coggle

Java GC

- 其它
  - 可以通过程序打印出程序当前使用的collector ManagementFactory.getGarbageCollectorMXBeans()
- collector的种类
  - HotSpot CMS
    - C4
    - G1
    - [1] p64, 减少了full gc，但是pause（应该是minor gc引起的）会变多
  - HostSpot
    - minor collection是STW的
- 参考资料
  - Java Garbage Collection Distilled [3]
  - G1: One Garbage Collector To Rule Them All [1]
  - The Garbage Collection Handbook by Richard Jones et al.
  - The Java Garbage Collection Mini-Book: 主要参考
  - Tips for Tuning the Garbage First Garbage Collector [2]
- Common misconceptions
  - can't have memory leaks in Java (you can)
  - garbage collection is inefficient (it's not; actually it's far more efficient than C's malloc() for example)
  - can tune the HotSpot CMS collector so that it won't ever stop the world (you can't)
- gc带来的最大问题，unpredictable pause during collection，也叫"stop the world event"
  - mitigate这个问题的方法有
    - breaking programs down into smaller units and distributing them
    - object pooling
    - having fixedsized objects to avoid fragmentation
    - using off-heap storage
- **General monitoring and tuning advice**
  - 任何tuning都是**throughput**，**latency**和**footprint**之间的trade-off
    - **footprint**
      gc对内存的影响
    - **throughput**
      gc对cpu资源的影响
    - **latency**
      gc对程序的执行所造成的pause
  - 一般来说，improve一个会影响另外两个；调优的时候先决定，对于一个程序，应该关注三者中的哪个
  - 选哪种collector
    选collector的一些"rule"
  - Tuning a collector
    - logging
      tuning时应该打开gc的log；一般来说gc的log对应用程序没什么影响
      - **理解gc log，会"读"gc log**
        不同的collector的gc log可能会有不同，但是都会有下面的信息：
        回收后的heap大小，回收占用的时间，回收得到的空间
        - Parallel collector的log
        - CMS的log
        - G1的log
      - 分析log file的工具
        - Chewiebug，for OpenJDK collectors
        - jClarity's Censum
        - VisualVM，oracle jdk自带的
    - **Memory footprint vs. CPU cycles**
      增加内存可以减少gc对cpu的影响（内存增了，需要gc的频率也少了，gc需要的cpu资源也少了）
    - Setting the size of the heap
    - Survivor ratio
- **Programming for less garbage**
  - 各个影响因素
    - **Number of weak or soft references**
    - **Fragmentation and compaction**
    - **live-set size**
      gc的某些操作和它的大小成正比
    - **heap size**
    - **mute rate**
    - **Object lifetime**
  - **Reducing allocation rate**
    - **String**
      a+b这样的操作，会产生临时对象
      用StringBuffer将数好一点
      如果程序中要大量处理string，可以考虑实现自己的string handling类
      考虑用那些直接修改对象的方法，而不是返回新拷贝的方法
    - **Array-capacity planning**
      HashMap等内部用到了数组，为了减少数组的重分配，可以在new HashMap时指定初始大小
    - **Avoid creating unnecessary objects**
      减少中间对象
      用static方法直接生成最终对象
      反复使用同一个对象
      不要在loop里生成大量对象
    - **Using primitives**
      回收基本类型的cost相对小一点
      一些临时的基本类型对象会存在于stack上，不需要回收
      如果基本类型可以表示，那么可以考虑尽量用基本类型
    - **数组**
      有些类，比如HashMap，内部使用到了数组；这些数组可能会因空间不足重分配
      Integer.valueOf()会对小型数据cache
  - **Weak references**
  - **Try-with-resources**
    可以通过这个机制来帮助释放资源
  - **Distributing programs**
    还提到了"rolling restarts"
  - **Other common techniques**
    object pooling，可以重用expensive的对象
    用固定大小的对象，防止碎片
    using off-heap storage
- 基础知识，术语
  - an **object** is either a class instance or an array
  - The **reference** values (often called only "references") logically point to these objects from **stack frames** or other objects.
    reference逻辑上来说是从stack或者其它object指向一个object.
