垃圾收集器与内存分配策略

笔记本：	jvm		
创建时间：	2018/9/16/周日 15:13	**更新时间：**	2018/9/17/周一 16:22
作者：	1634896520@qq.com		
URL：	https://www.cnblogs.com/jing99/p/6071808.html		

垃圾收集器与内存分配策略

思维导图：https://www.jianshu.com/p/088d71f20a47

一、概述

当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

程序计数器、虚拟机栈、本地方法栈这三个区域随线程而生，随线程灭而灭。

而**Java堆**和**方法区**则不一样，一个接口中多个实现类需要的内蹙眉可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的就是这部分内存。

二、如何判断对象已经死去？

（一）引用计数算法

给对象添加一个引用计数器，如果这个对象被引用，那么引用计数器就会加一，若没有，则会减一，如果引用计数器的值为零，那么就会被回收。

一般情况下，这样的判断方法实现简单，而且效率也挺高，但是值得注意的是，这样会产生一个弊端，那就是如果两个对象，相互引用，又没有其他地方引用它们。那么引用计数器的值始终都是大于零的，所以就不会进行回收，那么就很容易造成FULL GC。

（二）可达性分析算法

通过一系列的称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，搜索所走过的路径叫作引用链，当一个对象到GC Roots没有任何引用链相连时，那么证明此对象是不可用的，应该进行回收。

在Java语言中，可作为GC Roots的对象包括下面几种：

- 虚拟机栈中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNT引用的对象

三、谈谈什么是引用

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为：**强引用，软引用，弱引用，虚引用**

- 强引用

  指在程序代码中普遍存在的，只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。

- 软引用

  用来描述一些有用但是并非必须的对象，在系统将要发生内存溢出之前，将会把这些对象列入进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。

- 弱引用

  用来描述非必须对象的，但是它的强度要比软引用要更小，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。

- 虚引用

  它是最弱的一种引用关系。通过可达性分析后发现没有与GC Roots相连接的引用链，那它会进行第一次标记，筛选的条件是此对象是否有必要执行finalize方法，当对象没有必要执行此方法的话，就会被回收，如果有必要执行，就会

将此对象从“即将回收”的集合中逃出来。

四、如何回收方法区

方法区也称“永久代”，永久代的垃圾收集主要回收两部分内存，：废弃常量和无用类，

类需要满足以下条件才能算是“无用的类”：’

- 该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例
- 加载该类的ClassLoarder已经被回收
- 该类对应的java.lang.class对象没有在任何地方引用，无法在任何地方通过反射问该类方法。

五、几种垃圾收集算法

（一）标记清除算法

首先标记出所有要回收的对象，在标记完成后统一回收所有被标记的对象。

主要不足有两个：

- 一个是效率问题，标记和清除两个过程的效率都不高
- 另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后再程序运行过程中需要分配较大的对象时，无法找到足够的内存而不得不提前出发另一次垃圾收集动作。

（二）复制算法

为了解决效率问题，便出现了复制算法。

它将可用内存容量划分为大小相等的两块，每次只使用其中的一块，当这一块的内存用完了，就将还存活着的对象复制到另一块上面，然后再把已经使用过的内存空间一次清理掉，这样使得每次都是对整个半区进行内存回收，内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

（三）标记-整理算法

复制算法在对象存活率比较高的时候，就会进行比较多的复制操作，效率不高

标记-整理算法的过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动。然后直接清理掉端边界以外的内存。

（四）分代收集算法

根据对象的存活周期的不同，将内存（Java堆）划分为新生代和老年代，新生代每次垃圾收集的时候，都会有大量的对象死去，只有少量的对象存活，这样我们就可以用“复制算法”，这样就只需要复制少量的对象就可以实现垃圾收集。当然，反之老年代，对象存活率比较高，就使用“标记-清除算法”或者“标记-整理算法”。

*六、HotSpot的算法实现

（一）.枚举根节点

在可达性分析中，可以作为GC Roots的节点有很多，但是现在很多应用仅仅方法区就有上百MB，如果逐个检查的话，效率就会变得不可接受。

而且，可达性分析必须在一个一致性的快照中进行-即整个分析期间，系统就像冻结了一样。否则如果一边分析，系统一边动态表化，得到的结果就没有准确性。这就导致了系统GC时必须停顿所有的Java执行线程。

