第三版：JVM 50 道

对象在哪块内存分配?

数组和对象在堆内存分配；某些对象没有逃逸出方法，可能被优化为在栈上分配

谈谈 JVM 中的常量池

JDK 1.8 开始

1、 字符串常量池：存放在堆中，包括 String 对象执行 intern() 方法后存的地方、双引号直接引用的字符串

2、 运行时常量池：存放在方法区，属于元空间，是类加载后的一些存储区域，大多数是类中 constant_pool 的内容

3、 类文件常量池：constant_pool，JVM 定义的概念

谈谈动态年龄判断

1、 这里涉及到 -XX:TargetSurvivorRatio 参数，Survivor 区的目标使用率默认 50，即 Survivor 区对象目标使用率为 50%。

2、 Survivor 区相同年龄所有对象大小的总和（Survivor 区内存大小 * 这个目标使用率)时，大于或等于该年龄的对象直接进入老年代。

3、 当然，这里还需要考虑参数 -XX:MaxTenuringThreshold 晋升年龄最大阈值

谈谈永久代

1、 JDK 8 之前，Hotspot 中方法区的实现是永久代（Perm）

2、 JDK 7 开始把原本放在永久代的字符串常量池、静态变量等移出到堆，JDK 8 开始去除永久代，使用元空间（Metaspace），永久代剩余内容移至元空间，元空间直接在本地内存分配。

JVM 有哪些运行时内存区域?

Java 8

1、 The pc Register，程序计数器

2、 Java Virtual Machine Stacks，Java 虚拟机栈

3、 Heap，堆

4、 Method Area，方法区

5、 Run-Time Constant Pool，运行时常量池

6、 Native Method Stacks，本地方法栈

运行时栈帧包含哪些结构?

1、 局部变量表

2、 操作数栈

3、 动态连接

4、 返回地址

5、 附加信息

JVM 的内存模型是什么?

JVM 试图定义一种统一的内存模型，能将各种底层硬件以及操作系统的内存访问差异进行封装，使 Java 程序在不同硬件以及操作系统上都能达到相同的并发效果。它分为工作内存和主内存，线程无法对主存储器直接进行操作，如果一个线程要和另外一个线程通信，那么只能通过主存进行交换。

JVM 如何确定垃圾对象?

1、 JVM 采用的是可达性分析算法，通过 GC Roots 来判定对象是否存活，从 GC Roots 向下追溯、搜索，会产生 Reference Chain。当一个对象不能和任何一个 GC Root 产生关系时，就判定为垃圾。

2、 软引用和弱引用，也会影响对象的回收。内存不足时会回收软引用对象；GC 时会回收弱引用对象。

哪些是 GC Roots?

1、 在虚拟机栈（栈帧中的本地变量表）中引用的对象，譬如各个线程被调用的方法堆栈中使用到的参数、局部变量、临时变量等。

2、 在方法区中类静态属性引用的对象，譬如 Java 类的引用类型静态变量。

3、 在方法区中常量引用的对象，譬如字符串常量池（String Table）里的引用。

4、 在本地方法栈中 JNI（即通常所说的 Native 方法）引用的对象。

5、 Java 虚拟机内部的引用，如基本数据类型对应的 Class 对象，一些常驻的异常对象（比如 NullPointExcepiton、OutOfMemoryError）等，还有系统类加载器。

6、 所有被同步锁（synchronized 关键字）持有的对象。

7、 反映 Java 虚拟机内部情况的 JMXBean、JVMTI 中注册的回调、本地代码缓存等。

被引用的对象就一定能存活吗?

不一定，看 Reference 类型，弱引用在 GC 时会被回收，软引用在内存不足的时候，即 OOM 前会被回收，但如果没有在 Reference Chain 中的对象就一定会被回收。

强引用、软引用、弱引用、虚引用是什么，有什么区别?

1、 强引用，就是普通的对象引用关系，如 String s = new String("ConstXiong")

2、 软引用，用于维护一些可有可无的对象。只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。SoftReference 实现

3、 弱引用，相比软引用来说，要更加无用一些，它拥有更短的生命周期，当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。WeakReference 实现

4、 虚引用是一种形同虚设的引用，在现实场景中用的不是很多，它主要用来跟踪对象被垃圾回收的活动。PhantomReference 实现

你做过 JVM 调优，说说如何查看 JVM 参数默认值?

