jvm参数含义：

-XX:NewSize=209715200 （设置年轻代内存大小）

-XX:MaxNewSize=209715200 （设置年轻代内存最大值）

-XX:InitialHeapSize=314572800 （设置堆内存大小）

-XX:MaxHeapSize=314572800 （设置堆内存最大值）

-XX:SurvivorRatio=2 （设置Eden区和Survivor区比例，此处为2:1:1，设置为8就是8:1:1）

-XX:MaxTenuringThreshold=15 （设置对象躲过N次Y Gc之后直接进入OC）

-XX:PretenureSizeThreshold=20971520 （设置大对象的阈值，超过则直接进入OC）

-XX:MetaspaceSize=10m

-XX:MaxMetaspaceSize=10m

-XX:ThreadStackSize=128k （设置栈的深度）

-XX:+UseParNewGC （Young GC采用的垃圾回收器）

-XX:+UseConcMarkSweepGC （Full GC采用的垃圾回收器）

-XX:+HeapDumpOnOutOfMemoryError （发生OOM时自动生成内存快照）

-XX:HeapDumpPath=/usr/local/app/oom （发生OOM时生成的内存快照存储路径）

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:gc.log （GC日志路径）

-XX:CMSInitiatingOccupancyFraction=92 （当老年代达到70%时，触发CMS垃圾回收）

-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 （设置N次full Gc后整理内存碎片，此处为0代表每次full GC后都进行碎片整理）

-XX:+CMSParallelInitialMarkEnabled （CMS垃圾回收器的“初始标记”阶段开启多线程并发执行）

-XX:+CMSScavengeBeforeRemark （这个参数会在CMS的重新标记阶段之前，先尽量执行一次Young GC）

-XX:SoftRefLRUPolicyMSPerMB=0 （这个参数设置大一些即可，千万别让一些新手同学设置为0，可以设置个1000，2000，3000，或者5000毫秒，都可以。**因为SoftReference的回收公式为clock - timestamp <= freespace * SoftRefLRUPolicyMSPerMB，一旦设置为0，则公式右边就为0，意味着对象刚被创建就要被回收，这样反而会造成不停的创建。**提高这个数值，就是让反射过程中JVM自动创建的

软引用的一些类的Class对象不要被随便回收，当时我们优化这个参数之后，就可以看到系统稳定运行了。）

-XX:+DisableExplicitGC （禁止显式执行GC，不允许你来通过代码触发GC）

-XX:CMSMaxAbortablePrecleanTime （置等待多久没有等到young gc就强制remark。默认是5s。）

-Xss （栈的深度，如果方法调用比较深就需要调大一点。例如代码中有递归）

-XX:+HeapDumpOnOutOfMemoryError （在OutOfMemoryError后获取一份HPROF二进制Heap Dump文件）

-XX:+PrintHeapAtGC（打印推信息， 获取Heap在每次垃圾回收前后的使用状况）

jstat -gc PID执行结果：

S0C：这是From Survivor区的大小

S1C：这是To Survivor区的大小

S0U：这是From Survivor区当前使用的内存大小

S1U：这是To Survivor区当前使用的内存大小

EC：这是Eden区的大小

EU：这是Eden区当前使用的内存大小

OC：这是老年代的大小【Old Capacity】

OU：这是老年代当前使用的内存大小【Old Used】

MC：这是方法区（永久代、元数据区）的大小【MetaSpace Capacity】

MU：这是方法区（永久代、元数据区）的当前使用的内存大小【MetaSpace Used】

CCSC：压缩类空间大小【Compress Class Space Capacity】

CCSU：压缩类空间使用大小【Compress Class Space Used】

YGC：这是系统运行迄今为止的Young GC次数

YGCT：这是Young GC的耗时

FGC：这是系统运行迄今为止的Full GC次数

FGCT：这是Full GC的总耗时

GCT：这是所有Gc的耗时

查看堆内存具体分配情况(已使用信息不显示)：

jstat -gccapacity PID：堆内存分析

jstat -gcnew PID：年轻代GC分析，这里的TT和MTT可以看到对象在年轻代存活的年龄和存活的最大年龄

jstat -gcnewcapacity PID：年轻代内存分析

jstat -gcold PID：老年代GC分析
jstat -gcoldcapacity PID：老年代内存分析
jstat -gcmetacapacity PID：元数据区内存分析

使用技巧：

1、jstat -gc PID 1000 10：每隔1秒钟更新出来最新的一行jstat统计信息，一共执行10次jstat统

计。

2、**jmap -dump:live,format=b,file=dump.hprof PID**导出内存快照。

3、**jhat -port 7000 dump.hprof （**在浏览器上访问当前这台机器的7000端口号，就可以通过图形化的方式去分析堆内存里的对象分布情况了**）。**

总结：

1、如果系统运算时间比较长，导致对象的年龄比较大，可以适当调大"-XX:MaxTenuringThreshold"，使对象年龄大一些再进入老年代，这样也可以减少进入老年代的对象；

