JVM GC篇 — 垃圾收集器（下）

前言

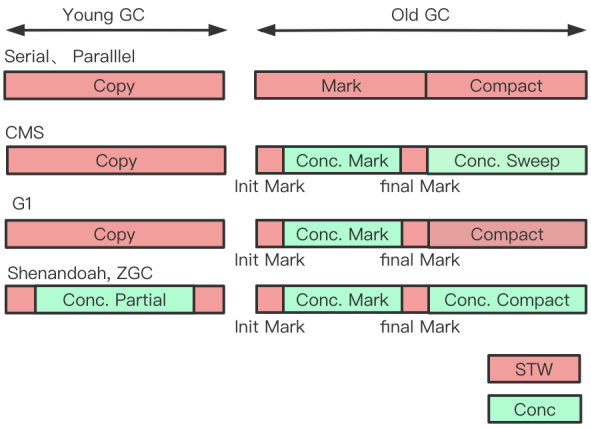
前面一小节中，我们梳理了经典的垃圾收集器。从单线程的Serial串行垃圾收集器开始，聊到了CMS最后到G1，并着重的分析了CMS和G1并发垃圾收集流程和部分实现原理。这些垃圾收集器经过数千台服务器的验证淬炼，已经变得相当的成熟可靠，但是真的就是完美的垃圾收集器吗？答案是否定的。今天我们一起来看看相较于“经典”垃圾收集器的全新一代垃圾收集器 Shenandoah 和 ZGC。这种垃圾收集器能够实现超大堆的容量下，轻松实现垃圾收集停顿不超过10ms的目标，但是目前还处于实验状态，官方命名为“低延迟垃圾收集器”（Low-Latency Garbage Collector 或者 Low-Pause-Time Garbage Collector）。

不可能的三角

我们一路走过来也看过很多的垃圾收集器，我们的垃圾收集器也在不断的进化，并发停顿时间稳定性各方面都变得越来越好，但是我们一直没有回答一个问题，什么的垃圾收集才是一个完美的垃圾收集器呢？其实衡量一个垃圾收集器我们一般有三个重要的指标：`内存占用（footprint）`、`吞吐量（Throughput）`和`延迟（Latency）`。这三者构成了一个**“不可能三角”**。这三个全部具备卓越表现的“完美”收集器是几乎不可能的，只要一款垃圾收集器能达到其中的两项，它就是一款优秀的垃圾收集器。

在内存占用、吞吐量和延迟这三项指标中，延迟的重要性日益凸显出来，其中GC发展过程中的后来者CMS和G1也是往这方向发展的产物。因为随着计算机硬件的发展、性能提升，我们越来越能容忍收集器多占用一些内存。硬件性能提升，对系统处理的能力也有直接的提升也就是吞吐量会提升。但是延迟不是这样的，随着内存的扩大，延迟的时间没有减少反而会带来负面的收益，由此不难看出延迟成为垃圾收集器的主要优化目标。

Serial 串行GC 和 Parallel 并行GC在进行垃圾回收的时候都必须全程需要STW，CMS和G1对于初始标记和最终标记阶段需要STW。G1虽然可以按照更小的粒度进行回收，从而抑制复制转移阶段的长时间停顿，但还是需要暂停。



Shenandoah

Shenandoah 是一款超低延迟垃圾收集器（Ultra-Low-pause-TIme Garbage Collector），其设计目标是管理大型的多核服务器上超大型堆内存，GC线程与应用线程并发执行使得暂停时间非常短暂。Shenandoah 是由RedHat公司独立发展的新型垃圾收集器，在2014年 RedHat 把Shenandoah贡献给了OpenJDK，并且推动它成为 OpenJDK 12的重要特性。但是这款垃圾收集器，也是**最孤独的一款垃圾收集器**。由于Shenandoah 不是由Oracle公司领导开发的HotSpot垃圾收集器，**不可避免的受到一些来自官方的排挤**。Oracle 明确拒绝在OracleJDK 12中支持Shenandoah收集器，因此Shenandoah是一款只有在 OpenJDK中才包含，而OracleJDK中反而不存在的垃圾收集器。

对G1的致敬

相比于有“正宗血统”的ZGC而言，Shenandoah更像是G1的继承者，它们都有着相似的内存布局，在初始标记、并发标记的处理思路都是高度的相同一致，甚至还有一些共享代码。它继承了G1的优点，反过来也促进G1打磨改进和bug修复，我们都知道G1当垃圾收集速度小于分配速度并且剩余堆空间不足时。G1会像CMS一样，并发失败并进行串行的FullGC来释放空间。而Shenandoah则促成了 Parallel Full GC for G1。

> Shenandoah继承了G1独特的设计思路，在G1原来的道路上又进行了拓展创新，同时反过来又推动了G1的打磨和bug修复，在我看来Shenandoah 理应成为 G1 的继承人，但是因为“血统不纯”而遭到排挤，属实有些无奈。

