https://mp.weixin.qq.com/s/5BtVdLgsNu8U0R8sWOl7xg

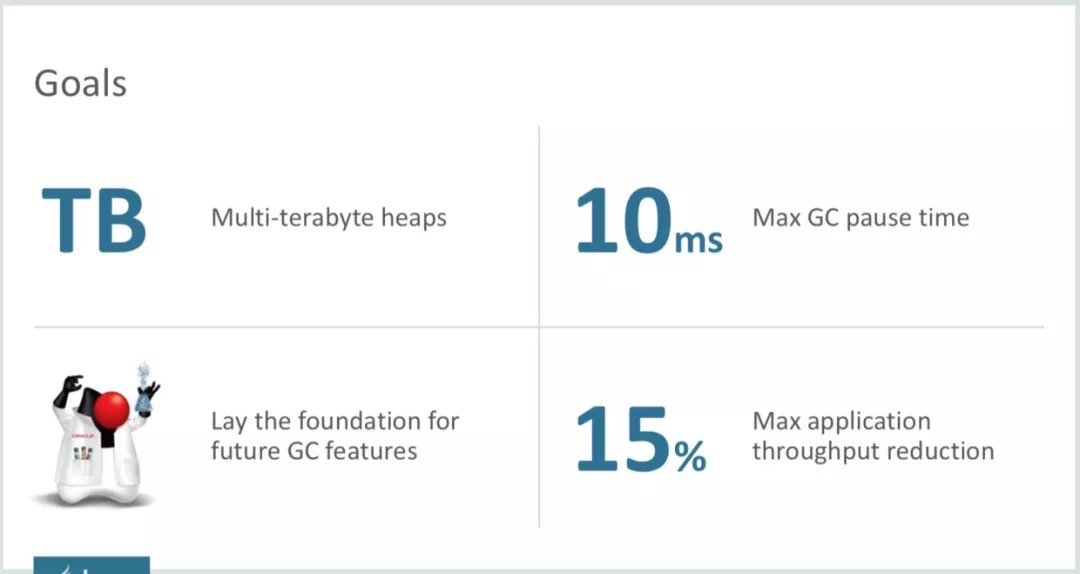
# 干掉cms，zgc才是未来

JAVA程序最爽的地方是它的GC机制，开发人员不需要关注内存申请和回收问题。同时，JAVA程序最头疼的地方也是它的GC机制，因为掌握JVM和GC调优是一件非常困难的事情。在ParallelOldGC、CMS、G1之后，JDK11带来的全新的「ZGC」为我们解决了什么问题？Oracle官方介绍它是一个Scalable、Low Latency的垃圾回收器。所以它的目的是「降低停顿时间」，由此会导致吞吐量会有所降低。吞吐量降低问题不大，横向扩展几台服务器就能解决问题了啦。

### **ZGC目标**

如下图所示，ZGC的目标主要有4个：

* 支持TB量级的堆。这你受得了吗？我们生产环境的硬盘还没有上TB呢，这应该可以满足未来十年内，所有JAVA应用的需求了吧。
* 最大GC停顿时间不超10ms。这你受得了吗？目前一般线上环境运行良好的JAVA应用Minor GC停顿时间在10ms左右，Major GC一般都需要100ms以上（G1可以调节停顿时间，但是如果调的过低的话，反而会适得其反），之所以能做到这一点是因为它的停顿时间主要跟Root扫描有关，而Root数量和堆大小是没有任何关系的。
* 奠定未来GC特性的基础。
* 最糟糕的情况下吞吐量会降低15%。这都不是事，停顿时间足够优秀。至于吞吐量，通过扩容分分钟解决。



另外，Oracle官方提到了它最大的优点是：它的停顿时间不会随着堆的增大而增长！也就是说，几十G堆的停顿时间是10ms以下，几百G甚至上T堆的停顿时间也是10ms以下。

