Please visit website: http://cxyroad.com

Golang 垃圾回收：一次 GC 周期的详细过程

前言

这篇文章想和大家讨论一下 Golang 内存管理机制中的垃圾回收（Garbage Collection，简称 GC），本篇文章着重讲述 GC 相关的理论知识，包括：栈内存和堆内存的特性、常用的垃圾回收的算法、Go GC 的历史进程、一次完整的 GC 周期。在讲述理论知识的同时，需要理清楚以下几个问题：

1. 栈内存会不会被 GC 管理?
2. GC 为什么要扫描栈内存?
3. 栈内存为什么不能使用屏障机制保障内存状态的一致性?
4. 1.5 版本 GC 为什么还要重新扫描一遍栈内存?
5. 1.8 版本以后，GC 到底还存不存在 STW?
6. 1.8 版本以后，如果还存在 STW，那什么时机触发，什么时机结束，作用是什么?
7. “三色标记 + 混合写屏障”如何保障并发 GC 不会出现内存一致性错误?

专业名词解释：

* **垃圾回收（Garbage Collection，简称 GC）**：一种内存管理策略，由垃圾收集器以类似守护协程的方式在后台运作，按照既定的策略为用户回收那些不再被使用的对象，释放对应的内存空间。
* **根对象（root 对象）**：指那些可以直接访问的对象，它们作为垃圾回收过程的起点。从这些根对象开始，垃圾回收器会递归地扫描和标记所有可达的对象，以便区分哪些对象是可达的（不应该被回收）和哪些对象是不可达的（可以被回收）。如：栈局部变量、全局变量、寄存器中的变量等
* **STW（stop the word）**：指的是在某些垃圾回收的阶段，必须暂停应用程序的 Goroutines，以确保内存状态的一致性和正确性。在 STW 事件期间，应用程序的执行会被完全暂停，直到垃圾回收器完成其必要的操作。

想一起学习 Go 语言进阶知识的同学可以 **点赞++收藏** 哦！

1.栈和堆内存

聊 GC 之前一定要先聊清楚内存是什么，因为 GC 的主要工作就是回收内存。垃圾回收器（GC）是一个系统，这个系统通过标识内存的哪些部分不再需要来代表应用程序回收内存，而在 Go 语言中，栈内存和堆内存有明确的管理机制，理解它们的区别和使用场景对于理解 GC 非常重要。

1.1 栈内存

首先声明一点：栈内存不需要 GC 进行管理。那什么是栈内存呢?

比如说：为了存储局部变量中的非指针变量，Go 语言会安排其内存的分配，并将其绑定到创建它的**词法作用域**中。 一般来说，这比依赖 GC 更有效率，因为 Go 语言编译器能够预先确定何时释放内存，并发出清理内存的机器指令。 通常，我们把这种分配内存的方式称为“**栈分配**”，因为空间存储在 goroutine 栈中。

栈内存是每个 goroutine 独有的内存空间，表现在源码里就是如下形态：（参考文章 G 对象的创建 一节 ）：

```
type g struct {
stack       stack   // goroutine 使用的栈
    // 下面两个成员用于栈溢出检查，实现栈的自动伸缩，抢占调度也会用到
stackguard0
stackguard0 uintptr
stackguard1 uintptr
    ...
}
```

栈内存用于存储局部变量、函数调用信息等（栈内存如何使用，可以参考文章 Go 函数调用栈结构）。Go 中栈内存有以下特点：

* **栈内存的特点是分配和回收速度非常快，但栈的大小是有限的**
+ 栈内存的分配和回收速度非常快，因为栈是一个连续的内存区域，分配和回收只需要调整栈指针；
+ 在 Go 语言中，每个 Goroutine 的栈初始大小一般非常小，通常是 2KB 或 4KB，这是为了节省内存和提高并发执行的效率；Go 中已经通过栈复制实现了动态扩缩栈内存的能力，无需任何手动干预，尽管栈内存可以动态扩展，但每个 Goroutine 的栈大小还是有限制的，通常限制在几 GB 以内。

* **栈内存的管理由编译器和运行时负责，而不是垃圾回收器**。
+ 垃圾回收器主要管理的是堆内存中的对象。当函数调用时，相关变量会被压入栈中，函数返回时，这些变量会被弹出栈并回收。
+ 栈上对象的生命周期由函数调用的开始和结束决定，当函数返回时，栈上的局部变量会被自动销毁。

1.2 堆内存

当 Go 语言无法确定一个对象的生存期时，也就无法以“**栈分配**”的方式为其分配内存，此时该对象会”**逃逸到堆**“，“堆”可以被认为是内存分配的一个大杂烩，Go语言的对象需要被放置在堆的某个地方，此时的对象是动态分配的，编译器和运行时无法预先确定其内存应该在何时释放，因此 GC 闪亮登场：专门标识和清理动态内存分配的系统。

