Java高级工程师

JVM(二)

垃圾回收器

收集和清除未引用的对象.垃圾回收器可以被显示调用"System.gc()",
但是执行没有保证.垃圾回收集是那些被创建的对象

垃圾回收器

垃圾回收器是重要的面试考点

gc回收条件

将对象设置为null，可以帮助gc标记回收，但是并不一定会执行回收。要看当时jvm运行状态和内存情况

四种类型的回收器

- 串行垃圾回收器（Serial Garbage Collector）
- 并行垃圾回收器（Parallel Garbage Collector）
- 并发标记扫描垃圾回收器（CMS Garbage Collector）
- G1垃圾回收器（G1 Garbage Collector）

串行垃圾回收器

- 串行垃圾回收器通过持有应用程序所有的线程进行工作。
- 它为单线程环境设计，只使用一个单独的线程进行垃圾回收，通过冻结所有应用程序线程进行工作，所以不适合服务器环境。
- 它最适合的是简单的命令行程序。
- 通过JVM参数-XX:+UseSerialGC可以使用串行垃圾回收器。

并行垃圾回收器（吞吐量优先，适用server）

- 并行垃圾回收器也叫做 throughput collector 。
- 它是JVM的默认垃圾回收器。与串行垃圾回收器不同，它使用多线程进行垃圾回收。
- 相似的是，它也会冻结所有的应用程序线程当执行垃圾回收的时候

并发标记扫描垃圾回收器（响应时长优先，适用接口服务）

- 标记垃圾回收使用多线程扫描堆内存，标记需要清理的实例并且清理被标记过的实例。

- 并发标记垃圾回收器只会在下面两种情况持有应用程序所有线程。
  当标记的引用对象在tenured(老年代)区域；
  在进行垃圾回收的时候，堆内存的数据被并发的改变。

- 相比并行垃圾回收器，并发标记扫描垃圾回收器使用更多的CPU来确保程序的吞吐量。如果我们可以为了更好的程序性能分配更多的CPU，那么并发标记上扫描垃圾回收器是更好的选择相比并发垃圾回收器。
  通过JVM参数 XX:+UseParNewGC 打开并发标记扫描垃圾回收器。

G1垃圾回收器

- G1垃圾回收器适用于堆内存很大的情况，他将堆内存分割成不同的区域，并且并发的对其进行垃圾回收。
- G1也可以在回收内存之后对剩余的堆内存空间进行压缩。并发扫描标记垃圾回收器在STW情况下压缩内存。
- G1垃圾回收会优先选择第一块垃圾最多的区域
- 通过JVM参数 –XX:+UseG1GC 使用G1垃圾回收器

垃圾回收算法

- 引用计数法(Reference Counting Collector)
- tracing算法(Tracing Collector) 或 标记-清除算法(mark and sweep)
- compacting算法 或 标记-整理算法
- copying算法(Compacting Collector)
- 分代算法

引用计数法

堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

引用计数法

优点:引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

引用计数法

缺点:无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0. 。

tracing算法(Tracing Collector) 或标记-清除算法(mark and sweep)

采用从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收

tracing算法(Tracing Collector) 或标记-清除算法(mark and sweep)

标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。

compacting算法 或 标记-整理算法

标记-整理算法采用标记-清除算法一样的方式进行对象的标记

compacting算法 或 标记-整理算法

在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。

compacting算法 或 标记-整理算法

成本更高，但是却解决了内存碎片的问题。

在基于Compacting算法的收集器的实现中，一般增加句柄和句柄表。

copying算法(Compacting Collector)

该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。

copying算法(Compacting Collector)

它开始时把堆分成 一个对象 面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于copying算法的垃圾 收集就从根集中扫描活动对象，并将每个 活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

copying算法(Compacting Collector)

一种典型的基于coping算法的垃圾回收是stop-and-copy算法，它将堆分成对象面和空闲区域面，在对象面与空闲区域面的切换过程中，程序暂停执行。

- 分代算法

- 分代的垃圾回收策略，是基于这样一个事实：

不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。

如何分代

- **年轻代/新生代**
- **老年代**
- **永久代**

年轻代（Young Generation）

- 1.所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

- 2.新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复。

- 3.当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收

- 4.新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)
年老代（Old Generation）

老年代（Old Generation）

- 1.在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。
-
- 2.内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。
- Note:这里有一个翻车率很高的问题,大家get到没有?

持久代（Permanent Generation）

- 用于存放静态文件，如Java类、方法等。

- 持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

- 思考: orm框架下的场景

gc回收区别

- minor gc: 清理年轻代内存
- major gc:清理老年代内存
- full gc:清理年轻代+老年代,会触发stop-the-world

gc回收触发时机

- minjor gc: 新生代Eden区满了，就会触发young gc（minjor gc）。

- full gc: 年老代或者永久代满了 或者 System.gc() 显示调用”有可能“触发。

总结

以上知识点是求职面试中对于JVM面试知识的一个索引

总结

大厂都要对jvm进行深度定制,所以考核绝对不是刷题那么简单

总结

从jvm架构入手去学习是唯一正确的方法