一文搞懂JAVA与GO垃圾回收

hewitt、 极客重生 2021-07-30 17:00

收录于合集
#深入理解编程语言

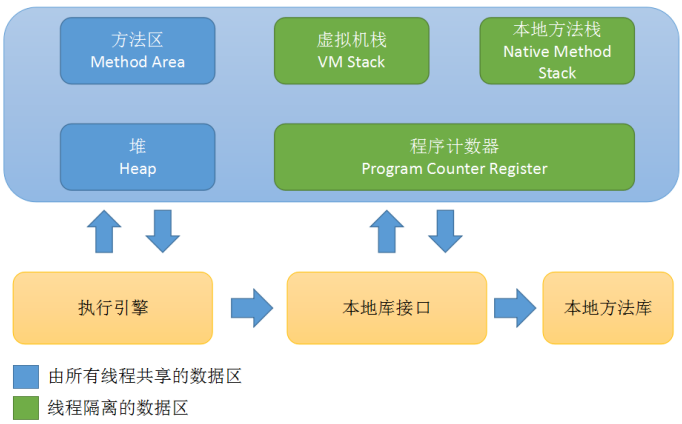
21个

> 导语 现代高级编程语言管理内存的方式分为两种：自动和手动。像 C、C++ 等编程语言使用手动管理内存的方式，编写代码过程中需要主动申请或者释放内存；而 PHP、Java 和 Go等语言使用自动的内存管理系统，由内存分配器和垃圾收集器来代为分配和回收内存，其中垃圾收集器就是我们常说的GC。本文中，笔者将从原理出发，介绍Java和Golang垃圾回收算法，并从原理上对他们做一个对比。本文系KM热门文章，hewitt授权，发表在本公众号上，非常好的文章，分享给大家，值得慢慢细读，记得三连支持，感谢。

Java垃圾回收

垃圾回收区域及划分

在介绍Java垃圾回收之前，我们需要了解Java的垃圾主要存在于哪个区域。JVM内存运行时区域划分如下图所示：

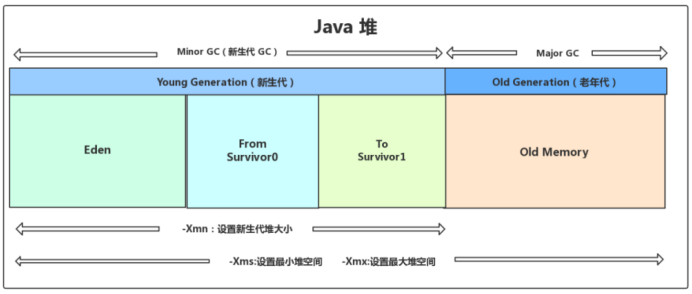


- **程序计数器**：是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器，各条线程之间计数器互不影响，独立存储

- **虚拟机栈**：它描述的是 Java 方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame，是方法运行时的基础数据结构）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。
- **本地方法栈**：它与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。
- **Java 堆**：它是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
- **方法区**：它与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

Java内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈3个区域随着线程而生，随着线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈的操作，每个栈帧中分配多少内存基本是在类结构确定下来时就已知的。而Java堆和方法区则不同，一个接口中的多个实现类需要的内存可能不同，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，而在java8中，方法区存放于元空间中，元空间与堆共享物理内存，因此，**Java堆和方法区是垃圾收集器管理的主要区域**。

从垃圾回收的角度，由于JVM垃圾收集器基本都采用分代垃圾收集理论，所以 Java 堆还可以细分为如下几个区域（以HotSpot虚拟机默认情况为例）：



其中，Eden 区、From Survivor0(“From”) 区、To Survivor1(“To”) 区都属于新生代，Old Memory 区属于老年代。

大部分情况，对象都会首先在 Eden 区域分配；在一次新生代垃圾回收后，如果对象还存活，则会进入 To 区，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（超过了 survivor 区的一半时，取这个值和 `MaxTenuringThreshold` 中更小的一个值，作为新的晋升年龄阈值），就会晋升到老年代中。经过这次 GC 后，Eden 区和 From区已经被清空。这个时候，From和To会交换他们的角色，保证名为 To 的 Survivor 区域是空的。Minor GC 会一直重复这样的过程。在这个过程中，有可能当次Minor GC后，Survivor 的"From"区域空间不够用，有一些还达不到进入老年代条件的实例放不下，则放不下的部分会提前进入老年代。

针对 HotSpot VM 的实现，它里面的 GC 其实准确分类只有两大种：

- **部分收集 (Partial GC)**：
  - 新生代收集（Minor GC / Young GC）：只对新生代进行垃圾收集；
  - 老年代收集（Major GC / Old GC）：只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；
  - 混合收集（Mixed GC）：对整个新生代和部分老年代进行垃圾收集。
- **整堆收集 (Full GC)**：收集整个 Java 堆和方法区。

Java堆内存常见分配策略

- 对象优先在 eden 区分配。大部分对象朝生夕灭
- 大对象直接进入老年代。大对象就是需要大量连续内存空间的对象（比如：字符串、数组），容易导致内存还有不少空间就提前触发垃圾收集获取足够的连续空间来安置它们。为了避免为大对象分配内存时，由于分配担保机制带来的复制而降低效率，建议大对象直接进入空间较大的老年代。
- 长期存活的对象将进入老年代，动态对象年龄判定：在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（超过了 survivor 区的一半时，取这个值和 `MaxTenuringThreshold` 中更小的一个值，作为新的晋升年龄阈值），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。
- 空间分配担保。在发生Minor GC之前，虚拟机会先检查老年代最大可用连续内存空间是否大于新生代所有对象总空间。如果这个条件成立,那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许【担保失败】
  - 如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小

- 如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的
- 如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC

判断对象死亡

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断哪些对象已经死亡（即不能再被任何途径使用的对象）。判断一个对象是否存活有引用计数、可达性分析这两种算法，两种算法各有优缺点。Java和Go都使用可达性分析算法，一些动态脚本语言（如:ActionScript）一般使用引用计数算法。

引用计数法

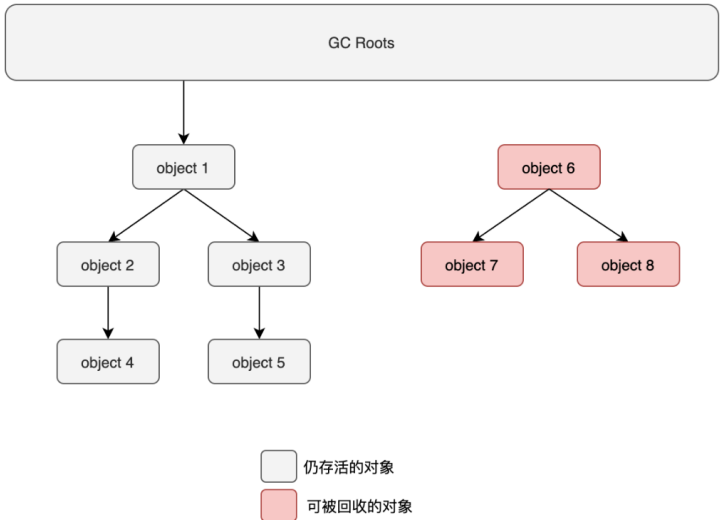
给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。 所谓对象之间的相互引用问题，如下面代码所示：除了对象 objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为 0，于是引用计数算法无法通知 GC 回收器回收他们。

```
public class ReferenceCountingGc {
  Object instance = null;
  public static void main(String[] args) {
     ReferenceCountingGc objA = new ReferenceCountingGc();
     ReferenceCountingGc objB = new ReferenceCountingGc();
     objA.instance = objB;
     objB.instance = objA;
     objA = null;
     objB = null;

  }
}
```

可达性分析算法

这个算法的基本思想就是通过一系列的称为 **“GC Roots”** 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。算法优点是能准确标识所有的无用对象，包括相互循环引用的对象；缺点是算法的实现相比引用计数法复杂。



不可达的对象并非“非死不可”

即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。

判断一个运行时常量池中的常量是废弃常量

> 1. JDK1.7 之前运行时常量池逻辑包含字符串常量池存放在方法区, 此时 hotspot 虚拟机对方法区的实现为永久代
> 2. JDK1.7 字符串常量池被从方法区拿到了堆中, 这里没有提到运行时常量池,也就是说字符串常量池被单独拿到堆,运行时常量池剩下的东西还在方法区, 也就是 hotspot 中的永久代 。
> 3. JDK1.8 hotspot 移除了永久代用元空间(Metaspace)取而代之, 这时候字符串常量池还在堆, 运行时常量池还在方法区, 只不过方法区的实现从永久代变成了元空间(Metaspace)

假如在字符串常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池了。

如何判断一个方法区的类是无用的类

类需要同时满足下面 3 个条件才能算是 “无用的类”，虚拟机可以对无用类进行回收。

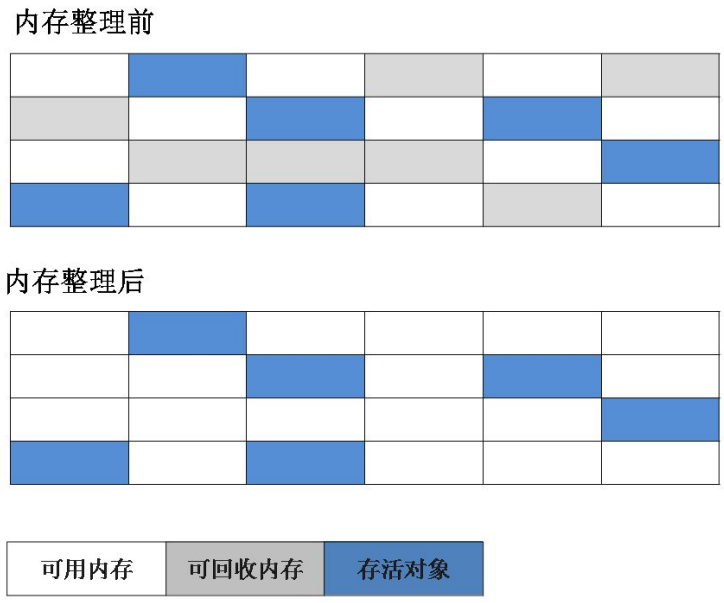
- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

