JVM面试

一、Part 1

1、JVM垃圾回收的时候如何确定垃圾？是否知道什么是GC Roots

1. 什么是垃圾?

   内存中已经不再使用到的空间就是垃圾

2. 要进行垃圾回收，如何判断一个对象是否可以被回收

   - **引用计数法**：java中，引用和对象是由关联的。如果要操作对象则必须用引用进行。

     因此很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收，简单说，给对象中添加一个引用计数器，每当有一个地方引用它，计数器加1，每当有一个引用失效时，计数器减1，任何时刻计数器数值为零的对象就是不可能再被使用的，那么这个对象就是可回收对象。

     但是它很难解决对象之间相互循环引用的问题

     JVM一般不采用这种实现方式。

   - **枚举根节点做可达性分析（跟搜索路径）**:

     为了解决引用计数法的循环引用问题，java使用了可达性分析的方法。

     所谓**GC ROOT**或者说**Tracing GC**的"根集合"就是一组比较活跃的引用。

     基本思路就是通过一系列"GC Roots"的对象作为起始点，从这个被称为GC Roots 的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。也即**给定一个集合的引用作为根出发**，通过引用关系遍历对象图，能被便利到的对象就被判定为存活；没有被便利到的就被判定为死亡

（1）哪些对象可以作为GC Roots对象

- 虚拟机栈（栈帧中的局部变量区，也叫局部变量表）中应用的对象。
- 方法区中的类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

2、如何盘点查看JVM系统默认值

如何查看运行中程序的JVM信息

jps查看进程信息

jinfo -flag 配置项 进程号

jinfo -flags 进程号 ==== 查看所有配置

（1）JVM参数类型

1. 标配参

   `-version` `-help`

   各个版本之间稳定，很少有很大的变化

2. x参数

   `-Xint` `-Xcomp` `-Xmixed`

   - -Xint :解释执行
   - -Xcomp：第一次使用就编译成本地代码
   - -Xmixed：混合模式

3. **xx参数**

   - Boolean类型

     公式：`-XX+或者-某个属性值` +表示开启，-表示关闭

     是否打印GC收集细节 -XX:+PrintGCDetails 开启 -XX:-PrintGCDetails 关闭

     是否使用串行垃圾回收器：-XX:-UseSerialGC

   - KV设值类型

     公式：-XX:属性key=属性值value

     case：

     -XX:MetaspaceSize=128m

     -XX:MaxTenuringThreshold=15

     -Xms----> -XX:InitialHeapSize

     -Xmx----> -XX:MaxHeapSize

(2) 查看参数

- -XX:+PrintFlagsInitial

  查看初始默认

  java -XX:+PrintFlagsInitial -version

- -XX:+PrintFlagsFinal

  查看修改后的 `:=`说明是修改过的

- -XX:+PrintCommandLineFlags

  查看使用的垃圾回收器

3、你平时工作用过的JVM常用基本配置参数有哪些

`-Xms` `-Xmx` `-Xmn`

-Xms128m -Xmx4096m -Xss1024K -XX:MetaspaceSize=512m -XX:+PrintCommandLineFlags -XX:+PrintGCDetails -XX:+UseSerialGC

1. -Xms

   初始大小内存，默认为物理内存1/64，等价于-XX:InitialHeapSize

2. -Xmx

   最大分配内存，默认物理内存1/4，等价于-XX:MaxHeapSize

3. -Xss

设置单个线程栈的大小，默认542K~1024K ，等价于-XX:ThreadStackSize

4. -Xmn

   设置年轻代的大小

5. -XX:MetaspaceSize

   设置元空间大小

   元空间的本质和永久代类似，都是对JVM规范中方法区的实现，不过元空间与永久代最大的区别在于：元空间并不在虚拟机中，而是在本地内存中。因此，默认元空间的大小仅受本地内存限制

6. -XX:+PrintGCDetails

   输出详细GC收集日志信息

   [名称：GC前内存占用->GC后内存占用(该区内存总大小)]

