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

```html
<h2 id="JVM初级面试题">JVM 初级面试题</h2>
<h3 id="一--JVM的构成">一. JVM 的构成</h3>
<ul>
<li>
<p>类加载器</p>
</li>
<li>
<p>运行时数据区(内存)</p>
<ul>
<li>
<p>程序计数器</p>
<blockquote>
<p>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始执行, 在物理上是通过寄存器实现</p>
</blockquote>
</li>
<li>
<p>方法区</p>
<blockquote>
<p>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</p>
<p>JDK8 之前是在堆内存中分一块区域(<strong>逻辑</strong>)叫做永久代, 用来存放运行时常池和类信息</p>
<p>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, 不再使用堆存的一部分, 运行时常量池仍然放在堆中</p>
</blockquote>
<p><strong>什么是常量池, 什么是运行时常量池?</strong></p>
<blockquote>
<p>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, 字面等信息</p>
<p>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</p>
</blockquote>
</li>
<li>
<p>堆</p>
<blockquote>
<p>线程共享的内存区域, 存放对象</p>
</blockquote>
</li>
<li>
<p>虚拟机栈(线程栈)</p>
<blockquote>
<p>每个线程默认 1M 的栈内存用来存储栈帧</p>
</blockquote>
</li>
<li>
<p>本地方法栈</p>
</li>
</ul>
</li>
<li>
<p>执行引擎</p>
<ul>
<li>
<p>解释器</p>
```

<blockquote>
解释 class 文件, 翻译成机器码
</blockquote>

即时编译器
<blockquote>
会对热点代码优化
</blockquote>

GC
<blockquote>
垃圾收集器
</blockquote>

本地方法接口

<h3 id="二--类的加载">二. 类的加载</h3>

加载
<blockquote>
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, 作为对外暴露的入口
</blockquote>

连接

验证
<blockquote>
验证格式, 安全性检查
</blockquote>

准备
<blockquote>
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, 值在准备阶段完成
</blockquote>

解析
<blockquote>
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, 字面等信息, 解析成真正的内存地址
</blockquote>


```
</ol>
</li>
<li>
<p>初始化</p>
<blockquote>
<p>执行<strong>类</strong>的构造方法, 就是静态代码块</p>
</blockquote>
<h5 id="初始化的情况">初始化的情况</h5>
<blockquote>
<p>main 方法的类总是被首先初始化</p>
<p>首次访问静态变量或静态方法</p>
<p>子类初始化时, 父类还没初始化, 会跟着初始化</p>
<p>Class.forName()</p>
<p>new 对象</p>
</blockquote>
<h5 id="不会初始化的情况">不会初始化的情况</h5>
<blockquote>
<p>访问 <strong>final</strong> 的静态常量, 基本类型或者字符串</p>
<p>类.class</p>
<p>创建类的数组的时候</p>
<p>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
</p>
<p>Class.forName()第二个参数为 false 的时候</p>
</blockquote>
</li>
</ol>
<h3 id="三--双亲委派">三. 双亲委派</h3>
<blockquote>
<p>类加载器分四种</p>
<ol>
<li>BootstrapClassLoader 启动类加载器, C++ 实现, 无法访问, 加载 jre/lib 下的类</li>
<li>ExtensionClassLoader 扩展类加载器, 上级为 BootstrapClassLoader, 加载 jre/lib/ext 下的类</l
>
<li>ApplicationClassLoader 应用程序加载器, 上级为 ExtensionClassLoader, 加载 classpath 下的
</li>
<li>自定义加载器, 继承抽象类 ClassLoader, 重写 findClass 方法, 上级为 ApplicationClassLoader</
i>
</ol>
</blockquote>
<p>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</p>
<h3 id="四--引用的类型">四. 引用的类型</h3>
<blockquote>
<p>一般来说有四种, 但还听说过第五种</p>
</blockquote>
<ol>
<li>
<p>强引用</p>
<blockquote>
<p>普通对象的引用, 平时我们一般都使用这个</p>
</blockquote>
</li>
<li>
<p>软引用</p>
<blockquote>
```

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
</blockquote>

弱引用
<blockquote>
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
</blockquote>

虚引用
<blockquote>
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, 会将这个引用加入队列, 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)
</blockquote>

终结器引用
<blockquote>
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 finalizeHander 线程调用 finalize 方法
</blockquote>

<h3 id="五--垃圾回收算法">五. 垃圾回收算法</h3>
<blockquote>
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, 听最早版本的 python 是使用引用计数
</blockquote>

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

<h3 id="六--垃圾收集器">六. 垃圾收集器</h3>

新生代
<blockquote>

Serial, 单线程, 使用复制算法
ParNew, 多线程并行, 使用复制算法
Parallel, 多线程并行, 吞吐量优先

</blockquote>

老年代
<blockquote>

SerialOld, 单线程, 使用标记清除整理算法

ParallelOld, 多线程并行, 吞吐量优先, 使用标记清除整理算法

CMS, 多线程并行 + 并发, 响应时间优先, 使用标记清除清除算法
<blockquote>
CMS 垃圾回收的 4 个步骤

初始标记(不能并发)
并发标记(可以并发)
重新标记(不能并发)
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 SerialOl 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, 是使用 MSC 算法合并整理内存的时候, 不能并发
</blockquote>

</blockquote>

整堆
<blockquote>
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(12K), 整体上是标记整理算法, 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)

新生代垃圾收集
混合垃圾收集(老年代内存占用达到 45%, 可以通过参数设置)
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
</blockquote>

<h3 id="七--MinorGC-YoungGC和FullGC">七. MinorGC/YoungGC 和 FullGC</h3>
<blockquote>
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
</blockquote>
<h3 id="八--新生代老年代">八. 新生代老年代</h3>

新生代
<blockquote>
默认占用堆内存的三分之一
</blockquote>

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 的对象复制到幸区 to, 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, 扔到老年代

幸存区 to, 默认新生代的十分之一

<blockquote>
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, 直接扔老年代
</blockquote>

老年代
<blockquote>
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
</blockquote>

<h3 id="九--happen-before">九. happen-before</h3>

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重排序一致
管程锁定规则: 解锁之前的改动要对获取锁的线程可见
volatile 变量规则: 对 volatile 变量写的操作一定对读的操作可见
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
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
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
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
对象终结规则: 确保对象的构造方法的操作对 finalize 方法可见

<h3 id="十--GC调优">十. GC 调优</h3>
<blockquote>
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, 所以对的应该把新生代调大一些, 最好伊甸区大于: 预估并发量 * 请求平均消耗的内存
</blockquote>