JVM相关问题

-----GC

1. 什么是垃圾

没有任何引用指向的一个对象或者多个对象（指循环引用）

1. 定位垃圾

引用计数 ---循环引用存在问题

根可达算法 --根包括 虚拟机栈中的变量，本地方法栈中的变量，静态变量，常量池，加载的类信息

1. 垃圾回收算法

标记清除 –容易产生碎片

拷贝算法 –没有碎片，效率高，但是空间利用率低（一半浪费）

标记压缩 –没有碎片，需要移动内存，效率低

1. 分代垃圾回收算法

模型：新生代+老年代+永久代（1.7）/元数据(1.8)

元数据区主要放Class

永久代在堆内，必须指定大小，元数据在堆外，可不指定最大大小，受限于物理内存显示

字符串常量1.7放在永久代，1.8放在堆

新生代年龄进入老年代 G1 15 CMS 6

老年代满🡪FULLGC

５.　常见垃圾回收器

１.Serial年轻代 Serial老年代

2. PS年轻代 ParNew年轻代（配合CMS老年代）

3.ParallelOld老年代

4． CMS老年代 （低延迟）

5.G1 ZGC

1.8默认 PS+ParallelOld(高吞吐量)

6. JVM参数

-开头 所有HotSpot都支持

-X开头 特定版本支持

-XX 开头 下个版本可能取消

-XX:+PrintCommandLineFlags 打印设置的JVM参数

-XX:+PrintFlagsFinal 打印最终参数值

-------CMS回收器详解

1. 主要阶段

**初始标记**

标记GC Roots可达的老年代对象

遍历新生代对象，标记可达的老年代对象

STW 但是速度快

**并发标记**

继续递归标记这些对象可达的对象

应用正常执行

如果应用执行过程中，老年代新产生对象等，标记为dirty，待重新标记时处理

**重新标记**

老年代的新对象被GC Roots引用

老年代的未标记对象被新生代对象引用

老年代已标记的对象增加新引用指向老年代其它对象

新生代对象指向老年代引用被删除

STW

**并发整理**

回收老年代，标记清除算法

1. 优点

速度快，STW时间少

缺点

CPU敏感（多线程）

不压缩，会有碎片

整理阶段不STW，这段时间如果产生新的垃圾，下一次才会被回收。如果此时OLD区不够用，触发Serial olggc，会长时间STW。 所以不能等到old区100%才gc，需要设立阈值。Jdk1.8默认92%，推荐70%左右。