7. -XX:SurvivorRatio

   设置新生代中Eden和S0/S1空间的比例

   默认-XX:SurvivorRatio=8,Eden:S0:S1=8:1:1

8. -XX:NewRatio

   设置年轻代与老年代在堆结构的占比

   默认-XX:NewRatio=2 新生代在1，老年代2，年轻代占整个堆的1/3

   NewRatio值几句诗设置老年代的占比，剩下的1给新生代

9. -XX:MaxTenuringThreshold

   设置垃圾的最大年龄

   默认-XX:MaxTenuringThreshold=15

   如果设置为0，年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大的值，则年轻代对象回在Survivor区进行多次复制，这样可以增加对对象在年轻代的存活时间，增加在年轻代即被回收的概率。

10. -XX:+UseSerialGC

    串行垃圾回收器

11. -XX:+UseParallelGC

    并行垃圾回收器

4、强引用、软引用、弱引用、虚引用作用分别是什么

4.1 强引用 Reference

当内存不足，JVM开始垃圾回收，对于强引用对象，就算出现了OOM也不会堆该对象进行回收。

强引用是我们最常见的普通对象引用，只要还有强引用指向一个对象，就能表面对象还“或者”，垃圾收集器不会碰这种对象。在Java中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被强引用变量引用时，它处于可达状态，他是不可能被垃圾回收机制回收的，既是该对象以后永远都不会被用到 JVM也不会回收。因此强引用时造成java内存泄漏的主要原因之一。

对于一个普通对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式的将应用（强）引用复制为null，一般认为就是可以被垃圾收集的了

4.2 软引用 SoftReference

软引用是一种相对强引用弱化了一些作用，需要用`java.lang.ref.SOftReference`类来实现，可以让对象豁免一些垃圾收集。

当系统内存充足时他不会被回收，当内存不足时会被回收。

软引用通常在对内存敏感的程序中，比如高速缓存就有用到软引用，内存足够的时候就保留，不够就回收。

4.3 弱引用 WeakReference

弱引用需要用java.lang.ref.WeakReference类来实现，比软引用的生存期更短

只要垃圾回收机制一运行，不管JVM的内存空间是否足够，都会回收。

- **谈谈WeakHashMap**

  key是弱引用

4.4 虚引用PhantomReference

顾名思义，就是形同虚设，与其他集中不同，虚引用并不会决定对象的生命周期

如果一个对象持有虚引用，那么他就和没有任何引用一样，在任何时候都可能被垃圾回收器回收，他不能单独使用也不能通过它访问对象，虚引用和引用队列（ReferenceQueeu)联合使用。

需应用的主要作用是跟踪对象被垃圾回收的状态。仅仅是提供了一种确保ui想被finalize以后，做某些事情的机制。

PhantomReference的get方法总是返回null，因此无法访问对应的引用对象。其意义在于说明一个对象已经进入finalization阶段，可以被gc回收，用来实现比finalization机制更灵活的回收操作

换句话说，设置虚引用关联的唯一目的，就是这个对象被收集器回收的时候收到一个系统通知或者后续添加进一步的处理。

java允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

4.4.1 引用队列Reference

创建引用的时候可以指定关联的队列，当gc释放对象内存的时候，会把引用加入到引用队列，如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动，相当于通知机制

当关联的引用队列中有数据的时候，意味着引用指向的对内存中的对象被回收。通过这种方式，jvm允许我们在对象被小回收，做一些我们自己想做的事情。

4.5 适用场景

- 加入一个应用需要读取大量的本地图片，如果每次读取图片都从硬盘读取会严重影响性能，如果一次性全部加载到内存又可能造成内存溢出，这时可以用软引用解决这个问题

  设计思路：用一个HashMap来保存图片路径和相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动共回收这些缓存图片对象所占的空间，避免OOM

```
Map<String, SoftReference<Bitmap>> imageCache = new HashMap<>();
```

5、请你谈谈对OOM的认识

- `java.lang.StackOverflowError`

  栈空间溢出，递归调用卡死

- `java.lang.OutOfMemoryError:Java heap space`

  堆内存溢出， 对象过大

- `java.lang.OutOfMemoryError:GC overhead limit exceeded`

  GC回收时间过长

  过长的定义是超过98%的时间用来做GC并且回收了而不倒2%的堆内存

  连续多次GC，都回收了不到2%的极端情况下才会抛出

  如果不抛出，那就是GC清理的一点内存很快会被再次填满，迫使GC再次执行，这样就恶性循环，

  cpu使用率一直是100%，二GC却没有任何成果

```
int i = 0;
List<String> list = new ArrayList<>();
try{
    while(true){
        list.add(String.valueOf(++i).intern());
    }
}catch(Throwable e){
    System.out.println("********");
    e.printStackTrace();
    throw e;
}
```

