JVM

Java内存结构

堆和方法区是共有的，其他三个是私有的

1. 堆内存

存放对象实例，垃圾收集器管理的主要区域

- 假设main()方法创建了对象A，在main方法对应的栈帧的局部变量表里，让一个引用的变量（对象名）来存放堆内存的对象的地址
- 新建的实例在堆内存，实例变量也是在堆内存

2. 方法区

- 老年代和元空间
  - 方法区（method area）只是JVM规范中定义的一个概念，具体实现由Java的虚拟机决定
  - JDK1.8之前，使用永久代来存储，永久代是一段连续的内存空间，我们在 JVM 启动之前可以通过设置-XX:MaxPermSize的值来控制永久代的大小
  - JDK1.8之后使用元空间来是实现，元空间并不在虚拟机中，而是使用本地内存。
  - 运行时常量池在jdk7之前存放在永久代中，jdk7之后从方法区移出来了，在堆中开辟了一块区域存放运行时常量池
- 放置从.class中加载进来的类，还有常量池
- 常量+静态变量+类元信息
  - 1.已被虚拟机加载的类信息
  - 2.静态变量
  - 3.即时编译器编译后的代码
  - 4.常量池
    - 字面量：字面量比较接近java语言层面的常量概念。如文本字符串,声明为final的常量值等
    - 符号引用
      - 类和接口的全限定名
      - 字段的名称和描述符
      - 方法的名称和描述符
- 方法区的类什么时候会被回收

- 首先该类的所有实例对象都已经从Java堆内存里被回收
- 其次加载这个类的ClassLoader已经被回收
- 最后，对该类的Class对象没有任何引用

3. Java虚拟机栈

每个线程独享，每个方法执行的时候都会创建一个栈帧

- 局部变量表
  - 基本数据类型
  - 对象引用类型
- 操作数栈
  - 当一个方法刚刚开始执行时，其操作数栈是空的
  - 随着方法执行和字节码指令的执行，会从局部变量表或对象实例的字段中复制常量或变量写入到操作数栈，再随着计算的进行将栈中元素出栈到局部变量表或者返回给方法调用者，也就是出栈/入栈操作。
- 动态链接
- 方法出口
  - 放回到上级调用方法的具体位置

> 对于JVM执行引擎来说，在在活动线程中，只有位于JVM虚拟机栈栈顶的元素才是有效的，即称为当前栈帧，与这个栈帧相关连的方法称为当前方法，定义这个方法的类叫做当前类。

4. 本地方法栈

- 虚拟机栈为虚拟机执行java方法服务
- 本地方法为虚拟机执行native方法服务.
- 调用本地操作系统里面的一些方法，可能是C语言写的方法，也可能是一些底层类库。存放各种native方法的局部变量表

5. 程序计数器

- 当前线程所执行的字节码的行号指示器。程序计数器是一块内存区域，用来记录线程当前**要执行的指令地址**。
- CPU时间片用完之后，等下次轮到自己的时候再执行。那么如何知道之前程序执行到哪里了呢?
  - 程序计数器就是为了记录该线程让出CPU时的执行地址的，待再次分配到时间片时线程就可以从自己私有的计数器指定地址继续执行。

- 如果执行的是**native方法，那么pc计数器记录的是undefined地址**，只有执行的是Java代码时pc计数器记录的才是下一条指令的地址。

JDK1.8对内存区域的修改

- 堆内存、栈内存、永久代
- 永久代里放一些常量池、
- 常量池放到堆内存中
- 类信息放到元区域

流程

首先，你的JVM进程会启动，就会先加载你的Kafka类到内存里。然后有一个main线程，开始执行你的Kafka中的main()方法。

main线程是关联了一个程序计数器的，那么他执行到哪一行指令，就会记录在这里

大家结合上图中的程序计数器来理解一下。

其次，就是main线程在执行main()方法的时候，会在main线程关联的Java虚拟机栈里，压入一个main()方法的栈帧。

接着会发现需要创建一个ReplicaManager类的实例对象，此时会加载ReplicaManager类到内存里来。

然后会创建一个ReplicaManager的对象实例分配在Java堆内存里，并且在main()方法的栈帧里的局部变量表引入一个“replicaManager”变量，让他引用ReplicaManager对象在Java堆内存中的地址。

看到这里，大家结合上面的两个图理解一下。

接着，main线程开始执行ReplicaManager对象中的方法，会依次把自己执行到的方法对应的栈帧压入自己的Java虚拟机栈

类加载过程

jar包里的类不是一次性全部加载的，是使用到时才加载（加载->连接(验证,准备,解析)->初始化->使用->卸载

1、加载

- 1.通过IO读入字节码文件到内存中，使用到类时才会加载
- 2.静态存储结构转化为方法区的运行时数据结构
- 3.在java堆中生成一个类对象,作为方法区的访问入口.

