Java虚拟机

Java内存区域

- 描述：Java内存区域也称为运行时数据区域，主要包括五个部分：【线程私有】：程序计数器、虚拟机栈、本地方法栈、【线程共享】：方法区、堆
- 程序计数器：程序计数器用于记录各个线程执行的虚拟机字节码指令的地址（分支、循环、跳转、异常、线程恢复等都依赖于计数器）
- 虚拟机栈
  - 每个线程在创建的时候都会创建一个「虚拟机栈」，每次方法调用都会创建一个「栈帧」。
  - 每个「栈帧」会包含：局部变量表、操作数栈、动态连接、方法出口等信息
- 本地方法栈：本地方法栈跟虚拟机栈的功能类似，虚拟机栈为虚拟机执行Java方法（字节码）服务，而本地方法栈为虚拟机用到的本地方法服务
- 方法区
  - 用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
  - 和堆一样不需要连续的内存，并且可以动态扩展，动态扩展失败一样会抛出 OutOfMemoryError 异常。
  - 对这块区域进行垃圾回收的主要目标是对常量池的回收和对类的卸载，但是一般比较难实现。
  - HotSpot虚拟机在「JDK8前」用「永久代」实现了「方法区」，在JDK8中，已经用「元空间」来替代了「永久代」作为「方法区」的实现
  - 元空间使用本地内存存储
  - 方法区是一个 JVM 规范，永久代与元空间都是其一种实现方式。
  - 运行时常量池
    - 方法区的一部分
    - 存储的是「类加载」时生成的「直接引用」等信息
- 堆
  - 所有对象都在这里分配内存，是垃圾收集的主要区域（"GC 堆"）【存放对象实例】
  - 堆不需要连续内存，但逻辑上应该被视为连续的，并且可以动态增加其内存，增加失败会抛出 OutOfMemoryError 异常。
  - 针对不同类型的对象采取不同垃圾回收算法
    - 新生代
    - 老年代
- 直接内存
  - 不是运行时数据区的一部分
  - 本地内存

