jvm的类加载过程?

加载-验证-准备-解析-初始化（-使用-卸载）

- 加载：在硬盘中查找并通过IO读入字节码文件，使用到的类才会被加载
- 验证：校验字节码文件的正确性
- 准备：给类的静态变量分配内存，并赋予默认值
- 解析：将符号引用替换为直接引用
    - 符号引用以一组符号来描述所引用的目标，在编译的时候一个每个java类都会被编译成一个class文件，但在编译的时候虚拟机并不知道所引用类的地址，多以就用符号引用来代替，而在这个解析阶段就是为了把这个符号引用转化成为真正的地址的阶段。
- 初始化：对类的静态变量初始化为指定的值，执行静态代码块
- 使用
- 卸载

Java对象的生命周期?

- 创建阶段
    - 对象的创建
- 应用阶段
    - 对象至少被一个强引用持有着
- 不可见阶段
    - 程序本身不在持有改对象的任何强引用，但是这些引用仍然是存在的
- 不可达阶段
    - 该对象不再被任何强引用所持有
- 收集阶段
    - 当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备
- 终结阶段
    - 当对象执行完finalize()方法后仍然处于不可达状态时，则该对象进入终结阶段
- 对象重新分配阶段
    - 垃圾回收器对该对象的所占用的内存空间进行回收

说说类加载器?

- 启动类加载器
- 扩展类加载器
- 应用程序类加载器
- 自定义加载器

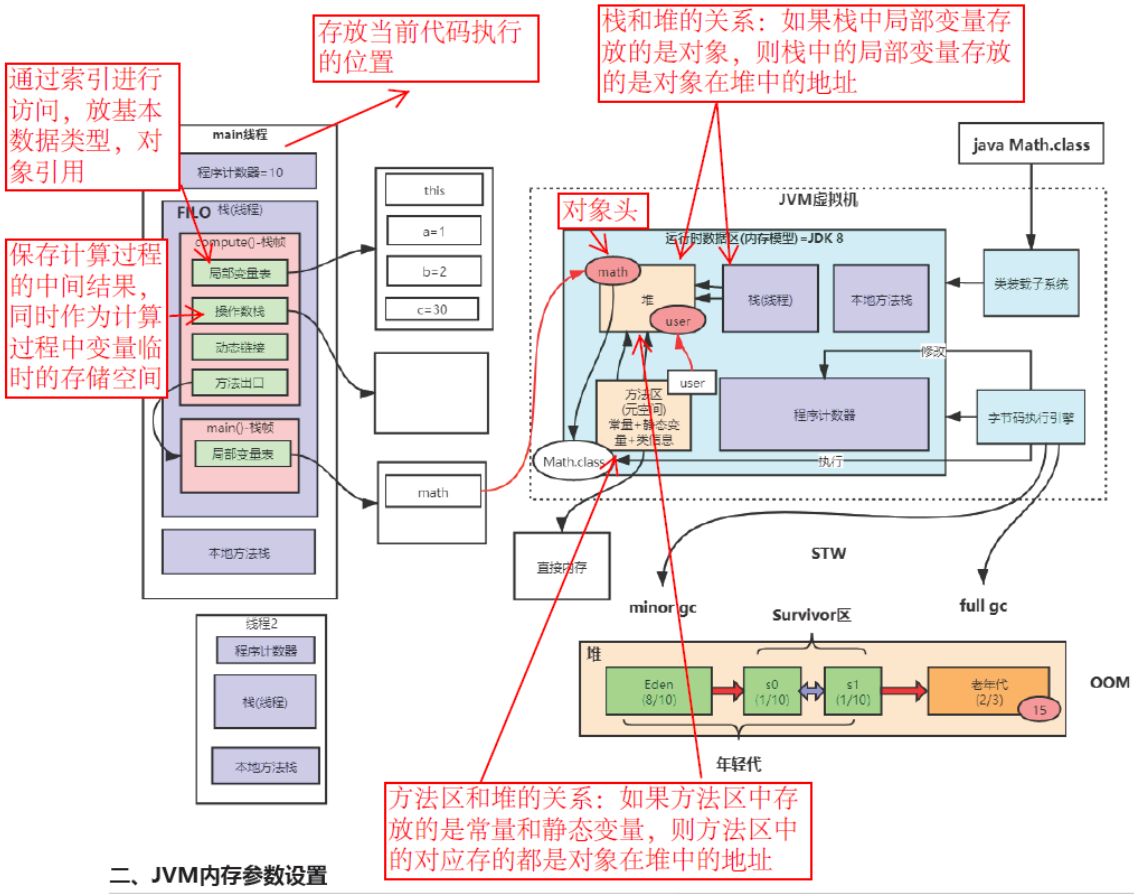
双亲委派机制?