2、验证

- 校验字节码文件的正确性
- 1.文件格式验证。为了保证输入的字节流能正确地解析并存储于方法区之内
- 2.元数据验证。进行语义分析,保证描述地信息符合java语言规范(父类继承等)
- 3.字节码验证。最复杂地阶段(指令验证)
- 4.符号引用验证。通过符号引用是否能找到字段,方法,类等

3、准备

- 给类的静态变量**分配内存**，并赋予默认值
- int等变量为0、对象为null、final直接赋值
- 为**类的静态变量分配内存**并设置类变量地初始化阶段
- 如static int n = 2；初始值是0，因为这个时候还没有执行任何Java方法(执行<clinit>之后才产生.)
- static final int n = 2；对应到常量池,在准备阶段被赋值为2.

4、解析

- 将**符号引用替换为直接引用**，该阶段会把一些静态方法(符号引用，比如main()方法)替换为**指向数据所存内存的指针或句柄**等(直接引用)，这是所谓的**静态链接**过程(类加载期间完成)
- 符号引用
  - 用符号来描述所引用的目标
- 直接引用
  - 指向目标的指针或者偏移量
- 动态链接是在程序运行期间完成的将符号引用替换为直接引用

5、初始化

- 对类的静态变量初始化为指定的值，执行静态代码块
- 执行<clinit>()方法，静态变量,静态块的初始化,(类和接口的初始化)，如果没有静态变量和静态块,则没有<clinit>()

- 如果一个类的父类还没有初始化，那么会先初始化他的父类

6、使用

7、卸载

> 类初始化不等同于类初始化

补充类初始化的时机如下：

1.当创建某个类的新实例时（如通过new或者反射，克隆，反序列化等）

2.当调用某个类的静态方法时

3.当使用某个类或接口的静态字段时

4.调用Java API中的某些反射方法时，比如类Class中的方法，或者java.lang.reflect中的类的方法时

5.当初始化某个子类时

6.当虚拟机启动某个被标明为启动类的类（即包含main方法的那个类） 所以System.ou.println(Test.class)不满足上面6种情况，也就没有做初始化

类加载器（双亲加载模型）

把Class字节码文件加载到内存中

加载器

1. 启动类加载器BootstrapClassLoader：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
2. 扩展类加载器ExtClassLoader：负责加载支撑JVM运行的位于JRE的lib\ext扩展目录中的JAR类包
3. 应用类加载器AppClassLoader：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类

4. 自定义加载器：负责加载用户自定义路径下的类包

双亲委派模型

- 加载某个类时会先不断委托父加载器寻找目标类，，如果所有父加载器**在自己的加载类路径下**都找不到目标类，则在自己的类加载路径中查找并载入目标类
- 检查顺序是自底向上,加载顺序是自顶向下的,
- 作用
  - 沙箱**安全机制**：自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意篡改
  - 避免**类的重复加载**：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次，保证被加载类的唯一性

Java引用类型

- 强引用
  - 有明显的new Object()的引用,只要这种引用还在,GC就不会回收它,哪怕 内存不够(会抛出OutOfMemory)
- 软引用
  - 有用但是非必需的引用(类似于缓存),内存不够时才回收
- 弱引用
  - 非必需对象,生存到下一次GC之前
- 虚引用
  - 唯一目的是在对象被收集器回收时收到一个系统通知或者后续添加进一步的处理.
  - 相当于没有引用,get方法总返回null

Java对象的布局

在JVM中，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充

- 对象头(64位系统里的对象头为16字节)
  - Mark Word
    - 用于存储自身的运行时数据
    - 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等，占用内存大小与虚拟机位长一致。
    - 在64位虚拟机下，Mark Word是64bit大小的，32位的是32bit
  - 类型指针（klass pointer）

        - 这一部分用于存储对象的类型指针，该指针指向它的类元数据，JVM通过这个指针确定对象是哪个类的实例。
        - 该指针的位长度为JVM的一个字大小，即32位的JVM为32位，64位的JVM为64位。
- 实例数据
    - 类中定义的成员变量
- 对齐填充
    - 对齐填充并不是必然存在的，也没有什么特别的意义，他仅仅起着占位符的作用，
    - 由于HotSpot VM的自动内存管理系统要求对象起始地址必须是8字节的整数倍
    - 换句话说，就是对象的大小必须是8字节的整数倍。