  - object存在**heap**上
    all object instances (including arrays) is allocated in heap
    heap memory can be shared between threads
    All instance fields, static fields, and array elements are stored in the Java heap
    - **但是**，local variables, formal-method parameters, and exception handler parameters reside outside heap; they are never shared between threads and are unaffected by the memory model
    - 除了heap，java会用一些其它的内存来storing material other than Java objects, such as the code cache, VM threads, VM and garbagecollection structures and so on
- **Key Term**
  - JVM的相关参数
    - -XX:SurvivorRatio=6
    - -XX:NewRatio=3，NewSize，MaxNewSize
    - -XX:MinHeapFreeRatio=<minimum>，-XX:MaxHeapFreeRatio=<maximum>
    - -XX:+DisableExplicitGC，禁用System.gc()
    - -XX:+UseConcMarkSweepGC
    - -XX:+UseSerialGC
    - -XX:MaxPermSize，默认64M（64位时，85M）
    - –XX:MaxTenuringThreshold
    - size of heap，**-Xms** initial size，**-Xmx** maximum size
    - -XX:+PrintGCApplicationStoppedTime
      某些JVM支持这个参数，如果支持的话，report GC pause time时会把reach to global safe point的时间也考虑在内
    - -XX:PretenureSizeThreshold=n
      和大对象的创建有关
    - -XX:+UseParallelOldGC和-XX:+UseParallelGC
      用哪个parallel collector
  - gc log相关的参数
    - -Xloggc:<filename>，输出gc log到文件
    - -XX:+PrintGCDetails，gc相关的统计信息
    - -XX:+PrintGCDateStamps，gc发生时的时间信息
    - -XX:+PrintTenuringDistribution，打印tenuring-age info
    - -XX:+PrintGCApplicationConcurrentTime
    - -XX:+PrintGCApplicationStoppedTime，stop point相关
    - -XX:+PrintAdaptiveSizePolicy
    - -XX:MaxGCPauseMillis
    - -XX:InitiatingHeapOccupancyPercent
    - -XX:ConcGCThreads
    - -XX:G1ReservePercent
  - **tracking**
    precise的垃圾回收器（包括现在所有的回收器）在collection时都用了tracking mechanism
    - 对于tracking来说，循环引用不要紧，一个cyclic object graph没有被tracking，那么它就会整个丢弃
    - Tracing collectors use three techniques: **mark/sweep/compact** collection, **copying** collection, and **mark/compact** collection
      一般的垃圾回收器会3种技术都用
      可能heap的一部分区域用一个技术，另外一部分区域用另外一个技术
      - **Mark(also known as trace)**
        - 目的是找到所有living对象
          从root开始找，root包括static variables，registers，thread stacks等
        - 花费的时间
          linearly with the size of the live set rather than the size of the heap.
