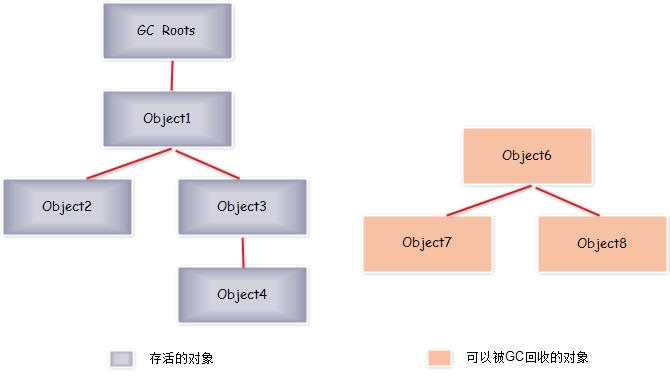
**JVM 垃圾回收的时候如何确定垃圾？知道什么是 GC Roots ?**

* 什么是垃圾
  + 简单来说就是内存中已经不在被使用到的空间就是垃圾
* 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
  + 枚举根节点做可达性分析

为了解决引用计数法的循环引用问题，Java 使用了可达性算法。

[](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/1350633405_4538.jpg)

跟踪收集器采用的为集中式的管理方式，全局记录对象之间的引用状态，执行时从一些列GC Roots的对象做为起点，从这些节点向下开始进行搜索所有的引用链，当一个对象到GC Roots 没有任何引用链时，则证明此对象是不可用的。

图中，对象Object6、Object7、Object8虽然互相引用，但他们的GC Roots是不可到达的，所以它们将会被判定为是可回收的对象。

哪些对象可以作为 GC Roots 的对象：

* 虚拟机栈（栈帧中的局部变量区，也叫局部变量表）中引用的对象
* 方法区中的类静态属性引用的对象
* 方法去常量引用的对象
* 本地方法栈中 JNI (Native方法)引用的对象

**你说你做过 JVM 调优和参数配置，请问如果盘点查看 JVM 系统默认值？**

**JVM 的参数类型**

* 标配参数
  + -version
  + -help
* X 参数（了解）
  + -Xint：解释执行
  + -Xcomp：第一次使用就编译成本地代码
  + -Xmixed：混合模式
* XX 参数
  + Boolean 类型：-XX：+ 或者 - 某个属性值（+ 表示开启，- 表示关闭）
    - -XX:+PrintGCDetails：打印 GC 收集细节
    - -XX:-PrintGCDetails：不打印 GC 收集细节
    - -XX:+UseSerialGC：使用了串行收集器
    - -XX:-UseSerialGC：不使用了串行收集器
  + KV 设置类型：-XX:key=value
    - -XX:MetaspaceSize=128m
    - -XX:MaxTenuringThreshold=15
  + jinfo 举例，如何查看当前运行程序的配置

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | public class HelloGC {  public static void main(String[] args) {  System.out.println("hello GC...");  try {  Thread.sleep(Integer.MAX\_VALUE);  } catch (InterruptedException e) {  e.printStackTrace();  }  } } |

* + 我们可以使用 jps -l 命令，查出进程 id

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | 1923 org.jetbrains.jps.cmdline.Launcher 1988 sun.tools.jps.Jps 1173 org.jetbrains.kotlin.daemon.KotlinCompileDaemon 32077 com.intellij.idea.Main 1933 com.cuzz.jvm.HelloGC 32382 org.jetbrains.idea.maven.server.RemoteMavenServer |

* + 在使用 jinfo -flag PrintGCDetails 1933 命令查看

|  |  |
| --- | --- |
| 1 | -XX:-PrintGCDetails |

* + 可以看出默认是不打印 GC 收集细节  
    也可是使用jinfo -flags 1933 查看所以的参数
  + 两个经典参数：-Xms 和 - Xmx（如 -Xms1024m）
    - -Xms 等价于 -XX:InitialHeapSize
    - -Xmx 等价于 -XX:MaxHeapSize