1、 jps -v 可以查看 jvm 进程显示指定的参数

2、 使用 -XX:+PrintFlagsFinal 可以看到 JVM 所有参数的值

3、 jinfo 可以实时查看和调整虚拟机各项参数

工作中常用的 JVM 配置参数有哪些?

Java 8 为例

日志

1、 -XX:+PrintFlagsFinal，打印 JVM 所有参数的值

2、 -XX:+PrintGC，打印 GC 信息

3、 -XX:+PrintGCDetails，打印 GC 详细信息

4、 -XX:+PrintGCTimeStamps，打印 GC 的时间戳

5、 -Xloggc:filename，设置 GC log 文件的位置

6、 -XX:+PrintTenuringDistribution，查看熬过收集后剩余对象的年龄分布信息

内存设置

1、 -Xms，设置堆的初始化内存大小

2、 -Xmx，设置堆的最大内存

3、 -Xmn，设置新生代内存大小

4、 -Xss，设置线程栈大小

5、 -XX:NewRatio，新生代与老年代比值

6、 -XX:SurvivorRatio，新生代中 Eden 区与两个 Survivor 区的比值，默认为 8，即 Eden:Survivor:Survivor=8:1:1

7、 -XX:MaxTenuringThreshold，从年轻代到老年代，最大晋升年龄。CMS 下默认为 6，G1 下默认为 15

8、 -XX:MetaspaceSize，设置元空间的大小，第一次超过将触发 GC

9、 -XX:MaxMetaspaceSize，元空间最大值

10、 -XX:MaxDirectMemorySize，用于设置直接内存的最大值，限制通过 DirectByteBuffer 申请的内存

11、 -XX:ReservedCodeCacheSize，用于设置 JIT 编译后的代码存放区大小，如果观察到这个值有限制，可以适当调大，一般够用即可

设置垃圾收集相关

1、 -XX:+UseSerialGC，设置串行收集器

2、 -XX:+UseParallelGC，设置并行收集器

3、 -XX:+UseConcMarkSweepGC，使用 CMS 收集器

4、 -XX:ParallelGCThreads，设置 Parallel GC 的线程数

5、 -XX:MaxGCPauseMillis，GC 最大暂停时间 ms

6、 -XX:+UseG1GC，使用 G1 垃圾收集器

CMS 垃圾回收器相关

1、 -XX:+UseCMSInitiatingOccupancyOnly

2、 -XX:CMSInitiatingOccupancyFraction，与前者配合使用，指定 MajorGC 的发生时机

3、 -XX:+ExplicitGCInvokesConcurrent，代码调用 System.gc() 开始并行 FullGC，建议加上这个参数

4、 -XX:+CMSScavengeBeforeRemark，表示开启或关闭在 CMS 重新标记阶段之前的清除（YGC）尝试，它可以降低 remark 时间，建议加上

5、 -XX:+ParallelRefProcEnabled，可以用来并行处理 Reference，以加快处理速度，缩短耗时

G1 垃圾回收器相关

1、 -XX:MaxGCPauseMillis，用于设置目标停顿时间，G1 会尽力达成

2、 -XX:G1HeapRegionSize，用于设置小堆区大小，建议保持默认

3、 -XX:InitiatingHeapOccupancyPercent，表示当整个堆内存使用达到一定比例（默认是 45%），并发标记阶段就会被启动

4、 -XX:ConcGCThreads，表示并发垃圾收集器使用的线程数量，默认值随 JVM 运行的平台不同而变动，不建议修改

参数查询官网地址：
https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

建议面试时最好能记住 CMS 和 G1 的参数，特点突出使用较多，被问的概率大

谈谈对 OOM 的认识

除了程序计数器，其他内存区域都有 OOM 的风险。

1、 栈一般经常会发生 StackOverflowError，比如 32 位的 windows 系统单进程限制 2G 内存，无限创建线程就会发生栈的 OOM

2、 Java 8 常量池移到堆中，溢出会出 java.lang.OutOfMemoryError: Java heap space，设置最大元空间大小参数无效

3、 堆内存溢出，报错同上，这种比较好理解，GC 之后无法在堆中申请内存创建对象就会报错

4、 方法区 OOM，经常会遇到的是动态生成大量的类、jsp 等

5、 直接内存 OOM，涉及到 -XX:MaxDirectMemorySize 参数和 Unsafe 对象对内存的申请

什么情况发生栈溢出?

