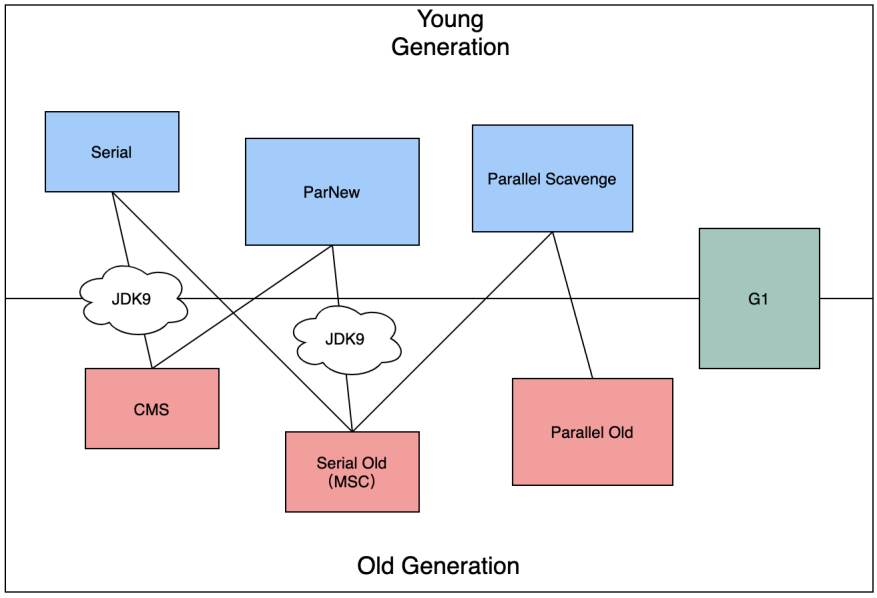
JVM GC篇 — 垃圾收集器（上）

前言

前面一小节我们介绍了介绍了GC的一般性原理和垃圾回收算法，通过对前面知识的学习，我们基本掌握了垃圾收集的一般理论知识。但是理论终究知识在之上，距离真正的实践还有一定的差距。前面我们提到了第一个带有内存动态分配和垃圾收集的编程语言并不是 Java 而是1960年诞生的Lisp。当然这一小节中我们并不会梳理 Lisp 实现的垃圾收集器。这一小节我们会先从经典垃圾收集器入手，梳理各个垃圾收集器的特性、优缺点和适用场景。当然随着垃圾收集器的效率的不断提升其实现的复杂度也在不断提升。我们会主要的篇幅去介绍CMS和G1原理及部分实现细节，升华对垃圾收集器的理解。今天是六一儿童节，各位朋友坐好来，我们的梳理开始了。😂

经典垃圾收集器

这里的提到的“经典”并非出于情怀，只是他们的确挺经典，是出于讨论范围的限定。我们这里讨论的是在 JDK 7 Update 4 之后（这个版本中正式提供了商用的 G1 收集器，此前 G1 仍处于实验状态）、JDK 11 正式发布之前，OracleJDK 中提供的 HotSpot 虚拟机所包含的全部可用的垃圾收集器。使用“经典”来描述是为了区分开目前几款正处于实验状态，但执行效果却具有革命性改进的高性能低延迟的垃圾回收器。这些经典的垃圾回收器虽然已经不算是最先进的技术，但是这些经典的垃圾回收器都在生产环境经过千锤百炼，足够成熟，基本上可以认为现在到未来几年内可以在生产环境上放心使用的垃圾收集器，因此掌握他们的特点和配置细节很重要。下图是各个经典垃圾收集器之间的关系。



上面这张图展现了七种作用于不同分代的垃圾回收器，图中垃圾收集器所处的位置则表示他们工作的区域即年轻代或老年代。其中上面`Young Generation`表示年轻代，下面的`Old Generation`则代表老年代。如果两个垃圾回收器之间存在连线，则说明他们两个可以搭配使用。

> ⚠️ 需要注意的是，这个关系并不是一成不变的，由于维护和兼容性测试的成本，在JDK8时将Serial+CMS、ParNew+Serial Old这两个组合声明为废弃（JEP173），并在 JDK 9 中完全取消了这些组合的支持（JEP214）。

接下来我们会围绕着这些垃圾回收器的目标、特性、原理和使用场景来了解与分析，并且会着重分析一些CMS和G1的实现细节。这里需要需要先明确一个观点，**目前在 JVM 中没有一个在任何一个场景都表现的很完美的万能垃圾回收器**，如果这样的垃圾回收器存在的话，也就不会出现 HotSpot 中这么多垃圾回收器并存的情况了。

Serial收集器（Serial + SerialOld）

Serial 收集器是最基础、也是历史最悠久的垃圾收集器。**曾经（JDK1.3.1之前）是HotSpot收集器的唯一选择**。从这个名字也不难猜到，这个垃圾收集器是一个`单线程工作`的垃圾收集器。“单线程”并不仅仅意味着他只会是会用一个处理器或是一个收集线程去完成垃圾收集，它更是要强调**它在进行垃圾回收的时候，必须暂停其他的工作线程（STW）**直到它工作结束。Serial垃圾收集器在年轻代工作使用的`标记复制算法（mark-copy）`，在工作在老年代的是 SerialOld 垃圾收集器使用`标记清除整理算法（mark-sweep-compact）`，它的特性和Serial一致都是`单线程`垃圾收集器。但是这样看似落后的垃圾收集器并没有被废弃。单线程真的就没有优点吗？实际上 Serial 依旧是HotSpot虚拟机运行在客户端模式下的默认垃圾收集器。单线程垃圾收集器虽然和多线程垃圾收集器相比显得低效，但是从另外一个角度来看意味着**简单可靠**，对于内存有限的环境来说**它又是内存消耗最少**的。对于只有单核的环境活着CPU核心数少的环境来说，减少上下文的切换它又是高效的。因此对于**内存小、CPU核心数少、不会频繁发生垃圾收集的环境，Serial收集器会是一个好的选择。**使用下面的JVM参数即可使用 Serial 垃圾收集器。

```Bash
-XX:+UseSerialGC
```

Parallel收集器（Parallel Scavenge + Parallel Old）

Parallel 收集器也叫做`并行收集器`，他能**并行的进行垃圾收集**。`Parallel Scavenge`是 Parallel 在新生代运行的垃圾收集器，采用的同样也是`标记复制算法（mark-copy）`，在老年代使用的是 Parallel Old 收集器，使用的也是`标记清除整理算法（mark-sweep-compact）`。`Parallel`和`Serial`一样进行垃圾收集时，**Parallel 在新生代和老年代都会触发STW，但是 Parallel 是采用并行标记，并行垃圾收集，因此STW时间和垃圾收集时间会相较于Serial短很多，垃圾收集速递大幅度提高。**后面我们会介绍 CMS 垃圾收集器，CMS 新生代使用的是 ParNew 这也是一款并行的垃圾收集器，那