堆内存是由 Go 运行时（runtime）管理的全局内存区域，用于存储在运行时动态分配的对象。与栈内存不同，堆内存可以在程序的整个生命周期内存在，直到不再被引用并被垃圾收集器回收。堆内存适用于需要在不同函数之间共享的对象或需要长期存在的对象。
具体来说，堆内存的管理涉及到以下几个方面：

1. **内存分配器**：Go 运行时包含一个内存分配器，用于在堆上分配内存。分配器会根据需要分配适当大小的内存块并返回指向该内存块的指针。
2. **垃圾收集器**：Go 语言采用了垃圾收集机制来自动管理堆内存。垃圾收集器会定期扫描堆内存，标记不再被引用的对象，并回收这些对象占用的内存。
3. **逃逸分析**：Go 编译器在编译时会进行逃逸分析，以确定变量是应该分配在栈上还是堆上。如果一个变量的生命周期超出了函数调用的范围（如：返回指针或在 goroutine 中使用），编译器会将其分配到堆上。（Go 语言的对象是否逃逸，取决于使用它的上下文和 Go 语言编译器的逃逸分析算法）

1.3 举个例子

为了更好地理解栈内存和堆内存，我们通过简单的例子来看一下栈内存和堆内存的使用：

```
package main
```

```
import "fmt"

type Data struct {
    value int
}

func createData() *Data {
    // 在堆上分配 Data 对象
    data := &Data{value: 42}
    // 返回指向 Data 对象的指针
    return data
}

func main() {
    // createData 函数返回的指针指向堆上的 Data 对象
    d := createData()
    // 打印指针指向的值
    fmt.Println(d.value)
}
```

在上面的代码中：

1. createData 函数在堆上分配了一个 Data 对象，并返回指向该对象的指针。
2. 当 createData 函数执行完成后，栈上的局部变量 data 会被回收。
3. 但是，由于 main 函数中的局部变量 d 仍然持有指向堆上 Data 对象的指针（这就是下文提到的对象之间的引用），所以 Data 对象不会被回收。
4. 当 main 函数执行完成后，栈上的局部变量 d 也会被回收，此时 Data 对象将不再被引用，垃圾收集器会在适当的时候回收该对象。

总结一下特点：

* 栈内存的局部变量在函数执行完成后会被回收，有明确生命周期。
* 指针变量指向的堆对象不会立即被回收，除非该对象不再被任何变量引用。

根据这个现实的例子还能解释一个问题：既然 GC 不管理栈内存，为什么还要扫描栈内存？**尽管垃圾回收器（GC）不直接管理栈内存，但它需要扫描栈内存以确保正确地识别和处理堆对象的引用，栈内存的局部变量是一类 GC 扫描的根对象。**

2.垃圾回收算法

========

本节罗列一些经典的 GC 算法的基本思想和优缺点，拓宽一下视野。

2.1 引用计数（reference counting）

主要思想

每个单元维护一个域，保存其它单元指向它的引用数量（类似有向图的入度）。当引用数量为 0 时，将其回收。引用计数是渐进式的，能够将内存管理的开销分布到整个程序之中。

优点

1. 渐进式。内存管理与用户程序的执行交织在一起，将 GC 的代价分散到整个程序。不像标记–清除算法需要 STW (Stop The World，GC 的时候挂起用户程序)。
2. 算法易于实现。
3. 内存单元能够很快被回收，内存利用率会很高。相比于其他垃圾回收算法，堆被耗尽或者达到某个阈值才会进行垃圾回收。

缺点

1. 原始的引用计数不能处理循环引用（这是最致命的缺点了）。不过针对这个问题，也有很多解决方案，比如强引用等。
2. 维护引用计数会降低运行效率：内存单元的更新删除等都需要维护相关的内存单元的引用计数，相比于一些追踪式的垃圾回收算法并不需要这些代价。

2.2 标记–清除（mark & sweep）

标记–清除算法是一种自动内存管理，基于追踪的垃圾收集算法。算法思想在 70 年代就提出了，是一种非常古老的算法。

主要思想

内存单元并不会在变成垃圾立刻回收，而是保持不可达状态，直到到达某个阈值或者固定时间长度。这个时候系统会挂起用户程序，也就是 STW（Stop-The-World），转而执行垃圾回收程序。垃圾回收程序对所有的存活单元进行一次全局遍历确定哪些单元可以回收。

算法分两个阶段：标记和清除：

标记阶段（Mark Phase）

在标记阶段，垃圾回收器遍历所有的根对象（Root Objects），并递归地标记所有可达的对象。被标记的对象表示它们仍然在使用中，不应被回收。

* 根对象扫描：从根对象开始，标记所有直接引用的对象。
* 递归标记：递归地遍历每个被标记的对象，标记它们所引用的其他对象，直到所有可达的对象都被标记。

清除阶段（Sweep Phase）

在清除阶段，垃圾回收器遍历堆中的所有对象，回收那些未被标记的对象。这些未被标记的对象被认为是不可达的，可以安全地释放其占用的内存。

* 遍历堆内存：遍历整个堆内存，检查每个对象的标记状态。
* 释放未标记对象：释放所有未被标记的对象的内存，将这些内存区域返回给内存管理器，以便重新分配。

优点

1. 简单易实现：标记-清除算法的实现相对简单，易于理解和实现。
2. 无需移动对象：该算法不需要移动对象，因此适用于那些对象地址不能改变的场景。

缺点

1. 内存碎片：标记-清除算法会产生内存碎片，因为它只是简单地释放未标记对象的内存，并不整理内存空间。
2. 暂停时间长：在标记和清除阶段，应用程序通常会暂停，可能导致较长的暂

