1. Full gc 次数太多了如何优化？

首先可以检查是不是内存泄漏导致的full gc；

首先用top -u查询当前进程那个cpu占用率最高

Jvm 分区

方法区、堆内存、虚拟机栈、本地方法栈、程序计数器。

1. 方法区

方法区主要用于存储虚拟机加载的类信息、常量、静态变量，以及编译器编译后的代码等数据。

在jdk1.8中，方法区已经不存在，原方法区中存储的类信息、编译后的代码数据等已经移动到了元空间（MetaSpace）中，元空间并没有处于堆内存上，而是直接占用的本地内存。

去永久代的原因有：

（1）字符串存在永久代中，容易出现性能问题和内存溢出。

（2）类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

（3）永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

2. 堆内存

堆内存主要用于存放对象和数组，它是JVM管理的内存中最大的一块区域，堆内存和方法区都被所有线程共享，在虚拟机启动时创建。在垃圾收集的层面上来看，由于现在收集器基本上都采用分代收集算法，因此堆还可以分为新生代（YoungGeneration）和老年代（OldGeneration），新生代还可以分为Eden、From Survivor、To Survivor。

3、程序计数器

程序计数器是一块非常小的内存空间，可以看做是当前线程执行字节码的行号指示器，每个线程都有一个独立的程序计数器，因此程序计数器是线程私有的一块空间，此外，程序计数器是Java虚拟机规定的唯一不会发生内存溢出的区域。

4、虚拟机栈

虚拟机栈也是每个线程私有的一块内存空间，它描述的是方法的内存模型，虚拟机会为每个线程分配一个虚拟机栈，每个虚拟机栈中都有若干个栈帧，每个栈帧中存储了局部变量表、操作数栈、动态链接、返回地址等。一个栈帧就对应Java代码中的一个方法，当线程执行到一个方法时，就代表这个方法对应的栈帧已经进入虚拟机栈并且处于栈顶的位置，每一个Java方法从被调用到执行结束，就对应了一个栈帧从入栈到出栈的过程。

5、本地方法栈

本地方法栈与虚拟机栈的区别是，虚拟机栈执行的是Java方法，本地方法栈执行的是本地方法（Native Method）,其他基本上一致

Jvm内存溢出与内存泄漏

1. 堆内存溢出

堆内存中主要存放对象、数组等，只要不断地创建这些对象，当这些对象所占空间超过最大堆容量时，就会产生OutOfMemoryError的异常

新产生的对象最初分配在新生代，新生代满后会进行一次Minor GC，如果Minor GC后空间不足会把该对象和新生代满足条件的对象放入老年代，老年代空间不足时会进行Full GC，之后如果空间还不足以存放新对象则抛出OutOfMemoryError异常。

常见的场景有：如果一次从数据库中加载过多的数据。

2、虚拟机栈/本地方法栈溢出

（1）StackOverflowError：当线程请求的栈的深度大于虚拟机所允许的最大深度

常见场景：无线递归某个方法

内存泄露 **memory leak**，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

内存泄漏最终会导致内存溢出。

Java内存泄漏的主要原因：**长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用导致不能被回收。**

1. 静态集合类引起的内存泄漏

像HashMap,Vector等的使用最容易出现内存泄漏，这些静态变量的生命周期和应用程序一致，他们所引用的所有对象Object也不能被释放。

1. **当集合里面对象的属性被修改后，再调用remove（）方法时不起作用。**
2. 监听器 在删除对象的时候没有删除这些监听器，从而增加了内存泄漏的机会。
3. 各种连接

**比如数据库连接，网络连接和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会被自动gc回收的。**

1. 单例模式

如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露。不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露，考虑下面的例子：

JVM垃圾回收

垃圾回收涉及到的内容有：1、判断对象是否已死；2、选择垃圾收集算法；3、选择垃圾收集的时间；4、选择适当的垃圾收集器清理垃圾

1. 判断对象是否已死

引用计数法

给每一个对象添加一个引用计数器，每当有一个地方引用它时，计数器值加1；每当有一个地方不再引用它时，计数器值减1，这样只要计数器的值不为0，就说明还有地方引用它，它就不是无用的对象。

但是两个无用的对象之间相互引用就会导致无法回收

1. 可达性分析

GC Roots，垃圾收集的起点，可以作为GC Roots的有（虚拟机栈中（本地变量表）中引用的对象、方法区中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JNI（Native方法）引用的对象。）