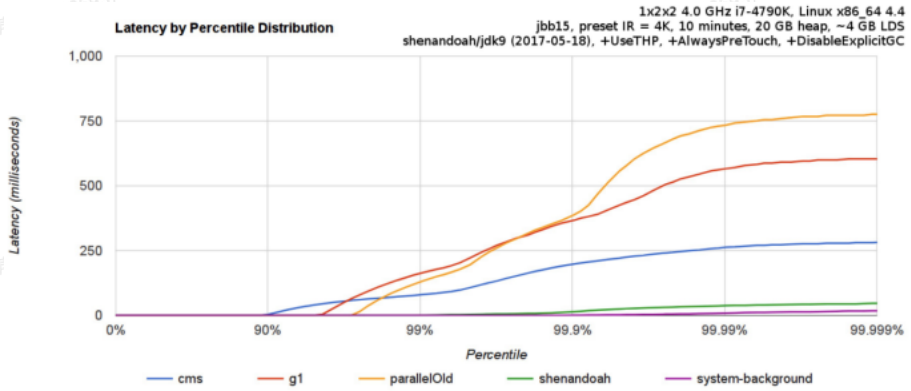
Shenandoah 的创新

Shenandoah 从 G1 上继承来很有优秀的特点，但是自身也有创新发展的点。主要集中但不仅限于下面三点：

- **支持并发整理**。前面我们梳理G1的垃圾收集流程，其中有初始标记（Init Mark）、最终标记（Final Mark）和最后的复制转移暂停（Evacuation Pause）阶段需要 STW，其他阶段都可以并发执行。Shenandoah 实现了并发的内存空间整理，具体细节我们在后面进行详细梳理。
- **不使用分代收集**。不会有专门的新生代的Region或老年代Region。在前面的小节中我们提到了**每次垃圾收集会统计每个Region的回收成本**，计算出平均值，标准差等，并通过最新的数据尽大地影响Region的回收收益。让统计出来的结果更加反应当前的Region的回收收益。整个堆内存有一个完善的回收评估，因此GC可以在不依赖分代理论的前提下，实现垃圾回收收益的最大化。
- **使用“连接矩阵”代替跨Region引用记忆集**。为了维护各个跨Region之间的引用关系，G1耗费了大量的内存和计算资源去维护记忆集，而在Shenandoah中，使用名为**“链接矩阵”**的数据结构来记录跨Region间的引用关系。降低了跨Region间引用记录消耗。“链接矩阵”可以理解为一张二维表格。如果Region N 有对象指向 Region M，就在表格的N行M列打上一个标记。例如下面的例子：**Region9指向Region6，Region7指向Region4，Region5指向Region3，Region3又指向Region1。**通过一个链接矩阵来代替各个Region自带的RSet来维护Region间的引用关系，极大的减少了内存消耗。

0	1	2	3	4	5	6	7	8	9
1									
2									
3	X								
4									
5			X						
6									
7				X					
8									
9						X			

得益于这些创新的设计，Shenandoah 相较于G1对系统的资源占用更小。因此在系统负载发生变化时，Shenandoah 的延迟相较于其他垃圾收集器也能保持在极度稳定的水平。

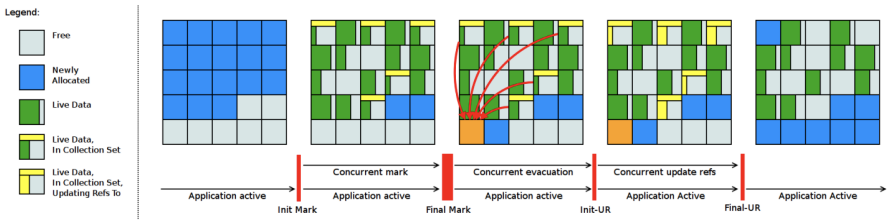


垃圾收集工作周期

Shenandoah 是并发的垃圾收集器，它的垃圾收集周期与CMS、G1类似，但是 Shenandoah 创新的支持并发整理，进一步提高了垃圾收集的并发度。以下是Shenandoah的一个工作周期的9个步骤：

- **初始标记阶段（Init Mark）**：与G1一样，为堆和应用程序准备并发标记，然后扫描根对象集。这是GC周期的第一次暂停，持续时间取决与根对象集的大小。因为根对象集很小，所以速度很快，暂停时间非常短。
- **并发标记阶段（Concurrent Mark）**：与G1一样，并发标记遍历堆，并跟踪可达的对象，该阶段与应用程序并发执行，持续时间取决于存活对象数据以及堆中对象图的结构。由于这个阶段并没有垃圾收集，且应用程序可以在此阶段自由分配新数据，因此并发标记阶段堆占用率会上升。
- **最终标记阶段（Final Mark）**：与G1一样，处理剩余的 SATB 扫描，并在这个阶段统计出回收价值高的Region，将这些Region构成一组回收集（Collection Set）。这个阶段也会有小阶段的暂停。
- **并发清理阶段（Concurrent Cleanup）**：这个阶段用于清理那些整个区域连一个存活对象都没找到的区域。
- **并发转移阶段（Concurrent Evacuation）**：这个阶段是 Shenandoah 与之前 HotSpot 中其他收集器的核心差异。在这个阶段Shenandoah要把回收集中的存活对象先复制到一份其他未被使用的Region之中。**这个过程在G1 中是会进行暂停执行的，但是在Shenandoah中这个过程是并发进行的。**在移动对象的过程中引用地址会发生变化，此时又不能影响对象的正常访问，这就是一个很复杂的问题了。这里是通过**读屏障和“Brooks Pointers”的转发指针**来解决的，后面我们会详细介绍。这个阶段的持续时间取决于要复制的集合大小。
- **初始引用更新阶段（Init Update Reference）**：并发回收阶段复制对象结束后，还**需要把堆中所有指向旧对象的引用修正到复制后的新地址，这个操作称为引用更新**。引用更新的初始化阶段实际上并未做什么具体的处理，设立这个阶段只是为了建立一个线程集合点，确保所有并发回收阶段中进行的收集器线程都已完成分配给他们的对象移动任务而已。初始引用更新时间很短，会产生一个非常短暂的停顿。

