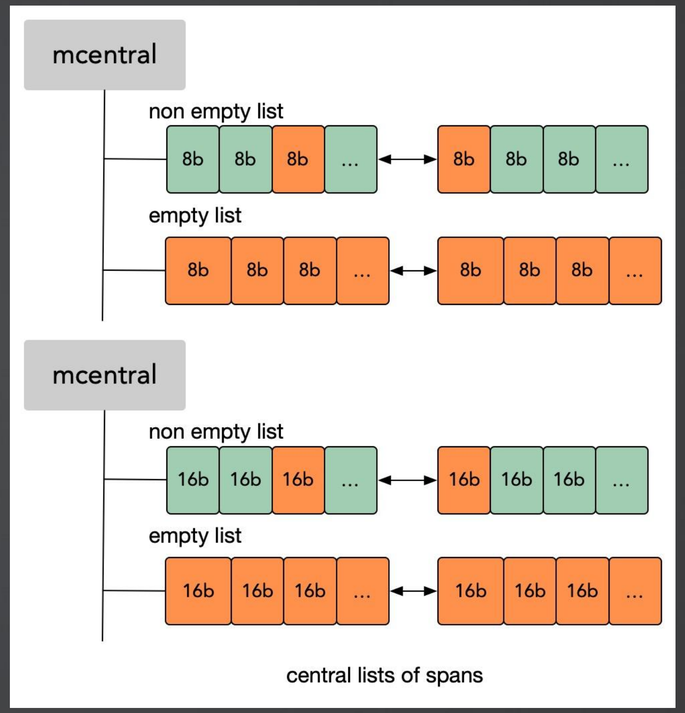


Go 内存分配 —— <= 32 KB

假设说对应 slot 大小的 span 已经全部分配完了，那么就会从 mcentral 里面再分配一个 span 出来（可能部分 slot 被用了）

优先找 non-empty list，这意味着这个 list 里面的 span 中的一部分 slot 是被用过的。

empty list 是已经放满了，没有空闲的 slot 了

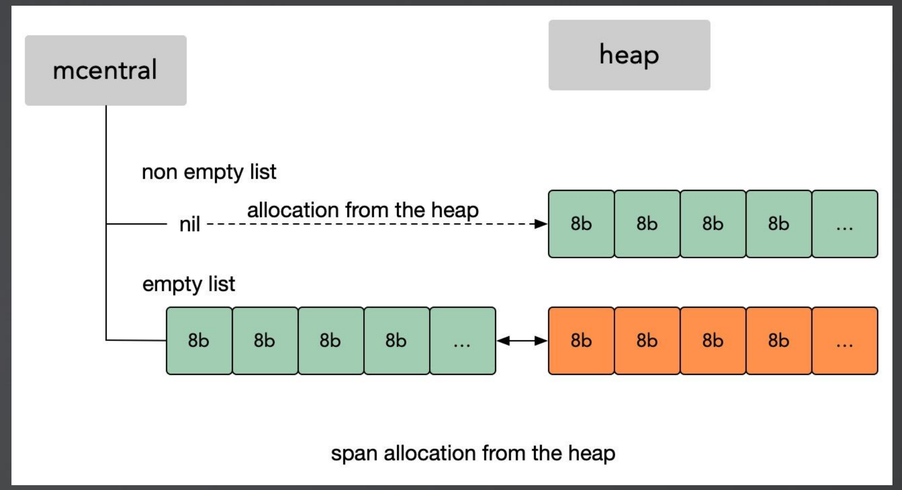


Go 内存分配 —— <= 32 KB

如果 mcentral 都没有，那么就直接从 mheap 里面分配。

mheap 会分配好内存，并且按照目标 slot 的大小切割好这个 span。

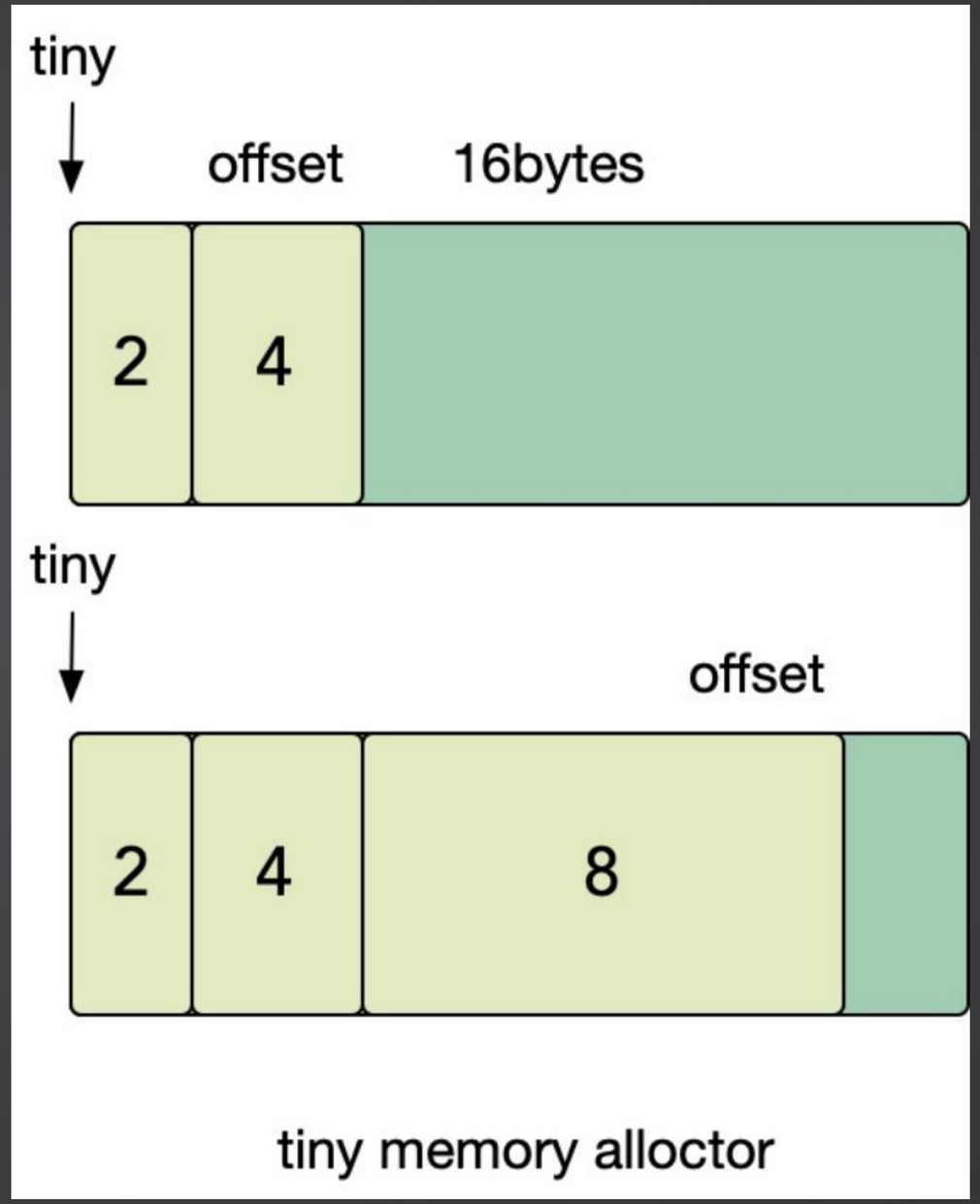
比如说你新对象 20B 一路分配找到了 mheap。那么 mheap 就会准备一个 span，然后将 span 内部按照 32B 均匀切割。