类加载机制

- 类是在运行期间第一次使用时动态加载的，而不是一次性加载所有类。
- 类的生命周期
  - 类加载过程：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、
  - 使用（Using）、卸载（Unloading）
- 类加载过程
  - 加载
    - 通过类的完全限定名称获取定义该类的二进制字节流。
    - 将该字节流表示的静态存储结构转换为方法区的运行时存储结构。
    - 在堆内存中生成一个代表该类的 Class 对象，作为方法区中该类各种数据的访问入口。
  - 连接
    - 验证
      - 确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。
      - 文件格式验证、元数据验证、字节码验证、符号引用验证
    - 准备
      - 为类变量分配内存并设置初始值：类变量是被 static 修饰的变量，一般为0， final关键字修饰则指定值， 使用的是方法区的内存。
      - 实例变量不会在这阶段分配内存，它会在对象实例化时随着对象一起被分配在堆中。
      - 类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。
      - 在 JDK 7 及之后，HotSpot 已经把原本放在永久代的字符串常量池、静态变量等移动到堆中，这个时候类变量则会随着 Class 对象一起存放在 Java 堆中。
    - 解析
      - 将常量池的符号引用替换为直接引用的过程。
      - 符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义的定位到目标就行
      - 直接引用是直接指向目标的指针或者偏移量
      - 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符 7 类符号引用进行。
      - 得到类或者字段、方法在内存中的指针或者偏移量。
  - 初始化
    - 初始化阶段才真正开始执行类中定义的 Java 程序代码（字节码）。
    - 初始化阶段是虚拟机执行类构造器 <clinit>() 方法的过程，而且要保证执行前父类的<clinit>()方法执行完毕。
    - <clinit>()方法由编译器收集，顺序执行所有类变量显式初始化和静态代码块中语句。
- 类初始化时机
  - 主动引用
    - 虚拟机规范中并没有强制约束何时进行加载，但是规范严格规定了有且只有下列五种情况必须对类进行初始化
    - 1、遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。
    - 2、使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行初始化，则需要先触发其初始化。
    - 3、当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
    - 4、当虚拟机启动时，用户需要指定一个要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类；
    - 5、当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为 REF_getStatic, REF_putStatic, REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化；
    - 6、当一个接口中定义了 JDK8 新加入的默认方法（被 default 关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化。
  - 被动引用
    - 所有引用类的方式都不会触发初始化，称为被动引用。
    - 1、通过子类引用父类的静态字段，不会导致子类初始化。
    - 2、通过数组定义来引用类，不会触发此类的初始化。
    - 3、常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。
- 类与类加载器
  - 两个类相等，需要类本身相等，并且使用同一个类加载器进行加载。这是因为每一个类加载器都拥有一个独立的类名称空间。
  - 这里的相等，包括类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果为 true，也包括使用 instanceof 关键字做对象所属关系判定结果为 true。
- 类加载器分类
  - BootstrapClassLoader(启动类加载器)：最顶层的加载类，由 C++实现，负责加载 %JAVA_HOME%/lib目录下的 jar 包和类或者被 -Xbootclasspath参数指定的路径中的所有类。
  - ExtensionClassLoader(扩展类加载器)：主要负责加载 %JRE_HOME%/lib/ext 目录下的 jar 包和类，或被 java.ext.dirs 系统变量所指定的路径下的 jar 包。
  - AppClassLoader(应用程序类加载器)：面向我们用户的加载器，负责加载当前应用 classpath 下的所有 jar 包和类。
- 双亲委派模型
  - 应用程序是由三种类加载器互相配合从而实现类加载，除此之外还可以加入自己定义的类加载器。
  - 类加载器之间的层次关系，称为双亲委派模型（Parents Delegation Model）
    - 启动类加载器
    - 扩展类加载器
    - 应用程序类加载器
    - 用户自定义类加载器
  - 除了顶层的启动类加载器外，其它的类加载器都要有自己的父类加载器。这里的父子关系一般通过组合关系（Composition）来实现，而不是继承关系（Inheritance）。
  - 工作过程
    - 在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。
    - 加载的时候，首先会把该请求委派给父类加载器的 loadClass() 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 BootstrapClassLoader 中。
    - 当父类加载器无法处理时，才由自己来处理。当父类加载器为 null 时，会使用启动类加载器作为父类加载器。
  - 好处
    - 双亲委派模型保证了 Java 程序的稳定运行，可以避免类的重复加载，也保证了 Java 的核心 API 不被篡改。
    - JVM 区分不同类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类

Java源码编译到执行的过程

- Java源码到执行的过程，从JVM的角度看可以总结为四个步骤：编译->加载->解释->执行
- 编译
  - 将源码文件编译成JVM可以解释的class文件【字节码文件】
  - 经过 语法分析、语义分析、注解处理最后才会生成class文件
- 加载
  - 将编译后的class文件加载到JVM中，加载过程可以分为三个步骤：加载->连接->初始化
  - 加载
    - 总结：查找并加载类的二进制数据，在JVM「堆」中创建一个java.lang.Class类的对象，并将类相关的信息存储在JVM「方法区」中
    - 时机：为了节省内存的开销，并不会一次性把所有的类都装载至JVM，而是等到「有需要」的时候才进行装载（比如new和反射等等）
    - 发生：class文件是通过「类加载器」装载到jvm中的，为了防止内存中出现多份同样的字节码，使用了双亲委派机制（它不会去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上）
    - 规则：JDK 中的本地方法类一般由根加载器（Bootstrp loader）装载，JDK 中内部实现的扩展类一般由扩展加载器（ExtClassLoader ）实现装载，而程序中的类文件则由系统加载器（AppClassLoader ）实现装载。
  - 连接
    - 总结：对class的信息进行验证、为「类变量」分配内存空间并对其赋默认值，可以分为三个步骤：验证、准备、解析
    - 验证：验证类是否符合 Java 规范和 JVM 规范
    - 准备：为类的静态变量分配内存，初始化为系统的初始值
    - 解析：虚拟机将常量池中的符号引用转为直接引用的过程
  - 初始化
    - 总结：顺序执行所有类变量显式初始化和静态代码块中语句
    - 收集class的静态变量、静态代码块、静态方法至()方法，随后从上往下开始执行
    - 如果「实例化对象」则会调用方法对实例变量进行初始化，并执行对应的构造方法内的代码。
- 解释
  - 把字节码转换为操作系统识别的指令
  - 两种方式把字节码信息解释成机器指令码
    - 字节码解释器
    - 即时编译器JIT
  - 直接解释+编译解释
    - JVM会对「热点代码」做编译，非热点代码直接进行解释。
    - 使用「热点探测」来检测是否为热点代码，「热点探测」一般有两种方式，计数器和抽样。
    - 即时编译器JIT把热点方法的指令码保存起来，下次执行的时候就直接执行缓存的机器语言
- 执行：操作系统把解释器解析出来的指令码，调用系统的硬件执行最终的程序指令。