他们区别是什么呢？Parallel 侧重的更多是可控制`吞吐量（Throughput）`，而CMS侧重更短的STW时间。Parallel也是`JDK8的默认垃圾回收器`

$$吞吐量 = \frac{运行用户代码时间}{运行用户代码时间 + 运行垃圾收集时间}$$

Parallel 适用多核心处理处理器，主要目的是增加吞吐量。因为对系统资源的有效使用能达到更高的吞吐量。在GC期间**所有的CPU内核都在并行清理垃圾**，所以总暂停时间更短；在两次GC周期的间隔期，没有GC线程在运行，不会消耗任何系统资源。使用下面的JVM参数即可以开启ParallelGC。

```Bash
-XX:+UseParallelGC
-XX:+UseParallelOldGC
-XX:+UseParallelGC -XX:+UseParallelOldGC
```

`Parallel Scavenge`提供了两个参数用于精准控制吞吐量，分别是控制最大垃圾收集时间的`-XX:MaxGCPauseMillis`参数，以及直接设置吞吐量大小的`-XX:GCTimeRatio`。其中`-XX:MaxGCPauseMillis`设置一个大于0的毫秒数，垃圾收集器尽量把垃圾收集时间控制在这个毫秒数内，如果设置过小那么收集器的吞吐量将下降。`-XX:GCTimeRatio`这个参数设置一个大于0小于100的整数，**也就是垃圾收集时间占总时间的比率。这个参数设置为N，那么用户代码执行时间与总执行时间之比为`N:N+1`**。例如`-XX:GCTimeRatio=19`，那么垃圾收集时间占用总时间的比例为5%(1/(1+ 19))。默认值为99，即1%垃圾回收时间占用。Parallel垃圾收集器是一款高吐的垃圾收集器，那它有什么缺点吗？Parallel垃圾收集器工作期间必须暂停其他的工作线程（STW），多个垃圾回收工作线程同时工作，虽然垃圾清理效率很高，但是和其他追求短暂STW的垃圾收集器（CMS、G1、ZGC、Shenandoah）相比还是偏长。并且Parallel收集器的**清理速度会随着堆的增大而变慢。**

> 什么是并行和什么又是并发，他们之间有什么区别？
> 并行是指利用多个处理器或者多核心处理器同时处理多个不同的任务。
> 并发是指一个处理器线程在多个不同的任务之间来回切换来实现“同时执行”。

CMS（ParNew + CMS）

CMS GC的官方名称为“**Mostly Concurrent Mark and Sweep Garbage Collector**”（最大并发标记清理垃圾收集器）。其中对年轻代使用的是并行的STW方式的标记复制算法（mark-copy），对老年代使用的是并发标记清除算法（标记-清除）。其中CMS的核心是放在老年代，而年轻代使用的 ParNew 垃圾收集器。**ParNew 收集器实质上就是 Serial 收集器的多线程版本**，除了使用多条线程进行垃圾收集之外，其余的行为包括 Serial 垃圾收集器的所有的控制参数、收集算法、STW、对象分配规则、回收策略等都和Serial收集器完全一致。虽然ParNew在不少运行在服务端HotSpot在新生代垃圾收集器的首选。其中一个非常重要的重要的原因就是，**ParNew 是唯一一个除了Serial 之外能和CMS搭配工作的垃圾收集器。**使用下面的JVM参数即可启用CMS：

```Bash
+XX:+UseConcMarkSweepGC #同时新生代使用ParNew垃圾收集器。
```

CMS的设计目标**是为了避免在老年代进行垃圾收集时出现长时间卡顿。**

> 之前我考虑过一个问题，CMS 减少的是老年代的卡顿，但是并没有减少新生代的垃圾收集的卡顿。老年代的GC次数远远少于新生代，为了减少老年代设计的CMS是否有意义？
> 当然是有意义的。因为新生代使用的标记复制算法并且存活对象少，在标记和复制阶段的耗时都非常的短，这个时间基本上可以忽略不计。但是在之前的算法中，老年代使用的是标记清除整理算法，并且存活对象很多。这样会有大量的时间浪费在整理上。所以这个阶段的STW时间是很长的。

避免老年代收集器长时间的卡顿，通过两种手段来达成此目标。

- **不对老年代进行整理，而是使用空闲列表（free-list）来管理内存空间的回收。**
- **在标记清理阶段的大部分工作和并发线程一起完成。**

在并发标记阶段并没有明显的应用线程暂停，但是值得注意的是它仍然和应用线程争抢 CPU 时间。默认情况下，**CMS使用的是并发线程数等于 (CPU 核心数 + 3)/4。**

> 这两个手段很好理解。不对老年代进行整理，也就避免了老年代整理带来的长时间STW。对于之前垃圾收集器不管是 Serial 还是 Parallel，在标记阶段或者清理整理阶段都会进行STW，CMS 采用的手段是通过一起并发执行来消除这个阶段的 STW。在解决问题的思路上可以说是很切中要点了。

如果服务器是多核CPU，并且主要调优目标是降低 GC 停顿导致的系统延迟，那么使用 CMS是一个很明知的选择。通过减少每一次GC停顿的时间，能很大程度上改善用户体验。但是**如果CPU资源不是很充足或是受限制的情况下，CMS的吞吐量会出现比较明显的问题**。对于绝大部分系统，CMS 和 Parallel 的吞吐和延迟的差别并不大。

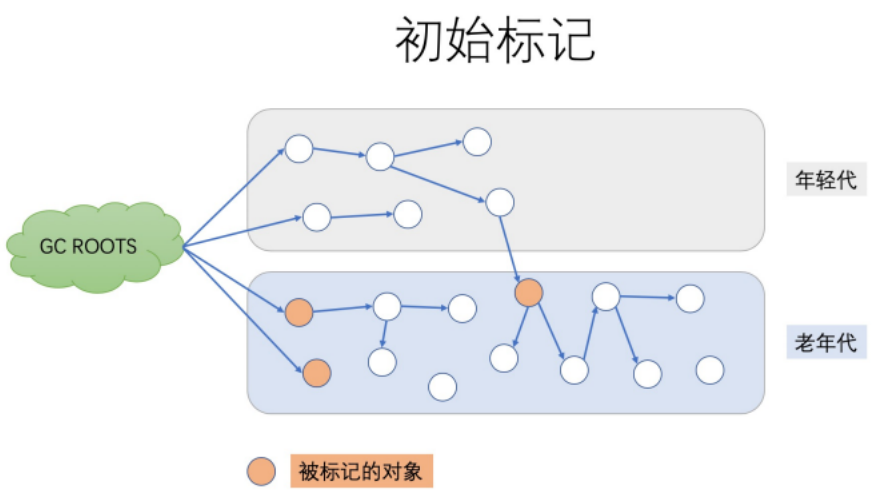
CMS 垃圾收集阶段过程

上面提到通过两个主要的手段来达成垃圾收集时的长时间卡顿的目标。但是我们前面提到如果老年代只标记清理不整理这样会产生很多不可用的碎片空间。我们前面还提到并发清理的时候，为了避免引用的变化，其他工作线程都需要进入安全点等待直至垃圾收集结束。那 CMS 是怎样巧妙地处理这些矛盾点呢？我们一起来看看 CMS GC 的几个阶段。