- **并发引用更新（Concurrent Update References）**： 真正开始进行引用更新操作，还要修正存在GC Roots中的引用。这个阶段是与应用线程一起并发执行的，时间的长短取决于内存中涉及的引用数量的多少。并发引用更新与并发标记不同，它不再需要沿着对象图来搜索，只需要按照内存物理地址的顺序，线性地搜索出引用类型，把旧值改成新值而已。
- **最终引用更新（Final Update Reference）**：解决了堆中的引用更新之后，还要修正存在与GC Roots中的引用。这个阶段是 Shenandoah 的最后一次停顿，停顿时间只与 GC Roots 的数量相关。
- **并发清理（Concurrent Cleanup）**：经过并发回收和引用更新之后，整个回收周集中所有的Region已无存活对象，这些 Region 都变成了 Immediate Garbage Regions了，最后在调用一次并发清理过程，供后面分配新的对象使用。



以上是Shenandoah垃圾收集的详细的9个步骤，同样也可以简化为4个步骤：**初始标记，并发标记、最终标记、并发整理**。而在并发整理中，并发引用更新又可以分为4个步骤，**并发转移、初始引用更新、并发引用更新、最终引用更新**。Shenandoah 一个垃圾回收周期中会有4次STW，虽然在STW次数上比G1还要多，但是**整体暂停时间也比G1短。**

> G1一次垃圾回收周期，平均算下来会有2次半STW。其中初始标记和最终标记阶段都会各有一次STW，其中在最后垃圾回收阶段，如果发生转移暂停（Evacuation pause）将会产生一次STW，因此我认为G1的再一次垃圾回收的暂停时间为2次半。

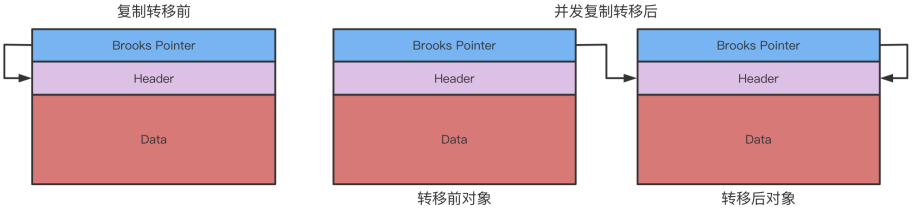
下面的表格是 RedHat 官方在2016年所发表在 Shenandoah 实现论文中给出的实测数据，测试内容是 ElasticSearch 对 200G 的维基百科数据进行索引。这个时候的Shenandoah，还没有发展到完全体，但是从数据来看，暂停时间已经有了质的飞跃。

收集器	运行时间	总停顿	最大停顿	平均停顿
Shenandoah	387.602s	320ms	89.79ms	53.01ms
G1	312.052s	11.7s	1.24s	450.12ms
CMS	285.264s	12.78s	4.39s	852.26ms
Parallel Scavenge	260.092s	6.59s	3.04s	823.75ms

Shenandoah并发整理实现—转发指针（Brooks Pointer）

前面我们提到 Shenandoah 相较于 G1 最大的变化之一就是支持并发整理，在G1中是通过复制算法转移暂停（Evacuation pause），这个复制过程需STW，并且这是G1垃圾回收中最长的一次暂停。而 shenandoah 通过两次短暂的暂停替代一次长时间暂停，来实现复制转移的并发执行。其中**最核心的概念——Brooks Pointer**。“Brooks”是一个人的名字。1984年，Rodney A. Brooks 在论文《Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware》中提出使用转发指针（Forwarding Pointer，也常被称为 Indirection Pointer）来实现对象移动与用户线程并发的一种解决方案。

Brooks提出的转发指针其实很简单，就是在**原有对象布局前面统一增加一个新的引用字段，在正常不处于并发移动的情况下，该指针指向自己，在并发移动后指针指向移动后的对象。**

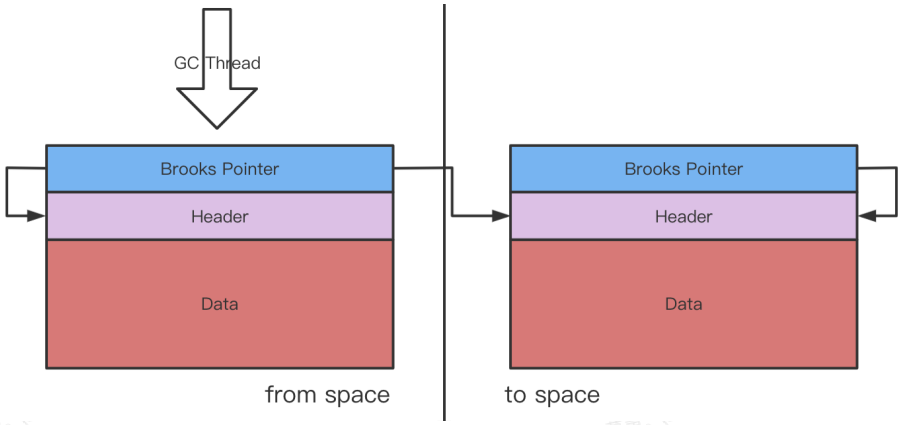


从结构上来看， Brooks 提出的转发之前与某些早期 Java 虚拟机使用过的句柄定位有一些相似之处，两者都是一种间接性的对象访问方式，差别是句柄通常是统一存储在专门的句柄池中，而转发指针是分散在每个对象头前面。再使用了转发指针之后，能实现并发的进行对象复制转移，但是带来的问题也是很明显的，在每次访问对象时，都会带来一次额外的转发开销，虽然这个开销已经被优化到一条汇编命令的程度，但是**对象定位命令被频繁使用到，这仍然是一笔不能忽视的性能开销。**

