

G1收集器(-XX:+UseG1GC)

G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征.

■ Familiar with this ?

- Eden Young Generation
- Survivor
- Tenured Old Generation



G1将Java堆划分为多个大小相等的独立区域 (Region) , JVM最多可以有2048个Region。

一般Region大小等于堆大小除以2048, 比如堆大小为4096M, 则Region大小为2M, 当然也可以用参数"-XX:G1HeapRegionSize"手动指定Region大小, 但是推荐默认的计算方式。

G1保留了年轻代和老年代的概念, 但不再是物理隔阂了, 它们都是(可以不连续) Region的集合。

默认年轻代对堆内存的占比是5%, 如果堆大小为4096M, 那么年轻代占据200MB左右的内存, 对应大概是100个Region, 可以通过 "-XX:G1NewSizePercent" 设置新生代初始占比, 在系统运行中, JVM会不停的给年轻代增加更多的Region, 但是最多新生代的占比不会超过60%, 可以通过 "-XX:G1MaxNewSizePercent" 调整。年轻代中的Eden和Survivor对应的region也跟之前一样, 默认8:1:1, 假设年轻代现在有1000个region, eden区对应800个, s0对应100个, s1对应100个。

一个Region可能之前是年轻代, 如果Region进行了垃圾回收, 之后可能又会变成老年代, 也就是说Region的区域功能可能会动态变化。

G1垃圾收集器对于对象什么时候会转移到老年代跟之前讲过的原则一样, **唯一不同的是对大对象的处理**, G1有专门分配大对象的Region叫**Humongous区**, 而不是让大对象直接进入老年代的Region中。在G1中, 大对象的判定规则就是一个大对象超过了一个Region大小的50%, 比如按照上面算的, 每个Region是2M, 只要一个大对象超过了1M, 就会被放入Humongous中, 而且一个大对象如果太大, 可能会横跨多个Region来存放。

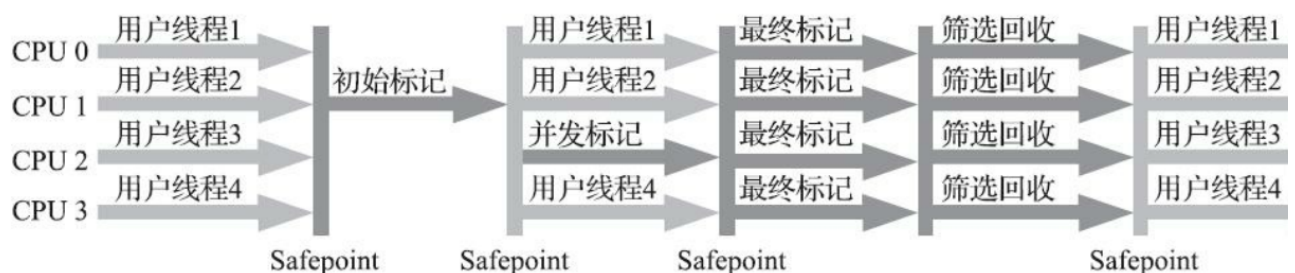
Humongous区专门存放短期巨型对象, 不用直接进老年代, 可以节约老年代的空间, 避免因为老年代空间不够的GC开销。

Full GC的时候除了收集年轻代和老年代之外, 也会将Humongous区一并回收。

G1收集器一次GC的运作过程大致分为以下几个步骤:

- **初始标记** (initial mark, STW) : 暂停所有的其他线程, 并记录下gc roots直接能引用的对象, **速度很快** ;

- **并发标记** (Concurrent Marking)：同CMS的并发标记
- **最终标记** (Remark, STW)：同CMS的重新标记
- **筛选回收** (Cleanup, STW)：筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间(可以用JVM参数 -XX:MaxGCPauseMillis指定)来制定回收计划，比如说老年代此时有1000个Region都满了，但是因为根据预期停顿时间，本次垃圾回收可能只能停顿200毫秒，那么通过之前回收成本计算得知，可能回收其中800个Region刚好需要200ms，那么就只会回收800个Region(**Collection Set**，要回收的集合)，尽量把GC导致的停顿时间控制在我们指定的范围内。这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。不管是年轻代或是老年代，回收算法主要用的是复制算法，将一个region中的存活对象复制到另一个region中，这种不会像CMS那样回收完因为有很多内存碎片还需要整理一次，G1采用复制算法回收几乎不会有太多内存碎片。(注意：CMS回收阶段是跟用户线程一起并发执行的，G1因为内部实现太复杂暂时没实现并发回收，不过到了Shenandoah就实现了并发收集，Shenandoah可以看成是G1的升级版)



