我们来聊聊JVM的G1吧

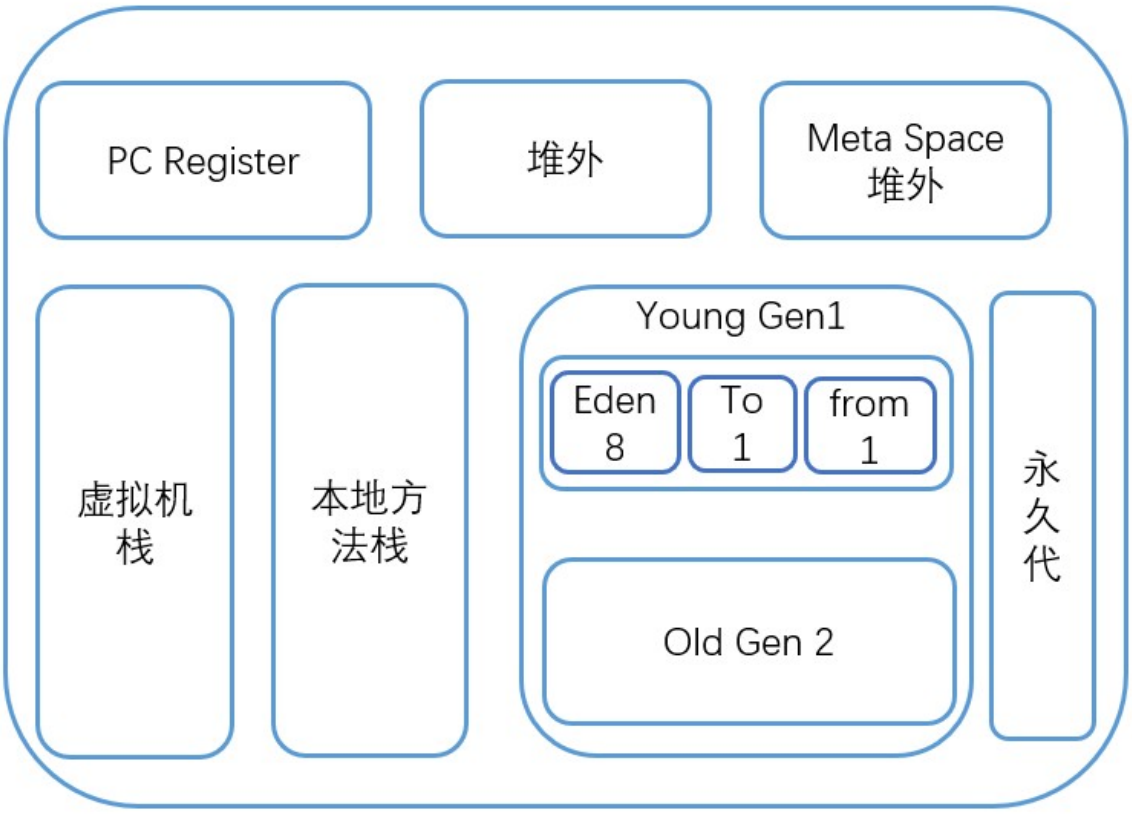
> 当我书写的时候，我觉得我开始平静起来。这是一个系列

我们来聊聊JVM的G1吧

- 前言
- 垃圾回收器简介
  - 垃圾回收器组合
  - 标记复制与对象晋升
  - 标记整理和标记清除算法
  - 总结一下
- 元空间什么时候被回收
- 重点关注G1
  - 设计目标
  - 内存布局
  - G1的回收过程
  - SATB 和 三色标记算法
  - 加快回收: Remembered Set、Collection Set
- 总结一下
- 参考资料

前言

在《我们来聊聊VM的GC吧》中我们概述了JVM运行时区域划分，在JDK 8之前，JVM的运行时区域划分如下图所示:



- PC(program counter) register: 每个线程都有PC寄存器，对于非本地方法存储的当前指令的地址，对于本地方法PC寄存器未定义。

> 本地方法的意思是: Java调用别的语言，Unsafe的allocateMemory就是一个本地方法， 只有一个方法声明没有方法体。

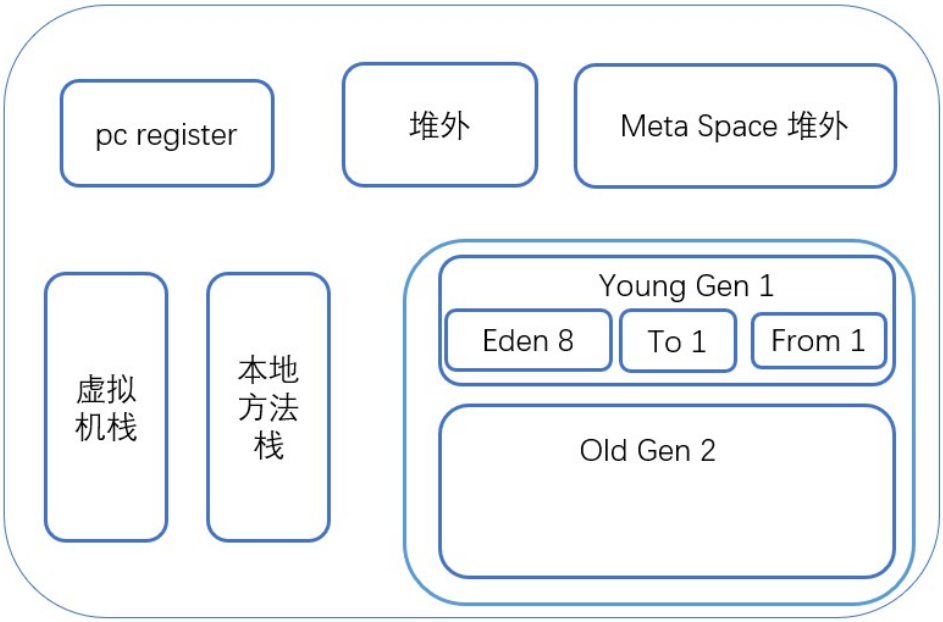
- 堆外

虽然元空间在堆外，但堆外的数据并不一定属于元空间，因为我们还有Unsafe，在JDK8可以像下面这样分配内存:

```
unsafe.allocateMemory(1024);
```

- 虚拟机栈: 每次方法调用都会触发栈上创建一个新帧，用于存储方法的局部变量和返回地址。
- 本地方法栈: 同虚拟栈类似，由于本地方法不会被编译为字节码，所以需要单独一块区域进行存储。
- 堆: 运行时存储对象，当我们new 一个实例或者创建一个数组，JVM都会从堆里面分配内存:
  - 在CMS 和
  - G1
- 永久代：在JVM规范里面并未提及永久代，这是一个逻辑概念，规范提出了方法区，我们可以理解为接口，在实际中的实现是永久代，用于存储类和接口定义，具体一点说就是类加载器加载字节码进入JVM，然后，JVM创建类的内部表示，用于运行时创建对象和调用方法，该内部表示收集了类和接口的字段、方法和构造函数的相关信息。

在JDK 8之后 hotspot替换了方法区的实际实现，也就是元数据区域:



知道了我们的内存被划分为哪几块区域，每块区域存储什么数据之后，我们还是将目光移动到堆上面，因为平时我们new 对象最多，一般来说通过new产生出来对象会被分配到堆上，我们大致讲解了分配内存的过程，刚开始是撞针分配，但是在多核时代，应用程序通常都要考虑多线程，在加上应用一般都会借助线程池来复用线程，在这种情况下我们可以做出预设，一般来说每个线程分配的内存是比较稳定的，这里稳定的意思是每次分配对象的大小，每轮GC分配区间的分配对象个数以及总大小。所以我们可以考虑每个线程每个线程分配内存后，将这块内存保留下来，用于下次分配，这样就不用每次都从主内存分配了，这条也就是TLAB策略。

垃圾回收器简介

垃圾回收器组合

然后我们来关注垃圾回收器, 不同的垃圾回收器管理堆的方式不同，这里提到的管理也就是如何利用空间，如何回收垃圾，在《我们来聊聊VM的GC吧》我们提到了几种常见的垃圾回收器组合:

- Parallel Scavenge(年轻代) + Parallel Old(老年代)： 关注吞吐量，在JDK 17的介绍里面，Parallel也被称为吞吐量收集器。
- Par New(年轻代) + CMS (老年代)： 关注延迟，JDK 14 CMS 垃圾回收器已经被移除。
- G1：在延迟和吞吐量之间取一个平衡。

  > G1弱化了分代的概念，但G1并不等于抛弃了分代的概念，G1选择将堆分割成若干区域，分区和分代的设计初衷来自一个预设绝大多数对象朝生夕死，我们可以先将目光关注在这些朝生夕死的对象上，从而避免垃圾回收器扫描全堆，降低对应用本身的影响，这也是ZGC走向分代的原因。

- ZGC: 延迟，JDK23中 分代ZGC已经变成默认选项。
- Serial: 内存占用和启动速度

标记复制与对象晋升

一般年轻代会采用标记-复制算法，所谓标记 - 复制也就是，将空间分为大小相同的from 和 to两个半区，每次进行回收时将一个半区的存活对象通过复制的方式转移到另一个半区。这也就是我们在上面画的JVM运行时区域划分中堆里面的年轻代分为eden、to 、from的原因，年轻代发生gc之后，年轻代发生的GC一般也被称为mirror gc，如果幸存者区域，也就是我们提到的 to 和from，还能够容纳存活的对象，就会将这些幸存者对象复制到幸存者区域里面。基于这种假设，如果幸存者的区域持续回收不掉，幸存者区域就会被打满，由此就引出了对象晋升，在并行垃圾回收器中对象晋升由下面两个参数控制:

- -XX:InitialTenuringThreshold
- -XX:MaxTenuringThreshold

在CMS和G1里面只受XX:MaxTenuringThreshold 这一个参数控制。

从名字上推断一个是初始值一个是最大值，其实对于这两个参数我是有点不理解的，我以为初始值就是对象年龄超过了初始值，年轻代的对象就会晋升到年老代，MaxTenuringThreshold就让我感到迷惑了，于是我就把这两个参数在JDK参数中的说明多读了两遍:

InitialTenuringThreshold: Sets the initial tenuring threshold for use in adaptive GC sizing in the parallel young collector. The tenuring threshold is the number of times an object survives a young collection before being promoted to the old, or tenured, generation.