- `java.lang.OutOfMemoryError:Direct buffer memory`

  执行内存挂了

  写NIO程序经常使用ByteBuffer来读取或写入数据，这是一种基于通道（Channel）与缓存区（Buffer)的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作，这样能在一些场景中显著提高性能，因为避免了在java堆和native堆中来回复制数据

  ByteBuffer.allocate(capability) 第一种方式是分配JVM堆内存，属于GC管辖，由于需要拷贝所以速度较慢

  ByteBuffer.alloctedDirect(capability)分配os本地内存，不属于GC管辖，不需要拷贝，速度较快

  但如果不断分配本地内存，堆内存很少使用，那么jvm就不需要执行GC，DirectByteBuffer对象们就不会被回收，这时候堆内存充足，但本地内存可能已经使用光了，再次尝试分配本地内存救护i出现oom，程序崩溃

- `java.lang.OutOfMemoryError:unable to create new native thread` **好案例**
  - 应用创建了太多线程，一个应用进程创建了多个线程，超过系统承载极限
  - 你的服务器并不允许你的应用程序创建这么多线程，linux系统默认允许单个进程可以创建的线程数是1024，超过这个数量，就会报错

  解决办法

  降低应用程序创建线程的数量，分析应用给是否针对需要这么多线程，如果不是，减到最低

  修改linux服务器配置

- `java.lang.OutOfMemoryError:Metaspace`

  元空间主要存放了虚拟机加载的类的信息、常量池、静态变量、即时编译后的代码

```
static class OOMTest{}
public static void main(String[] args){
    int i = 0;
    try{
       while(true){
           i++;
           Enhancer enhancer = new Enhancer();
           enhancer.setSuperclass(OOMTest.class);
           enhancer.setUseCache(false);
           enhancer.setCallback(new MethodInterceptor(){
               @Override
               public Object intercept(Object o,Method method,Object[] objects,
                                       MethodProxy methodProxy)throws Throwable{
                   return methodProxy.invokeSuper(o,args);
               }
           });
           enhancer.create();
       }
    } catch(Throwable e){
        System.out.println(i+"次后发生了异常");
        e.printStackTrace();
    }
}
```