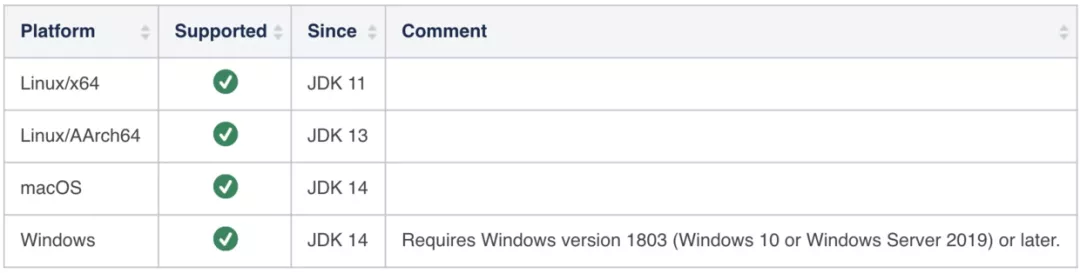
### **ZGC概述**

接下来从几个维度概述一下ZGC。

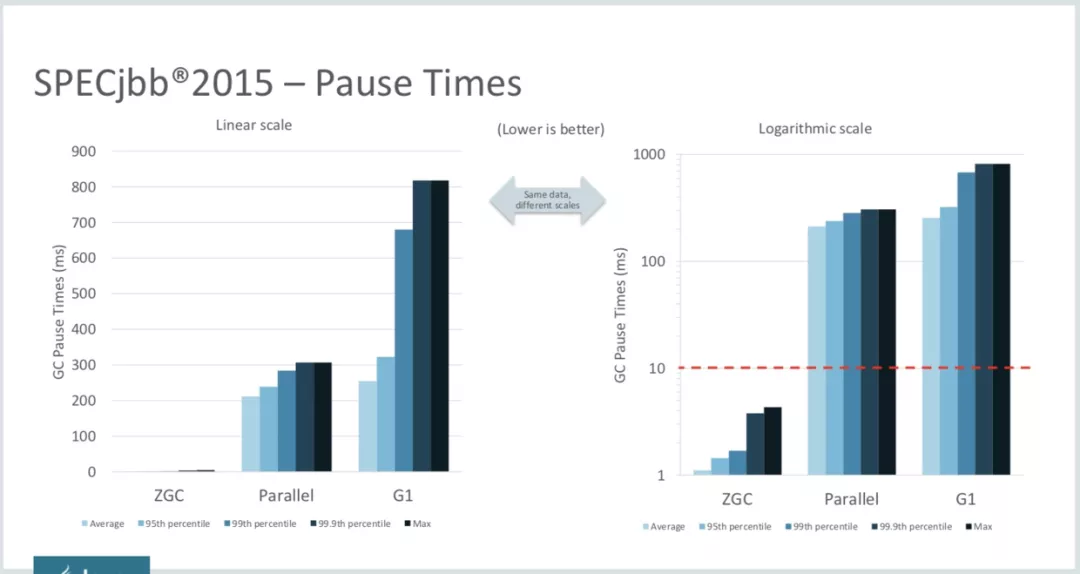
* New GC
* Single Generation
* Region Based
* Partial Compaction
* NUMA-aware
* Colored Pointers
* Load Barriers
* ZGC tuning
* Change Log