Go 内存分配 —— tiny 分配

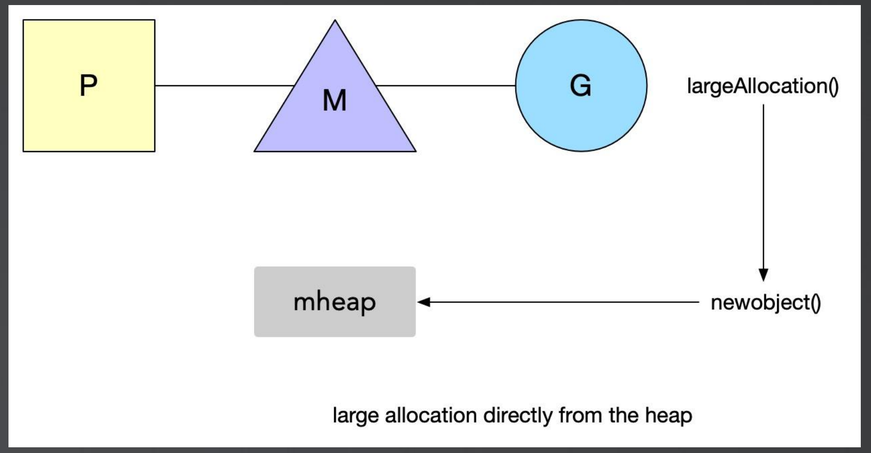
针对一些特别小的对象(<16B)，并且又没有指针的，mcache 里面有一个叫做 tiny 的，就会从这里分配。

这个分配就是你要多少给你多少



Go 内存分配 —— >32KB

直接从 mheap 里面分配。比如说你新对象有 33 KB，那么就会给你五个页（40KB）组成一个 span。



Go 内存分配 ——总结

1. < 16B 并且无指针，走 tiny
2. 否则，<= 32KB，沿着 mcache -> mcentral -> mheap -> 操作系统 找有空闲 slot 的 span，分配内存；
3. > 32KB，直接 mheap 分配；

面试题

- GO 怎么分配内存的
- 解释一下 mspan, mcache, mcentrel 和 mheap 的概念
- 什么是 tiny 分配器，什么样的对象会使用 tiny 分配器？
- 什么样的是大对象？大对象 Go 怎么分配内存？