6、GC垃圾回收算法和垃圾收集器的关系？分别是什么

垃圾收集器就是算法的具体实现

6.1 GC算法

- 引用计数
- 复制
- 标记清理
- 标记整理

6.2 4种主要垃圾收集器

1. Serial 串行回收

   为单线程换进该设计且只是用过一个线程进行垃圾回收，会暂停所有的用户线程，不适合服务器环境

2. Paralle 并行回收

   多个垃圾收集线程并行工作，此时用户线程是暂停的，适用于科学计算/大数据处理首台处理等弱交互场景

3. CMS 并发标记清除

   用户线程和垃圾手机线程同时执行（不一定是并行，可能交替执行），不需要停顿用户线程，互联网公司多用它，适用堆响应时间有要求的场景

4. G1

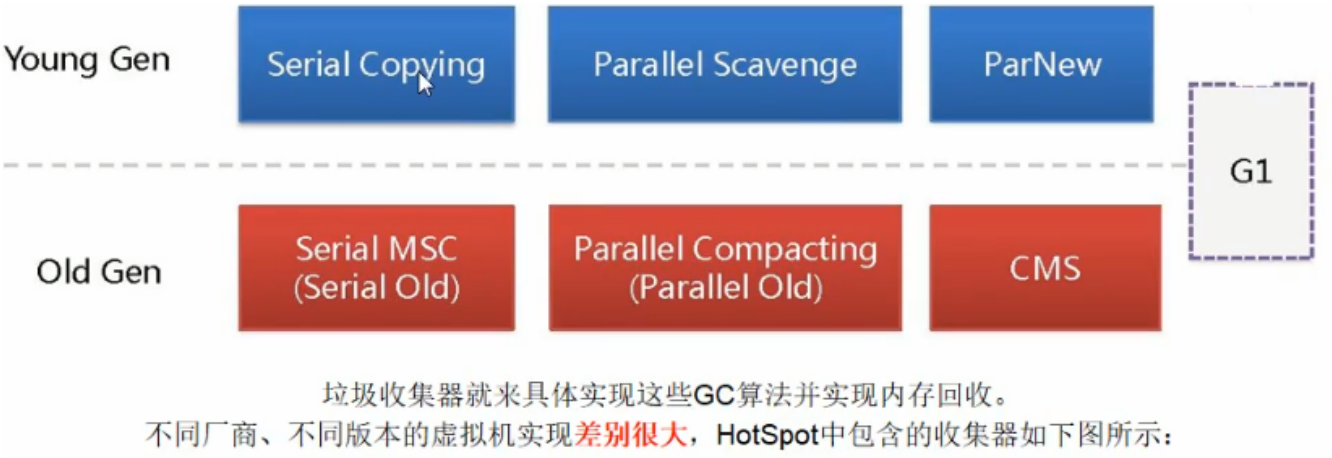
   将堆内存分割城不同的区域然后并发的对其进行垃圾回收

5. ZGC

7、怎么查看服务器默认的垃圾收集器是哪个？生产上如何配置垃圾收集器？对垃圾收集器的理解？

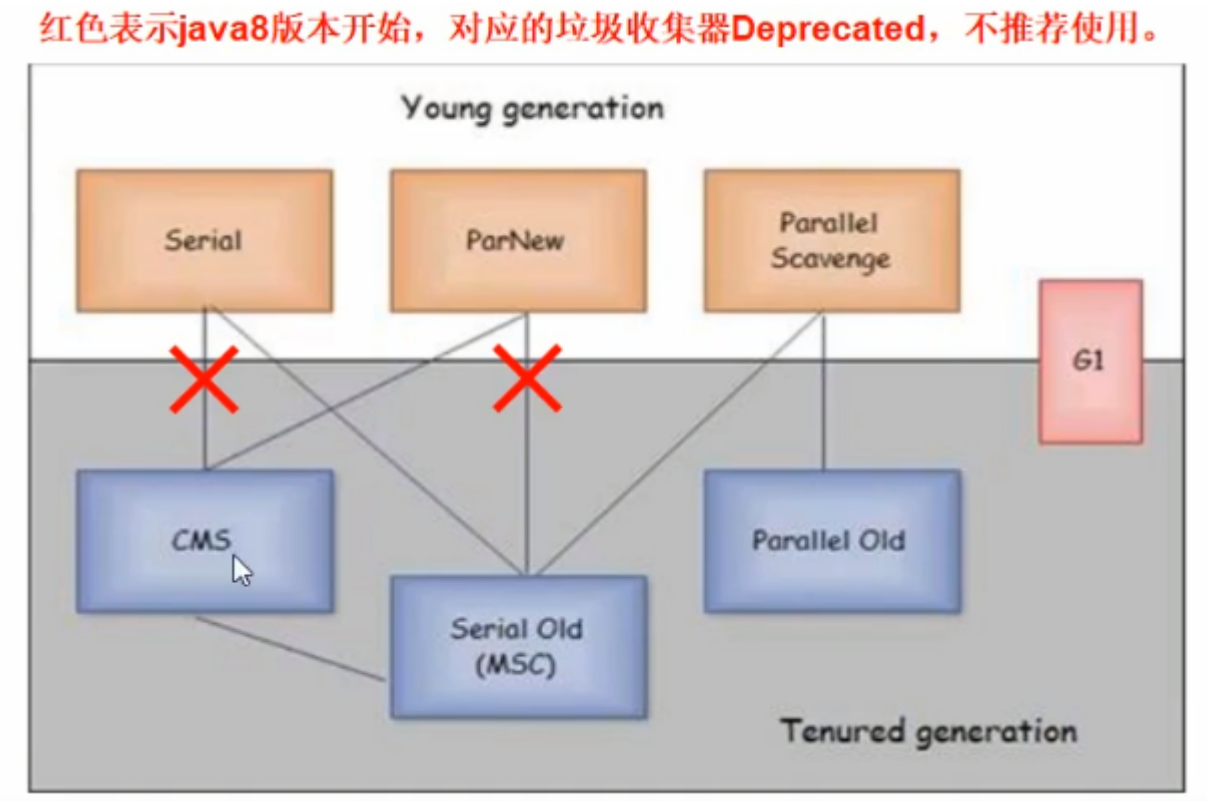
1. 查看：`java -XX:+PrintCommandLinedFlags -version`
2. 配置：