### **New GC**

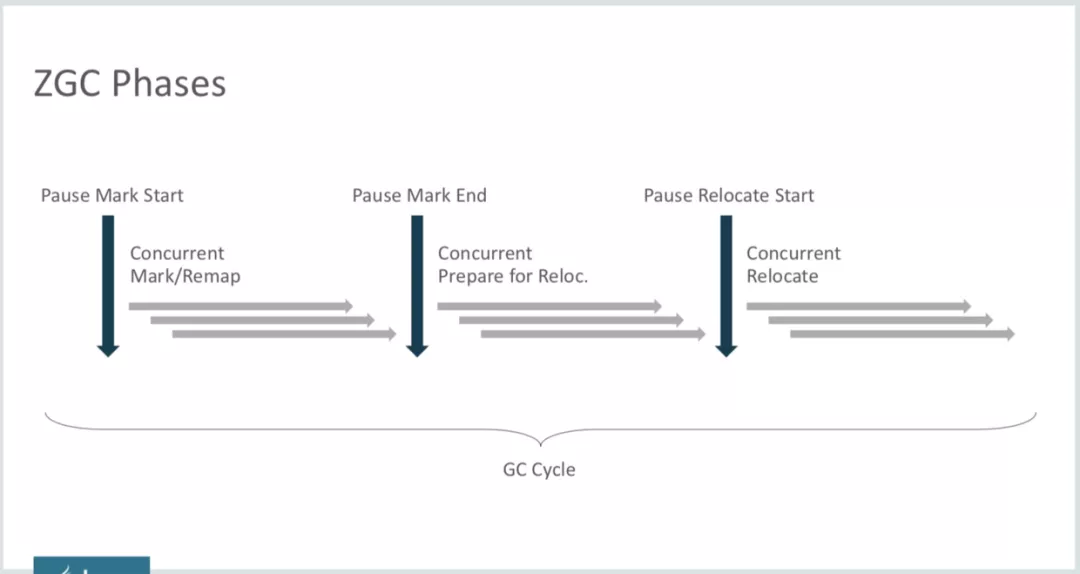
ZGC是一个全新的垃圾回收器，它完全不同以往HotSpot的任何垃圾回收器，比如：PS、CMS、G1等。如果真要说它最像谁的话，那应该是Azul公司的商业化垃圾回收器：「C4」，ZGC所采用的算法就是Azul Systems很多年前提出的Pauseless GC，而实现上它介于早期Azul VM的Pauseless GC与后来Zing VM的C4之间。不过需要说明的是，JDK11中ZGC只能运行在Linux64操作系统之上。JDK14新增支持了MacOS和Window平台：



如下图所示，是ZGC和Parallel以及G1的压测对比结果（CMS在JDK9中已经被标记deprecated，更高版本中已经被彻底移除，所以不在对比范围内）。我们可以明显的看到，停顿时间方面，ZGC是100%不超过10ms的，简直是秒天秒地般的存在：

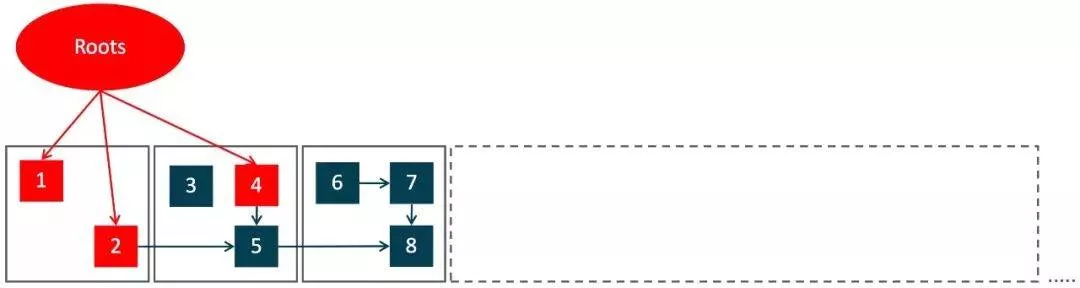


接下来，再看一下ZGC的垃圾回收过程，如下图所示。由图我们可知，ZGC依然没有做到整个GC过程完全并发执行，依然有3个STW阶段，其他3个阶段都是并发执行阶段：



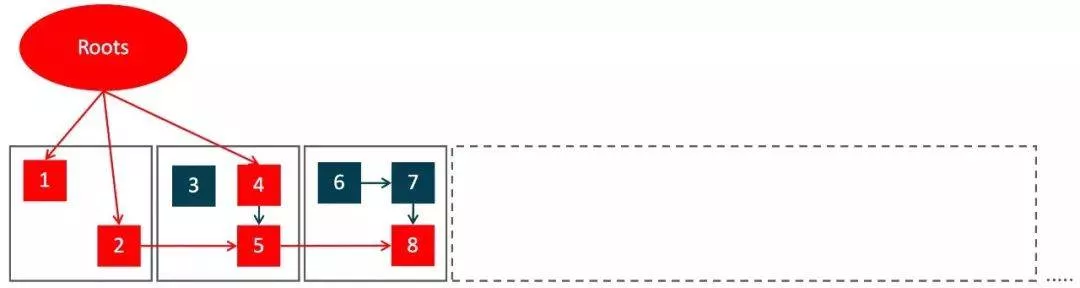
#### **Pause Mark Start**

这一步就是初始化标记，和CMS以及G1一样，主要做Root集合扫描，「GC Root是一组必须活跃的引用，而不是对象」。例如：活跃的栈帧里指向GC堆中的对象引用、Bootstrap/System类加载器加载的类、JNI Handles、引用类型的静态变量、String常量池里面的引用、线程栈/本地(native)栈里面的对象指针等，但不包括GC堆里的对象指针。所以这一步骤的STW时间非常短暂，并且和堆大小没有任何关系。不过会根据线程的多少、线程栈的大小之类的而变化。