垃圾收集算法

当确定了哪些对象可以回收后，就要需要考虑如何对这些对象进行回收，目前垃圾回收算法主要有以下几种。

标记清除算法

该算法分为“标记”和“清除”阶段：首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。



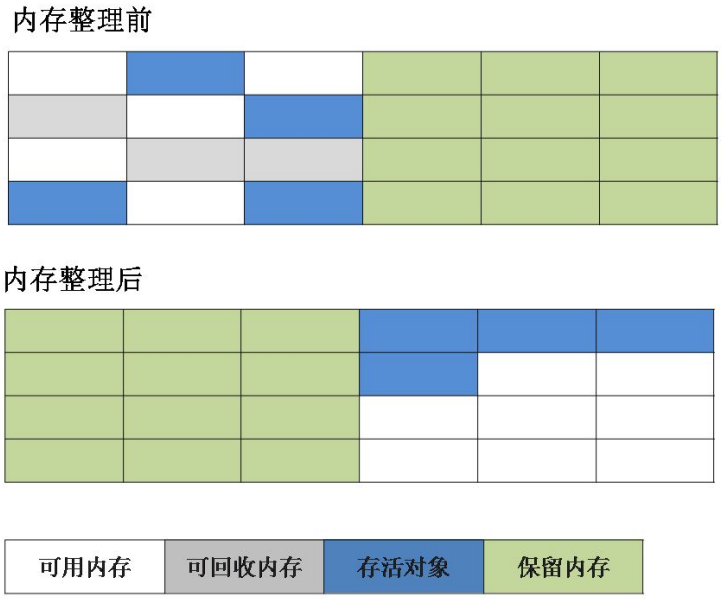
适用场合：存活对象较多的情况、适用于年老代（即旧生代）

缺点：

- **空间问题**，容易产生内存碎片，再来一个比较大的对象时（典型情况：该对象的大小大于空闲表中的每一块儿大小但是小于其中两块儿的和），会提前触发垃圾回收
- **效率问题**，扫描了整个空间两次（第一次：标记存活对象；第二次：清除没有标记的对象）

标记复制算法

为了解决效率问题，“标记-复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。使用复制算法，回收过程中就不会出现内存碎片，也提高了内存分配和释放的效率

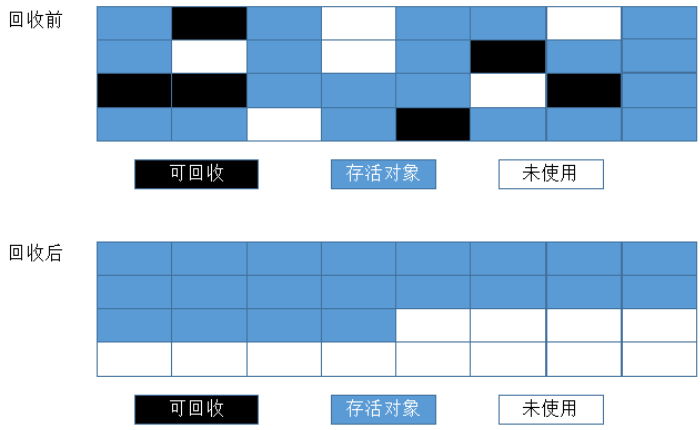


适用场合：存活对象较少的情况下比较高效、用于年轻代（即新生代）

缺点：需要一块儿空的内存空间，整理阶段，由于移动了可用对象，需要去更新引用。

标记整理算法

对于对象存活率较高的场景，复制算法要进行较多复制操作，使得效率会变低，这种场景更适合标记-整理算法，与标记-清理一样，标记整理算法先标记出对象的存活状态，但在清理时，是先把所有存活对象往一端移动，然后直接清掉边界以外的内存。



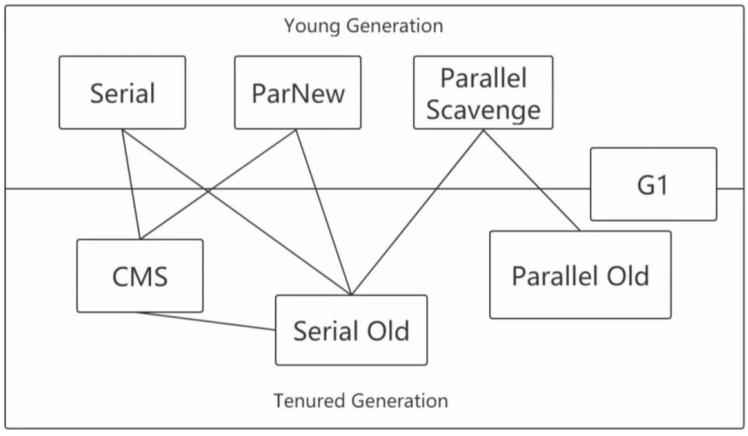
适用场合：对象存活率较高（即老年代）

缺点：整理阶段，由于移动了可用对象，需要去更新引用。

分代收集算法

当前Java虚拟机的垃圾收集都采用分代收集算法，根据对象存活周期的不同将内存分为几块。比如在新生代中，每次收集都会有大量对象死去，所以可以选择”标记-复制“算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们选择“标记-清除”或“标记-整理”算法进行垃圾收集。

垃圾收集器



垃圾收集器	特点	算法	适用场景

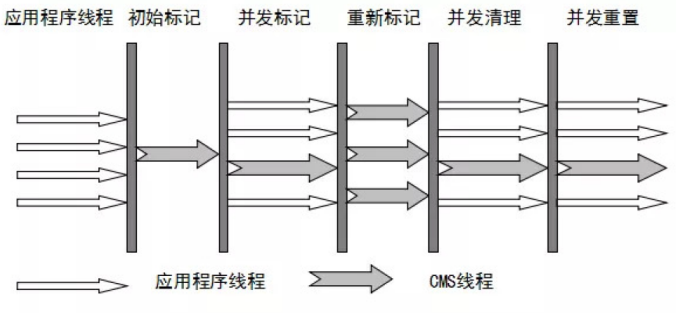
垃圾收集器	特点	算法	适用场景
Serial	最基本、历史最悠久的单线程垃圾收集器。	新生代采用标记-复制算法，老年代采用标记-整理算法。	运行在 Client 模式下的虚拟机
ParNew	Serial 收集器的多线程版本	新生代采用标记-复制算法，老年代采用标记-整理算法	运行在 Server 模式下的虚拟机
Parallel Scavenge	使用标记-复制算法的多线程收集器，关注吞吐量	新生代采用标记-复制算法，老年代采用标记-整理算法。	JDK1.8 默认收集器在注重吞吐量及CPU资源的场合

Serial Old	Serial 收集器的老年代版本	标记-整理算法	在 JDK<1.5与 Parallel Scavenge 收集器搭配使用作为CMS收集器的后备方案
Parallel Old	Parallel Scavenge 收集器的老年代	标记-整理算法	在注重吞吐量及CPU资源的场合
CMS	多线程的垃圾收集器（*用户线程和垃圾回收线程可以同时进行*）	标记-清除算法	希望系统停顿时间最短，注重服务的响应速度的场景
G1	一款面向服务器的垃圾收集器，并行并发，空间整合，可预测的停顿时间	标记-复制算法	服务端应用、针对具有大内存多处理器的机器

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为直到现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，我们能做的就是根据具体应用场景选择适合自己的垃圾收集器。接下来我们将重点介绍CMS垃圾处理器和G1处理器。CMS处理器是和Golang中垃圾回收机制比较类似的一个垃圾处理器，而G1是Java8以来使用最多的垃圾处理器。

CMS 收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的并发收集器，也是老年代垃圾收集器，第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。CMS收集器在Minor GC时会暂停所有的应用线程，并以多线程的方式进行垃圾回收。在Full GC时不再暂停应用线程，而是使用若干个后台线程定期的对老年代空间进行扫描，及时回收其中不再使用的对象。



CMS 收集器是一种 “标记-清除”算法实现的，它的运作过程分为7个步骤：

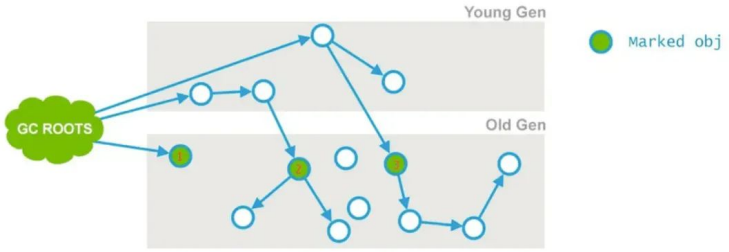
1. 初始标记(CMS-initial-mark) ,会导致stw;
2. 并发标记(CMS-concurrent-mark)，与用户线程同时运行；
3. 预清理（CMS-concurrent-preclean），与用户线程同时运行；
4. 可被终止的预清理（CMS-concurrent-abortable-preclean） 与用户线程同时运行；
5. 重新标记(CMS-remark) ，会导致swt；
6. 并发清除(CMS-concurrent-sweep)，与用户线程同时运行；
7. 并发重置状态等待下次CMS的触发(CMS-concurrent-reset)，与用户线程同时运行