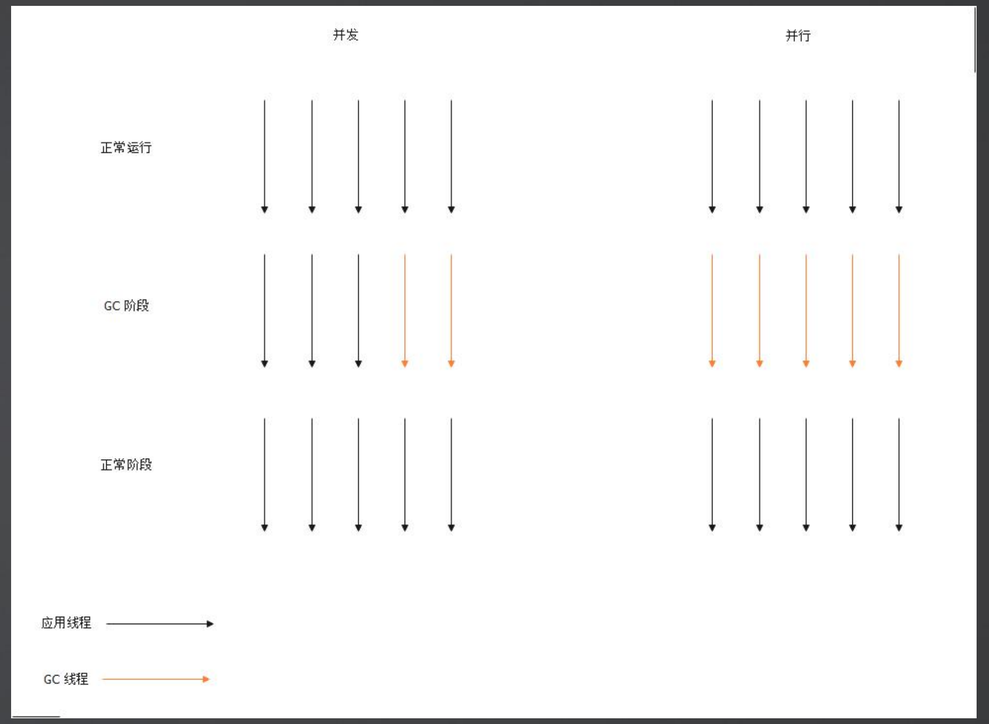
内存回收

Go 垃圾回收

Go 的垃圾回收算法叫做并发标记清扫算法。

- 并发强调的在垃圾回收过程中，应用程序也在运行
- 标记意味着 Go 使用了可达性分析来标记对象，而不是引用计数

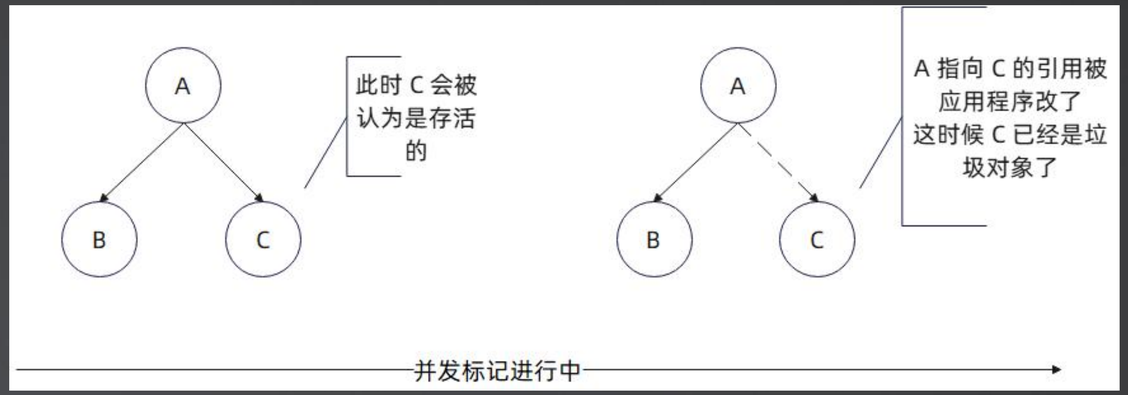
注意区别在垃圾回收语境下，并发强调的是垃圾回收线程（goroutine）和应用程序线程（goroutine）是并发的。



Go 垃圾回收 —— 基本步骤

- 初始标记阶段：扫描 GC root（STW的）
- 并发标记阶段：标记 goroutine 和应用程序 goroutine 并发运行
- 再标记阶段：重新标记在并发标记阶段发生了引用变更的对象（STW 的）
- 清扫阶段：未被标记的对象就被认为是垃圾对象，会被回收