- JVM加载某个类时，会首先委托父加载器寻找目标类，父加载器找不到再委托上层父加载器加载，如果所有父加载器在自己的加载路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类。
- **为什么要设置双亲委派机制?**

    - 沙箱安全机制：比如用户自己写的java.lang.String.Class类不会被加载，这样可以防止核心api被篡改
- 避免类的重复加载：当父亲加载了该类时，就没有必要子ClassLoader再加载一次，保证被**加载类的唯一性。**
- 自定义加载器（只需继承java.lang.ClassLoader——loadClass，一个findClass）
- **双亲委派机制源码**
    - 我们来看下应用程序类加载器AppClassLoader加载类的双亲委派机制源码，AppClassLoader的loadClass方法最终会调用其父类ClassLoader的loadClass方法，该方法的大体逻辑如下：
        1. 首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接返回。
        2. 如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加载器加载（即调用parent.loadClass(name, false);）。或者是调用bootstrap类加载器来加载。
        3. 如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的findClass方法来完成类加载。
- **为什么要打破双亲委派机制?**
    - tomcat类加载要解决的问题：
        - 比如以tomcat类加载为例，一个web容器中，可能有两个应用程序，不用应用程序依赖第三方类库的不同版本，此时就不能要求同一个类库在同一个服务器中只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。
        - web容器也有自己依赖的类库，不能与应用程序的类库混淆。基于安全考虑，应该让容器的类库和程序的类库隔离开来。
    - 在tomcat中，WebappClassLoader各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见，比如加载war包里相关的类，每个war包应用都有自己的WebappClassLoader，实现相互隔离，比如不同war包引入了不同的spring版本，这样实现就能加载各自的spring版本。
    - 结论：在tomcat进行加载类时，其中的webAppClassLoader加载自己目录下的class文件，不会传递给父加载器，打破了双亲委派机制。
- **同一个jvm中，两个相同包名和类名的对象一定不能共存吗?**
    - 不是。判断两个类对象是否是同一个，（除了包名类名一致）还要看两个对象的类加载器是不是一样的。

JVM内存模型?

- Class 文件中存放了大量的符号引用，栈中保存了一个引用，指向常量池中位置符号引用。
    - 这些符号引用一部分会在类加载阶段或第一次使用时转化为直接引用，这种转化称为**静态解析。**另一部分将在每一次运行期间转化为直接引用，这部分称为**动态连接。**
    - 符号引用以一组符号来描述所引用的目标，在编译的时候一个每个java类都会被编译成一个class文件，但在编译的时候虚拟机并不知道所引用类的地址，多以就用符号引用来代替，而在这个解析阶段就是为了把这个符号引用转化成为真正的地址的阶段。



二、JVM内存参数设置

jvm运行时数据区：虚拟机栈（局部变量表，操作数栈，动态连接，方法出口），堆（所有new出来的对象），方法区（类信息，常量，静态变量），程序计数器（存放当前代码执行的位置），本地方法栈（为虚拟机使用本地方法native服务）

新生代为什么要设置survival区?

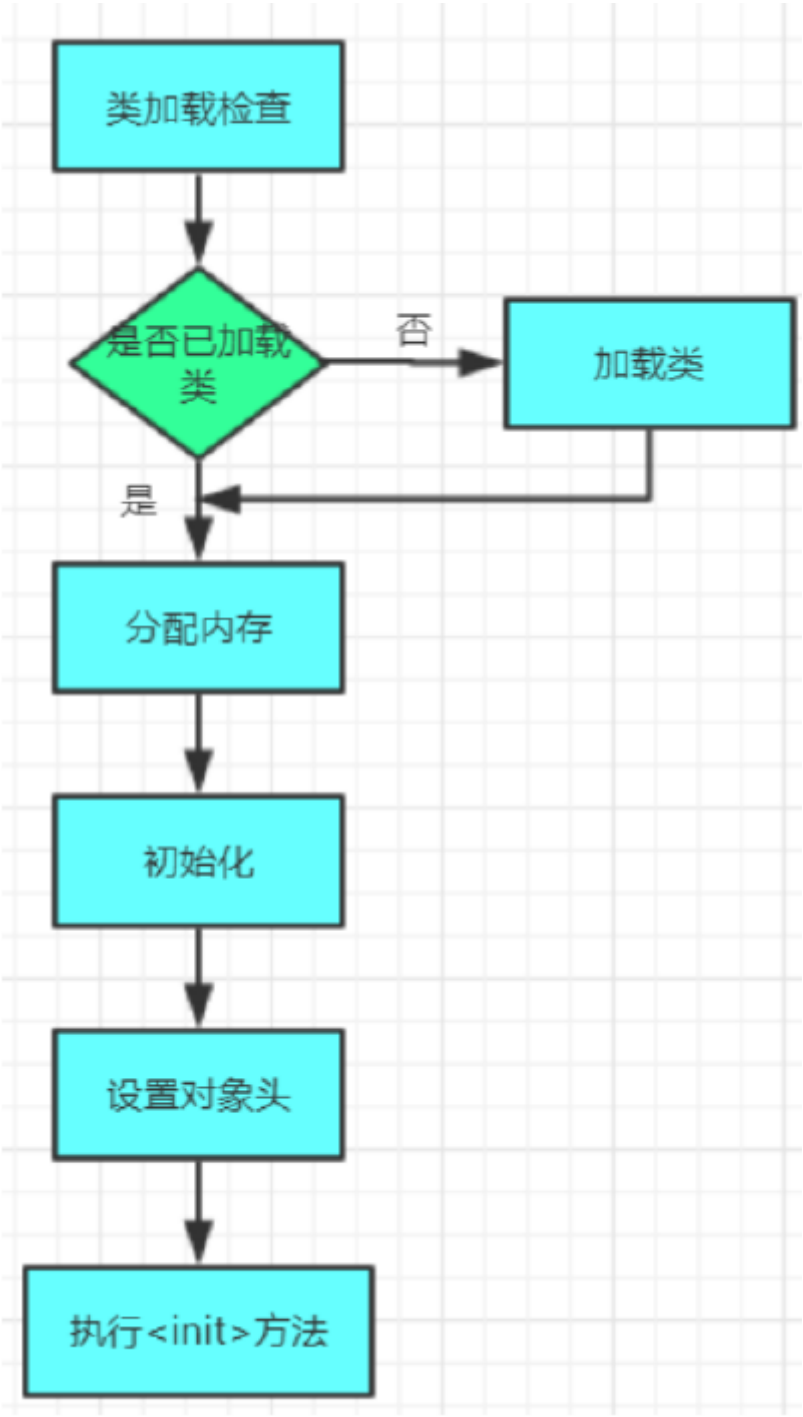
如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Full GC，频繁的触发Full Gc会影响系统的执行和响应速度。

Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证。

为什么设置两个survivor区?

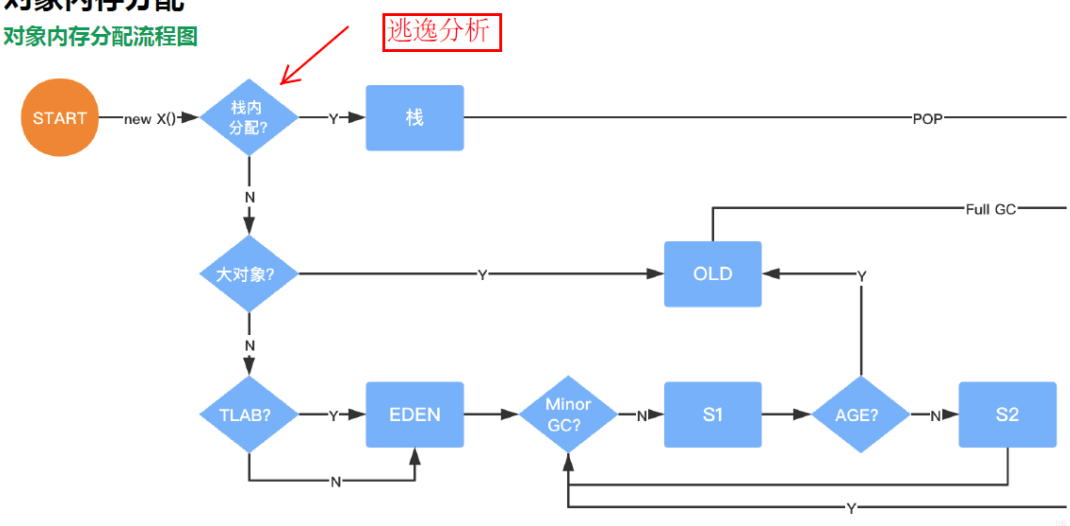
如果只有一个survivor，那么第一次gc时，eden存活的对象复制到s0中，第二次gc时，s0中也可能会有对象变成垃圾，考虑到垃圾回收的碎片化问题，我们这个时候就必须用**标记整理**算法清除垃圾对象，同时**标记整理算法**清除垃圾对象所消耗的性能远远大于使用两块survivor通过**标记复制算法**清理垃圾对象的性能。因此，我们所以需要两个survivor。

对象的创建过程?



对象内存分配

对象内存分配流程图



说一下对象在堆中的内存分配过程：

当我们new出来一个对象，在先经过类加载检查过后，就要开始给对象分配内存。首先看jvm是否开启了逃逸分析，如果是就在栈内进行分配；如果没有开启逃逸分析，则判断该对象是否是大对象，如果是大对象直接进入老年代；如果不是就在新生代的Eden区进行分配，当Eden区没有足够的空间进行分配时，虚拟机将发起一次Minor GC；对象在多次Minor GC后，在Survivor区中存活的对象年龄达到阈值时，就会被晋升到老年代。

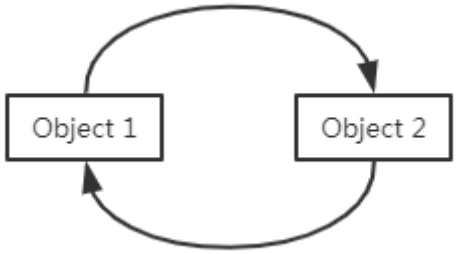
分配内存会发生OOM的情况：

- 年轻代每次minor gc之前JVM都会计算下老年代剩余可用空间，如果这个可用空间小于年轻代里现有的所有对象大小之和(包括垃圾对象)，就会看看老年代的可用内存大小，是否大于之前每一次minor gc后进入老年代的对象的平均大小。如果上一步结果是小于，那么就会触发一次Full gc，对老年代和年轻代一起回收一次垃圾，如果回收完还是没有足够空间存放新的对象就会发生"OOM"。
- 如果minor gc之后剩余存活的需要挪动到老年代的对象大小还是大于老年代可用空间，那么也会触发full gc，full gc完之后如果还是没有空间放minor gc之后的存活对象，则也会发生“OOM”。