初始标记（STW）

该阶段单线程执行，主要分分为两步：

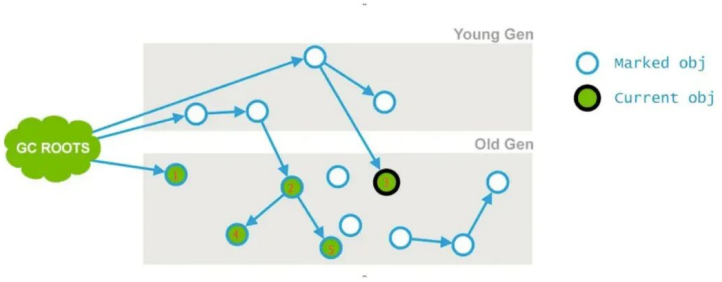
1. 标记GC Roots可达的老年代对象；
2. 遍历新生代对象，标记可达的老年代对象；

该过程结束后，对象分布如下：



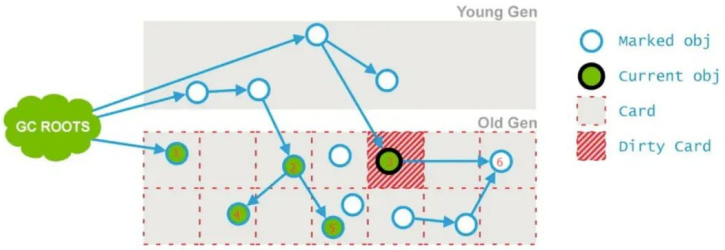
并发标记

该阶段GC线程和应用线程并发执行，遍历初始标记阶段标记出来的存活对象，然后继续递归标记这些对象可达的对象。使用三色可达性分析算法进行标记，因为该阶段并发执行的，在运行期间可能发生新生代的对象晋升到老年代、或者是直接在老年代分配对象、或者更新老年代对象的引用关系等等，对于这些对象，都是需要进行重新标记的，否则有些对象就会被遗漏，发生漏标的情况。为了提高重新标记的效率，该阶段会使用三色可达性分析中的增量更新解决这一问题：把上述对象所在的Card标识为Dirty，后续只需扫描这些Dirty Card的对象，避免扫描整个老年代。

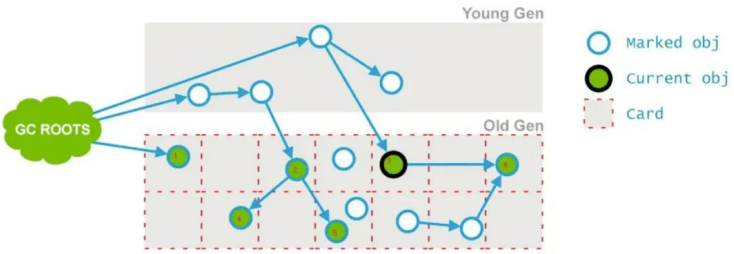


预清理阶段

前一个阶段已经说明，不能标记出老年代全部的存活对象，是因为标记的同时应用程序会改变一些对象引用，这个阶段就是用来处理前一个阶段因为引用关系改变导致没有标记到的存活对象的，它会扫描所有标记为Dirty的Card 如下图所示，在并发清理阶段，节点3的引用指向了6；则会把节点3的card标记为Dirty；



最后将6标记为存活,如下图所示：



可终止的预处理

这个阶段的目标跟“预清理”阶段相同，也是为了减轻重新标记阶段的工作量，。可中断预清理的价值：在进入重新标记阶段之前尽量等到一个Minor GC，尽量缩短重新标记阶段的停顿时间。另外可中断预清理会在Eden达到50%的时候开始，这时候离下一次minor gc还有半程的时间，这个还有另一个意义，即避免短时间内连着的两个停顿。

在该阶段，主要循环的做两件事：

1. 处理 From 和 To 区的对象，标记可达的老年代对象
2. 和上一个阶段一样，扫描处理Dirty Card中的对象

当然了，这个逻辑不会一直循环下去，打断这个循环的条件有三个：

1. 可以设置最多循环的次数 `CMSMaxAbortablePrecleanLoops`，默认是0，意思没有循环次数的限制。
2. 如果执行这个逻辑的时间达到了阈值`CMSMaxAbortablePrecleanTime`，默认是5s，会退出循环。
3. 如果新生代Eden区的内存使用率达到了阈值`CMSScheduleRemarkEdenPenetration`，默认50%，会退出循环。（这个条件能够成立的前提是，在进行Precleaning时，Eden区的使用率小于十分之一）

重新标记（STW）

这个阶段会导致第二次stop the word，该阶段的任务是完成标记整个年老代的所有的存活对象。重新扫描堆中的对象，进行可达性分析,标记活着的对象。这个阶段扫描的目标是：新生代的对象 + Gc Roots + 前面被标记为dirty的card对应的老年代对象。如果预清理的工作没做好，这一步扫描新生代的时候就会花很多时间，导致这个阶段的停顿时间过长。这个过程是多线程的。

为什么要扫描新生代呢，因为对于老年代中的对象，如果被新生代中的对象引用，那么就会被视为存活对象，即使新生代的对象已经不可达了，也会使用这些不可达的对象当做cms的“gc root”，来扫描老年代；因此对于老年代来说，引用了老年代中对象的新生代的对象，也会被老年代视作“GC ROOTS”:当此阶段耗时较长的时候，可以加入参数-XX:+CMSScavengeBeforeRemark，在重新标记之前，先执行一次ygc，回收掉年轻带的对象无用的对象，并将对象放入幸存带或晋升到老年代，这样再进行年轻带扫描时，只需要扫描幸存区的对象即可，一般幸存带非常小，这大大减少了扫描时间。

并发清理

通过以上5个阶段的标记，老年代所有存活的对象已经被标记并且现在要通过Garbage Collector采用清扫的方式回收那些不能用的对象了。这个阶段主要是清除那些没有标记的对象并且回收空间；由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

G1 收集器

G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.

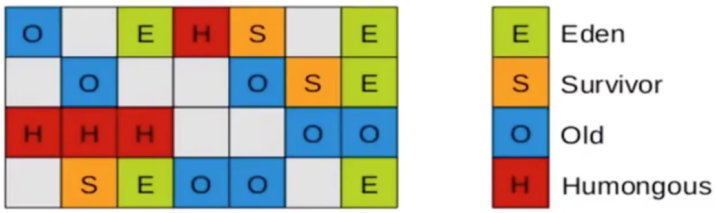
G1垃圾收集器相对比其他收集器而言，最大的区别在于它取消了年轻代、老年代的物理划分，取而代之的是将堆划分为若干个区域（Region），这些区域中包含了有逻辑上的年轻代、老年代区域。这样做的好处就是，我们再也不用单独的空间对每个代进行设置了，不用担心每个代内存是否足够。

它具备一下特点：

- 并行与并发：G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。
- 分代收集：虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
- 空间整合：与 CMS 的“标记-清理”算法不同，G1 从整体来看是基于“标记-整理”算法实现的收集器；从局部上来看是基于“标记-复制”算法实现的。

- 可预测的停顿：这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内。

G1算法将堆划分为若干个区域（Region），它仍然属于分代收集器。不过，这些区域的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有cms内存碎片问题的存在了。



对象分配策略，它分为3个阶段：

- TLAB(Thread Local Allocation Buffer)线程本地分配缓冲区
- Eden区中分配
- Humongous区分配

TLAB为线程本地分配缓冲区，它的目的为了使对象尽可能快的分配出来。如果对象在一个共享的空间中分配，我们需要采用一些同步机制来管理这些空间内的空闲空间指针。在Eden空间中，每一个线程都有一个固定的分区用于分配对象，即一个TLAB。分配对象时，线程之间不再需要进行任何的同步。对TLAB空间中无法分配的对象，JVM会尝试在Eden空间中进行分配。如果Eden空间无法容纳该对象，就只能在老年代中进行分配空间。

在G1中，还有一种特殊的区域，叫Humongous区域。如果一个对象占用的空间超过了分区容量50%以上，G1收集器就认为这是一个巨型对象。这些巨型对象，默认直接会被分配在年老代，但是如果它是一个短期存在的巨型对象，就会对垃圾收集器造成负面影响。为了解决这个问题，G1划分了一个Humongous区，它用来专门存放巨型对象。如果一个H区装不下一个巨型对象，那么G1会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC。

G1 收集器的运作大致分为以下几个步骤：

- **初始标记**
- **并发标记**
- **最终标记**
- **筛选回收**

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来) 。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

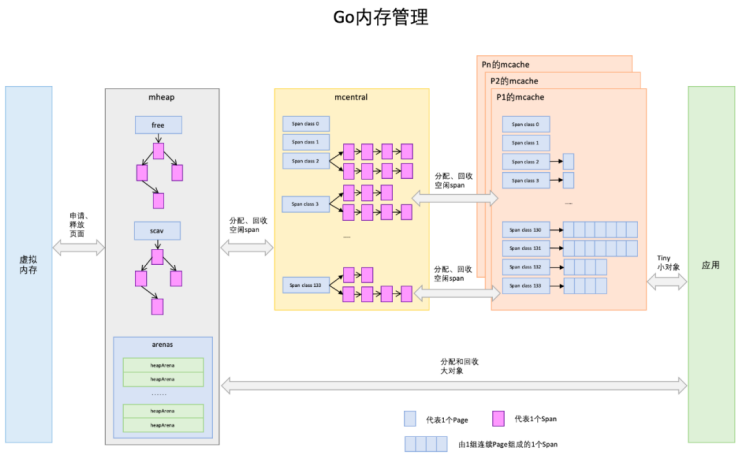
关于G1详细原理，可以参考G1收集器的收集原理中的介绍。

Golang垃圾回收

从Go v1.12版本开始，Go使用了**非分代的、并发的、基于三色标记和清除的垃圾回收器**。和C/C++一样，Go是一种静态类型的编译型语言。因此，Go不需要VM，Go应用程序二进制文件中嵌入了一个小型运行时(Go runtime)，可以处理诸如垃圾收集(GC)，调度和并发之类的语言功能。首先让我们看一下Go内部的内存管理是什么样子的

