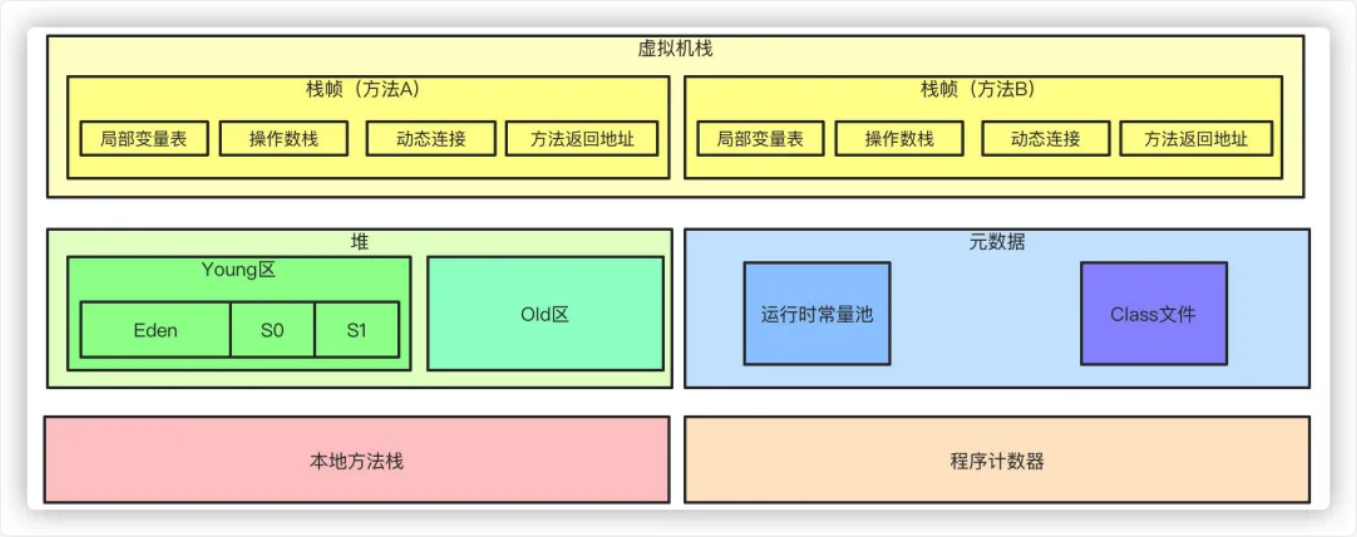
《我想进大厂》之JVM夺命连环10问

原创 科技缪缪 艾小仙 2020-10-24

收录于话题
#面试大全 29 #文章精选汇总 78

这是面试专题系列第五篇JVM篇。这一篇可能稍微比较长，没有耐心的同学建议直接拖到最后。

说说JVM的内存布局?



Java虚拟机主要包含几个区域：

堆：堆Java虚拟机中最大的一块内存，是线程共享的内存区域，基本上所有的对象实例数组都是在堆上分配空间。堆区细分为Yound区年轻代和Old区老年代，其中年轻代又分为Eden、S0、S1 3个部分，他们默认的比例是8:1:1的大小。

栈：栈是线程私有的内存区域，每个方法执行的时候都会在栈创建一个栈帧，方法的调用过程就对应着栈的入栈和出栈的过程。每个栈帧的结构又包含局部变量表、操作数栈、动态连接、方法返回地址。

局部变量表用于存储方法参数和局部变量。当第一个方法被调用的时候，他的参数会被传递至从0开始的连续的局部变量表中。

操作数栈用于一些字节码指令从局部变量表中传递至操作数栈，也用来准备方法调用的参数以及接收方法返回结果。

动态连接用于将符号引用表示的方法转换为实际方法的直接引用。

元数据：在Java1.7之前，包含方法区的概念，常量池就存在于方法区（永久代）中，而方法区本身是一个逻辑上的概念，在1.7之后则是把常量池移到了堆内，1.8之后移出了永久代的概念(方法区的概念仍然保留)，实现方式则是现在的元数据。它包含类的元信息和运行时常量池。