垃圾回收

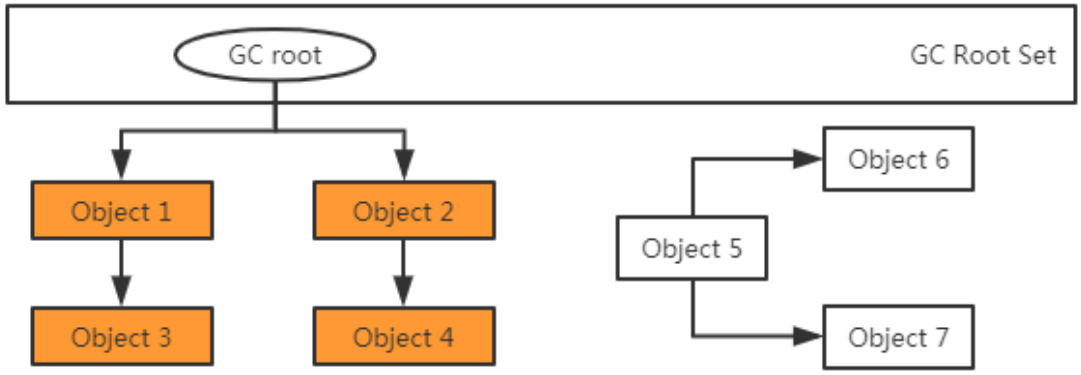
怎么判断一个对象是否是垃圾?

- **引用计数法**：在对象中添加一个引用计数器，每当一个地方用他时，计数器就加1；当引用失效时，计数器就减1；任何时刻计数器为0的对象就是不可能被使用的。



但是，如上图所示，obj1和obj2其实都可以被回收，但是他们之间有相互引用，此时各自的计数器不为0，则还是不会回收。

- **可达性分析算法**：通过一系列的GC root，将根节点作为起始节点集合，从根节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为引用链，如果某个对象没有任何不在引用链上，则就是可以被回收的对象。



你刚刚谈到了根节点，那你知道哪些对象可以作为根对象吗？

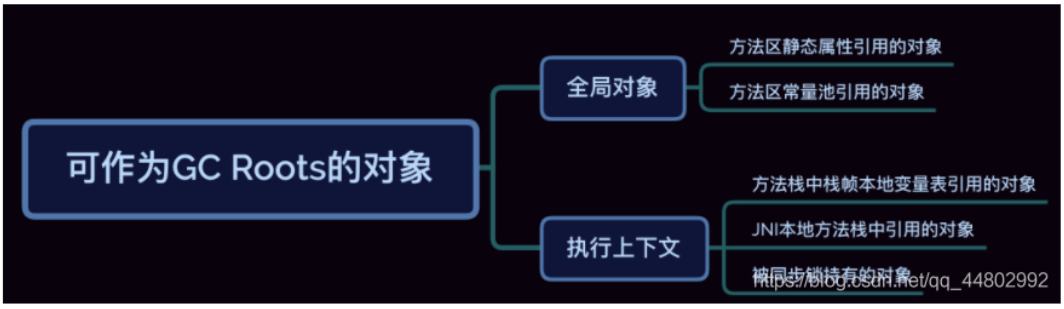
你刚刚谈到了引用，那你知道java里面有哪几种引用吗？

你刚刚谈到了可达性分析算法，那如果在该算法中被判定不可达对象，是不是一定会被回收呢？

谈谈你熟悉的垃圾回收器和他们的工作过程？

3、哪些对象可以作为GC Roots?

可以作为GC Roots的对象可以分为两大类：全局对象和执行上下文。



下面就一起来理解一下为什么这几类对象可以被作为GC Roots。

1、方法区静态属性引用的对象

全局对象的一种，Class对象本身很难被回收，回收的条件非常苛刻，只要Class对象不被回收，静态成员就不能被回收。

2、方法区常量池引用的对象

也属于全局对象，例如字符串常量池，常量本身初始化后不会再改变，因此作为GC Roots也是合理的。

3、方法栈中栈帧本地变量表引用的对象

属于执行上下文中的对象，线程在执行方法时，会将方法打包成一个栈帧入栈执行，方法里用到的局部变量会存放到栈帧的本地变量表中。只要方法还在运行，还没出栈，就意味这本地变量表的对象还会被访问，GC就不应该回收，所以这一类对象也可作为GC Roots。

4、JNI本地方法栈中引用的对象

和上一条本质相同，无非是一个是Java方法栈中的变量引用，一个是native方法(C、C++)方法栈中的变量引用。

5、被同步锁持有的对象

被synchronized锁住的对象也是绝对不能回收的，当前有线程持有对象锁呢，GC如果回收了对象，锁不就失效了嘛。

- 引用：
  - 强引用：只要强引用的关系还在，就不会被回收
  - 软引用：当系统要发生内存溢出前，才会被回收（用途：对象缓存）
  - 弱引用：只能生存到下一次垃圾回收（不论内存够否，都要被回收）（用途：对象缓存）
  - 虚引用：任何时候都有可能被垃圾回收（用途：跟踪对象被垃圾回收器回收的活动）

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	在内存不足时	对象缓存	内存不足时终止
弱引用	在垃圾回收时	对象缓存	gc运行后终止
虚引用	Unknown	Unknown	Unknown

- finalize（）方法最终**判定对象是否存活**
  - 被可达性分析算法找到后，并不是立马被回收的

- 第一次标记并进行筛选（前提是被可达性分析之后没有与GC相连的）
  - 筛选的条件是否有必要执行finalize方法，当对象没有覆盖finalize方法，对象将直接被回收
- 第二次标记
  - 当对象覆盖了finalize方法，只要重新与引用链上的任意对象关联即可
  - 如果仍然没有关联，则直接被回收

【注】：一个对象的finalize方法只会被执行一次。

垃圾收集算法

算法	年龄代	优点	缺点
标记-复制：将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。	young	不会产生碎片	1：内存空间减半；2：若是大部分对象是存活的，会产生大量的开销
标记-清除：标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象	old	节约空间	1：会产生大量碎片；2：耗时严重
标记-整理：标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象，然后再让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存	old	无碎片，空间连续	耗时（STW），需要移动对象

垃圾收集器

为什么CMS没有采用标记-复制算法?