> 计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。

这样的设计虽然好，但是会带来一个比较棘手的问题即**多用户线程并发操作和GC线程之间的并发竞争问题。**如果是应用线程并发读操作的场景，问题都不是很大，因为Brooks Poniter，都可以指向新的对象，但是如果是写入线程操作和GC线程并发的场景下，会发生什么呢？不妨想象一下这三个事件同时发生：

1. GC线程复制了新的对象副本。
2. 应用线程更新对象某个字段。
3. GC线程更新转发指针的引用值为新对象副本。



如果不做任何保护措施，按照事件1，事件2，事件3顺序发生，那么就会出现应用线程将更新写入到复制前的旧对象中，且新对象中还不含应用线程的更新，这显然是不正确的。这个问题我们只要保证GC线程在执行过程中，不被应用线程穿插执行就可以解决，即事件按照1、3、2或者 2、1、3的顺序执行。了解过并发编程的同学肯定能很自然的想到解决办法：GC线程和应用线程加锁访问。实际上**Shenandoah 也是通过CAS的方式来实现这里的并发控制。**

Shenandoah 还有要解决的问题——读屏障的性能问题，前面的我们提到写屏障，其他垃圾收集器用写屏障来维护卡表，亦或者用来实现并发标记，写屏障已经积累了一部分的处理任务了。一部分读屏障在Shenandoah中会被继续使用，同时 Shenandoah 还需要往读写屏障中加入额外的转发处理，尤其是使用读屏障的代价会比写屏障大得多，因为读屏障会被更加频繁的使用，因此读屏障的使用也需要更加的小心，不允许任何的重量级操作。**Shenandoah 是目前第一个使用到读屏障的收集器。**它的开发者也意识到了数量庞大的读屏障带来性能开销会是 Shenandoah 被诟病的关键点之一。所以计划在 JDK 13 中将Shenandoah的内存屏障模型改进为**基于引用访问屏障（Load Reference Barrier）的实现**，所谓“引用访问屏障”是值内存屏障指拦截对象类型为引用类型的读写操作，而不去处理原生数据类型和其他非引用字段的读写，这能省去大量的成本。

> 可以把读屏障理解为一段代码，或者是一个指令，后面挂着对应的处理函数。例如下面的代码中，两行load操作对应的代码都插入了读屏障，但ZGC在第一个读屏障触发之后，不但将a的值更新为最新的，通过 self healing 机制使得 obj.x 的指针也会被修正，第二个读屏障再触发时就直接进入FastPath，基本上没有什么性能损耗了； 而Shenandoah 则不会修正obj.x的值，所以第二个读屏障又要进行一次SlowPath。

Java 复制代码

```
1  Object a = obj.x;
2  Object b = obj.x;
```

启发式参数

启发式参数告知 Shenandoah GC何时开始GC处理，以及确定要归集的堆块。可以使用`-XX:ShenandoahGCHeuristics=`来选择不同的启发模式，有些启发模式可以配置一些参数，帮 助我们更好地使用GC。可用的启发模式如下:

- 自适应模式（adaptive）此为默认参数，通过观察之前的一些GC周期，以便在堆耗尽之前尝试启动下一个GC周期。

```bash
-XX:ShenandoahInitFreeThreshold=# :触发“学习”集合的初始阈值
-XX:ShenandoahMinFreeThreshold=# :启发式无条件触发GC的可用空间阈值
-XX:ShenandoahAllocSpikeFactor=# :要保留多少堆来应对内存分配峰值
-XX:ShenandoahGarbageThreshold=# :设置在将区域标记为收集之前需要包含的垃圾百分比
```