对象在内存空间中的占用情况

- 1、对象自己本身的信息（如64位系统里的对象头为16字节）
- 2、对象的实例变量占用的空间（如int4字节）

堆内存中的布局

- JDK1.7
    - Yound(新生代)：Eden、from survior、to survivor
    - Tenured(老年代)
    - Perm(永久区)
    - virtual区
- JDK1.8
    - Yound
    - Tenured
    - metaSpace(取代永久代)

JVM垃圾分析算法/GCRoots

内存中不再使用到的对象就是垃圾

1. 判断对象是否可回收的算法
    a. 引用计数算法（基本不使用
    - 对对象添加一个引用计数器,每当进行一次引用时或者 引用失效时,计数器进行加或减,表示这个对象不会再被引用了
    - 缺陷：难以解决循环引用问题

    b. 可达性分析算法（jvm使用

- 对每个对象，分析一下有谁在引用他，然后一层一层往上判断，看是否有一个是GCRoots
- 过一系列“GC Roots”的对象作为起始点,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。

2. GC Roots（一些比较活跃的引用的集合）
   a. 虚拟机栈(栈帧中的本地变量表)中引用的对象
   b. 方法区中静态属性引用的对象
   c. 方法区中常量引用的对象
   d. 本地方法栈中JNI(即一般所说的Native方法)引用的对象
3. finialize()方法

一个对象要被回收了，如果它重写了Object类的finialize（）方法，会尝试调用一下，看是否把自己的实例变量赋值给某个GCRoots

4. Minor GC/Young GC

- 新生代内存不够的时候（eden区存活对象比to Survivor大）
- minorGC之前要检查老年代是否够大（先判断老年代是否比整个eden大，（handlePromotionFailure设置的时候）判断是否比往常存活对象大）

5. Full GC

- 1、在Minor GC之前，检查发现MinorGC之后要进入老年代的对象太多，老年代放不下
- 2、Minor GC之后，剩余对象太多老年代放不下

6. 年轻代对象进入老年代

- 15次Young GC还存活的进入老年代
- Young GC之后当前survivor区域里一批对象的总大小大于该区域的50%，那么此时大于等于这批对象年龄的对象可以直接进入老年代（年龄a+年龄b+年龄c+...>50%，从a开始的都放入老年代）
- 超大对象直接放入老年代，XX:PretenureSizeThreshold设置为字节数
- Eden区的对象超过to Survivor，直接转移到老年代中

> Q：为什么老年代的full GC要比新生代的Minor GC慢很多倍（一般十倍以上

新生代执行速度其实很快，因为直接从GC Roots出发就追踪哪些对象是活的就行了，新生代存活对象是很少的，这个速度是极快的，不需要追踪多少对象。

然后直接把存活对象放入Survivor中，就一次性直接回收Eden和之前使用的Survivor了。

但是CMS的Full GC呢?

在并发标记阶段，他需要去追踪所有存活对象，老年代存活对象很多，这个过程就会很慢；

其次并发清理阶段，他不是一次性回收一大片内存，而是找到零零散散在各个地方的垃圾对象，速度也很慢；

最后完事儿了，还得执行一次内存碎片整理，把大量的存活对象给挪在一起，空出来连续内存空间，这个过程还得“Stop the World”，那就更慢了。

万一并发清理期间，剩余内存空间不足以存放要进入老年代的对象了，引发了“Concurrent Mode Failure”问题，那更是麻烦，还得立马用“Serial Old”垃圾回收器，“Stop the World”之后慢慢重新来一遍回收的过程，这更是耗时了。

所以综上所述，老年代的垃圾回收，就是一个字：慢

垃圾收集算法

1. 标记-清除算法
- 分为标记需要回收的对象和清除两个阶段
- 内存空间耗尽时，jvm会停下应用程序的运行并开启GC线程， 然后开始进行标记工作
- 优点
  - 解决了循环引用的问题
- 不足
  - 1.效率问题,标记和清楚都需要遍历所有的对象，并且在GC时停止应用程序，效率不高(暂停是为了在遍历时不会产生新的对象引用)
  - 2.空间问题,产生空间碎片
2. 复制算法
- 内存分成两块,每次使用其中一块,当这一块用完之后将还存活的对象 赋值到另外一块,然后把已使用的内存空间一次清除.
- 标记出from空间中标记出哪些对象是不能进行垃圾回收的
- 优点:
  - 垃圾对象较多时效率较高
  - 清除后,内存无碎片
- 缺点
  - 内存利用率低,垃圾对象少的时候不适用.
3. 标记-整理算法
- 标记过程与标记-清除一样,但后续时**让所有存活的对象向一端移动, 然后直接清理掉端边界以外的内存.(**因为多了移动内存位置的步骤,效率有一定影响)
4. 分代收集算法
- 根据回收对象的特点进行选择