      - **Compact/relocate**
        - dead object回收后会产生碎片，所以要**periodically** relocate live object，使得碎片可以合并起来
          relocate live object的过程中，live object的位置变了，那么所有指向这个object的引用就要更新一下，这个过程叫做**remapping**
          remapping花费的时间和live set成正比
        - compacting collector分两种：in-place compactor（不需要额外空间，一般把live object从一端移到另一端），evacuating compactor（借助额外空间）
      - **copy**
        - 把heap分成大小相等的两个区域：from和to
          新对象在from里分配
          回收时，把from里所有活的移到to里，更新引用，交换from和to的距离
          花费的时间和live set成正比
        - 当copying collector和generational collector相结合时（HotSpot-based）：把young generation的区域分成3个的部分：1个Eden和2个比Eden小很多的survivor；Eden永远是"from"，两个survivor会在from和to之间互相切换；
          回收（young）时，把from的对象移到"to survivor"里或者promote到old generation区域里（"from survivor"里的对象可能会promote）
          survivor空间小没关系，如果"to survivor"放不下，会移到old generation里；这个叫做**premature promotion**；old generation的大小要和Eden一样大
      - **Sweep**
        - sweep包括：scan through the entire heap, identify all the dead objects
          , and recycle their space in some way
          扫描整个heap，标识dead对象，准备回收这些dead对象
        - 花费的时间和heap的大小成正比
    - 针对short-lived和long-lived对象，heap被分成了两个部分
      young(or "new") generation
      old(or "tenured") generation
      - 如果一个对象活得够久，那么它会**promote**到old generation
        对young generation的回收叫做"**minor garbage-collection event"**
    - 几乎所有的Java collector都或多或少地用到 generational collection
    - **Generational collection**
      - 基于"**the weak generational hypothesis**"假设，就是新创建的对象只会live很短的时间
      - 因为young object dies young，所以GC把重点放在young generation区域
        减少回收old generation的频率（因为很多时候打young generation的回收就够用了，没必要回收old？）
      - **write barrier**
        - a write barrier that intercepts the storing of every Java reference in order to track potential references from old objects into the young generation.
          通过一个"write barrier来拦截所有的reference的storing
          这是一个**blind** barrier，就是会拦截所有的reference，而不仅仅是从old到young的reference；因为条件判断费时间
      - **Remembered sets**
        - "remembered set" as a way of keeping track of all references pointing from the old generation into the young generation.
          它记住所有从old到young的引用
        - considered to be **part of the roots** for young-generation collecting
          在检查young generation的对象有没有被引用时，remembered set作为root的一部分（另一部分应该是stack frame之类的）
          有了remembered set，可以避免扫描整个old generation
        - remembered set is tracked in a **card table** which may use a byte or a bit to indicate that a range of words in the old-generation heap may contain a reference to the young generation
    - Parallel vs. serial, concurrent vs. stop-the-world
      - **single-threaded or parallel**
        collector可以单线程，也可以是parallel的
        单线程的只能用一个cpu core
        parallel的使用多个线程来进行collection，可以利用多核的优势
      - **stop-the-world or concurrent**
        stop-the-world (STW) 的collector在GC时会停止程序的执行
        concurrent的collector在GC期间允许程序继续执行
      - **Incremental vs. monolithic**
        Incremental的collector把GC操作分成多个small discrete steps；每个step之间可能有large gap
        monolithic的collector做一次性"把GC做完
    - **Precise vs. conservative**
      - **OopMaps** in HotSpot-based JVM
        OopMaps是structures that record where object references (OOPs) are located on the Java stack for every piece of code it is safe to do a GC in.
      - A collector is **precise** if it can fully identify and process all object references at the time of collection.
        A collector has to be precise if it is to**move** objects，因为一个对象被移动后要重建所有原来指向它的引用
      - A collector is termed **conservative** if it is unaware of some object references at collection time, or is unsure about whether a field is a reference or not
      - **All commercial server JVMs use precise collectors**
        and use a form of moving collector at some point in the garbage collection cycle.