当一个对象到GC Roots没有任何引用链相连（GC Roots到这个对象不可达）时，就说明此对象是不可用的，是死对象。

1. 方法区回收

上面说的都是对堆内存中对象的判断，方法区中主要回收的是废弃的常量和无用的类。 判断常量是否废弃可以判断是否有地方引用这个常量，如果没有引用则为废弃的常量。

判断类是否废弃需要同时满足如下条件：

该类所有的实例已经被回收（堆中不存在任何该类的实例）

加载该类的ClassLoader已经被回收

该类对应的java.lang.Class对象在任何地方没有被引用（无法通过反射访问该类的方法）

2. 垃圾回收算法

常用的垃圾回收算法有三种：标记-清除算法、复制算法、标记-整理算法。

1. 标记-清除算法：分为标记和清除两个阶段，首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。缺点是产生大量碎片化的空间。
2. 复制算法：把内存分为大小相等的两块，每次存储只用其中一块，当这一块用完了，就把存活的对象全部复制到另一块上，同时把使用过的这块内存空间全部清理掉，往复循环。

缺点：实际可使用的内存空间缩小为原来的一半，比较适合。

1. 标记-整理算法：先对可用的对象进行标记，然后所有被标记的对象向一段移动，最后清除可用对象边界以外的内存
2. 分代收集算法：把堆内存分为新生代和老年代，新生代又分为Eden区、From Survivor和To Survivor。一般新生代中的对象基本上都是朝生夕灭的，每次只有少量对象存活，因此采用复制算法，只需要复制那些少量存活的对象就可以完成垃圾收集；老年代中的对象存活率较高，就采用标记-清除和标记-整理算法来进行回收。

在这些区域的垃圾回收大概有如下几种情况：

大多数情况下，新的对象都分配在Eden区，当Eden区没有空间进行分配时，将进行一次Minor GC，清理Eden区中的无用对象。清理后，Eden和From Survivor中的存活对象如果小于To Survivor的可用空间则进入To Survivor，否则直接进入老年代）；Eden和From Survivor中还存活且能够进入To Survivor的对象年龄增加1岁（虚拟机为每个对象定义了一个年龄计数器，每执行一次Minor GC年龄加1），当存活对象的年龄到达一定程度（默认15岁）后进入老年代，可以通过-XX:MaxTenuringThreshold来设置年龄的值。

当进行了Minor GC后，Eden还不足以为新对象分配空间（那这个新对象肯定很大），新对象直接进入老年代。

占To Survivor空间一半以上且年龄相等的对象，大于等于该年龄的对象直接进入老年代，比如Survivor空间是10M，有几个年龄为4的对象占用总空间已经超过5M，则年龄大于等于4的对象都直接进入老年代，不需要等到MaxTenuringThreshold指定的岁数。

在进行Minor GC之前，会判断老年代最大连续可用空间是否大于新生代所有对象总空间，如果大于，说明Minor GC是安全的，否则会判断是否允许担保失败，如果允许，判断老年代最大连续可用空间是否大于历次晋升到老年代的对象的平均大小，如果大于，则执行Minor GC，否则执行Full GC

4、常见垃圾收集器

现在常见的垃圾收集器有如下几种

新生代收集器：Serial、ParNew、Parallel Scavenge

老年代收集器：Serial Old、CMS、Parallel Old

堆内存垃圾收集器：G1

（1）Serial 收集器

Serial是一款用于新生代的单线程收集器，采用复制算法进行垃圾收集。Serial进行垃圾收集时，不仅只用一条线程执行垃圾收集工作，它在收集的同时，所有的用户线程必须暂停（Stop The World）。Serial收集器以单线程，采用复制算法进行垃圾收集工作，收集完之后，用户线程继续开始执行。

2）ParNew 收集器

ParNew就是一个Serial的多线程版本，ParNew收集器以多线程，采用复制算法进行垃圾收集工作，收集完之后，用户线程继续开始执行。

3）Parallel Scavenge 收集器

Parallel Scavenge也是一款用于新生代的多线程收集器，与ParNew的不同之处是，ParNew的目标是尽可能缩短垃圾收集时用户线程的停顿时间.