**盘点家底查看 JVM 默认值**

* 查看初始默认值：-XX:+PrintFlagsInitial

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | cuzz@cuzz-pc:~/Project/demo$ java -XX:+PrintFlagsInitial [Global flags]  intx ActiveProcessorCount = -1 {product}  uintx AdaptiveSizeDecrementScaleFactor = 4 {product}  uintx AdaptiveSizeMajorGCDecayTimeScale = 10 {product}  uintx AdaptiveSizePausePolicy = 0 {product}  uintx AdaptiveSizePolicyCollectionCostMargin = 50 {product}  uintx AdaptiveSizePolicyInitializingSteps = 20 {product}  uintx AdaptiveSizePolicyOutputInterval = 0 {product}  uintx AdaptiveSizePolicyWeight = 10 {product}  ... |

* 查看修改更新：-XX:+PrintFlagsFinal

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | bool UsePSAdaptiveSurvivorSizePolicy = true {product} bool UseParNewGC = false {product} bool UseParallelGC := true {product} bool UseParallelOldGC = true {product} bool UsePerfData = true {product} bool UsePopCountInstruction = true {product} bool UseRDPCForConstantTableBase = false {C2 product} |

* = 与 := 的区别是，一个是默认，一个是人物改变或者 jvm 加载时改变的参数
* 打印命令行参数(可以看默认垃圾回收器)：-XX:+PrintCommandLineFlags

|  |  |
| --- | --- |
| 1 2 | cuzz@cuzz-pc:~/Project/demo$ java -XX:+PrintCommandLineFlags -XX:InitialHeapSize=128789376 -XX:MaxHeapSize=2060630016 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC |

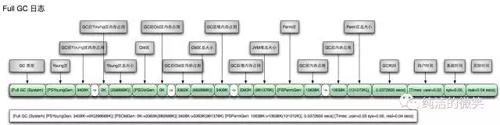
**你平时工作用过的 JVM 常用的基本配置参数有哪些？**

* -Xms
  + 初始大小内存，默认为物理内存 1/64
  + 等价于 -XX:InitialHeapSize
* -Xmx
  + 最大分配内存，默认为物理内存的 1/4
  + 等价于 -XX:MaxHeapSize
* -Xss
  + 设置单个线程栈的大小，一般默认为 512-1024k
  + 等价于 -XX:ThreadStackSize
* -Xmn
  + 设置年轻代的大小
  + **整个JVM内存大小=年轻代大小 + 年老代大小 + 持久代大小**，持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。
* -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 规范中的方法区的实现，不过元空间于永久代之间最大区别在于，**元空间并不在虚拟中，而是使用本地内存**，因此默认情况下，元空间的大小仅受本地内存限制）
  + 元空间默认比较小，我们可以调大一点
* -XX:+PrintGCDetails
  + 输出详细 GC 收集日志信息
    - 设置 JVM 参数为： -Xms10m -Xmx10m -XX:+PrintGCDetails
    - 代码

|  |  |
| --- | --- |
| 1 2 3 4 5 | public class HelloGC {  public static void main(String[] args) {  byte[] bytes = new byte[20 \* 1024 \* 1024];  } } |

* + - 打印结果

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | [GC (Allocation Failure) [PSYoungGen: 1231K->448K(2560K)] 1231K->456K(9728K), 0.0015616 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC (Allocation Failure) [PSYoungGen: 448K->384K(2560K)] 456K->392K(9728K), 0.0016999 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (Allocation Failure) [PSYoungGen: 384K->0K(2560K)] [ParOldGen: 8K->358K(7168K)] 392K->358K(9728K), [Metaspace: 3028K->3028K(1056768K)], 0.0066696 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]  [GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 358K->358K(9728K), 0.0005321 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 358K->340K(7168K)] 358K->340K(9728K), [Metaspace: 3028K->3028K(1056768K)], 0.0051543 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]  Heap  PSYoungGen total 2560K, used 81K [0x00000000ffd00000, 0x0000000100000000, 0x0000000100000000)  eden space 2048K, 3% used [0x00000000ffd00000,0x00000000ffd14668,0x00000000fff00000)  from space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)  to space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)  ParOldGen total 7168K, used 340K [0x00000000ff600000, 0x00000000ffd00000, 0x00000000ffd00000)  object space 7168K, 4% used [0x00000000ff600000,0x00000000ff655188,0x00000000ffd00000)  Metaspace used 3060K, capacity 4496K, committed 4864K, reserved 1056768K  class space used 336K, capacity 388K, committed 512K, reserved 1048576K Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  at com.cuzz.jvm.HelloGC.main(HelloGC.java:12) |