目前主流Java虚拟机使用的都是准确式GC，所以当执行系统都停顿下来之后，并不需要一个不漏的检查完所有执行上下文和全局的引用位置，虚拟机应该有办法直接知道哪些地方存放着对象引用。在HotSpot实现中，使用一组称为 **OopMap** 的数据结构来达到这个目的。OopMap会在类加载完成的时候，记录对象内什么偏移量上是什么类型的数据，在JTI编译过程中，也会在特定的位置记录下栈和寄存器哪些位置是引用。这样，在GC扫描的时候就可以直接得到这些信息了。

（二）.安全点

如果OopMap内容变化的指令非常多，HotSpot并不会为每条指令都产生OopMap，只是在特定的位置记录了这些信息，这些位置成为“安全点”（SafePoint）。程序执行时只有在达到安全点的时候才停顿开始GC。一般具有较长运行时间的指令才能被选为安全点，如方法调用、循环跳转、异常跳转等。

接下来要考虑的便是，如何在GC时保证所有的线程都“跑”到安全点上停顿下来。这里有两种方案： **抢先式中断**（Preemptive Suspension）**和主动式中断**（Voluntary Suspension）。

抢先式中断会把所有线程中断，如果某个线程不在安全点上，就恢复线程让它跑到安全点上。几乎没有虚拟机采用这种方式。

主动式中断思想是需要中断线程时，不直接对线程操作，而是设置一个GC标志，各个线程会轮询这个标志并在需要时自己中断挂起。这样，轮询标志的地方和安全点是重合的。

（三）.安全区域

安全点机制保证程序执行时，在不太长的时间内就会遇到可进入GC的安全点，但是，程序“不执行”的时候呢，程序不执行就是没有分配CPU时间，这时线程无法响应JVM的中断请求，JVM显然不太可能的等待线程重新被分配CPU时间。

安全区域是指一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的。

在线程执行到安全区域代码时，首先标识自己进入安全区域，当这段时间里JVM发起GC，不用管标识为安全区域的线程了。在线程要离开安全区域时，要检查系统是否已经完成了根节点枚举，如果完成，线程继续执行，否则等待直到收到可以安全离开安全区域的信号为止。

七、垃圾收集器

没有最好的收集器，只有合适的收集器，不同的需求选择不同的收集器进行搭配。

（一）Serial收集器

- 单线程收集器
- 在它进行垃圾收集的时候，必须暂停掉所有的工作进程，直到它收集结束，这叫作“Stop the world”
- 虚拟机运行在**Client模式**下的默认新生代收集器。

优点在于：

简单而高效，对于限定的单个CPU来说，Serial由于没有线程交互的开销，专心做垃圾收集，自然可以获得最高的单线程收集效率。

（二）ParNew收集器

这款收集器，其实就是Serial收集器的多线程版本，它与Serial收集器的功能几乎完全一样，但它却是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为，除了Serial收集器以外，也只有它才能够与CMS收集器进行配合工作。

ParNew收集器在单CPU甚至两个CPU环境下的效率并没有Serial收集器好。

（三）Parallel Scavenge收集器

Parallel Scavenge是一个新生代收集器，它也是使用复制算法来实现的收集器，又是并行（多条垃圾收集线程并行工作）的多线程收集器。

CMS等收集器的关注点是尽可能的减少垃圾收集时，线程停顿的时间，而Parallel Scavenge收集器的目的则是达到一个可控制的吞吐量，所谓吞吐量就是cpu用于运行代码的时间与cpu消耗时间的比值。

GC停顿时间的缩短是靠牺牲吞吐量和新生代空间来换取的。Parallel Scavenge有一个参数叫作：-XX:UseAdaptiveSizePolicy ，当打开这个参数以后，就不需要手工去设置一些新生代的大小，Eden与Survivor区的比例了，晋升老年代对象的大小等细节参数了，这个参数会使虚拟机会根据当前系统的运行情况收集性能监控信息，动态的自动调整这些参数以提供最合适的停顿时间与最大的吞吐量，这种调节方式叫作“GC的自适应调节策略”。