阶段1：初始标记阶段（inital mark）（STW）

这个阶段**伴随STW暂停**。初始标记的目标是标记所有的根对象，包括根对象直接引用的对象，以及被新生代中所有存活对象所引用的对象（老年代单独回收）。

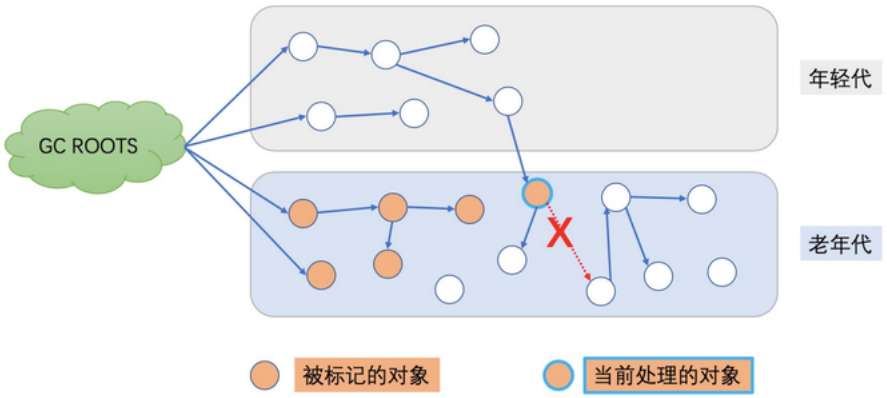
> 我们前面提到对新生代进行垃圾收集标记阶段，会把脏卡中的对象（老年代指向新生代引用的对象）加入GC Roots，这里也是类似的处理方向，只不过方向反过来了，把新生代中指向老年代的对象加入GC Roots。



阶段2：并发标记（Concurrent Mark）

在这个阶段，CMS GC遍历老年代，标记所有的存活对象，从前一阶段的“Initial Mark”找到的根对象开始算起。“并发标记阶段，就是与应用程序同时运行，不用暂停的阶段（这个阶段没有STW）。⚠️请注意并非所有的老年代中存活的对象都在此阶段被标记，因为在**标记过程中对象的引用关系还在发生这变化。**

并发标记



在上面的图中，当前处理对象的一个引用被应用程序给断开了，即这个对象的应用关系发生了变化。那我们要怎么处理这种变化的引用关系呢？先标记脏卡。

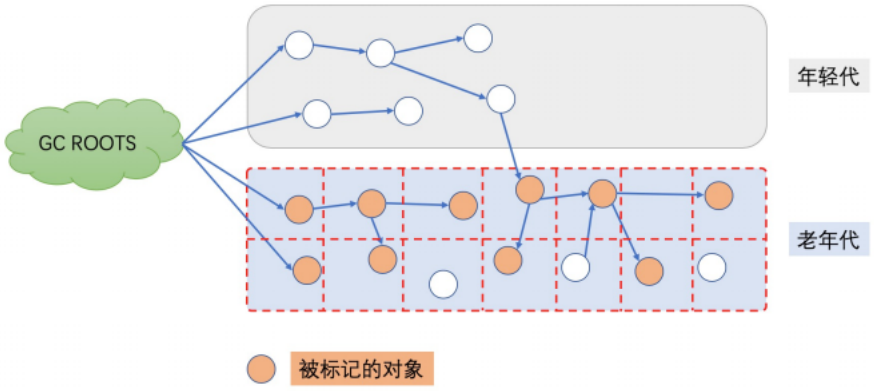
阶段3：并发预处理（Concurrent Preclean）

在这个阶段不需要STW停顿，因为前面一个阶段并发标记与程序一起运行，可能有一些对象的引用关系已经发生了变化。如果在并发标记中引用发生了变化，那么JVM 将通过 Card 的方式将发生改变的区域标记为“脏”卡，这是老年代清理过程中的卡片标记，并发标记在前面一小节有详细的介绍。**这个阶段还会处理在执行并发标记阶段新进入老年代的对象（新晋升的对象）。**

> 这里的脏卡和我们前面在一般原理和垃圾收集中讲到的脏卡有一些不一样，前面年轻代标记阶段提到的脏卡，是为了避免每次新生代标记的时候避免扫描整个老年代，而是通过每次复制之后修改地址时，顺带标记出老年代对新生代有引用的对象所在的“卡”为脏卡。下次新生代GC时，将脏卡中的对象加入 GC Roots即可。
>
> 这里的CMS中的脏卡是指在CMS 并发标记的过程中引用发生变化卡，在后续垃圾收集过程中进行特别处理。
>
> 共同点都是都是JVM老年代的卡片标记技术，标记某个内存块，为后续垃圾收集操作提供标识。

在预清理阶段，这些**脏对象会被统计出来，他们所引用的对象也会被标记**，此阶段完成后，用以标记的 card 也会被清空。

并发预清理完成



这个阶段还会进行一些必要的细节处理，还会为`Final Remark`做一些准备工作。

阶段4：可取消的并发预清理（Concurrent Abortable Preclean）

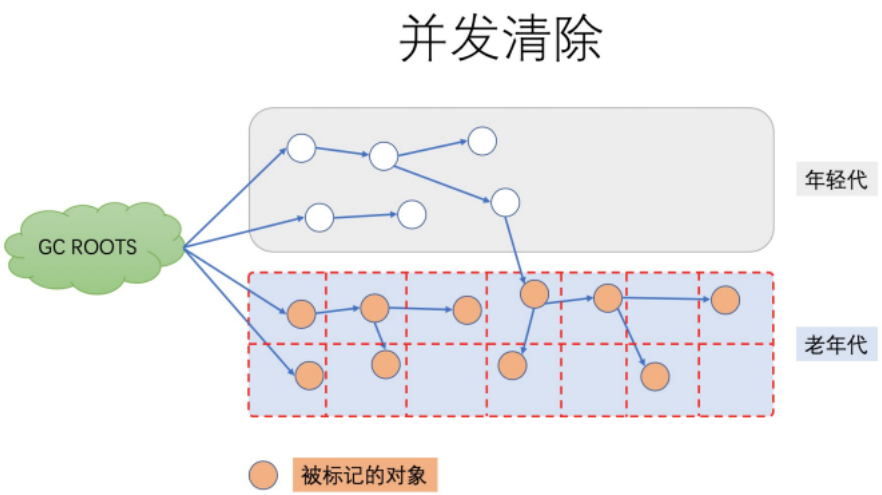
这个阶段也不会STW，这个阶段在 STW的Final Remark 之前尽可能地多做一些工作。这个阶段可显著影响STW停顿持续时间。