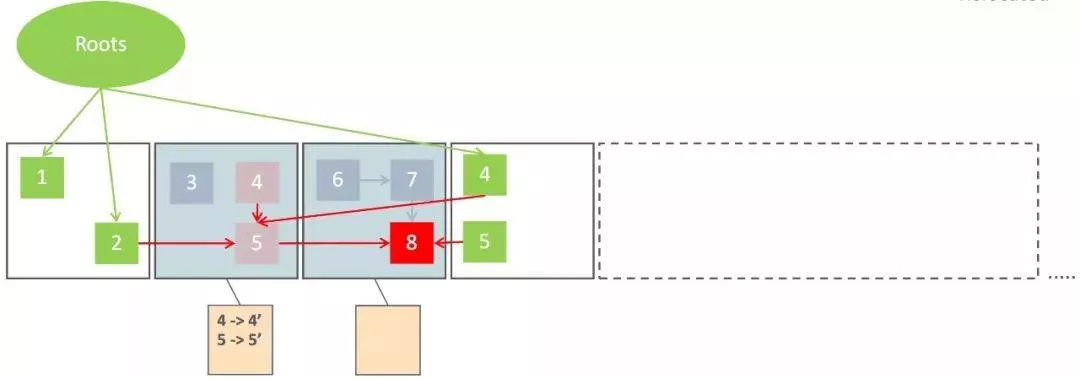
#### **Concurrent Mark/Remap**

第二步就是并发标记阶段，这个阶段在第一步的基础上，继续往下标记存活的对象。并发标记后，还会有一个短暂的暂停（Pause Mark End），确保所有对象都被标记。



#### **Concurrent Prepare for Relocate**

即为Relocation阶段做准备，选取接下来需要标记整理的Region集合，这个阶段也是并发执行的。接下来又会有一个Pause Relocate Start步骤，它的作用是只移动Root集合对象引用，所以这个STW阶段也不会停顿太长时间。



#### **Concurrent Relocate**

最后，就是并发回收阶段了，这个阶段会把上一阶段选中的需要整理的Region集合中存活的对象移到一个新的Region中（这个行为就叫做「Relocate」，即重新安置对象），如上图所示。Relocate动作完成后，原来占用的Region就能马上回收并被用于接下来的对象分配。细心的同学可能有疑问了，这就完了？Relocate后对象地址都发生变化了，应用程序还怎么正常操作这些对象呢？这就靠接下来会详细说明的Load Barrier了。

