



# 23 | Pauseless GC : 挑战无暂停的垃圾回收

2021-12-20 海纳

《编程高手必学的内存知识》

[课程介绍 >](#)



讲述：海纳

时长 20:06 大小 18.41M



你好，我是海纳。

在前面的几节课程中，我们学习了 CMS、G1 等垃圾回收算法，这两类 GC 算法虽然一直在想办法降低 GC 时延，但它们仍然存在相当可观的停顿时间。

如何进一步降低 GC 的停顿时间，是当前垃圾回收算法领域研究的最热点话题之一。我们就来学习这类旨在减少 GC 停顿的垃圾回收算法，也就是**无暂停 GC** ( Pauseless GC )。由于 Hotspot 的巨大影响力和普及程度，以及它的代码最容易获得，我们这节课就以 ZGC 为例来深入讲解无暂停 GC。



而且，ZGC 对 Java 程序员的意义和 G1 是同样重要的。如果说 CMS 代表的是过去式，而 G1 是一种过渡（尽管这个过渡期会很长），那么 ZGC 无疑就是 JVM 自动内存管理器的

未来。

通过这节课的学习，你就能了解到无暂停 GC 的基本思想和可以使用的条件，从而为未来正确地使用无暂停 GC 做好充分的准备。

无暂停 GC 这个词你可能比较陌生，让你觉得这个算法很难，我们不妨先来了解一下它的前世今生，你就能知其然，经过后面对它原理的讲解，你就能知其所以然了。

## 无暂停 GC 简介

JVM 的核心开发者 Cliff Click 供职于 Azul Systems 公司期间，撰写了一篇很重要的论文，也就是 [Pauseless GC](#)，提出了无暂停 GC 的想法和架构设计。同时，Azul 公司也在他们的 JVM 产品 Zing 中实现了一个无暂停 GC，将 GC 的停顿时间大大减少，这就是 [C4 垃圾回收器](#)。

同时，Red hat 公司的 GC 研究小组也开启了一款名为 Shenandoah 的垃圾回收器，它的工作原理与 C4 不同，但它在停顿时间这一项上的表现也非常出色。人们把 Shenandoah GC 也归为无暂停 GC。

时隔多年，Oracle 公司也开发了一款面向低时延的垃圾回收器，它的基本思想和 C4 垃圾回收器的一致，并且也在 openjdk 社区开源。

了解了无暂停 GC 的历史后，我们再分别从功能原理和代码实现上来讨论无暂停 GC。从功能原理上看，**无暂停 GC 与 CMS、Scavenge 等传统算法不同，它的停顿时间不会随着堆大小的增加而线性增加**。以 ZGC 为例，它的最大停顿时间不超过 10ms，注意不是平均，也不是随机，而是最大不超过 10ms。是不是感到很震惊呢？这节课我们就一起揭开 ZGC 的神秘面纱，探究这极低时延背后的真相。

从代码实现上看，ZGC 很复杂，包含很多细节，整个 GC 周期甚至划分了十个不同的阶段。代码阅读起来也相当困难。不过不用担心，这节课重点介绍的不是 ZGC 的代码实现，而是 ZGC 背后的原理，当我们理解它的原理之后，再去探究实现细节，才会事半功倍。我们就先从刚才提到的那个问题，也就是它为什么可以做到最大 10ms 的停顿时间开始吧。

## ZGC 停顿时间的真相

ZGC 和 G1 有很多相似的地方，它的主体思想也是采用复制活跃对象的方式来回收内存。在回收策略上，它也同样将内存分成若干个区域，回收时也会选择性地先回收部分区域。

ZGC 与 G1 的区别在于：**它可以做到并发转移（拷贝）对象**。关于并发转移的概念，这里我还是提醒你一下，并发转移指的是在对象拷贝的过程中，应用线程和 GC 线程可以同时进行，这是其他 GC 算法目前没有办法做到的。