   有七种：`UseSerialGC` `UseParallelGC` `UseConcMarkSweepGC` `UseParNewGC` `UseParallelOldGC` `UseG1GC`



垃圾收集器就来具体实现这些GC算法并实现内存回收。

不同厂商、不同版本的虚拟机实现差别很大，HotSpot中包含的收集器如下图所示：



8、垃圾收集器

8.1 部分参数说明

1. DefNew：Default New Generation
2. Tenured：Old
3. ParNew：Parallel New Generation
4. PSYoungGen：Parallel Scavenge
5. ParOldGen：Parallel Old Generation

8.2 Server/Client模式分别是什么意思

1. 适用范围：只需要掌握Server模式即可，Client模式基本不会用
2. 操作系统
    - 32位Window操作系统，不论硬件如何都默认使用Client的JVM模式
    - 32位其他操作系统，2G内存同时有2个cpu以上用Server模式，低于该配置还是Client模式
    - 64位only server模式

8.3 新生代

1. **串行GC (Serial)/(Serial Copying)**

   串行收集器：Serial收集器 1:1

   一个单线程的收集器，在进行垃圾收集的时候，必须暂停其他所有的工作线程知道它收集结束。

   串行收集器是最古老，最稳定以及效率最高的收集器，只使用一个线程去回收但其在进行垃圾收集过程中可能会产生较长的停顿（Stop-The-World 状态）。虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个CPU环境来说，没有线程交互的开销可以获得最高的单线程垃圾收集效率，因此Serial垃圾收集器依然是java虚拟机运行在Client模式下默认的新生代垃圾收集器。

   对应的JVM参数是：**-XX:+UseSerialGC**

   开启后会使用：Serial(Young区)+Serial Old(Old区)的收集器组合

   表示：新生代、老年代都会使用串行回收收集器，新生代使用复制算法，老年代使用标记整理算法

   DefNew ----- Tenured

2. **并行GC (ParNew)**

   并行收集器 N:1

   使用多线程进行垃圾回收，在垃圾收集时，会Stop-the-world暂停其他所有的工作线程知道它收集结束。

   ParNew收集器其实就是Serial收集器新生代的并行多线程版本，最常见的应用场景时配合老年代的CMS GC工作，其余的行为和Serial收集器完全一样，ParNew垃圾收集器在垃圾收集过程成中童谣也要暂停所有其他的工作线程，他是很多java虚拟机运行在Server模式下**新生代**的默认垃圾收集器。

   常用对应JVM参数：**-XX:+UseParNewGC** 启用ParNew收集器，只影响新生代的收集，不影响老年代

   开启后会使用：ParNew(Young区)+Serial Old(Old 区)的收集器组合，新生代使用复制算法，老年代采用标记-整理算法

   ParNew ------ Tenured

   备注:

   -XX:ParallelGCThreads 限制线程数量，默认开启和CPU数目相同的线程数

3. **并行回收GC (Parallel)/(Parallel Scavenge)**

   并行收集器 N:N

   类似ParNew也是一个新生代垃圾收集器，使用复制算法，也是一个并行的多线程的垃圾收集器，俗称吞吐量有限收集器。串行收集器在新生代和老年代的并行化。

   他重点关注的是：

   - **可控制的吞吐量**（Thoughput=运行用户代码时间/(运行用户代码时间+垃圾收集时间)，业绩比如程序运行100分钟，垃圾收集时间1分钟，吞吐量就是99%）。高吞吐量意味着高效利用CPU时间，他多用于在后台运算而不需要太多交互的任务。
   - 自适应调节策略也是ParallelScavenge收集器与ParNew收集器的一个重要区别。（自适应调节策略：虚拟机会根据当前系统的运行情况手机性能监控信息，动态调整这些参数以提供最合适的停顿时间(-XX:MaxGCPauseMillis)或最大的吞吐量

   -XX:ParallelGCThreads=数字N 表示启动多少个GC线程

   cpu>8 n=5/8

   cpu<8 n=实际个数

   -XX:+UseParallelGC、**-XX:+UseParallelOldGC**

8.4 老年代

1. **串行GC（Serial Old）/（Serial MSC）**
2. **并行GC（Parallel Old）/（Parallel MSC）**

   Parallel Scavenge的老年代版本，使用多线程的标记-整理算法，Parallel Old收集器在JDK1.6开始提供

   1.6之前，新生代使用ParallelScavenge收集器只能搭配年老代的Serial Old收集器，只能保证新生代的吞吐量优先，无法保证整体的吞吐量。在1.6之前（ParallelScavenge+SerialOld）