* + GC  
    [](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/a9a0eb99b30cf3fd8973f464eb4678bf50f760cc.jpg)
  + FullGC  
    [](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/e04f3f3b68cff61027e5ba8eba9613bb7c69a08a.jpg)
* -XX:SurvivorRatio
  + 设置新生代中 eden 和 S0/S1 空间比例
  + 默认 -XX:SurvivorRatio=8，Eden : S0 : S1 = 8 : 1 : 1
* -XX:NewRatio
  + 配置年轻代和老年代在堆结构的占比
  + 默认 -XX:NewRatio=2 新生代占1，老年代占2，年轻代占整个堆的 1/3
* -XX:MaxTenuringThreshold
  + 设置垃圾最大年龄

**强引用、软引用、弱引用和虚引用分别是什么？**

在Java语言中，除了基本数据类型外，其他的都是指向各类对象的对象引用；Java中根据其生命周期的长短，将引用分为4类。

* 强引用
  + 特点：我们平常典型编码Object obj = new Object()中的obj就是强引用。通过关键字new创建的对象所关联的引用就是强引用。 当JVM内存空间不足，JVM宁愿抛出OutOfMemoryError运行时错误（OOM），使程序异常终止，也不会靠随意回收具有强引用的“存活”对象来解决内存不足的问题。对于一个普通的对象，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地将相应（强）引用赋值为 null，就是可以被垃圾收集的了，具体回收时机还是要看垃圾收集策略。
* 软引用
  + 特点：软引用通过SoftReference类实现。 软引用的生命周期比强引用短一些。只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象：即JVM 会确保在抛出 OutOfMemoryError 之前，清理软引用指向的对象。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。后续，我们可以调用ReferenceQueue的poll()方法来检查是否有它所关心的对象被回收。如果队列为空，将返回一个null,否则该方法返回队列中前面的一个Reference对象。
  + 应用场景：软引用通常用来实现内存敏感的缓存。如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。
  + 代码验证  
    我设置 JVM 参数为 -Xms10m -Xmx10m -XX:+PrintGCDetails

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | public class SoftReferenceDemo {  public static void main(String[] args) {  Object obj = new Object();  SoftReference<Object> softReference = new SoftReference<>(obj);  obj = null;   try {  // 分配 20 M  byte[] bytes = new byte[20 \* 1024 \* 1024];  } catch (Exception e) {  e.printStackTrace();  } finally {  System.out.println("软引用：" + softReference.get());  }   } } |

* + 输出

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | [GC (Allocation Failure) [PSYoungGen: 1234K->448K(2560K)] 1234K->456K(9728K), 0.0016748 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [GC (Allocation Failure) [PSYoungGen: 448K->384K(2560K)] 456K->392K(9728K), 0.0018398 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]  [Full GC (Allocation Failure) [PSYoungGen: 384K->0K(2560K)] [ParOldGen: 8K->358K(7168K)] 392K->358K(9728K), [Metaspace: 3030K->3030K(1056768K)], 0.0057246 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]  [GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 358K->358K(9728K), 0.0006038 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 358K->340K(7168K)] 358K->340K(9728K), [Metaspace: 3030K->3030K(1056768K)], 0.0115080 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]  软引用：null Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  at com.cuzz.jvm.SoftReferenceDemo.main(SoftReferenceDemo.java:21) Heap  PSYoungGen total 2560K, used 98K [0x00000000ffd00000, 0x0000000100000000, 0x0000000100000000)  eden space 2048K, 4% used [0x00000000ffd00000,0x00000000ffd18978,0x00000000fff00000)  from space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)  to space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)  ParOldGen total 7168K, used 340K [0x00000000ff600000, 0x00000000ffd00000, 0x00000000ffd00000)  object space 7168K, 4% used [0x00000000ff600000,0x00000000ff6552f8,0x00000000ffd00000)  Metaspace used 3067K, capacity 4496K, committed 4864K, reserved 1056768K  class space used 336K, capacity 388K, committed 512K, reserved 1048576K |

* + 发现当内存不够的时候就会被回收。
* 弱引用
  + 特点：弱引用通过WeakReference类实现。 弱引用的生命周期比软引用短。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。由于垃圾回收器是一个优先级很低的线程，因此不一定会很快回收弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。
  + 应用场景：弱应用同样可用于内存敏感的缓存。
  + 代码验证