前面几节课中我们介绍的垃圾回收算法，在进行对象转移时都是需要“**世界停止**”（Stop The World, STW）的，而对象转移往往是垃圾回收过程最耗时的一个环节，并且随着堆的增大，这个时间也会跟着增加。ZGC 则不同，**在应用线程运行的同时，GC 线程也可以进行对象转移，这样就相当于把整个 GC 最耗时的环节放在应用线程后台默默执行，不需要一个长时间的 STW 来等待**。这也正是 ZGC 停顿时间很小的主要原因。

你可能会问，如何能在应用线程修改对象引用关系的同时，GC 线程还能正确地转移对象，或者说 GC 线程将对象转移的过程中，应用线程是如何访问正在被搬移的对象呢？接下来我就带你了解并发转移的关键技术。

## 并发转移关键技术

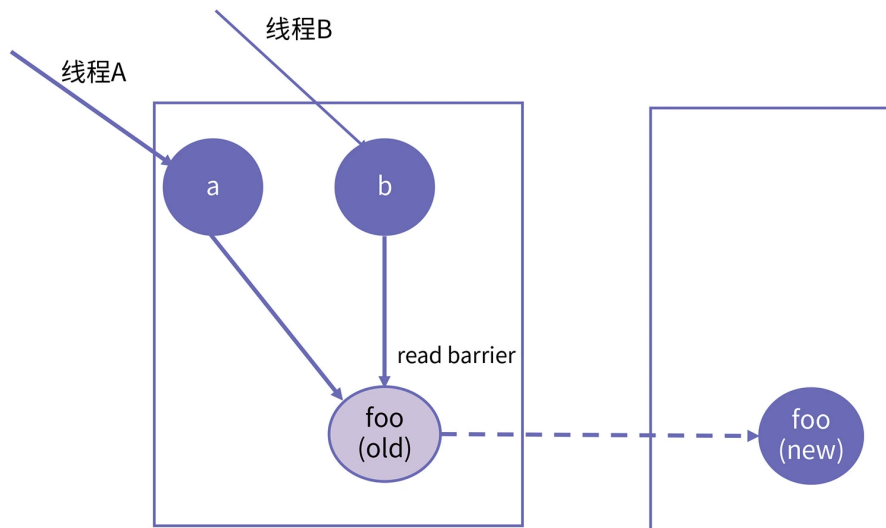
在此之前，我们首先回顾一下并发标记算法的原理。在并发标记的过程中，应用线程可能会修改对象之间的引用关系，为了保证在对象标记的过程中活跃对象不被漏标，我们引入了**三色标记算法**。虽然三色标记算法会在当前回收周期内产生浮动垃圾，但是不会漏标，而且多标记的垃圾对象也会在下一个回收周期被清理。

在介绍三色标记算法时，我们还讲到了 write barrier 概念。write barrier 主要是通过拦截写动作，在对象赋值时加入额外操作。这节课，我们就来讲解一个与 write barrier 对应的操作，它是无暂停 GC 算法中普遍采用的一个操作，那就是 read barrier，也就是在对象读取时加入额外操作。

## read barrier

通过前面的学习，我们知道 CMS 算法和 G1 算法都使用了 write barrier 来保证并发标记的完整性，防止漏标现象。ZGC 的并发标记也不例外，这个技术我们已经深入讨论过了，这里就不再啰嗦了。除此之外，ZGC 提升效率的核心关键在于并发转移阶段使用了 read barrier。

请你试想一下，当应用线程去读一个对象时，GC 线程刚好正在搬移这个对象。如果 GC 线程没有搬移完成，那么应用线程可以去读这个对象的旧地址；如果这个对象已经搬移完成，那么可以去读这个对象的新地址。那么判断这个对象是否搬移完成的动作就可以由 read barrier 来完成。



上图中，对象 a 和对象 b 都引用了对象 foo，当 foo 正在拷贝的过程中，应用线程 A 可以访问旧的对象 foo 得到正确的结果，当 foo 拷贝完成之后，应用线程 B 就可以通过 read barrier 来获取对象 foo 的新地址，然后直接访问对象 foo 的新地址。

