链滴

JVM 初级面试题

作者：xiaokedamowang
原文链接：https://ld246.com/article/1629673888928
来源网站：链滴

JVM初级面试题

一. JVM的构成

- 类加载器
- 运行时数据区(内存)
    - 程序计数器

记录下一条指令的地址, 简单来说就是线程执行到了哪, CPU时间片切换回来程序需要知道从哪开始执行
 在物理上是通过寄存器实现

    - 方法区

方法区是一种规范, 规定了存储类相关的信息, 永久代和元空间是他的实现

JDK8之前是在堆内存中分一块区域(**逻辑**)叫做永久代, 用来存放运行时常量池和类信息

JDK8的时候把永久代修改成了元空间, 在本地内存(操作系统内存)中开辟一块区域, 不再使用堆内存的部分, 运行时常量池仍然放在堆中

什么是常量池, 什么是运行时常量池?

常量池就是一张常量表, 存在class文件中, 通过常量符号可以找到对应的方法名, 参数类型, 字面量等息

运行时常量池就是在运行中, class中的常量池信息会加载到内存中

    - 堆

线程共享的内存区域, 存放对象

    - 虚拟机栈(线程栈)

每个线程默认1M的栈内存用来存储栈帧

    - 本地方法栈
- 执行引擎
    - 解释器

解释class文件, 翻译成机器码

    - 即时编译器

会对热点代码优化

    - GC

垃圾收集器

- 本地方法接口

二. 类的加载

1. 加载

加载进内存, 生成对应的C++结构instanceKlass, 生成代表这个类的Class对象, 作为对外暴露类的入口

2. 连接

    1. 验证

验证格式, 安全性检查

    2. 准备

给静态变量分配内存, 设置默认值, 如果是**final**的基本类型, 或者字符串, 赋值在准备阶段完成

    3. 解析

将常量池中的符号引用解析成直接引用, 就是把原本class中存储的常量符号, 比如方法名, 字面量等信息 解析成真正的内存地址

3. 初始化

执行**类**的构造方法, 就是静态代码块

初始化的情况

main方法的类总是被首先初始化

首次访问静态变量或静态方法

子类初始化时, 父类还没初始化, 会跟着初始化

Class.forName()

new对象

不会初始化的情况

访问**final**的静态常量, 基本类型或者字符串

类.class

创建类的数组的时候

手动使用加载器加载类的时候, 比如: classLoad.loadClass("xxx.xxx.xxx"), 只会加载类, 不会初始化

Class.forName()第二个参数为false的时候

三. 双亲委派

类加载器分四种

1. BootstrapClassLoader启动类加载器, C++实现, 无法访问, 加载jre/lib下的类
2. ExtensionClassLoader扩展类加载器, 上级为BootstrapClassLoader, 加载jre/lib/ext下的类
3. ApplicationClassLoader应用程序加载器, 上级为ExtensionClassLoader, 加载classpath下的类

4. 自定义加载器, 继承抽象类ClassLoader, 重写findClass方法, 上级为ApplicationClassLoader

双亲委派指的是类加载器加载1个类的时候, 会优先从上级获取, 如果没有才自己加载

四. 引用的类型

一般来说有四种, 但还听说过第五种

1. 强引用

普通对象的引用, 平时我们一般都使用这个

2. 软引用

当GC发生的时候, 并且内存不够, 此时会把软引用的对象释放, 可以配合引用队列使用

3. 弱引用

当GC发生的时候, 不管内存够不够, 都会释放该引用的对象, 可以配合引用队列使用

4. 虚引用

虚引用主要是为了跟踪垃圾回收过程, **必须**配合引用队列使用, 在对象被垃圾回收的时候, 会将这个引加入队列, 在虚引用出队列前, 不会彻底销毁该对象, 一般用来释放堆外内存(直接内存)