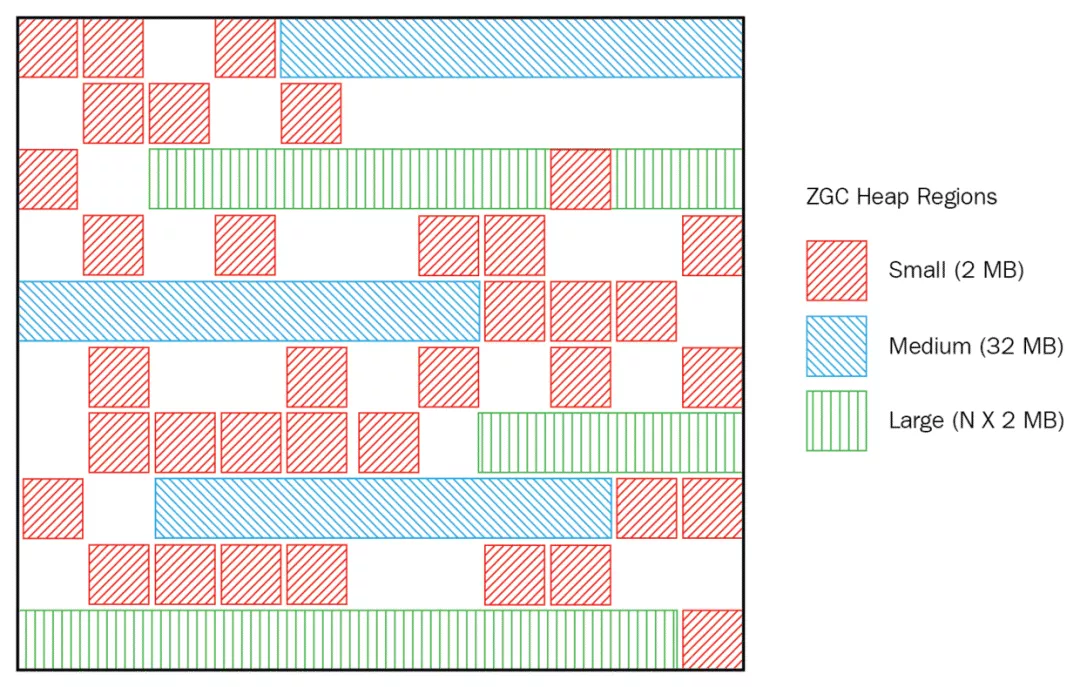
### **Single Generation**

单代，即ZGC「没有分代」。我们知道以前的垃圾回收器之所以分代，是因为源于“「大部分对象朝生夕死」”的假设，事实上大部分系统的对象分配行为也确实符合这个假设。

那么为什么ZGC就不分代呢？因为分代实现起来麻烦，作者就先实现出一个比较简单可用的单代版本。用符合我们国情的话来解释，大概就是说：工作量太大了，人力又不够，老板，先上个1.0版本吧！！！

### **Region Based**

这一点和G1一样，都是基于Region设计的垃圾回收器，ZGC中的Region也被称为「ZPages」，ZPages被动态创建，动态销毁。不过，和G1稍微有点不同的是，G1的每个Region大小是完全一样的，而ZGC的Region大小分为3类：2MB，32MB，N×2MB，如此一来，灵活性就更好了：



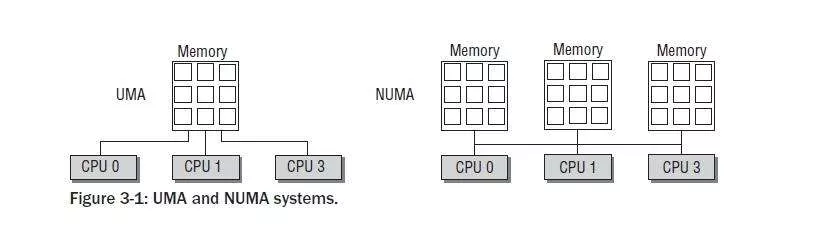
### **Partial Compaction**

部分压缩，这一点也很G1类似。以前的ParallelOldGC，以及CMS GC在压缩Old区的时候，无论Old区有多大，必须整体进行压缩（CMS GC默认情况下只是标记清除，只会发生FGC时才会采用Mark-Sweep-Compact对Old区进行压缩），如此一来，Old区越大，压缩需要的时间肯定就越长，从而导致停顿时间就越长。

而G1和ZGC都是基于Region设计的，在回收的时候，它们只会选择一部分Region进行回收，这个回收过程采用的是Mark-Compact算法，即将待回收的Region中存活的对象拷贝到一个全新的Region中，这个新的Region对象分配就会非常紧凑，几乎没有碎片。垃圾回收算法这一点上，和G1是一样的。

### **NUMA-aware**

NUMA对应的有UMA，UMA即Uniform Memory Access Architecture，NUMA就是Non Uniform Memory Access Architecture。UMA表示内存只有一块，所有CPU都去访问这一块内存，那么就会存在竞争问题（争夺内存总线访问权），有竞争就会有锁，有锁效率就会受到影响，而且CPU核心数越多，竞争就越激烈。NUMA的话每个CPU对应有一块内存，且这块内存在主板上离这个CPU是最近的，每个CPU优先访问这块内存，那效率自然就提高了：

