一文读懂 Java 11 的 ZGC 为何如此高效

Java 11 的新功能已经完全冻结，其中有些功能绝对非常令人兴奋，本文着重介绍 ZGC。

Java 11 包含一个全新的垃圾收集器--ZGC，它由 Oracle 开发，承诺在数 TB 的堆上具有非常低的暂停时间。 在本文中，我们将介绍开发新 GC 的动机，技术概述以及由 ZGC 开启的一些可能性。

那么为什么需要新 GC 呢？毕竟 Java 10 已经有四种发布多年的垃圾收集器，并且几乎都是无限可调的。 换个角度看，G1 是 2006 年时引入 Hotspot VM 的。当时最大的 AWS 实例有 1 vCPU 和 1.7GB 内存，而今天 AWS 很乐意租给你一个 x1e.32xlarge 实例，该类型实例有 128 个 vCPU 和 3,904GB 内存。 ZGC 的设计目标是：支持 TB 级内存容量，暂停时间低（<10ms），对整个程序吞吐量的影响小于 15%。 将来还可以扩展实现机制，以支持不少令人兴奋的功能，例如多层堆（即热对象置于 DRAM 和冷对象置于 NVMe 闪存），或压缩堆。

GC 术语

为了理解 ZGC 如何匹配现有收集器，以及如何实现新 GC，我们需要先了解一些术语。最基本的垃圾收集涉及识别不再使用的内存并使其可重用。现代收集器在几个阶段进行这一过程，对于这些阶段我们往往有如下描述：

- 并行- 在 JVM 运行时，同时存在应用程序线程和垃圾收集器线程。 并行阶段是由多个 gc 线程执行，即 gc 工作在它们之间分配。 不涉及 GC 线程是否需要暂停应用程序线程。
- 串行- 串行阶段仅在单个 gc 线程上执行。与之前一样，它也没有说明 GC 线程是否需要暂停应用程序线程。
- STW - STW 阶段，应用程序线程被暂停，以便 gc 执行其工作。 当应用程序因为 GC 暂停时，这通常是由于 Stop The World 阶段。
- 并发 -如果一个阶段是并发的，那么 GC 线程可以和应用程序线程同时进行。 并发阶段很复杂，因为它们需要在阶段完成之前处理可能使工作无效（译者注：因为是并发进行的，GC 线程在完成一阶段的同时，应用线程也在工作产生操作内存，所以需要额外处理）的应用程序线程。
- 增量 -如果一个阶段是增量的，那么它可以运行一段时间之后由于某些条件提前终止，例如需要执行更高优先级的 gc 阶段，同时仍然完成生产性工作。 增量阶段与需要完全完成的阶段形成鲜明对比。

权衡

值得指出的是，所有这些属性都需要权衡利弊。 例如，并行阶段将利用多个 gc 线程来执行工作，但这样做会导致线程协调的开销。 同样，并发阶段不会暂停应用程序线程，但可能涉及更多的开销和复杂性，才能同时处理使其工作无效的应用程序线程。

ZGC

现在我们了解了不同 gc 阶段的属性，让我们继续探讨 ZGC 的工作原理。 为了实现其目标，ZGC 给 Hotspot Garbage Collectors 增加了两种新技术：着色指针和读屏障。

着色指针

着色指针是一种将信息存储在指针（或使用 Java 术语引用）中的技术。因为在 64 位平台上（ZGC 仅支持 64 位平台），指针可以处理更多的内存，因此可以

使用一些位来存储状态。 ZGC 将限制最大支持 4Tb 堆（42-bits），那么会剩下 22 位可用，它目前使用了 4 位： finalizable， remap， mark0 和 mark1。我们稍后解释它们的用途。

着色指针的一个问题是，当您需要取消着色时，它需要额外的工作（因为需要屏蔽信息位）。 像 SPARC 这样的平台有内置硬件支持指针屏蔽所以不是问题，而对于 x86 平台来说，ZGC 团队使用了简洁的多重映射技巧。

多重映射

要了解多重映射的工作原理，我们需要简要解释虚拟内存和物理内存之间的区别。 物理内存是系统可用的实际内存，通常是安装的 DRAM 芯片的容量。 虚拟内存是抽象的，这意味着应用程序对（通常是隔离的）物理内存有自己的视图。操作系统负责维护虚拟内存和物理内存范围之间的映射，它通过使用页表和处理器的内存管理单元（MMU）和转换查找缓冲器（TLB）来实现这一点，后者转换应用程序请求的地址。