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | public class WeakReferenceDemo {  public static void main(String[] args) {  Object obj = new Object();  WeakReference<Object> weakReference = new WeakReference<>(obj);  System.out.println(obj);  System.out.println(weakReference.get());   obj = null;  System.gc();  System.out.println("GC之后....");    System.out.println(obj);  System.out.println(weakReference.get());  } } |

* + 输出

|  |  |
| --- | --- |
| 1 2 3 4 5 | java.lang.Object@1540e19d java.lang.Object@1540e19d GC之后.... null null |

* + 值得注意的是String name = "cuzz" 这种会放入永久代，以及 Integer age = 1 在 int 中 -128 到 127 会被缓存，所以是强引用，然后 GC 也不会被回收。
  + 引用队列

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | public class ReferenceQueueDemo {  public static void main(String[] args) throws InterruptedException {  Object obj = new Object();  ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();  WeakReference<Object> weakReference = new WeakReference<>(obj, referenceQueue);  System.out.println(obj);  System.out.println(weakReference.get());  System.out.println(weakReference);   obj = null;  System.gc();  Thread.sleep(500);   System.out.println("GC之后....");  System.out.println(obj);  System.out.println(weakReference.get());  System.out.println(weakReference);  } } |

* + 输出

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | java.lang.Object@1540e19d java.lang.Object@1540e19d java.lang.ref.WeakReference@677327b6 GC之后.... null null java.lang.ref.WeakReference@677327b6 |

* + 会把该对象的包装类即weakReference放入到ReferenceQueue里面，我们可以从queue中获取到相应的对象信息，同时进行额外的处理。比如反向操作，数据清理等。
* 虚引用
  + 特点：虚引用也叫幻象引用，通过PhantomReference类来实现。无法通过虚引用访问对象的任何属性或函数。幻象引用仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制。如果一个对象仅持有虚引用，那么它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。虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。  
    ReferenceQueue queue = new ReferenceQueue ();  
    PhantomReference pr = new PhantomReference (object, queue);  
    程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。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一些程序行动。
  + 应用场景：可用来跟踪对象被垃圾回收器回收的活动，当一个虚引用关联的对象被垃圾收集器回收之前会收到一条系统通知。

**请谈谈你对 OOM 的认识？**

* java.lang.StackOverflowError
  + 在一个函数中调用自己就会产生这个错误
* java.lang.OutOfMemoryError : Java heap space
  + new 一个很大对象
* java.lang.OutOfMemoryError : GC overhead limit exceeded
  + 执行垃圾收集的时间比例太大， 有效的运算量太小，默认情况下,，如果GC花费的时间超过 **98%**， 并且GC回收的内存少于 **2%**， JVM就会抛出这个错误。
* java.lang.OutOfMemoryError : Direct buffer memory  
  配置参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=5m

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | public class DirectBufferDemo {  public static void main(String[] args) {  System.out.println("maxDirectMemory : " + sun.misc.VM.maxDirectMemory() / (1024 \* 1024) + "MB");  ByteBuffer byteBuffer = ByteBuffer.allocateDirect(6 \* 1024 \* 1024);  } } |

* 输出

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | maxDirectMemory : 5MB [GC (System.gc()) [PSYoungGen: 1315K->464K(2560K)] 1315K->472K(9728K), 0.0008907 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [Full GC (System.gc()) [PSYoungGen: 464K->0K(2560K)] [ParOldGen: 8K->359K(7168K)] 472K->359K(9728K), [Metaspace: 3037K->3037K(1056768K)], 0.0060466 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]  Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory  at java.nio.Bits.reserveMemory(Bits.java:694)  at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)  at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:311)  at com.cuzz.jvm.DirectBufferDemo.main(DirectBufferDemo.java:17) Heap  PSYoungGen total 2560K, used 56K [0x00000000ffd00000, 0x0000000100000000, 0x0000000100000000)  eden space 2048K, 2% used [0x00000000ffd00000,0x00000000ffd0e170,0x00000000fff00000)  from space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)  to space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)  ParOldGen total 7168K, used 359K [0x00000000ff600000, 0x00000000ffd00000, 0x00000000ffd00000)  object space 7168K, 5% used [0x00000000ff600000,0x00000000ff659e28,0x00000000ffd00000)  Metaspace used 3068K, capacity 4496K, committed 4864K, reserved 1056768K  class space used 336K, capacity 388K, committed 512K, reserved 1048576K |