    - **safe points**
      - 理想情况下，safe point应该经常出现，STW发生时，所有的线程都要到达safe point，如果某一个线程没到，那就要等待。
      - A **global safe point** involves bringing **all** threads to a safe point
      - Garbage-collection events occur at safe points
      - safe point is a **point or range** in a **thread**'s execution **when** the collector can **identify all the references** in that thread's execution stack
      - Bringing a thread to safe point is the act of getting a thread to reach a safe point and then not executing past it
        一个线程到了safe point不意味着它应该停止执行，比如JVM的代码可以执行，因为它不影响safe point
      - 除了GC涉及到safe point，其它场景也可能需要到达safe point，比如JVM de-optimise code时；比如当JVM某一个类不属于继承体系时，它会优化这个类的方法的调用；当JVM发现且续这个类被加入到一个继承体系时，JVM需要de-optimise code；
        de-optimise code需要的safe point和GC需要的safe point两码事不同
- **Two-region collectors**
  应该是指分成old和young的collector
  - **heap的结构**
    分成"young/new"和**tenured**两个区域
    young又分成**Eden**和**survivor**；大部分对象都在Eden中分配；回收Eden时继续存活的对象放到survivor里；
    Eden满了之后会发生**minor GC event**；把对象移到survivor或者tenured叫做**promotion**；移到tenured也叫**tenuring**；
    两个**reserved areas**，它们的目的是To allow a degree of resizing of the pools without having to move everything
  - 对象是怎么分配的
    - 如果对象很大，比如大数组，而且TLAB里放下这个对象，那么会在old generation区域分配
      相关选项，-XX:PretenureSizeThreshold=n
    - **bump-the-pointer**
      tracks the last object allocated to the Eden space, which is always allocated at the top.
      If another object is created afterwards, it checks only that the size of that object is suitable for the Eden space, and if it is, it will be placed at the top of the Eden space.
    - **TLAB**
      thread-local allocation buffers
      每个线程都有个TLAB用来创建对象；如果一个线程用完了TLAB，这个线程会从Eden里再请求一个新的
  - **PermGen**
    - 里面存着objects that it believed were effectively immortal, along with per-class metadata such as hierarchy information, method data, stack and variable sizes, the runtime constant pool, resolved symbolic reference, and Vtables.
    - java8里没有PermGen里，取而代之的是放在native memory里的**Metaspace**
    - 如果class的metadata的大小超过了PermGen，那么就会out of memory
    - 只要old generation或者PermGen中的任意一个满了，就会触发对它们倆的回收
  - **Serial collector**
    - 最简单的collector；适合单处理器；内存的footprint很小；
      minor collection和major collection都是单线程的
      在young generation区域用的是copying collector，对old用的是mark/sweep collection algorithm；
      sweep时用的是**sliding compaction**，就是把old区域的对象都移到一端，让free space集中在另一端；
      sweep好了之后，就可以**bump-the-pointer** 了；
  - **Parallel collector**
    - 它是server-side的默认collector
      monolithic, stop-the-world copying collector for the new generation
      monolithic, stop-the-world mark/sweep for the old generation
      它有两种，分布叫：**Parallel**和**Parallel Old**
    - **Parallel**会多线程地用young-generation-collection algorithm used by the Serial collector；用以提然STW但是影响小，一般几百ms或者更少
      对old回收和Serial Collector一样
    - **Parallel Old**所不管old还是young都是多线程的；自Java 7u4之后默认会用"Parallel Old"；
      如果你内存多，想要高throughput，有可以尽量慢的长pause，那么推荐这个
    - Parallel collector只在STW时才处理weak和soft reference
    - 相关选项-XX:+UseParallelOldGC和-XX:+UseParallelGC
  - **Concurrent Mark Sweep (CMS)**
    - 目标是reduce, or delay, the **old**-generation pauses 为young区域的回收和Parallel collector一样
    - 程序在执行的同时，concurrent multipass marker也在跑；会有竞争，叫**concurrent marking race**；CMS用**SATB（snapshot at the beginning）**来处理这种race；marking开始的时候，记下所有的live object的snapshot；SATB用到了**pre-write barrier**，它和blind write barrier结合起来用；
      - "concurrent mode failure"，mark的速度没有程序修改的速度快
      - CMS的sweep也是并发的；sweep时用到了free list；old generation allocation会先尝试从free list分配；如果free list碎片太多，会STW和compact
      - promote到tenured时，也会先放到free list里；如果"promotion failure"会STW然后再promote
      - CMS会消耗CPU资源，minor collection更expensive，需要内存更多
      - G1是作为CMS的replacement而产生的