多重映射涉及将不同范围的虚拟内存映射到同一物理内存。 由于设计中只有一个 remap，mark0 和 mark1 在任何时间点都可以为 1，因此可以使用三个映射来完成此操作。 ZGC 源代码中有一个很好的图表可以说明这一点。

读屏障

读屏障是每当应用程序线程从堆加载引用时运行的代码片段（即访问对象上的非原生字段 non-primitive field）：

```
void printName( Person person ) {
String name = person.name; // 这里触发读屏障
// 因为需要从 heap 读取引用
//
```

```
System.out.println(name); // 这里没有直接触发读屏障
}
```

在上面的代码中，String name = person.name 访问了堆上的 person 引用，然后将引用加载到本地的 name 变量。此时触发读屏障。 Systemt.out 那行不会直接触发读屏障，因为没有来自堆的引用加载（name 是局部变量，因此没有从堆加载引用）。 但是 System 和 out，或者 println 内部可能会触发其他读屏障。

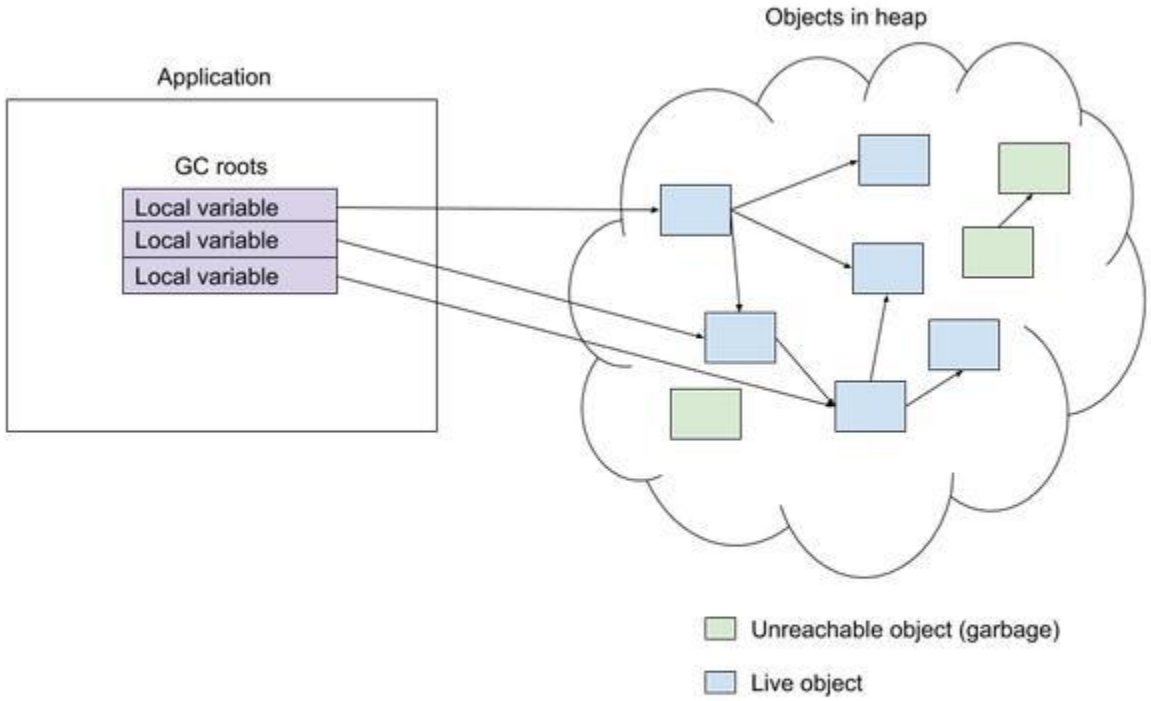
这与其他 GC 使用的写屏障形成对比，例如 G1。读屏障的工作是检查引用的状态，并在将引用（或者甚至是不同的引用）返回给应用程序之前执行一些工作。 在 ZGC 中，它通过测试加载的引用来执行此任务，以查看是否设置了某些位。 如果通过了测试，则不执行任何其他工作，如果失败，则在将引用返回给应用程序之前执行某些特定于阶段的任务。