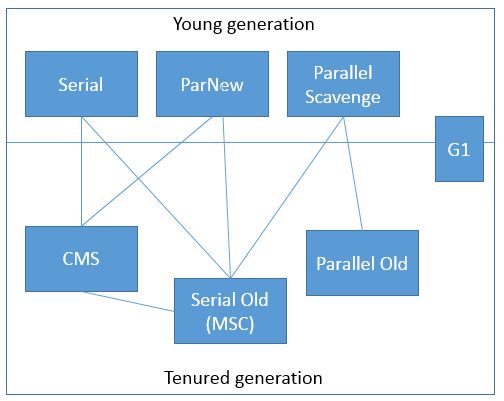
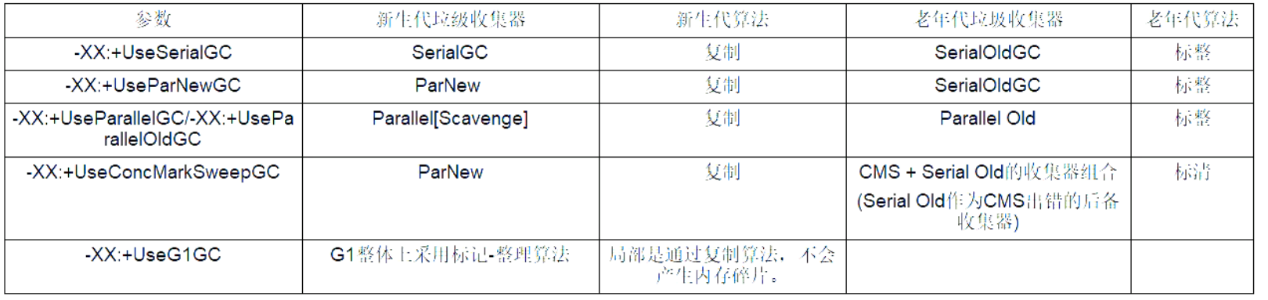
* java.lang.OutOfMemoryError : unable to create new native thread
  + 创建线程数太多了
* java.lang.OutOfMemoryError : Metaspace
  + Java 8 之后的版本使用元空间（Metaspace）代替了永久代，元空间是方法区在 HotSpot 中的实现，它与持久代最大的区别是：元空间并不在虚拟机中的内存中而是使用本地内存。
  + 元空间存放的信息：
    - 虚拟机加载的类信息
    - 常量池
    - 静态变量
    - 即时编译后的代码

具体的实现可以看看这个帖子：[几种手动OOM的方式](http://www.dataguru.cn/thread-351920-1-1.html)

**GC 垃圾回收算法和垃圾收集器的关系？谈谈你的理解？**

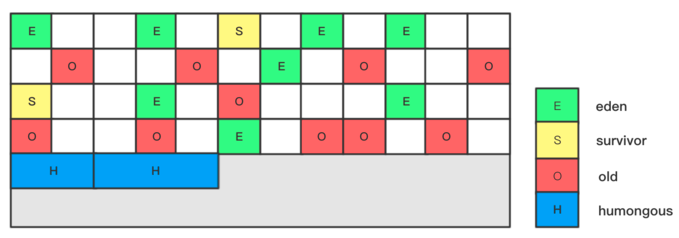
* 四种 GC 垃圾回收算法
  + 引用计数
  + 复制回收
  + 标记清除
  + 标记整理
* GC 算法是内存回收的方法论，垃圾收集其就是算法的落实的实现。
* 目前为止还没有完美的收集器的出现，更加没有万能的收集器，只是针对具体应用最适合的收集器，进行分代收集。
* 串行垃圾回收器（Serial）
  + 它为单线程环境设计且只使用一个线程进行垃圾回收，会暂停所有的用户线程，所以不适合服务环境。
* 并行垃圾回收器（Parallel）
  + 多个垃圾收集线程并行工作，此时用户线程是暂停的，用于科学计算、大数据处理等弱交互场景。
* 并发垃圾回收器（CMS）
  + 用户线程和垃圾收集线程同时执行（不一定是并行，可能是交替执行），不需要停顿用户线程，互联网公司多用它，适用对相应时间有要求的场景。
* G1 垃圾回收器
  + G1 垃圾回收器将堆内存分割成不同的区域然后并发的对其进行垃圾回收。