阶段5：最终标记阶段（Final Remark）（STW）

最终标记阶段是本次GC时间中的第二次（也是最后一次）**STW停顿**。本阶段的目标是完成老年代中所有存活对象的标记。因为之前的预清理阶段是并发执行的，有可能GC线程跟不上应用程序的修改速度。所以需要一次STW暂停来说处理各种复杂的情况。通常 CMS 会尝试在年轻代尽可能空的情况下执行 Final Remark，以免连续触发多次 STW事件。在以上5个阶段完成之后，老年代中的所有存活对象都被标记了，然后GC将清除所有不使用的对象来回收老年代空间。

阶段6：并发清除（Concurrent Sweep）

这个阶段不需要 STW停顿，JVM在此阶段清理不再使用的对象，并回收他们占用的内存空间。



阶段7：并发重置（Concurrent Reset）

这个阶段与应用程序并发执行，重置 CMS 算法相关内置数据，并为下次GC循环做好准备。

汇总四阶段

有的书上我看介绍 CMS 的垃圾回收步骤只有5步甚至是4步，书中梳理出来的4步分别为：

- 初始标记：仅仅标识 GC Root 能直接关联到的对象。（STW）
- 并发标记：由初始标记的关联对象遍历标记整个对象图的过程。
- 重新标记：修正因为应用线程并发执行，导致的部分标记产生变动的对象的标记。（STW）
- 并发清理：清理删除标记阶段判断已经“死亡”的对象。
- （并发重置：重置内部设置）。

其实这上面的4（5）个阶段和我们前面梳理的七个步骤是差不多的，只是7个阶段多了2个并发预处理的过程，这两个流程都没有进行本质上的标记或者清理，要么是处理脏卡，要么是为最终标记做铺垫。还有最后一个书中是没有的提到的并发重置阶段，这3个阶段都没有做本质上的标记或清理。所以从7个阶段缩减到4个也是可以理解的。CMS 整个过程中也是需要STW的只不过STW的时间很短。大多数时间都是都在并发的进行垃圾回收。

CMS 的缺点

CMS是一款优秀的垃圾收集器，但是的优点也很明显并发收集、低停顿。在一些官方文档中也称之为“并发低停顿收集器”，CMS是HotSpot虚拟机追求低停顿的一次成功尝试，但是它还远远达不到成功的程度，至少它有下面三个缺点：

1. 对处理器资源非常敏感。面向并发设计的程序对处理器都很敏感。
2. 无法处理“浮动垃圾”（floating garbage），有可能会出现“Concurrent Mode failed”并发失败进而导致一次完整STW的Serail FullGC。

> 浮动垃圾指的是在 CMS 并发标记和清理期间，由于应用程序并没有停止运行，这个过程中会有垃圾不断的产生，但这一部分垃圾是出现在标记过程中之后的，因此 CMS 在当轮垃圾回收的过程中没法处理它们，所以这些垃圾只有到下一次垃圾收集时才能处理。这一部分垃圾就成为“浮动垃圾”（floating garbage）因此 CMS 不能等到老年代被填满了才进行垃圾收集，必须为可能产生的浮动垃圾预留一些空间。如果浮动垃圾堆满了预留的空间那就会出现并发失败的问题了。可以通过下面的参数设置当已经使用的空间达到多少时触发 CMS。你可以思考这个参数设置不当会造成什么后果😏。

```
-XX:CMSInitiatingOccupancyFraction=70 #总使用空间达到70%触发CMS
```

3. 内存碎片，因为CMS采用的是标记清除算法。内存碎片不可避免。内存碎片过多是将给大对象的分配带来麻烦。时常会会为了给大对象分配空间但由于内存碎片而不得不进行一次 FullGC。为了解决这个问题 CMS 提供了两个JVM 参数，但是在 JDK9开始废弃。

```
-XX:UseCMSCompactAtFullCollection #默认开启，在CMS不得不FullGC的时候开启内存合并整理。
-XX:CMSFullGCsBeforeCompaction=n #n次FullGC之后，下次FullGC前进行内存碎片整理，默认值0，每次进入FullGC都会进行内存碎片整理。
```

Garbage First（G1）

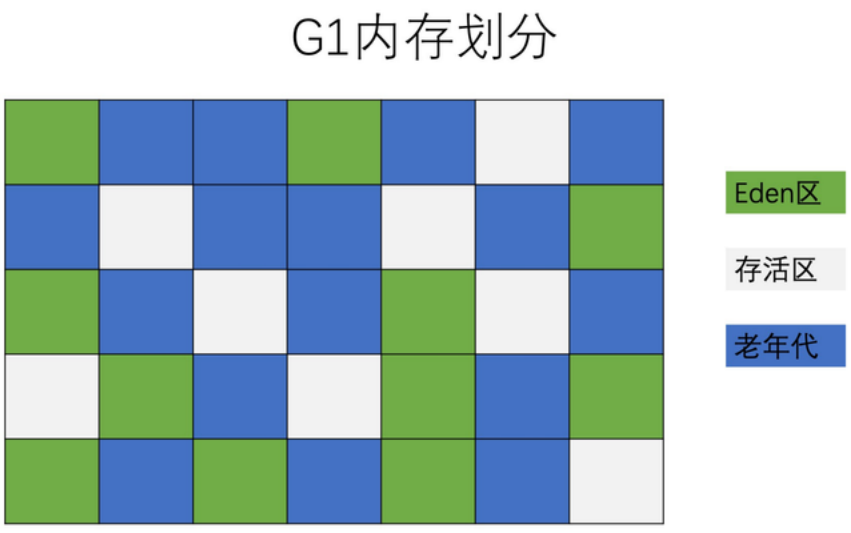
Garbage First 简称（G1）收集器是垃圾收集技术发展历史上的里程碑的成果，它开创了收集器面向局部收集的思路和**基于Region的内存布局形式**。G1是一款面向服务端的垃圾收集器，HotSpot 开发团队最初赋予它的期望是代替 CMS 垃圾收集器。**在JDK9中G1已经代替 Parallel 收集器成为JDK的默认垃圾收集器。**G1既然作为CMS的挑战者，G1对垃圾收集做出了哪些革命性的变革，G1又是否能很好的解决CMS中存在的问题呢?