-Xss 可以设置线程栈的大小，当线程方法递归调用层次太深或者栈帧中的局部变量过多时，会出现栈溢出错误 java.lang.StackOverflowError

你有哪些手段来排查 OOM 的问题?

1、 增加两个参数 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof，当 OOM 发生时自动 dump 堆内存信息到指定目录

2、 同时 jstat 查看监控 JVM 的内存和 GC 情况，先观察问题大概出在什么区域

3、 使用 MAT 工具载入到 dump 文件，分析大对象的占用情况，比如 HashMap 做缓存未清理，时间长了就会内存溢出，可以把改为弱引用

遇到过元空间溢出吗?

元空间在本地内存上，默认是没有上限的，不加限制出了问题会影响整个服务器的，所以也是比较危险的。-XX:MaxMetaspaceSize 可以指定最大值。

一般使用动态代理的框架会生成很多 Java 类，如果占用空间超出了我们的设定最大值，会发生元空间溢出。

遇到过堆外内存溢出吗?

1、 Unsafe 类申请内存、JNI 对内存进行操作、Netty 调用操作系统的 malloc 函数的直接内存，这些内存是不受 JVM 控制的，不加限制的使用，很容易发生溢出。这种情况有个显著特点，dump 的堆文件信息正常甚至很小。

2、 -XX:MaxDirectMemorySize 可以指定最大直接内存，但限制不住所有堆外内存的使用。

谈谈你知道的垃圾回收算法

判断对象是否可回收的算法有两种：

1、 Reference Counting GC，引用计数算法

2、 Tracing GC，可达性分析算法

3、 JVM 各厂商基本都是用的 Tracing GC 实现

4、 大部分垃圾收集器遵从了分代收集(Generational Collection)理论。

5、 针对新生代与老年代回收垃圾内存的特点，提出了 3 种不同的算法：

1、 标记-清除算法(Mark-Sweep)

标记需回收对象，统一回收；或标记存活对象，回收未标记对象。
缺点：

大量对象需要标记与清除时，效率不高
标记、清除产生的大量不连续内存碎片，导致无法分配大对象

2、 标记-复制算法(Mark-Copy)

可用内存等分两块，使用其中一块 A，用完将存活的对象复制到另外一块 B，一次性清空 A，然后改分配新对象到 B，如此循环。
缺点：

不适合大量对象不可回收的情况，换句话说就是仅适合大量对象可回收，少量对象需复制的区域
只能使用内存容量的一半，浪费较多内存空间

3、 标记-整理算法(Mark-Compact)