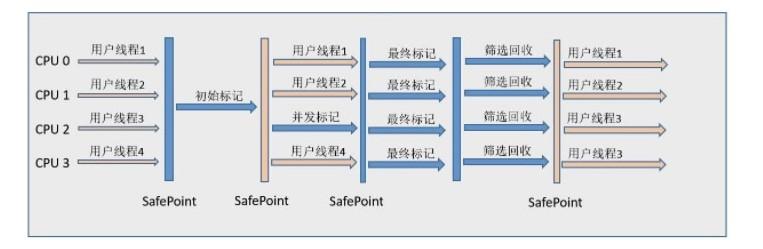
**怎么查看服务器默认垃圾收集器是哪个？生产是如何配置垃圾收集器？谈谈你对垃圾收集器的理解？**

* 怎么查看服务器默认垃圾收集器是哪个？
  + Java -XX:+PrintCommandLineFlags
* Java 的 GC 回收的类型主要有：
  + UseSerialGC，UseParallelGC，UseConcMarkSweepGC，UseParNewGC，UseParallelOldGC，UseG1GC
  + Java 8 以后基本不使用 Serial Old
* 垃圾收集器  
  [](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/timg.jpg)
* 参数说明
  + DefNew : Default New Generation
  + Tenured : Old
  + ParNew : Parallel New Generation
  + PSYoungGen : Parallel Scavenge
  + ParOldGen : Parallel Old Generation
* Server/Client 模式分别是什么意思
  + 最主要的差别在于：-Server模式启动时，速度较慢，但是一旦运行起来后，性能将会有很大的提升。
  + 当虚拟机运行在-client模式的时候，使用的是一个代号为C1的轻量级编译器, 而-server模式启动的虚拟机采用相对重量级，代号为C2的编译器，C2比C1编译器编译的相对彻底，服务起来之后,性能更高。
  + 所以通常用于做服务器的时候我们用服务端模式，如果你的电脑只是运行一下java程序，就客户端模式就可以了。当然这些都是我们做程序优化程序才需要这些东西的，普通人并不关注这些专业的东西了。其实服务器模式即使编译更彻底，然后垃圾回收优化更好，这当然吃的内存要多点相对于客户端模式。
* 新生代
  + 串行 GC (Serial/ Serital Copying)
  + 并行 GC (ParNew)
  + 并行回收 GC (Parallel/ Parallel Scanvenge)
* 老年代
  + 串行 GC (Serial Old/ Serial MSC)
  + 并行 GC (Parallel Old/ Parallel MSC)
  + 并发标记清除 GC (CMS)
    - 是一种以获取最短回收停顿时间为目标的收集器，适合应用在互联网站或者 B/S 系统的服务器上，这个类应用尤其重视服务器的响应速度，希望系统停顿时间最短。
    - CMS 非常适合堆内存大、CPU 核数多的服务器端应用，也是 G1 出现之前大型应用首选收集器。
    - 并发停顿比较少，并发指的是与用户线程一起执行。
    - 过程
      * 初始标记（initail mark）：只是标记一下 GC Roots 能直接关联的对象，速度很快，需要暂停所有的工作线程
      * 并发标记（concurrent mark 和用户线程一起）：进行 GC Roots 的跟踪过程，和用户线程一起工作，不需要暂停工作线程。
      * 重新标记（remark）：为了修正在并发标记期间，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。
      * 并发清除（concurrent sweep 和用户线程一起）：清除 GC 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，直接清除。由于耗时最长的并发标记和并发清除过程中，垃圾收集线程和用户线程可以一起并发工作，所以总体来看 CMS 收集器的内存回收和用户线程是一起并发地执行。
    - 优缺点
      * 优点：并发收集停顿低
      * 缺点：并发执行对 CPU 资源压力大，采用的标记清除算法会导致大量碎片
    - 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 必须要在老年代堆用尽之前完成垃圾回收，否者 CMS 回收失败，将触发担保机制，串行老年代收集器将会以 STW 的方式进行一次 GC，从而造成较大的停顿时间。
    - 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渐耗尽，最后将不得不通过担保机制对堆内存进行压缩。CMS 也提供了参数 -XX:CMSFullGCsBeForeCompaction (默认0，即每次都进行内存整理) 来指定多少次 CMS 收集之后，进行一次压
* 垃圾收集器配置代码总结
  + 配置新生代收集器，老年代收集器会自动配置上。  
    [](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/1558237229584.png)
* 如何选择垃圾收集器
  + 单 CPU 或者小内存，单机程序：-XX:UseSerialGC
  + 多 CPU 需要最大吞吐量，如后台计算型应用：-XX:UseParallelGC 或者 -XX:UseParallelOldGC
  + 多 CPU 追求低停顿时间，需要快速响应，如互联网应用：-XX:+UseConcMarkSweepGC