> 在自适应GC里面对并行垃圾回收器的年轻代设置晋升阈值，这个晋升阈值是在年轻代里面存活的对象在精力几次垃圾回收之后才被晋升到老年代。

MaxTenuringThreshold: Sets the maximum tenuring threshold for use in adaptive GC sizing. The current largest value is 15. The default value is 15 for the parallel collector and is 4 for CMS.

> 在自适应GC中设置最大技术能拿阈值，当前的最大阈值是15，并行垃圾回收器里面是15，在CMS垃圾回收器里面是4。

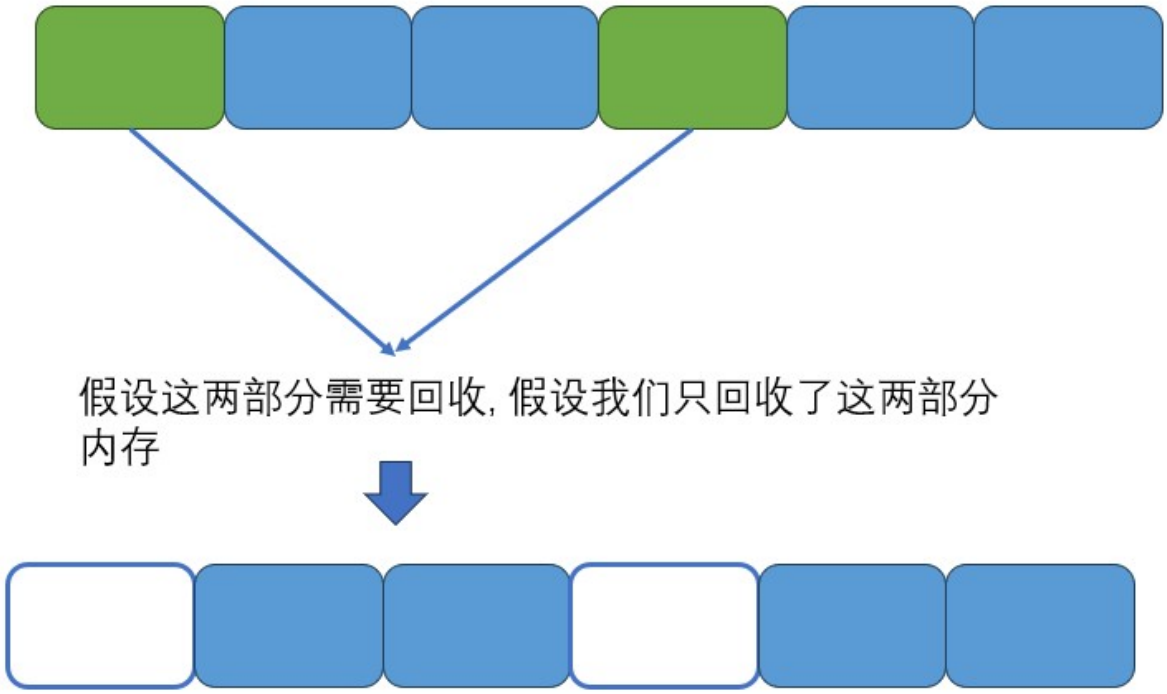
于是我猜测在并行垃圾回收器里面，InitialTenuringThreshold 是左区间，MaxTenuringThreshold是右区间。有了猜想，就需要验证，于是在搜索引擎上瞎搜索，搜到一个jdk的bug：

> VM exits if MaxTenuringThreshold is set below the default InitialTenuringThreshold, and InitialTenuringThreshold is not set
>
> 如果MaxTenuringThreshold 低于默认的InitialTenuringThreshold，且InitialTenuringThreshold 没有设置，则VM会退出。