> 在jvm中，年轻代适合使用复 制算法，老年代适合使用标记清除或标记压缩算法。

垃圾收集器

- 查看默认垃圾收集器
  - -XX:PrintCommandLineFlags
  - JDK8是ParralleGC
- 新生代/老年代的垃圾收集器
  - 新生代：Serial,Parallel,ParNew

    - 老年代：SerialOld, Parallel Old, CMS
    - 全部：G1（JDK9默认）
- 搭配
    - Serial+SerialOld
    - ParNew+Serial Old
    - Parallel+Parallel Old(java8默认)
    - CMS+ParNew(Serial备用)
- 串行垃圾收集器(Serial)
    - Serial、Serial Old
    - 使用单线程进行垃圾回收，垃圾回收时，只有一个线程在工作， 并且java应用中的所有线程都要暂停,STW(stop the world)
    - 交互性较强的应用中不能使用,如Web
- 并行垃圾收集器(Parrall)
    - 大吞吐量并行垃圾收集器在收集的过程中也会暂停应用程序，这个和串行垃圾回收器是**一样的**，只是并行执行，速度更快些，暂停的时间更短一些。
    - ParNew收集器（-XX:+UseParNewGC
        - 工作在新生代,只是将串行的垃圾收集器改为了并行。
    - ParallelGC收集器（吞吐量优先收集器）
        - 与ParNew相比:**由自适应调节策略:收集性能监控信息,** 动态地调整这些参数以提供最合适地停顿时间或最大吞吐量
    - ParallelOld
- CMS(并发)垃圾收集器
    - 并发标记清除-垃圾收集器,已获取最短回收停顿时间为目标的垃圾收集器，标记-清除算法
        - 所以需要通过参数开启每次清除之后stw整理碎片；或者开启执行多少次GC之后再一次整理碎片
    - 默认启动的线程个数是（CPU核数+3）/4
    - 1.初始化标记
        - 标记一下GC Root可以直接关联的对象,速度很快,需要暂停所有线程
    - 2.并发标记
        - 和用户进程一起运行(不需要暂停),主要标记过程,标记全部对象.
    - 3.重新标记
        - 修正并发标记期间发生的变动;需要暂停
    - 4.并发清理

        - 清除不可达对象,不需要暂停;(由于时间最长的并发标记和并发清除不用暂停,因此看起来是一起并发地执行的)
- **G1垃圾收集器**
    - jdk9中使用G1垃圾收集器作为默认收集器.高吞吐量的同时停顿时间更短.
    - G1可以同时回收新生代和老年代对象，不需要两个垃圾回收器配合使用
    - **主要关注点在于达到可控的停顿时间，在这个基础上尽可能提高吞吐量。通过划分为大量小region，追踪region中可以回收的对象大小和预估时间，把影响控制在指定时间范围内。**
    - 与CMS比较
        - G1 被设计用来长期取代 CMS 收集器，和 CMS 相同的地方在于，它们都属于并发收集器，在大部分的收集阶段都不需要挂起应用程序。区别在于
            - G1 **没有 CMS 的碎片化问题（或者说不那么严重）**标记-整理算法
            - 停顿时间可控：添加了预测时间,可以指定期望停顿时间（-**XX:MaxGCPauseMillis=100**）
            - 并行与并发：G1能更充分的利用CPU，多核环境下的硬件优势来缩短stop the world的停顿时间。
    - 空间分配
        - 其他收集器中年轻代和老年代是各自独立且连续的区域块。
        - 而 G1 将整个堆划分为一个个大小相等的小块（每一块称为一个 region），每一块的内存是连续的。和分代算法一样，G1 中每个块也会充当 Eden、Survivor、Old 三种角色，但是它们不是固定的，这使得内存使用更加地灵活。
        - region(1M-32M,最多2048个分区)。取消了新生代和年老代的**物理划分,将堆划分为逻辑**上的年轻代,老年代区域(不再是物理隔离的，混杂在一起。
        - Eden,Survivor,Old,Humongous(存放巨型对象)
        - 怎么设定region大小
            - 给堆设置一个大小，jvm自动用堆除以2048
            - 刚开始默认新生代为5%，运行中不断增加，但是不会超过60%
        - 大对象
            - 提供了专门的region来存放大对象
            - 超过一个redion的50%即是大对象
            - 不管什么GC都会顺带着大对象一起回收
    - 停顿时间
        - -**XX:MaxGCPauseMillis=100**
        - G1 使用了**停顿预测模型**来满足用户指定的停顿时间目标，并基于目标来选择进行垃圾回收的区块数量。G1 采用增量回收的方式，每次**回收一些区块**，而不是整堆回收。