G1收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的Region(这也就是它的名字Garbage-First的由来)，比如一个Region花200ms能回收10M垃圾，另外一个Region花50ms能回收20M垃圾，在回收时间有限情况下，G1当然会优先选择后面这个Region回收。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限时间内可以尽可能高的收集效率。

被视为JDK1.7以上版本Java虚拟机的一个重要进化特征。它具备以下特点：

- **并行与并发**：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短Stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程来执行GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
- **分代收集**：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。
- **空间整合**：与CMS的“标记--清理”算法不同，G1从整体来看是基于“**标记整理**”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿**：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立**可预测的停顿时间模型**，能让使用者明确指定在一个长度为M毫秒的时间片段(通过参数"-XX:MaxGCPauseMillis"指定)内完成垃圾收集。

毫无疑问，可以由用户指定期望的停顿时间是G1收集器很强大的一个功能，设置不同的期望停顿时间，可使得G1在不同应用场景中取得关注吞吐量和关注延迟之间的最佳平衡。不过，这里设置的“期望值”必须是符合实际的，不能异想天开，毕竟G1是要冻结用户线程来复制对象的，这个停顿时间再怎么低也得有个限度。

它默认的停顿目标为两百毫秒，一般来说，回收阶段占到几十到一百甚至接近两百毫秒都很正常，但如果我们把停顿时间调得非常低，譬如设置为二十毫秒，很可能出现的结果就是由于停顿目标时间太短，导致每次选出来的回收只占堆内存很小的一部分，收集器收集的速度逐渐跟不上分配器分配的速度，导致垃圾慢慢堆积。很可能一开始收集器还能从空闲的堆内存中获得一些喘息的时间，但应用运行时间一长就不行了，最终占满堆引发Full GC反而降低性能，所以通常把期望停顿时间设置为一两百毫秒或者两三百毫秒会比较合理的。

G1垃圾收集分类

YoungGC

YoungGC并不是说现有的Eden区放满了就会马上触发，G1会计算下现在Eden区回收大概要多久时间，如果回收时间远远小于参数 `-XX:MaxGCPauseMills` 设定的值，那么增加年轻代的region，继续给新对象存放，不会马上做Young GC，直到下一次Eden区放满，G1计算回收时间接近参数 `-XX:MaxGCPauseMills` 设定的值，那么就会触发Young GC

MixedGC

不是FullGC，老年代的堆占有率达到参数(`-XX:InitiatingHeapOccupancyPercent`)设定的值则触发，回收所有的Young和部分Old(根据期望的GC停顿时间确定old区垃圾收集的优先顺序)以及大对象区，正常情况G1的垃圾收集是先做MixedGC，主要使用复制算法，需要把各个region中存活的对象拷贝到别的region里去，拷贝过程中如果发现**没有足够的空region**能够承载拷贝对象就会触发一次Full GC

Full GC

停止系统程序，然后采用单线程进行标记、清理和压缩整理，好空闲出来一批Region来供下一次MixedGC使用，这个过程是非常耗时的。(Shenandoah优化成多线程收集了)