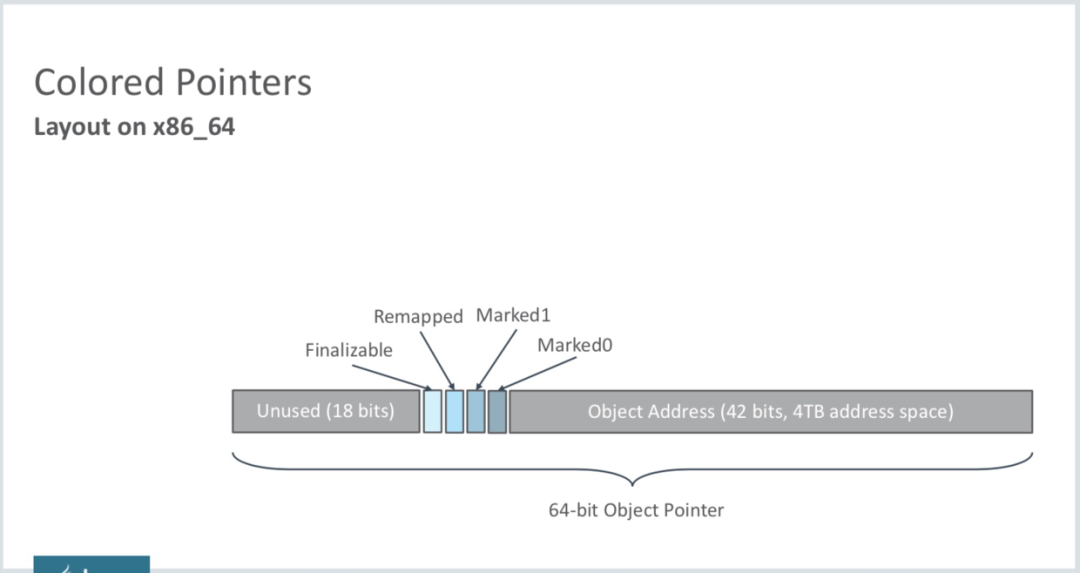


服务器的NUMA架构在中大型系统上一直非常盛行，也是高性能的解决方案，尤其在系统延迟方面表现都很优秀。ZGC是能自动感知NUMA架构并充分利用NUMA架构特性的。

### **Colored Pointers**

Colored Pointers，即颜色指针是什么呢？如下图所示，ZGC的核心设计之一。以前的垃圾回收器的GC信息都保存在对象头中，而ZGC的GC信息保存在指针中。每个对象有一个64位指针，这64位被分为：

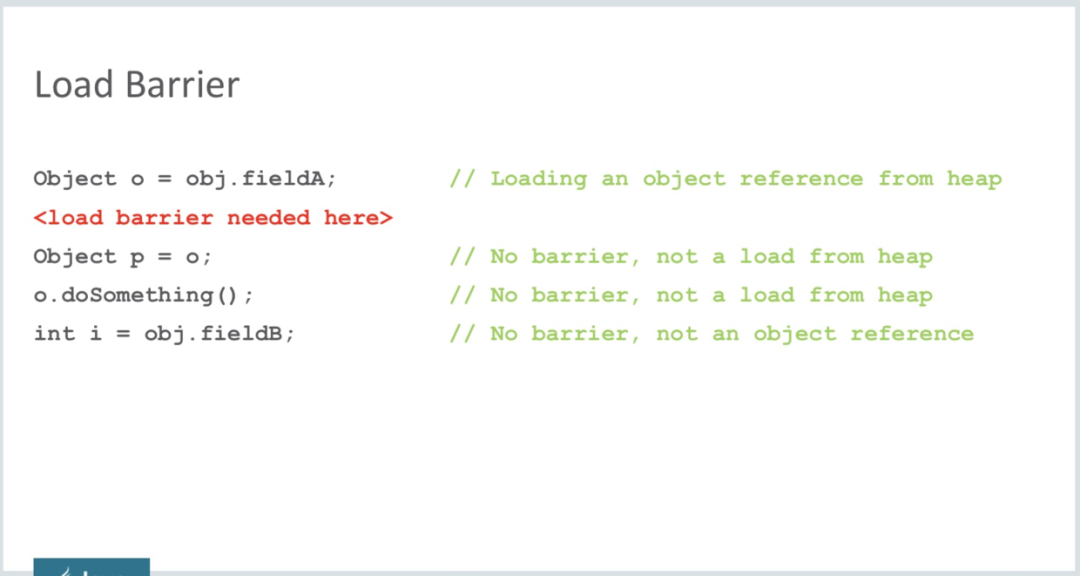
* 18位：预留给以后使用；
* 1位：Finalizable标识，次位与并发引用处理有关，它表示这个对象只能通过finalizer才能访问；
* 1位：Remapped标识，设置此位的值后，对象未指向relocation set中（relocation set表示需要GC的Region集合）；
* 1位：Marked1标识；
* 1位：Marked0标识，和上面的Marked1都是标记对象用于辅助GC；
* 42位：对象的地址（所以它可以支持2^42=4T内存）：