   Parallel Old正式为了在老年代同样提供吞吐量游戏的垃圾收集器，如果系统对吞吐量要求比较高，JDK1.8后可以考虑新生代Parallel Scavenge和年老代Parallel Old收集器的搭配策略

   常用参数：**-XX:+UseParallelOldGC**》》》》》新生代Paralle+老年代Paralle Old

3. **并发标记清除GC（CMS）**

   CMS收集器（Concurrent Mark Sweep：并发标记清除）是一种以获取最短回收停顿时间为目标的收集器。

   适合在互联网站或者B/S系统的服务器上，这列应用尤其中使服务器的响应速度，希望系统停顿时间最短。

   CMS非常适合堆内存大、CPU核数多的服务区端应用，也是G1出现之大型应用的首选收集器。

   并发标记清除收集器：ParNew+CMS+Serial Old

   CMS，并发收集低停顿，并发指的是与用户线程一起执行

   JVM参数：**-XX:+UseConcMarkSweepGC**，开启该参数后会自动将-XX:UseParNewGC打开

   开启该参数后，使用ParNew+CMS+Serial Old的收集器组合，Serial Old将作为CMS出错的后备收集器

   4步过程：

   - 初始标记（CMS initial mark）

     只是标记一下GC Roots能够直接关联的对象，速度很快，仍然需要暂停所有的工作线程。

   - 并发标记（CMS concurrent mark）和用户线程一起

     进行GC Roots跟踪过程，和用户线程一起工作，不需要暂停工作线程。主要标记过程，标记全部对象

   - 重新标记（CMS remark）

为了修正并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程，由于并发标记时，用户线程依然运行，因此在正式清理前，再做修正

- 并发清除（CMS concurrent sweep）和用户线程一起

  清除GC Roots不可达对象，和用户线程一起工作，不需要暂停工作线程。基于标记结果，直接清理对象

  由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作，所以总体上看来CMS收集器的内存回收和用户线程是一起并发的执行。

优缺点：

- 并发收集停顿低
- 并发执行，cpu资源压力大

  由于并发进行，CMS在收集与应用线程会同时增加对堆内存的占用，也就是说，CMS必须要在老年代堆内存用尽之前完成垃圾回收，否则CMS回收失败时，将出发担保机制，串行老年代收集器将会以STW的方式进行一次GC，从而造成较大停顿时间。
- 采用的标记清除算法会导致大量的碎片

  标记清除算法无法整理空间碎片，老年代空间会随着应用时长被逐步耗尽，最后将不得不通过担保机制堆堆内存进行压缩。CMS也提供了参数-XX:CMSFullGCsBeForeCompaction(默认0，即每次都进行内存整理)来制定多少次CMS收集之后，进行一次压缩的FullGC。

8.5 如何选择垃圾选择器

- 单CPU或小内存，单机内存

  -XX:+UseSerialGC
- 多CPU，需要最大吞吐量，如后台计算型应用

  -XX:+UseParallelGC -XX:+UseParallelOldGC
- 多CPU，最求低停顿时间，需快速相应，如互联网应用

  -XX:+ParNewGC -XX:+UseConcMarkSweepGC

参数	新生代垃圾收集器	新生代算法	老年代垃圾收集器	老年代算法
UseSerialGC	SerialGC	复制	SerialOldGC	标整
UseParNewGC	ParNew	复制	SerialOldGC	标整
UseParallelGC UseParallelOldGC	Parallel[Scavenge]	复制	Parallel Old	标整
UseConcMarkSweepGC	ParNew	复制	CMS+Serial Old的收集器组合 (Serial Old 作为CMS出错的后备收集器)	标清
UseG1GC	G1整体上采用标整	局部是通过复制算法		

9、G1垃圾收集器

将堆内存分割城不同的区域然后并发的对其进行垃圾回收

9.1 其他收集器特点

- 年轻代和老年代是各自独立且了连续的内存块
- 年轻代收集使用单eden+S0+S1进行复制算法
- 老年代收集必须扫描真个老年代区域
- 都是以尽可能少而快速地执行GC为设计原则