标记

现在我们了解了这两种新技术是什么，让我们来看看 ZG 的 GC 循环。

GC 循环的第一部分是标记。标记包括查找和标记运行中的应用程序可以访问的所有堆对象，换句话说，查找不是垃圾的对象。

ZGC 的标记分为三个阶段。第一阶段是 STW，其中 GC roots 被标记为活对象。 GC roots 类似于局部变量，通过它可以访问堆上其他对象。 如果一个对象不能通过遍历从 roots 开始的对象图来访问，那么应用程序也就无法访问它，则该对象被认为是垃圾。从 roots 访问的对象集合称为 Live 集。GC roots 标记步骤非常短，因为 roots 的总数通常比较小。



该阶段完成后，应用程序恢复执行，ZGC 开始下一阶段，该阶段同时遍历对象图并标记所有可访问的对象。 在此阶段期间，读屏障针使用掩码测试所有已加载的引用，该掩码确定它们是否已标记或尚未标记，如果尚未标记引用，则将其添加到队列以进行标记。

在遍历完成之后，有一个最终的，时间很短的的 Stop The World 阶段，这个阶段处理一些边缘情况（我们现在将它忽略），该阶段完成之后标记阶段就完成了。

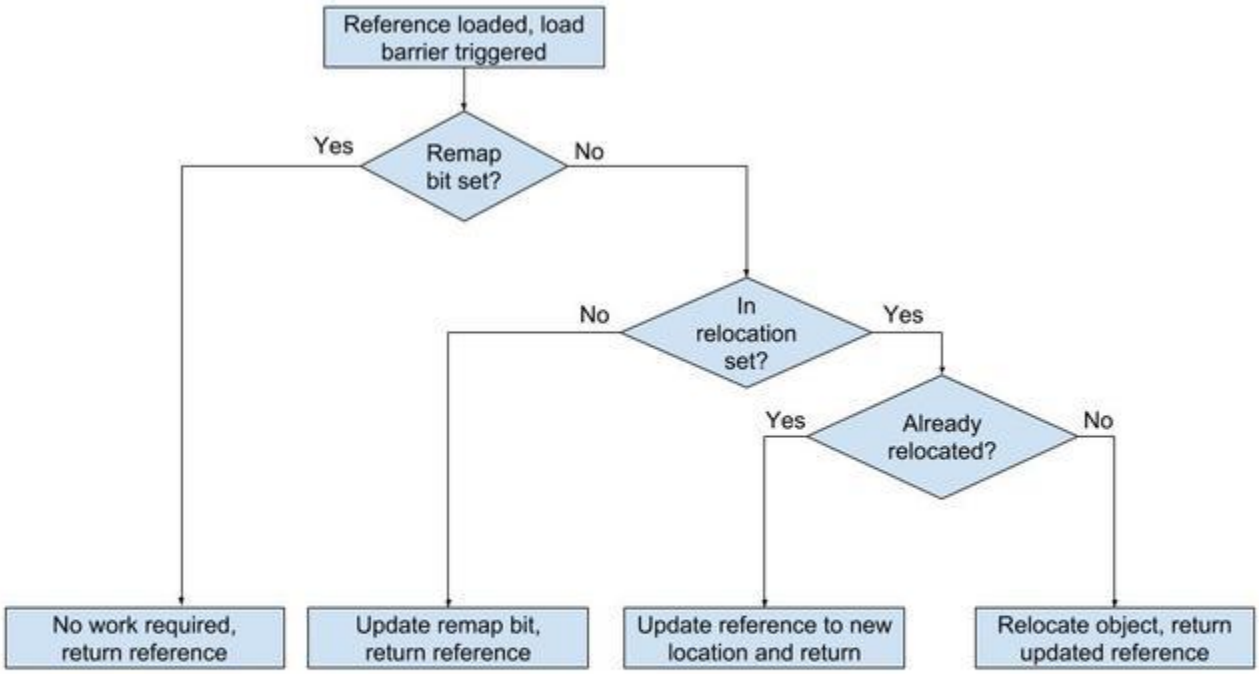
重定位

GC 循环的下一个主要部分是重定位。重定位涉及移动活动对象以释放部分堆内存。 为什么要移动对象而不是填补空隙？ 有些 GC 实际是这样做的，但是它导

致了一个不幸的后果，即分配内存变得更加昂贵，因为当需要分配内存时，内存分配器需要找到可以放置对象的空闲空间。相比之下，如果可以释放大块内存，那么分配内存就很简单，只需要将指针递增新对象所需的内存大小即可。