        - Full GC 的时候还是单线程运行的，所以我们应该尽量避免发生 Full GC
    - 使用
        - 开启G1垃圾收集器
        - 设置堆的最大内存
        - 设置最大的停顿时间
    - 收集过程
        - 1、Young GC（复制算法）
        - 2、MixedGC
        - *、必要时的 Full GC（复制时候发现没有空闲的region可以回收了）
    - 三种模式:YoungGC,MinedGC,FullGC
    - YoungGC
        - 新生代达到了堆内存的60%，而Eden还占满了对象，就会触发新生代的GC（该过程根据设定的停顿时间来执行）
        - 主要对Eden区进行GC,在Eden区空间耗尽时会被触发
        - RSet(已记忆集合)
    - **MixedGC**
        - 回收整个Young,一部分老年区
        - 默认老年区占堆的**45%**时启动
        - GC步骤1、全局并发标记
            - 初始标记
            - 并发标记
            - 最终标记
            - 筛选回收
        - 复制算法，一次混合回收可以反复回收多次
    - 优化建议
        - 不设置年轻代大小,固定的年轻代大小会覆盖暂停时间目标
        - 暂停时间设置不要太苛刻,G1吞吐量的目标时90%的应用程序和10%的垃圾回收时间

调优及参数配置

- 运行参数
    - -参数：标配参数
    - -X参数:非标准参数

- -XX参数:使用率较高(主要是用于jvm调优和debug操作)
- -Xms:jvm堆内存的初始大小
- -Xmx:jvm堆内存的最大大小

- 内存分配中常用的参数
  - Xms和Xmx通常一样，PermSize和MaxPermSize也一样

在JVM内存分配中，有几个参数是比较核心的，如下所示。

1. **-Xms**：Java堆内存的大小
2. **-Xmx**：Java堆内存的最大大小
3. **-Xmn**：Java堆内存中的新生代大小，扣除新生代剩下的就是老年代的内存大小了
4. **-XX:PermSize**：永久代大小
5. **-XX:MaxPermSize**：永久代最大大小
6. **-Xss**：每个线程的栈内存大小

  - 1.8之后，PermSize改为了MetaSpaceSize

- 查看JVM系统默认值
  - jps查看进程信息
  - jinfo-flag 配置项 进程号:查看该配置是否允许
  - jinfo-flag 进程号:查看所有配置
  - -XX:+PrintFlagsInitial:查看初始默认
  - -XX:+printFlagsFinal:查看修改更新
  - -XX:+printCommandLineFlags:查看默认垃圾回收器
- 常用配置

- -Xms:初始大小的内存,默认为物理内存1/16
- 等价于-XX:InitialHeapSize
- -Xmx:最大分配内存,默认物理内存1/4
- -Xss:设置单个线程栈的大小,一般默认是512K~1024K
- -Xmm:年轻代的大小
- -XX:MetaspaceSize:设置元空间大小
- -XX:+PrintGCDetails
- -XX:SurvivorRatio=8:Eden和survivor比例,8:1:1
- -XX:NewRatio=2:老年代和年轻代的比例2:1
- -XX:MaxTenuringThreshold=15:对象在交换15次之后放到老年代(在0-15之间)

- OOM
  - StackOverflowError
    - 占空间溢出,递归调用卡死
  - OutofMemoryError:Javaheap space
    - 堆内存溢出,对象过大.
  - OutofMemoryError:GCoverhead limit exceeded
    - 回收时间过长,每次回收的对象太少
  - OOM:Directbuffer memory
    - metaspace内存用完,不归GC管
  - OOM:unableto create new native thread
    - 1.应用创建太多线程（降低线程数量）
    - 2服务器不允许创建这么多线程,Linux最多1024（修改服务器配置,扩大允许数量）
  - OOM：MetaSpace
- 常用参数
  - Xmx\Xms\Xss(单线程大小)\Xmm(年轻代大小)
  - PringFlagInitial/PrintFlagFinal:查看初始配置/最后配置
  - -XX:MetaspaceSize
  - SurvivorRatio/NewRatio