JVM面试思考准备

JVM的知识体系-------见JVM知识体系思维导图
JVM的知识体系视频-------详细见炼数成金视频
JVM面试参考：
https://www.cnblogs.com/wangyayun/p/6557851.html
http://blog.csdn.net/hsk256/article/details/49104955
http://blog.csdn.net/sunsfan/article/details/52909024

一.jvm体系总体分四大块：

1.类的加载机制
2.jvm内存结构
3.GC算法 垃圾回收
4.GC分析 命令调优

二.类的加载机制

1.什么是类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

2.类的生命周期

1.加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象

2.连接，连接又包含三块内容：验证、准备、初始化。1）验证，文件格式、元数据、字节码、符号引用验证；2）准备，为类的静态变量分配内存，并将其初始化为默认值；3）解析，把类中的符号引用转换为直接引用

3.初始化，为类的静态变量赋予正确的初始值

4.使用，new出对象程序中使用

5.卸载，执行垃圾回收

3.类加载器

1.启动类加载器：Bootstrap ClassLoader，负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库

2.扩展类加载器：Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载DK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.*开头的类），开发者可以直接使用扩展类加载器。

3.应用程序类加载器：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器

4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

4.双亲委派模型

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

自底向上的检查，自顶向下的加载

注意双亲委派模式的问题：无法识别应用程序类加载器中的类

解决方案：设置一个上下文加载器角色解决

5.类加载机制

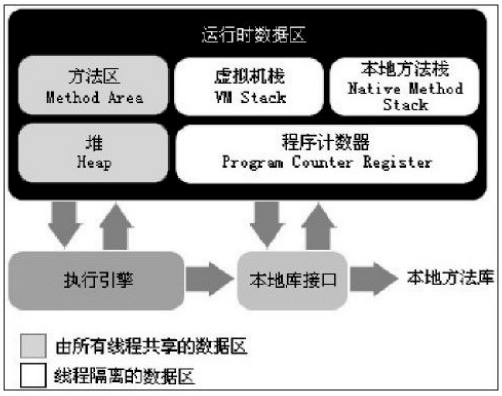
1.全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

2.父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

3.缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

二.jvm内存结构

jvm内存模式图



1.方法区和对是所有线程共享的内存区域；而java栈、本地方法栈和程序员计数器是运行是线程私有的内存区域。

2.Java堆（Heap）,是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

3.方法区（Method Area）,方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

在Java 8里已被废除了，被元空间取代；

4.程序计数器（Program Counter Register）,程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。

5.JVM栈（JVM Stacks）,与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

6.本地方法栈（Native Method Stacks）,本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

如和判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:

1. **引用计数法**

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A,B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2.**可达性算法**(引用链法)

该算法的思想是：从一个被称为**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。

在java中可以作为GC Roots的对象有以下几种:

- 虚拟机栈中引用的对象

- 方法区类静态属性引用的对象
- 方法区常量池引用的对象
- 本地方法栈JNI引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比**不一定会被回收**。当一个对象不可达GC Root时，这个对象并**不会立马被回收**，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记

如果对象在可达性分析中没有与GC Root的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行finalize()方法。当对象没有覆盖finalize()方法或者已被虚拟机调用过，那么就认为是没必要的。

如果该对象有必要执行finalize()方法，那么这个对象将会放在一个称为F-Queue的对队列中，虚拟机会触发一个Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果finalize()执行缓慢或者发生了死锁，那么就会造成F-Queue队列一直等待，造成了内存回收系统的崩溃。GC对处于F-Queue中的对象进行第二次被标记，这时，该对象将被移除"即将回收"集合，等待回收。

注意：

可以通过覆盖finalize方法来实现对象的"自救"，避免在标记后被回收，但通常不建议这么做；

对象的引用类型可分为：强引用、软引用（在内存溢出前会将这种类型的对象进行第二次回收）、弱引用（弱引用对象只能生存到下次垃圾回收之前）、虚引用（不会对生存时间存在影响，也无法通过它获取对象，主要目的就是在回收时收到一个系统通知）；

对象分配规则

1.对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。

2.大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

3.长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。

4.动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

5.空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

三.GC算法

GC最基础的算法有三种：标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

1.标记 -清除算法，"标记-清除"（Mark-Sweep）算法，如它的名字一样，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，在标记完成后统一回收掉所有被标记的对象。

2.复制算法，"复制"（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

3.标记-压缩算法，标记过程仍然与"标记-清除"算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

4.分代收集算法，"分代收集"（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

垃圾回收器

1.Serial收集器，串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。

2.ParNew收集器，ParNew收集器其实就是Serial收集器的多线程版本。

3.Parallel收集器，Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。

4.Parallel Old 收集器，Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和"标记－整理"算法