（四）Serial Old收集器

这是一个Serial的老年代版本，同样也是一个单线程收集器，使用“标记-整理算法”，主要也是给Client模式下的虚拟机使用。

如果在Server模式下，它还有两个用途：

- 在JDK1.5以及之前的版本中与 Parallel Scavenge收集器一起使用
- 另一种用途就是作为CMS收集器的后备预案，在并发收集发生时使用。

（五）Parallel Old 收集器

同理Serial Old，这其实就是 Parallel Scavenge的一个老年代版本，使用的是多线程和“标记清理算法”。

在注重吞吐量，以及cpu资源敏感的场合，都可以优先考虑Parallel Scanvenge收集器和Parallel Old收集器。

（六） CMS收集器

以获得最短回收停顿时间为目的的收集器，希望系统停顿时间最短，以给用户带来较好的体验，基于“标记-清除算法”实现的

整个过程分为四个阶段：

- 初始标记（需要Stop the world,速度很快）
- 并发标记（耗时很长，但能与用户线程一起进行）
- 重新标记（需要Stop the world，比初始标记速度慢，远比并发标记速度快）
- 并发清除（耗时很长，但能与用户线程一起进行）

缺点：

- CMS收集器对CPU资源非常敏感：在并发过程，虽然不会导致用户线程停顿，但是这会占用一部分cpu资源，而导致应用程序变慢，总吞吐量会降低。
- CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致一次Full GC的产生。
- 因为它是基于“标记-清除算法”实现的，那么它就容易出现大量的空间碎片问题。

*（七）G1收集器（最前沿成果之一）

这是一款面向服务器端的收集器，它有如下特点：

- 并行与并发
- 分代收集
- 空间整合
- 可预测的停顿

G1垃圾收集算法主要应用在多CPU大内存的服务中，在满足高吞吐量的同时，竟可能的满足垃圾回收时的暂停时间，该设计主要针对如下应用场景：

- 垃圾收集线程和应用线程并发执行，和CMS一样
- 空闲内存压缩时避免冗长的暂停时间
- 应用需要更多可预测的GC暂停时间
- 不希望牺牲太多的吞吐性能
- 不需要很大的Java堆（翻译的有点虚，多大才算大？）

具体详见：https://www.jianshu.com/p/0f1f5adffdc1

八、理解GC日志

https://blog.csdn.net/hp910315/article/details/50936629

九、垃圾收集常用参数总结

参　　数	描　　述
UseSerialGC	虚拟机运行在Client 模式下的默认值，打开此开关后，使用Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为CMS 收集器出现Concurrent Mode Failure失败后的后备收集器使用
UseParallelGC	虚拟机运行在Server 模式下的默认值，打开此开关后，使用Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用Parallel Scavenge + Parallel Old 的收集器组合进行内

	存回收
SurvivorRatio	新生代中Eden 区域与Survivor 区域的容量比值， 默认为8， 代表 Eden ：Survivor=8 : 1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次Minor GC 之后，年龄就加1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行GC 时进行内存回收的线程数
GCTimeRatio	GC 时间占总时间的比率，默认值为99，即允许1% 的GC 时间。仅在使用Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置GC 的最大停顿时间。仅在使用Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用CMS 收集器时生效
UseCMSCompactAtFullCollection	设置CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用CMS 收集器时生效

十、内存回收与分配策略

（一）对象优先在Eden分配

大多数情况下，对象在新生代Eden区中分配，当Eden区没有足够的空间进行分配的时候，虚拟机就会再次发起一次MinorGC（新生代GC）。

（二）大对象直接进入老年代

所谓大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组，大对象对于虚拟机来说是一个坏消息，经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获得足够的连续空间来安置他们。

虚拟机提供了一个-XX:PretenureSizeThreshold参数，令大于这个参数值的对象直接在老年代分配，这样做的目的是为了避免在Eden区以及两个Survivor区之间发生大量的内存复制（新生代采用复制算法收集内存）。

、

（三）长期存活的对象进入老年代

如果对象在Eden区出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1。对象在Survivor区中每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度的时候（默认为15岁），就将会被晋升到老年代中。

（四）动态对象年龄判定

虚拟机并不是永远的要求对象的年龄必须达到MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的综合大于Survivor空间的一半，年龄大于或者等于该年龄的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。

（五）空间分配担保

在发生Minor GC之前，虚拟机会检查老年代最大可用连续空间是否大于新生代所有对象总空间。如果这个条件成立，那么Minor GC可以确保是安全的，如果不成立，则虚拟机会查看HanlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行下一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC

那么什么是冒险？

新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况，就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代。