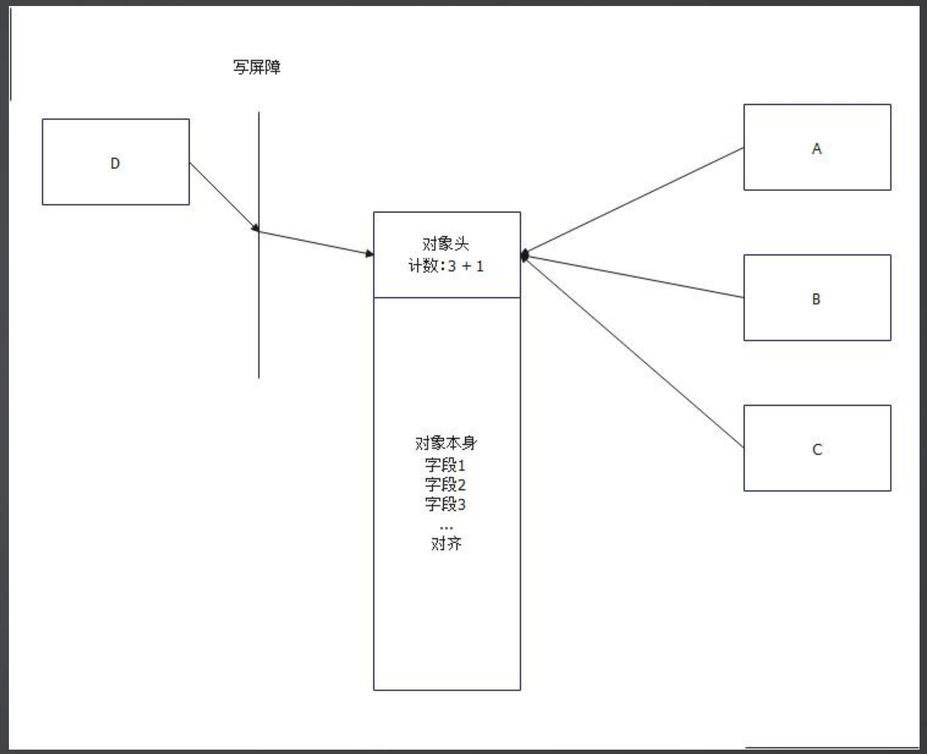
STW： Stop The World，即应用程序完全停下来



Go 垃圾回收 —— 标记

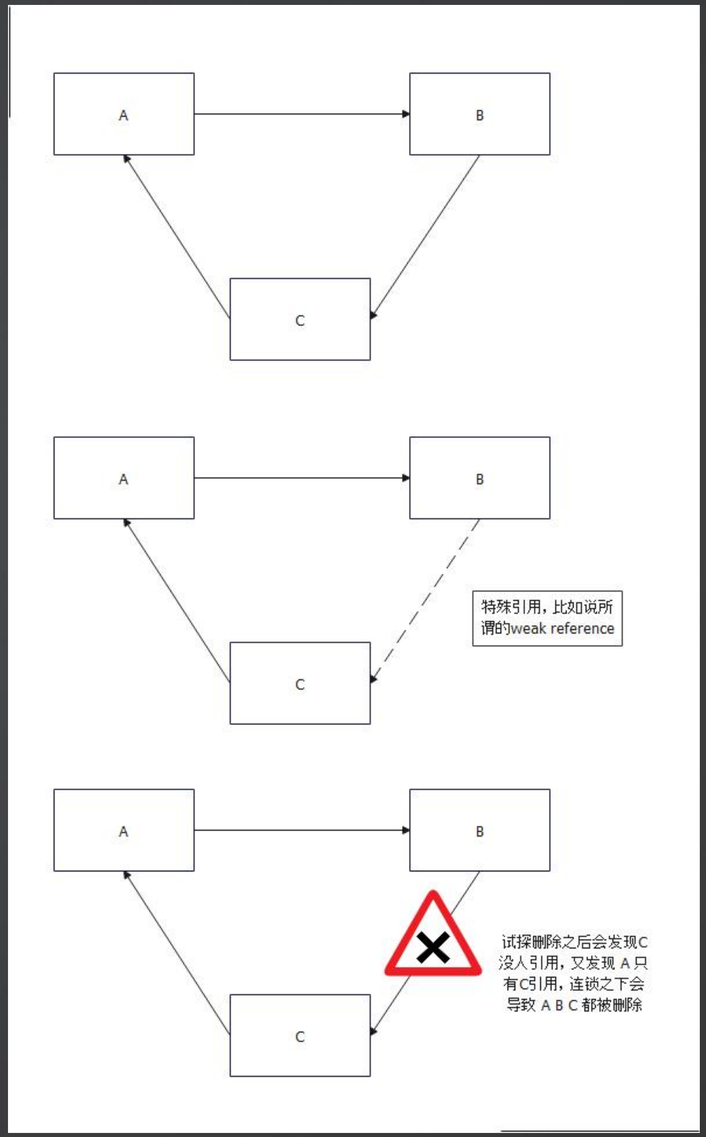
从垃圾回收的理论上来说，判断对象是否存活有两种方法：

- 引用计数法：
  - 为每一个对象维持一个引用计数。利用写屏障来维持计数；
  - 引用计数大多数时候都是在对象头（或者对象元数据）里面开辟一个字段来维持
  - 循环引用问题难以解决
- 可达性分析（追踪法）



引用计数的循环引用问题

- 特殊引用：基本上依赖于用户自己解决。在可能成环的地方使用一些特殊引用，这种引用不会导致计数增加
- 后备追踪器：在可能出现环的地方，用可达性分析进一步分析；
- 试探删除策略