停时间，影响应用程序的响应性能。

针对内存碎片问题，出现了 **标记压缩（Mark-Compact）** 算法，是在标记清除算法的基础上做了升级，在第二步”清除“的同时还会对存活对象进行压缩整合，使得整体空间更为紧凑，从而解决内存碎片问题。标记压缩算法在功能性上呈现得很出色，而其存在的缺陷也很简单，就是实现时会有很高的复杂度。

2.3 节点复制（Copying Garbage Collection）

节点复制（Copying Garbage Collection），也称为复制收集，是一种垃圾回收算法，主要用于管理年轻代（Young Generation）的内存。

主要思想

将存活的对象从一个内存区域复制到另一个内存区域，以此来回收内存。复制算法通过避免内存碎片和简化内存分配，提高了垃圾回收的效率。复制垃圾回收将堆内存划分为两个等大小的区域：活动区（From-Space）和空闲区（To-Space）。在任何给定时间，只有一个区域用于分配新对象，另一个区域保持空闲。当活动区用满时，垃圾回收器会将存活的对象复制到空闲区，然后交换两个区域的角色。

优点

1. 高效的内存分配：
* 线性分配：复制垃圾回收使用线性分配内存，不需要维护复杂的空闲列表，简化了内存分配过程。
* 避免碎片：通过将存活的对象集中在一起，复制垃圾回收避免了内存碎片问题，提高了内存利用率。
2. 快速的垃圾回收：
* 只处理存活对象：复制垃圾回收只处理存活的对象，忽略不可达的对象，因此回收速度较快。
* 简化的引用更新：通过复制对象，所有引用都指向新的内存位置，简化了引用更新过程。
3. 支持并发和并行垃圾回收：
* 并发复制：复制垃圾回收可以支持并发和并行的垃圾回收，提高了垃圾回收的性能。

缺点

1. 内存开销：
* 双倍内存：复制垃圾回收需要将堆内存划分为两个等大小的区域，导致内存开销较大。
* 空间浪费：在任何给定时间，只有一半的内存区域被使用，另一半保持空闲，导致空间浪费。
2. 适用性有限：
* 适用于年轻代：复制垃圾回收主要适用于年轻代，因为年轻代对象存活率较低，复制的开销较小。
* 不适用于老年代：对于老年代对象，存活率较高，复制开销较大，因此不适用于老年代。

2.4 分代收集（Generational Garbage Collection）

分代收集（Generational Garbage Collection）是一种优化的垃圾回收算法，基于对象生命周期的假设：大多数对象在创建后很快就会变得不可达（称为“年轻代”对象），而少数对象会存活较长时间（称为“老年代”对象）。通过将堆内存分为多个代，并对不同代采用不同的垃圾回收策略，分代收集提高了垃圾回收的效率和性能。

主要思想

1. 年轻代（Young Generation）：
* 特点：存放新创建的对象。大多数对象在此代中很快变得不可达。
* 回收策略：采用较频繁的垃圾回收（通常是复制算法或标记-清除算法的变种），因为绝大多数对象在此代中会迅速被回收。
2. 老年代（Old Generation）：
* 特点：存放从年轻代晋升的长寿命对象。对象在此代中存活时间较长。
* 回收策略：采用较少频繁但更全面的垃圾回收（通常是标记-清除或标记-整理算法），因为对象在此代中存活时间较长，回收频率不需要太高。
3. 持久代（Permanent Generation）（在一些 JVM 实现中，如 HotSpot）：
* 特点：存放类元数据和方法区内容。在 Java 8 之后被元空间（Metaspace）取代。
* 回收策略：根据需要进行回收，通常不需要频繁回收。

优点

1. 提高垃圾回收效率：
* 针对性强：通过将对象分代并采用不同的回收策略，可以针对不同生命周期的对象进行优化，提高垃圾回收的效率。

* 减少暂停时间：年轻代回收通常非常快速，减少了垃圾回收的暂停时间。
2. 提高内存利用率：
* 减少内存碎片：通过不同代的回收策略，可以减少内存碎片，提高内存利用率。
* 优化内存分配：分代收集可以更好地管理内存分配，使得内存分配和回收更加高效。
3. 更好地适应多线程环境：
* 并发回收：分代收集算法可以更好地适应多线程环境，通过并发回收进一步提高性能。