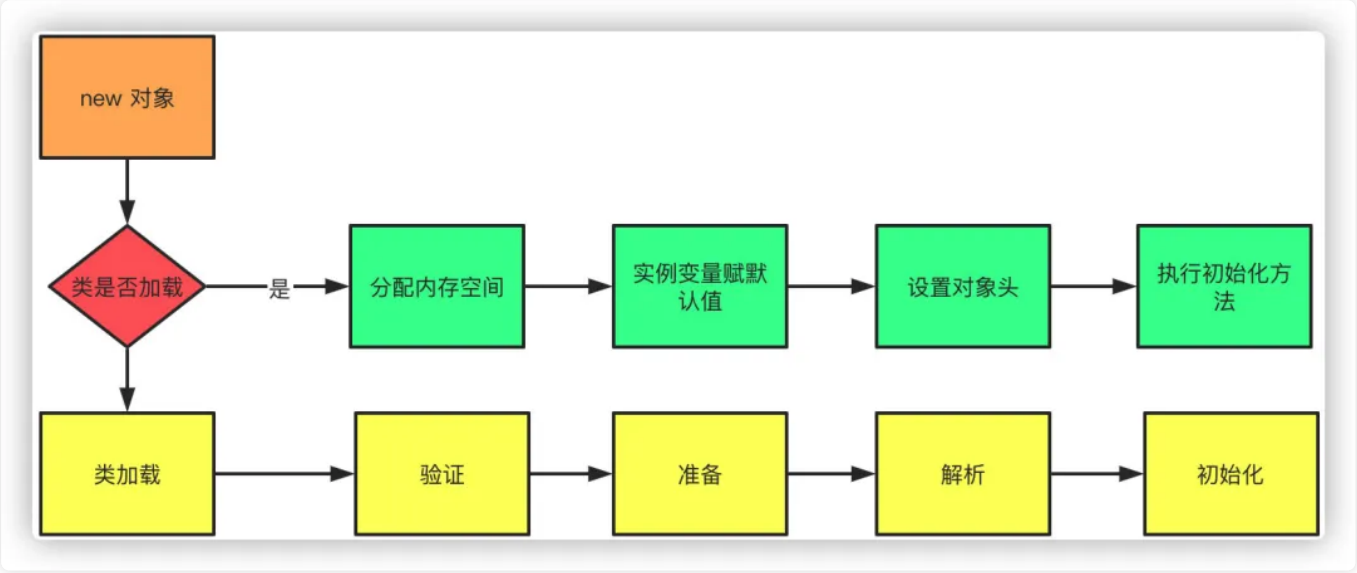
Class文件就是类和接口的定义信息。

运行时常量池就是类和接口的常量池运行时的表现形式。

本地方法栈：主要用于执行本地native方法的区域

程序计数器：也是线程私有的区域，用于记录当前线程下虚拟机正在执行的字节码的指令地址

知道new一个对象的过程吗?



当虚拟机遇见new关键字时候，实现判断当前类是否已经加载，如果类没有加载，首先执行类的加载机制，加载完成后再为对象分配空间、初始化等。

1. 首先校验当前类是否被加载，如果没有加载，执行类加载机制
2. 加载：就是从字节码加载成二进制流的过程

3. 验证：当然加载完成之后，当然需要校验Class文件是否符合虚拟机规范，跟我们接口请求一样，第一件事情当然是先做个参数校验了
4. 准备：为静态变量、常量赋默认值
5. 解析：把常量池中符号引用(以符号描述引用的目标)替换为直接引用(指向目标的指针或者句柄等)的过程
6. 初始化：执行static代码块(cinit)进行初始化，如果存在父类，先对父类进行初始化

Ps：静态代码块是绝对线程安全的，只能隐式被java虚拟机在类加载过程中初始化调用！(此处该有问题static代码块线程安全吗？)

当类加载完成之后，紧接着就是对象分配内存空间和初始化的过程

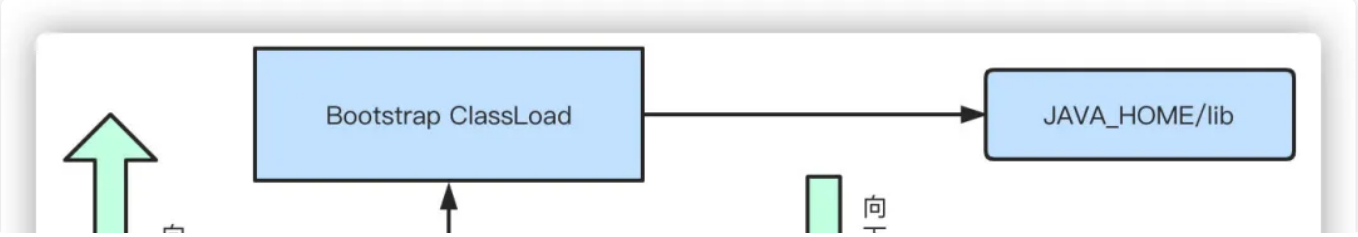
1. 首先为对象分配合适大小的内存空间
2. 接着为实例变量赋默认值
3. 设置对象的头信息，对象hash码、GC分代年龄、元数据信息等
4. 执行构造函数(init)初始化