Go 垃圾回收——可达性分析与三色标记

可达性分析（追踪法）：最为典型的就是三色标记法。

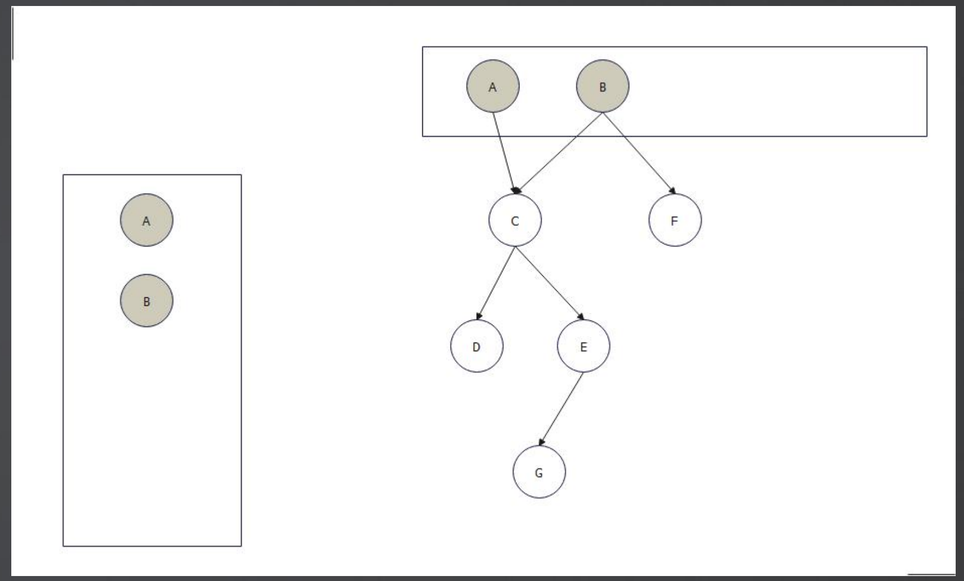
白色 - 没有被标记（垃圾对象）；

灰色 - 正在被标记；

黑色 - 已经标记（存活）

简要步骤（用于辅助记忆，不含细节）

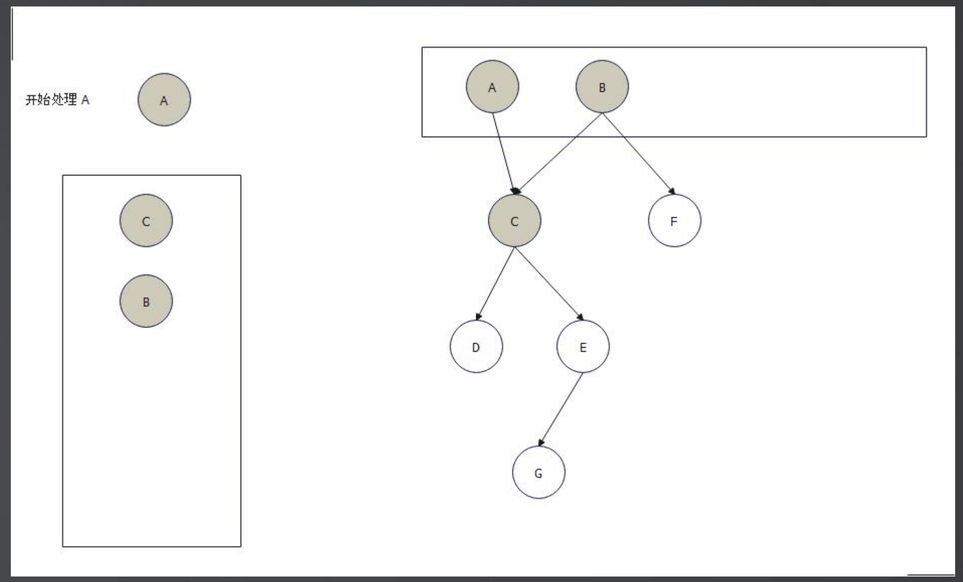
1. 最开始，所有对象都是白色的
2. GC root 出发，把 GC Root 标记为灰色
3. 挑一个灰色对象进一步处理
4. 把该对象引用的对象标记为灰色，加入待处理栈
5. 把当前对象标记为黑色，结束
6. 重复 3-5，直到所有的灰色对象都处理了