5.CMS收集器，CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。

6.G1收集器，G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

一款面向服务端应用的垃圾收集器，后续会替换掉CMS垃圾收集器；

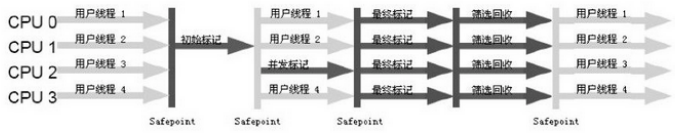
并行与并发（充分利用多核多CPU缩短STW时间）

分代收集（独立管理整个Java堆，但针对不同年龄的对象采取不同的策略）

空间整合（局部看是基于复制算法，从整体来看是基于标记-整理算法，都不会产生内存碎片）

可预测的停顿（可以明确指定在一个长度为M毫秒的时间片内垃圾收集不会超过N毫秒）

G1收集器的原理：



G1收集器

- 一款面向服务端应用的垃圾收集器，后续会替换掉CMS垃圾收集器；
- 特点：

```
并行与并发（充分利用多核多CPU缩短STW时间）
分代收集（独立管理整个Java堆，但针对不同年龄的对象采取不同的策略）
空间整合（局部看是基于复制算法，从整体来看是基于标记-整理算法，都不会产生内存碎片）
可预测的停顿（可以明确指定在一个长度为M毫秒的时间片内垃圾收集不会超过N毫秒）
```

- 将堆分为大小相等的独立区域，避免全区域的垃圾收集；新生代和老年代不再物理隔离，只是部分Region的集合；
- G1跟踪各个Region垃圾堆积的价值大小，在后台维护一个优先列表，根据允许的收集时间优先回收价值最大的Region；
- Region之间的对象引用以及其他收集器中的新生代与老年代之间的对象引用，采用Remembered Set来避免全堆扫描；
- 分为几个步骤，和CMS的过程比较类似：

```
初始标记（标记一下GC Roots能直接关联的对象并修改TAMS值，需要STW但耗时很短）
并发标记（从GC Root从堆中对象进行可达性分析找存活的对象，耗时较长但可以与用户线程并发执行）
最终标记（为了修正并发标记期间产生变动的那一部分标记记录，这一期间的变化记录在Remembered
Set Log里，然后合并到Remembered Set里，该阶段需要STW但是可并行执行）
筛选回收（对各个Region回收价值排序，根据用户期望的GC停顿时间制定回收计划来回收）；
```

四.调优命令

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

jstack可以看当前栈的情况，jmap查看内存，jhat 进行dump堆的信息

1.jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

2.jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

3.jmap，JVM Memory Map命令用于生成heap dump文件

4.jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看

5.jstack，用于生成java虚拟机当前时刻的线程快照。

6.jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

调优工具和性能分析工具

一定要设置将发生OOM的时候的堆内存Dump出来

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

1.jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控

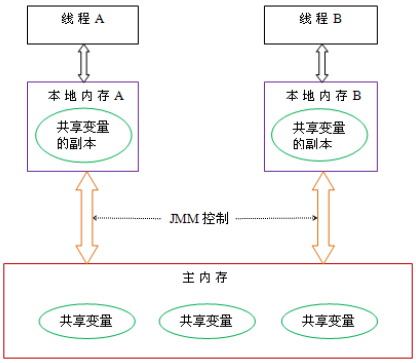
2.jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。

3.MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗

4.GChisto，一款专业分析gc日志的工具

java内存模型

java内存模型(JMM)是线程间通信的控制机制.JMM定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

注意：指令重排序的过程，导致线程内部是顺序执行的，在外部看来顺序会混乱

指令重排序，内存栅栏等

指令重排序：编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。

SafePoint是什么

比如GC的时候必须要等到Java线程都进入到safepoint的时候VMThread才能开始执行GC，

循环的末尾 (防止大循环的时候一直不进入safepoint，而其他线程在等待它进入safepoint)