Golang内存管理

> 这里先简单介绍一下 Golang 运行调度。在 Golang 里面有三个基本的概念：G, M, P。
>
> - G: Goroutine 执行的上下文环境。
> - M: 操作系统线程。
> - P: Processer。进程调度的关键，调度器，也可以认为约等于 CPU。
>
> 一个 Goroutine 的运行需要 G + P + M 三部分结合起来。



TCMalloc

Go将内存划分和分组为页（Page），这和Java的内存结构完全不同，没有分代内存，这样的原因是Go的内存分配器采用了TCMalloc的设计思想：

Page

与TCMalloc中的Page相同，x64下1个Page的大小是8KB。上图的最下方，1个浅蓝色的长方形代表1个Page。

Span

与TCMalloc中的Span相同，Span是内存管理的基本单位，代码中为mspan，一组连续的Page组成1个Span，所以上图一组连续的浅蓝色长方形代表的是一组Page组成的1个Span，另外，1个淡紫色长方形为1个Span。

mcache

mcache是提供给P（逻辑处理器）的高速缓存，用于存储小对象（对象大小<= 32Kb）。尽管这类似于线程堆栈，但它是堆的一部分，用于动态数据。所有类大小的mcache包含scan和noscan类型mspan。Goroutine可以从mcache没有任何锁的情况下获取内存，因为一次P只能有一个锁G。因此，这更有效。mcache从mcentral需要时请求新的span。

mcentral

mcentral与TCMalloc中的CentralCache类似，是所有线程共享的缓存，需要加锁访问，它按Span class对Span分类，串联成链表，当mcache的某个级别Span的内存被分配光时，它会向mcentral申请1个当前级别的Span。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mcache中的span。当此处的span被释放时，它将被移至non-empty span链表。
- non-empty：有空闲对象的span双向链表。当从mcentral请求新的span，mcentral将从该链表中获取span并将其移入empty span链表。

mheap

mheap与TCMalloc中的PageHeap类似，它是堆内存的抽象，把从OS申请出的内存页组织成Span，并保存起来。当mcentral的Span不够用时会向mheap申请，mheap的Span不够用时会向OS申请，向OS的内存申请是按页来的，然后把申请来的内存页生成Span组织起来，同样也是需要加锁访问的。

栈

这是栈存储区，每个Goroutine（G）有一个栈。在这里存储了静态数据，包括函数栈帧，静态结构，原生类型值和指向动态结构的指针。这与分配给每个P的mcache不是一回事。

内存分配

Go中的内存分类并不像TCMalloc那样分成小、中、大对象，但是它的小对象里又细分了一个Tiny对象，Tiny对象指大小在1Byte到16Byte之间并且不包含指针的对象。小对象和大对象只用大小划定，无其他区分。

核心思想：把内存分为多级管理，降低锁的粒度(只是去mcentral和mheap会申请锁), 以及多种对象大小类型，减少分配产生的内存碎片。

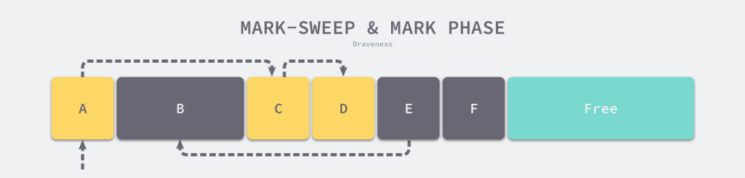
- **微小对象(Tiny)（size <16B）**：使用mcache的微小分配器分配小于16个字节的对象，并且在单个16字节块上可完成多个微小分配。
- **小对象（尺寸16B～32KB）**：大小在16个字节和32k字节之间的对象被分配在G运行所在的P的mcache的对应的mspan size class上。
- **大对象（大小> 32KB）**：大于32 KB的对象直接分配在mheap的相应大小类上(size class)。如果mheap为空或没有足够大的页面满足分配请求，则它将从操作系统中分配一组新的页（至少1MB）
- 如果对应的大小规格在 mcache 中没有可用的块，则向 mcentral 申请
- 如果 mcentral 中没有可用的块，则向 mheap 申请，并根据 BestFit 算法找到最合适的 mspan。如果申请到的 mspan 超出申请大小，将会根据需求进行切分，以返回用户所需的页数。剩余的页构成一个新的 mspan 放回 mheap 的空闲列表。
- 如果 mheap 中没有可用 span，则向操作系统申请一系列新的页（最小 1MB）。Go 会在操作系统分配超大的页（称作 arena）。分配一大批页会减少和操作系统通信的成本。

标记清除算法

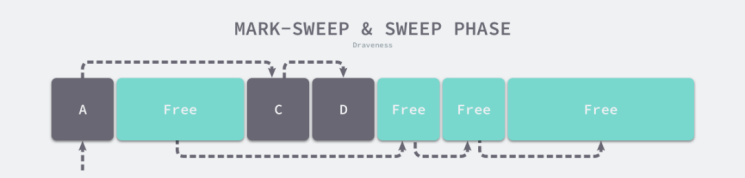
标记清除（Mark-Sweep）算法是最常见的垃圾收集算法，标记清除收集器是跟踪式垃圾收集器，其执行过程可以分成标记（Mark）和清除（Sweep）两个阶段：

1. 标记阶段 — 从根对象出发查找并标记堆中所有存活的对象；
2. 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；

如下图所示，内存空间中包含多个对象，我们从根对象出发依次遍历对象的子对象并将从根节点可达的对象都标记成存活状态，即 A、C 和 D 三个对象，剩余的 B、E 和 F 三个对象因为从根节点不可达，所以会被当做垃圾：



标记阶段结束后会进入清除阶段，在该阶段中收集器会依次遍历堆中的所有对象，释放其中没有被标记的 B、E 和 F 三个对象并将新的空闲内存空间以链表的结构串联起来，方便内存分配器的使用。



这里介绍的是最传统的标记清除算法，垃圾收集器从垃圾收集的根对象出发，递归遍历这些对象指向的子对象并将所有可达的对象标记成存活；标记阶段结束后，垃圾收集器会依次遍历堆中的对象并清除其中的垃圾，整个过程需要标记对象的存活状态，用户程序在垃圾收集的过程中也不能执行，我们需要用到更复杂的机制来解决 STW 的问题。

三色可达性分析

为了解决原始标记清除算法带来的长时间 STW，多数现代的追踪式垃圾收集器都会实现三色可达性分析标记算法的变种以缩短 STW 的时间。三色可达性分析标记算法按“是否被访问过”将程序中的对象分成白色、黑色和灰色三类：

- 白色对象 — 潜在的垃圾，其内存可能会被垃圾收集器回收；

  对象尚未被垃圾收集器访问过，在可达性分析刚开始的阶段，所有的对象都是白色的，若在分析结束阶段，仍然是白色的对象，即代表不可达。

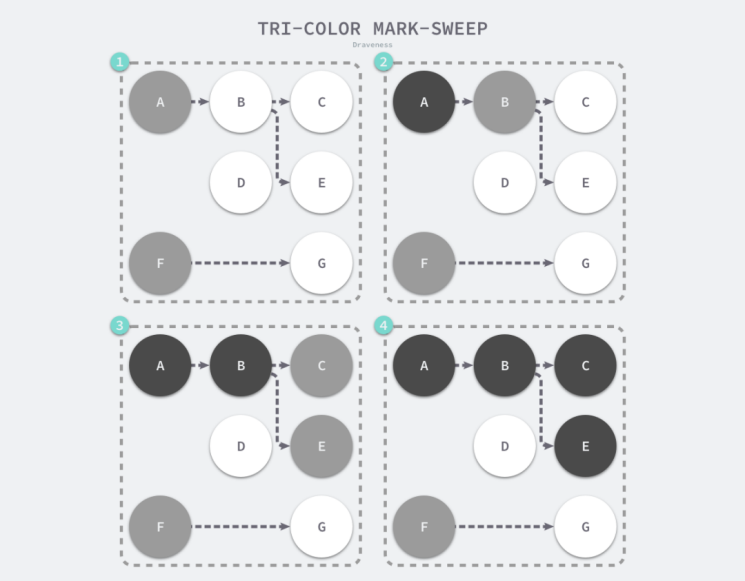
- 黑色对象 — 活跃的对象，包括不存在任何引用外部指针的对象以及从根对象可达的对象；

  表示对象已经被垃圾收集器访问过，且这个对象的所有引用都已经被扫描过，黑色的对象代表已经被扫描过而且是安全存活的，如果有其他对象只想黑色对象无需再扫描一遍，黑色对象不可能直接（不经过灰色对象）指向某个白色对象。

- 灰色对象 — 活跃的对象，因为存在指向白色对象的外部指针，垃圾收集器会扫描这些对象的子对象；

  表示对象已经被垃圾收集器访问过，但是这个对象上至少存在一个引用还没有被扫描过。

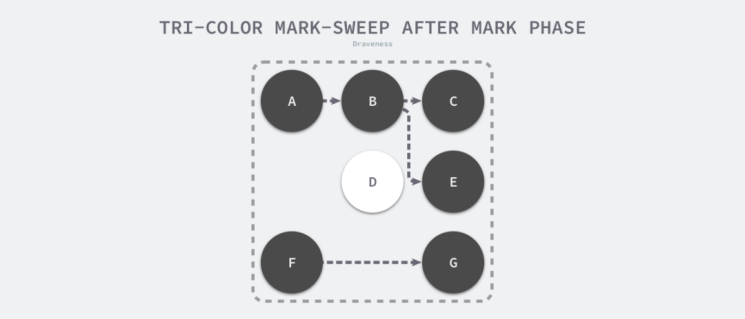
在垃圾收集器开始工作时，程序中不存在任何的黑色对象，垃圾收集的根对象会被标记成灰色，垃圾收集器只会从灰色对象集合中取出对象开始扫描，当灰色集合中不存在任何对象时，标记阶段就会结束。