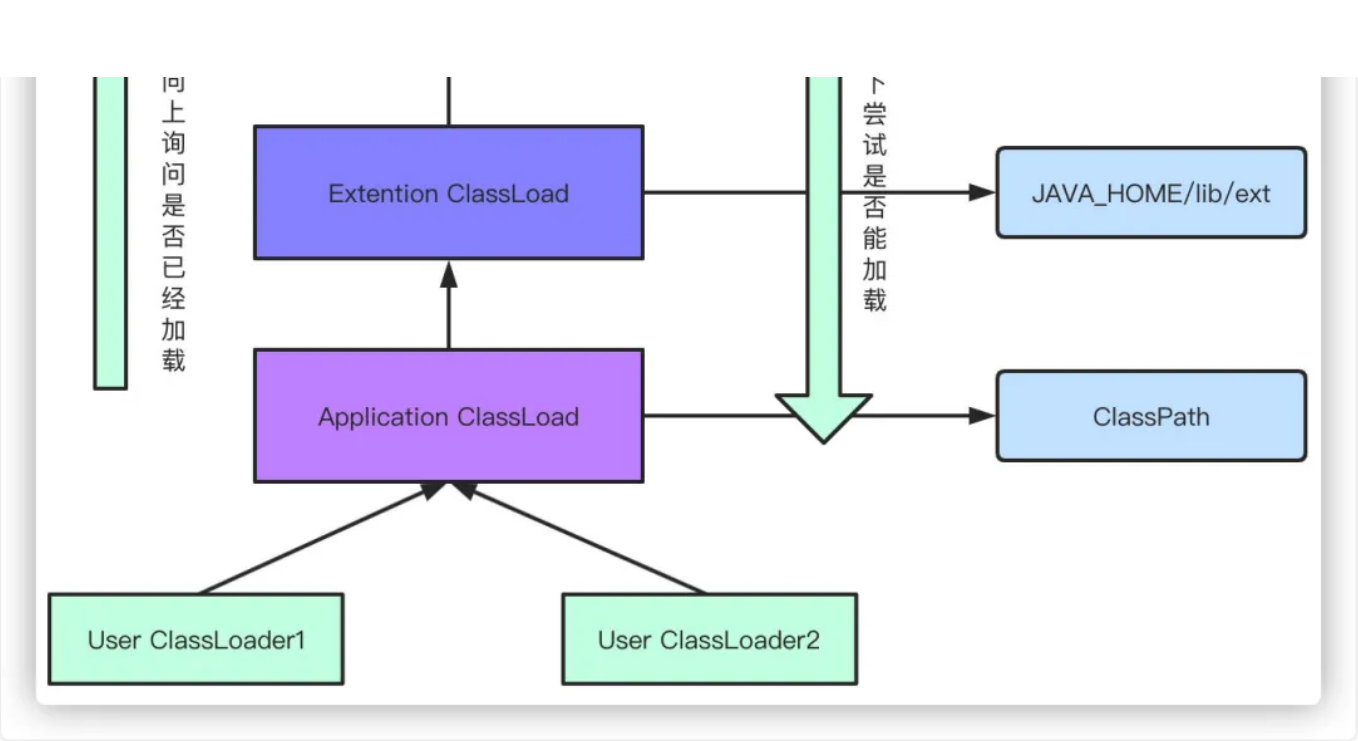
知道双亲委派模型吗?

类加载器自顶向下分为：

1. Bootstrap ClassLoader启动类加载器：默认会去加载JAVA_HOME/lib目录下的jar
2. Extention ClassLoader扩展类加载器：默认去加载JAVA_HOME/lib/ext目录下的jar
3. Application ClassLoader应用程序类加载器：比如我们的web应用，会加载web程序中ClassPath下的类
4. User ClassLoader用户自定义类加载器：由用户自己定义

当我们在加载类的时候，首先都会向上询问自己的父加载器是否已经加载，如果没有则依次向上询问，如果没有加载，则从上到下依次尝试是否能加载当前类，直到加载成功。





说说有哪些垃圾回收算法?

标记-清除

统一标记出需要回收的对象，标记完成之后统一回收所有被标记的对象，而由于标记的过程需要遍历所有的GC ROOT，清除的过程也要遍历堆中所有的对象，所以标记-清除算法的效率低下，同时也带来了内存碎片的问题。

复制算法

为了解决性能的问题，复制算法应运而生，它将内存分为大小相等的两块区域，每次使用其中的一块，当一块内存使用完之后，将还存活的对象拷贝到另外一块内存区域中，然后把当前内存清空，这样性能和内存碎片的问题得以解决。但是同时带来了另外一个问题，可使用的内存空间缩小了一半!

因此，诞生了我们现在的常见的年轻代+老年代的内存结构：Eden+S0+S1组成，因为根据IBM的研究显示，98%的对象都是朝生夕死，所以实际上存活的对象并不是很多，完全不需要用到一半内存浪费，所以默认的比例是8:1:1。

这样，在使用的时候只使用Eden区和S0S1中的一个，每次都把存活的对象拷贝另外一个未使用的Survivor区，同时清空Eden和使用的Survivor，这样下来内存的浪费就只有10%了。

如果最后未使用的Survivor放不下存活的对象，这些对象就进入Old老年代了。

PS：所以有一些初级点的问题会问你为什么要分为Eden区和2个Survior区？有什么作用？就是为了节省内存和解决内存碎片的问题，这些算法都是为了解决问题而产生的，如果理解原因你就不需要死记硬背了

标记-整理

针对老年代再用复制算法显然不合适，因为进入老年代的对象都存活率比较高了，这时候再频繁的复制对性能影响就比较大，而且也不会再有另外的空间进行兜底。所以针对老年代的特点，通过标记-整理算法，标记出所有的存活对象，让所有存活的对象都向一端移动，然后清理掉边界以外的内存空间。