标记存活的对象，统一移到内存区域的一边，清空占用内存边界以外的内存。
缺点：

移动大量存活对象并更新引用，需暂停程序运行

谈谈你知道的垃圾收集器

Serial 特点：

1、 JDK 1.3 开始提供

2、 新生代收集器

3、 无线程交互开销，单线程收集效率最高

4、 进行垃圾收集时需要暂停用户线程

5、 适用于客户端，小内存堆的回收

ParNew 特点：

1、 是 Serial 收集器的多线程并行版

2、 JDK 7 之前首选的新生代收集器

3、 第一款支持并发的收集器，首次实现垃圾收集线程与用户线程基本上同时工作

4、 除 Serial 外，只有它能与 CMS 配合

Parallel Scavenge 特点：

1、 新生代收集器

2、 标记-复制算法

3、 多线程并行收集器

4、 追求高吞吐量，即最小的垃圾收集时间

5、 可以配置最大停顿时间、垃圾收集时间占比

6、 支持开启垃圾收集自适应调节策略，追求适合的停顿时间或最大的吞吐量

Serial Old 特点：

与 Serial 类似，是 Serial 收集器的老年代版本
使用标记-整理算法

Parallel Old 特点：

1、 JDK 6 开始提供

2、 Parallel Scavenge 的老年代版

3、 支持多线程并发收集

4、 标记-整理算法

5、 Parallel Scavenge + Parallel Old 是一个追求高吞吐量的组合

CMS 特点：

1、 标记-清除算法

2、 追求最短回收停顿时间

3、 多应用于关注响应时间的 B/S 架构的服务端

4、 并发收集、低停顿

5、 占用一部分线程资源，应用程序变慢，吞吐量下降

6、 无法处理浮动垃圾，可能导致 Full GC

7、 内存碎片化问题

G1 特点：

1、 JDK 6 开始实验，JDK 7 商用

2、 面向服务端，JDK 9 取代 Parallel Scavenge + Parallel Old

3、 结合标记-整理、标记-复制算法

4、 首创局部内存回收设计思路

5、 基于 Region 内存布局，采用不同策略实现分代

6、 不再使用固定大小、固定数量的堆内存分代区域划分

7、 优先回收价收益最大的 Region

8、 单个或多个 Humongous 区域存放大对象

9、 使用记忆集解决跨 Region 引用问题

10、 复杂的卡表实现，导致更高的内存占用，堆的 10% ~ 20%

11、 全功能垃圾收集器

12、 追求有限的时间内最高收集效率、延迟可控的情况下最高吞吐量

13、 追求应付内存分配速率，而非一次性清掉所有垃圾内存

14、 适用于大内存堆

Shenandoah 特点：

1、 追求低延迟，停顿 10 毫秒以内

2、 OpenJDK 12 新特性，RedHat 提供

3、 连接矩阵代替记忆集，降低内存使用与伪共享问题出现概率

ZGC 特点：

1、 JDK 11 新加的实验性质的收集器

2、 追求低延迟，停顿 10 毫秒以内

3、 基于 Region 内存布局

4、 未设分代

5、 读屏障、染色指针、内存多重映射实现可并发的标记-整理算法

6、 染色指针和内存多重映射设计精巧，解决部分性能问题，但降低了可用最大内存、操作系统受限、只支持 32 位、不支持压缩指针等

7、 成绩亮眼、性能彪悍

生产环境用的什么 JDK？如何配置的垃圾收集器？

Oracle JDK 1.8

JDK 1.8 中有 Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1，默认使用 Parallel Scavenge + Parallel Old。

1、 Serial 系列是单线程垃圾收集器，处理效率很高，适合小内存、客户端场景使用，使用参数 -XX:+UseSerialGC 显式启用。

2、 Parallel 系列相当于并发版的 Serial，追求高吞吐量，适用于较大内存并且有多核 CPU 的环境，默认或显式使用参数 -XX:+UseParallelGC 启用。可以使用 -XX:MaxGCPauseMillis 参数指定最大垃圾收集暂停毫秒数，收集器会尽量达到目标；使用 -XX:GCTimeRatio 指定期望吞吐量大小，默认 99，用户代码运行时间:垃圾收集时间=99:1。

3、 CMS，追求垃圾收集暂停时间尽可能短，适用于服务端较大内存且多 CPU 的应用，使用参数 -XX:+UseConcMarkSweepGC 显式开启，会同时作用年轻代与老年代，但有浮动垃圾和内存碎片化的问题。

4、 G1，主要面向服务端应用的垃圾收集器，适用于具有大内存的多核 CPU 的服务器，追求较小的垃圾收集暂停时间和较高的吞吐量。首创局部内存回收设计思路，采用不同策略实现分代，不再使用固定大小、固定数量的堆内存分代区域划分，而是基于 Region 内存布局，优先回收价收益最大的 Region。使用参数 -XX:+UseG1GC 开启。

我们生产环境使用了 G1 收集器，相关配置如下

1、 -Xmx12g

2、 -Xms12g

3、 -XX:+UseG1GC

4、 -XX:InitiatingHeapOccupancyPercent=45

5、 -XX:MaxGCPauseMillis=200

6、 -XX:MetaspaceSize=256m

7、 -XX:MaxMetaspaceSize=256m

8、 -XX:MaxDirectMemorySize=512m

9、 -XX:G1HeapRegionSize 未指定

核心思路：

1、 每个内存区域设置上限，避免溢出

2、 堆设置为操作系统的 70%左右，超过 8 G，首选 G1

3、 根据老年代对象提升速度，调整新生代与老年代之间的内存比例

4、 等过 GC 信息，针对项目敏感指标优化，比如访问延迟、吞吐量等

如何查看 JVM 当前使用的是什么垃圾收集器?

-XX:+PrintCommandLineFlags 参数可以打印出所选垃圾收集器和堆空间大小等设置

如果开启了 GC 日志详细信息，里面也会包含各代使用的垃圾收集器的简称

如何开启和查看 GC 日志?