后来者的独门秘籍

在前面梳理CMS的缺点中，我们发现无法处理 浮动垃圾 和 内存碎片 是CMS的硬伤。这些问题G1也要面对，但是G1采用了一种非常巧妙的方式去解决。浮动垃圾，这是与应用线程并发运行的垃圾收集器的通病，这个问题的根本矛盾是**垃圾收集线程在不断进行垃圾收集，与此同时应用线程又在不断产生垃圾，是垃圾生产与垃圾回收之间的不平衡。**这是一个不可调和的矛盾。如果我们用发展的眼光看问题，我们会发现如果浮动 **垃圾的产生速度>垃圾收集的速度**，那么再大的空间都会被迫触发一次FullGC，如果 **垃圾产生的速度<垃圾收集速度**，产生浮动垃圾对整个垃圾收集也不会产生太大的影响。

> 这就像你妈在打扫房间而你又在一旁制造垃圾，这个时候你妈除了把你胖揍一顿拿你毫无办法。谁让这垃圾收集线程与应用线程之间是要并发的关系呢。解决的办法也很简单，只要你妈打扫卫生的速度大于你生产垃圾的速度，你妈就能把垃圾打扫完并把你胖揍一顿。因此只要 内存分配速率 < 垃圾收集速率 那么一切都很完美，浮动垃圾随他去吧，。

那么 内存碎片的问题，G1是怎么处理的呢？ “化整为零”，这个思路很特别，跳出原有的思维束缚。G1的内存结构和传统的内存结构非常的不同，每个内存块还是有Eden区，Survivor区和Old的区的划分。但不再分成年轻代和老年代，而是划分为多个（通常是2048个）可以存放对象的 小块内存区域（smaller heap regions） 。每一个小块，可能一会被定位为Eden、Survivor或Old区，所有的Eden区拼在一起就是年轻代，所有的Old区拼在一起就是老年代。



除了这些问题上，创新性的解决思路，G1还有一个小目标，**希望能做出一款能够建立起“停顿时间模型”（Pause Prediction Model）的收集器**，停顿时间模型的意思是能够**支持指定在一个长度为M毫秒**

的时间片段内，消耗在垃圾收集器上的时间大概不会超过N毫秒的这样的目标。

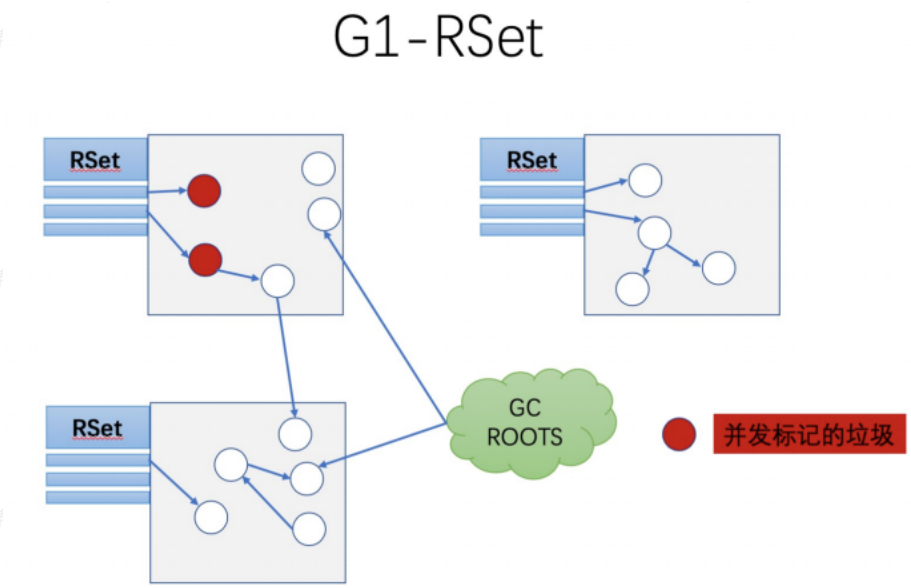
有了上面的思路的转变对于G1的认知是否有转变呢？ 面对浮动垃圾的问题，G1没有直面的去解决而是通过提高垃圾回收速率的方式即 垃圾回收速率>内存分配速率 的方式来处理。面对前面CMS遇到的内存碎片问题，**G1将整个内存“打碎成块”然后通过整体复制的方式直接避免了内存碎片的产生**。这个解决的方式是不是很巧妙。通过上述的解决方案，再来看G1的小目标。希望能做出一款能够建立起“停顿时间模型”（Pause Prediction Model）的收集器，是不是就有了思路呢？

只要每次垃圾收集的周期只要收集的垃圾的速率>内存分配速率是不是就可以了呢？是的！每个内存块又是独立的。这样每次GC周期目标就很明确了，只要收集全部的新生代和部分老年代的内存块，并且保证每次GC周期垃圾收集速率 > 内存分配速率。每次GC周期不用完成所有内存块的垃圾收集工作，停顿时间模型也可以基于G1建立起来，这同时也呼应了这款垃圾收集器的名字 Garbage First。这种设计思路从工程实现上来看是从 G1 上开始兴起，所以说 G1 是收集器发展的一个里程碑。

G1 遇到的困难及解决方案

虽然G1提出了前面的很多解决 CMS 遇到问题的理论，但是理论到实现之间还隔着很长一段路。G1从理论走线实践至少有一下三个关键细节问题需要妥善解决。

- **跨 Region 之间的引用问题。**根据前面的学习，我们可以知道使用记忆集Remember Set（也就是我们前面提到的Card Table），通过记忆集的方式，可以避免全堆作为 GC Roots 进行扫描。在之前的技术中，不管新生代垃圾收集时用到还是CMS并发标记标记引用变动时用到记忆集，都是简单老年代的标记。但是G1在记忆集中的运用则复杂的多，每一个Region都需要维护着自己的记忆集。里面记录着**别的Region指向自己的指针，并标记着这些指针在哪些范围卡页之内。**Remeber Set 本质的集合是一种 Hash 表，Key 是别的Region的起始地址，Value是一个集合，里面存储的元素卡表的索引号，这种结构不同于卡表，卡表记录的是“我指向谁”，而这种结构记录的更多的是“谁指向我”。这种结构比卡表实现起来更加的复杂，同时Region的数量又比传统收集器分代数量多得多，**因此G1收集器相比其他的传统垃圾收集器有着更高的内存占用负担**，更具经验这个**额外开销大致相当于堆容量的10%～20%。**