请你思考一下，如果这里只用 write barrier 是否可行？当 foo 正在拷贝的过程中，应用线程 A 如果要写这个对象，那么只能在旧的对象 foo 上写，因为还没有搬移完成；如果当 foo 拷贝完成之后，应用线程 B 再去写对象 foo，是写到 foo 的新地址，还是旧地址呢？

如果写到旧地址，那么对象 foo 就白搬移了，如果写到新地址，那么又和线程 A 看到的内容不一样？所以使用 write barrier 是没有办法解决并发转移过程中，应用线程访问一致性问题，从而无法保证应用线程的正确性。因此，为了实现并发转移，ZGC 使用了 read barrier。

与此同时，我们还需要关注一个问题，就是在大多数的应用中，读操作要比写操作多一个数量级，所以 read barrier 对性能更加敏感（ZGC 最初的设计目标之一是吞吐量不低于





地址视图应该怎么理解呢？其实很简单，对于一个对象来说，如果它地址的第 42 位是 1，那么它就被认为是处于 Marked0 视图。依次类推，如果第 43 位是 1，这个对象就处于 Marked1 视图；如果第 44 位是 1，该对象就处于 Remapped 视图。

地址视图的巧妙之处就在于，**一个在物理内存上存放的对象，被映射在了三个虚拟地址上**。前面我们学习地址映射的时候知道，一个物理地址可以被映射到多个虚拟地址，这个映射方式在同一个进程内同样适用。例如下面的代码：

[复制代码](#)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <sys/mman.h>
5 #include <unistd.h>
6
7 #define PAGE_SIZE 4096
8
9 int main() {
10     int fd = memfd_create("anonymous", MFD_CLOEXEC);
11     ftruncate(fd, PAGE_SIZE);
12     char* shm0 = (char*)mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHAR
13     char* shm1 = (char*)mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHAR
14     char* shm2 = (char*)mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHAR
15     sprintf(shm0, "hello colored pointer");
16     printf("%s\n", shm1);
17     printf("%s\n", shm2);
18     sprintf(shm1, "wow!");
19     printf("%s\n", shm0);
20     printf("%s\n", shm2);
21     close(fd);
22     munmap(shm0, PAGE_SIZE);
23     munmap(shm1, PAGE_SIZE);
24     munmap(shm2, PAGE_SIZE);
25     return 0;
26 }
```

使用以下命令，编译并执行这个程序：

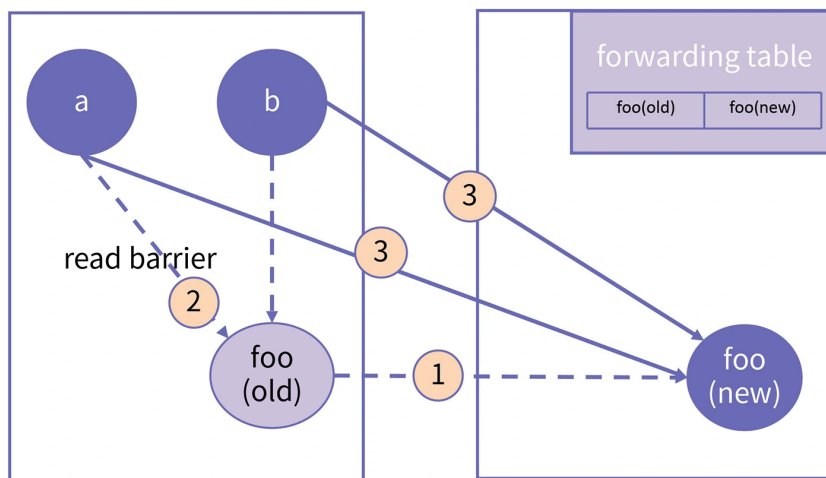
[复制代码](#)