那么什么是GC ROOT？有哪些GC ROOT？

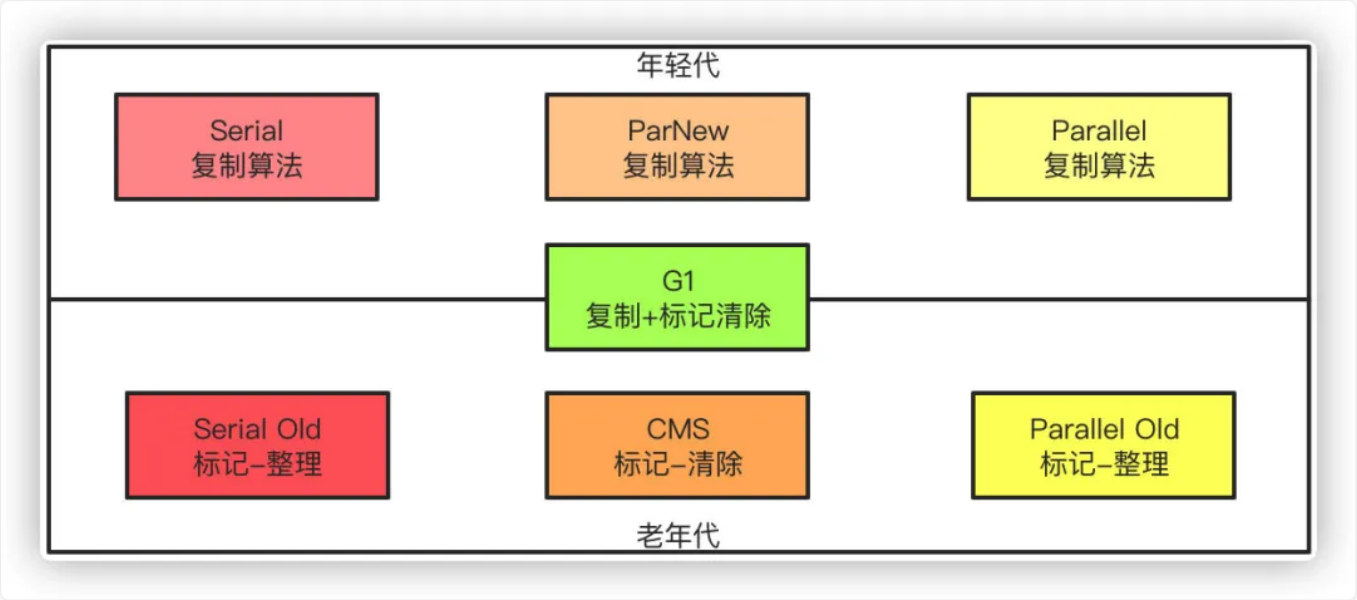
上面提到的标记的算法，怎么标记一个对象是否存活？简单的通过引用计数法，给对象设置一个引用计数器，每当有一个地方引用他，就给计数器+1，反之则计数器-1，但是这个简单的算法无法解决循环引用的问题。

Java通过可达性分析算法来达到标记存活对象的目的，定义一系列的GC ROOT为起点，从起点开始向下开始搜索，搜索走过的路径称为引用链，当一个对象到GC ROOT没有任何引用链相连的话，则对象可以判定是可以被回收的。

而可以作为GC ROOT的对象包括：

1. 栈中引用的对象
2. 静态变量、常量引用的对象
3. 本地方法栈native方法引用的对象

垃圾回收器了解吗？年轻代和老年代都有哪些垃圾回收器？



年轻代的垃圾收集器包含有Serial、ParNew、Parallell，老年代则包括Serial Old老年代版本、CMS、Parallel Old老年代版本和JDK11中的船新的G1收集器。