- **并发标记阶段如何保证收集线程和用户线程互不干扰的运行。**这里要解决的两个问题，一个是并发的进行标记，用户线程改变对象引用关系时，必须保证其不能打破原本的对象图结构，导致标记结果出现问题。我们可以采用三色标记法解决这个问题，三色标记法在前面一小节《JVM GC篇 — 一般原理与垃圾收集算法》中有详细介绍。此外**垃圾收集对应用线程的影响还体现在回收过程中对象的分配上**，程序要继续运行就要在垃圾回收的过程中创建对象分配内存空间。G1为每个Region设计了**两个名为TAMS（Top at Mark Start）的指针，把一部分空间划分出来用于并发回收过程中新的对象分配**。G1在这个地址上的对象是隐式标记过的，即默认它们是“存活”的，不纳入回收范围。在G1中也有 CMS 中“Concurrent Mode failed”的类似场景，如果**垃圾回收速度赶不上内存分配速度**，G1也会被迫进行`FullGC`，从而导致较长时间的STW。
- **如何建立可靠的停顿预测模型。**解决了上面的两个问题还有一个问题怎么建立可靠的停顿预测模型？这是G1设计的小目标，用户通过`-XX:MaxGCPauseMillis`参数指定停顿时间只意味着垃圾

收集发生之前的期望，但是G1是怎么在垃圾收集时，满足用户的期望值的呢？G1 使用的停顿预测模型是以`衰减均值（Decaying Average）`，为理论基础来实现的。在垃圾收集的过程中，G1收集器会记录每个 Region 的回收耗时、每个 Region 记忆集里面的脏卡数量等各个**可测量的步骤花费的成本，并且分析得出平均值、标准偏差、置信度等统计信息**。这里强调的“`衰减平均值`”是指的**它会比普通的平均值更容易收到新数据的影响**，平均值代表整体平均状态，但是平均值更准确地代表了“最近的”平均状态。换句话说，**Region的统计状态越新越能决定其回收的价值**。然后通过这些信息进行分析，完成在**不超过期望停顿时间的约束下达到收集收益的最大化。**

G1 的收集流程

如果不考虑应用线程在运行过程中的动作（用写屏障维护记忆集操作），G1 的垃圾回收过程可以分为下面四个步骤：

- **初始化标记（Inital Marking）**：仅仅是标记下**GC Roots能直接关联到的对象**，并且**修改`TAMS`指针**的位置的值，让下一个阶段用户线程并发运行的时候，能正确地在**可用的Region中分配对象**，这个阶段需要暂停线程，但耗时很短，而且是借用进行的 Minor GC 的时候同步完成的，所以 G1 收集器在这个阶段**实际并没有额外的停顿。**

> “借用进行的 Minor GC 的时候同步完成的”这里是Mixed模式的GC 即收集**年轻代和部分老年代**，一次Minor GC之后，老年代占据堆内存的百占比超过InitiatingHeapOccupancyPercent（默认45%）时，就会触发一次 MixedGC

- **并发标记（Concurrent Marking）**：从GC Root开始对堆中的对象进行可达性分析，**递归扫描整个堆里面的对象图，找出要回收的对象**。这个阶段时间比较长的，可与应用线程并发执行。当扫描完成还要并发时的引用变化，为最终的最终标记阶段处理SATB打下基础。
- **最终标记（final Marking）**：对应用线程做一个短暂的暂停，用于**处理并发阶段结束后仍然遗留下的少量的SATB。**
- **筛选回收（Living Data Counting and Evacuation）**：负责更新Region的统计数据，对各个 Region 的回收价值和成本进行排序，**根据用户期望的停顿时间来制定回收计划**，可以选择任意多个Region构成回收集，然后**把回收集中存活的对象复制到空的 Region 中，再清空整个旧Region空间**。因为在复制对象过程中设计的存活对象的移动，所以必须要暂停应用线程，同时由多个收集器线程并行完成复制过程。这个暂停的过程叫做**转移暂停（Evacuation Pause）**

> 写屏障，这里的写屏障和多线程Java 内存模型中的内存模型不一样，这里的写屏障只是在字节码层面，在执行某个写操作时候一个**类似AOP的结构**，可以将一些操作插入在写入前后。

一个细节 — 单个Region垃圾收集与内存分配的并发策略

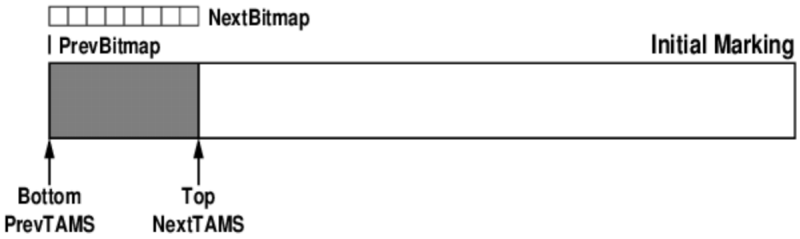
我们前面也梳理了一遍G1的垃圾收集过程，大体上有一个清晰明了的认识，但是有一个细节我始终不明白就是在“G1 遇到的问题中的第二个问题，并发标记阶段如何保证收集线程和用户线程互不干扰的运行。”里面提到了 `G1为每个Region设计了两个名为TAMS（Top at Mark Start）的指针，把一部分空间划分出来用于并发回收过程中新的对象分配。` **`TAMS指针`**到底是什么？**为什么需要两个TAMS指针才能划出一块空间**呢？难道不是一个就可以划分两片区域一片垃圾回收一片分配对象空间就可以了吗？这个问题的本质问题就是**单个Region在并发标记阶段垃圾收集与内存分配之间的并发策略是怎样的？**为了让并发的线程操作Region中的区块，我们需要额外的数据结构协助，两个`bitmap`和两个指针（`TAMS指针`）。我们先来看看两个bitMap和他们在并发标记过程中的一些行为动作：

> bitMap 在 G1 中是用来标记垃圾位置的，垃圾不会直接在 region 中被标记出来，而是使用一个bitMap来标记待回收对象位置。

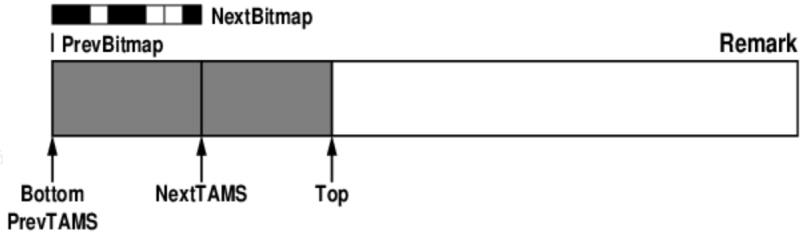
- 这两个`bitMap`分别是`previousBitMap`，`nextBitMap`。
- `previousBitMap`是**上一轮**`concurrent marking`阶段完成标记后的**没有被回收**的垃圾位置。
- `nextBitMap`是当前正在进行的`concurrent marking`阶段的bitmap。
- 当`concurrent marking`标记完成后，两个`bitmap`会交换角色。

bitmap 上面的数组结构（`RoaringBitMap`和`redis`中的`BitMap`都是这样的结构），其中**白色的区域是存活对象，灰色的是待回收的垃圾对象**。除了bitmap我们还需要两个`TAMS`（Top at Mark Start）指针。接下来我们来看看在垃圾回收各个阶段，这两个指针是怎么配合划分线程进行内存来分配和垃圾回收。