Go 垃圾回收——可达性分析与三色标记

可达性分析（追踪法）：最为典型的就是三色标记法。
简要步骤（用于辅助记忆，不含细节）

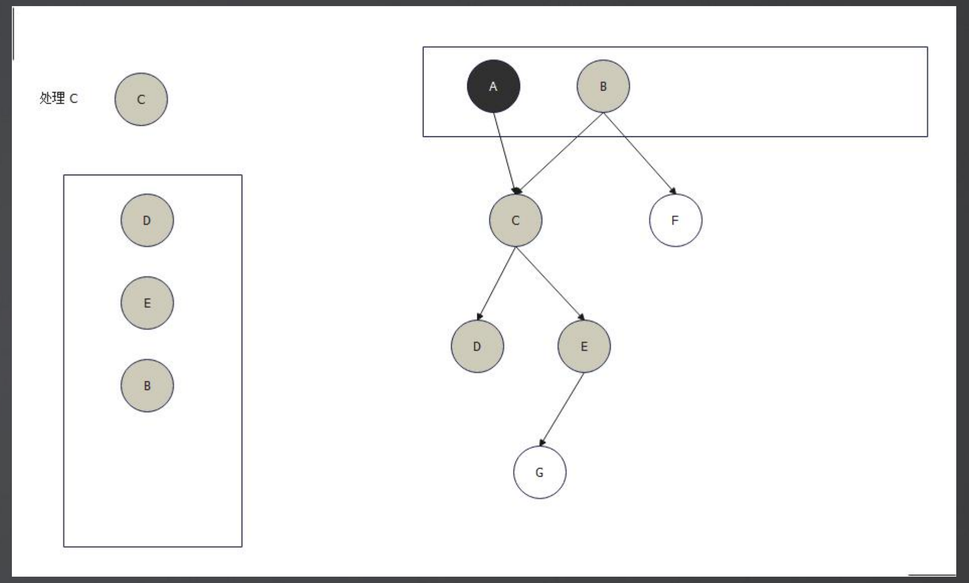
1. 最开始，所有对象都是白色的
2. GC root 出发，把 GC Root 标记为灰色
3. 挑一个灰色对象进一步处理
4. 把该对象引用的对象标记为灰色，加入待处理栈
5. 把当前对象标记为黑色，结束
6. 重复 3-5，直到所有的灰色对象都处理了



Go 垃圾回收——可达性分析与三色标记

可达性分析（追踪法）：最为典型的就是三色标记法。
简要步骤（用于辅助记忆，不含细节）

1. 最开始，所有对象都是白色的
2. GC root 出发，把 GC Root 标记为灰色
3. 挑一个灰色对象进一步处理
4. 把该对象引用的对象标记为灰色，加入待处理栈
5. 把当前对象标记为黑色，结束
6. 重复 3-5，直到所有的灰色对象都处理了