缺点

1. 额外的复杂性：
* 实现复杂：分代收集算法需要额外的实现复杂性，包括管理不同代的内存区域和处理对象的晋升。
* 调优复杂：需要对不同代的垃圾回收策略进行调优，以达到最佳性能。
2. 晋升代的开销：
* 晋升过程：对象从年轻代晋升到老年代会产生一定的开销，尤其是在对象频繁晋升的情况下，可能会影响性能。
* 老年代回收：老年代的垃圾回收通常较为复杂和耗时，可能会导致较长的暂停时间。
3. 不适用于所有应用：
* 对象生命周期不均匀：对于某些生命周期不均匀的应用（如大量持久对象和少量短期对象），分代收集的优势可能不明显，甚至可能带来额外的开销。

3.Go GC 历史进程

本节将讲述 Go GC 的历史进程，主要是经历了以下三个重要阶段：Go 1.3 标记清除，Go 1.5 并发三色标记法 + 插入写屏障，Go 1.8 三色标记 + 混合写屏障机制。核心思想是优化 STW 时间，提升用户体验。

3.1 Go 1.3 标记清除

该版本基于最基本的标记–清除算法实现
具体步骤：

1. STW：暂停程序业务逻辑。
2. 标记：从根对象出发找出所有可达的对象，并做上标记。
3. 清除：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。
4. 恢复程序执行。

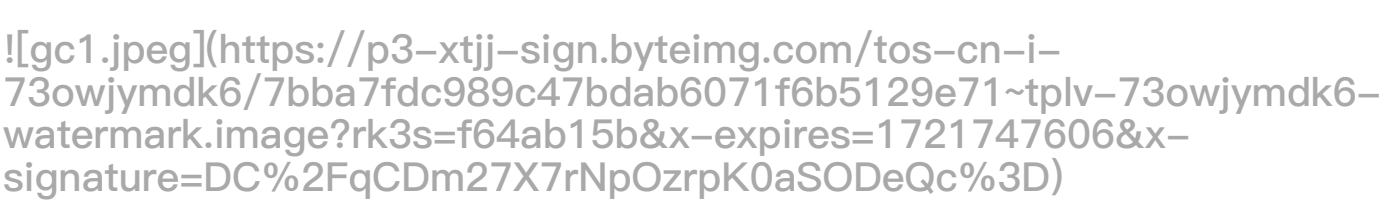
* 优点：实现简单，易于理解。
* 缺点：整个 GC 过程都需要 STW，严重影响性能。
* 后续优化：清除阶段不需要 STW，由于这些不可达对象不可能再被引用，因此可以在不暂停应用程序执行的情况下，直接清理这些对象。

3.2 Go 1.5 并发三色标记法 + 插入写屏障

在 Go 1.5 的版本引入了并发垃圾回收机制，允许用户协程和后台的 GC 协程并发运行，大大地提高了用户体验，降低了 STW 时间。

3.2.1 三色标记算法

Go 的 GC 是如何实现并行的呢？其中的关键之一在于三色标记清除算法。



颜色标记的意义

* 黑色：表示对象已经被扫描，并且所有引用的对象也已经被标记，黑色对象不会被回收。
* 灰色：表示对象已经被发现并且被标记，但其引用的对象还没有被扫描，灰色对象需要进一步处理。
* 白色：表示对象还没有被标记，如果在垃圾回收结束时，对象仍然是白色的，那么它将被回收。

三色标记步骤

1. 初始化
* 所有对象开始时都被标记为白色。
* 根对象被标记为灰色，并放入一个待处理队列中。
2. 标记阶段

* 重复以下步骤，直到待处理队列为空：
+ 从待处理队列中取出一个灰色对象，并将其标记为黑色。
+ 遍历该对象的所有引用：如果被引用的对象是白色的，将其标记为灰色，并放入待处理队列中。
3. 清除阶段
* 遍历所有对象：
+ 如果对象是白色的，说明它是不可达的，可以被回收。
+ 如果对象是黑色的，说明它是可达的，保留不动。

