学习笔记 | 理解 Java 虚拟机

作者：ellenbboe
原文链接：https://ld246.com/article/1562755951061
来源网站：链滴

<h4 id="类加载过程">类加载过程</h4>
java 将源代码通过静态编译器转化成字节码文件,class 字节码文件可以保证通过不同平台的解释对 class 字节码文件来实现多处运行
字节码文件被 classloader 加载到 java 虚拟机的内存中构建为 class 对象(注意 classload 只是<stron>加载 class 文件到内存,还有没有进行链接和初始化),而 class.forname 会全做,所以 classlader 比 forname 运行快
<blockquote>
加载:将 class 字节数据流加载到 java 内存中.创建 Class 实例
链接:验证 class 文件的正确性,给静态变量空间并设置内存值
初始化:赋值
</blockquote>
字节码一般通过 JIT 混合执行模式加载到内存
<blockquote>
解释器一开始使用解释执行,省去编译时间,然后 Jvm 使用 JIT 的动态编译技术将热点代码转化成器码交给 cpu 执行
</blockquote>
<h4 id="反射">反射</h4>
在运行的时候可以得到任何类的信息(方法,变量之类的)
new 可以调用任何构造方法,Class 的 newInstance 只能调用无参构造方法
获取 class:
<blockquote>
class.forname(真实类路径)
xx.class
实例.class
</blockquote>
<h4 id="Java引用">Java 引用</h4>
<blockquote>
简记:强软弱虚
</blockquote>
强引用:只要存在就不会被回收
软引用:会在内存不足的时候进行回收
弱引用:不管内存怎么样都会被回收
虚引用:不会影响生存时间
引用队列(ReferenceQueue):存储关联的且被 GC 回收的软引用,弱引用,虚引用(对象被回收,它们的引会被存放到引用队列中)
<h4 id="对象的finalize--">对象的 finalize()</h4>
尽量避免使用该方法拯救对象
当对象不在 gc root 的引用链中,虚拟机会标记对象并进行筛选:查看对象是否覆盖 finalize()或者说 finaize 方法是否已经执行过了,假如对象没有覆盖或者已执行过,则直接被 GC 回收,否则就将对象放到 F-queue 队列中等待处理,为了防止出现死循环或者缓慢执行,虚拟机不承诺等待对象法运行结束 -.- 对象可以将自身(this)赋值给某个变量防止被回收
<h4 id="classloader层级">classloader 层级</h4>
自定义 classloader->application classloader->extension classloader->bootstrap cassloader
自定义 classloader 要继承 classloader,重写 findclass()方法,调用 defineclass()方法
bootstrap classloader 不存在 JVM 的体系中,所以 extclassloader 的 getparents 是 null
<blockquote>
双亲委派机制:从左往右检查类是否加载过,如果到了 bootstrap classloader 还没有加载过就从右往左尝试可否加载该类
自定义类加载器的好处:实现了不同中间件的类隔离,避免类的冲突,使每个类都能用自己依赖的 jar 包,还能从不同地方进行加载(findclass 通过不同方法得到字节流然后调用 defineclas)
</blockquote>
<h4 id="内存布局">内存布局</h4>

