

Golang GC 原理

作者: [happyue](#)

原文链接: <https://ld246.com/article/1586001874904>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

一、内存泄漏

内存泄露，是从操作系统的角度上来阐述的，形象的比喻就是“操作系统可提供给所有进程的存储空间(虚拟内存空间)正在被某个进程榨干”，导致的原因就是程序在运行的时候，会不断地动态开辟存储空间，这些存储空间在运行结束之后并没有被及时释放掉。应用程序在分配了某段内存之后由于设计的错误，会导致程序失去了对该段内存的控制，造成了内存空间的浪费。

如果程序在内存空间内申请了一块内存，之后程序运行结束之后，没有把这块内存空间释放掉，且对应的程序又没有很好的 `gc` 机制去对程序申请的空间进行回收，这样就会导致内存泄露。

二、GC 原理

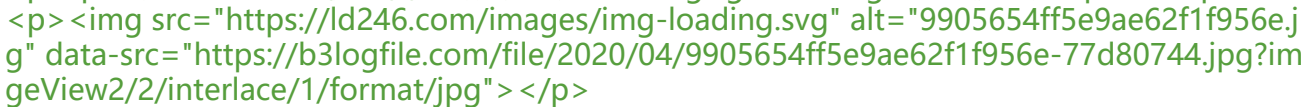
root

首先标记 root 根对象，根对象的子对象也是存活的。

根对象包括：全局变量，各个 G stack 上的变量等。

标记

span 是内存管理的最小单位，所以猜测 `gc` 的粒度也是 span。

 `gcmarkBits` 位图标记 span 的块是否被引用。对应内存分配中的 bitmap 区。

如图所示，通过 `gcmarkBits` 位图标记 span 的块是否被引用。对应内存分配中的 bitmap 区。

三色标记

灰色：对象已被标记，但这个对象包含的子对象未标记

黑色：对象已被标记，且这个对象包含的子对象也已标记，`gcmarkBits` 对应的位为 1（该对象会在本次 `GC` 中被清理）

白色：对象未被标记，`gcmarkBits` 对应的位为 0（该对象将会在本次 `GC` 中被清理）

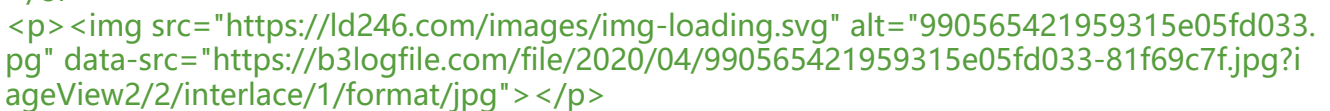
例如，当前内存中有 A~F 一共 6 个对象，根对象 a,b 本身为栈上分配的局部变量，根对象 a、b 分别引用了对象 A、B，而 B 对象又引用了对象 D，则 `GC` 开始前各对象的状态下图所示：

初始状态下所有对象都是白色的。

接着开始扫描根对象 a、b；由于根对象引用了对象 A、B，那么 A、B 变为灰色对象，接下来就开分析灰色对象，分析 A 时，A 没有引用其他对象很快就转入黑色，B 引用了 D，则 B 转入黑色的同还需要将 D 转为灰色，进行接下来的分析。

灰色对象只有 D，由于 D 没有引用其他对象，所以 D 转入黑色。标记过程结束

最终，黑色的对象会被保留下来，白色对象会被回收掉。

 `stop the world` 是 `gc` 的最大性能问题，对于 `gc` 而，需要停止所有的内存变化，即停止所有的 goroutine，等待 `gc` 结束之后才恢。

STW

`stop the world` 是 `gc` 的最大性能问题，对于 `gc` 而，需要停止所有的内存变化，即停止所有的 goroutine，等待 `gc` 结束之后才恢。

触发

阈值：默认内存扩大一倍，启动 `gc`

定期：默认 2min 触发一次 `gc`，`src/runtime/proc.go` `forcegcperiod`

手动：`runtime.GC()`

三、GC 过程



617759242.png" data-src="https://b3logfile.com/file/2020/04/55033320190331152306006167759242-16c704ac.png?imageView2/2/interlace/1/format/jpg"></p>

<p>GO 的 GC 是并行 GC, 也就是 GC 的大部分处理和普通的 go 代码是同时运行的, 这让 GO 的 GC 流程比较复杂.</p>

Stack scan: Collect pointers from globals and goroutine stacks. 收集根对象 (全局变量, G stack), 开启写屏障。全局变量、开启写屏障需要 STW, G stack 只需要停止该 G 就好, 时间比少。

Mark: Mark objects and follow pointers. 标记所有根对象, 和根对象可以到达的所有对象不被收。

Mark Termination: Rescan globals/changed stack, finish mark. 重新扫描全局变量, 和上一改变的 stack (写屏障), 完成标记工作。这个过程需要 STW。

Sweep: 按标记结果清扫 span

<p>目前整个 GC 流程会进行两次 STW(Stop The World), 第一次是 Stack scan 阶段, 第二次是 Mark Termination 阶段.</p>

第一次 STW 会准备根对象的扫描, 启动写屏障(Write Barrier)和辅助 GC(mutator assist).

第二次 STW 会重新扫描部分根对象, 禁用写屏障(Write Barrier)和辅助 GC(mutator assist).

<p>从 1.8 以后的 golang 将第一步的 stop the world 也取消了, 这又是一次优化; 1.9 开始, 写屏的实现使用了 Hybrid Write Barrier, 大幅减少了第二次 STW 的时间.</p>

<h3 id="写屏障">写屏障</h3>

<p>因为 go 支持并行 GC****, GC 的扫描和 go 代码可以同时运行, 这样带来的问题是 GC 扫描的过程中 go 代码有可能改变了对象的依赖树。</p>

<p>例如开始扫描时发现根对象 A 和 B, B 拥有 C 的指针。</p>

GC 先扫描 A, A 放入黑色

B 把 C 的指针交给 A

GC 再扫描 B, B 放入黑色

C 在白色, 会回收; 但是 A 其实引用了 C。

<p>为了避免这个问题, go 在 GC 的标记阶段会启用写屏障(Write Barrier).</p>

<p>启用了写屏障(Write Barrier)后, 在 GC 第三轮 rescan 阶段, 根据写屏障记将 C 放入灰色, 防止 C 丢失。</p>

<p>文章来源: https://www.cnblogs.com/linguoguo/p/10611657.html</p>