因为传统上大家认为年老代的对象可能会长时间存活且存活率高，或者是比较大，这样拷贝起来不划算，还不如采用就地收集的方式

CMS收集器你说到是用的标记清除算法会产生内存碎片，那为什么不使用复制算法呢（答的是没有办法做到并发）？如果要你来改进复制算法，让它可以用于CMS实现并发，你怎么设计?

https://blog.csdn.net/varyall/article/details/80458095（链接，cms为什么没有采取标记复制）

CMS(老年代)收集器

- **标记过程：**
  - 初始标记：暂停其他所有的线程，只标记与GC roots对象直接关联到的对象（STW,速度很快）
  - 并发标记：从GC roots中扫描到整个对象图，不需要停顿用户线程
  - 重新标记：修正并发标记期间用户线程运行导致的标记发生变化的标记记录（STW，比并发标记时间短，比初始标记要长，主要用到三色标记里的增量更新算法）
  - 并发清除：清理删除掉标记阶段判断的已死亡的垃圾对象
  - 并发重置：重置本次GC过程中的标记数据（全部标为白色）
- **CMS存在的问题：**

    - 对**CPU资源敏感**（和服务器抢资源，cms的默认启动线程为1/4，当cpu低于四核时，cms会对用户的程序影响很大，多核cpu影响较小）
    - 无法处理**浮动垃圾**（在并发标记和并发清除阶段产生的垃圾，等待下一次GC）
    - cms使用的是"**标记-清除**"算法，会产生大量的碎片，造成**空间不连续**（可以采用开启参数-XX:UseCMSCompactAtFullCollection，在jvm执行完标记-清除后再做整理）
    - 有可能会造成**并发失败**（Concurrent mode failure，特别是在并发标记和并发清理的阶段，会触发full gc）,cms预留的内存无法（默认92%）为新对象（大对象）分配内存，此时会进入STW，会转变使用**serial old**垃圾收集器来回收。
- CMS的相关核心参数
    - -XX:+UseConcMarkSweepGC　　启用cms
    - -XX:+UseCMSCompactAtFullCollection　　FullGC后做压缩整理
    -
- 并发标记要解决的问题：由于并发标记的所需要的时间很长，让用户线程和垃圾收集器线程同时运行，增加用户的体验感。

- **并发标记阶段可能出的问题**：
    1. 并发标记底层实现原理：

       三色标记（黑色【全部已经访问过节点】），灰色【至少还有一个节点没有被访问】，白色【尚未被访问过】）
    - 并发标记导致的结果
    1. 多标——会在下一次gc进行清理
    2. 漏标——采用增量更新(Update)或者原始快照（SATB）
    - 产生漏标的两个条件（同时）：
        1. 在并发标记的过程中赋值器**插入**了一条或多条从**黑色**对象到**白色**对象的引用
        2. 在并发标记过程中赋值器器**删除**了从**灰色**对象到**白色**对象的直接或间接的引用

      【导致了在并发标记过程中新增的引用（白色对象）被回收期当作垃圾进行回收了】
    - 解决方式：

      【写屏障】：在赋值操作前后进行操作——类似于AOP

      【增量更新+写**后**屏障】：当黑色对象插入新的指向白色对象的引用时，会将这个引用记录存放到一个**容器**中，等并发扫描结束后，（重新标记）从容器中**重新扫描记录**的引用。——理解为：黑色对象插入新的白色对象的引用后，黑色对象就会变为灰色。

      【原始快照+写**前**屏障】：当灰色对象要删除指向白色对象的引用时，将这个要删除的记录存入到容器中，等并发扫描结束后，（重新标记）从容器中重新以**灰色对象为根节点扫描，将白色对象标记为黑色对象。**——这个过程中有可能白色对象没有黑色对象指向它【造成多标变成浮动垃圾，会在下一轮gc进行删除】。
- **记忆集**（remember set）与**卡表**（card table）——主要解决大规模**跨代引用**的问题**【也是通过写屏障进行的，在老年代引用新生代的数据时，把卡表对应的位置写为1】**（记忆集与卡表的关系类似于Java中Map与HashMap的关系）
    - 在**新生代**建立了**记忆集**的数据结构，一般采用**卡表**实现的，而卡表底层则是通过数组实现的，里面存储的是0101的数据（1代表的是脏数据，表示新生代里面的数据被有被老年代引用着）。**卡表**中对应着**老年代**的一块内存块（**卡页**），当有老年代引用新生代的对象时，将**卡页中对应的卡表中元素改为1**（称这个数据变脏）。在进行GC root时，除了扫描本身就在年轻代的对象外，还要扫描卡表中为1的对应着的卡页中的对象(老年代的对象)。

G1垃圾收集器

概念：G1将Java堆划分为多个大小相等的独立区域（Region），JVM最多可以有2048个Region，G1有专门分配大对象的Region叫Humongous