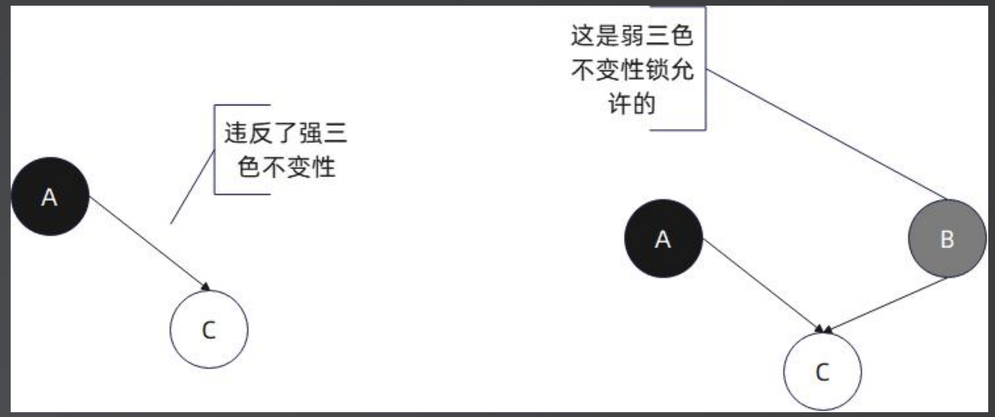


Go 垃圾回收——强弱三色不变性

因为在并发标记过程中，应用程序可能会修改已经标记过的对象，为了解决这个问题，引入了强弱三色不变性规则。

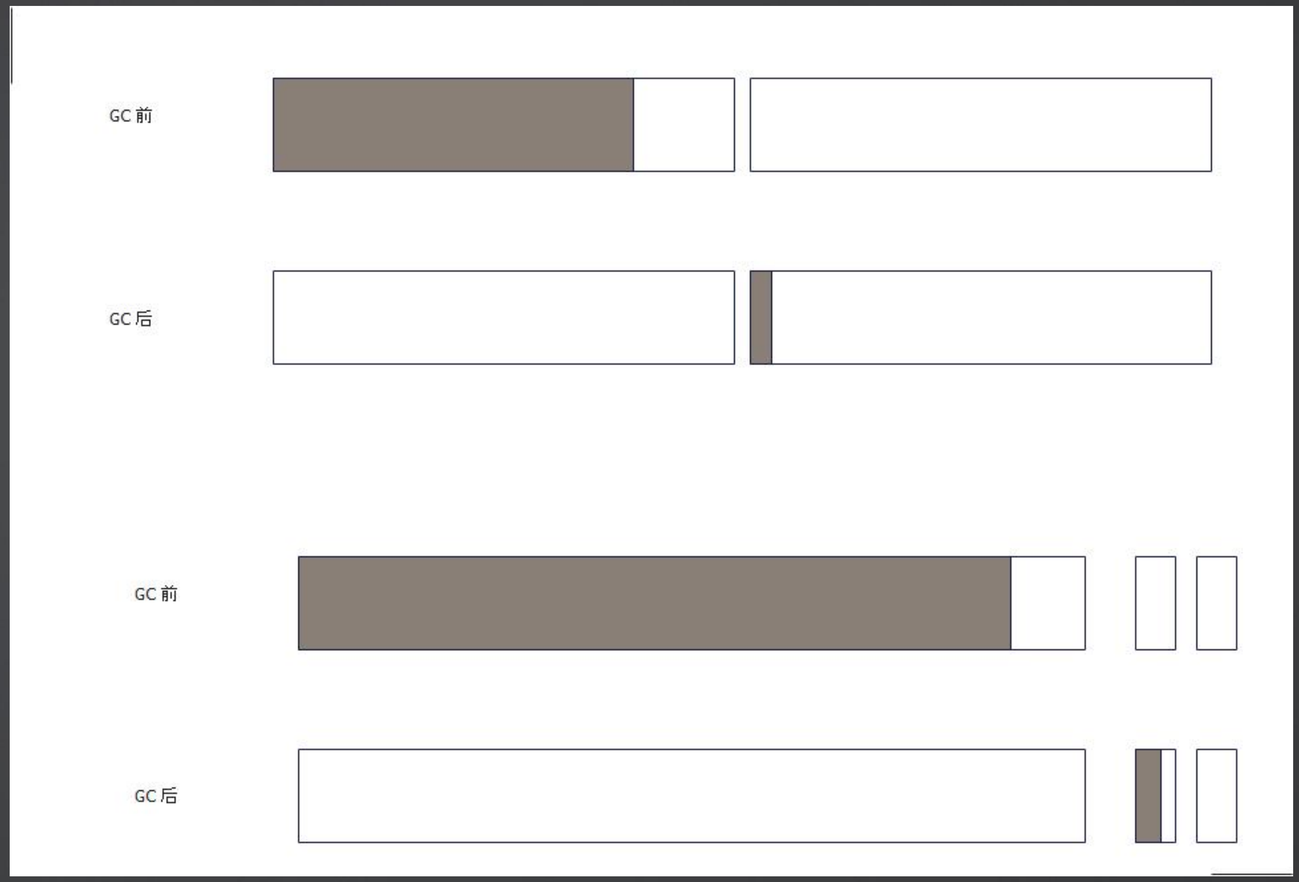
强三色不变性：不允许黑色对象引用白色对象

弱三色不变性：黑色对象可以引用白色对象，白色对象存在其他灰色对象对它的引用，或者可达它的链路上游存在灰色对象



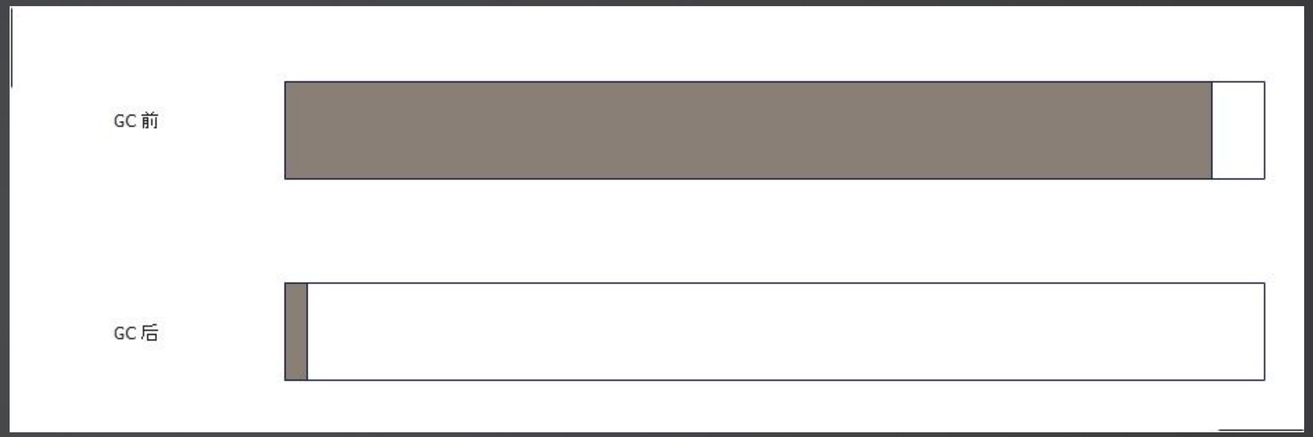
基本算法——标记复制

- 标记-复制：核心是将存活对象复制到另外一块区域。
  - 理论上来说，是将内存平均分成 A，B；每次使用 A，就把存活对象拷贝到B，而后接着用B，把存活对象拷贝到A，交替进行
  - 但是因为存活对象的比例是极小的，所以为了提高利用率，可以拆成A，B，C 三块，其中B 和 C 专门用来存放存活对象。三者的比例根据应用特征来确定，例如 8:1:1
  - 在采用非均分的方式下，要考虑兜底：
    - 要么是退化到整理或者清扫算法
    - 要么是尝试额外弄一块额外的内存
- 标记-整理
- 标记-清扫和标记-清扫-压缩



基本算法——标记整理

- 标记-复制
- 标记-整理：核心是将存活对象压缩到内存的一侧。其实从实现来看，可以认为和复制有点像。但是它要求整理必须要从某一端开始
- 标记-清扫和标记-清扫-压缩：