9.2 G1是什么

G1（Garbage-First）收集器，是一款面向服务端应用的收集器

应用在多处理器和大容量内存环境中，在实现高吞吐量的同时，尽可能的满足垃圾收集暂停时间的要求

- 和CMS收集器一样，能与应用程序线程并发执行
- 整理空闲空间更快
- 需要更多的时间来预测GC停顿时间
- 不希望牺牲大量的吞吐性能
- 不需要更大的Java Heap

G1收集器的设计目标是取代CMS收集器，和CMS相比，在以下放木表现更出色：

- G1是一个由整理内存过程的垃圾收集器，不会产生很多内存碎片
- G1的Stop The World（STW）更可控，G1在停顿时间上添加了预测机制，用户可以指定期望停顿时间。

CMS垃圾收集器虽然减少了暂停应用程序的运行时间，但是他还是存在着内存碎片问题。于是为了取出内存碎片问题，同时又保留CMS垃圾收集器低暂停时间的优点，JAVA7发布了G1垃圾收集器。

主要改变的是Eden，Survivor和Tenured等内存区域不再是连续的了，而是变成了一个个大小一样的region（区域化），每个region从1M到32M不等。一个region有可能属于Eden，Survivor或Tenured内存区域。

9.2.1 特点

1. G1能充分利用多CPU、多核环境优势，尽量缩短STW。
2. G1整体上采用标记-整理算法，局部是通过复制算法，不会产生内存碎片。
3. 宏观上G1之中不再区分年轻代和老年代。把内存划分成多个独立的子区域（Region），可以近似理解为一个棋盘
4. G1收集器里面讲整个的内存去都混合在一起了，但其本身依然在小范围内要进行年轻代和老年代的区分，保留了新生代和老年代，但它们不再是物理隔离，而是一部分Region的集合且不需要Region是连续的，也就是说依然会采用不同GC方式来处理不同的区域。
5. G1虽然也是分代收集器，但整个内存分区不存在物理上的年轻代与老年代的区别，也不需要完全独立的survivor（to space）堆做复制准备。G1只有逻辑上的分代概念，或者说每个分区都可能随G1的运行在不同代之间前后切换

9.3 底层原理

- Region区域化垃圾收集器

  最大好处是化整为零，避免全内存扫描，只需要按照区域来进行扫描即可

  区域化内存划片Region，整体编为了 一系列不连续的内存区域，避免了全内存区的GC操作。

  核心思想是讲整个堆内存区域分成大小相同的子区域，在JVM启动时会自动共设置这些子区域的大小

  在堆的使用上，G1并不要求对象的存储一定是物理上连续的，只要逻辑上连续即可，每个分区也不会固定地为某个代服务，可以按需在年轻代和老年代之间切换。启动时可以通过参数-XX:G1HeapRegionSize可指定分区大小（1~32M,且必须是2的幂），默认将整堆划分为2048个分区。

  大小范围在1-32M，最多能设置2048个区域，也即能够支持的最大内存为64G

  G1算法将堆划分为诺干个区域，他仍然属于分代收集器

  - 这些Region的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或Survivor空间，这些Region的一部分包含老年代，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩，这样也就不会有CMS内存碎片问题的存在了
  - 在G1中，还有一种特殊区域，**Humongous**区域，如果一个对象张勇的空间超过了分区容量50%以上，G1收集器就认为i这是一个巨型对象。这些巨型对象默认直接会被分配在年老代，但是如果他是一个短期存在的巨型对象，就会对垃圾收集器造成负面影响。为了解决这个问题，G1划分了一个Humongous区，他用来专门存放巨型对象。如果一个H区装不下，那么G1就会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC
- 回收步骤

  针对Eden区进行收集，Eden区耗尽后会被触发，主要小区域收集+形成连续的内存块，避免内存碎片

  - Eden区的数据移动到Survivor区，假如出现Survivor区空间不够，Eden区数据就会晋升到Old区
  - Survivor区的数据移动到新的Survivor区，部分数据晋升到Old区
  - 最后Eden区收拾干净了，GC结束，用户的应用程序继续执行。
- 4步过程

  1. 初始标记：只标记GC Roots能直接关联到的对象
  2. 并发标记：进行GC Roots Tracing的过程

3. 最终标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象
4. 筛选回收：根据时间来进行价值最大化的回收