G1收集器参数设置

- XX:+UseG1GC:使用G1收集器
- XX:ParallelGCThreads:指定GC工作的线程数量
- XX:G1HeapRegionSize:指定分区大小(1MB~32MB，且必须是2的N次幂)，默认将整堆划分为2048个分区
- XX:MaxGCPauseMillis:目标暂停时间(默认200ms)
- XX:G1NewSizePercent:新生代内存初始空间(默认整堆5%)
- XX:G1MaxNewSizePercent:新生代内存最大空间
- XX:TargetSurvivorRatio:Survivor区的填充容量(默认50%)，Survivor区域里的一批对象(年龄1+年龄2+年龄n的多个年龄对象)总和超过了Survivor区域的50%，此时就会把年龄n(含)以上的对象都放入老年代
- XX:MaxTenuringThreshold:最大年龄阈值(默认15)
- XX:InitiatingHeapOccupancyPercent:老年代占用空间达到整堆内存阈值(默认45%)，则执行新生代和老年代的混合收集(**MixedGC**)，比如我们之前说的堆默认有2048个region，如果有接近1000个region都是老年代的region，则可能就要触发MixedGC了
- XX:G1MixedGCLiveThresholdPercent(默认85%) region中的存活对象低于这个值时才会回收该region，如果超过这个值，存活对象过多，回收的的意义不大。
- XX:G1MixedGCCountTarget:在一次回收过程中指定做几次筛选回收(默认8次)，在最后一个筛选回收阶段可以回收一会，然后暂停回收，恢复系统运行，一会再开始回收，这样可以让系统不至于单次停顿时间过长。
- XX:G1HeapWastePercent(默认5%): gc过程中空出来的region是否充足阈值，在混合回收的时候，对Region回收都是基于复制算法进行的，都是把要回收的Region里的存活对象放入其他Region，然后这个Region中的垃圾对象全部清理掉，这样的话在回收过程就会不断空出来新的Region，一旦空闲出来的Region数量达到了堆内存的5%，此时就会立即停止混合回收，意味着本次混合回收就结束了。

G1垃圾收集器优化建议

假设参数 `-XX:MaxGCPauseMills` 设置的值很大，导致系统运行很久，年轻代可能都占用了堆内存的60%了，此时才触发年轻代gc。

那么存活下来的对象可能就会很多，此时就会导致Survivor区域放不下那么多的对象，就会进入老年代中。

或者是你年轻代gc过后，存活下来的对象过多，导致进入Survivor区域后触发了动态年龄判定规则，达到了Survivor区域的50%，也会快速导致一些对象进入老年代中。

所以这里核心还是在于调节 `-XX:MaxGCPauseMills` 这个参数的值，在保证他的年轻代gc别太频繁的同时，还得考虑每次gc过后的存活对象有多少,避免存活对象太多快速进入老年代，频繁触发mixed gc.

什么场景适合使用G1

1. 50%以上的堆被存活对象占用
2. 对象分配和晋升的速度变化非常大
3. 垃圾回收时间特别长，超过1秒
4. 8GB以上的堆内存(建议值)
5. 停顿时间是500ms以内

每秒几十万并发的系统如何优化JVM

Kafka类似的支撑高并发消息系统大家肯定不陌生，对于kafka来说，每秒处理几万甚至几十万消息时很正常的，一般来说部署kafka需要用大内存机器(比如64G)，也就是说可以给年轻代分配个三四十G的内存用来支撑高并发处理，这里就涉及到一个问题了，我们以前常说的对于eden区的young gc是很快的，这种情况下它的执行还会很快吗？很显然，不可能，因为内存太大，处理还是要花不少时间的，假设三四十G内存回收可能最快也要几秒钟，按kafka这个并发量放满三四十G的eden区可能也就一两分钟吧，那么意味着整个系统每运行一两分钟就会因为young gc卡顿几秒钟没法处理新消息，显然是不行的。那么对于这种情况如何优化了，我们可以使用G1收集器，设置 `-XX:MaxGCPauseMills` 为50ms，假设50ms能够回收三到四个G内存，然后50ms的卡顿其实完全能够接受，用户几乎无感知，那么整个系统就可以在卡顿几乎无感知的情况下一边处理业务一边收集垃圾。

G1天生就适合这种大内存机器的JVM运行，可以比较完美的解决大内存垃圾回收时间过长的的问题。

ZGC收集器(-XX:+UseZGC)

参考文章：<https://wiki.openjdk.java.net/display/zgc/Main>
<http://cr.openjdk.java.net/~pliden/slides/ZGC-Jfokus-2018.pdf>

ZGC是一款JDK 11中新加入的具有实验性质的低延迟垃圾收集器，ZGC可以说源自于是Azul System公司开发的C4 (Concurrent Continuously Compacting Collector) 收集器。