通过对配置ZGC后对象指针分析我们可知，对象指针必须是64位，那么ZGC就无法支持32位操作系统，同样的也就无法支持压缩指针了（CompressedOops，压缩指针也是32位）。

### **Load Barriers**

这个应该翻译成读屏障（与之对应的有写屏障即Write Barrier，之前的GC都是采用Write Barrier，这次ZGC采用了完全不同的方案），这个是ZGC一个非常重要的特性。在标记和移动对象的阶段，每次「从堆里对象的引用类型中读取一个指针」的时候，都需要加上一个Load Barriers。那么我们该如何理解它呢？看下面的代码，第一行代码我们尝试读取堆中的一个对象引用obj.fieldA并赋给引用o（fieldA也是一个对象时才会加上读屏障）。如果这时候对象在GC时被移动了，接下来JVM就会加上一个读屏障，这个屏障会把读出的指针更新到对象的新地址上，并且把堆里的这个指针“修正”到原本的字段里。这样就算GC把对象移动了，读屏障也会发现并修正指针，于是应用代码就永远都会持有更新后的有效指针，而且不需要STW。那么，JVM是如何判断对象被移动过呢？就是利用上面提到的颜色指针，如果指针是Bad Color，那么程序还不能往下执行，需要「slow path」，修正指针；如果指针是Good Color，那么正常往下执行即可：



❝ 这个动作是不是非常像JDK并发中用到的CAS自旋？读取的值发现已经失效了，需要重新读取。而ZGC这里是之前持有的指针由于GC后失效了，需要通过读屏障修正指针。❞

后面3行代码都不需要加读屏障：Object p = o这行代码并没有从堆中读取数据；o.doSomething()也没有从堆中读取数据；obj.fieldB不是对象引用，而是原子类型。