**G1 垃圾收集器你了解吗？**

* 以前收集器的特点
  + 年轻代和老年代是各自独立且连续的内存块
  + 年轻代收集器使用 eden + S0 + S1 进行复制算法
  + 老年代收集必须扫描整个老年代区域
  + 都是以尽可能的少而快速地执行 GC 为设计原则
* G1 是什么
  + G1 是一种面向服务端的垃圾收集器，应用在多核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器的暂停时间要求。
  + 像 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 收集器的设计目的是取代 CMS 收集器，同时与 CMS 相比，G1 垃圾收集器是一个有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 更可控，G1 在停顿上添加了预测机制，用户可以指定期望的停顿时间。
  + G1 是在 2012 年才在 jdk.1.7u4 中可以呀用，在 jdk9 中将 G1 变成默认垃圾收集器来代替 CMS。它是以款面向服务应用的收集器。
  + 主要改变是 Eden、Survivor 和 Tenured 等内存区域不再是连续的，而是变成了一个个大小一样的 region，每个 region 从 1M 到 32M 不等，一个 region 有可能属于 Eden、Survivor 或者 Tenured 内存区域。
* 特点
  + G1 能充分利用多 CPU、多核环境硬件优势，尽量缩短 STW。
  + G1 整体采用标记-整理算法，局部是通过是通过复制算法，**不会产生内存碎片**。
  + 宏观上看 G1 之中不在区分年轻代和老年代，被内存划分为多个独立的子区域。
  + G1 收集器里面讲整个的内存区域混合在一起，**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，但她们不在是物理隔离，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的 GC 方式来处理不同的区域。
  + 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和老年代的区别，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。
* 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。
  + **Region**

[](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/5611237-f643066bd97c7703.png)  
G1的内存结构和传统的内存空间划分有比较的不同。G1将内存划分成了多个大小相等的Region（默认是512K），Region逻辑上连续，物理内存地址不连续。同时每个Region被标记成E、S、O、H，分别表示Eden、Survivor、Old、Humongous。其中E、S属于年轻代，O与H属于老年代。  
H表示Humongous。从字面上就可以理解表示大的对象（下面简称H对象）。**当分配的对象大于等于Region大小的一半**的时候就会被认为是巨型对象。H对象默认分配在老年代，可以防止GC的时候大对象的内存拷贝。通过如果发现堆内存容不下H对象的时候，会触发一次GC操作。

* + 回收步骤
    - 参看：[G1从入门到放弃](https://www.jianshu.com/p/548c67aa1bc0" \t "_blank)
  + 四步过程  
    [](http://blog.cuzz.site/2019/05/10/JVM%E9%9D%A2%E8%AF%95/u=1236259389,1737476709&fm=26&gp=0.jpg)

**生产环境服务器变慢，诊断思路和性能评估谈谈？**

* 整机：top
* CPU：vmstat
* 内存：free
* 硬盘：df
* 磁盘IO：iostat
* 网络IO：ifstat

**假如生产环境出现 CPU 过高，请谈谈你的分析思路和定位？**

* 先用 top 命令找出 CPU 占比最高的
* ps -ef 或者 jps 进一步定位，得知是一个怎么样的一个后台程序
* 定位到具体的线程或代码
  + ps -mp 11111 -o THREAD,tid,time
  + -m 显示所有的线程
  + -p 进程使用cpu的时间
  + -o 该参数后是用户自定义格式
* 将需要的线程 ID 转化为 16 进制格式
* jstat <进程ID> | grep <线程ID(16进制)> -A60

**对于 JDK 自带的 JVM 监控和性能分析工具用过哪些？一般机是怎么用到的？**