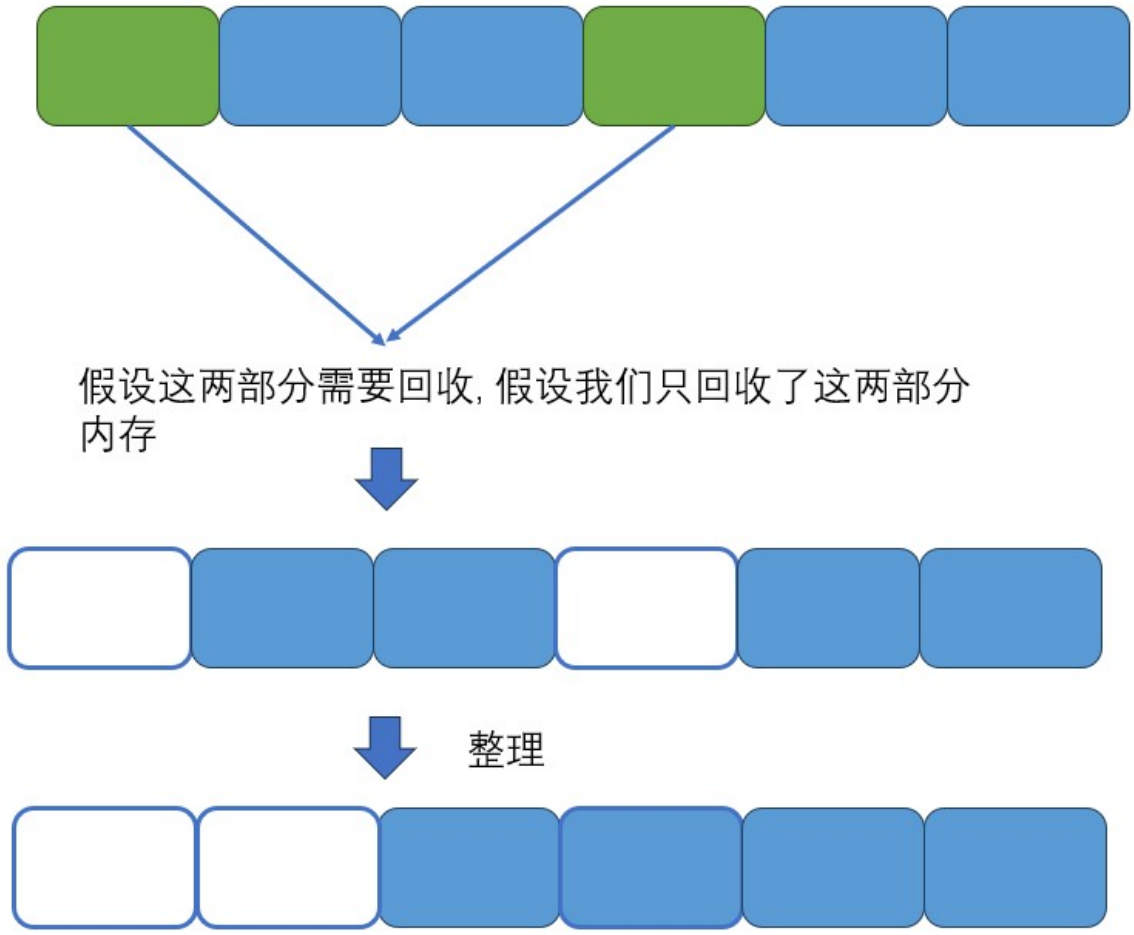
看起来MaxTenuringThreshold 要大于InitialTenuringThreshold才对，结合参考资料实际的阈值是由JVM动态调整的，因此我们可以做出推导，这两个参数为JVM划定了一个区间。写到这里有些唏嘘，以前总以为是固定的参数，其实JVM早就做到了自适应。我好像总以为我知道，其实我不知道。

标记整理和标记清除算法

接下来我们讲标记-整理算法，这是老年代的垃圾回收器一般会采取的算法，其实标记整理和标记清除算法可以放到一块说，一般没有只清除不整理，当我们将垃圾回收，也就是释放对象占据的内存，可剩余的空间可能是不连续的，造成了内存碎片像下面一样



于是我们就为这个算法打上了第一道补丁，也就是整理，标记整理，如下图所示:



这也就是说标记整理算法在第一个阶段和标记清除算法的过程是相似的，，第二阶段则会对存活对象按照整理顺序（Compaction Order）进行整理。CMS垃圾回收器采取的就是标记-清除-整理算法，CMS垃圾回收器在实现上分为前台收集(foreground collector)和后台收集(background collector)，foreground collector 触发条件比较简单，一般是遇到对象分配但空间不够，就会直接触发 GC，来立即进行空间回收。采用的算法是 mark sweep，不压缩。在CMS发生foreground gc才是FullGC。而 background collector 就相对复杂一些，触发条件就更多了，一般来说由-XX:CMSInitiatingOccupancyFraction来控制，也就是老年代占堆的比例，但注意我说的是一般，一般的意思是还会有其他条件触发后台GC，具体的参看参考资料[8]。 那并行垃圾回收器在什么时候触发GC，这方面倒是没看到参考资料。

总结一下

三种算法在是否移动对象、空间和时间方面的一些对比，假设存活对象数量为 *L*、堆空间大小为 *H*，则：

	移动对象	空间开销	时间开销
Mark-Sweep	否	低（有碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，sweep 阶段与整堆大小成正比 *O(H)*
Mark-Compact	是	低（无碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，compaction 阶段与存活对象的大小成正比 *O(L)*
Copying	是	高	与存活对象大小成正比 *O(L)*