正是因为Load Barriers的存在，所以会导致配置ZGC的应用的吞吐量会变低。官方的测试数据是需要多出额外4%的开销：

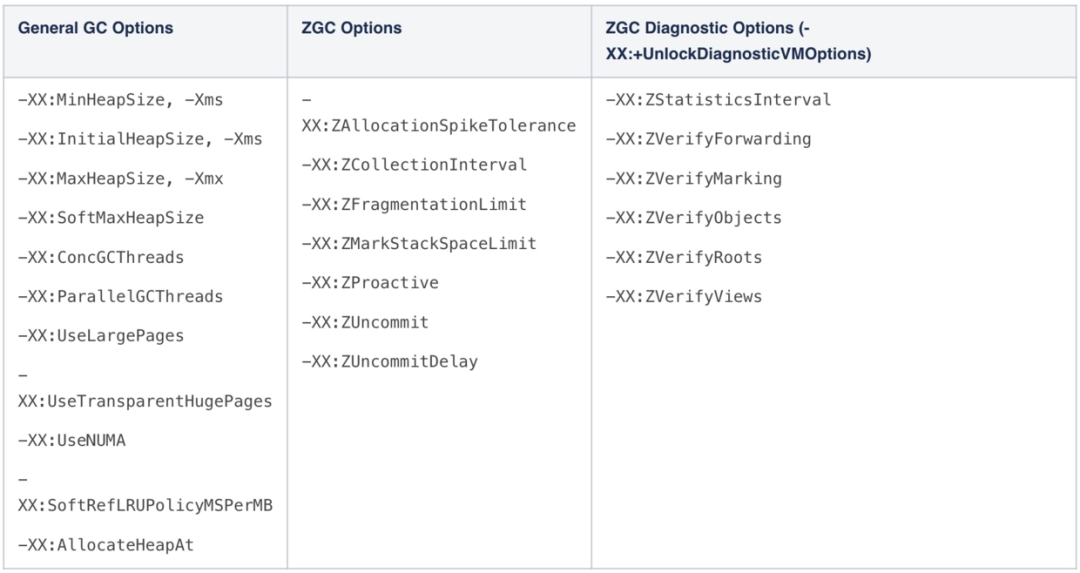


那么，判断对象是Bad Color还是Good Color的依据是什么呢？就是根据上一段提到的Colored Pointers的4个颜色位。当加上读屏障时，根据对象指针中这4位的信息，就能知道当前对象是Bad/Good Color了。

❝ 「扩展阅读」：既然低42位指针可以支持4T内存，那么能否通过预约更多位给对象地址来达到支持更大内存的目的呢？答案肯定是不可以。因为目前主板地址总线最宽只有48bit，4位是颜色位，就只剩44位了，所以受限于目前的硬件，ZGC最大只能支持16T的内存，JDK13就把最大支持堆内存从4T扩大到了16T。❞

### **ZGC tuning**

启用ZGC比较简单，设置JVM参数即可：-XX:+UnlockExperimentalVMOptions 「-XX:+UseZGC」。调优也并不难，因为ZGC调优参数并不多，远不像CMS那么复杂。它和G1一样，可以调优的参数都比较少，大部分工作JVM能很好的自动完成。下图所示是ZGC可以调优的参数：



* UseNUMA

ZGC默认是开启支持NUMA的，不过，如果JVM探测到系统绑定的是CPU子集，就会自动禁用NUMA。我们可以通过参数-XX:+UseNUMA显示启动，或者通过参数-XX:-UseNUMA显示禁用。如果运行在NUMA服务器上，并且设置-XX:+UseNUMA，那对性能提升是显而易见的。

* UseLargePages

配置ZGC使用large page通常就会得到更好的性能，比如在吞吐量、延迟、启动时间等方面。而且没有明显的缺点，除了配置过程复杂一点。因为它需要root权限，这也是默认并没有开启使用large page的原因。

* ConcGCThreads

ZGC是一个并发垃圾收集器，那么并发GC线程数就非常重要了。如果设置并发GC线程数越多，意味着应用线程数就会越少，这肯定是非常不利于应用系统稳定运行的。这个参数ZGC能自动设置，如果没有十足的把握。最好不要设置这个参数。

* ParallelGCThreads

这是个并行线程数，与上一个参数ConcGCThreads有所不同，ConcGCThreads表示GC线程和应用线程「并发」执行时GC线程数量。而ParallelGCThreads表示GC时STW阶段的「并行」GC线程数量（例如第一阶段的Root扫描），这时候只有GC线程，没有应用线程。笔者这里解释了JVM中「并发和并行的区别」，也是JVM中比较容易理解错误的地方。

* ZUncommit

掌握这个参数之前，我们先说一下JVM申请以及回收内存的行为。以前的垃圾回收器比如ParallelOldGC和CMS，只要JVM申请过的内存，即使发生了GC回收了很多内存空间，JVM也不会把这些内存归还给操作系统。这就会导致top命令中看到的RSS只会越来越高，而且一般都会超过Xmx的值。

不过，默认情况下，ZGC是会把不再使用的内存归还给操作系统的。这对于那些比较注意内存占用情况的应用和服务器来说，是很有用的。这种行为可以通过JVM参数\*\*-XX:-ZUncommit\*\*关闭。不过，无论怎么归还，JVM至少会保留Xms参数指定的内存大小，这就是说，当Xmx和Xms一样大的时候，这个参数就不起作用了。

和这个参数一起起作用的还有另一个参数：-「XX:ZUncommitDelay=sec」，默认300秒。这个参数表示不再使用的内存最多延迟多长时间才会被归还给操作系统。因为不再使用的内存不应该立即归还给操作系统，这样会造成频繁的归还和申请行为，所以通过这个参数来控制不再使用的内存需要经过多久的时间才归还给操作系统。