基本算法——标记清扫

- 标记-复制
- 标记-整理
- 标记-清扫和标记-清扫-压缩：核心是在于，标记存活对象之后，归还空闲内存。
  - 清扫之后会导致可用内存分成一段段，所以会引入压缩过程。
  - 压缩过程类似整理，将存活对象复制到一边
  - 压缩过程开销巨大，所以并不是每一次清扫都压缩，比如说可能十次清扫才会压缩一次
  - 标记-清扫算法要用比较特殊的内存分配方式，因为空闲内存不是连续的，一般采用的都是空闲链表法来记录空闲内存，GO使用的是位图标记法
  - 空闲链表法在正向内存分配的时候会更慢，因为要找可用的连续内存

GC Root 是什么

GC root，顾名思义，是指在 GC 启动的时候必然存活的一组对象。一般来说，GC root 包含：

- 栈上对象，于 Java 来说，还包含本地方法栈；
- 全局对象，如常量池；
- 非收集部分指向收集部分的引用。常见于分代 GC 和增量式 GC 中；

对于栈中的对象，可以进一步划分成局部变量，方法参数...部分情况下，如果寄存器直接存放了值，那么这个值会被保守地解释为指针（它可能根本不是指针，而是一个基本变量），而后尝试进一步解析；

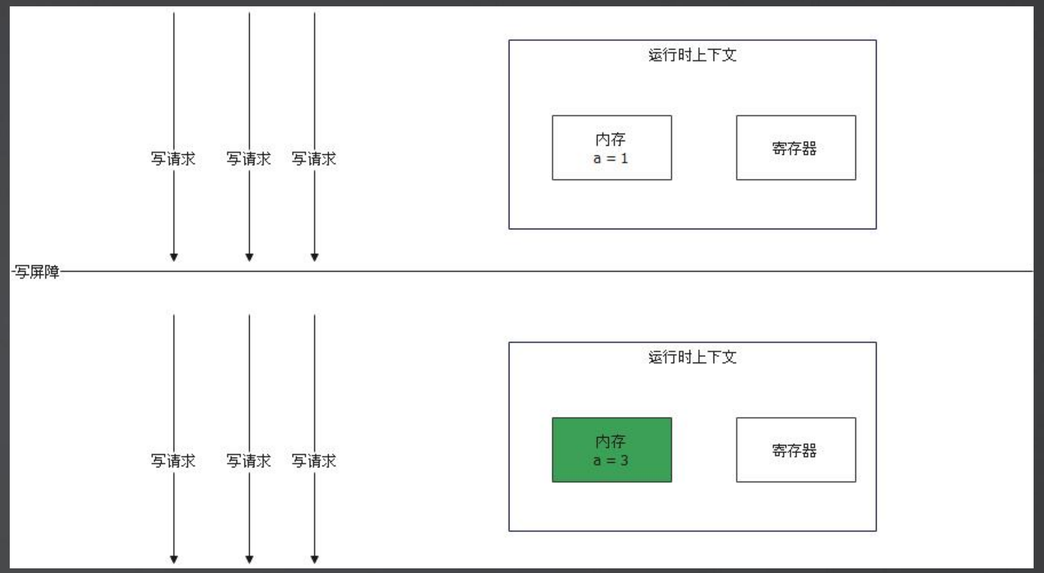
GC 写屏障读屏障

从形态上来说，读写屏障都是一段代码。

- 写屏障：在写的时候触发
- 读屏障：在读的时候触发

读写屏障类似 AOP，钩子，回调，只不过它很底层，是运行时环境帮我们处理。

读写屏障可以读取或者修改当时的执行上下文，从而实现不同的功能



一个非常类似的场景，就是 ORM 框架的回调机制。比如说当你在插入数据或者更新数据的时候，同步更新缓存。其本质是一样，即捕捉到变更的事件，然后修改事件本身或者修改事件的上下文，从而嵌入自己的特殊处理逻辑

GC 基本知识补充

- 分代假说。分代假说在别的语言里面使用频繁，在 Go 里面没有使用。（为什么？个人猜测是实现太多复杂而收益对于 Go 来说不是很大）
  - 绝大部分对象都是朝生夕死的
  - 越老的对象越不可能死亡
- 安全点：并不是任何时刻都可以发起 GC的。发起 GC 要满足一定的条件，这些满足条件的点就被称为安全点。这个概念在 JVM 上比较关注
  - 实际上安全点并不仅仅是用于 GC，可以认为处于安全点的时候，程序可以做一些自我检查。这当然就包括启动 GC
  - 安全点：没有引用变更，没有线程（协程）状态变更。
- 停顿时间和总时间：大多数时候，我们只关注停顿时间；只有在追求吞吐量的环境下，我们才会关注总时间；

Go 内存分配/垃圾回收优化

- 逃逸分析：尽量使对象分配到栈上，优化那些到堆上的内存分配
- 对象池复用

Go 本身不管是垃圾回收，还是内存分配，其实没什么参数可以调优。这和 Java 很不一样，Java 的 JVM 调优就要麻烦很多，要考虑不同的算法，堆的大小，分代大小，触发阈值等。但是 Go 都调不了，所以面试更加简单。

如果你是别的语言也是有垃圾回收的，记得在面试的时候进行横向对比。

面试题

- GO 的垃圾回收过程
- 什么是三色标记法
- 什么是强弱三色不变性
- Go 会发生几次 STW?
- STW 有什么问题?

Q & A

THANKS