垃圾回收机制

如何判定对象为垃圾对象

1. 引用计数法  
   在对象当中添加一个引用计数器\*1,
2. 可达性分析法\*2

2如何回收垃圾对象

1. 回收策略  
   1标记清除算法\*3  
   2复制算法\*4  
   3 标记拷贝算法  
   4标记整理算法  
   5分带收集算法
2. 垃圾回收器  
   1 Serial  
   2 Parnew  
   3 Cms  
   4 G1

3何时回收来及

\*1 引用计数器 当有地方引用对象时,引用计数器就+1,当引用失效的时候,计数器的值就-1,但已经不再使用,堆中如果出现相互引,对象虽然已失去使用意义,但引用计数器仍不为0🡪-verbose:gc打印垃圾回收日志

\*2 可达性分析法 定义垃圾回收的根节点,虚拟机栈,本地方法栈,方法区常量,方法区类属性中没有引用可以指向的堆中的对象

\*3 标记清除算法 通过垃圾判定,标记所需清除的算法🡪标记已在判断垃圾对象时执行🡪标记清除的两大问题🡪1执行效率低2内存碎片:被清除的空间虽然释放,但是空间却零碎不连续,无法储存较大的数据

\*4 复制算法 堆被分为新生代,老年代🡪新声代🡪Eden伊甸园,Survivor存活区

老年代🡪Tenured Gen

每次垃圾回收结束后,都会把剩余的对象复制到另一块内存当中,并清空原来的内存,按顺序排好,这样就解决了内存碎片问题

这里把内存分为四块,新生代🡪Eden区最大,新创建的对象都会进入这里,第二块,第三块都叫Survivor(分为from, to区)🡪具体看第二章的\*6

🡪老年代Tenures Gen

内存分区\*3

1 线程共享区:🡪多个线程共享同一个区域

1. 方法区🡪里面有: 1运行时常量池,2已被加载的类元信息,3常量,4静态变量,5编译后代码数据

类信息:🡪类的版本,字段,方法

常量池:🡪当粗出数据相同时会相互覆盖,类似于HashTable;

1. java堆🡪堆内存\*6: 对象实例\*4🡪垃圾回收的主要区域,需要学习的词汇🡪新生代,老年代,Eden空间

创建对象\*5

给对象分配内存:🡪解决线程安全问题,1加线程锁,2每个线程划分一个区域,只在该区域内分配线程🡪就是空间换时间

初始化对象:给空间分配默认值

执行初始化方法,调用init

2 线程独享区:🡪每一个线程有一个独立的内存

1. 虚拟机栈🡪存放方法运行时所需数据,栈帧🡪中的(局部变量表)就是(栈内存)

栈针\*1:每一个方法执行都会创建一个栈针直至方法结束,用于存储🡪局部变量表,操作数栈,动态链接,方法出口等

局部变量表\*2:存放编译期可知的各种基本数据类型,引用类型,retrunAddress类型大小

操作数栈 也是一个栈,用于临时保存用于赋值和操作计算的数字

动态链接 🡪用于存储执行方法,通过方法名生成子密码🡪用于保存找到方法区方法的内存地址,这个地址就是动态链接

1. 本地方法栈🡪jvm调用的native方法,即本地方法服务
2. 程序计数器🡪记录当前线程所执行到的字节码行号

该区域占内存很少

每一个线程都有一个独立的程序计数器用来计数代码

执行native,计数器的值变为undefined

该区域没有规定内存溢出情况

\*1栈针 储存在栈中,每执行一个方法就在栈内存入一个栈针,一个方法中调用另一个方法就在存入另一个栈针,没执行完一个方法后进行出栈操作,递归可以造成栈溢出

\*2 局部变量表 在编译期记录🡪在他里面还存栈针实际所需大小,一个栈针存的引用类型实际上只是一个引用地址,引用的对象实际存放在堆中,所以可以在编译期就能知道并记录栈针的大小