3.2.2 屏障机制

为了解决并发 GC 下的漏标问题，提出了屏障机制，我们先来看看并发 GC 产生的问题。

并发 GC 产生的问题

由于 GC 引入了并发机制，三色标记可能会产生以下几个问题：

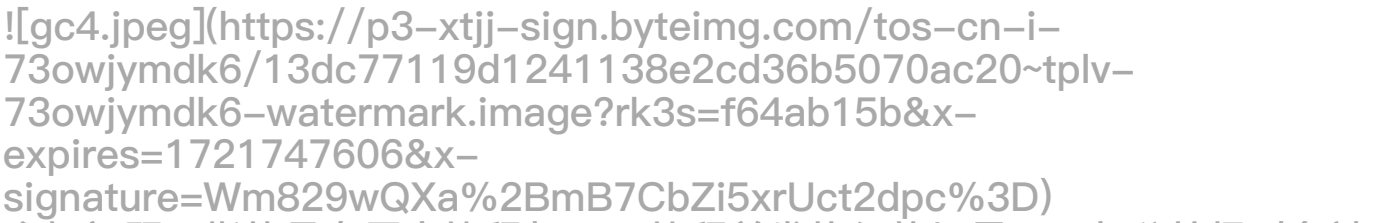
1. 漏标问题

漏标问题：指的是在用户协程与 GC 协程并发执行的场景下，部分存活对象未被正确标记，从而被误删的情况。如图所示：

* 初始时刻： C 持有 D 的引用；
* GC 协程：A 被 GC 扫描完成，不会再次扫描，标记为黑色（此时 C 为灰色，还未完成扫描，D 为白色）；
* 用户协程：A 建立对 D 的引用，C 删除对 D 的引用；
* GC 协程：开始扫描 C，C 被置为黑色，由于 C 删除了对 D 的引用，此时 D 未被扫描到，依然为白色，但其被 A 对象引用，不是该被回收的对象；
* GC 协程：标记完成，回收白色对象，D 被误删除。

漏标的情况不能容忍，因为删除了程序正在使用的对象，会造成程序异常。

2. 多标问题


多标问题：指的是在用户协程与 GC 协程并发执行的场景下，部分垃圾对象被误标记，从而导致 GC 未按时将其回收的问题。如图所示：

* 初始时刻：A 持有 B 的引用，B 持有 C 的引用
* GC 协程：A 被扫描完成，置为黑色；B 因为被 A 引用，所以被置灰，等待扫描
* 用户协程：A 删除对 B 的引用，B 成为垃圾，然而 B 已经被置灰，最终会被 GC 标记为黑色
* GC 协程：B 被标记为黑色，标记阶段完成，进行清除，B 因为被误标记为黑色，因此不会被 GC 清除。

多标问题对比于漏标问题而言，是相对可以接受的。因误标记侥幸存活的对象被称为“浮动垃圾”，在下一轮 GC 就会被正确回收，因此错误可以得到弥补。

强弱三色不变式

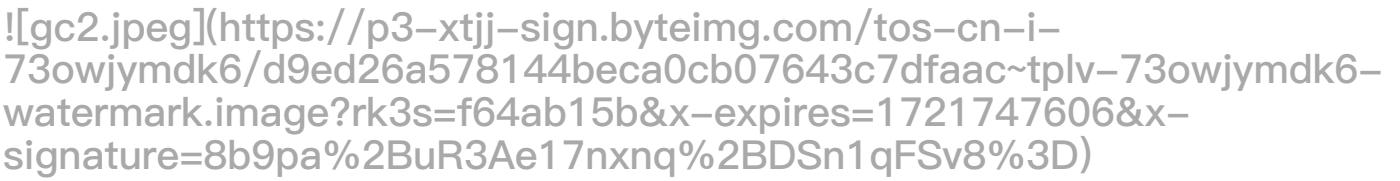
漏标问题的本质：一个已经扫描完成的黑对象，指向了一个被 灰\白 对象删除引用的白色对象，且该白色对象上游没有灰色对象。
因此，只要破坏了漏标问题的本质，就可以阻止漏标问题的发生，而破坏本质的理论被称为强弱三色不变式。

* 强三色不变式：不允许黑色对象引用白色对象（直接破坏黑色引用白色对象）
* 弱三色不变式：黑色对象可以引用白色对象，但白色对象上游必须有灰色对象（保障白色对象还有机会被 GC 扫描标记）

插入写屏障


插入写屏障：规则维护了强三色不变式，保证当一个黑色对象指向一个白色对象前，会先触发屏障，将白色对象置为灰色，再建立引用。

删除写屏障



删除写屏障：实现了弱三色不变式，保证当一个白色对象即将被上游删除引用前，会触发屏障将其置灰，之后再删除上游指向其的引用。

但是引入删除写屏障，有一个弊端，就是一个白色对象的引用被删除后，置灰，即使没有其他存活的对象引用它，它仍然会活到下一轮。如此一来，会产生很多的冗余扫描成本，且降低了回收精度。

3.2.3 Go 1.5 版本 GC 的弊端

在 Go 1.5 版本中，为了确保垃圾收集的正确性，需要在一次正常的三色标记流程结束后进行一次 STW 操作，并重新扫描栈上的对象。这是因为在并发标记阶段，可能存在一些特殊情况导致需要重新扫描栈上的对象。这些特殊情况包括：

1. 写屏障的使用：在并发标记阶段，程序的正常执行和垃圾收集是并发进行的。为了确保并发标记的正确性，Go 使用了写屏障（Write Barrier）。写屏障的作用是当程序修改某个对象的引用时，会通知垃圾收集器，以确保被引用的对象不会被错误地标记为垃圾。然而，写屏障并不能完全避免一些特殊情况下的遗漏 - 栈内存中变量的引用变化。
2. 栈对象的变化：在并发标记阶段，程序的栈可能会发生变化，例如函数调用、新变量的分配等。这些变化可能导致一些对象在标记阶段被遗漏，特别是那些在并发标记开始后才被栈引用的对象。
3. 重新扫描确保准确性：为了确保所有栈上的对象都被正确标记，并且没有遗漏任何存活对象，Go 1.5 中的垃圾收集器在并发标记结束后，会进行一次 STW 操作，并重新扫描栈上的对象。这一步骤可以确保在并发标记阶段没有被标记到的对象能够在最终阶段被正确标记。