-------G1回收器详解

Jdk1.9开始默认

模型

将堆分为等大的区域（Eden,Suvivor,Old,Humongous）

默认分成2048个区，也可以手动指定大小 –XX:G1HeapRegionSize

如果一个对象大小超过一个区的一般，则分配到H区



重要机制

Card Table 和Remember Set

每个区分成多个card,一般512byte

当某个Card里面的对象引用了其他区的对象(内存空间发生改变)，则这个Card标记为dirty，被引用对象的RS里面记录下这个card的信息，方便向上查找

RS的更新不是实时的

在赋值操作时，会有一个write barrier，将更新rs的任务放到一个队列里面，  
队列有白绿黄红四个状态，表示积压情况



Gc种类

YGC

MixedGC

FullGC

Young GC 阶段：

STW

阶段1：根扫描

静态和本地对象被扫描

阶段2：更新RS

处理dirty card队列更新RS

阶段3：处理RS

检测从年轻代指向年老代的对象

阶段4：对象拷贝

拷贝存活的对象到survivor/old区域

阶段5：处理引用队列（对象地址变化了，引用值也要跟着变）

软引用，弱引用，虚引用处理

拷贝算法

G1会记录每个阶段的耗时情况，由于调优

 -XX:MaxGCPauseMillis 设置每次gc的期望耗时

如果超过期望，会减少每次gc的region数量（增加gc频率，需要合理设置，否则影响吞吐量）

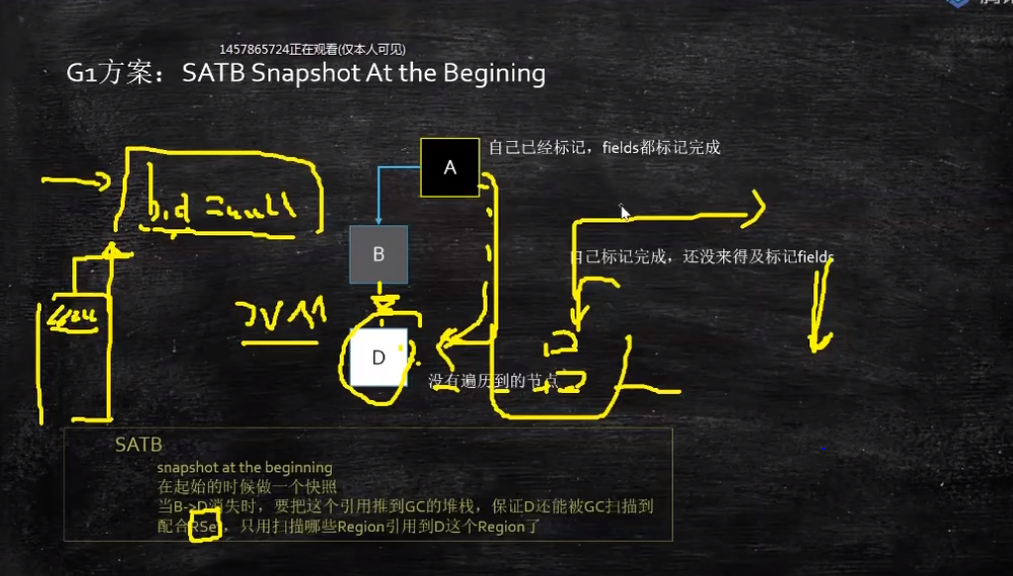
Mixed GC

当老年代使用量达到一定比例（-XX:InitiatingHeapOccupancyPercent=N，默认45%）

Mixed GC 选定所有年轻代里的 Region，外加根据 global concurrent marking 统计得出收集收益高的若干老年代 Region 进行回收。（gabage first）

STW,进行YoungGC (同时进行初始标记)

恢复应用线程，并发标记（三色标记算法），并发标记时，如果一个对象引用消失，会将这个对象放到堆栈里，然后再重新标记的时候，便利堆栈的对象，通过RS找还有没有其他引用，杜绝漏标现象