```
1 $ gcc -Wall -D_GNU_SOURCE multi_mmap.c -o multi
2 $ ./multi
```

上面的例子先在内存中创建了一个匿名文件（第 10 行），然后将这个匿名文件映射到 shm0, shm1, shm2 三个虚拟地址上（第 12-14 行）。当我们修改 shm0 时，shm1 和 shm2 的内容也会跟着变化。地址视图也是用了同样的原理，三个地址视图映射的是同一块物理内存，映射地址的差异只在第 42-45 位上。这样一个对象可以由三个虚拟地址访问，其访问的内容是相同的。

有了地址视图之后，我们就可以在一个对象转移之后，修改它的地址视图了，同时还可以维护一张映射表（下称 forwarding table）。在这个映射表中，key 是旧地址，value 是新地址。当对象再次被访问时，通过插入的 read barrier 来判断对象是否被搬移过。如果 forwarding table 中有这个对象，说明当前访问的对象已经转移，read barrier 这时就会将对这个对象的引用直接更改为新地址。

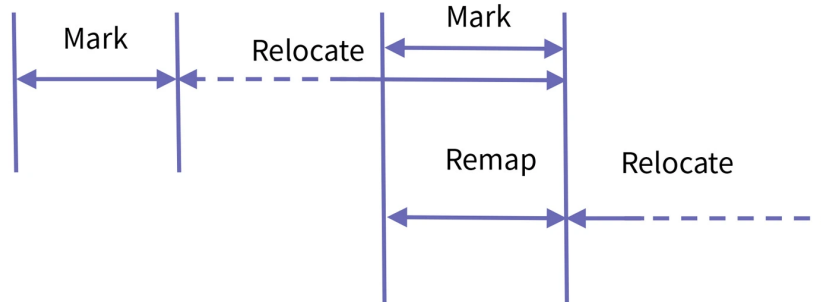
我还是举一个例子来说明，搬移一个对象以及访问它的引用所需要的步骤，如下图所示：



上图中，当 foo 对象发生转移之后，对象 a 再访问 foo 时就会触发 read barrier。read barrier 会查找 forwarding table 来确定对象是否发生了转移，确定 foo 被转移到新地址 foo ( new ) 之后，直接将这一次对 foo 的访问更改为 foo ( new )。由于整个过程是依托于 read barrier 自动完成的，这个过程也叫“自愈”。在介绍了 ZGC 的关键技术之后，我们来重点讲下 ZGC 的回收原理。

## ZGC 的回收原理

ZGC 虽然在实现上有十个左右的小步骤，但在总体思想上可以概括为三个核心步骤，我们通过 [Pauseless GC 原始论文](#) 的内容来介绍。



极客时间

在这张图中，你可以看到 Pauseless 的三个核心步骤分别是：**Mark**、**Relocate** 和 **Remap**。接下来我们就简单了解下这三个核心步骤都做了哪些事情。按照步骤的先后顺序，我们先来介绍 Mark。

### Mark

事实上，ZGC 也不是完全没有 STW 的。在进行初始标记时，它也需要进行短暂的 STW。不过在这个阶段，ZGC 只会扫描 root，之后的标记工作是并发的，所以整个初始标记阶段停顿时间很短。也正是因为这一点，ZGC 的最大停顿时间是可控的，也就是说**停顿时间不会随着堆的增大而增加**。

初始标记工作完成之后，就可以根据 root 集合进行并发标记了。前面我们提到的三个地址视图 Marked0、Marked1、Remapped 在这里就起了作用。

在 GC 开始之前，地址视图是 Remapped。那么在 Mark 阶段需要做的事情是，将遍历到的对象地址视图变成 Marked0，也就是修改地址的第 42 位为 1。前面我们讲过，三个地址视图映射的物理内存是相同的，所以修改地址视图不会影响对象的访问。