声明图片来自于参考文档[6], 把标记(mark)、清除(sweep)、compaction(整理)、复制(copying)这几种动作的耗时放在一起看，大致有这样的关系:

$$\begin{cases} compaction \geq copying > mark > sweep \\ mark + sweep > copying \end{cases}$$

虽然 整理 与 复制 都涉及移动对象，但取决于具体算法，整理可能要先计算一次对象的目标地址，然后修正指针，最后再移动对象。复制则可以把这几件事情合为一体来做，所以可以快一些。另外，还需要留意 GC 带来的开销不能只看 收集 的耗时，还得看 分配 。如果能保证内存没碎片，分配就可以用撞针方式，只需要挪一个指针就完成了分配，非常快。而如果内存有碎片就得用 freelist 之类的方式管理，分配速度通常会慢一些。

元空间什么时候被回收

按道理来说元空间在堆外不在垃圾回收器的管辖范围之内，但是堆里面会有一些对象持有元空间对象的特殊引用，当gc回收这些特殊的对象的时候，响应的metaspace会被释放。根据上面的讨论我们知道元空间，存储类和接口定义，具体一点说就是类加载器加载字节码进入JVM，然后，JVM创建类的内部表示，用于运行时创建对象和调用方法，该内部表示收集了类和接口的字段、方法和构造函数的相关信息。那么也就是说当某个类被卸载的时候，元空间中对应的数据会被回收。

重点关注G1

设计目标

The Garbage-First (G1) garbage collector is targeted for multiprocessor machines scaling to a large amount of memory. It attempts to meet garbage collection pause-time goals with high probability while achieving high throughput with little need for configuration. G1 aims to provide the best balance between latency and throughput using current target applications and environments whose features include:

G1适用于内存容量比较大的多处理器机器，它尝试在尽可能的满足垃圾回收暂停时间目标的同时在很少的配置下满足吞吐量。G1旨在使用当前目标应用程序延迟和吞吐量之间的最佳平衡，特点包括:

- Heap sizes up to tens of GBs or larger, with more than 50% of the Java heap occupied with live data.

> 堆的大小可达数十GB甚至更大，超过百分之五十的堆被存活数据占用。

- Rates of object allocation and promotion that can vary significantly over time.

> 对象分配和晋升的速率可能随时间显著变化

- A significant amount of fragmentation in the heap.

> 堆中存在大量碎片

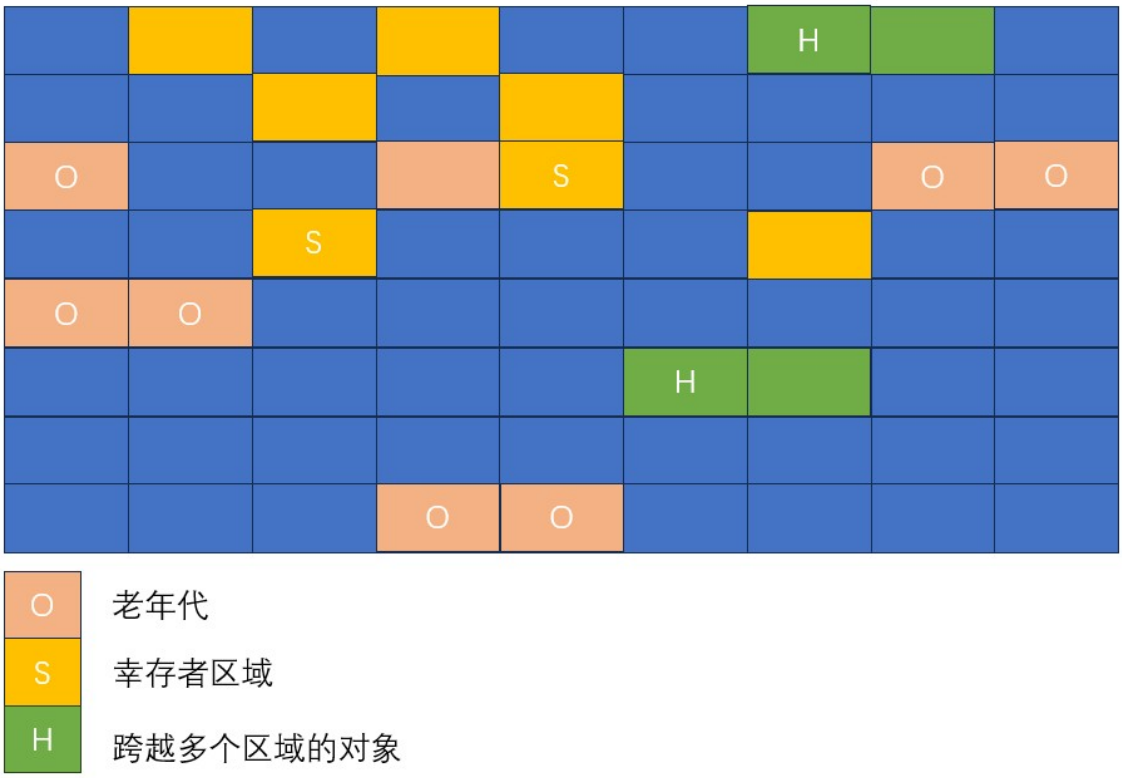
- Predictable pause-time target goals that aren’t longer than a few hundred milliseconds, avoiding long garbage collection pauses.