- 标记过程：
  - **初始标记(STW)**：只标记与GC roots对象直接关联到的对象（STW,速度很快）
  - **并发标记**：同CMS的并发标记
  - **最终标记**(STW)：同CMS的并发标记
  - **筛选回收**(STW)：该阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。比如说老年代此时有1000个Region都满了，但是因为根据预期停顿时间，本次垃圾回收可能只能停顿200毫秒，那么通过之前回收成本计算得知，可能回收其中800个Region刚好需要200ms，那么就只会回收800个Region(Collection Set，要回收的集合)，尽量把GC导致的停顿时间控制在我们指定的范围内

    【注】：回收算法用的是**复制**算法——几乎没有内存碎片
  - 并发重置：重置本次GC过程中的标记数据

G1收集器最大的特点：可以**自定义指定收集器**的停顿时间——-XX：MaxGCPalseMills(一般系统默认200ms)

- **G1的回收集列表：**

  G1收集器在**后台维护了一个优先列表**，**每次根据允许的收集时间，优先选择回收价值最大的Region**(这也就是它的名字Garbage-First的由来)，比如一个Region花200ms能回收10M垃圾，另外一个Region花50ms能回收20M垃圾，在回收时间有限情况下，G1当然会优先选择后面这个Region回收。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限时间内可以尽可能高的收集效率。

- **G1的停顿时间怎么设置?**

  一般来说， 回收阶段占到几十到一百甚至接近两百毫秒都很正常。
  - 但如果我们把停顿时间调得非常**低**， 譬如设置为二十毫秒， 很可能出现的结果就是由于停顿目标时间太短， 导致每次选出来的回收集只占堆内存很小的一部分， 收集器收集的速度逐渐跟不上分配器分配的速度， 导致垃圾慢慢堆积。 最终占满堆引发Full GC反而降低性能
  - 如果把停顿时间设置的**过大**，导致系统运行很久，年轻代可能都占用了堆内存的60%了，此时才触发年轻代gc。那么存活下来的对象可能就会很多，此时就会导致Survivor区域放不下那么多的对象，就会进入老年代中，可能就会频繁触发mixed gc。
- **G1垃圾收集分类**
  - Young GC（默认5%，不会超多60%）
    - yong gc会根据用户设置的停顿时间进行判断，当需要回收的时间远远小于设置的时间，会开辟新的region存放对象，不会马上做young gc，等到下一次eden满，同时计算的收集时间接近设定的值，才会进行gc
  - Mixed GC
    - 老年代的堆占有率达到参数（默认45%）设定的值则触发，回收所有的Young和部分Old(根据期望的GC停顿时间确定old区垃圾收集的优先顺序)以及**大对象**区正常情况G1的垃圾收集是先做MixedGC，主要使用**复制**算法，需要把各个region中存活的对象拷贝到别的region里去，拷贝过程中如果发现没有足够的空region能够承载拷贝对象就会触发一次Full GC
  - Full GC
    - 停止系统程序，然后采用单线程进行标记、清理和压缩整理

为什么G1用SATB？CMS用增量更新?

我的理解：SATB相对增量更新效率会高(当然SATB可能造成更多的浮动垃圾)，因为不需要在重新标记阶段再次**深度扫描**被删除引用对象，而CMS对增量引用的根对象会做深度扫描，G1因为很多对象都位于不同的region(可能会存在很多跨代引用)，CMS就一块老年代 区域，重新深度扫描对象的话G1的代价会比CMS高，所以G1选择SATB不深度扫描对象，只是简单标记，等到下一轮GC再深度扫描。

JVM调优案例：

jvm进行GC的时候为什么需要STW?