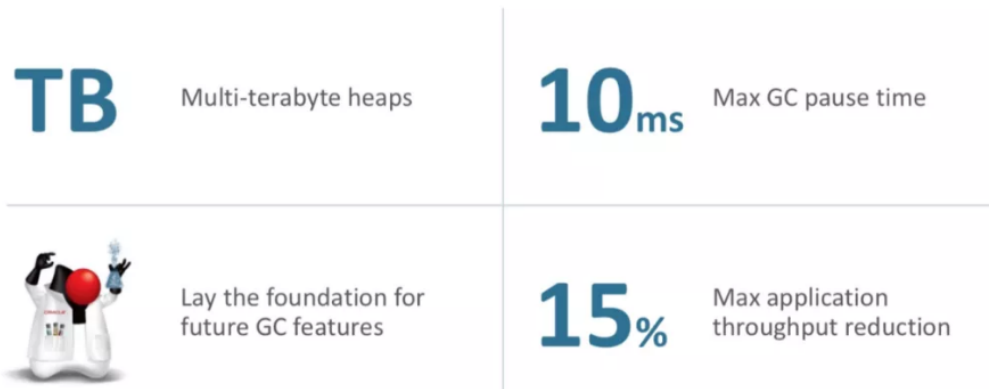
Supported Platforms

Platform	Supported	Since	Comment
Linux/x64	✓	JDK 11	
Linux/AArch64	✓	JDK 13	
macOS	✓	JDK 14	
Windows	✓	JDK 14	Requires Windows version 1803 (Windows 10 or Windows Server 2019) or later.

ZGC目标

如下图所示，ZGC的目标主要有4个：

Goals



- **支持TB量级的堆。**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。**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。

不分代(暂时)

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。

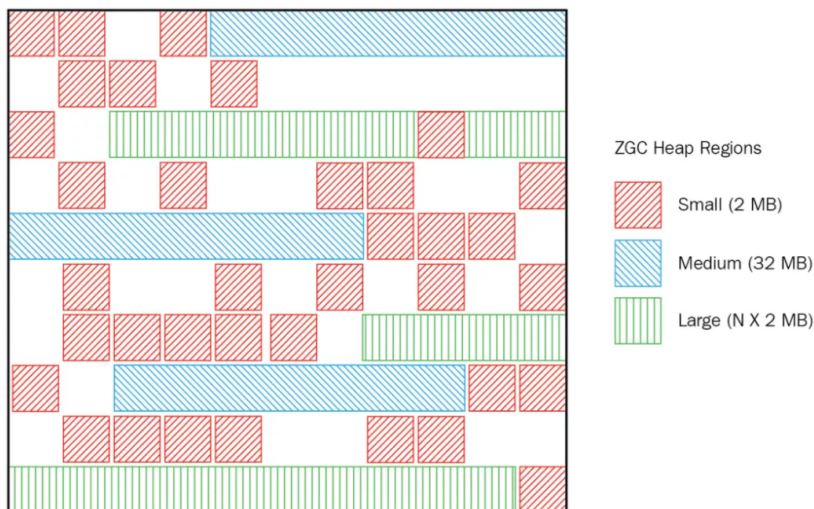
那么为什么ZGC就不分代呢？因为分代实现起来麻烦，作者就先实现出一个比较简单可用的单代版本，后续会优化。

ZGC内存布局

ZGC收集器是一款基于Region内存布局的，暂时不设分代的，使用了**读屏障**、**颜色指针**等技术来实现可并发的标记-整理算法的，以低延迟为首要目标的一款垃圾收集器。

ZGC的Region可以具有如图3-19所示的大、中、小三类容量：

- 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或以上的对象。每个大型Region中
 - 只会存放一个大对象，这也预示着虽然名字叫作“大型Region”，但它的实际容量完全有可能小于中型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的一种处理动作，用于复制对象的收集器阶段，稍后会介绍到）的，因为复制一个大对象的代价非常高昂。



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：

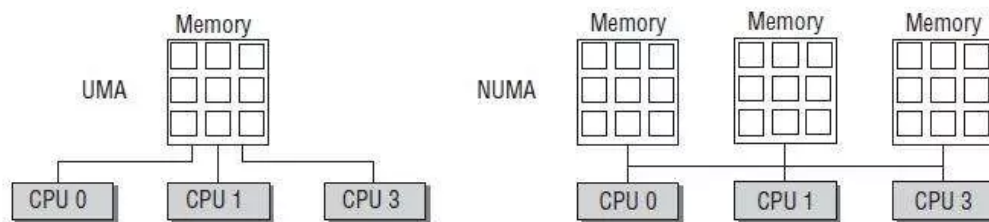


Figure 3-1: UMA and NUMA systems.