> 可预测的暂停时间目标,不超过几百毫秒,避免长时间的垃圾收集暂停。

注意G1的主要设计目标是平衡延迟和吞吐量，在这个基础之上做到可预测的暂停时间，因此就引出了G1的一个调优参数: -XX:MaxGCPauseMillis=200。 最大暂停时间目标，我们可以由这个参数来指定我们希望最长的暂停时间，G1会为尽力达到这个目标，尽力的意思是存在尽力也做不到的情况。为了做到这个模板G1引入了暂停预测模型(Pause Prediction Model)，G1使用这个模型来达到用户定义的最大暂停时间，并根据暂停时间目标选择要采集的区域数量。

内存布局

现在让我们将关注点放在G1上，这也是我们本篇重点讨论的垃圾回收器G1，下图是G1垃圾回收器中堆布局图:



注意有些区域我们专门标了H，H是Humongous(庞大的)，它表示这些区域存储的是巨大对象(humongous object)，即大小大于等于region一半的对象。为了避免反复拷贝移动，这类对象会被直接分配到老年代，这种大对象的分配可能会导致垃圾收集暂停过早的发生，因为G1会在每次分配巨型对象时检查堆占用的阈值，如果当前占用超过该阈值，则会立即强制执行Initial Mark 和 young colletion。Initial Mark 和 young collection 是G1垃圾收集过程中的两个动作，而上面的堆占用阈值则由一个参数控制，也就是-XX:InitiatingHeapOccupancyPercent 默认是45， 你是不是还想根据这个参数调优，答案是不再需要了，JDK 9 加入了IHOP自适应，见参考文档[14]。 那这个XX:InitiatingHeapOccupancyPercent 参数是堆占用比例嘛，答案是要分版本在8b12之前是这样，在8b12之后是老年代占堆总体容量的比值，见参考文档[15]。在JDK 11(见参考文档17)和JDK 8(见参考文档16) 的文档都出现了错误，对-XX:InitiatingHeapOccupancyPercent 的参数描述均为:

> Sets the Java heap occupancy threshold that triggers a marking cycle. The default occupancy is 45 percent of the entire Java heap.

但是在JDK 17 发行文档中(见参考文档[18])是这么说的:

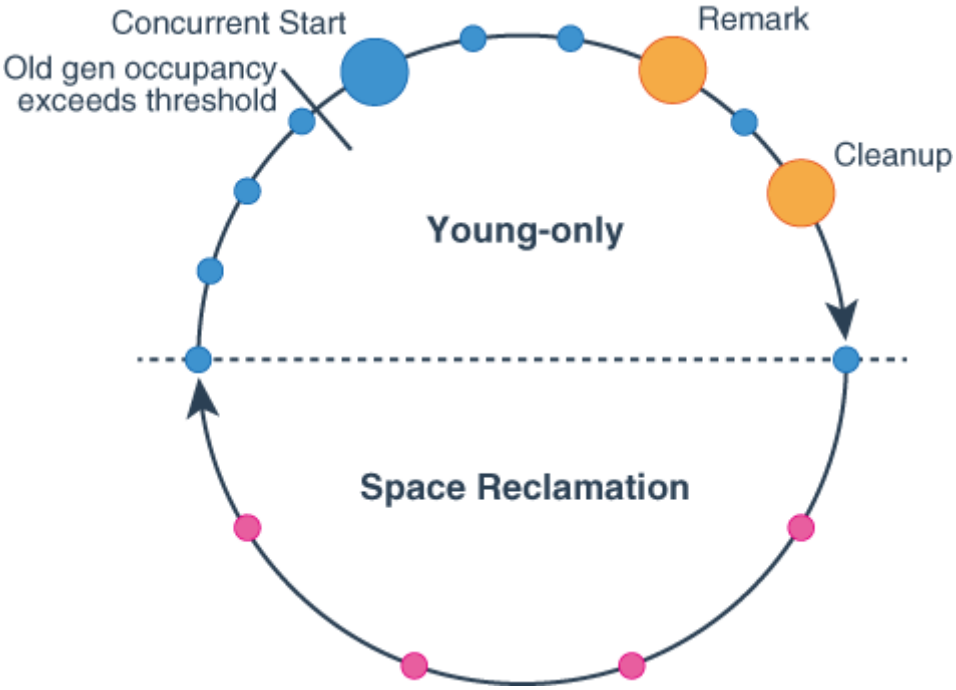
> Defaults for controlling the initiating heap occupancy indicate that adaptive determination of that value is turned on, and that for the first few collection cycles G1 will use an occupancy of 45% of the old generation as mark start threshold.

