## 垃圾回收机制

用户会通过内存分配器在堆上申请内存，而垃圾回收器负责回收堆上的内存空间，共同管理着程序中的堆内存空间。

golang的gc，从最初的标记-清除，到v1.5实现三色标记并发gc。

1.1标记-清除算法，先标记所有需要回收的对象，标记完成后，统一回收掉所有被标记的对象。通过可达性分析，从GC Roots，根据引用关系向下搜索。根节点root主要指的是全局变量、各个G的栈上的变量。缺点：是保证GC期间，标记对象的状态不能变化，需要stop the world。清除会产生大量不连续的内存碎片，当需要分配大对象时，不得不提前触发另一次GC

1.2 三色标记法

白色（尚未被访问，结束仍白，表示不可达），黑色（本身和所有引用都已经被扫描），灰色（引用没有被扫描）。程序和收集器是并发工作的，存活的对象可能被标记死亡，悬挂指针。黑色部分引用了新对象

1.3 写屏障

Go支持并行GC，屏障会记录第一次扫描时每个对象的状态，和第二次作对比，如果引用关系发生了变化就标灰，防止丢失

1.4 辅助GC

防止扫描后回收垃圾的速度跟不上用户程序分配对象的速度，同时运行的用户程序goroutine分配了内存，这个goroutine会被要求辅助GC做一部分工作。标记+回收。

1. gc流程



gc有标记用的和清扫用的两种后台任务。标记用的后台任务会在需要时启动，可以同时启动的任务数量大约是P的数量的25%。

整个GC会两次stop the world，mark阶段的开始和mark termination。

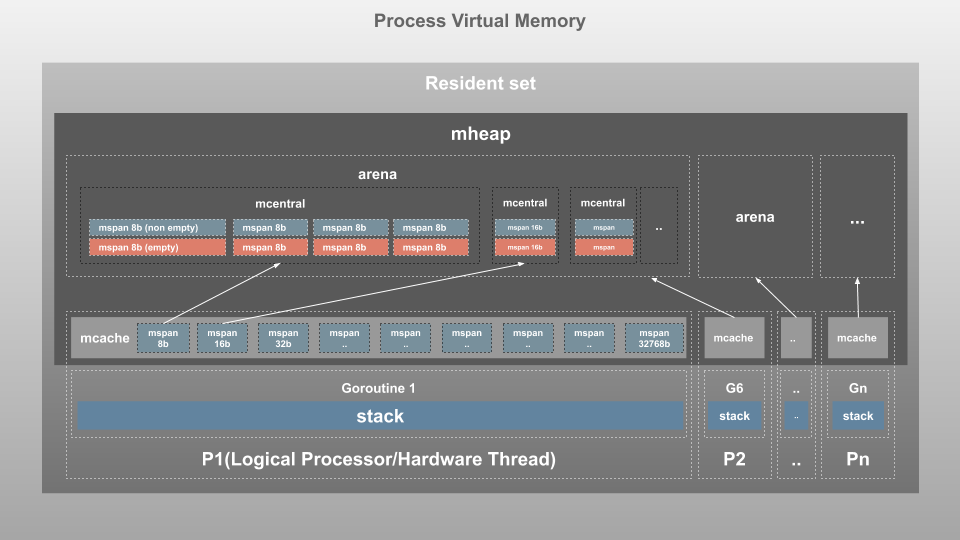
mark包括1、2

1. mark prepare：初始化gc，包括开启写屏障和辅助gc，准备对root对象的扫描统计任务等。需要STW
2. gc drains：扫描所有root对象，包括全局指针和goroutine栈上的指针，将其加入标记队列（灰色队列），并循环处理灰色队列的对象，直到灰色队列为空。后台并行执行
3. mark termination：完成标记工作，重新扫描全局指针和栈，禁用写屏障和辅助gc。因为mark和用户程序是并行的，所以状态会变化，需要re-scan。需要STW。
4. sweep:按照标记回收所有白色的对象，后台并行
5. Sweep Termination: 对未清扫的span进行清扫, 只有上一轮的GC的清扫工作完成才可以开始新一轮的GC。

## Go的内存分配器

Go静态语言，不需要vm。通过go的二进制文件中嵌入了go runtime，实现gc、调度、并发。内存空间由栈区、堆区组成，函参、返回值、局部变量分配到栈上，这部分由编译器管理。go的内存管理=需要内存时自动分配内存，不需要gc。这些工作由runtime完成。

Runtime将goroutines(G)调度到P（逻辑处理器），每台P有M（逻辑机器）。



残余内存为os分配给进程的虚拟内存，Mheap为最大的内存块，进行gc的地方，存动态数据(编译时无法计算出大小的data)

留驻内存被划分为多个8k的页，由全局的mheap对象管理。mheap通过把页归类成不同的结构进行管理，mheap中管理内存页的基本单位为mspan（双向链表），类别从8bytes到32KB



每个span存在两个，一个用于带指针的对象，一个用于无指针的对象。

栈内存管理