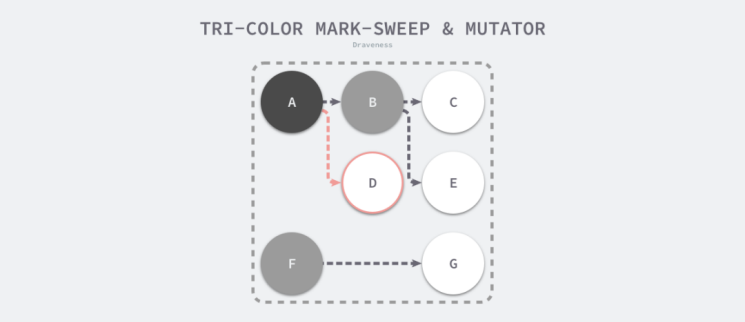
三色标记垃圾收集器的工作原理很简单，我们可以将其归纳成以下几个步骤：

1. 从灰色对象的集合中选择一个灰色对象并将其标记成黑色；
2. 将黑色对象指向的所有对象都标记成灰色，保证该对象和被该对象引用的对象都不会被回收；
3. 重复上述两个步骤直到对象图中不存在灰色对象；

当三色的标记清除的标记阶段结束之后，应用程序的堆中就不存在任何的灰色对象，我们只能看到黑色的存活对象以及白色的垃圾对象，垃圾收集器可以回收这些白色的垃圾，下面是使用三色标记垃圾收集器执行标记后的堆内存，堆中只有对象 D 为待回收的垃圾：



因为用户程序可能在标记执行的过程中修改对象的指针，所以三色标记清除算法本身是不可以并发或者增量执行的，它仍然需要 STW，在如下所示的三色标记过程中，用户程序建立了从 A 对象到 D 对象的引用，但是因为程序中已经不存在灰色对象了，所以 D 对象会被垃圾收集器错误地回收。本来不应该被回收的对象却被回收了，这在内存管理中是非常严重的错误，我们将这种错误称为**悬挂指针**，即指针没有指向特定类型的合法对象，影响了内存的安全性，想要并发或者增量地标记对象还是需要使用屏障技术。

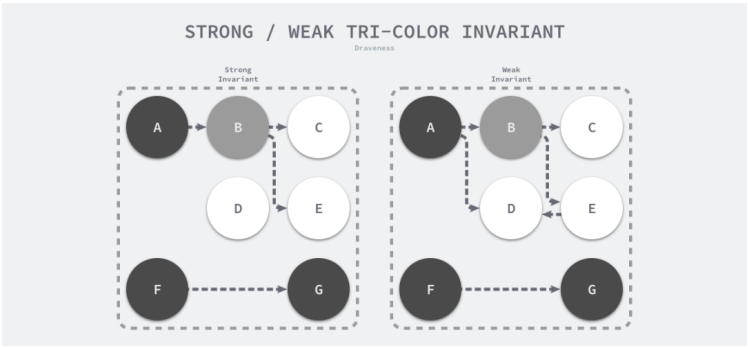


屏障技术

内存屏障技术是一种屏障指令，它可以让 CPU 或者编译器在执行内存相关操作时遵循特定的约束，目前多数的现代处理器都会乱序执行指令以最大化性能，但是该技术能够保证内存操作的顺序性，在内存屏障前执行的操作一定会先于内存屏障后执行的操作。

想要在并发或者增量的标记算法中保证正确性，我们需要达成以下两种三色不变性（Tri-color invariant）中的一种：

- 强三色不变性 — 黑色对象不会指向白色对象，只会指向灰色对象或者黑色对象；
- 弱三色不变性 — 黑色对象指向的白色对象必须包含一条从灰色对象经由多个白色对象的可达路径；



上图分别展示了遵循强三色不变性和弱三色不变性的堆内存，遵循上述两个不变性中的任意一个，我们都能保证垃圾收集算法的正确性，而屏障技术就是在并发或者增量标记过程中保证三色不变性的重要技术。

垃圾收集中的屏障技术更像是一个钩子方法，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，根据操作类型的不同，我们可以将它们分成读屏障（Read barrier）和写屏障（Write barrier）两种，因为读屏障需要在读操作中加入代码片段，对用户程序的性能影响很大，所以编程语言往往都会采用写屏障保证三色不变性。

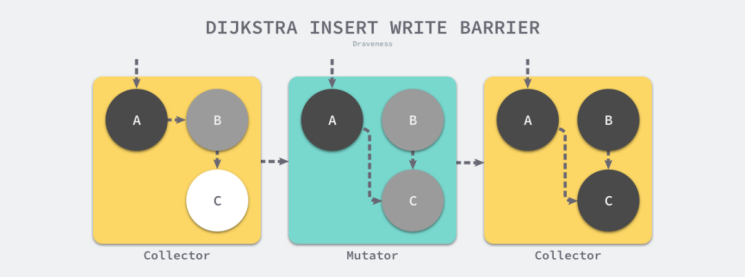
我们在这里想要介绍的是 Go 语言中使用的两种写屏障技术，分别是 Dijkstra 提出的插入写屏障和 Yuasa 提出的删除写屏障，这里会分析它们如何保证三色不变性和垃圾收集器的正确性。

插入写屏障

Dijkstra 在 1978 年提出了插入写屏障，通过如下所示的写屏障，用户程序和垃圾收集器可以在交替工作的情况下保证程序执行的正确性：

```
func DijkstraWritePointer(slot *unsafe.Pointer, ptr unsafe.Pointer)
   shade(ptr)  //先将新下游对象 ptr 标记为灰色
   *slot = ptr
}

//说明：
添加下游对象(当前下游对象slot, 新下游对象ptr) {
//step 1
标记灰色(新下游对象ptr)

//step 2
当前下游对象slot = 新下游对象ptr
}

//场景：
A.添加下游对象(nil, B) //A 之前没有下游， 新添加一个下游对象B， B被标记为灰色
A.添加下游对象(C, B) //A 将下游对象C 更换为B， B被标记为灰色
```

上述插入写屏障的伪代码非常好理解，每当执行类似 *slot = ptr 的表达式时，我们会执行上述写屏障通过 shade 函数尝试改变指针的颜色。如果 ptr 指针是白色的，那么该函数会将该对象设置成灰色，其他情况则保持不变。



假设我们在应用程序中使用 Dijkstra 提出的插入写屏障，在一个垃圾收集器和用户程序交替运行的场景中会出现如上图所示的标记过程：

1. 垃圾收集器将根对象指向 A 对象标记成黑色并将 A 对象指向的对象 B 标记成灰色；
2. 用户程序修改 A 对象的指针，将原本指向 B 对象的指针指向 C 对象，这时触发写屏障将 C 对象标记成灰色；
3. 垃圾收集器依次遍历程序中的其他灰色对象，将它们分别标记成黑色；

Dijkstra 的插入写屏障是一种相对保守的屏障技术，它会将**有存活可能的对象都标记成灰色**以满足强三色不变性。在如上所示的垃圾收集过程中，实际上不再存活的 B 对象最后没有被回收；而如果我们在第二和第三步之间将指向 C 对象的指针改回指向 B，垃圾收集器仍然认为 C 对象是存活的，这些被错误标记的垃圾对象只有在下一个循环才会被回收。

插入式的 Dijkstra 写屏障虽然实现非常简单并且也能保证强三色不变性，但是它也有明显的缺点。因为栈上的对象在垃圾收集中也会被认为是根对象，所以为了保证内存的安全，Dijkstra 必须为栈上的对象增加写屏障或者在标记阶段完成重新对栈上的对象进行扫描，这两种方法各有各的缺点，前者会大幅度增加写入指针的额外开销，后者重新扫描栈对象时需要暂停程序，垃圾收集算法的设计者需要在这两者之前做出权衡。

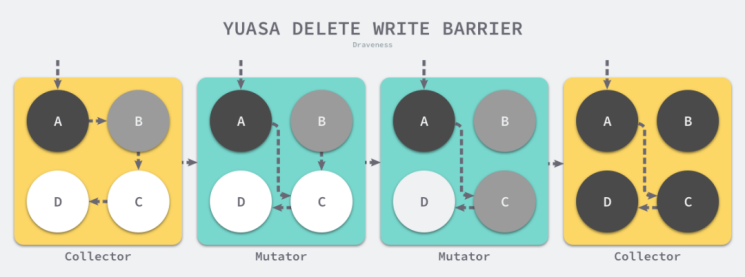
删除写屏障

Yuasa 在 1990 年的论文 Real-time garbage collection on general-purpose machines 中提出了删除写屏障，因为一旦该写屏障开始工作，它会保证开启写屏障时堆上所有对象的可达，所以也被称作快照垃圾收集（Snapshot GC）

该算法会使用如下所示的写屏障保证增量或者并发执行垃圾收集时程序的正确性：

```
// 黑色赋值器 Yuasa 屏障
func YuasaWritePointer(slot *unsafe.Pointer, ptr unsafe.Pointer) {
  shade(*slot) 先将*slot标记为灰色
  *slot = ptr
}

//说明:
添加下游对象(当前下游对象slot， 新下游对象ptr) {
 //step 1
 if (当前下游对象slot是灰色 || 当前下游对象slot是白色) {
       标记灰色(当前下游对象slot)    //slot为被删除对象， 标记为灰色
}
 //step 2
 当前下游对象slot = 新下游对象ptr
}

//场景
A.添加下游对象(B, nil)  //A对象，删除B对象的引用。B被A删除，被标记为灰(如果B之前为白)
A.添加下游对象(B, C)    //A对象，更换下游B变成C。B被A删除，被标记为灰(如果B之前为白)
```