G1的回收过程

接下来让我们大致讲一下 G1的回收过程，也就是理清楚 Initial Mark 和 young collection 这两个名词对应的含义，从宏观上来看，G1在两个阶段之间交替进行：

1. 新生代收集阶段(Young -only phase)：在这个阶段只执行新生代的收集，每次新生代收集会将存活下来的对象提升到老年代。

2. 在空间回收阶段(space-reclamation phase): 在这个阶段G1不仅处理年轻代，而且增量式地回收老年代的空间。然后循环重新开始，重新进入新生代收集的阶段。



声明图片来自参考文档[18]。上面的描述引出的一个问题就是G1的垃圾回收模式，G1有三种回收模式:

1. young gc: 只回收年轻代
2. mixed gc: 回收年轻代的同时也增量式的回收老年代的空间。
3. full gc: 在mixed gc 回收的内存不够的时候触发full gc，回收全堆。

但是不幸的是在JDK 10之前，G1的full gc都是单线程的，扫描全堆会非常慢，在JDK 10 的JEP 307 才为G1的Full GC引入了并行化。我们接着来讲G1的垃圾回收过程:

1. 新生代收集阶段(Young-Only phase): 这个阶段从几次普通的新生代收集开始，在这几次收集之后一些幸存下来的对象将会被晋升到老年代。当老年代的占用到达一定的阈值(即 Initiating Heap Occupancy)， 新生代收集阶段和空间回收阶段之间的转换就开始了。此时GC会安排一次concurrent start young collection(这个词暂时没有想到很好的翻译，暂时先用英文，其实这个词表达的是一个阶段，这个阶段不仅对新生代开始收集，还会启动并发标记过程)代替普通的年轻代收集。
    - Concurrent Start：这种类型的收集除了执行普通的新生代收集外，还会启动并发标记过程。并发标记确定老年代区域中所有当前可达(存活)的对象，以便在接下来的空间回收过程保留，并发标记没有完成时，也可能发生普通的新生代收集。并发标记过程以两个特殊的stop-the-world结束: remark和cleanup。
    - Remark (标记) ： 这个暂停最终完成标记本身,执行全局引用处理和类卸载,回收完全空的区域,并清理内部数据结构。在Remark阶段和Cleanup阶段之间,G1会并发地计算一些信息,用于后续在老年代Region中回收空闲空间。这个并发计算的结果将在Cleanup阶段中被使用
    - CleanUp (清理) :这个暂停将会决定下面是否进入空间回收阶段，如果G1决定进入空间回收阶段，那么在当前的新生代阶段收集将会以一次特殊的新生代收集结束，也就是Prepare Mixed young collection。
2. 空间回收阶段: 这个阶段会包含多个混合收集(Mixed collections), 这些混合收集不仅会处理新生代区域，还会处理选定的老年代区域。当G1判断疏散更多的老年代空间不会产生更多的可用空间来证明付出的努力是值得的，空间回收阶段结束了。

在空间回收阶段结束之后，G1会重新进入新生代收集这个阶段，如果在这个阶段里面如果出现了内存不够用，G1会执行一次FullGC。

SATB 和 三色标记算法

本来打算细致的讲一下SATB，但是发现不太好讲，这里只做简单介绍，上面我们提到在并发开始(Concurrent Start)这个阶段会并发标记老年代区域中所有当前可达(存活)对象，以便在接下来的空间回收过程保留。那么G1是如何标记存活的对象的呢，答案是SATB(Snapshot-At-The-Beginning)算法，它在初始标记暂停时对堆进行虚拟快照，此时所有在标记开始时存活的对象在标记的剩余过程中都被视为存活。这意味着浮动垃圾会比较严重，如果在标记结束之后，这个对象已经需要回收了，那么SATB会让这个对象躲过这次GC。

那G1是如何标记的呢，G1通过一种三色标记算法来维持正确性，三色标记算法将对象分为三种状态:

1. 白：对象没有被标记到，标记阶段结束后，会被当做垃圾回收掉
2. 灰：对象被标记了，但是它的field还没有被标记或标记完。
3. 黑：对象被标记了，且它的所有field也被标记完了

SATB的做法精度比较低，所以造成的浮动垃圾比较多。

加快回收: Remembered Set、Collection Set

为了加快垃圾回收过程，G1引入了Remembered Set、Collection Set，这是典型的以空间换时间的工具:

- Collection Set（CSet），它记录了GC要收集的Region集合，集合里的Region可以是任意年代的。在GC的时候，对于old->young和old->old的跨代对象引用，只要扫描对应的CSet中的RSet即可。
- Remembered Set: 逻辑上说每个Region都有一个RSet，RSet记录了其他Region中的对象引用本Region中对象的关系，属于points-into结构（谁引用了我的对象）。