ZGC 将堆分成许多页面，在此阶段开始时，它同时选择一组需要重定位活动对象的页面。选择重定位集后，会出现一个 Stop The World 暂停，其中 ZGC 重定位该集合中 root 对象，并将他们的引用映射到新位置。与之前的 Stop The World 步骤一样，此处涉及的暂停时间仅取决于 root 的数量以及重定位集的大小与对象的总活动集的比率，这通常相当小。所以不像很多收集器那样，暂停时间随堆增加而增加。

移动 root 后，下一阶段是并发重定位。 在此阶段，GC 线程遍历重定位集并重新定位其包含的页中所有对象。 如果应用程序线程试图在 GC 重新定位对象之前加载它们，那么应用程序线程也可以重定位该对象，这可以通过读屏障（在从堆加载引用时触发）实现，如流程图如下所示：



这可确保应用程序看到的所有引用都已更新，并且应用程序不可能同时对重定位的对象进行操作。

GC 线程最终将对重定位集中的所有对象重定位，然而可能仍有引用指向这些对象的旧位置。 GC 可以遍历对象图并重新映射这些引用到新位置，但是这一步代价很高昂。 因此这一步与下一个标记阶段合并在一起。在下一个 GC 周期的标记阶段遍历对象对象图的时候，如果发现未重映射的引用，则将其重新映射，然后标记为活动状态。

概括

试图单独理解复杂垃圾收集器（如 ZGC）的性能特征是很困难的，但从前面的部分可以清楚地看出，我们所碰到的几乎所有暂停都只依赖于 GC roots 集合大小，而不是实时堆大小。标记阶段中处理标记终止的最后一次暂停是唯一的例外，但是它是增量的，如果超过 gc 时间预算，那么 GC 将恢复到并发标记，直到再次尝试。

性能

那 ZGC 到底表现如何？

Stefan Karlsson 和 Per Liden 在今年早些时候的 Jfokus 演讲中给出了一些数字。 ZGC 的 SPECjbb 2015 吞吐量与 Parallel GC（优化吞吐量）大致相当，但平均暂停时间为 1ms，最长为 4ms。 与之相比 G1 和 Parallel 有很多次超过 200ms 的 GC 停顿。

然而，垃圾收集器是复杂的软件，从基准测试结果可能无法推测出真实世界的性能。我们期待自己测试 ZGC，以了解它的性能如何因工作负载而异。

未来的可能性

着色指针和读屏障提供了一些有趣的可能。

多层堆和压缩

随着闪存和非易失性存储器变得越来越普遍，一种可能是 JVM 中允许多层堆，可以让很少使用的对象存储在较慢的存储层上。

该功能可以通过扩展指针元数据来实现，指针可以实现计数器位并使用该信息来决定是否需要移动对象到较慢的存储上。如果将来需要访问，则读屏障可以从存储中检索到对象。

或者对象可以以压缩形式保存在内存中，而不是将对象重定位到较慢的存储层。当请求时，可以通过读屏障将其解压并重新分配。

ZGC 的状态

在撰写本文时，ZGC 仍然是实验性的。

您可以使用 Java 11 Early Access 版本（ http://jdk.java.net/11/ ）进行测试，但值得指出的是，可能需要一段时间才能解决新版本中的所有问题。对于垃圾收集器来说，从 G1 发布到最终支持之间超过三年。

概要

随着拥有数百 GB 到数 TB RAM 的服务器变得越来越普及，Java 有效使用该规模堆的能力变得越来越重要。

ZGC 是个令人兴奋的新垃圾收集器，旨在为大堆提供非常低的暂停时间。 它通过使用着色指针和读屏障来实现这一点，这些是 Hotspot 新近开发的 GC 技术，并为未来增加了很多可能性。 ZGC 在 Java 11 中作为实验性的功能提供，现在可以使用 Early Access 版本试用。