上述代码会在老对象的引用被删除时，将白色的老对象涂成灰色，这样删除写屏障就可以保证弱三色不变性，老对象引用的下游对象一定可以被灰色对象引用。



假设我们在应用程序中使用 Yuasa 提出的删除写屏障，在一个垃圾收集器和用户程序交替运行的场景中会出现如上图所示的标记过程：

1. 垃圾收集器将根对象指向 A 对象标记成黑色并将 A 对象指向的对象 B 标记成灰色；
2. 用户程序将 A 对象原本指向 B 的指针指向 C，触发删除写屏障，但是因为 B 对象已经是灰色的，所以不做改变；
3. 用户程序将 B 对象原本指向 C 的指针删除，触发删除写屏障，白色的 C 对象被涂成灰色；

4. 垃圾收集器依次遍历程序中的其他灰色对象，将它们分别标记成黑色；

上述过程中的第三步触发了 Yuasa 删除写屏障的着色，因为用户程序删除了 B 指向 C 对象的指针，所以 C 和 D 两个对象会分别违反强三色不变性和弱三色不变性：

- 强三色不变性 — 黑色的 A 对象直接指向白色的 C 对象；
- 弱三色不变性 — 垃圾收集器无法从某个灰色对象出发，经过几个连续的白色对象访问白色的 C 和 D 两个对象；

Yuasa 删除写屏障通过对 C 对象的着色，保证了 C 对象和下游的 D 对象能够在这一次垃圾收集的循环中存活，避免发生悬挂指针以保证用户程序的正确性。

混合写屏障

在 Go 语言 v1.7 版本之前，运行时会使用 Dijkstra 插入写屏障保证强三色不变性，但是运行时并没有在所有的垃圾收集根对象上开启插入写屏障。因为应用程序可能包含成百上千的 Goroutine，而垃圾收集的根对象一般包括全局变量和栈对象，如果运行时需要在几百个 Goroutine 的栈上都开启写屏障，会带来巨大的额外开销，所以 Go 团队在实现上选择了在标记阶段完成时暂停程序、将所有栈对象标记为灰色并重新扫描，在活跃 Goroutine 非常多的程序中，重新扫描的过程需要占用 10 ~ 100ms 的时间。

Go 语言在 v1.8 组合 Dijkstra 插入写屏障和 Yuasa 删除写屏障构成了如下所示的混合写屏障，该写屏障会将被覆盖的对象标记成灰色并在当前栈没有扫描时将新对象也标记成灰色：

```
writePointer(slot, ptr):
  shade(*slot)
  if current stack is grey:
     shade(ptr)
  *slot = ptr
```

为了移除栈的重扫描过程，除了引入混合写屏障之外，在垃圾收集的标记阶段，我们还需要将创建的所有新对象都标记成黑色，防止新分配的栈内存和堆内存中的对象被错误地回收，因为栈内存在标记阶段最终都会变为黑色，所以不再需要重新扫描栈空间。总结来说主要有这几点：

- GC 开始将栈上的对象全部扫描并标记为黑色；
- GC 期间，任何在栈上创建的新对象，均为黑色；
- 被删除的堆对象标记为灰色；
- 被添加的堆对象标记为灰色；

演进过程

Go 语言的垃圾收集器从诞生的第一天起就一直在演进，除了少数几个版本没有大更新之外，几乎每次发布的小版本都会提升垃圾收集的性能，而与性能一同提升的还有垃圾收集器代码的复杂度，本节将从 Go 语言 v1.0 版本开始分析垃圾收集器的演进过程。

- v1.0 — 完全串行的标记和清除过程，需要暂停整个程序；
- v1.1 — 在多核主机并行执行垃圾收集的标记和清除阶段；
- v1.3 — 运行时基于**只有指针类型的值包含指针**的假设增加了对栈内存的精确扫描支持，实现了真正精确的垃圾收集；
  - 将`unsafe.Pointer`类型转换成整数类型的值认定为不合法的，可能会造成悬挂指针等严重问题；
- v1.5 — 实现了基于**三色标记清扫的并发**垃圾收集器；
  - 大幅度降低垃圾收集的延迟从几百 ms 降低至 10ms 以下；
  - 计算垃圾收集启动的合适时间并通过并发加速垃圾收集的过程；
- v1.6 — 实现了去中心化的垃圾收集协调器；
  - 基于显式的状态机使得任意 Goroutine 都能触发垃圾收集的状态迁移；
  - 使用密集的位图替代空闲链表表示的堆内存，降低清除阶段的 CPU 占用；
- v1.7 — 通过**并行栈收缩**将垃圾收集的时间缩短至 2ms 以内；
- v1.8 — 使用**混合写屏障**将垃圾收集的时间缩短至 0.5ms 以内；
- v1.9 — 彻底移除暂停程序的重新扫描栈的过程；
- v1.10 — 更新了垃圾收集调频器（Pacer）的实现，分离软硬堆大小的目标；
- v1.12 — 使用**新的标记终止算法**简化垃圾收集器的几个阶段；
- v1.13 — 通过新的 Scavenger 解决瞬时内存占用过高的应用程序向操作系统归还内存的问题；
- v1.14 — 使用全新的页分配器**优化内存分配的速度**；
- v1.15 — 改进编译器和运行时内部的CL 226367，它使编译器可以将更多的x86寄存器用于垃圾收集器的写屏障调用
- v1.16 — Go runtime默认使用MADV_DONTNEED更积极的将不用的内存释放给OS

GC 过程

Golang GC 相关的代码在`runtime/mgc.go`文件下，可以看见gc总共分为4个阶段：

- **1. sweep termination（清理终止）**

a. 暂停程序，触发STW。所有的 P（处理器）都会进入 safe-point（安全点）；

b. 清理未被清理的 span 。如果当前垃圾收集是强制触发的，需要处理还未被清理的内存管理单元；

- **2. the mark phase（标记阶段）**

  a. 将GC状态`gcphase`从`_GCoff`改成`_GCmark`、开启写屏障、启用协助线程（mutator assists）、将根对象入队

  b. 恢复程序执行，标记进程（mark workers）和协助程序会开始并发标记内存中的对象，写屏障会覆盖的重写指针和新指针（标记成灰色），而所有新创建的对象都会被直接标记成黑色；

  c. GC执行根节点的标记，这包括扫描所有的栈、全局对象以及不在堆中的运行时数据结构。扫描goroutine 栈会导致 goroutine 停止，并对栈上找到的所有指针加置灰，然后继续执行 goroutine。

  d. GC遍历灰色对象队列，会将灰色对象变成黑色，并将该指针指向的对象置灰。

  e. 由于GC工作分布在本地缓存中，GC 会使用分布式终止算法（distributed termination algorithm）来检测何时不再有根标记作业或灰色对象，如果没有了 GC 会转为mark termination（标记终止）

- **3. mark termination（标记终止）**

  a. STW

  b. 将GC状态`gcphase`切换至 `_GCmarktermination` ，关闭gc工作线程和协助程序

  c. 执行housekeeping，例如刷新mcaches

- **4. the sweep phase（清理阶段）**

  a. 将GC状态`gcphase`切换至 `_GCoff` 来准备清理阶段，初始化清理阶段并关闭写屏障

  b. 恢复用户程序，从现在开始，所有新创建的对象会标记成白色；如果有必要，在使用前分配清理spans

  c. 后台并发清理所有的内存管理类单元

GC过程代码示例

```
package main

import (
"os"
"runtime"
"runtime/trace"
)

func gcfinished() *int {
p := 1
runtime.SetFinalizer(&p, func(_ *int) {
println("gc finished")
})
return &p
}
func allocate() {
_ = make([]byte, int((1<<20)*0.25))
}
func main() {
f, _ := os.Create("trace.out")
defer f.Close()
trace.Start(f)
defer trace.Stop()
gcfinished()
// 当完成 GC 时停止分配
for n := 1; n < 50; n++ {
println("#allocate: ", n)
allocate()
}
println("terminate")
}
```

运行程序

```
hewittwang@HEWITTWANG-MB0 rtx % GODEBUG=gctrace=1 go run new1.go
gc 1 @0.015s 0%: 0.015+0.36+0.043 ms clock, 0.18+0.55/0.64/0.13+0.52 ms
cpu, 4->4->0 MB, 5 MB goal, 12 P
gc 2 @0.024s 1%: 0.045+0.19+0.018 ms clock, 0.54+0.37/0.31/0.041+0.22 ms cpu, 4->4->0
MB, 5 MB goal, 12 P
....
```

栈分析

```
gc 2        : 第一个GC周期
@0.024s     : 从程序开始运行到第一次GC时间为0.024 秒
1%          : 此次GC过程中CPU 占用率

wall clock
0.045+0.19+0.018 ms clock
0.045 ms    : STW，Marking Start，开启写屏障
0.19 ms     : Marking阶段
0.018 ms    : STW，Marking终止，关闭写屏障

CPU time
0.54+0.37/0.31/0.041+0.22 ms cpu
0.54 ms     : STW，Marking Start
0.37 ms     : 辅助标记时间
0.31 ms     : 并发标记时间
0.041 ms    : GC 空闲时间
0.22 ms     : Mark 终止时间

4->4->0 MB,  5 MB goal
4 MB        : 标记开始时，堆大小实际值
4 MB        : 标记结束时，堆大小实际值
0 MB        : 标记结束时，标记为存活对象大小
5 MB        : 标记结束时，堆大小预测值

12 P        : 本次GC过程中使用的goroutine 数量
```