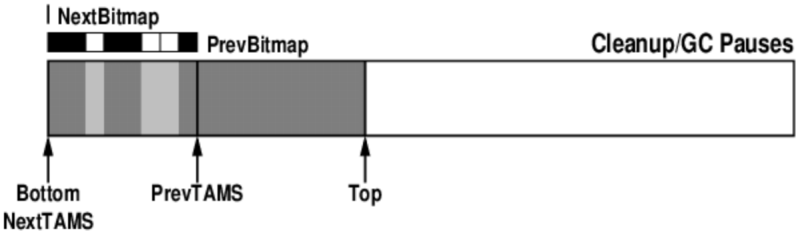
- **初始标记阶段**，从下面的图中我们不难发现，初始标记阶段`PrevTAMS`指针和`Bottom`指针（**region的初始位置**）位置一致，同时第二个`NextTAMS`的指针和Top指针位置一致，其中**top指针是已分配的内存和未分配内存的切分点。**同时初始化`NextBitMap`，由于这是一块干净的Region，因此PrevBitMap是空的。



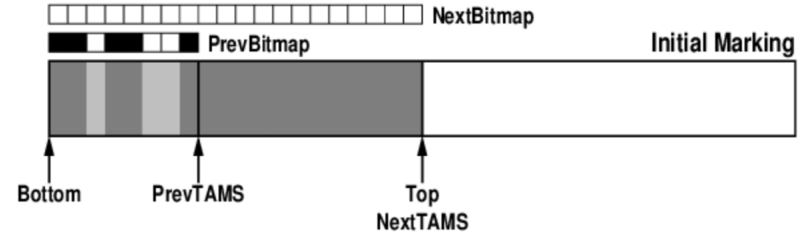
- **并发标记和最终阶段**，下面的图是标记完成后Region中的情况，其中GC线程在`PrevTAMS`和`NextTAMS`**之间进行并发标记**，而**新对象的内存分配在`NextTAMS`和`TOP`之间进**行，由于是刚分配的对象GC默认这里的对象都是存活状态。这样就巧妙的解决了垃圾收集和对象内存分配的并发问题。



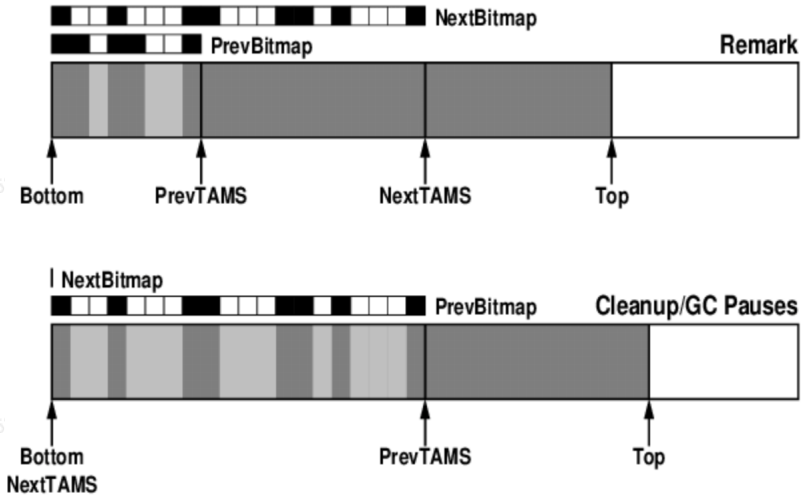
- **清理阶段**，最终清理阶段，我们将`NextBitMap`赋值给`PrevBitMap`。如果**不进行暂停转移并发清理垃圾对象**即可并**继续使用**这个Region。如果进行**对象转移**，那么将**把存活对象复制到一个新的**Region中并**清空**当前Region。



- **下一轮初始标记**，同样的我们初始化新的`NextBitMap`，这里还有一个重要的操作就是**把`NextTAMS`指向`TOP`的位置**，告诉GC本轮标记的工作空间范围。然后接下来`TAMS`和`bitMap`就按照上面的标记和清理阶段不断循环下去。



接下来Region的中的TAMS指针和bitMap的情况如下图所示。



如果有静态图还是有点懵，整个流程的动图如下 😉

> 这个问题我也是看书时候看懵了，为什么需要两个TAMS呢？就像彻底搞清楚这个流程，然后在网上发现了《面试官问我G1回收器怎么知道你是什么时候的垃圾？》这篇文章，这篇文章写的非常有意思，强烈建议看一遍原文。

G1 中常用的参数

- `-XX:+UseG1GC` ：启用G1 GC，JDK7和JDK8要求必须显示申请启动G1 GC；
- `-XX:G1NewSizePercent` ：初始年轻代占整个Java Heap的大小，默认值为5%；
- `-XX:G1MaxNewSizePercent` ：最大年轻代占整个Java Heap的大小，默认值为60%；
- `-XX:G1HeapRegionSize` ：设置每个Region的大小，单位MB，需要为1，2，4，8，16，32中的某个值，默认是堆内存的1/2000。如果这个值设置比较大，那么大对象就可以进入Region了。
- `-XX:ConcGCThreads` ：与Java应用一起执行的GC线程数量，默认是**Java线程的1/4**，减少这个参数的数值可能会提升并行回收的效率，提高系统内部吞吐量。如果这个数值过低，参与回收垃圾的线程不足，也会导致并行回收机制耗时加长。
- `-XX:+InitiatingHeapOccupancyPercent` （简称IHOP）：G1内部并行回收循环启动的阈值，默认为Java Heap的45%。这个可以理解为老年代使用大于等于45%的时候，JVM会启动垃圾回收。这个值非常重要，它决定了在什么时间启动老年代的并行回收。
- `-XX:G1HeapWastePercent` ：G1停止回收的最小内存大小，默认是堆大小的5%。GC会收集所有的Region中的对象，但是如果下降到了5%，就会停下来不再收集了。就是说，不必每次回收就把所有的垃圾都处理完，可以遗留少量的下次处理，这样也降低了单次消耗的时间。
- `-XX:G1MixedGCCountTarget` ：设置并行循环之后需要有多少个混合GC启动，默认值是8个。老年代Regions的回收时间通常比年轻代的收集时间要长一些。所以如果混合收集器比较多，可以允许G1延长老年代的收集时间。
- `-XX:+G1PrintRegionLivenessInfo` ：这个参数需要和 `-XX:+UnlockDiagnosticVMOptions` 配合启动，打印JVM的调试信息，每个Region里的对象存活信息。
- `-XX:G1ReservePercent` ：G1为了保留一些空间用于年代之间的提升，默认值是堆空间的10%。因为大量执行回收的地方在年轻代（存活时间较短），所以如果你的应用里面有比较大的堆内存空间、比较多的大对象存活，这里需要保留一些内存。
- `-XX:+G1SummarizeRSetStats` ：这也是一个VM的调试信息。如果启用，会在VM退出的时候打印出RSets的详细总结信息。如果启用 `-XX:G1SummaryRSetStatsPeriod` 参数，就会阶段性地打印RSets信息。