GC垃圾回收

- 判断一个对象是否可被回收
  - 引用计数算法
    - 当对象被引用则+1，对象引用失效则-1。当计数器为0时，说明对象不再被引用，可以被可回收
    - 缺点：如果对象存在循环依赖，那就无法定位该对象是否应该被回收
  - 可达性分析算法
    - 以 GC Roots 为起始点进行搜索，可达的对象都是存活的，不可达的对象可被回收。
    - 「GC Roots」是一组必须「活跃」的「引用」，跟GC Roots无关联的引用即是垃圾，可被回收
    - GC Roots 一般包含
      - 虚拟机栈中局部变量表中引用的对象
      - 本地方法栈中 JNI 中引用的对象
      - 方法区中类静态属性引用的对象
      - 方法区中的常量引用的对象
  - 无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。
- 引用类型
  - 四种引用类型
    - 强引用
      - 被强引用关联的对象不会被回收
      - 使用 new 一个新对象的方式来创建强引用
    - 软引用
      - 被软引用关联的对象只有在内存不够的情况下才会被回收
      - 使用 SoftReference 类来创建软引用。
    - 弱引用
      - 被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。
      - 使用 WeakReference 类来创建弱引用。
    - 虚引用
      - 虚引用主要用来跟踪对象被垃圾回收的活动。
      - 使用 PhantomReference 来创建虚引用。
  - 在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多
  - 软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生
- 垃圾回收算法
  - 标记-清除
    - 首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。
    - 问题
      - 效率问题：标记和清除过程效率都不高；
      - 空间问题：标记清除后会产生大量不连续的内存碎片，导致无法给大对象分配内存。
  - 标记-整理
    - 让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。
    - 优点：不会产生内存碎片，吞吐量大
    - 不足：需要移动大量对象，处理效率比较低
  - 标记-复制
    - 将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。
    - 不足：只使用了内存的一半。
  - 商业虚拟机用标记-复制回收新生代
    - 分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor，Eden:Survivor 8:1
    - 回收时，将 Eden 和 Survivor 中还存活着的对象全部复制到另一块 Survivor 上，最后清理 Eden 和使用过的 Survivor
    - Appel式回收
  - 分代收集
    - 现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。
    - 新生代
      - 标记-复制算法
      - 新生代指的是死得快的对象
    - 老年代
      - 标记-清除或者标记-整理算法
      - 老生代指的是活得久的对象，对象存活率较高如果使用标记-复制算法要进行较多的复制操作，效率降低
- 垃圾收集器
  - 新生代
    - Serial、ParNew(Serial的多线程版本)、Parallel Scavenge(吞吐量优先)、CMS、Serial Old（Serial老年代版本）、Parallel Old（Parallel Scavenge老年代版本）、G1
    - Serial（标记-复制）、ParNew（标记-复制）、Parallel Scavenge（标记-复制）、CMS（标记-清除）
  - 老年代
    - Serial Old（标记-整理）、Parallel Old（标记-整理）、CMS（标记-清除）
  - 单线程与多线程
    - 垃圾收集器使用线程的数量
    - Serial、Serial Old、：单线程
    - ParNew、Parallel Scavenge、Parallel Old：多线程
  - 串行、并行、并发
    - 串行
      - 串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序
      - 使用单线程进行垃圾回收，并且垃圾回收过程中阻塞用户线程，这种现象称之为STW（Stop-The-World）
      - 单线程的垃圾收集器：Serial、Serial Old
    - 并行
      - 并行在串行的基础之上做了改进，将单线程改为了多线程进行垃圾回收，这样可以缩短垃圾回收的时间。但是垃圾回收过程也会阻塞用户线程
      - 多线程的垃圾收集器：ParNew、Parallel Scavenge、Parallel Old
    - 并发Concurrent
      - 用户线程和垃圾收集线程同时执行（不一定是并行，可能是交替执行，在时间段内交替运行），不需要停顿用户线程。
      - 「部分」的GC场景下可以让GC线程与用户线程并发执行