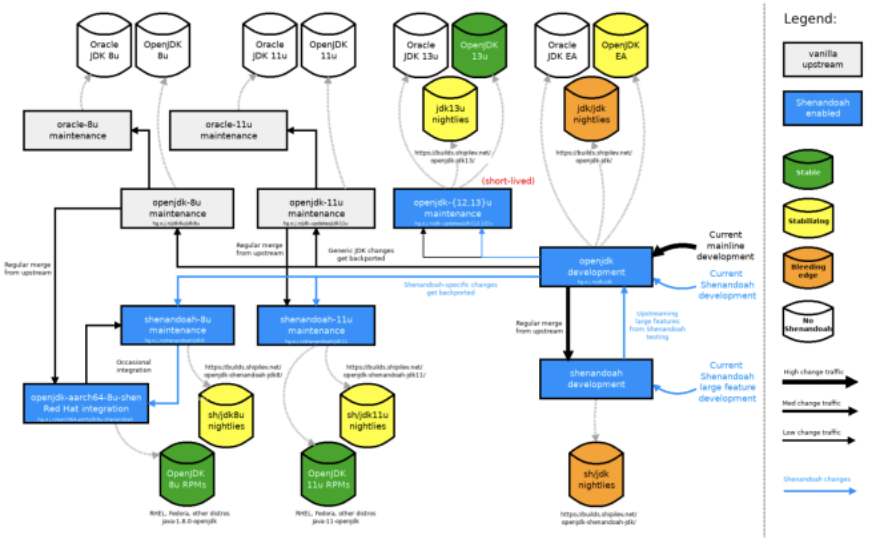
- 静态模式（static） 根据堆使用率和内存分配压力决定是否启动GC周期。

```bash
-XX:ShenandoahFreeThreshold=# : 设置空闲堆百分比阈值;
-XX:ShenandoahAllocationThreshold=# : 设置内存分配量百分比阈值;
-XX:ShenandoahGarbageThreshold=# : 设置小堆块标记为可回收的百分比阈值
-XX:ShenandoahFreeThreshold=# :设置启动GC周期时的可用堆百分比阈值;
-XX:ShenandoahAllocationThreshold=# :设置从上一个GC周期到新的GC周期开始之前的内存分配百分比阈值;
-XX:ShenandoahGarbageThreshold=# :设置在将区域标记为收集之前需要包含的垃圾百分比阈值;
```

- 紧凑模式（compact） 只要有内存分配，就会连续运行GC回收，并在上一个周期结束后立即开始下一个周期。此模式通常会有吞 吐量开销，但能提供最迅速的内存空间回收。

```bash
-XX:ConcGCThreads=# :设置并发GC线程数，可以减少并发GC线程的数量，以便为应用程序运行 留出更多空间
-XX:ShenandoahAllocationThreshold=# :设置从上一个GC周期到新的GC周期开始之前的内 存分配百分比
```

- 被动模式（passive） 内存一旦用完，则发生STW，用于系统诊断和功能测试。
- 积极模式（aggressive）它将尽快在上一个GC周期完成时启动新的GC周期（类似于“紧凑型”），并且将全部的存活对象归集到一 块，这会严重影响性能，但是可以被用来测试GC本身。有时候启发式模式会在判断后把更新引用阶段和并发标记阶段合并。可以通过`-XX:ShenandoahUpdateRefsEarly=[on|off]`强制启用和禁用这个特性。 同时针对于内存分配失败时的策略，可以通过调节 ShenandoahPacing 和 ShenandoahDegeneratedGC 参数，对线程进行一定的调节控制。如果还是没有足够的内存，最坏的情 况下可能会产生Full GC，以使得系统有足够的内存不至于发生OOM。 更多有关如何配置、调试 Shenandoah 的参数信息，请参阅Shenandoah官方wiki页面。 前面我们也提到 Shenandoah 是垃圾收集器中的“孤儿”，只有OpenJDK 中才包含它，下图是各版本JDK对Shenandoah的集成情况：



这张图展示了Shenandoah GC目前在各个JDK版本上的进展情况，可以看到OpenJDK12和13上都可以用。 在Red Hat Enterprise Linux、Fedora系统中则可以在JDK8和JDK11版本上使用（肯定的，这两个Linux发行 版都是Red Hat的，谁让这个GC也是Red Hat开发维护的呢）。 默认情况下，OpenJDK

12+发布版本通常包括Shenandoah； Fedora 24+中OpenJDK 8+发布版本包括Shenandoah； RHEL 7.4+中OpenJDK 8+发布版本中包括Shenandoah作为技术预览版； 基于RHEL/Fedora的发行版或其他使用它们包装的发行版也可能启用了Shenandoah（CentOS、 Oracle Linux、Amazon Linux中也带了Shenandoah）。