Serial：单线程版本收集器，进行垃圾回收的时候会STW（Stop The World），也就是进行垃圾回收的时候其他的工作线程都必须暂停

ParNew：Serial的多线程版本，用于和CMS配合使用

Parallel Scavenge：可以并行收集的多线程垃圾收集器

Serial Old：Serial的老年代版本，也是单线程

Parallel Old：Parallel Scavenge的老年代版本

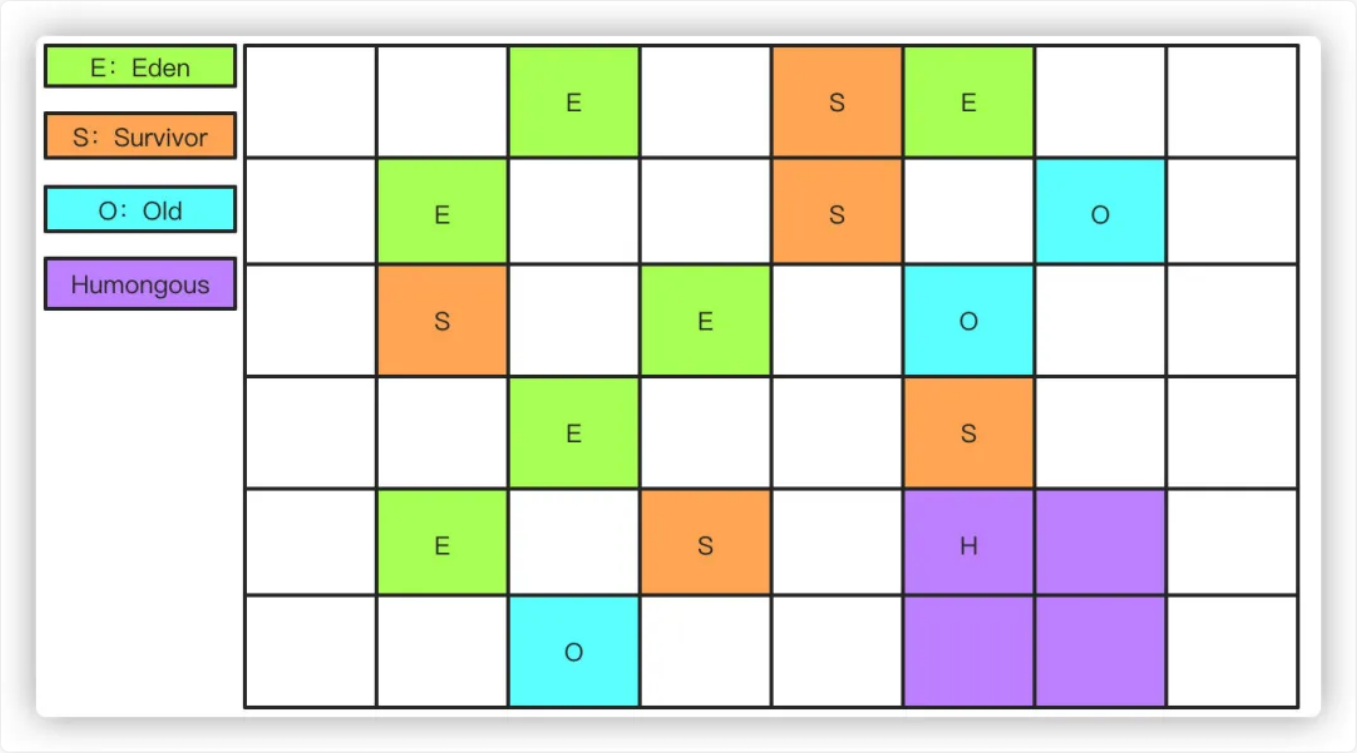
CMS（Concurrent Mark Sweep）：CMS收集器是以获取最短停顿时间为目标的收集器，相对于其他的收集器STW的时间更短暂，可以并行收集是他的特点，同时他基于标记-清除算法，整个GC的过程分为4步。

1. 初始标记：标记GC ROOT能关联到的对象，需要STW
2. 并发标记：从GCRoots的直接关联对象开始遍历整个对象图的过程，不需要STW
3. 重新标记：为了修正并发标记期间，因用户程序继续运作而导致标记产生改变的标记，需要STW
4. 并发清除：清理删除掉标记阶段判断的已经死亡的对象，不需要STW

从整个过程来看，并发标记和并发清除的耗时最长，但是不需要停止用户线程，而初始标记和重新标记的耗时较短，但是需要停止用户线程，总体而言，整个过程造成的停顿时间较短，大部分时候是可以和用户线程一起工作的。

G1（Garbage First）：G1收集器是JDK9的默认垃圾收集器，而且不再区分年轻代和老年代进行回收。

G1的原理了解吗?



G1作为JDK9之后的服务端默认收集器，且不再区分年轻代和老年代进行垃圾回收，他把内存划分为多个Region，每个Region的大小可以通过-XX：G1HeapRegionSize设置，大小为1~32M，对于大对象的存储则衍生出Humongous的概念，超过Region大小一半的对象会被认为是大对象，而超过整个Region大小的对象被认为是超级大对象，将会被存储在连续的N个Humongous Region中，G1在进行回收的时候会在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间优先回收收益最大的Region。