总结一下

在这篇我们主要回顾了JVM 运行时区域划分, 每次调用一个方法会产生一个栈帧，栈帧存储局部变量和返回地址，在Java里面方法分为本地方法和非本地方法，非本地方法调用的时候存储在虚拟机栈里面，本地方法存储在本地方法栈里面，元空间用于存储类的定义信息，程序计数器对于非本地方法存储指令地址。然后我们回顾了几种常见的垃圾回收算法: 标记-复制、标记-清除、标记-整理。 标记-复制最快但是消耗空间，标记-清除如果不整理会导致内存碎片比较严重，CMS垃圾回收器采用的是标记-清除-整理算法，CMS垃圾回收器分为前台GC和后台GC，前台GC是我们常规意义上的FullGC，后台GC是CMS GC，触发条件有多种，一般是堆占用比例。G1在年轻代实现的是标记-复制，老年代采用的是标记整理，G1的主要设计目标为平衡吞吐量和延迟，可预测的暂停时间，和其他垃圾回收器一样，年轻代在内存不足以容纳应用线程所需要分配内存的时候触发yong gc，在这个阶段会开始并发标记，并发标记里面会判断是否需要提前进入mixed gc阶段，在mixed gc阶段会进行增量式回收，增量式回收的意思是不仅回收年轻代也会回收老年代。 G1会根据预测暂停模型主动的回收。 G1在新生代收集阶段，会执行并发标记，并发标记存活的对象，这在官方文档中被称为concurrent start，在这个阶段之后，G1会进入Remark，这个阶段回收垃圾，同时会并发的计算region信息，在clean up阶段，G1会根据暂停预测模型计算是否要提前进入混合回收阶段(mixed gc)。在《深入理解Java虚拟机(第3版)》对G1的工作步骤描述的更详细，也是跟我们的描述对的上的，只是更为详细:

初始标记(Initial Marking)：这阶段仅仅只是标记GC Roots能直接关联到的对象并修改TAMS(Next Top at Mark Start)的值，让下一阶段用户程序并发运行时，能在正确的可用的Region中创建新对象，这阶段需要停顿线程，但是耗时很短。而且是借用进行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。

并发标记(Concurrent Marking)：从GC Roots开始对堆的对象进行可达性分析，递归扫描整个堆里的对象图，找出存活的对象，这阶段耗时较长，但是可以与用户程序并发执行。当对象图扫描完成以后，还要重新处理SATB记录下的在并发时有引用变动的对象。

最终标记(Final Marking)：对用户线程做另一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的最后那少量的 SATB 记录。

筛选回收(Live 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，根据用户所期望的停顿时间来制定回收计划。

本来还想写写高版本中G1的进化的，有些进化甚至没被列入jep里面，梳理这些也是一个庞大的过程，索性就将这一部分砍掉吧。

参考资料

[1] The JVM Run-Time Data Areas https://www.baeldung.com/java-jvm-run-time-data-areas

[2] Managing Memory and Garbage Collection https://docs.oracle.com/cd/E19900-01/819-4742/abeic/index.html

[3] JVM内存：年轻代、老年代、永久代（推荐 转） https://www.cnblogs.com/snowwhite/p/9532311.html

[4] Java HotSpot VM Options https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

[5] MaxTenuringThreshold - how exactly it works? https://stackoverflow.com/questions/13543468/maxtenuringthreshold-how-exactly-it-works

[6] MaxTenuringThreshold与阈值的动态调整理论详解 https://www.cnblogs.com/webor2006/p/11031563.html

[7] VM exits if MaxTenuringThreshold is set below the default InitialTenuringThreshold, and InitialTenuringThreshold is not set https://bugs.openjdk.org/browse/JDK-8014765?page=com.atlassian.jira.plugin.system.issuetabpanels%3Aall-tabpanel

[8] JVM 源码解读之 CMS GC 触发条件 http://www.disheng.tech/blog/jvm-%E6%BA%90%E7%A0%81%E8%A7%A3%E8%AF%BB%E4%B9%8B-cms-gc-%E8%A7%A6%E5%8F%91%E6%9D%A1%E4%BB%B6/

[9] https://tech.meituan.com/2020/11/12/java-9-cms-gc.html

[10] Java Metaspace Full GC https://stackoverflow.com/questions/53101801/java-metaspace-full-gc

[12] Java garbage collection: The 10-release evolution from JDK 8 to JDK 18 https://blogs.oracle.com/javamagazine/post/java-garbage-collectors-evolution

[13] Garbage First Garbage Collector Tuning https://www.oracle.com/technical-resources/articles/java/g1gc.html

[14] JDK-8199262 https://bugs.openjdk.org/browse/JDK-8199262 https://bugs.openjdk.org/browse/JDK-8199262

[15] 官方文档竟然有坑！关于G1参数InitiatingHeapOccupancyPercent的正确认知 #我在性能调优路上的打怪日记# https://heapdump.cn/article/2712390

[16] JDK 8 参数官方文档 https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

[17] JDK 11 参数官方文档 https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE

[18] HotSpot Virtual Machine Garbage Collection Tuning Guide https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html#GUID-0394E76A-1A8F-425E-A0D0-B48A3DC82B42

[19] How do objects age on G1 (Garbage First) garbage collector? https://stackoverflow.com/questions/35057097/how-do-objects-age-on-g1-garbage-first-garbage-collector