常见的 GC 日志开启参数包括：

1、 -Xloggc:filename，指定日志文件路径

2、 -XX:+PrintGC，打印 GC 基本信息

3、 -XX:+PrintGCDetails，打印 GC 详细信息

4、 -XX:+PrintGCTimeStamps，打印 GC 时间戳

5、 -XX:+PrintGCDateStamps，打印 GC 日期与时间

6、 -XX:+PrintHeapAtGC，打印 GC 前后的堆、方法区、元空间可用容量变化

7、 -XX:+PrintTenuringDistribution，打印熬过收集后剩余对象的年龄分布信息，有助于 MaxTenuringThreshold 参数调优设置

8、 -XX:+PrintAdaptiveSizePolicy，打印收集器自动设置堆空间各分代区域大小、收集目标等自动调节的相关信息

9、 -XX:+PrintGCApplicationConcurrentTime，打印 GC 过程中用户线程并发时间

10、 -XX:+PrintGCApplicationStoppedTime，打印 GC 过程中用户线程停顿时间

11、 -XX:+HeapDumpOnOutOfMemoryError，堆 oom 时自动 dump

12、 -XX:HeapDumpPath，堆 oom 时 dump 文件路径

Java 9 JVM 日志模块进行了重构，参数格式发生变化，这个需要知道。

GC 日志输出的格式，会随着上面的参数不同而发生变化。关注各个分代的内存使用情况、垃圾回收次数、垃圾回收的原因、垃圾回收占用的时间、吞吐量、用户线程停顿时间。

借助工具可视化工具可以更方便的分析，在线工具 GCeasy；离线版可以使用 GCViewer。

如果现场环境不允许，可以使用 JDK 自带的 jstat 工具监控观察 GC 情况。

JVM 监控与分析工具你用过哪些？介绍一下。

1、 jps，显示系统所有虚拟机进程信息的命令行工具

2、 jstat，监视分析虚拟机运行状态的命令行工具

3、 jinfo，查看和调整虚拟机参数的命令行工具

4、 jmap，生成虚拟机堆内存转储快照的命令行工具

5、 jhat，显示和分析虚拟机的转储快照文件的命令行工具

6、 jstack，生成虚拟机的线程快照的命令行工具

7、 jcmd，虚拟机诊断工具，JDK 7 提供

8、 jhsdb，基于服务性代理实现的进程外可视化调试工具，JDK 9 提供

9、 JConsole，基于 JMX 的可视化监视和管理工具

10、 jvisualvm，图形化虚拟机使用情况的分析工具

11、 Java Mission Control，监控和管理 Java 应用程序的工具

1、 MAT，Memory Analyzer Tool，虚拟机内存分析工具

2、 vjtools，唯品会的包含核心类库与问题分析工具

3、 arthas，阿里开源的 Java 诊断工具

4、 greys，JVM 进程执行过程中的异常诊断工具

5、 GCHisto，GC 分析工具

6、 GCViewer，GC 日志文件分析工具

7、 GCeasy，在线版 GC 日志文件分析工具

8、 JProfiler，检查、监控、追踪 Java 性能的工具

9、 BTrace，基于动态字节码修改技术(Hotswap)实现的 Java 程序追踪与分析工具

下面可以重点体验下：

JDK 自带的命令行工具方便快捷，不是特别复杂的问题可以快速定位；

阿里的 arthas 命令行也不错；

可视化工具 MAT、JProfiler 比较强大。

JIT 是什么?

Just In Time Compiler 的简称，即时编译器。为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器就是 JIT。

谈谈双亲委派模型

1、 Parents Delegation Model，这里的 Parents 翻译成双亲有点不妥，类加载向上传递的过程中只有单亲；parents 更多的是多级向上的意思。

2、 除了顶层的启动类加载器，其他的类加载器在加载之前，都会委派给它的父加载器进行加载，一层层向上传递，直到所有父类加载器都无法加载，自己才会加载该类。

3、 双亲委派模型，更好地解决了各个类加载器协作时基础类的一致性问题，避免类的重复加载；防止核心 API 库被随意篡改。

JDK 9 之前

1、 启动类加载器（Bootstrp ClassLoader），加载 /lib/rt.jar、-Xbootclasspath

2、 扩展类加载器（Extension ClassLoader）

sun.misc.Launcher$ExtClassLoader，加载 /lib/ext、java.ext.dirs

3、 应用程序类加载器（Application ClassLoader，sun.misc.Launcher$AppClassLoader），加载 CLASSPTH、-classpath、-cp、Manifest

4、 自定义类加载器

JDK 9 开始 Extension ClassLoader 被 Platform ClassLoader 取代，启动类加载器、平台类加载器、应用程序类加载器全都继承于 jdk.internal.loader.BuiltinClassLoader