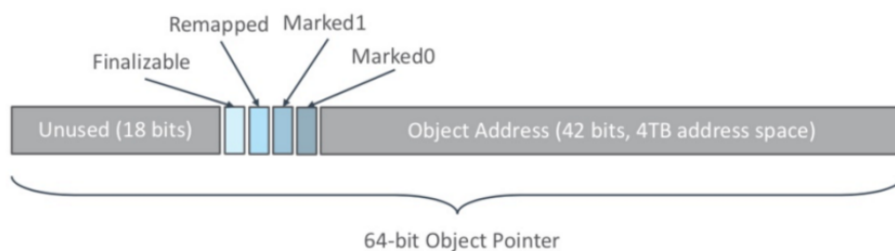
服务器的NUMA架构在中大型系统上一直非常盛行，也是高性能的解决方案，尤其在系统延迟方面表现都很优秀。ZGC是能自动感知NUMA架构并充分利用NUMA架构特性的。

颜色指针

Colored Pointers，即颜色指针，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。

Colored Pointers

Layout on x86_64



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，此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）：

为什么有2个mark标记？

每一个GC周期开始时，会交换使用的标记位，使上次GC周期中修正的已标记状态失效，所有引用都变成未标记。

GC周期1：使用mark0, 则周期结束所有引用mark标记都会成为01。

GC周期2：使用mark1, 则期待的mark标记10，所有引用都能被重新标记。

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）。

颜色指针的三大优势：

1. 一旦某个Region的存活对象被移走之后，这个Region立即就能够被释放和重用掉，而不必等待整个堆中所有指向该Region的引用都被修正后才能清理，这使得理论上只要还有一个空闲Region，ZGC就能完成收集。
2. 颜色指针可以大幅减少在垃圾收集过程中内存屏障的使用数量，ZGC只使用了读屏障。
3. 颜色指针具备强大的扩展性，它可以作为一种可扩展的存储结构用来记录更多与对象标记、重定位过程相关的数据，以便日后进一步提高性能。

读屏障

之前的GC都是采用Write Barrier，这次ZGC采用了完全不同的方案读屏障，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：

Load Barrier

```
Object o = obj.fieldA;           // Loading an object reference from heap
<load barrier needed here>
Object p = o;                   // No barrier, not a load from heap
o.doSomething();                // No barrier, not a load from heap
int i = obj.fieldB;             // No barrier, not an object reference
```

“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：

Load Barrier

```
mov    0x20(%rax), %rbx           // Object o = obj.fieldA;
test   %rbx, (0x16)%r15           // Bad color?
jnz    slow_path                  // Yes -> Enter slow path and
                                  // mark/relocate/remap, adjust
                                  // 0x20(%rax) and %rbx
```