通过在并发标记结束后进行一次 STW 操作并重新扫描栈上的对象，Go 语言的垃圾收集器可以确保准确性和完整性，避免因并发标记的复杂性导致的错误。

阅读到这里是不是有一个疑问：既然屏障机制可以解决并发 GC 下的漏标问题，为什么还要 re-scan 栈上对象？答：屏障机制只对堆上的对象有用，栈上对象不使用屏障机制。

换句话说：写屏障主要用于跟踪从堆到堆、从栈到堆、以及从全局变量到堆的引用变化，从栈引用栈对象时，不会触发写屏障。

为什么栈内存不能使用屏障机制?

1. 栈内存管理简单：栈内存的管理相对简单，不需要复杂的垃圾回收机制。栈上的变量在函数返回时自动回收，不需要像堆内存那样进行标记和清除。
2. 栈内存分配和回收频繁：栈上的对象通常是局部变量，生命周期较短，并且在函数调用和返回时会频繁地分配和释放。这使得栈上的对象变化非常频繁和复杂，难以通过屏障机制来精确跟踪。
3. 性能开销：屏障机制会增加额外的性能开销，尤其是在高频率的读写操作中。栈内存的访问频繁，使用屏障机制会显著降低性能。
4. 内存一致性：栈内存的生命周期短且结构简单，不太可能出现内存一致性问题。堆内存中的对象引用变化较多，因此需要屏障机制来确保一致性。

3.3 Go 1.8 并发三色标记法 + 混合写屏障机制

在 Go 1.8 版本之前，栈上的对象在并发标记阶段可能会发生变化，导致需要在标记结束后进行一次 STW 操作并重新扫描栈上的对象，以确保所有存活的对象都被正确标记。然而，这个过程会增加垃圾收集的暂停时间，影响程序的性能。

在 Go 1.8 版本后，引入了混合写屏障机制（hybrid write barrier），避免了对栈 re-scan 的过程，极大的减少了 STW 的时间。混合写屏障的精度虽然比插入写屏障稍低，但它完全消除了对栈的 STW 重新扫描，从而进一步减少了 STW 的时间。

Go 1.8 版本后的 GC 混合写屏障有以下四个关键步骤：

1. GC 刚开始的时候，会将栈上的可达对象全部标记为黑色。（在扫描栈帧的过程中，栈上的局部变量会被标记为黑色，因为它们被视为已处理的根对象，不需要再重新扫描）
2. GC 期间，任何在栈上新创建的对象，均为黑色（避免在扫描过程中，重复扫描这些对象）。
3. 堆上被删除的对象标记为灰色（删除写屏障）。
4. 堆上新添加的对象标记为灰色（插入写屏障）。

在标记阶段一开始，所有栈上的对象都会被直接标记为黑色，后续任何在栈上新创建的对象，均为黑色。黑色对象表示已经访问且不需要再扫描其引用，由于栈上的对象已经标记为黑色，不再需要在标记结束后重新扫描栈上的对象，从而减少了 STW 的时间。

4.一次 GC 周期的详细过程

一次 GC 周期分为大致 3 个重要阶段：初始 STW 事件阶段，并发标记阶段，并发清除阶段。接下来，咱们就一起聊一聊一次 GC 周期执行的详细过程。

4.1 初始 STW 事件阶段

4.1.1 暂停所有 Goroutines（STW）

为了确保内存状态一致，垃圾回收器会暂停所有正在运行的 Goroutines，对于 Go 程序来说，就相当于停止了整个世界（STW），也就是我们常说的卡顿，所以这个时间必须非常短，才不会对程序造成太大影响；比如一个 web 服务，如果 STW 时间比较长，就会造成超时等异常现象。

目的：暂停所有 Goroutines 后，内存状态到达一致性，是开启混合写屏障机制的前提，所以暂停所有 Goroutines 是必要的。

4.1.2 开启混合写屏障机制

开启混合写屏障机制：在初始 STW 事件阶段暂停所有 Goroutines 的同时，垃圾回收器会开启混合写屏障机制。（注意：混合写屏障机制主要是针对堆内存中的对象引用修改而设计的，它的主要目的是确保在垃圾回收过程中，任何新的对象引用修改都能被正确记录和处理）

为什么现在开始混合写屏障机制：暂停所有 Goroutines 后，内存状态到达一致性，一旦 Goroutines 恢复执行，内存状态一致性会被打破，无法得以保证；所以暂停所有 Goroutines 的主要目的是为了顺利开启混合写屏障机制，从而保障后续的内存状态一致性，这是确保垃圾回收过程有效和准确的关键步骤。