在新生代，当用户线程都执行到安全点时，所有线程暂停执行，ParNew收集器以多线程，采用复制算法进行垃圾收集工作，收集完之后，用户线程继续开始执行；在老年代，当用户线程都执行到安全点时，所有线程暂停执行，Parallel Old收集器以多线程，采用标记整理算法进行垃圾收集工作。

（5）CMS(Concurrent Mark Sweep) 收集器

CMS收集器是一种以最短回收停顿时间为目标的收集器，以“最短用户线程停顿时间”著称。整个垃圾收集过程分为4个步骤：

① 初始标记：标记一下GC Roots能直接关联到的对象，速度较快

② 并发标记：进行GC Roots Tracing，标记出全部的垃圾对象，耗时较长

③ 重新标记：修正并发标记阶段引用户程序继续运行而导致变化的对象的标记记录，耗时较短

④ 并发清除：用标记-清除算法清除垃圾对象，耗时较长

整个过程耗时最长的并发标记和并发清除都是和用户线程一起工作，所以从总体上来说，CMS收集器垃圾收集可以看做是和用户线程并发执行的。

CMS收集器也存在一些缺点：

对CPU资源敏感：默认分配的垃圾收集线程数为（CPU数+3）/4，随着CPU数量下降，占用CPU资源越多，吞吐量越小

无法处理浮动垃圾：在并发清理阶段，由于用户线程还在运行，还会不断产生新的垃圾，CMS收集器无法在当次收集中清除这部分垃圾。同时由于在垃圾收集阶段用户线程也在并发执行，CMS收集器不能像其他收集器那样等老年代被填满时再进行收集，需要预留一部分空间提供用户线程运行使用。当CMS运行时，预留的内存空间无法满足用户线程的需要，就会出现“Concurrent Mode Failure”的错误，这时将会启动后备预案，临时用Serial Old来重新进行老年代的垃圾收集。

因为CMS是基于标记-清除算法，所以垃圾回收后会产生空间碎片，可以通过-XX:UserCMSCompactAtFullCollection开启碎片整理（默认开启），在CMS进行Full GC之前，会进行内存碎片的整理。还可以用-XX:CMSFullGCsBeforeCompaction设置执行多少次不压缩（不进行碎片整理）的Full GC之后，跟着来一次带压缩（碎片整理）的Full GC。

（7）G1 收集器

G1 收集器是jdk1.7才正式引用的商用收集器，它采用“化整为零”的思路，把整个堆内存划分为多个大小相等的独立区域（Region），每一个方块就是一个区域，每个区域可能是Eden、Survivor、老年代，每种区域的数量也不一定。

为了在GC Roots Tracing的时候避免扫描全堆，在每个Region中，都有一个Remembered Set来实时记录该区域内的引用类型数据与其他区域数据的引用关系（在前面的几款分代收集中，新生代、老年代中也有一个Remembered Set来实时记录与其他区域的引用关系），在标记时直接参考这些引用关系就可以知道这些对象是否应该被清除，而不用扫描全堆的数据。

G1收集器可以“建立可预测的停顿时间模型”，它维护了一个列表用于记录每个Region回收的价值大小（回收后获得的空间大小以及回收所需时间的经验值），这样可以保证G1收集器在有限的时间内可以获得最大的回收效率

1. 初始标记：标记出GC Roots直接关联的对象，这个阶段速度较快，需要停止用户线程，单线程执行   
   ② 并发标记：从GC Root开始对堆中的对象进行可达新分析，找出存活对象，这个阶段耗时较长，但可以和用户线程并发执行   
   ③ 最终标记：修正在并发标记阶段引用户程序执行而产生变动的标记记录   
   ④ 筛选回收：筛选回收阶段会对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来指定回收计划（用最少的时间来回收包含垃圾最多的区域，这就是Garbage First的由来——第一时间清理垃圾最多的区块），这里为了提高回收效率，并没有采用和用户线程并发执行的方式，而是停顿用户线程。

五、类加载

编写的Java代码需要经过编译器编译为class文件（从本地机器码转变为字节码的过程），class文件是一组以8位字节为基础的二进制流，这些二进制流分别以一定形式表示着魔数（用于标识是否是一个能被虚拟机接收的Class文件）、版本号、字段表、访问标识等内容。代码编译为class文件后，需要通过类加载器把class文件加载到虚拟机中才能运行和使用

1、类加载步骤

类从被加载到内存到使用完成被卸载出内存，需要经历加载、连接、初始化、使用、卸载这几个过程，其中连接又可以细分为验证、准备、解析。

（1）加载

在加载阶段，虚拟机主要完成三件事情：

1. 通过一个类的全限定名（比如com.danny.framework.t）来获取定义该类的二进制流；
2. 将这个字节流所代表的（静态存储结构转化为方法区的运行时存储结构）；