\*3 直接内存 并不在 jvm内存当中,但nio时会用到,这里提一下

\*4 对象的实例化 根据new的参数定位到常量池的引用🡪如果常量池找不到该类符号引用,就说明没有被加载,则进行加载

\*5 对象 包含三部分Header,InstanceData,Padding

Header对象头🡪储存自身运行时的数据(Mark Wor):哈希值hashCode()方法,GC分代年龄,锁状态标志,线程持有锁,偏向线程功能id,偏向时间戳

类型指针,对象指向对象指向类数据元数据的指针

InstaneData对象数据🡪真正存储数据区域,虚拟机分配内存策略,相同长度字段会被分配到一起,long和doubles,shorts和char

Padding 堆区填充

对象的访问定位🡪使用句柄定位🡪句柄的优点,内存整理时就不用修改栈中的引用地址了

直接指针定位🡪优点的效率高

\*6 堆 区域划分\*7🡪

占1/3🡪年轻代:

占8/10🡪Eden区🡪当伊甸园区满后,便会运行执行引擎释放无效内存\*8🡪当经历过一次 minor gc后仍未别回收的对象就会被放入Survivor区,分代年龄就会+1🡪分代年龄储存在对象头里面

占2/10🡪Survivor区🡪该区域被放满后也会触发 minor gc ,并从from区移动到to区🡪清空from区,也就是所谓的复制算法执行过程🡪to区满后,又会回收并执行复制算法移动到from区

当一个对象执行15次gc仍未被销毁,就会转移到老年代

占2/3🡪老年代 :当老年代也满时就会触发full gc对老年代进行垃圾回收🡪full GC执行时程序会停止运行,所以Full GC调优是优化一大手段

\*7 jvm虚拟机 除了运行数据区,还有类装载子系统,执行引擎

\*8 gc root根🡪暂时理解为,栈中变量表,当引用变量所在的栈针出栈后🡪堆中的对象就变为无效内存

\*8标记拷贝算法，将被标记的垃圾回收，并且将后面的有效内容拷贝到回收的空间，相当于边整理边回收，效率低的原因，因为在进行内存挪动的时候必须要线程同步，同步失效率变低。

静态变量储存的只是一个指针,对象内容还是在堆中

电脑硬件分层🡪cpu其中包含cpu寄存器(cpu缓存)—可以储存数据,比内存快当更贵

内存RAM

Java内存模型🡪简写jmm🡪java内存模型就和计算机分层有类似之处,每一个线程在开启后,共享线程里的数据会被拷贝到🡪工作内存当中🡪因此定义一个变量后,其中一个线程改变共享内存的值可能并不会影响另一个值,为了使多线程可见可以在变量上加 volatile修饰词🡪volatile早期实现原理为总线加锁,线程共享区的内存一旦被访问就会被锁住,只有其操作完毕后才会释放锁🡪性能低,多线程被迫变为单线程

🡪当今解决方案已更换为MESI缓存一致性协议\*1🡪

定义了volatile关键字的变量🡪汇编语言底层会在变量前加上lock前缀🡪变量在被某个线程修改的时候🡪立刻同步到主内存,并对内存加锁直到赋值完成🡪赋值占用的时间很小几乎不影响效率,其他的线程会通过cpu总线嗅探机制(监听)监测到自身所需变量的值发生改变,同时触发总线中的MESI协议🡪使被修改的值在工作内存当中失效

\*1 计算机硬件交互最终通过🡪总线\*2 来完成🡪当交换时就会使用MESI缓存一致性协议\*1🡪当数据一旦通过总线🡪cpu嗅探机制—监听机制🡪监听数据的变动🡪其他线程的相同数据就会改为失效🡪当计算机重新读取该部分的内存,内容至为空,计算机就会重新从主内存中获取该部分的值

\*2 总线 负责计算机个中硬件之间数据交换的中转站

并发编程:

三大特性:

1. 可见性:
   1. volatile可以保证并发编程的可见性,a线程对b变量的操作,其他线程都能看到,并接受改变🡪但不保证原子性
2. 原子性
3. 有序性

jvm模型：

类加载子系统

运行时数据区

执行引擎

JVM内存模型分为新生代，老年代，新版本已经没有永久代了变为元空间，永久代必须设置大小限制，元空间可以不设置大小

字符串常量1.7存在永久代，1.8存在堆中

新生代的回收称之为YGC young回收，在YGC中大多数对象都会被回收