# [G1垃圾收集器](https://www.cnblogs.com/wjh123/p/11146195.html)

一、以前收集器的特点

* 年轻代和老年代是各自独立且连续的内存块
* 年轻代收集器使用 eden + S0 + S1 进行复制算法
* 老年代收集必须扫描整个老年代区域
* 都是以尽可能的少而快速地执行 GC 为设计原则

二、G1 是什么

* G1 是一种面向服务端的垃圾收集器，应用在多核处理器和大容量内存环境中，在实现高吞吐量的同时，尽可能的满足垃圾收集器的暂停时间要求。
* 像 CMS 收集器一样，能与应用程序线程并发执行，

         整理空闲空间更快，

         需要更多的时间来预测 GC 停顿时间，

         不希望牺牲大量的吞吐性能，不需要更大的 JAVA Heap。

* G1 收集器的设计目的是取代 CMS 收集器，同 CMS 相比，

         G1 垃圾收集器是一个有整理内存过程的垃圾收集器，不会产生很多内存碎片。

         G1 的 Stop The World 更可控，G1 在停顿上添加了预测机制，用户可以指定期望的停顿时间。

* G1 是在 2012 年才在 jdk.1.7u4 中可以呀用，在 jdk9 中将 G1 变成默认垃圾收集器来代替 CMS。它是一款面向服务应用的收集器。

        主要改变是 Eden、Survivor 和 Tenured 等内存区域不再是连续的，而是变成了一个个大小一样的 region，每个 region 从 1M 到 32M 不等，一个 region 有可能属于 Eden、Survivor 或者 Tenured 内存区域。

三、特点

* G1 能充分利用多 CPU、多核环境硬件优势，尽量缩短 STW。
* G1 整体采用标记-整理算法，局部是通过是通过复制算法，**不会产生内存碎片**。
* 宏观上看 G1 之中不再区分年轻代和老年代，被内存划分为多个独立的子区域。
* G1 收集器里面讲整个的内存区域混合在一起，**但其本身依然在小范围内要进行年轻代和老年代的区分**。保留了新生代和老年代，但她们不在是物理隔离，而是一部分 Region 的集合且不需要 Region 是连续的，也就是说依然会采用不同的 GC 方式来处理不同的区域。
* G1 虽然也是分代收集器，但整个内存分区不存在物理上的年轻代和老年代的区别，也不需要完全独立的 Survivor to space 堆做复制准备。G1 只有逻辑上的分代概念，或者说每个分区都可能随 G1 的运行在不同代之间前后切换。

四、底层原理

1、Region 区域化垃圾收集器：最大好处是化整为零，避免全内存扫描，只需要按照区域来进行扫描即可。

2、回收步骤：

3、四步过程：

* 初始标记：标记一下GC Roots能直接关联到的对象，需要停顿线程，但耗时很短
* 并发标记：是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行
* 最终标记：修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录
* 筛选回收：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停顿时间来制定回收计划

五、常用配置参数：

-XX:UseG1GC

-XX:G1HeapRegionSize=n:设置G1区域的大小，值是2的幂

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

六、与CMS相比的优势

* 没有内存碎片
* 可以精确控制停顿

# [G1收集器](https://www.cnblogs.com/chenpt/p/9803298.html)

一款面向服务端应用的垃圾收集器。

**特点如下：**

并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿时间。部分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程序继续运行。

分代收集：G1能够独自管理整个Java堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

空间整合：G1运作期间不会产生空间碎片，收集后能提供规整的可用内存。

可预测的停顿：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毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

**G1为什么能建立可预测的停顿时间模型？**

因为它有计划的避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。这样就保证了在有限的时间内可以获取尽可能高的收集效率。

**G1与其他收集器的区别**：

其他收集器的工作范围是整个新生代或者老年代、G1收集器的工作范围是整个Java堆。在使用G1收集器时，它将整个Java堆划分为多个大小相等的独立区域（Region）。虽然也保留了新生代、老年代的概念，但新生代和老年代不再是相互隔离的，他们都是一部分Region（不需要连续）的集合。

**G1收集器存在的问题：**

Region不可能是孤立的，分配在Region中的对象可以与Java堆中的任意对象发生引用关系。在采用可达性分析算法来判断对象是否存活时，得扫描整个Java堆才能保证准确性。其他收集器也存在这种问题（G1更加突出而已）。会导致Minor GC效率下降。

**G1收集器是如何解决上述问题的？**

**采用Remembered Set来避免整堆扫描**。G1中每个Region都有一个与之对应的Remembered Set，虚拟机发现程序在对Reference类型进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用对象是否处于多个Region中（即检查老年代中是否引用了新生代中的对象），如果是，便通过CardTable把相关引用信息记录到被引用对象所属的Region的Remembered Set中。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆进行扫描也不会有遗漏。

**如果不计算维护 Remembered Set 的操作，G1收集器大致可分为如下步骤：**

**初始标记**：仅标记GC Roots能直接到的对象，并且修改TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象。（需要线程停顿，但耗时很短。）

**并发标记**：从GC Roots开始对堆中对象进行可达性分析，找出存活对象。（耗时较长，但可与用户程序并发执行）

**最终标记**：为了修正在并发标记期间因用户程序执行而导致标记产生变化的那一部分标记记录。且对象的变化记录在线程Remembered Set  Logs里面，把Remembered Set  Logs里面的数据合并到Remembered Set中。（需要线程停顿，但可并行执行。）

**筛选回收**：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停顿时间来制定回收计划。（可并发执行）

**G1收集器运行示意图：**



--