5. 终结器引用

虚拟机会给我们的对象创建终结器引用, 当对象被回收时, 会把终结器引用放入引用队列, 等待finalizeHnder线程调用finalize方法

五. 垃圾回收算法

先了解1下如果怎么确定一个对象是否是垃圾: 引用计数, 可达性分析(根可达), java使用后者, 听说最早本的python是使用引用计数

1. 标记清除, 优点: 速度快, 缺点: 内存碎片
2. 标记整理, 优点: 没有内存碎片, 缺点: 速度慢
3. 复制, 优点: 没有内存碎片, 缺点: 浪费一部分空间

六. 垃圾收集器

1. 新生代
    1. Serial, 单线程, 使用复制算法
    2. ParNew, 多线程并行, 使用复制算法
    3. Parallel, 多线程并行, 吞吐量优先
2. 老年代
    1. SerialOld, 单线程, 使用标记清除整理算法
    2. ParallelOld, 多线程并行, 吞吐量优先, 使用标记清除整理算法
    3. CMS, 多线程并行+并发, 响应时间优先, 使用标记清除清除算法

CMS垃圾回收的4个步骤

1. 初始标记(不能并发)
2. 并发标记(可以并发)
3. 重新标记(不能并发)
4. 并发清除(可以并发)

由于使用标记清除算法, 所以无法整理内存碎片, 当内存无法满足程序需求的时候, 会启动SerialOld垃圾收集器, 导致一次非常慢的FullGC, 可以修改配置使FullGC的时候采用MSC算法压缩堆内存, 但是使用SC算法合并整理内存的时候, 不能并发

3. 整堆

G1, 多线程并行+并发, 同时注重吞吐量和响应时间, 将整个堆分成多个Region区域(**512K**), 整体上是记整理算法, 两个区域之间是复制算法

G1垃圾收集器的三种情况(网上资料计较混乱, 无法验证真伪)

1. 新生代垃圾收集
2. 混合垃圾收集(老年代内存占用达到45%, 可以通过参数设置)
3. FullGC

整个过程: 初始标记(不能并发), 并发标记(三色标记), 最终标记(不能并发), 垃圾清除

七. MinorGC/YoungGC和FullGC

MinorGC/YoungGC是清理新生代的垃圾收集

FullGC是清理老年代的垃圾回收, JDK8之前永久代内存不够也会触发FullGC

八. 新生代老年代

1. 新生代

默认占用堆内存的三分之一

- 伊甸区, 默认新生代的十分之八, 新建的对象放在伊甸区, 每次发生YongGC时, 会把不需要清除对象放入幸存区from, 并且年龄加1
- 幸存区from, 默认新生代的十分之一, 每次发生YongGC时, 会把幸存区from的对象复制到幸存区o, 然后幸存区to和幸存区from身份交换, 幸存的对象年龄加1, 当达到16岁(默认, 可以配置)时, 扔到老代
- 幸存区to, 默认新生代的十分之一

当创建的对象特别大, 伊甸区内存不够时, 或者幸存的对象超过幸存区50%大小的时候, 直接扔到老年代

2. 老年代

老年代里的对象一般来说是存活比较久的, 当老年代满了的时候会发生FullGC

九. happen-before

1. **程序次序规则**: 在一个线程内哪怕指令重排序, 但代码产生的结果要保证和不重排序一致
2. **管程锁定规则**: 解锁之前的改动要对获取锁的线程可见
3. **volatile变量规则**: 对volatile变量写的操作一定对读的操作可见
4. **线程启动规则**: 在线程启动之前的操作, 一定要对被启动的线程可见
5. **线程终止规则**: 线程结束的时候, 确保结束之前的操作对其他线程可见
6. **线程中断规则**: 线程中断的时候, 确保中断之前的操作对该线程可见
7. **传递规则**: A线程可见B线程的操作, B线程可见C线程的操作, 那么要确保A线程可见C线程的操作
8. **对象终结规则**: 确保对象的构造方法的操作对finalize方法可见

十. GC调优

其实这个没啥好说的, 根据不同的情况再调整参数, 此处针对Web场景补充1点点

一般我们Web项目的对象都是垃圾, 都是数据库查出来, 返回给前端, 然后就可以回收了, 所以对应的应把新生代调大一些, 最好伊甸区大于: 预估并发量 * 请求平均消耗的内存