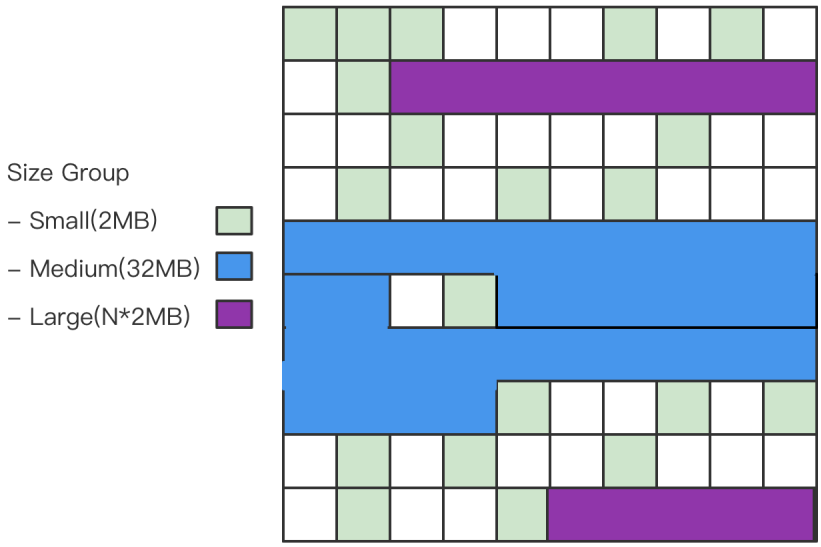
ZGC

我们前面梳理了 Shenandoah 有关知识点，Shenandoah 是一款在任何堆大小下都非常优秀的低延迟垃圾收集器。但是由于这款垃圾收集器不是由HotSpot主导设计的，Shenandoah 不免一直受到官方的排挤。直至目前为止，Shenandoah 是唯一一款仅存在 OpenJDK 的垃圾收集器。Shenandoah 已经如此的优秀了，那有着“正宗血统”的ZGC有什么秘密武器呢？ZGC（Z Garbage collector）是一款在2011年加入的具有实现性质的低延迟垃圾收集器，由Oracle公司主持研发。并且Oracle于2018年将ZGC提交给 OpenJDK，推动其进入OpenJDK 11的发布清单中。ZGC和Shenandoah的目标高度相似，**都希望在尽可能对吞吐量影响不大的前提下，实现在任意堆内存大小下都可以把垃圾收集的停顿时间限制在10ms以内的低延迟。**但是他们的实现方式却有很大的差别。RedHat开发的Shenandoah像是G1的继承者的话，那Oracle开发的ZGC则更像是 Azul System开发的PGC（pauseless GC）和 C4（Concurrent Continuously Compacting Collector）的同胞兄弟。

ZGC的内存布局

ZGC的内存分布和Shenandoah、G1的一致，都是采用基于 Region 的内存布局，ZGC与他们稍微有些不同的是在一些官方资料中，ZGC的Region被称为Page或ZPage。ZGC的Region有一个很不一样的特点，它具有动态性—动态的创建和销毁，以及以及动态的区域容量大小。在x86的平台架构下，**ZGC的Region可以具有大、中、小三种类型容量**：

- 小型Region（Small Region）：容量固定为`2MB`，用于放置**小于256KB**的小对象。
- 中型Region（Medium Region）：容量固定为`32MB`，用于放置**大于等于256KB但小于4MB**的对象。
- 大型Region（Large Region）：容量`不固定`，可以动态变化，但必须为**2MB的整数倍**，用于放置4MB 或以上的大对象，每个大型 Region 中只会存放一个大象，他的**实际容量可能小于中型Region**，最小容量可低至4MB。大型Region在ZGC的实现中是不会被重分配（重分配是ZGC的一个处理动作，用于收集器并发复制阶段，后面会介绍到），因为复制一个大对象的代价非常高昂。



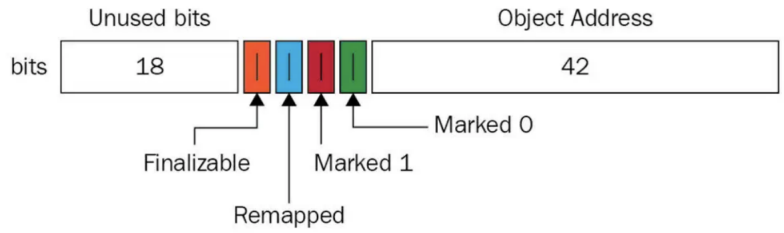
ZGC并发整理实现—染色指针（Colored Pointer）

新一代低延迟垃圾收集器，相较于以G1为代表的后经典垃圾收集器的最大的不同就是实现了并发整理。前面我们介绍**Shenandoah是通过转发指针（Brooks Pointer）和读屏障**来实现。这个实现方案虽然能解决并发复制问题，但整体上还是会带来一些性能和内存上的小瑕疵，不是那么完美。不同于Shenandoah 的解决方案，ZGC选择了另外一条复杂精巧的解决方案—**染色指针技术**（Color Pointer，其他类似的技术中可能称它为Tag Pointer或Version Pointer）。

前面提到，计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。shenandoah 是通过增加一个 Brooks Pointer作为中间层实现转发，而ZGC就是通过染色指针作为中间层。 那这个对象是否移动的信息，即又是哪个指针被染色了呢？在JVM中，我们通常在对象头上增加额外存储一些对象信息，其中对象的hashCode、分代年龄、锁记录等就是这样存储的。这种在能够访问到对象的场景下访问额外信息是很自然流畅的。 但是在这个场景，它明显是不适用的。此时我们在访问对象的过程中，还未获取到对象又怎么能访问到对象头中的信息呢？这个信息**ZGC放在了指向对象的引用指针上。**

> 在64位系统中，理论上可以访问的内存高达16EB字节（2的64次幂），实际上基于需求、性能、成本的考虑实际上也用不了这么多，在AMD64架构中，只支持到了52位4PB的地址总线和48位（256TB）的虚拟地址空间，所以目前64位的硬件最大只能支持到256TB。此外操作系统一侧还会加上自己的约束，64位的Linux则分别支持47位（128TB）的虚拟内存地址和46位（64TB）物理内存地址。而windows只支持44位（16TB）物理内存地址。

尽管Linux下64位指针的高18位不能用来寻址，但是剩余的46位指针所能支持的64TB的内存在今天仍然能充分满足大型服务器的需要，因此，ZGC的染色指针盯上这46位寻址地址，截取最高的4位来存储4个标识信息。通过这4个标记，判断对象当前状态，进而进行下一步寻址。但**这4位的占用也进一步压缩了地址空间，让原本的46位空间变成了44位，即ZGC最大可管理的内存空间减少到4TB。**