GC 触发条件

运行时会通过`runtime.gcTrigger.test`方法决定是否需要触发垃圾收集，当满足触发垃圾收集的基本条件（即满足`_GCoff`阶段的退出条件）时 — 允许垃圾收集、程序没有崩溃并且没有处于垃圾收集循环，该方法会根据三种不同方式触发进行不同的检查：

```
//mgc.go 文件 runtime.gcTrigger.test
func (t gcTrigger) test() bool {
  //测试是否满足触发垃圾手机的基本条件
  if !memstats.enablegc || panicking != 0 || gcphase != _GCoff {
     return false
  }
  switch t.kind {
    case gcTriggerHeap:    //堆内存的分配达到达控制器计算的触发堆大小
      // Non-atomic access to gcController.heapLive for performance. If
      // we are going to trigger on this, this thread just
      // atomically wrote gcController.heapLive anyway and we'll see our
      // own write.
      return gcController.heapLive >= gcController.trigger
    case gcTriggerTime:     //如果一定时间内没有触发，就会触发新的循环，该出发条件由`runtime.forcegcperiod`变量控制，默认为 2 分钟；
      if gcController.gcPercent < 0 {
        return false
     }
      lastgc := int64(atomic.Load64(&memstats.last_gc_nanotime))
      return lastgc != 0 && t.now-lastgc > forcegcperiod
    case gcTriggerCycle:     //如果当前没有开启垃圾收集，则触发新的循环；
      // t.n > work.cycles, but accounting for wraparound.
      return int32(t.n-work.cycles) > 0
  }
  return true
}
```

用于开启垃圾回收的方法为`runtime.gcStart`，因此所有调用该函数的地方都是触发GC的代码

- `runtime.mallocgc` 申请内存时根据堆大小触发GC
- `runtime.GC` 用户程序手动触发GC
- `runtime.forcegchelper` 后台运行定时检查触发GC

申请内存触发 runtime.mallocgc

Go运行时会将堆上的对象按大小分成微对象、小对象和大对象三类，这三类对象的创建都可能会触发新的GC

1. 当前线程的内存管理单元中不存在空闲空间时，创建微对象`(noscan && size < maxTinySize)`和小对象需要调用`runtime.mcache.nextFree`从中心缓存或者页堆中获取新的管理单元，这时如果span满了就会导致返回的`shouldhelpgc=true`，就可能触发垃圾收集；
2. 当用户程序申请分配 32KB 以上的大对象时，一定会构建`runtime.gcTrigger`结构体尝试触发垃圾收集；

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    省略代码 ...
    shouldhelpgc := false
  dataSize := size
  c := getMCache()        //尝试获取mCache。如果没启动或者没有P,返回nil;

    省略代码 ...
    if size <= maxSmallSize {
        if noscan && size < maxTinySize { // 微对象分配
  省略代码 ...
            v := nextFreeFast(span)
            if v == 0 {
                v, span, shouldhelpgc = c.nextFree(tinySpanClass)
            }
      省略代码 ...
      } else {       //小对象分配
          省略代码 ...
            if v == 0 {
                v, span, shouldhelpgc = c.nextFree(spc)
            }
          省略代码 ...
      }
    } else {
        shouldhelpgc = true
        省略代码 ...
    }
  省略代码 ...
    if shouldhelpgc {       //是否应该触发gc
```

```
        if t := (gcTrigger{kind: gcTriggerHeap}); t.test() {    //如果满足gc触
            gcStart(t)
        }
    }
    省略代码 ...
    return x
}
```

这个时候调用t.test()执行的是gcTriggerHeap情况，只需要判断gcController.heapLive >= gcController.trigger的真假就可以了。heapLive 表示垃圾收集中存活对象字节数， trigger表示触发标记的堆内存大小的；当内存中存活的对象字节数大于触发垃圾收集的堆大小时，新一轮的垃圾收集就会开始。

1. heapLive — 为了减少锁竞争，运行时只会在中心缓存分配或者释放内存管理单元以及在堆上分配大对象时才会更新；
2. trigger — 在标记终止阶段调用 runtime.gcSetTriggerRatio 更新触发下一次垃圾收集的堆大小，它能够决定触发垃圾收集的时间以及用户程序和后台处理的标记任务的多少，利用反馈控制的算法根据堆的增长情况和垃圾收集 CPU 利用率确定触发垃圾收集的时机。

手动触发 runtime.GC

用户程序会通过 runtime.GC 函数在程序运行期间主动通知运行时执行，该方法在调用时会阻塞调用方直到当前垃圾收集循环完成，在垃圾收集期间也可能会通过 STW 暂停整个程序：

```
func GC() {
    //在正式开始垃圾收集前，运行时需要通过runtime.gcWaitOnMark等待上一个循环的标记终
    n := atomic.Load(&work.cycles)
    gcWaitOnMark(n)

  //调用 `runtime.gcStart` 触发新一轮的垃圾收集
    gcStart(gcTrigger{kind: gcTriggerCycle, n: n + 1})

    //`runtime.gcWaitOnMark` 等待该轮垃圾收集的标记终止阶段正常结束;
    gcWaitOnMark(n + 1)

    // 持续调用 `runtime.sweepone` 清理全部待处理的内存管理单元并等待所有的清理工作
    for atomic.Load(&work.cycles) == n+1 && sweepone() != ^uintptr(0) {
```

```
            sweep.nbgsweep++
            Gosched()  //等待期间会调用 `runtime.Gosched` 让出处理器
        }

        //
        for atomic.Load(&work.cycles) == n+1 && !isSweepDone() {
            Gosched()
        }

        // 完成本轮垃圾收集的清理工作后，通过 `runtime.mProf_PostSweep` 将该阶段的堆内
        mp := acquirem()
        cycle := atomic.Load(&work.cycles)
        if cycle == n+1 || (gcphase == _GCmark && cycle == n+2) {    //仅限于没
            mProf_PostSweep()
        }
        releasem(mp)
    }
```

后台运行定时检查触发 runtime.forcegchelper

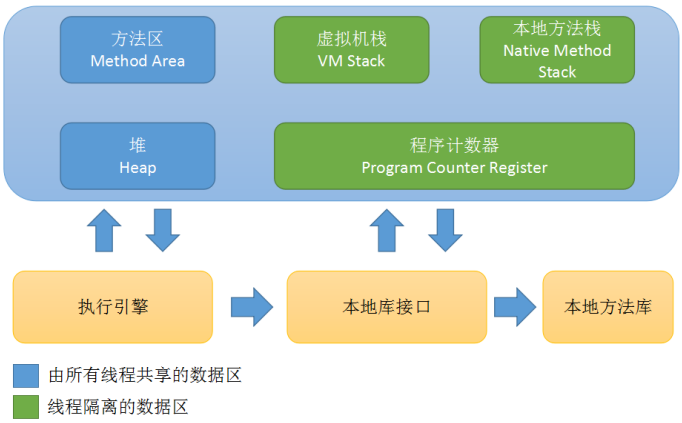
运行时会在应用程序启动时在后台开启一个用于强制触发垃圾收集的 Goroutine，该 Goroutine 调用 `runtime.gcStart` 尝试启动新一轮的垃圾收集：

```
// start forcegc helper goroutine
func init() {
    go forcegchelper()
}

func forcegchelper() {
    forcegc.g = getg()
    lockInit(&forcegc.lock, lockRankForcegc)
    for {
        lock(&forcegc.lock)
        if forcegc.idle != 0 {
            throw("forcegc: phase error")
        }
        atomic.Store(&forcegc.idle, 1)

```

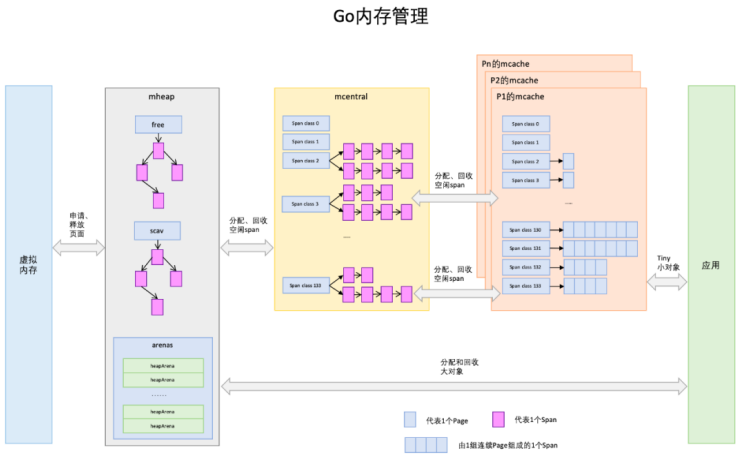
```
        //该 Goroutine 会在循环中调用runtime.goparkunlock主动陷入休眠等待其他 Gorou
        goparkunlock(&forcegc.lock, waitReasonForceGCIdle, traceEvGoBlock,