除此之外，应用线程在并发标记的过程中也会产生新的对象。类似于 G1 中的 SATB 机制，新分配的对象都认为是活的，它们地址视图也都标记为 Marked0。至此，所有标记为 Marked0 的对象都认为是活跃对象，活跃对象会被记录在一张活跃表中。

而视图仍旧是 Remapped 的对象，就认为是垃圾。接下来，我们进入 Relocate 阶段，也就是转移阶段。

## Relocate

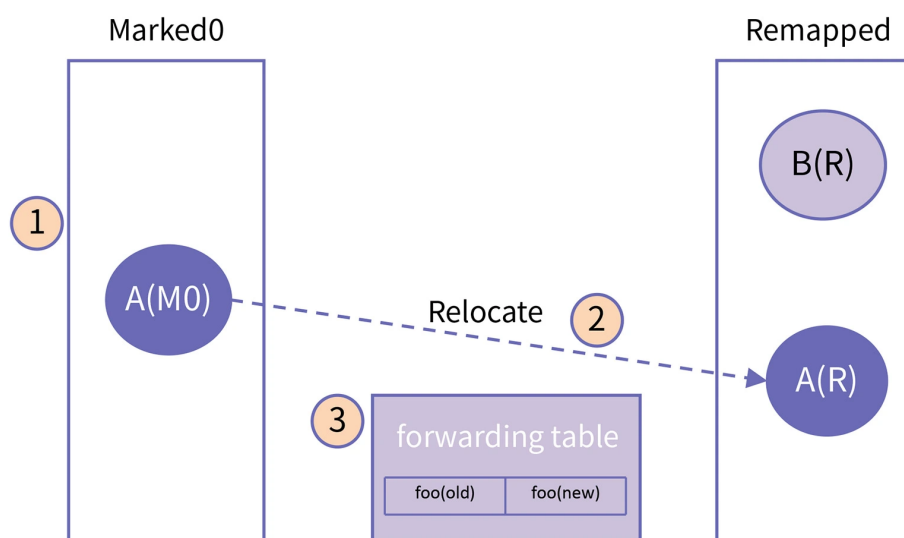
**Relocate 阶段的主要任务是搬移对象**，在经过 Mark 阶段之后，活跃对象的视图为 Marked0。搬移工作要做两件事情：

**选择一块区域，将其中的活跃对象搬移到另一个区域；**

**将搬移的对象放到 forwarding table。**

关于第一点，我们前面提到 ZGC 是分块的，块区域叫 Page；G1 也是分块的，只不过被分成的块叫 Region。虽然细节上有些差异，但它们总体的思想是类似的。

至于 forwarding table，我们在前面也提到过，它是一张维护对象搬移前和搬移后地址的映射表，key 是对象的旧地址，value 是对象的新地址。



在 Relocate 阶段，应用线程新创建的对象地址视图标记为 Remapped。如果应用线程访问到一个地址视图是 Marked0 的对象，说明这个对象还没有被转移，那么就需要将这个对象进行转移，转移之后再加入到 forwarding table，然后再对这个对象的引用直接指向新地址，完成自愈。这些动作都是发生在 read barrier 中的，是由应用线程完成的。

当 GC 线程遍历到一个对象，如果对象地址视图是 Marked0，就将其转移，同时将地址视图置为 Remapped，并加入到 forwarding table；如果访问到一个对象地址视图已经是 Remapped，就说明已经被转移了，也就不做处理了。

那么关于三个地址视图我们已经用到了其中两个，你一定好奇 Marked1 视图什么时候使用。接下来我们就进入 Remap 阶段，为你揭晓 Marked1 视图的作用。

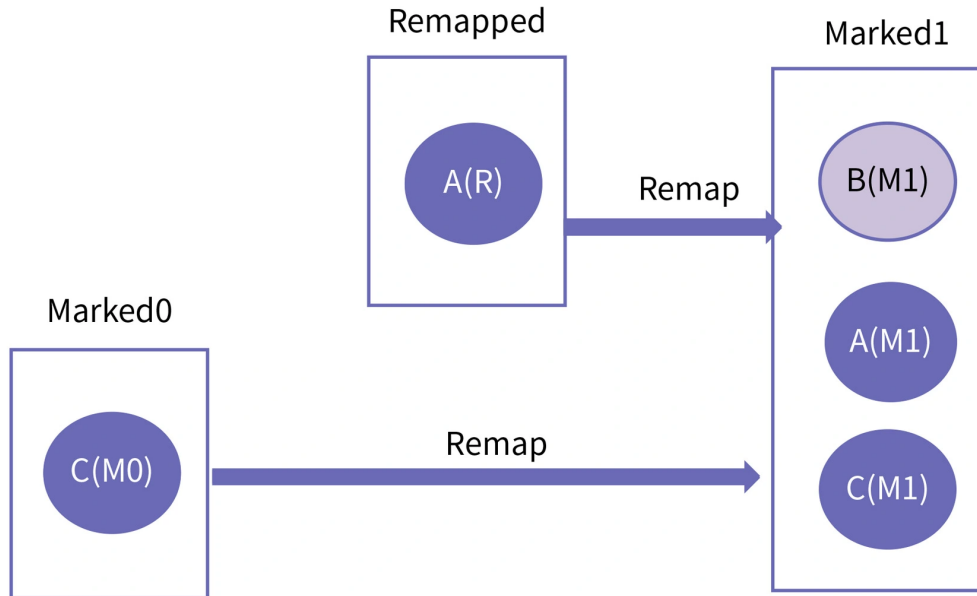
## Remap

**Remap 阶段主要是对地址视图和对象之间的引用关系做修正。**因为在 Relocate 阶段，GC 线程会将活跃对象快速搬移到新的区域，但是却不会同时修复对象之间的引用（请注意这一点，这是 ZGC 和以前我们遇到的所有基于 copy 的 GC 算法的最大不同）。这就导致还有大量的指针停留在 Marked0 视图。

这样就会导致活跃视图不统一，需要再对对象的引用关系做一次全面的调整，这个过程也是要遍历所有对象的。不过，因为 Mark 阶段也需要遍历所有对象，所以，可以把当前 GC 周期的 Remap 阶段和下一个 GC 周期的 Mark 阶段复用。

但是由于 Remap 阶段要处理上一轮的 Marked0 视图指针，又要同时标记下一轮的活跃对象，为了区分，可以再引入一个 Mark 标记，这就是 Marked1 标志。可以想象，Marked0 和 Marked1 视图在每一轮 GC 中是交替使用的。