G1的回收过程分为以下四个步骤：

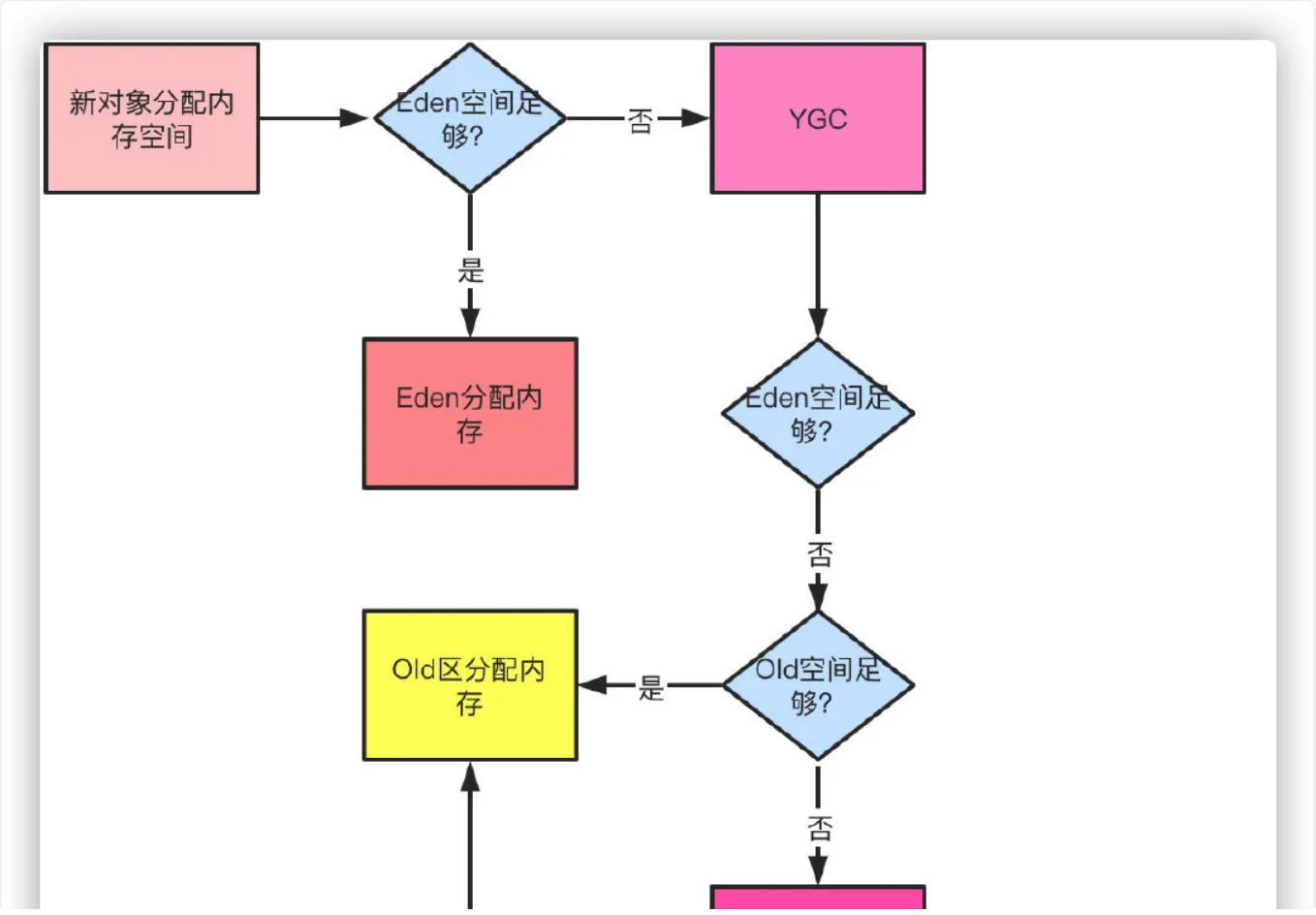
1. 初始标记：标记GC ROOT能关联到的对象，需要STW

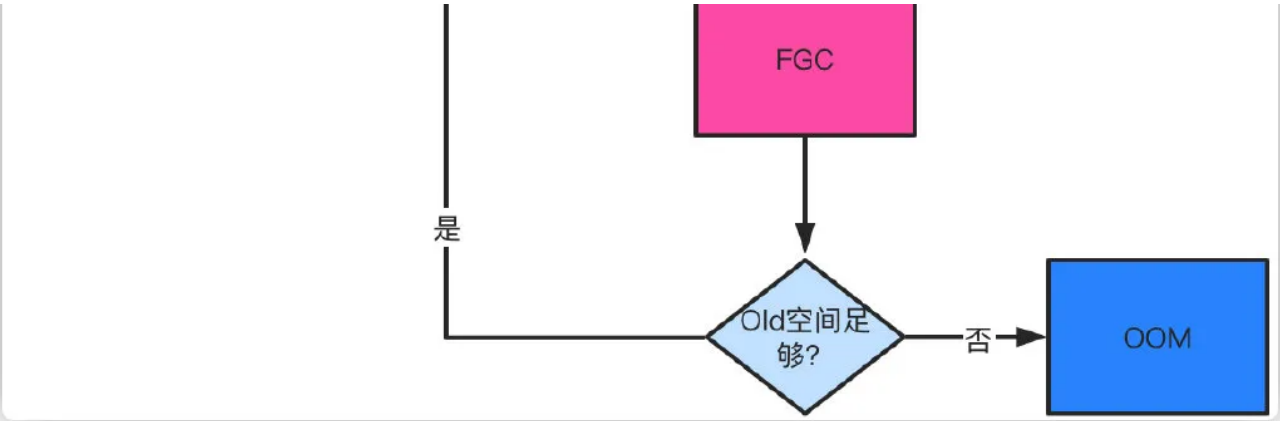
2. 并发标记：从GCRoots的直接关联对象开始遍历整个对象图的过程，扫描完成后还会重新处理并发标记过程中产生变动的对象
3. 最终标记：短暂暂停用户线程，再处理一次，需要STW
4. 筛选回收：更新Region的统计数据，对每个Region的回收价值和成本排序，根据用户设置的停顿时间制定回收计划。再把需要回收的Region中存活对象复制到空的Region，同时清理旧的Region。需要STW