方法返回前

调用方法的call之后

抛出异常的位置

你知道哪些JVM性能调优

设定堆最小内存大小-Xms

-Xmx：堆内存最大限制。

设定新生代大小。

新生代不宜太小，否则会有大量对象涌入老年代

-XX:NewSize：新生代大小

-XX:NewRatio 新生代和老生代占比

-XX:SurvivorRatio：伊甸园空间和幸存者空间的占比

设定垃圾回收器

年轻代用 -XX:+UseParNewGC （串行） 年老代用-XX:+UseConcMarkSweepGC （CMS）

设定锁的使用

多线程下关闭偏向锁，比较浪费资源

g1 和 cms 区别,吞吐量优先和响应优先的垃圾收集器选择

CMS是一种以最短停顿时间为目标的收集器

响应优先选择CMS,吞吐量高选择G1

当出现了内存溢出，你怎么排错

用jmap看内存情况，然后用 jstack主要用来查看某个Java进程内的线程堆栈信息

然后看报错的信息信息定位发生OOM的位置：堆，java虚拟机栈，永久区，直接内存

然后根据具体的问题提出具体的解决方案

JVM的一些参数设定：

```
-server       --启用能够执行优化的编译器，显著提高服务器的性能
-Xmx4000M     --堆最大值
-Xms4000M     --堆初始大小
-Xmn600M      --年轻代大小
-XX:PermSize=200M          --持久代初始大小
-XX:MaxPermSize=200M       --持久代最大值
-Xss256K                   --每个线程的栈大小
-XX:+DisableExplicitGC     --关闭System.gc()
-XX:SurvivorRatio=1        --年轻代中Eden区与两个Survivor区的比值
-XX:+UseConcMarkSweepGC    --使用CMS内存收集
-XX:+UseParNewGC           --设置年轻代为并行收集
-XX:+CMSParallelRemarkEnabled         --降低标记停顿
-XX:+UseCMSCompactAtFullCollection    --在FULL GC的时候，对年老代进行压缩，可能会影响性能，但是可
-XX:CMSFullGCsBeforeCompaction=0      --此值设置运行多少次GC以后对内存空间进行压缩、整理
-XX:+CMSClassUnloadingEnabled         --回收动态生成的代理类 SEE: http://stackoverflow.com/qu
-XX:LargePageSizeInBytes=128M         --内存页的大小不可设置过大， 会影响Perm的大小
-XX:+UseFastAccessorMethods           --原始类型的快速优化
-XX:+UseCMSInitiatingOccupancyOnly    --使用手动定义初始化定义开始CMS收集，禁止hostspot自行触发C
-XX:CMSInitiatingOccupancyFraction=80  --使用cms作为垃圾回收，使用80%后开始CMS收集
-XX:SoftRefLRUPolicyMSPerMB=0          --每兆堆空闲空间中SoftReference的存活时间
-XX:+PrintGCDetails                    --输出GC日志详情信息
-XX:+PrintGCApplicationStoppedTime     --输出垃圾回收期间程序暂停的时间
-Xloggc:$WEB_APP_HOME/.tomcat/logs/gc.log  --把相关日志信息记录到文件以便分析.
-XX:+HeapDumpOnOutOfMemoryError            --发生内存溢出时生成heapdump文件
-XX:HeapDumpPath=$WEB_APP_HOME/.tomcat/logs/heapdump.hprof  --heapdump文件地址
```

JVM调优：

- JVM调优：CPU使用率与Load值偏大（Thread count以及GC count）、关键接口响应时间很慢（GC time以及GC log中的STW的时间）、发生Full GC或者Old CMS GC非常频繁（内存泄露）；
- JVM停顿（尽量避免Full GC、关闭偏向锁、输出GC日志到内存文件系统、关闭JVM输出的jstat日志）；
- 将Java性能优化分为4个层级：应用层、数据库层、框架层、JVM层。每层优化难度逐级增加，涉及的知识和解决的问题也会不同。比如应用层需要理解代码逻辑，通过Java线程栈定位有问题代码行等；数据库层面需要分析SQL、定位死锁等；框架层需要懂源代码，理解框架机制；JVM 层需要对GC的类型和工作机制有深入了解，对各种 JVM 参数作用了然于胸；
- 围绕Java性能优化，有两种最基本的分析方法：现场分析法和事后分析法。现场分析法通过保留现场，再采用诊断工具分析定位。现场分析对线上影响较大，部分场景不太合适。事后分析法需要尽可能多收集现场数据，然后立即恢复服务，同时针对收集的现场数据进行事后分析和复现。
- OS 的诊断主要关注的是 CPU、Memory、I/O 三个方面。top、vmstat、 free –m、iostat；常用的Java应用诊断包括线程、堆栈、GC 等方面的诊断，可以使用jstack 、jstat、jmap；

名　称	主要作用
jps	JVM Process Status Tool，显示指定系统内所有的 HotSpot 虚拟机进程
jstat	JVM Statistics Monitoring Tool，用于收集 HotSpot 虚拟机各方面的运行数据
jinfo	Configuration Info for Java，显示虚拟机配置信息

（续）

名　称	主要作用
jmap	Memory Map for Java，生成虚拟机的内存转储快照（heapdump 文件）
jhat	JVM Heap Dump Browser，用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果
jstack	Stack Trace for Java，显示虚拟机的线程快照

偏向锁：

是否了解偏向锁？

- JVM锁有4种状态：无锁、偏向锁（通过MarkWord的线程ID）、轻量级锁（通过MarkWord的锁记录指针）、重量级锁；

锁	优　点	缺　点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长