虽然染色指针有4TB的内存限制，不能支持32位平台，不能支持压缩指针等约束，但它带来的收益却是非常可观的。其中染色包括下面的三大主要优势：

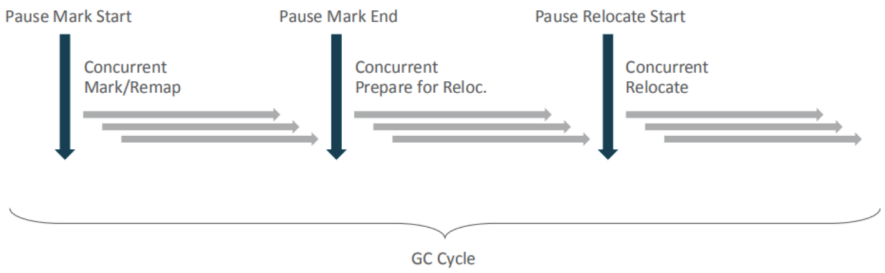
- **染色指针可以使得一旦某个Region的存活对象被移走之后，这个Region立即就能被释放重用，而不用等待整个堆中的所有指向该Region的引用都被修正之后才能被清理。**这一点相比起Shenandoah是一个颇大的优势。使得只要理论上还有一个空闲的Region，ZGC就能完成收集，而Shenandoah需要等待所有的引用更新完成才能释放回收集中的Region，这意味着如果堆中几乎所有的对象都存活，那么需要1:1的空闲Region完成对象的复制。相较于这一点ZGC有绝对的优势，为什么会有这样的效果，这全部归功于ZGC“自愈”的特性。
- **染色指针可以大幅度减少在垃圾收集过程中内存屏障的使用数量**，设置内存屏障，尤其是写屏障的目的是为了记录对象引用的变动情况，如果将这些信息直接维护在指针中，则可以省下一些专门的记录操作，ZGC没有使用任何的读屏障，只使用了部分的写屏障。没有了读屏障，ZGC的吞吐量自然也更高一些。
- **染色指针可以作为一种扩展数据结构用来记录更多的信息，以便日后提高性能**。但是同时我们也需要占用更多寻址位置。Linux下寻址的指针的前18位是没有使用的，这个时候如果扩展利用，不仅可以扩展ZGC可管理的最大堆4TB上限，还可以扩展更多的功能提高垃圾回收效率。

垃圾收集工作周期

上面我们简单介绍了ZGC 的内存布局和染色指针技术，接下来我们一起来看看ZGC的垃圾回收工作周期。ZGC的工作周期大致可以分为4个大阶段，并且这个4个阶段全部都是可以并发执行的，仅是两个阶段中间会存在短暂的停顿小阶段。以下ZGC的回收阶段：

ZGC的原理

ZCG的GC周期如图所示:

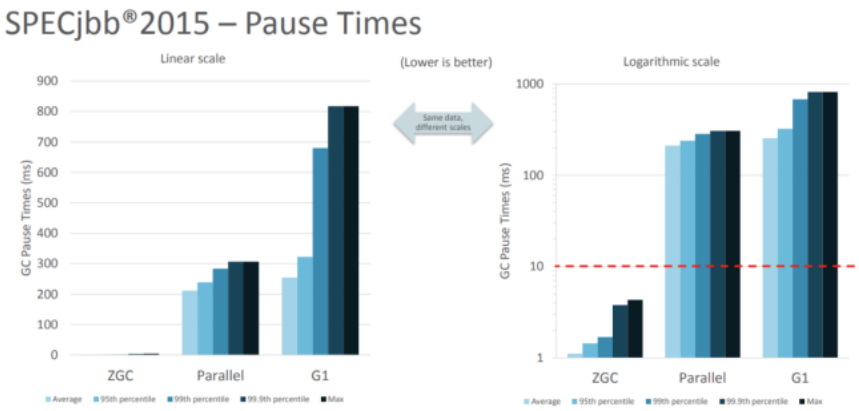


- **并发标记（Concurrent Mark）**：与G1、Shenandoah 一样，并发标记是遍历对象图做可达性分析的阶段，前后也要经过类似于G1、shenandoah 的初始标记、最终标记的短暂停顿，而且这些停顿阶段所做的事情在目标上也是相类似的。不同的是，ZGC的标记是在指针上而不是在对象上进行，标记阶段会更新染色指针中的`Marked0`、`Marked1`。
- **并发预备重分配（Concurrent Prepare for Relocate）**：这个阶段需要根据特定的查询条件统计得出本次收集过程要清理哪些 Region，将这些 Region 组成重分配集（Relocation Set）。重分配集与G1收集器的回收集（Collection Set）还是有区别的，ZGC 划分Region的目的并不是像G1那样做收益优先的增量回收。相反ZGC每次回收都会扫描所有的Region，用更大范围的扫描成本换取G1中记忆集的维护成本。ZGC的重分配集只是决定了里面的存活对象会被重新复制到其他的Region中，里面的 Region 会被释放，而并不能说回收行为就只是针对这个集合中的Region进行的。因为标记的过程是针对全堆的。
- **并发重分配（Concurrent Relocate）**：重分配是ZGC执行过程中的核心阶段，这个过程要把重分配集中的存活对象复制到新的Region上，并重分配集中的某个Region维护一张转发表（Forward Table），记录从旧对象到新对象的转向关系。得益于染色指针的支持，**ZGC收集器能仅从引用上就明确得知一个对象是否处于重分配集合中，如果用户线程此时并发访问位于重分配集中的对象，这次访问将会被遇到的内存屏障截获，然后立即根据Region上的转发记录将访问转发到新复制的对象上，并且同时修正更新该引用的值使其直接指向新对象。**ZGC将这种行为称为指针的“自愈”（Self-Healing）能力。