总的来说除了并发标记之外，其他几个过程也还是需要短暂的STW，G1的目标是在停顿和延迟可控的情况下尽可能提高吞吐量。

什么时候会触发YGC和FGC？对象什么时候会进入老年代？

当一个新的对象来申请内存空间的时候，如果Eden区无法满足内存分配需求，则触发YGC，使用中的Survivor区和Eden区存活对象送到未使用的Survivor区，如果YGC之后还是没有足够空间，则直接进入老年代分配，如果老年代也无法分配空间，触发FGC，FGC之后还是放不下则报出OOM异常。





YGC之后，存活的对象将会被复制到未使用的Survivor区，如果S区放不下，则直接晋升至老年代。而对于那些一直在Survivor区来回复制的对象，通过-XX：MaxTenuringThreshold配置交换阈值，默认15次，如果超过次数同样进入老年代。

此外，还有一种动态年龄的判断机制，不需要等到MaxTenuringThreshold就能晋升老年代。如果在Survivor空间中相同年龄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就可以直接进入老年代。

频繁FullGC怎么排查?

这种问题最好的办法就是结合有具体的例子举例分析，如果没有就说一般的分析步骤。发生FGC有可能是内存分配不合理，比如Eden区太小，导致对象频繁进入老年代，这时候通过启动参数配置就能看出来，另外有可能就是存在内存泄露，可以通过以下的步骤进行排查：

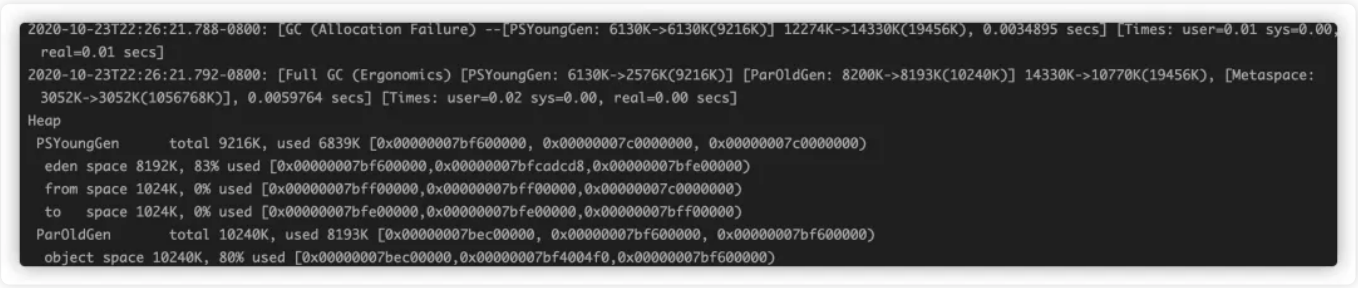
1. jstat -gcutil或者查看gc.log日志，查看内存回收情况

```
→ jstat -gcutil 40893 500 10
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  78.00   9.10  74.81  77.70     54    0.088     0    0.000    0.088
  6.25   0.00  36.00   9.10  74.81  77.70     56    0.090     0    0.000    0.090
  0.00   0.00  82.00   9.10  74.81  77.70     57    0.091     0    0.000    0.091
  0.00   6.25  52.00   9.10  74.81  77.70     59    0.094     0    0.000    0.094
  0.00   6.25  12.00   9.10  74.81  77.70     61    0.096     0    0.000    0.096
  0.00   0.00   0.00   9.10  74.81  77.70     63    0.099     0    0.000    0.099
  0.00   6.25   2.00   9.10  74.81  77.70     65    0.102     0    0.000    0.102
  0.00   6.25   0.00   9.10  74.81  77.70     67    0.105     0    0.000    0.105
  6.25   0.00  96.01   9.10  74.81  77.70     68    0.106     0    0.000    0.106
  6.25   0.00  74.00   9.10  74.81  77.70     70    0.107     0    0.000    0.107
```

S0 S1 分别代表两个Survivor区占比

E代表Eden区占比，图中可以看到使用78%

O代表老年代，M代表元空间，YGC发生54次，YGCT代表YGC累计耗时，GCT代表GC累计耗时。



[GC [FGC 开头代表垃圾回收的类型

PSYoungGen: 6130K->6130K(9216K)] 12274K->14330K(19456K), 0.0034895 secs代表YGC前后内存使用情况

Times: user=0.02 sys=0.00, real=0.00 secs，user表示用户态消耗的CPU时间，sys表示内核态消耗的CPU时间，real表示各种墙时钟的等待时间

这两张图只是举例并没有关联关系，比如你从图里面看能到是否进行FGC，FGC的时间花费多长，GC后老年代，年轻代内存是否有减少，得到一些初步的情况来做出判断。

2. dump出内存文件在具体分析，比如通过jmap命令jmap -dump:format=b,file=dumpfile pid，导出之后再通过**Eclipse Memory Analyzer**等工具进行分析，定位到代码，修复