混合写屏障机制的作用：记录对象引用的修改，混合写屏障机制在开启后

，任何新的对象引用修改都会被记录下来。

* 确保了在分批扫描 Goroutines 栈帧时，即使其他 Goroutines 被恢复运行并修改对象引用，这些修改也能被正确记录和处理。确保了在并发标记阶段，所有引用的修改会被正确记录和处理。
* 当对象引用被修改时，混合写屏障会记录这些修改，将新引用的对象标记为灰色（堆上被删除的对象标记为灰色，堆上新添加的对象标记为灰色）。

4.1.3 分批扫描 Goroutines 栈帧，标记栈内局部变量

分批扫描栈帧：垃圾回收器会分批扫描所有被暂停的 Goroutines 的栈帧。每个 Goroutine 的栈扫描完毕后，该 Goroutine 会被立即恢复运行，而不是等待所有 Goroutines 全部扫描完成，也就是说扫描栈帧和程序是并行的，目的是降低 stw 时间，在扫描栈帧的时候，部分 Goroutines 就已经被放开运行了，在用户感知上，stw 已经结束了。

标记栈局部变量为黑色：在扫描栈帧的过程中，栈上的局部变量会被标记为黑色，因为它们被视为已处理的根对象（**GC 刚开始的时候，会将栈上的可达对象全部标记为黑色**）。这样栈对象就不需要再次进行扫描了，因为已经是黑色了。

标记引用的堆对象为灰色：在标记栈局部变量为黑色的同时，垃圾回收器会将这些局部变量引用的堆对象标记为灰色，并将它们加入到待处理队列中，以便在后续的并发标记阶段处理。

当某个 Goroutine 被扫描完成，会立即恢复该 Goroutine 的执行，此时在栈上新创建的对象会被标记为黑色（**GC期间，任何在栈上新创建的对象，均为黑色**）。**标记为黑色的目的**：在垃圾回收期间，任何在栈上新创建的对象被标记为黑色的主要目的是为了避免在扫描过程中重复扫描这些对象。由于栈上的局部变量在函数调用结束时会被自动销毁，因此它们不需要被垃圾回收器管理。将栈上对象标记为黑色，可以避免在垃圾回收的标记阶段对这些对象进行重复扫描，提高垃圾回收的效率，减少垃圾回收器的工作量，优化性能。

4.1.4 标记其他根对象

标记其他根对象为灰色：除了栈局部变量，垃圾回收器还会标记其他根对象（如全局变量和寄存器中的变量）为灰色，并将它们加入到待处理队列中。

* 全局变量：遍历所有全局变量和静态变量，将它们标记为灰色。

* 寄存器中的变量：遍历当前 CPU 寄存器中的变量，将它们标记为灰色。

为什么不和栈中局部变量一样直接标记为黑色：这个问题涉及到垃圾回收器的工作原理和优化策略。标记其他根对象（如全局变量和寄存器中的变量）为灰色，而不是像栈局部变量那样标记为黑色，是为了更高效地处理对象引用，确保垃圾回收过程的准确性和效率。详细解释如下：

* **递归处理引用**：灰色对象需要进一步处理：灰色对象表示它们已经被发现并被标记，但其引用的对象还没有被扫描。将全局变量和寄存器中的变量标记为灰色，可以确保这些对象引用的所有对象都会在后续的标记过程中被正确处理。递归标记：在并发标记阶段，垃圾回收器会递归处理灰色对象，将其引用的对象标记为灰色，并将其本身标记为黑色。这确保了所有可达对象都能被正确标记。
* **分阶段处理**：通过将全局变量和寄存器中的变量标记为灰色，可以将标记工作分阶段进行。在初始 STW 事件阶段，只需标记这些根对象为灰色，而在并发标记阶段再递归处理它们引用的对象。这种分阶段处理有助于减少初始 STW 事件的暂停时间。因为标记为灰色的对象只需被发现并放入待处理队列，而不需要立即递归扫描其引用的对象。

4.2 并发标记阶段

并发标记阶段是 Go 垃圾回收过程中的一个关键阶段，在此阶段，垃圾回收器和应用程序并行运行。

4.2.1 并发运行

并发运行：在初始 STW 事件阶段完成后，垃圾回收器进入并发标记阶段，此时应用程序和垃圾回收器并行运行。这意味着应用程序可以继续执行其正常操作，而垃圾回收器在后台执行标记操作。

4.2.2 处理灰色对象

处理灰色对象：垃圾回收器从待处理队列中取出灰色对象，标记它们引用的对象为灰色，并将它们本身标记为黑色，表示它们已经被扫描并且不会被回收。

递归处理灰色对象：垃圾回收器递归处理灰色对象，将其引用的对象标记为灰色，并将它们本身标记为黑色。这个过程一直持续到所有灰色对象都被处

理完毕。对于每个灰色对象，垃圾回收器会递归地标记其引用的所有对象，直到没有新的灰色对象需要处理。

4.2.3 混合写屏障

记录对象引用的修改：在并发标记阶段期间，当对象引用被修改时，混合写屏障会记录这些修改，将新引用的对象标记为灰色（**堆上被删除的对象标记为灰色，堆上新添加的对象标记为灰色**），将新引用的对象标记为灰色，以便在标记过程中处理。