> 这种自愈能力相较于Shenandoah的Brooks Pointer，只有第一次会和Brooks需要进行一次转发，而后续的访问都是直接访问。

- **并发重映射（Concurrent Remap）**：重映射所做的就是修正堆中执行重分配集中旧对象的所有引用，这一点从目标角度看是与Shenandoah 并发引用更新阶段一样的，但是ZGC的并发重映射并不是一个“迫切”要去完成的任务。因为ZGC的引用是可以“自愈”的，最多只是多使用一次转发操作。**重映射的主要目的是为了不变慢，当然还有清理转发表这样的附带的收益**。因此**并发重映射放在了下一次垃圾收集工作周期中的并发标记阶段处理**。这样就减少了一次遍历对象图的开销。

得益于这些优秀的设计，ZGC的性能相较于上一代Parallel 、G1也是异常的出众，表现用“**令人震惊，革命性的ZGC**”来描述毫不为过。



参数介绍

我们可以使用`-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`参数可以用来启用，ZGC一些常用参数如下：

- `-XX:ZCollectionInterval`:固定时间间隔进行gc，默认值为0。
- `-XX:ZAllocationSpikeTolerance`:内存分配速率预估的一个修正因子，默认值为2，一般不需要更改。
- `-XX:ZProactive`:是否启用主动回收策略，默认值为true，建议开启。
- `-XX:ZUncommit`:将不再使用的内存还给OS，JDK13以后可以使用； JVM会让内存不会降到 Xms 以下，所以如果Xmx和Xms配置一样这个参数就会失效。
- `-XX:+UseLargePages -XX:ZPath`:使用大内存页。 Large Pages在Linux称为Huge Pages，配置zgc使用Huge Pages可以获得更好的性能（吞吐量、延迟、启动时间）。 配置Huge Pages时，一般配合ZPath使用。配置方法可以见:https://wiki.openjdk.java.net/display/zgc/Main
- `-XX:UseNUMA`:启用NUMA支持【挂载很多CPU，每个CPU指定一部分内存条的系统】。 ZGC默认开启NUMA支持，意味着在分配堆内存时，会尽量使用NUMA-local的内存。开启和关闭可以使用`-XX:+UseNUMA`或者`-XX:-UseNUMA`。
- `-XX:ZFragmentationLimit`:根据当前region已大于ZFragmentationLimit，超过则回收，默认为25。
- `-XX:ZStatisticsInterval`:设置打印ZStat统计数据(cpu、内存等log)的间隔。

此外还有前面提过的并发线程数参数`-XX:ConcGCThreads=n`，这个参数对于并发执行的GC 策略都很重要，需要根据CPU核心数考虑，配置太多导致线程切换消耗太大，配置太少导致回收垃圾速度跟不上系统使用的速度。

总结

这一节我们梳理了全新一代“低延迟垃圾回收器”垃圾处理器，即Shenandoah和ZGC，Shenandoah由RedHat主持开发，立项时间早于ZGC。整体设计上和G1类似，在G1的基础上创新的抛弃了分代整理，使用“链接矩阵”代替RSet并支持并整理，极大的缩短了shenandoah 的STW时间。Shenandoah的垃圾回收流程整体上和G1类似，在最后的整理阶段G1通过STW对象复制转移来进行堆空间整理，而Shenandoah 这是通过这个过程拆分成复制、并发引用更新，最终引用更新和清理这几个阶段，实现大部分情况下并发执行。只有在复制、最终引用更新时候需要进行短暂的STW。计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。同样，Shenandoah通过增加一个Brooks Pointer和读屏障来解决并发整理过程中的对象定位问题。正因为并发整理的引入，Shenadoah 的暂停时间相较于传统的垃圾收集器有了质的飞跃。

Shenandoah 由于不是Oracle官方开发的垃圾收集器，难免受到一些来自官方的排挤。“根正苗红”的ZGC也有非常亮眼的表现，从技术角度来看ZGC采用了一条完全不同的路线，ZGC使用染色指针来巧妙的实现这个中间层。ZGC通过在引用上做文章，它占用引用地址的4个高位来记录是否被标记，是否重映射等信息。但是这样的占用同时也给ZGC带来了诸如最大管理4TB，不能开启压缩指针等限制。最终实验数据证明这些“代价”换来的提升是完全值得的。ZGC的垃圾回收工作周期和G1、shenandoah是相似的，不同的点也是在最后并发整理上，得益于染色指针的优秀设计，ZGC能在并发整理重映射的过程中可以更快的释放Region，提高空间利用率，并且指针还具有自愈功能，这相较于Shenandoah每次访问次次转发又有了极大的性能提升。而延迟上ZGC也是把传统垃圾收集器秒的一塌糊涂。

这一路过来我们看了这么多垃圾收集器，我相信你也建立以对垃圾收集器全面的认知体系。每个垃圾收集器都很优秀，如果把Java比作一个女王，那么垃圾收集器就是她的权利的皇冠，而ZGC则是皇冠上最亮眼的那一颗宝石。垃圾收集器的梳理就暂时告一段落了，但是通过这两小节两万字的总结我很庆幸，我很高兴我能如此认真全面的认识Java垃圾收集器。继续前进，加油！

学习资料

- 深入理解Java虚拟机（第三版）
- ZGC 和 Shenandoah 介绍