STW，重新标记

立即清空全部是垃圾的区域

部分为空的，拷贝的新的region，回收原来的(STW)（优先选择垃圾较多的region）

恢复应用线程

Full GC：

收集整个堆。

触发时机：对象内存分配速度过快，Mixed GC 来不及回收，老年代被填满。

这时候会切换到 G1 之外的 Serial Old GC 来收集整个 GC heap（注意，包括 young、old、perm），这才是真正的 Full GC。

缺点

Rs占用1~20%的空间，依然有STW的阶段，依然无法避免FullGC

-------ZGC回收器详解

JDK11引入

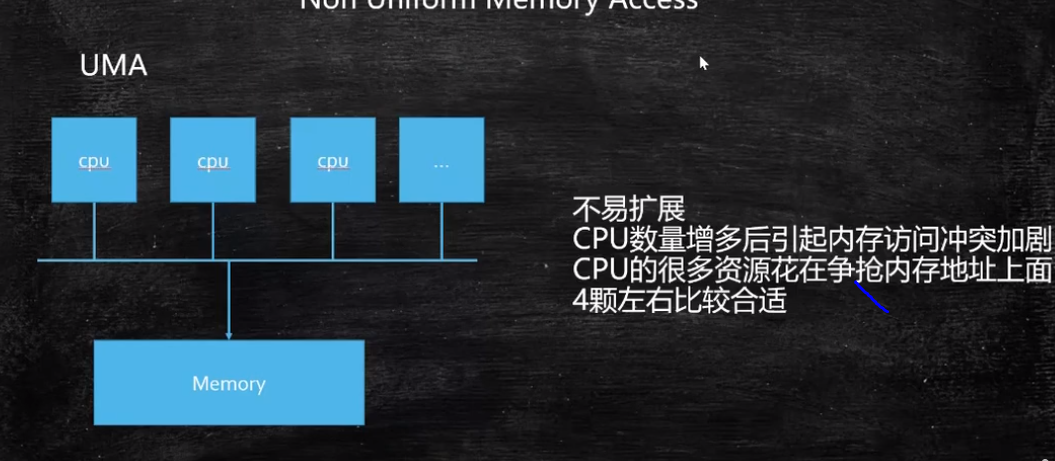
支持4TB的堆

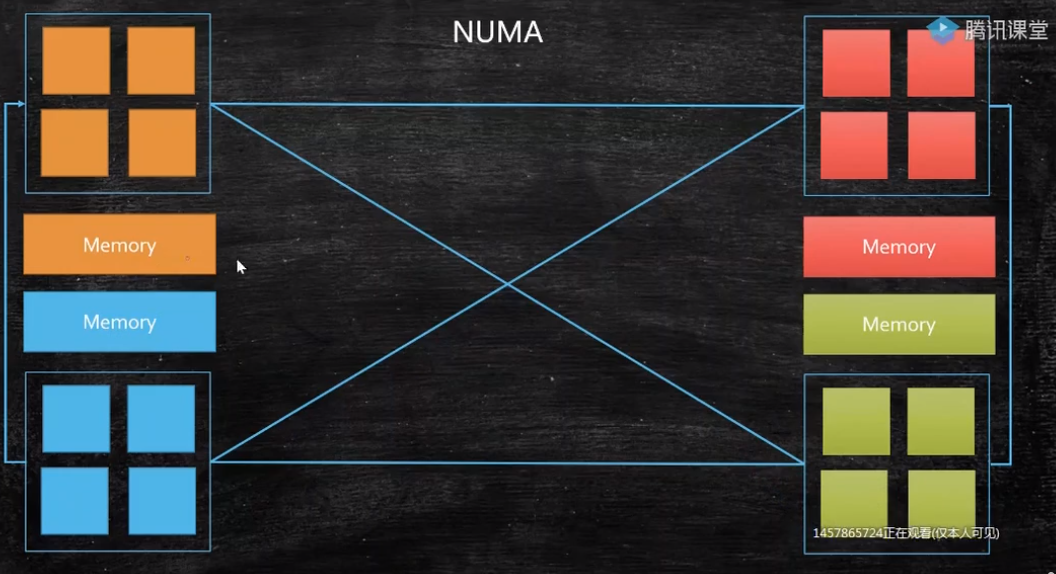
10ms延迟 吞吐量最多降低15% 延迟不会随着堆的大小而增加

能利用NUMA

*UMA 多个cpu用同一块内存，存在大量竞争*

*NUMA 内存分块，不同的块分给不同的CPU*





目前不区分老年代年轻代，将来可能分冷热对象

重点算法---颜色指针

以前的gc算法，gc信息记录在对象上

ZGC直接记录在指向对象的指针上，因此只要对象被回收，可以立即恢复内存区域

低42为指向对象，43~46为颜色指针，高18为为unused

42位地址空间，所以最大4T JDK13中，扩大为44位，最大支持16T

目前最大16T， 因为主板目前地址总线最宽48bit,4位被颜色指针占用



核心思想

原来的gc都是写屏障，也就是再对象赋值的时候，记录相关信息，比如地址变动

ZGC是读屏障，在访问一个对象的时候，如果发现指针上，颜色标记的位置是被标记，移动中，回收中等，就会自旋等待，等地址挪动到位之后，在访问该对象（4%左右性能损耗）

**类加载**

加载

类加载过程的一个阶段，ClassLoader通过类的全限定名查找Class字节码文件，并利用字节码文件创建一个class对象。

验证

目的在于确保class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身的安全，主要包括四种验证：文件格式的验证，元数据的验证，字节码验证，符号引用验证。

准备

为静态变量分配初始值，这里不包含final修饰的static ，因为final在编译的时候就已经分配了。这里不会为实例变量分配初始化，类变量会分配在方法区中，实例变量会随着对象分配到Java堆中。

解析

这里主要的任务是把常量池中的符号引用替换成直接引用

初始化

这里是类记载的最后阶段，如果该类具有父类就进行对父类进行初始化，执行其静态初始化器（静态代码块）和静态初始化成员变量。（前面已经对static 初始化了默认值，这里我们对它进行赋值，成员变量也将被初始化）

**forName和loaderClass区别**

Class.forName()得到的class是已经初始化完成的。

Classloader.loaderClass得到的class是还没有链接（验证，准备，解析三个过程被称为链接）的。

**双亲委派模型**

工作原理：

如果一个类收到了类加载的请求，它并不会自己先去加载，而是把这个请求委托给父类加载器去执行，如果父类加载器还存在父类加载器，则进一步向上委托，依次递归，请求最后到达顶层的启动类加载器，如果完成类的加载任务，就会成功返回，倘若父类加载器无法完成任务，子类加载器才会尝试自己去加载，这就是双亲委派模式。

优势：

1. 有优先层级关系，避免类的重复加载
2. 更加安全，很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心api，会带来相关隐患。。

判断类是否已被加载：findLoadedClass

JVM调优举例

典型案例1

线上某服务，时不时会突然失去响应一段时间。

观察jvm性能监控

老年代大小经常出现突然增大的尖刺现象，部分时候还会导致fullgc，推测时fullgc导致服务失去响应

找准拉dump时机。可能1-2天才发生一次，要趁老年代升高的时候拉

拉dump方式：公司组件，dump之前会拉出机器。 Dump命令jmap –dump

分析工具

Jvisual mat IBM工具

找到数量最多，占用空间最大的对象，List Map 都是酒店房型的数据对象----定位到代码

对应一个数据导出功能，供应商导出酒店房型数据，可能有几十上百万，没有分批。

优化方案：分批，1000左右一批。 后续该功能迁移到job上处理，不放在接口服务上

其他案例：

Metaspace引发fullgc codecache空间不足

常用命令

Jstat 查看jvm内存信息 gc信息

Jmap dump

Jstack 查看线程堆栈信息