线程私有:程序计数器(方法走到哪里的标识,辅助线程的执行和恢复),虚拟机栈(存放栈帧),本地方栈
共享:java 堆(垃圾回收的主要区域,放的是对象实例,常量池) 元数据(放类信息)
<h5 id="虚拟机栈">虚拟机栈</h5>
栈帧是方法运行的基本结构
栈帧里面有局部变量表(存放方法参数和局部变量),操作栈(方法过程中运算和存放数据的地方),动态链接方法返回地址
<h5 id="java堆">java 堆</h5>
分为新生代(eden 和两个 survivor)和老年代(old)
<blockquote>
-Xss:每一个线程中虚拟机栈的大小
-Xms:堆的初始大小
-Xmx:堆的最大空间
</blockquote>
minor gc=young gc 针对新生代
full gc 针对整个堆
新对象在新生代,太大的放老年代,大部分对象在 eden 区,当 eden 满了调用 younggc,有引用的对象转到其中一个 survivor,下一次 ygc 的时候将存活的对象放到另一块 survivor 区域,并清空当前空间,果塞不下就放到老年代,新生代转移 15 次后转到老年代,老年代都放不下了就调用 full gc
<blockquote>
new 出来的字符串实例存放在堆中,常量池会存放他的引用,""字符串直接放在常量池中
</blockquote>
<h4 id="对象实例化过程">对象实例化过程</h4>
确认类元信息时候存在,不存在就进行加载,生成 Class 对象
分配对象内存
设置默认值(不同状态的零值)
设置对象头(对象的基本信息,如 hash 码,gc 信息)
执行 init 方法,初始化成员变量,执行实例化代码块,调用构造方法,将堆中的对象地址赋值给引用变量
<h4 id="垃圾回收">垃圾回收</h4>
<h5 id="废弃常量判断">废弃常量判断</h5>
没有任何对象引用常量
<h5 id="无用类判断">无用类判断</h5>
该类的所有实例都被回收
加载该类的 classloader 被回收
该类的 Class 对象没有被引用
<h5 id="垃圾的确定方法">垃圾的确定方法</h5>
1.引用计数法 缺点 无法判断循环引用
2.可达性分析法 通过 gc root 判断时候被引用
可作为 gc root 对象:
1.类静态属性中引用的对象
2.常量引用的对象
3.虚拟机栈中引用的对象
4.本地方法栈中引用的对象
<h5 id="垃圾回收算法">垃圾回收算法</h5>
1.标记清除:会产生大量不连续的碎片,效率不高
2.标记整理:效率不高,但不会产生碎片
3.复制:运行高效,但内存减小一半,由老年代做后备仓库 -.-
4.分代回收(新生代使用复制,老年代使用标记清除或标记整理)
<blockquote>
新生代变更快,使用复制;老年代存活率高,使用标记清除或标记整理 -.-
</blockquote>
<h5 id="垃圾回收器">垃圾回收器</h5>
新生代用

<blockquote>
<h6 id="Serial回收器">Serial 回收器</h6>
</blockquote>
单线程(一个 cpu 或者一个收集线程并且工作时暂停其他所有工作线程) client
<h6 id="ParNew回收器">ParNew 回收器</h6>
多线程的 Serial 回收器,也会在工作时暂停其他所有工作线程 server
######Parallel Scavenge 回收器
注重吞吐量,尽快完成任务,而其他 CMS 等回收器注重缩短用户线程停顿的时间(用户交互) 自适应调节数 UseAdaptiveSizePolicy
老年代用
<blockquote>
<h6 id="Serial-Old回收器">Serial Old 回收器</h6>
</blockquote>
单线程,工作原理和 Serial 一样
<h6 id="Parallel-Old收集器">Parallel Old 收集器</h6>
与 Parallel Scavenge 配合工作,工作原理和 Parallel Scavenge 一样
<h6 id="CMS回收器">CMS 回收器</h6>
4 个步骤完成回收 1,3 需要 stop the world
1.初始标记:标记 GC roots 能直接关联的对象
2.并发标记:标记堆中存活的对象,和用户线程一起工作
3.重新标记:修正并发标记期间用户线程导致标记产生变动的对象的标记记录
4.并发清除:清除未被标记的对象
隐藏步骤 5:重置线程
缺点:
1.对 cpu 资源敏感,cpu 资源不足时会影响用户程序运行速度
2.并发清理阶段产生的垃圾要等到下一次 GC 时才能回收
3.会产生大量碎片,没有连续空间存放对象时会出发 full gc
不刻意区分新生代和老年代
<blockquote>
<h6 id="G1回收器">G1 回收器</h6>
</blockquote>
将 java 堆划分为大小相同的区域(region)
region 一共有 4 种:eden,survivor,old,humongous(特殊的 old,专门放大型对象)
一个 region 中的对象可以被全堆中的对象引用
使用 remembered set 避免进行全局扫描,当虚拟机发现程序在对 reference 类型数据写操作时,判断用的对象是否在同一 region,如果是,将引用信息放入被引用的对象的 region 的 remembered set 中
步骤:
1.初始标记:标记 gc roots 能直接关联的对象(stop the world)
2.并发标记:标记堆中存活的对象,虚拟机会将对象标记的变化记录在 remember set logs 中
3.最终标记:为了修正在并发标记的时候用户线程对标记产生变动的一些标记记录,虚拟机将上一阶段的 ogs 合并到各自的 region 中(stop the world,可以并行)
4.筛选回收:对各个 region 的回收价值和成本进行排序,优先回收垃圾最多的区域