类加载代码逻辑

```
protected synchronized Class<?> loadClass(String name, boolean
resolve) throws ClassNotFoundException {
  // 首先，检查请求的类是否已经被加载过了
  Class c = findLoadedClass(name);
  if (c == null) {
    try {
      if (parent != null) {
        c = parent.loadClass(name, false);
      } else {
        c = findBootstrapClassOrNull(name);
      }
    } catch (ClassNotFoundException e) {
      // 如果父类加载器抛出 ClassNotFoundException
      // 说明父类加载器无法完成加载请求
    }
```

```
        if (c == null) {
            // 在父类加载器无法加载时
            // 再调用本身的 findClass 方法来进行类加载
            c = findClass(name);
        }
    }
    if (resolve) {
        resolveClass(c);
    }
    return c;
}
```

列举一些你知道的打破双亲委派机制的例子。为什么要打破?

1、 JNDI 通过引入线程上下文类加载器，可以在 Thread.setContextClassLoader 方法设置，默认是应用程序类加载器，来加载 SPI 的代码。有了线程上下文类加载器，就可以完成父类加载器请求子类加载器完成类加载的行为。打破的原因，是为了 JNDI 服务的类加载器是启动器类加载，为了完成高级类加载器请求子类加载器（即上文中的线程上下文加载器）加载类。

2、 Tomcat，应用的类加载器优先自行加载应用目录下的 class，并不是先委派给父加载器，加载不了才委派给父加载器。打破的目的是为了完成应用间的类隔离。

3、 OSGi，实现模块化热部署，为每个模块都自定义了类加载器，需要更换模块时，模块与类加载器一起更换。其类加载的过程中，有平级的类加载器加载行为。打破的原因是为了实现模块热替换。

4、 JDK 9，Extension ClassLoader 被 Platform ClassLoader 取代，当平台及应用程序类加载器收到类加载请求，在委派给父加载器加载前，要先判断该类

是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责那个模块的加载器完成加载。打破的原因，是为了添加模块化的特性。

说一下垃圾分代收集的过程

分为新生代和老年代，新生代默认占总空间的 1/3，老年代默认占 2/3。
新生代使用复制算法，有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1。

当新生代中的 Eden 区内存不足时，就会触发 Minor GC，过程如下：

1、 在 Eden 区执行了第一次 GC 之后，存活的对象会被移动到其中一个 Survivor 分区；

2、 Eden 区再次 GC，这时会采用复制算法，将 Eden 和 from 区一起清理，存活的对象会被复制到 to 区；

3、 移动一次，对象年龄加 1，对象年龄大于一定阀值会直接移动到老年代

4、 Survivor 区相同年龄所有对象大小的总和 (Survivor 区内存大小 * 这个目标使用率)时，大于或等于该年龄的对象直接进入老年代。其中这个使用率通过 -XX:TargetSurvivorRatio 指定，默认为 50%

5、 Survivor 区内存不足会发生担保分配

6、 超过指定大小的对象可以直接进入老年代

Major GC，指的是老年代的垃圾清理，但并未找到明确说明何时在进行 Major GC
FullGC，整个堆的垃圾收集，触发条件：

1、 每次晋升到老年代的对象平均大小>老年代剩余空间

2、 MinorGC 后存活的对象超过了老年代剩余空间

3、 元空间不足

4、 System.gc() 可能会引起

5、 CMS GC 异常，promotion failed:MinorGC 时，survivor 空间放不下，对象只能放入老年代，而老年代也放不下造成；concurrent mode failure:GC 时，同时有对象要放入老年代，而老年代空间不足造成

6、 堆内存分配很大的对象

如何找到死锁的线程?

死锁的线程可以使用 jstack 指令 dump 出 JVM 的线程信息。

jstack -l <pidthreads.txt

有时候需要 dump 出现异常，可以加上 -F 指令，强制导出

jstack -F -l <pidthreads.txt

如果存在死锁，一般在文件最后会提示找到 deadlock 的数量与线程信息

invokedynamic 指令是干什么的?

Java 7 开始，新引入的字节码指令，可以实现一些动态类型语言的功能。Java 8 的 Lambda 表达式就是通过 invokedynamic 指令实现，使用方法句柄实现。

什么是方法内联?

为了减少方法调用的开销，可以把一些短小的方法，纳入到目标方法的调用范围之内，这样就少了一次方法调用，提升速度

什么是逃逸分析?