2、QPS(Query Per Second),每秒钟的查询数量;

3、TPS(Transactions Per Second).QPS（TPS）= 并发数/平均响应时间　或者　并发数 = QPS*平均响应时间;

4、如果你在代码里大量用了类似上面的反射的东西，那么JVM就是会动态的去生成一些类放入Metaspace区域里的。

5、简单来说，每个类其实本身自己也是一个对象，就是一个Class对象，一个Class对象就代表了一个类。同时这个Class对象代表的类，可以派生出来很多实例对象。举例来说，Class Student，这就是一个类，他本身是由一个Class类型的对象表示的。

6、**类的信息是放在Metaspace的，但是类的Class对象是放在堆内存的。类有两份信息，一个是类的信息放meta区，一个是类的Class对象放在堆区。**

7、System.gc()主要是建议gc，但是不一定会gc，但是确实可能会提升gc频率。建议别使用。

8、Full GC频繁的原因可能有以下几种：

a. 内存分配不合理，导致对象频繁进入老年代，进而引发频繁的Full GC；

b. 系统一次性加载过多数据进内存，搞出来很多大对象，导致频繁有大对象进入老年代，必然频繁触发Full GC；

c. 存在内存泄漏等问题，就是内存里驻留了大量的对象塞满了老年代，导致稍微有一些对象 进入老年代就会引发Full GC；

d. 永久代（MetaSpace）里的类太多，触发了Full GC；

e. 工程师错误的执行“System.gc()”；

9、Young GC的整个过程中系统是禁止运行的，处于Stop the World状态。负责Young GC的垃圾回收器有很多种，但是常用的就是ParNew垃圾回收器，他的核心执行原理就如上所述，只不过他运行的时候是基于多线程并发执行垃圾回收的。

10、导致对象进入老年代的几种情况：

a. 一个对象在年轻代里躲过15次垃圾回收，年龄太大了，寿终正寝，进入老年代；

b. 对象太大了，超过了一定的阈值，直接进入老年代，不走年轻代；

c. 一次Young GC过后存活对象太多了，导致Survivor区域放不下了，这批对象会进入老年代；

d. 可能几次Young GC过后，Surviovr区域中的对象占用了超过50%的内存，此时会判断如果年龄1+年龄2+年龄N的对象总和超过了Survivor区域的50%，此时年龄N以及之上的对象都进入老年代，这是动态年龄判定规则；

Tip：一般情况下躲过15次GC是因为系统处理比较慢，大对象一般在特殊情况下会有，对于
那种加载大量数据长时间处理以及高并发的场景，很容易导致Young GC后存活对象过多的。

11、一旦老年代对象过多，就可能会触发Full GC，Full GC必然会带着Old GC，也就是针对老年代的GC，而且一般会跟着一次Young GC，也会触发永久代的GC。

12、Full GC触发的条件：

a. 老年代自身可以设置一个阈值，有一个JVM参数可以控制，一旦老年代内存使用达到这个阈值，就会触发Full GC，一般建议调节大

一些，比如92%；

b. 在执行Young GC之前，如果判断发现老年代可用空间小于了历次Young GC后升入老年代的平均对象大小的话，那么就会在Young GC之前触发Full GC，先回收掉老年代一批对象，然后再执行Young GC；

c. 如果Young GC过后的存活对象太多，Survivor区域放不下，就要放入老年代，要是此时老年代也放不下，就会触发Full GC，回收老年代一批对象，再把这些年轻代的存活对象放入老年代中；

d. 触发Full GC几个比较核心的条件就是这几个，总结起来，其实就是老年代一旦快要搞满了，空间不够了，必然要垃圾回收一次；

13、一般Full GC的表象如下：

a. 机器CPU负载过高；

b. 频繁Full GC报警；

c. 系统无法处理请求或者处理过慢。

案例：

1、-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=5参数设置过大，导致每次Full GC之后产生很多内存碎片，从而使老年代可用连续内存减小导致更加频繁的Full GC。

解决方案：-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0，每次FUll Gc之后都进行一次碎片整理，虽然会使一次Full GC的耗时加长，但是会使Full GC的频率变小。

2、使用默认的jvm参数导致内存分配过小。

解决方案：定制一套jvm参数模板，开启参数 -XX:+CMSParallelInitialMarkEnabled，-XX:+CMSScavengeBeforeRemark，提升Full

GC效率。

3、-XX:SurvivorRatio=5,-XX:CMSInitiatingOccupancyFraction=68，Eden区过小，老年代内存使用率过小。并且系统运行一段时间老年代就会突然出现几百MB的数据，排查代码发现有select * from table操作，全表查导致大对象产生，从而产生了频繁的额Full GC。

解决方案：调大两个参数的值，然后解决代码Bug。

4、线上系统大促的时候几乎每秒一次Full GC导致系统卡死，但是jstat分析发现各区内存占用正常，最终发现原因是代码里边写了System.Gc()导致大促的时候频繁调用此语句从而提高了Full Gc发生的概率。