        if debug.gctrace > 0 {
            println("GC forced")
        }
        // Time-triggered, fully concurrent.
        gcStart(gcTrigger{kind: gcTriggerTime, now: nanotime()})
    }
}
```

Java和Go GC对比

垃圾回收区域



Java内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈3个区域随着线程而生，随着线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈的操作，每个栈帧中分配多少内存基本是在类结构确定下来时就已知的。而Java堆和方法区则不同，一个接口中的多个实现类需要的内存可能不同，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，因此，**Java堆和方法区是Java垃圾收集器管理的主要区域**。



go内存会分成堆区（Heap）和栈区（Stack）两个部分，程序在运行期间可以主动从堆区申请内存空间，这些内存由内存分配器分配并由垃圾收集器负责回收。栈区的内存由编译器自动进行分配和释放，栈区中存储着函数的参数以及局部变量，它们会随着函数的创建而创建，函数的返回而销毁。如果只申请和分配内存，内存终将枯竭。Go使用垃圾回收收集不再使用的span，把span释放交给mheap，mheap对span进行span的合并，把合并后的span加入scav树中，等待再分配内存时，由mheap进行内存再分配。**因此，Go堆是Go垃圾收集器管理的主要区域。**

触发垃圾回收的时机

Java 当应用程序空闲时,即没有应用线程在运行时,GC会被调用。因为GC在优先级最低的线程中进行,所以当应用忙时,GC线程就不会被调用,但以下条件除外。

Java堆内存不足时,GC会被调用。但是这种情况由于java是分代收集算法且垃圾收集器种类十分多，因此其触发各种垃圾收集器的GC时机可能不完全一致，这里我们说的为一般情况。

1. 当Eden区空间不足时Minor GC
2. 对象年龄增加到一定程度时Young GC
3. 新生代对象转入老年代及创建为大对象、大数组时会导致老年代空间不足，触发Old GC
4. System.gc()调用触发Full GC
5. 各种区块占用超过阈值的情况

Go则会根据以下条件进行触发：

- `runtime.mallocgc` 申请内存时根据堆大小触发GC
- `runtime.GC` 用户程序手动触发GC
- `runtime.forcegchelper` 后台运行定时检查触发GC

收集算法

当前Java虚拟机的垃圾收集采用**分代收集**算法，根据对象存活周期的不同将内存分为几块。比如在新生代中，每次收集都会有大量对象死去，所以可以选择“标记-复制”算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

当前Go的都是基于**标记清除**算法进行垃圾回收。

垃圾碎片的处理

由于Java的内存管理划分，因此容易产生垃圾对象，JVM这些年不断的改进和更新GC算法，JVM在处理内存碎片问题上更多采用**空间压缩和分代收集**的思想，例如在新生代使用“标记-复制”算法，G1收集器支持了对象移动以消减长时间运行的内存碎片问题，划分region的设计更容易把空闲内存归还给OS等设计。

由于Go的内存管理的实现，很难实现分代，而移动对象也可能会导致runtime更庞大复杂，因此Go在关于内存碎片的处理方案和Java并不太一样。

1. Go语言span内存池的设计，减轻了很多内存碎片的问题。

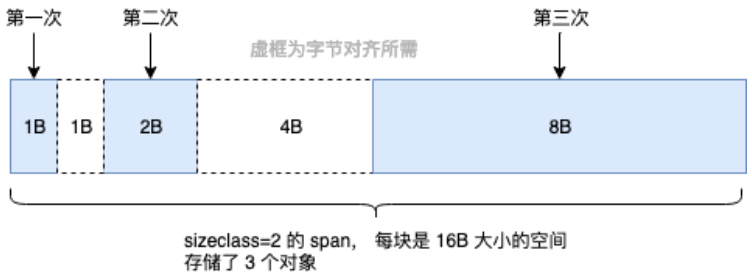
Go内存释放的过程如下：当 mcache 中存在较多空闲 span 时，会归还给 mcentral；而 mcentral 中存在较多空闲 span 时，会归还给 mheap；mheap 再归还给操作系统。这种设计主要有以下几个优势：

- 内存分配大多时候都是在用户态完成的，不需要频繁进入内核态。
- 每个 P 都有独立的 span cache，多个 CPU 不会并发读写同一块内存，进而减少 CPU L1 cache 的 cacheline 出现 dirty 情况，增大 cpu cache 命中率。
- 内存碎片的问题，Go 是自己在用户态管理的，在 OS 层面看是没有碎片的，使得操作系统层面对碎片的管理压力也会降低。
- mcache 的存在使得内存分配不需要加锁。

2. tcmalloc分配机制，Tiny对象和大对象分配优化，在某种程度上也导致基本没有内存碎片会出现。

比如常规上 sizeclass=1的 span，用来给 <= 8B 的对象使用，所以像 int32, byte, bool 以及小字符串等常用的微小对象，都会使用 sizeclass=1 的 span，但分配给他们 8B 的空间，大部分是用不上的。并且这些类型使用频率非常高，就会导致出现大量的内部碎片。

因此 Go 尽量不使用 sizeclass=1 的 span， 而是将 < 16B 的对象为统一视为 tiny 对象。分配时，从 sizeclass=2 的 span 中获取一个 16B 的 object 用以分配。如果存储的对象小于 16B，这个空间会被暂时保存起来 (mcache.tiny 字段)，下次分配时会复用这个空间，直到这个 object 用完为止。



以上图为例，这样的方式空间利用率是 (1+2+8) / 16 * 100% = 68.75%，而如果按照原始的管理方式，利用率是 (1+2+8) / (8 * 3) = 45.83%。源码中注释描述，说是对 tiny 对象的特殊处理，平均会节省 20% 左右的内存。如果要存储的数据里有指针，即使 <= 8B 也不会作为 tiny 对象对待，而是正常使用 sizeclass=1 的 span。

Go中，最大的 sizeclass 最大只能存放 32K 的对象。如果一次性申请超过 32K 的内存，系统会直接绕过 mcache 和 mcentral，直接从 mheap 上获取，mheap 中有一个 freelarge 字段管理着超大 span。

3. Go的对象(即struct类型)是可以分配在栈上的。

Go会在编译时做静态逃逸分析(Escape Analysis), 如果发现某个对象并没有逃出当前作用域，则会将对象分配在栈上而不是堆上，从而减轻了GC内存碎片回收压力。

比如如下代码

```
1  func F() {
2    temp := make([]int, 0, 20) //只是内函数内部申请的临时变量，并不会作为返回值返回，
3    temp = append(temp, 1)
4  }
5
6  func main() {
7    F()
8  }
```

运行代码如下，结果显示temp变量被分配在栈上并没有分配在堆上

```
1  hewittwang@HEWITTWANG-MB0 rtx % go build -gcflags=-m
2  # hello
```

```
./new1.go:4:6: can inline F
./new1.go:9:6: can inline main
./new1.go:10:3: inlining call to F
./new1.go:5:14: make([]int, 0, 20) does not escape
./new1.go:10:3: make([]int, 0, 20) does not escapeh
```

当我们把上述代码更改：

```
package main
import "fmt"

func F() {
  temp := make([]int, 0, 20)
  fmt.Print(temp)
}

func main() {
  F()
}
```

运行代码如下，结果显示temp变量被分配在堆上，这是由于temp传入了print函数里，编译器会认为变量之后还会被使用。因此就申请到堆上，申请到堆上面的内存才会引起垃圾回收，如果这个过程（特指垃圾回收不断被触发）过于高频就会导致 gc 压力过大，程序性能出问题。

```
hewittwang@HEWITTWANG-MB0 rtx % go build -gcflags=-m
# hello
./new1.go:9:11: inlining call to fmt.Print
./new1.go:12:6: can inline main
./new1.go:8:14: make([]int, 0, 20) escapes to heap
./new1.go:9:11: temp escapes to heap
./new1.go:9:11: []interface {}{...} does not escape
<autogenerated>:1: .this does not escape
```

“GC Roots” 的对象的选择

在Java中由于内存运行时区域的划分，通常会选择以下几种作为“GC Roots” 的对象：

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 本地方法栈(Native 方法)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- Java虚拟机内部引用
- 所有被同步锁持有的对象

而在Java中的不可达对象有可能会逃脱。即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；此外Java中由于存在运行时常量池和类，因此也需要对运行时常量池和方法区的类进行清理。

而Go的选择就相对简单一点，即全局变量和G Stack中的引用指针，简单来说就是全局量和go程中的引用指针。因为Go中没有类的封装概念，因而Gc Root选择也相对简单一些。

写屏障

为了解决并发三色可达性分析中的悬挂指针问题，出现了2种解决方案，分别是分别是“Dijkstra插入写屏障”和“Yuasa删除写屏障”

在java中，对上述2种方法都有应用，比如CMS是基于**Dijkstra插入写屏障**做并发标记的，G1、Shenandoah则是使用**Yuasa删除写屏障来实现的**

在 Go 语言 v1.7 版本之前，运行时会使用 Dijkstra 插入写屏障保证强三色不变性，Go 语言在 v1.8 组合 Dijkstra 插入写屏障和 Yuasa 删除写屏障构成了混合写屏障，混合写屏障结合两者特点，通过以下方式实现并发稳定的gc：

1. 将栈上的对象全部扫描并标记为黑色
2. GC期间，任何在栈上创建的新对象，均为黑色。
3. 被删除的对象标记为灰色。
4. 被添加的对象标记为灰色。

由于要保证栈的运行效率，混合写屏障是针对于堆区使用的。即**栈区不会触发写屏障，只有堆区触发**，由于栈区初始标记的可达节点均为黑色节点，因而也不需要第二次STW下的扫描。本质上是融合了插入屏障和删除屏障的特点，解决了插入屏障需要二次扫描的问题。同时针对于堆区和栈区采用不同的策略，保证栈的运行效率不受损。

总结

	Java	
GC区域	Java堆和方法区	
触发GC时机	分代收集导致触发时机很多	申请内存
垃圾收集算法	分代收集。在新生代（“标记-复制”）；老年代（“标记-清除”或“标记-整理”）	

垃圾种类	死亡对象（可能会逃脱）、废弃常量和无用的类	全局变量
标记阶段	三色可达性分析算法（插入写屏障，删除写屏障）	三色可达
空间压缩整理	是	
内存分配	指针碰撞/空闲列表	
垃圾碎片解决方案	分代GC、对象移动、划分region等设计	Go语言span内存分

从垃圾回收的角度来说，经过多代发展，Java的垃圾回收机制较为完善，Java划分新生代、老年代来存储对象。对象通常会在新生代分配内存，多次存活的对象会被移到老年代，由于新生代存活率低，产生空间碎片的可能性高，通常选用“标记-复制”作为回收算法，而老年代存活率高，通常选用“标记-清除”或“标记-整理”作为回收算法，压缩整理空间。

Go是非分代的、并发的、基于三色标记和清除的垃圾回收器，它的优势要结合它tcmalloc内存分配策略才能体现出来，因为小微对象的分配均有自己的内存池，所有的碎片都能被完美复用，所以GC不用考虑空间碎片的问题。

参考文献

- Go语言设计与实现
- 一个专家眼中的Go与Java垃圾回收算法大对比
- Go语言问题集
- CMS垃圾收集器
- Golang v 1.16版本源码