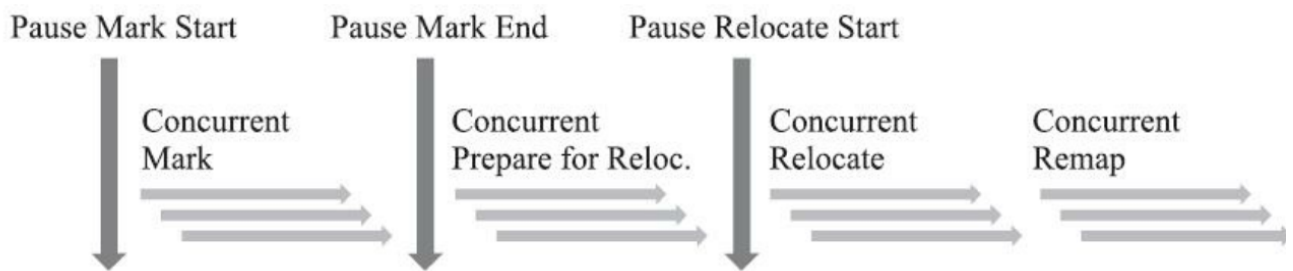
~4% execution overhead on SPECjbb®2015

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。

PS：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运作过程

ZGC的运作过程大致可划分为以下四个大的阶段：



- **并发标记 (Concurrent Mark)**：与G1一样，并发标记是遍历对象图做可达性分析的阶段，它的初始标记(Mark Start)和最终标记(Mark End)也会出现短暂的停顿，与G1不同的是，ZGC的标记是在指针上而不是在对象上进行的，标记阶段会更新染色指针中的Marked 0、Marked 1标志位。
- **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)。ZGC每次回收都会扫描所有的Region，用范围更大的扫描成本换取省去G1中记忆集的维护成本。
- **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，并为重分配集中的每个Region维护一个**转发表 (Forward Table)**，记录从旧对象到新对象的转向关系。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之中，如果用户线程此时并发访问了位于重分配集中的对象，这次访问将会被预置的内存屏障(读屏障)所截获，然后立即根据Region上的转发表记录将访问转发到新复制的对象上，并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。

1 ZGC的颜色指针因为“自愈”(Self-Healing)能力，所以只有第一次访问旧对象会变慢，一旦重分配集中某个Region的存活对象都复制完毕后，

2 这个Region就可以立即释放用于新对象的分配，但是转发表还得留着不能释放掉，因为可能还有访问在使用这个转发表。

- **并发重映射 (Concurrent Remap)**：重映射所做的就是修正整个堆中指向重分配集中旧对象的所有引用，但是ZGC中对象引用存在“自愈”功能，所以这个重映射操作并不是很迫切。ZGC很巧妙地把并发重映射阶段要做的工作，合并到了下一次垃圾收集循环中的并发标记阶段里去完成，反正它们都是要遍历所有对象的，这样合并就节省了一次遍历对象图的开销。一旦所有指针都被修正之后，原来记录新旧对象关系的转发表就可以释放掉了。

ZGC存在的问题

ZGC最大的问题是**浮动垃圾**。ZGC的停顿时间是在10ms以下，但是ZGC的执行时间还是远远大于这个时间的。假如ZGC全过程需要执行10分钟，在这个期间由于对象分配速率很高，将创建大量的新对象，这些对象很难进入当次GC，所以只能在下次GC的时候进行回收，这些只能等到下次GC才能回收的对象就是浮动垃圾。

1 ZGC没有分代概念，每次都需要进行全堆扫描，导致一些“朝生夕死”的对象没能及时的被回收。

解决方案

目前唯一的办法是增大堆的容量，使得程序得到更多的喘息时间，但是这个也是一个治标不治本的方案。如果需要从根本上解决这个问题，还是需要引入分代收集，让新生对象都在一个专门的区域中创建，然后专门针对这个区域进行更频繁、更快的收集。

ZGC参数设置

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：

General GC Options	ZGC Options	ZGC Diagnostic Options (-XX:+UnlockDiagnosticVMOptions)
-XX:MinHeapSize, -Xms	-	-XX:ZStatisticsInterval
-XX:InitialHeapSize, -Xms	XX:ZAllocationSpikeTolerance	-XX:ZVerifyForwarding
-XX:MaxHeapSize, -Xmx	-XX:ZCollectionInterval	-XX:ZVerifyMarking
-XX:SoftMaxHeapSize	-XX:ZFragmentationLimit	-XX:ZVerifyObjects
-XX:ConcGCThreads	-XX:ZMarkStackSpaceLimit	-XX:ZVerifyRoots
-XX:ParallelGCThreads	-XX:ZProactive	-XX:ZVerifyViews
-XX:UseLargePages	-XX:ZUncommit	
-	-XX:ZUncommitDelay	
XX:UseTransparentHugePages		
-XX:UseNUMA		
-		
XX:SoftRefLRUPolicyMSPerMB		
-XX:AllocateHeapAt		