③ 在内存中生成一个代表这个类的java.lang.Class对象，作为程序访问方法区中这个类的外部接口。

（2）验证

验证的目的是为了确保class文件的字节流包含的内容符合虚拟机的要求，且不会危害虚拟机的安全。

文件格式验证：

元数据验证：主要对字节码描述的信息进行语义分析，保证其符合Java语言规范，比如验证这个类是否有父类（java.lang.Object除外），如果这个类不是抽象类，是否实现了父类或接口中没有实现的方法，等等。

字节码验证：字节码验证更为高级，通过数据流和控制流分析，确保程序是合法的、符合逻辑的。

符号引用验证：对类自身以外的信息进行匹配性校验，举个栗子，比如通过类的全限定名能否找到对应类、在类中能否找到字段名/方法名对应的字段/方法，如果符号引用验证失败，将抛出“java.lang.NoSuchFieldError”、“java.lang.NoSuchMethodError”等异常。

（3）准备

正式为【类变量】分配内存并设置类变量【初始值】，这些变量所使用的内存都分配在方法区。

（4）解析

解析是将常量池中【符号引用】替换为【直接引用】的过程。

（5）初始化

在准备阶段，已经为类变量赋了初始值，在初始化阶段，则根据程序员通过程序定制的主观计划去初始化类变量的和其他资源，

（1）类加载器的作用

加载class：类加载的加载阶段的第一个步骤，就是通过类加载器来完成的，类加载器的主要任务就是“通过一个类的全限定名来获取描述此类的二进制字节流”，在这里，类加载器加载的二进制流并不一定要从class文件中获取，还可以从其他格式如zip文件中读取、从网络或数据库中读取、运行时动态生成、由其他文件生成（比如jsp生成class类文件）等。

确定类的唯一性：类加载器除了有加载类的作用，还有一个举足轻重的作用，对于每一个类，都需要由加载它的加载器和这个类本身共同确立这个类在Java虚拟机中的唯一性。也就是说，两个相同的类，只有是在同一个加载器加载的情况下才“相等”，这里的“相等”是指代表类的Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果，也包括instanceof关键字对对象所属关系的判定结果。

（2）类加载器的分类

以开发人员的角度来看，类加载器分为如下几种：启动类加载器（Bootstrap ClassLoader）、扩展类加载器（Extension ClassLoader）、应用程序类加载器（Application ClassLoader）和自定义类加载器（User ClassLoader），其中启动类加载器属于JVM的一部分。

1. 启动类加载器（Bootstrap ClassLoader）：启动类加载器主要负责加载JAVA\_HOME lib目录或者被-Xbootclasspath参数指定目录中的部分类，具体加载哪些类可以通过“System.getProperty(“sun.boot.class.path”)”来查看。
2. 扩展类加载器（Extension ClassLoader）由sun.misc.Launcher.ExtClassLoader实现，负责加载JAVA\_HOMElibext目录或者被java.ext.dirs系统变量指定的路径中的所有类库，可以用通过“System.getProperty(“java.ext.dirs”)”来查看具体都加载哪些类。
3. 应用程序类加载器（Application ClassLoader）由sun.misc.Launcher.AppClassLoader实现，负责加载用户类路径（我们通常指定的classpath）上的类，如果程序中没有自定义类加载器，应用程序类加载器就是程序默认的类加载器。
4. 自定义类加载器（User ClassLoader），JVM提供的类加载器只能加载指定目录的类（jar和class），如果我们想从其他地方甚至网络上获取class文件，就需要自定义类加载器来实现，自定义类加载器主要都是通过继承ClassLoader或者它的子类来实现

双亲委派模型

双亲委派模型的工作过程是，如果一个类加载器收到了类加载的请求，它首先不会加载类，而是把这个请求委派给它上一层的父加载器，每层都如此，所以最终请求会传到启动类加载器，然后从启动类加载器开始尝试加载类，如果加载不到（要加载的类不在当前类加载器的加载范围），就让它的子类尝试加载，每层都是如此。

那么双亲委派模型有什么好处呢？最大的好处就是它让Java中的类跟类加载器一样有了“优先级”。前面说到了对于每一个类，都需要由加载它的加载器和这个类本身共同确立这个类在Java虚拟机中的唯一性，比如java.lang.Object类（存放在JAVA\_HOMElibt.jar中），如果用户自己写了一个java.lang.Object类并且由自定义类加载器加载，那么在程序中是不是就是两个类？所以双亲委派模型对保证Java稳定运行至关重要。

定义自已的类加载器分为两步：

1、继承java.lang.ClassLoader

2、重写父类的findClass方法