- `-XX:+G1TraceConcRefinement` ：这个也是一个VM的调试信息，如果启用，并行回收阶段的日志就会被详细打印出来。
- `-XX:+GCTimeRatio` ：大家知道，GC的有些阶段是需要Stop-the-World，即停止应用线程的。这个参数就是计算花在Java应用线程上和花在GC线程上的时间比率，默认是9，跟新生代内存的分配比例一致。这个参数主要的目的是让用户可以控制花在应用上的时间，G1的计算公式是`1/(1+GCTimeRatio)`。这样如果参数设置为9，则最多10%的时间会花在GC工作上面。Parallel GC的默认值是99，表示1%的时间被用在GC上面，这是因为Parallel GC贯穿整个GC，而G1则根据Region来进行划分，不需要全局性扫描整个内存堆。

> 🤔 思考，为什么G1会允许最多10%的时间会花在GC工作上面？这个值还会影响堆空间的大小，当超过预期的时间用在GC上时候，GC会通过适当扩大堆空间的方式来降低GC时间占比。从这个角度来看，G1相比之前的垃圾收集器，在堆扩大的策略上并没有那么激进的。这可能也得益于G1内存Region化的设计和可靠的暂停预测模型。

- `-XX:+UseStringDeduplication` ：手动开启Java String对象的去重工作，这个是JDK8u20版本之后新增的参数，主要用于相同String避免重复申请内存，节约Region的使用。
- `-XX:MaxGCPauseMills` ：预期G1每次执行GC操作的暂停时间，单位是毫秒，默认值是200毫秒， G1会尽量保证控制在这个范围内。

这里面最重要的参数，就是：

1. `-XX:+UseG1GC` ：启用G1 GC；
2. `-XX:+InitiatingHeapOccupancyPercent` ：决定什么情况下发生G1 GC；
3. `-XX:MaxGCPauseMills` ：期望每次GC暂定的时间，比如我们设置为50，则G1 GC会通过调节每次GC的操作时间，尽量让每次系统的GC停顿都在50上下浮动。如果某次GC时间超过50ms，比如说100ms，那么系统会自动在后面动态调整GC行为，围绕50毫秒浮动。

GC选择经验

收集器	串行、并行或并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单机CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后预备方案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境下在Server模式下配置CMS使用
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台计算而不需要太多交互任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台计算而不需要太多交互任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网或者B/S系统服务端上的Java应用
G1	并发	both	标记-清除+复制算法	响应速度优先	面向服务端应用，将来替换CMS

综合的看下来，G1是HotSpot JVM 中最先进的**准产品级（production-ready）**垃圾收集器，并且 HotSpot 工程师的主要精力都放在不断改进 G1 上，在更新的JDK版本中，将会带来更加强大的功能和优化。作为CMS的替代者，G1弥补了CMS中各种不足，包括暂停时间可预测，并彻底解决了堆内存的碎片化问题。**对于单业务延迟非常敏感的系统来说，如果CPU资源不受限制，G1 是HotSpot 中最好的选择**。当然这些优化和延迟也是要付出代价的。由于G1额外的写屏障和守护线程，G1运行也会消耗比其他垃圾收集器多得多的CPU和内存资源。因此在**小内存，CPU资源比较宽裕，且服务响应速度优先**的应用上，CMS是更好的选择。

> G1 适合大内存，需要较低延迟的场景。

具体场景适合什么样的垃圾收集器，只有尝试了才知道，但是并不是什么时候都能有试的机会，因此我们一般指导原则：

- **系统系统吞吐量优先，CPU资源能最大程度处理业务，选择ParallelGC。**
- **如果系统低延迟优先，每次GC时间尽可能短，但配置资源有限，选择CMS GC。**
- **如果系统内存大，同时追求整体GC时间可控，选择G1。**

对于内存大小的考量：

- 4G以上算比较大的，G1 性价比更高。
- 8G以上，非常推荐G1。

总结

垃圾收集器上篇我写了小两周终于是整理完了。在梳理之前我对CMS和G1内心都是很敬畏的，因为之前对他们的原理和实现的理解都是一知半解模棱两可的。在这次梳理的过程中，我也花了很大的篇幅在介绍说明CMS和G1的原理，我查了很多的资料学习CMS和G1的原理以及部分实现细节，收获非常多。回到总结上，这一小节我们介绍了经典的垃圾收集器，我们先从基础的 Serial（Serial + Serial Old）串行垃圾收集器，Parallel（Parallel Scavenge + Parallel Old）并行垃圾收集器开始，介绍了他们特性、优缺点和适用的场景。开胃菜结束后，我们开始梳理CMS，对CMS的回收过程进行了详细的分析，他们分别是初始标记、并发标记、并发预处理、可取消的并发预处理、最终标记、并发清除、并发重置这7个步骤，总结简化下来4个阶段分别是初始标记、并发标记、最终标记、并发清理。深入理解CMS垃圾收集器的回收流程为G1的回收流程的梳理打下铺垫。我们还介绍了CMS的几个缺点，分别是并发处理对资源敏感、会产生浮动垃圾、标记-清除算法会产生内存碎片。最后一道大菜介绍了我们的G1垃圾收集器，从CMS的痛点入手，分析梳理G1中的解决方案和G1的设计小目标—建立可靠预测暂停模型。随后我们顺着CMS的垃圾回收流程，过了一遍G1的垃圾收集流程，不难发现两者设计一脉相承大同小异。但是梳理了这么多“大”的知识点，没有细节总感觉少了些什么，随后我们梳理了一个细节：单个Region垃圾收集与内存分配的并发策略，来深入理解Region的垃圾回收过程。在G1的最后，我们列出了G1常用的参数以供各位朋友查询配置。在梳理完经典垃圾收集器之后，我们简单聊了聊GC的选择策略，吞吐量选择Parallel，低延迟但内存空间没那么大CMS，内存大GC时间可控选G1。这里插一嘴我个人的看法，我不是很喜欢CMS。它是对低延迟的垃圾收集器的一次成功尝试这不可否认，但是算不上一款成功的垃圾处理器。但是换一个角度想如果没有CMS的尝试，又怎么会有后来大成者G1呢？🤔

学习资料

- 深入理解 Java 虚拟机（第三版）
- 常用垃圾收集器的具体实现
- 面试官问我G1回收器怎么知道你是什么时候的垃圾?
- G1详解