9.4 case

9.5 常用配置参数（不是重点）

- **-XX:+UseG1Gc**
- **-XX:G1HeapRegionSize=n**

  设置的G1区域的大小，值是2的幂，范围是1-32MB，目标是根据最小的java堆大小划分出约2048个区域
- **-XX:MaxGCPauseMillis=n**

  最大GC停顿时间，这是个软目标，JVM将尽可能（但不保证）停顿小于这个时间
- **-XX:InitiatingHeapOccupancyRercent=n**

  堆占用了多少的时候就触发GC，默认45
- **-XX:ConcGcThreads=n**

  并发GC使用的线程数
- **-XX:G1ReservePercent=n**

  设置作为空闲空间的预留内存百分比，以降低目标空间溢出的风险，默认10%

9.6 和CMS相比优势

1. G1不会产生内存碎片
2. 可以精确控制停顿。该收集器十八真个堆划分成多个固定大小的区域，每根据允许停顿的时间去收集垃圾最多的区域

10、生产环境服务器变慢，诊断思路和性能评估谈谈？

1. 整机：top 系统性能

   uptime 精简版

   load average：系统负载均衡 1min 5min 15min 系统的平均负载值 相加/3>60%压力够
2. CPU：vmstat
   - 查看CPU

     vmstat -n 2 3 第一个参数是采样的时间间隔数单位s，第二个参数是采样的次数
     - procs

       r：运行和等待CPU时间片的进程数，原则上1核CPu的运行队列不要超过2，真个系统的运行队列不能超过总核数的2倍，否则表示系统压力过大

       b：等待资源的进程数，比如正在等待磁盘I/O，网络I/O等
     - cpu

       us：用户进程消耗cpu时间百分比，us高，用户进程消耗cpu时间多，如果长期大于50%，优化程序

       sy：内核进程消耗的cpu时间百分比

us+sy：参考值为80%，如果大于80，说明可能存在cpu不足

id：处于空闲的cpu百分比

wa：系统等待IO的cpu时间百分比

st：来自于一个虚拟机偷取的cpu时间的百分比

- 查看额外
  - 查看所有cpu核信息 mpstat -P ALL 2
  - 每个进程使用cpu的用量分解信息 pidstat -u 1 -p 进程编号

3. 内存：free

   查看内存 free -m free -g

   pidstat -p 进程编号 -r 采样间隔秒数

4. 硬盘：df

   查看磁盘剩余空间 df -h

5. 磁盘IO：iostat
   - 磁盘I/O性能评估

     iostat -hdk 2 3

     - rkB/s每秒读取数据kb；
     - wkB/s每秒读写数据量kb
     - svctm I/O请求的平均服务时间，单位毫秒；
     - await I/O请求的平均等待时间，单位毫秒；值越小，性能越好；
     - util 一秒中又百分几的时间用于I/O操作，接近100%时，表示磁盘带宽跑满，需要优化程序或加磁盘
     - rkB/s，wkB/s根据系统该应用不同回有不同的值，担忧规律遵循：长期、超大数据读写，肯定不正常，需要优化程序读取。
     - svctm的值与await的值很接近，表示几乎没有I/O等待，磁盘性能好，如果await的值远高于svctm的值，则表示I/O队列等待太长，需要优化程序或更换更快磁盘
   - pidstat -d 采样间隔秒数 -p 进程号

6. 网络IO：ifstat

   ifstat l

11、加入生产环境CPU占用过高，谈谈分析思路和定位?

结合Linux和JDK命令一块分析

11.1 案例步骤

1. 先用top命令找出cpu占比最高的
2. ps -ef或者jps进一步定位，得知是一个怎样的后台程序惹事
   - jps -l
   - ps -ef|grep java|grep -v grep
3. 定位到具体线程或者代码
   - ps -mp 进程编号 -o Thread,tid,time

     定位到具体线程

-m ： 显示所有的线程

-p pid 进程使用cpu的时间

-o：该参数后是用户自定义格式

4. 将需要的线程ID转换为16禁止格式（英文小写格式）
    - printf "%x\n" 线程ID
5. jstack 进程Id|grep tid(16进制线程id小写英文) -A60

   查看运行轨迹，堆栈异常信息

12、对于JDK自带的JVM监控和性能分析工具你用过那些？一般怎么用？