如果没有进行STW，有可能会导致在垃圾回收的过程中还会不断的产生新的垃圾，无法为新对象分配足够的内存。

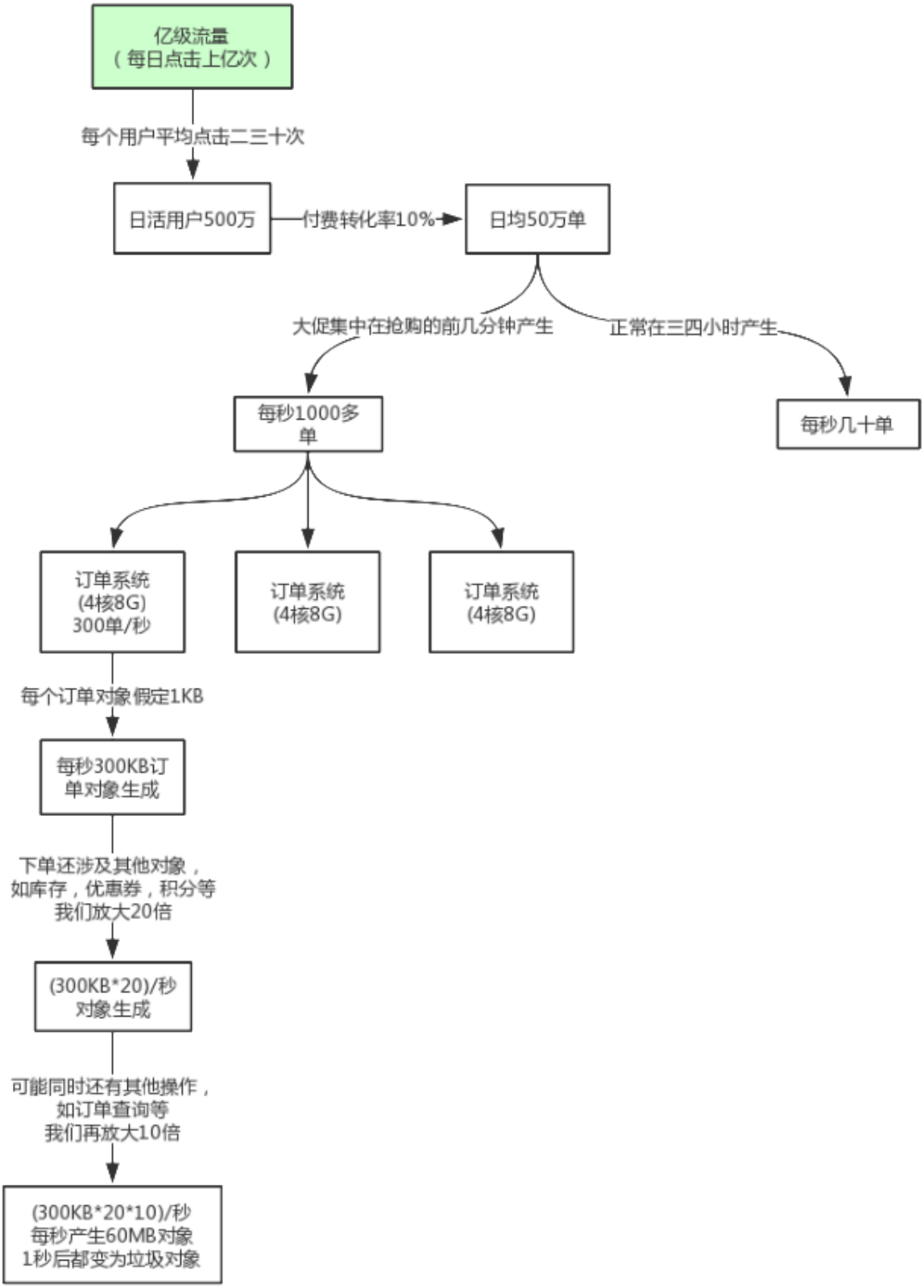
jvm调优是要调什么?

```
jvm调优是为了调优gc，主要就是两个点，一个是减少Full gc的次数，一个是缩短一次Full gc的时间
```

- 怎么设置jvm内存参数大小

像京东、淘宝、唯品会这些比较大的电商平台，都属于亿级流量电商，什么是亿级流量电商呢，就是用户在网站上每日的点击是上亿的。我对他们的交易系统做过jvm调优的分析。

- 每日点击量上亿次，每个用户平均点击二三十次，日活跃用户大概有500万，假设付费转换率为10%，则日均有50万单，如果有秒杀活动（大概半小时）则每秒会有1000多单，假设我们有三台服务器，每台服务器是4核8G的，则每台服务器并发大概有300单/秒。



对象在内存中的组成，有对象头，实例数据以及对齐填充，其中主要的大小时实例数据占用的，一个对象有很多字段，一个字段占几个字节，一个对象撑死几百个字段，我们这里假设一个订单对象大小1KB，就是1024个字节，这已经算比较大的，一般对象不会超过这么大，

那么接着上面的推算，每秒就会有300KB的订单对象生成，但是这里只是估算，下单过程中不仅仅会产生订单对象 ，还会涉及库存，优惠券，积分等其他相关对象，既然是估算，我们肯定要放大一下才可靠。

假设我们这里放大20倍，那么每秒就会有300KB*20也就是6M的对象产生。但是订单系统也不只提供下单操作，可能还有订单查询，订单退款等操作，我们再把这业务放大10倍，那么每秒就会产生60M的对象放入伊甸园区，而且这些对象有一个特点，当我的订单对象一旦生成完毕之后，在堆中生成的对象都会变为垃圾对象，都应该被回收掉。

我们这里假设机器是4核8G，那么一般可能会给我们虚拟机分配个4G的内存，就会给堆分配个3G的内存，那么方法区分配256M，单个栈内存分配1M。

如果没有进行调优，会发生什么问题？

根据之前我们讲的jvm内存分配，我们可以得知各个区域的所占内存，每秒有60M对象产生进去Eden区，那么13秒就会占满Eden区域，发生minor gc，这些订单对象其中90%其实已经是垃圾对象了，为什么说90%，因为在gc的那一刻，这些对象肯定还有一部分的业务操作还没有完成，所以他们不会被回收，我们这里假设每次minor gc都有60M对象还不会被回收。

- 那么这60M对象会从Eden区移到S0区域，我们之前的文章有说过对象进入老年代有很多条件，比如较大的对象，这里的60M虽然小于100M，同样会直接被移入老年代，为什么呢？因为还有一个条件是对象动态年龄判断，就是如果你移动的这批对象超过了Survivor区的50%，同样会把这批对象移入老年代。

- 那么按现在这种参数设置，每13秒就会有60M的对象被移入老年代，那么大概五六分钟老年代的2G就会被占满，那么老年代一满就要发生full gc，但是full gc使我们最不愿意看到的结果。对于这么大的一个系统，每过五六分钟就发生一次full gc，就会让系统卡顿一次，用户体验很差，这是很大的问题。