分析对象动态作用域

1、 当一个对象在方法里面被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，这种称为方法逃逸；

2、 被外部线程访问到，譬如赋值给可以在其他线程中访问的实例变量，这种称为线程逃逸；

3、 从不逃逸

4、 如果能证明一个对象不会逃逸到方法或线程之外，或者逃逸程度比较低（只逃逸出方法而不会逃逸出线程），则可能为这个对象实例采取不同程度的优化，如栈上分配、标量替换、同步消除。

描述一下什么情况下，对象会从年轻代进入老年代

1、 对象的年龄超过一定阀值，-XX:MaxTenuringThreshold 可以指定该阀值

2、 动态对象年龄判定，有的垃圾回收算法，比如 G1，并不要求 age 必须达到 15 才能晋升到老年代，它会使用一些动态的计算方法

3、 大小超出某个阀值的对象将直接在老年代上分配，值默认为 0，意思是全部首选 Eden 区进行分配，-XX:PretenureSizeThreshold 可以指定该阀值，部分收集器不支持

4、 分配担保，当 Survivor 空间不够的时候，则需要依赖其他内存（指老年代）进行分配担保，这个时候，对象也会直接在老年代上分配

safepoint 是什么?

1、 为了减少对象引用的扫描，使用 OopMap 的数据结构在特定的位置记录下栈里和寄存器里哪些位置是引用；

2、 但为了避免给每条指令都生成 OopMap 记录占用大量内存的问题，只在特定位置记录这些信息。

3、 安全点的选定既不能太少以至于让收集器等待时间过长，也不能太过频繁以至于过分增大运行时的内存负荷。安全点位置的选取基本上是以“是否具有让程序长时间执行的特征”为标准进行选定的，如方法调用、循环跳转、异常跳转等都属于指令序列复用。

MinorGC、MajorGC、FullGC 什么时候发生?

1、 MinorGC 在年轻代空间不足的时候发生

2、 MajorGC 指的是老年代的 GC，出现 MajorGC 一般经常伴有 MinorGC

3、 FullGC 老年代无法再分配内存；元空间不足；显示调用 System.gc；像 CMS 一类的垃圾回收器，在 MinorGC 出现 promotion failure 时也会发生 FullGC

说说类加载的过程

1、 加载（Loading），通过一个类的全限定名来获取定义此类的二进制字节流；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数据结构；在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

2、 验证（Verification），确保 Class 文件的字节流中包含的信息符合《Java 虚拟机规范》的全部约束要求，保证这些信息被当作代码运行后不会危害虚拟机自身的安全。

3、 准备（Preparation），正式为类中定义的变量（即静态变量，被 static 修饰的变量）分配内存并设置类变量初始值。

4、 解析（Resolution），是 JVM 将常量池内的符号引用替换为直接引用的过程。

5、 初始化（Initialization），执行类构造器 <clinit 方法的过程，执行所有类变量的赋值动作和静态语句块（static{}块）。

其中验证、准备、解析统称为称为连接（Linking）

可以描述一下 class 文件的结构吗?

1、 Class 文件包含了 Java 虚拟机的指令集、符号表、辅助信息的字节码(Byte Code)，是实现跨操作系统和语言无关性的基石之一。

2、 一个 Class 文件定义了一个类或接口的信息，是以 8 个字节为单位，没有分隔符，按顺序紧凑排在一起的二进制流。

3、 用 "无符号数" 和 "表" 组成的伪结构来存储数据。

4、 无符号数：基本数据类型，用来描述数字、索引引用、数量值、字符串值，如 u1、u2 分别表示 1 个字节、2 个字节

10、 表：无符号数和其他表组成，命名一般以 "_info" 结尾

组成部分

1、 魔数 Magic Number

Class 文件头 4 个字节，0xCAFEBABE
作用是确定该文件是 Class 文件

2、 版本号

4 个字节，前 2 个是次版本号 Minor Version，后 2 个主版本号 Major Version
从 45 (JDK1.0) 开始，如 0x00000032 转十进制就是 50，代表 JDK 6
低版本的虚拟机跑不了高版本的 Class 文件