ZGC触发时机

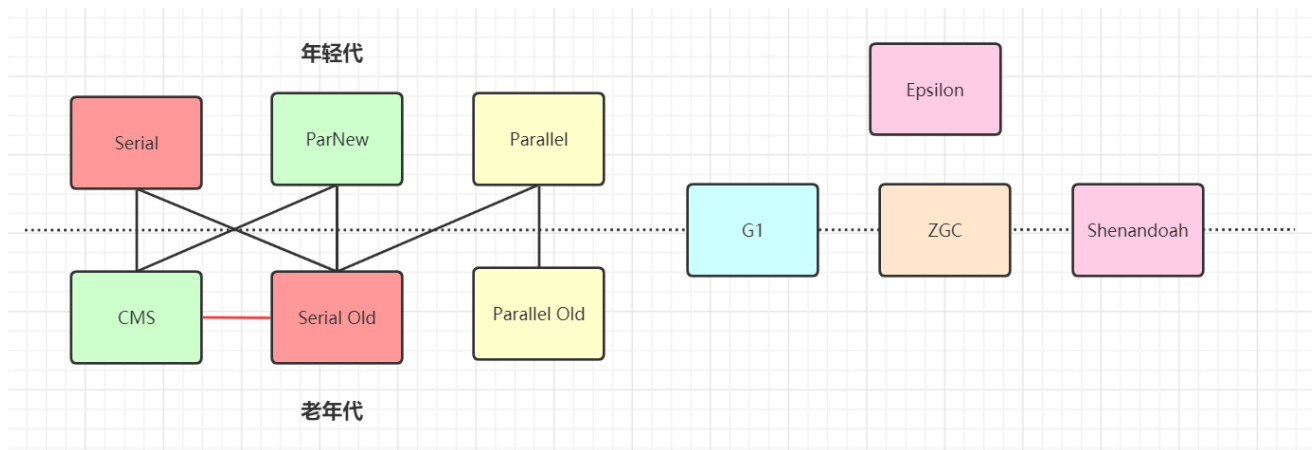
ZGC目前有4中机制触发GC：

- 定时触发，默认为不使用，可通过ZCollectionInterval参数配置。
- 预热触发，最多三次，在堆内存达到10%、20%、30%时触发，主要统计GC时间，为其他GC机制使用。
- 分配速率，基于正态分布统计，计算内存99.9%可能的最大分配速率，以及此速率下内存将要耗尽的时间点，在耗尽之前触发GC（耗尽时间 - 一次GC最大持续时间 - 一次GC检测周期时间）。
- 主动触发，（默认开启，可通过ZProactive参数配置）距上次GC堆内存增长10%，或超过5分钟时，对比距上次GC的间隔时间跟（49 * 一次GC的最大持续时间），超过则触发。

如何选择垃圾收集器

1. 优先调整堆的大小让服务器自己来选择
2. 如果内存小于100M，使用串行收集器
3. 如果是单核，并且没有停顿时间的要求，串行或JVM自己选择
4. 如果允许停顿时间超过1秒，选择并行或者JVM自己选
5. 如果响应时间最重要，并且不能超过1秒，使用并发收集器
6. 4G以下可以用parallel，4-8G可以用ParNew+CMS，8G以上可以用G1，几百G以上用ZGC

下图有连线的可以搭配使用



JDK 1.8默认使用 Parallel(年轻代和老年代都是)

JDK 1.9默认使用 G1

安全点与安全区域

安全点就是指代码中一些特定的位置,当线程运行到这些位置时它的状态是确定的,这样JVM就可以安全的进行一些操作,比如GC等, 所以GC不是想什么时候做就立即触发的, 是需要等待所有线程运行到安全点后才能触发。

这些特定的安全点位置主要有以下几种:

1. 方法返回之前
2. 调用某个方法之后
3. 抛出异常的位置
4. 循环的末尾

大体实现思想是当垃圾收集需要中断线程的时候, 不直接对线程操作, 仅仅简单地设置一个标志位, 各个线程执行过程时会不停地主动去轮询这个标志, 一旦发现中断标志为真时就自己在最近的安全点上主动中断挂起。轮询标志的地方和安全点是重合的。

安全区域又是什么?

Safe Point 是对正在执行的线程设定的。

如果一个线程处于 Sleep 或中断状态, 它就不能响应 JVM 的中断请求, 再运行到 Safe Point 上。

因此 JVM 引入了 Safe Region。

Safe Region 是指在一段代码片段中, **引用关系不会发生变化**。在这个区域内的任意地方开始 GC 都是安全的。

文档: 06-VIP-垃圾收集器G1&ZGC详解

1 <http://note.youdao.com/noteshare?id=74c84c1ab69727580cdb2646d35674fc&sub=E9BACBC06D254D93904AB0A7CCF38A86>