调优方案：将年轻代的参数设置的大一些

当我们进行参数修改之后，你再来按上面的分析来分析这个过程，你会发现一个奇迹效果。

此时需要25秒把伊甸园区放满，放满minor gc后有60M对象不被回收，要移到S0区，这时60M<200M/2，是可以移入S0区域的，下一次伊甸园区再放满做minor gc的时候，这时这60M对象所对应的订单已经生成了，已经变成了垃圾对象，是可以直接被回收的，所以没有什么对象是需要被移入老年代的。

那么这么一设置的话，这个系统是不是正常情况下基本不会再发生full gc了呢？就算发生，也是很久才会一次了。

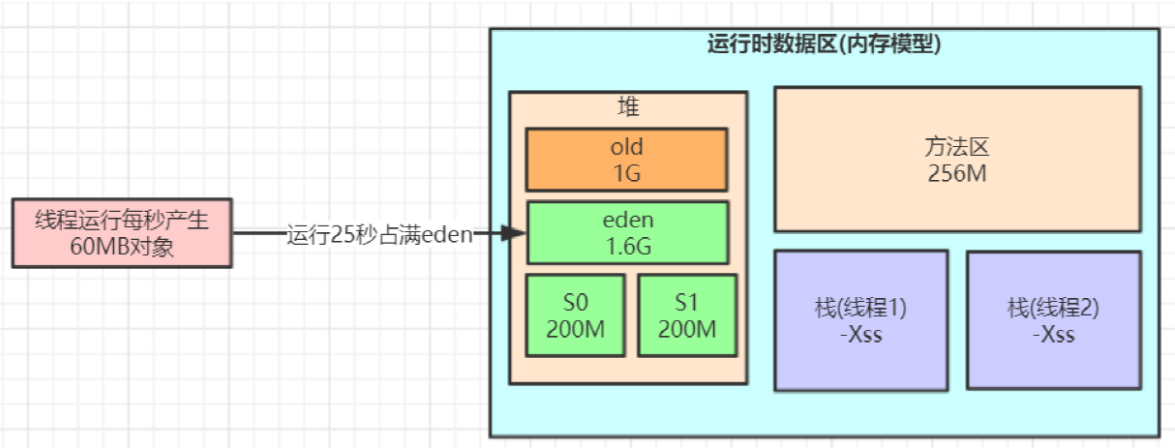
+--------------------CMS+PareNew----------------------

如果在300单/秒的基础上增加到500单/秒，那么每秒生成的对象可能有100多M，再加上整个系统可能压力剧增，一个订单要好几秒才能处理完，下一秒可能又有很多订单过来。还有就是某次minor gc完了之后还有超过100-200M的对象存活，那么就会直接进入老年代，我们可以估算下大概每隔五六分钟出现一次这样的情况，那么大概半小时到一小时之间就可能因为老年代满了触发一次Full GC。其实在半小时后发生full gc，这时候已经过了抢购的最高峰期，后续可能几小时才做一次Full GC，再进行碎片的整理。

可以设置对象年龄为5次（很少有对象能存活5次，一般就是Spring的Bean对象，线程池对象，缓存对象），设置大对象参数为1M（很少有超多1M的大对象，一般是系统初始化分配的缓存对象），设置cms的老年代使用的占比（92），设置Full gc后的碎片整理，设置每过full gc后进行碎片整理。

使用CMS进行调优参数设定：

-Xms3072M -Xmx3072M -Xmn2048M -Xss1M -XX:MetaspaceSize=256M
-XX:MaxMetaspaceSize=256M -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=5
-XX:PretenureSizeThreshold=1M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC
-XX:CMSInitiatingOccupancyFraction=92 -XX:+UseCMSCompactAtFullCollection（开始整理碎片） -XX:CMSFullGCsBeforeCompaction=0（每次full gc就会整理）



调优策略：**就是尽可能让对象都在新生代里分配和回收，尽量别让太多对象频繁进入老年代，避免频繁对老年代进行垃圾回收，同时给系统充足的内存大小，避免新生代频繁的进行垃圾回收。**

垃圾收集器的选择

- Serial单线程收集，stw
- Parallel多线程收集，默认cpu核数的线程，stw
- jdk8默认的垃圾收集器：ParallelGC(年轻代)和ParallelOldGC(老年代)；**如果内存较大(超过4个G，只是经验值)，系统对停顿时间比较敏感，我们可以使用ParNew+CMS**
- **JDK 1.8默认使用 Parallel(年轻代和老年代都是)**
- **JDK 1.9默认使用 G1**

什么场景适合使用G1

1. 50%以上的堆被存活对象占用
2. 对象分配和晋升的速度变化非常大
3. 垃圾回收时间特别长，超过1秒
4. 8GB以上的堆内存(建议值)
5. 停顿时间是500ms以内

如何选择垃圾收集器

1. 优先调整堆的大小让服务器自己来选择
2. 如果内存小于100M，使用串行收集器
3. 如果是单核，并且没有停顿时间的要求，串行或JVM自己选择
4. 如果允许停顿时间超过1秒，选择并行或者JVM自己选
5. 如果响应时间最重要，并且不能超过1秒，使用并发收集器
6. 4G以下可以用parallel，4-8G可以用ParNew+CMS，8G以上可以用G1，几百G以上用ZGC