3、 常量池

常量容量计数值(constant_pool_count)，u2，从 1 开始。如 0x0016 十进制 22 代表有
21 项常量

每项常量都是一个表，目前 17 种

特点：Class 文件中最大数据项目之一、第一个出现表数据结构

4、 访问标志

2 个字节，表示类或接口的访问标志

5、 类索引、父类索引、接口索引集合

类索引(this_class)、父类索引(super_class)，u2

接口索引集合(interfaces)，u2 集合

类索引确定类的全限定名、父类索引确定父类的全限定名、接口索引集合确定实现接口

索引值在常量池中查找对应的常量

6、 字段表(field_info)集合

描述接口或类申明的变量

fields_count，u2，表示字段表数量；后面接着相应数量的字段表

9 种字段访问标志

7、 方法表(method_info)集合

描述接口或类申明的方法

methods_count，u2，表示方法表数量；后面接着相应数量的方法表

12 种方法访问标志

方法表结构与字段表结构一致

8、 属性表(attribute_info)集合

class 文件、字段表、方法表可携带属性集合，描述特有信息

预定义 29 项属性，可自定义写入不重名属性

说说 JVM 如何执行 class 中的字节码。

1、 JVM 先加载包含字节码的 class 文件，存放在方法区，实际运行时，虚拟机会执行方法区内的代码。Java 虚拟机在内存中划分出栈和堆来存储运行时的数据。

2、 运行过程中，每当调用进入 Java 方法，都会在 Java 方法栈中生成一个栈帧，用来支持虚拟机进行方法的调用与执行，包含了局部变量表、操作数栈、动态链接、方法返回地址等信息。

3、 当退出当前执行的方法时，不管正常返回还是异常返回，Java 虚拟机均会弹出当前线程的当前栈帧，并将之舍弃。

4、 方法的调用，需要通过解析完成符号引用到直接引用；通过分派完成动态找到被调用的方法。

5、 从硬件角度来看，Java 字节码无法直接执行。因此，Java 虚拟机需要将字节码翻译成机器码。翻译过程由两种形式：第一种是解释执行，即将遇到的字节一边码翻译成机器码一边执行；第二种是即时编译(Just-In-Time compilation,JIT)，即将一个方法中包含的所有字节码编译成机器码后再执行。在 HotSpot 里两者都有，解释执行在启动时节约编译时间执行速度较快；随着时间的推移，编译器逐渐会返回作用，把越来越多的代码编译成本地代码后，可以获取更高的执行效率。

生产环境 CPU 占用过高，你如何解决?

1、 top + H 指令找出占用 CPU 最高的进程的 pid

2、 top -H -p

在该进程中找到，哪些线程占用的 CPU 最高的线程，记录下 tid

3、 jstack -l

threads.txt，导出进程的线程栈信息到文本，导出出现异常的话，加上 -F 参数

4、 将 tid 转换为十六进制，在 threads.txt 中搜索，查到对应的线程代码执行栈，在代码中查找占 CPU 比较高的原因。其中 tid 转十六进制，可以借助 Linux 的 printf "%x" tid 指令

我用上述方法查到过，jvm 多条线程疯狂 full gc 导致的 CPU 100% 的问题和 JDK1.6 HashMap 并发 put 导致线程 CPU 100% 的问题

生产环境服务器变慢，如何诊断处理?

1、 使用 top 指令，服务器中 CPU 和 内存的使用情况，-H 可以按 CPU 使用率降序，-M 内存使用率降序。排除其他进程占用过高的硬件资源，对 Java 服务造成影响。

2、 如果发现 CPU 使用过高，可以使用 top 指令查出 JVM 中占用 CPU 过高的线程，通过 jstack 找到对应的线程代码调用，排查出问题代码。

3、 如果发现内存使用率比较高，可以 dump 出 JVM 堆内存，然后借助 MAT 进行分析，查出大对象或者占用最多的对象来自哪里，为什么会长时间占用这么多；如果 dump 出的堆内存文件正常，此时可以考虑堆外内存被大量使用导致出现问题，需要借助操作系统指令 pmap 查出进程的内存分配情况、gdb dump 出具体内存信息、perf 查看本地函数调用等。

4、 如果 CPU 和 内存使用率都很正常，那就需要进一步开启 GC 日志，分析用户线程暂停的时间、各部分内存区域 GC 次数和时间等指标，可以借助 jstat 或可视化工具 GCeasy 等，如果问题出在 GC 上面的话，考虑是否是内存不够、根据垃圾对象的特点进行参数调优、使用更适合的垃圾收集器；分析 jstack 出来的各个线程状态。如果问题实在比较隐蔽，考虑是否可以开启 jmx，使用 visualmv 等可视化工具远程监控与分析。