在 Remap 阶段，新分配对象的地址视图是 Marked1，如果遇到对象地址视图是 Marked0 或者 Remapped，就把地址视图置为 Marked1。具体过程如下图所示：



这个过程结束以后，就完成了地址视图的调整，同时也完成了新一轮的 Mark。可以看到，Marked0 和 Marked1 其实是交替进行的，通过地址视图的切换，在应用线程运行的同时，默默就把活对象搬走了，把垃圾回收了。

好了，关于 ZGC 的回收原理我们就讲到这里。ZGC 的回收过程大致分为三个主要阶段，其中 **Mark 阶段负责标记活跃对象、Relocate 阶段负责活跃对象转移、ReMap 阶段负责地址视图统一**。因为 Remap 阶段也需要进行全局对象扫描，所以 Remap 和 Mark 阶段是重叠进行的。

## 总结

好啦，这节课到这里就结束啦。这节课，我们先介绍了无暂停 GC 的发展历史，然后介绍了无暂停回收算法的特点，那就是能够将垃圾回收的最大停顿时间控制在 10ms 以内，并且停顿时间不会随着堆的增大而线性增加。

我们选取了 openjdk 的 ZGC 作为举例，详细介绍了 ZGC 停顿时间的真相，同时也分析了 ZGC 的回收原理。ZGC 之所以能够做到这么低的停顿时间，是因为它的大部分工作都是并发执行的，其中也包括了垃圾回收过程中最耗时的对象转移阶段。

**ZGC 能够做到并发转移，背后有两大关键技术，分别是 read barrier 和 colored pointer。** read barrier 的作用在于应用线程可以在对象转移之后，通过 forwarding table 实现“自愈”。而 colored pointer 实现了地址视图，十分高效地完成了 read barrier 需要完成的工作，在实现并发转移的同时，保证吞吐率不出现大幅下降。

最后我们介绍了 ZGC 的回收原理，**整个回收过程可以大致分为 Mark、Relocate、Remap 三个阶段**，其中 Mark 和 Remap 阶段是可以重叠的。

GC 开始时，地址视图为 Remapped，Mark 阶段的主要工作是**标记活跃对象，然后将地址视图向 Marked0 迁移**，处于 Marked0 的对象都被认为是活跃对象。

Relocate 阶段开始时，地址视图为 Marked0，该阶段主要做**对象搬移工作，将地址视图向 Remapped 迁移**。应用线程如果访问一个已经被转移的对象，就会触发 read barrier，完成“自愈”，最终访问的是 Remapped 视图的新对象。

而 Remap 阶段是**地址视图的修复阶段**，在 Remap 阶段开始时，地址视图为 Remapped。Remap 阶段的功能是做**地址视图统一**，对于仍处于 Marked0 和 Remapped 视图的活跃对象，将其地址视图更新为 Marked1。当然也可以是对于仍处于 Marked1 和 Remapped 视图的活跃对象，将其地址视图更新为 Marked0。Remap 和 Mark 阶段交替进行，交替操作 Marked0 和 Marked1 视图。