      - CMS、G1
- CMS垃圾收集器
  - CMS(Concurrent Mark Sweep)并发 标记-清除算法
  - 流程
    - 初始标记：标记 GC Roots 能直接关联到的对象，速度很快，会阻塞用户线程。
    - 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不会阻塞
    - 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，阻塞用户线程。
    - 并发清除：开启用户线程， GC线程对未标记的区域清扫
  - 缺点
    - 吞吐量低：低停顿时间是以应用程序变慢，牺牲吞吐量为代价的，导致 CPU 利用率不够高。
    - 浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。
    - 由于浮动垃圾的存在，因此需要预留出一部分内存，如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure：无法处理浮动垃圾，可能出现 Concurrent Mode Failure。
    - 虚拟机将临时启用 Serial Old 来替代 CMS。
    - 标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。
- G1垃圾收集器
  - G1开创了收集器面向局部收集的设计思路和基于Region的内存布局形式
  - G1（Garbage-First），它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。
  - G1 把堆划分成多个大小相等的独立区域（Region），可以单独进行垃圾回收，新生代和老年代不再物理隔离，每个Region根据需要扮演新生代的Eden 、Survivor空间或者老年代空间。
  - Region
    - 通过记录每个 Region 垃圾回收时间以及回收所获得的空间，并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。
    - 每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region，通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描。
    - 年轻代的Region，它的 Remembered Set只保存了来自老年代的引用；老年代的 Region ，它的 Remembered Set 也只会保存老年代对它的引用
  - 步骤
    - 初始标记
    - 并发标记
    - 最终标记
      - 为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，JVM将这段时间对象变化记录在线程的 Remembered Set Logs 里面
      - 最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要阻塞用户线程，但是可并行执行。
    - 筛选回收
      - 首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。
      - 此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。
  - 特点
    - 并行与并发
    - 空间整合：与 CMS 的"标记-清除"算法不同，G1 从整体来看是基于"标记-整理"算法实现的收集器；从局部（两个Region）上来看是基于"标记-复制"算法实现的。
    - 可预测的停顿：降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型
- 内存分配与回收策略
  - GC分类
    - 只对新生代进行垃圾收集：Minor GC新生代收集
    - 只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集：Major GC 老年代收集
    - 对整个新生代和部分老年代进行垃圾收集：Mixed GC混合收集 G1
    - 收集整个 Java 堆和方法区：Full GC整堆收集
  - 内存分配策略
    - 对象优先在Eden分配，Eden空间不够发起Minor GC
    - 大对象直接进入老年代
    - 长期存活的对象Minor GC一次年龄+1直到移动到老年代
    - 动态年龄判定：Survivor中相同年龄所有对象大小的总和大于Survivor空间一半，年龄大于等于的对象直接进入老年代
  - 空间分配担保
    - 在发生 Minor GC 之前，先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果是，那么 Minor GC 安全
    - 检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，大于则Minor GC，否则Full GC
  - Minor GC触发条件：当 Eden 空间满时，就将触发一次 Minor GC
  - Full GC触发条件
    - 调用 System.gc()
    - 老年代空间不足
    - 空间分配担保失败
    - JDK 1.7 及以前的永久代空间不足
    - 执行 CMS GC 的过程中，报 Concurrent Mode Failure