确保内存状态一致性：混合写屏障机制确保了在并发标记阶段，任何新的对象引用修改都能被正确记录和处理，保持内存状态的一致性。

4.2.4 标记完成 STW 事件

在并发标记阶段，当垃圾回收器确定所有的灰色对象都已经被处理完毕时，会触发标记完成 STW

* **暂停所有 Goroutines**：在并发标记阶段结束时，垃圾回收器会再次暂停所有 Goroutines，以确保所有标记操作都已完成。暂停所有 Goroutines 可以防止在标记完成阶段发生新的引用修改，确保内存状态的一致性。
* **处理剩余的写屏障记录**：垃圾回收器需要处理所有剩余的写屏障记录，屏障记录和标记是并发的，因此存在处理延迟，可能存在部分屏障记录没有被处理，因此在标记完成 STW 事件阶段，需要特别处理所有剩余的屏障记录，以确保内存状态的一致性和正确性；确保所有引用修改都能被正确标记，处理剩余的写屏障记录，可以确保在并发标记阶段发生的所有引用修改都能被正确记录和处理。
* **关闭混合写屏障机制**：在确保所有引用修改都被正确处理后，垃圾回收器会关闭混合写屏障机制。关闭混合写屏障机制表示垃圾回收器不再需要记录引用修改，标记阶段已经完成。此时内存中只存在黑色和白色两种颜色对象，白色对象就是下一个阶段要清理的对象，因为不会再有引用指向白色对象，可以进行并行清理。
* **恢复所有 Goroutines**：标记完成后，垃圾回收器会立即恢复所有 Goroutines，应用程序继续正常运行。

4.3 并发清除阶段

并发清除阶段是垃圾回收过程中的最后一个主要阶段，在此阶段，垃圾回收器会清除所有没有被标记的对象。这一阶段的主要目的是释放内存，以便供应用

程序重新使用。

1. 开始并发清除阶段

在并发清除阶段，垃圾回收器和应用程序并行运行。应用程序继续执行其正常操作，而垃圾回收器在后台执行清除操作。

2. 清除未标记的对象

遍历堆内存：垃圾回收器会遍历整个堆内存，查找所有未被标记的白色对象。白色对象：在标记阶段结束时，任何未被标记为黑色或灰色的对象都是白色对象，这些对象被认为是不可达的，需要被清除。

释放内存：垃圾回收器会释放所有未被标记的白色对象的内存。内存释放：释放内存可以使这些内存区域重新可用，以便供应用程序分配新的对象。

3. 更新内存管理数据结构

更新空闲列表：在清除未标记的对象后，垃圾回收器会更新内存管理的数据结构，如空闲列表，以反映新的内存状态。空闲列表：空闲列表是一个数据结构，用于管理可用的内存块。更新空闲列表可以确保新的内存分配请求能够正确地使用已释放的内存。

4. 并发清除阶段的结束

清除完成：当垃圾回收器遍历完所有堆内存并清除所有未标记的对象后，并发清除阶段结束。

准备下一次垃圾回收：清除阶段完成后，垃圾回收器会准备下一次垃圾回收的必要数据和状态。并发清除阶段结束后，通常不会有专门的 STW 事件。然而，垃圾回收器可能会执行一些小的、短暂的暂停操作来进行必要的状态更新和清理工作，但这些操作通常非常快，不会对应用程序的性能产生显著影响。

结论
==

Go GC 的发展经历了三个重要的历史阶段：

1. Go 1.3 标记清除，实现简单，但整个 GC 过程都需要 STW，严重影响性能；后续优化为清除阶段不再需要 STW，提升部分性能；
2. Go 1.5 并发三色标记法 + 插入写屏障，引入并发垃圾回收器，实现 GC 与业务程序并行处理，可以完成并发的标记和清除，进一步减少 STW 时间，更好的利用多核处理器，提升 GC 性能；但在一次正常的三色标记流程结束后，需要进行一次 STW， re-scan 栈上对象，影响性能。
3. Go 1.8 三色标记 + 混合写屏障机制，消除了对栈本身的重新扫描，STW 的时间进一步得到缩减。

一次完整的 GC 分为三个阶段：初始 STW 事件、并发标记阶段、并发清除阶段；随着 Go 版本的不断更迭，GC 的各项细节问题也得到了优化，优化方向包括：

* 增量式改进：通过增量标记和清理，进一步减少了每次 STW 暂停的时间。
* 优化内存管理：通过基于页的内存管理和减少内存碎片，提升了内存分配和释放的效率。
* 高效的数据结构：使用更高效的数据结构来维护垃圾回收状态，减少了开销。

现代 Go 版本的垃圾收集器通过一系列优化措施，已经将 STW 时间减少到亚毫秒级别，极大地提高了程序的性能和响应性。未来版本可能会进一步优化垃圾收集器，以实现更短的 STW 时间。

原文链接: https://juejin.cn/post/7388842078798807049