通过地址视图的切换以及使用 read barrier 完成对象“自愈”过程，使得 ZGC 能够高效、准确的完成并发转移，大大降低了垃圾回收过程中的停顿时间，以至于达到无暂停 GC 的效果。

好啦，以上就是无暂停垃圾回收算法的核心内容了。

## 思考题

请你思考一下：ZGC 在对象转移之后旧对象原来占用的内存空间是否可以重复利用？请你结合 colored pointer 的功能思考。一点提示：可以思考一下为什么不使用 forwarding 指针技术，而要使用 forwarding table 呢？欢迎在留言区分享你的想法，我在留言区等你。


## 吊打面试官

- ZGC是零时延的GC算法吗？简单介绍下他背后的原理？

ZGC也不是完全没有STW的GC算法，整个GC周期中也会进行少量的停顿，只不过停顿的时间可控制在10ms以内，而GC过程最耗时的对象拷贝操作是并发执行的，所以它能做到极低时延。

而做到并发转移的背后又有两大关键技术支持，分别是read barrier和colored pointer，read barrier 保证了并发转移过程中，应用线程访问转移后的对象“自愈”。colored pointer则是通过创建地址视图，巧妙的解决了GC线程和应用线程访问对象时的相互影响，正是这两个技术的存在，并发转移才得以实现，也是ZGC背后的核心原理。

高频面试真题

 极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | G1 GC：分区回收算法说的是什么？

下一篇 24 | GC实例：Python和Go的内存管理机制是怎样的？



## 训练营推荐

# Java 学习包免费领 NEW

面试题答案均由大厂工程师整理

阿里、美团等  
大厂真题

18 大知识点  
专项练习

大厂面试  
流程解析

可复用的  
面试方法

面试前  
要做的准备

### 精选留言 (3)

写留言



费城的二鹏

2021-12-20

使用 forwarding table 的好处是，在remap前就可以释放已经被迁移的 page，只要保留 forwarding 即可，减少内存占用。

而如果使用 forwarding point 则需要一直保留这个page 直到重映射完成。

另一方面是性能考虑，提升了吞吐量

展开 v

作者回复: 厉害！言简意赅。



1



一子三木

2021-12-20

由于remap 会等待下一次mark，这里的下一次mark是下一次垃圾回收吗？如果是那假如很久不触发回收，那之前标记的对象都还是通过forwarding table获取了哦？

共 1 条评论 >



费城的二鹏

2021-12-20

我想问一个关于染色指针的问题。

假如有 a, b, c 三个对象。a 和 b 均引用 c, 在扫描过程中, 对指针做标记, 是在 a 和 b 对象存储的引用上加标记吗? 还是一个公共空间加标记? 如何确保修改了 a 引用 c 的地址后, b 也可以做修改? ...

展开 ∨

共 1 条评论 >

