## 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

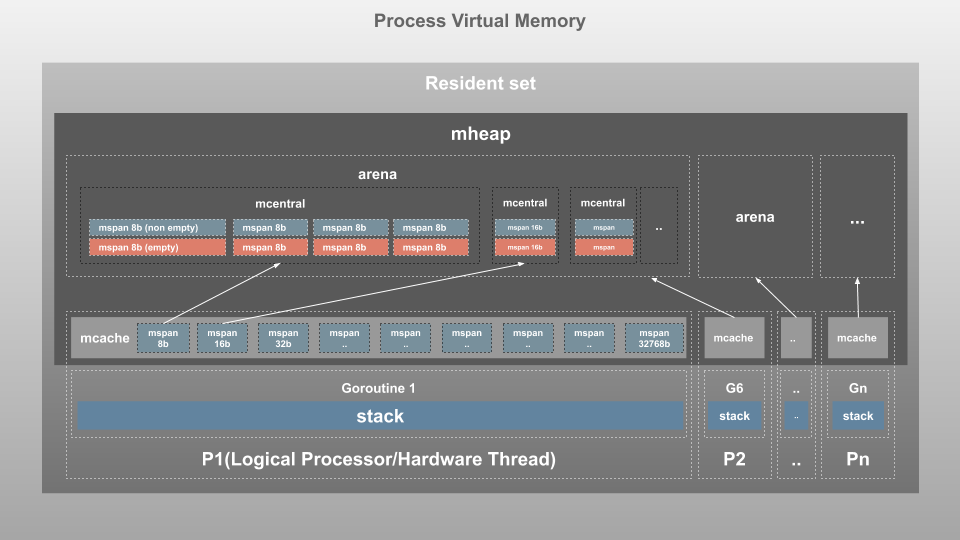
1. mark prepare：初始化gc，包括开启写屏障和辅助gc，准备对root对象的扫描统计任务等。需要STW
2. gc drains：扫描所有root对象，包括全局指针和goroutine栈上的指针，将其加入标记队列（灰色队列），并循环处理灰色队列的对象，直到灰色队列为空。后台并行执行
3. mark termination：完成标记工作，重新扫描全局指针和栈，禁用写屏障和辅助gc。因为mark和用户程序是并行的，所以状态会变化，需要re-scan。需要STW。
4. sweep:按照标记回收所有白色的对象，后台并行
5. Sweep Termination: 对未清扫的span进行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

**go的内部内存结构**

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



**内存管理组件**

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。

相同大小级别的span可以归为一类，用mcentral管理。每个mcentral包含两个mspanList：empty：双向span链表，包括没有空闲对象的span或缓存的span。当此处的span被释放时，它将被移动到non-empty span。

non-empty：有空闲对象的span双向链表。当从mentral请求新的span，mcentral会从该链表中获取span并把它移动到empty span链表中。

如果mcentral没有可用的span，那么就向mheap请求新页。在go源码实现中，mheap结构体由两组非常重要的字段，长度为134的mcentral数组（67需要scan的mcentral+67noscan的mcentral）。另一组就是管理堆区内存区域的arenas及相关字段。

arena：堆在已分配的虚拟内存中，根据需要增长和缩小。当需要更多的内存时，mheap从虚拟内存中，以每块64MB（64bit机器）的单位获取新内存，这块内存就叫做arena。

mcache是提供给P的告诉缓存，用于存储小单位（<=32KB）。用于存储动态数据。所有类大小的mcache包含scan和noscan类型的mspan。这些大对象的申请请求是以获取central lock为代价的，在任何给定的时间点只能满足一个P的请求。mcache需要的时候会从mcentral需要时请求新的span。

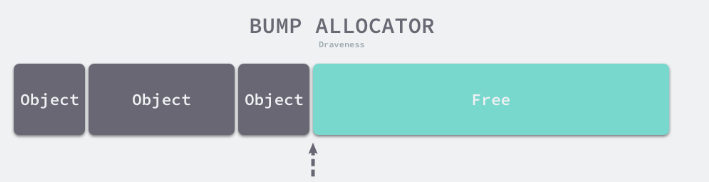
**设计原理**

每次用户程序申请内存时，通过内存分配器申请新内存，分配器会负责从堆中初始化响应的内存区域

**分配方法**

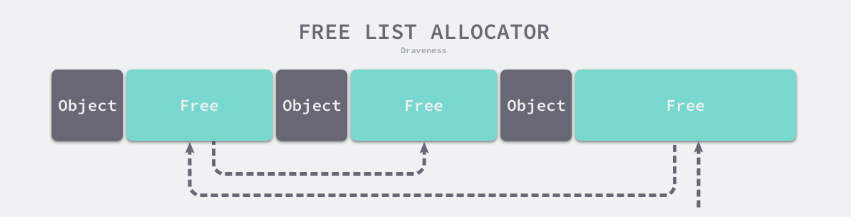
线性分配器、空闲链表分配器

线性分配器（原理：在内存中维护一个指向内存特定位置的指针，用户每次申请内存，分配器只需要检查剩余的空闲内存，然后返回分配的内存区域并修改指针在内存的位置）



被释放无法被重用。比较适合包括拷贝的垃圾回收算法：像标记-复制、标记-整理等算法，通过copy的方法整理对象的位置，将空闲内存合并，这样就可以利用线性分配器的效率提升内存分配器的性能。

空闲链表分配器：



每次用户申请内存的时候，空闲链表分配器会依次遍历空闲的内存块，找到足够大的内存，申请资源修改链表。

首次适应：从链表头开始遍历，选择大于申请内存大小的第一个内存块

循环首次适应：从上次遍历结束的位置开始遍历

最优遍历：从头，找最合适的

隔离适应：内存分割成多个链表，每个链表的内存块大小相同，申请内存时先找到满足条件的链表，再从链表中选择合适的内存块。

栈内存管理