这里还会可能存在一个提问的点，比如CPU飙高，同时FGC怎么办？办法比较类似

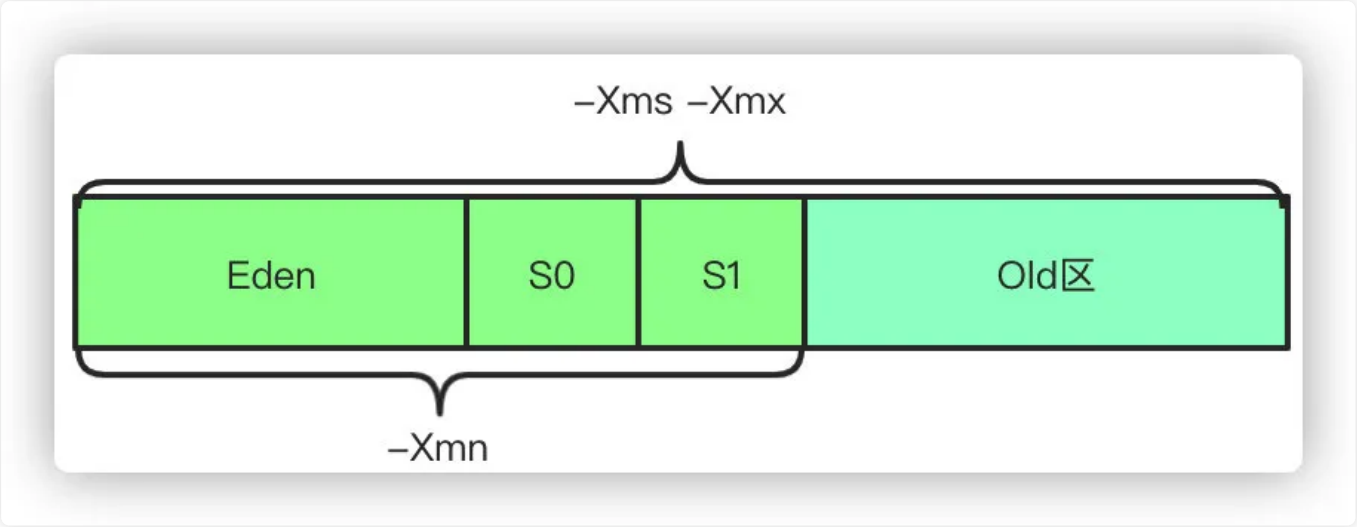
1. 找到当前进程的pid，top -p pid -H 查看资源占用，找到线程
2. printf “%x\n” pid，把线程pid转为16进制，比如0x32d
3. jstack pid|grep -A 10 0x32d查看线程的堆栈日志，还找不到问题继续
4. dump出内存文件用MAT等工具进行分析，定位到代码，修复

JVM调优有什么经验吗?

要明白一点，所有的调优的目的都是为了用更小的硬件成本达到更高的吞吐，JVM的调优也是一样，通过对垃圾收集器和内存分配的调优达到性能的最佳。

简单的参数含义

首先，需要知道几个主要的参数含义。



1. -Xms设置初始堆的大小，-Xmx设置最大堆的大小
2. -XX:NewSize年轻代大小，-XX:MaxNewSize年轻代最大值，-Xmn则是相当于同时配置-XX:NewSize和-XX:MaxNewSize为一样的值
3. -XX:NewRatio设置年轻代和年老代的比值，如果为3，表示年轻代与老年代比值为1:3，默认值为2
4. -XX:SurvivorRatio年轻代和两个Survivor的比值，默认8，代表比值为8:1:1
5. -XX:PretenureSizeThreshold 当创建的对象超过指定大小时，直接把对象分配在老年代。
6. -XX:MaxTenuringThreshold设定对象在Survivor复制的最大年龄阈值，超过阈值转移到老年代
7. -XX:MaxDirectMemorySize当Direct ByteBuffer分配的堆外内存到达指定大小后，即触发Full GC

调优

1. 为了打印日志方便排查问题最好开启GC日志，开启GC日志对性能影响微乎其微，但是能帮助我们快速排查定位问题。-XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:gc.log
2. 一般设置-Xms=-Xmx，这样可以获得固定大小的堆内存，减少GC的次数和耗时，可以使得堆相对稳定

3. -XX:+HeapDumpOnOutOfMemoryError让JVM在发生内存溢出的时候自动生成内存快照，方便排查问题
4. -Xmn设置新生代的大小，太小会增加YGC，太大会减小老年代大小，一般设置为整个堆的1/4到1/3
5. 设置-XX:+DisableExplicitGC禁止系统System.gc()，防止手动误触发FGC造成问题

- END -

往期推荐

《我想进大厂》之mysql夺命连环13问

《我想进大厂》之Redis夺命连环11问

面试官：哪些场景会产生OOM？怎么解决？

《我想进大厂》之MQ夺命连环11问

点击二维码识别关注

点在看，让更多看见。

收录于话题 #面试大全·29个

上一篇

来自朋友最近阿里、腾讯、美团等P7岗位面试题

下一篇

面试官：说说CountDownLatch，CyclicBarrier，Semaphore的原理?

大厂经典面试题：Redis为什么这么快?

捡田螺的小男孩

步步深入：MySQL 架构总览->查询执行流程->SQL 解析顺序

ImportNew

这篇零拷贝，牛了牛了

程序员cxuan