解决方案：设置-XX:+DisableExplicitGC参数禁止手动gc。

5、在系统里做了一个JVM本地的缓存，把很多数据都加载到内存里去缓存起来，然后提供查询服务的时候直接从本地内存走。但是因为没有限制本地缓存的大小，并且没有使用LRU之类的算法定期淘汰一些缓存里的数据，导致缓存在内存里的对象越来越多，进而造成了内存泄漏。使用MAT的Leak(泄漏)Suspects(嫌疑的)分析内存快照。

解决方案：只要使用类似EHCache之类的缓存框架就可以了，他会固定最多缓存多少个对象，定期淘汰删除掉一些不怎么访问的缓存，以便于新的数据可以进入缓存中。

6、将查出来的几十万条数据用String.split()方法切割以后产生大量对象。

解决方案：用jmap生成快照文件，通过MAT进行快照文件分析找到问题根源。

一个类从加载到使用，一般会经历哪些过程（加载->验证->准备->解析->初始化->使用->卸载）：

a.加载:将编译好的".class"字节码文件加载到JVM中；

b.验证:根据JVM规范，校验加载进来的".class"字节码文件；

c.准备:给类和类变量分配一定的内存空间，且给类变量设置默认的初始值(0或者nul)；

d.解析:把符号引用替换为直接引用的过程；

e.初始化:根据类初始化代码给类变量赋值；

注：执行new函数来实例化类对象会触发类加载到初始化的全过程；或者是包含"main()"方 法的主类，必须是立马初始化的。如果初始化一个类的时候，发现他的父类还没初始化， 那么必须先初始化他的父类。

Java里有哪些类加载器：

1. 启动类加载器: 主要负责加载我们在机器上安装的Java目录(lib目录)下的核心类库；
2. 扩展类加载器：主要负责加载Java目录下的"lib/ext"目录中得类；
3. 应用程序类加载器：主要负责加载"ClassPath"环境变量所指定的路径中的类，大致 可以理解为加载我们写好的java代码；
4. 自定义类加载器：根据自己的需求加载类；

什么是双亲委派机制：

JVM的类加载器是有亲子层级结构的，启动类加载器最上层，扩展类加载器第二层，应用程序类加载器第三层，自定义类加载器第四层。当应用程序类加载器需要加载一个类时， 他会先委派给自己的父类加载器去加载，最终传导到顶层的类加载器去加载，但是如果父类 加载器在自己负责加载的范围内，没找到这个类，那么就会下推加载权利给自己的子类加载器。

JVM中有哪些内存区域：

1. 方法区: 在JDK1.8以后,这块区域的名字改了,叫做"Metaspace"。主要存放我们自己写的 各种类相关的信息；
2. 程序计数器：字节码指令通过字节码执行引擎被一条一条执行，才能实现我们写好的代码 执行的效果，程序技术器就是用来记录当前执行的字节码指令的位置，也就是记录目前执行到了哪一条字节码指令，JVM是支持多个线程的，所以就会 有多个线程来并发执行不同的代码指令，因此，每个线程都会有自己的一个 程序计数器，专门记录当前这个线程目前执行到了哪一条字节码指令；
3. Java虚拟机栈：保存每个方法内的局部变量等数据。每个线程都会有自己的Java虚拟机栈；
4. 如果线程执行了一个方法，就会对这个方法调用创建对应的一个栈帧(栈帧 里就有这个方法的局部变量表,操作数栈,动态链接,方法出口等)，然后压入 线程的Java虚拟机栈。方法执行完毕之后就从Java虚拟机栈出栈。因此，每个线程在执行代码时，除

了程序计数器以外，还搭配了一个Java虚拟机 内存区域来存放每个方法中得局部变量。

5. Java堆内存：存放我们在代码中创建的各种对象。对象实例里面会包含一些数据。而Java 虚拟机栈的栈帧局部变量表里面的对象，其实是一个引用类型的局部变量，存放 了对应Java堆内存对象的地址。可以理解为局部变量表里的引用指向了Java堆内存中的对象。

发生OOM(OutOfMemory)区域：

1. Metaspace区域是用来存放类信息的，**那是不是有可能在这个Metaspace区域里就会发生OOM；**
2. 每个线程的虚拟机栈的内存大小是固定的，**第二块可能发生OOM的区域，就是每个线程的虚拟机栈内存；**
3. 在JVM中分配给堆内存的空间其实一般是固定的，**第三块可能发生内存溢出的区域，就是堆内存空间。**

问题：“第二个案例，生成新的类.class文件不是应该一样的吗？不是只有一份吗？” 这个问题也已经弄明白。原因就是虽然是相同的类，但是每次都是动态生成，所以一个相同类生成了很多副本，这些副本的类信息都保存在metaspace，所以造成了metaspace的oom了。

MAT工具使用方式：https://blog.csdn.net/zhanshenzhi2008/article/details/89070049