# 深入理解java虚拟机

- 类加载机制
  - 编译原理
    
    - 前端编译
      - javac
      - .java文件转变成.class（字节码）文件
    - 后端编译
      - 字节码转变成机器码
      - 解释执行
        - JVM 通过解释字节码将其翻译成对应的机器指令，逐条读入，逐条解释翻译
      - JIT （即时编译）
        - 把部分“热点代码”翻译成本地机器相关的机器码，并进行优化，然后再把翻译后的机器码缓存起来
  - 类的生命周期
    
    

    Java代码经过编译器编译为class文件（本地机器码转变为字节码），类加载器把class文件加载到虚拟机中运行和使用
    - 加载（Loading）
      - ①通过类的全限定名获取存储该类的class文件 ②解析成运行时数据，即instanceKlass实例，存放在方法区 ③在堆区生成该类的Class对象，即instanceMirrorKlass实例
      - 何时加载：new、反射、main()、初始化子类加载父类等
    - 连接
      - 验证（Verification）
        - 校验字节码文件的正确性
      - 准备（Preparation）
        - 为静态变量分配内存、赋初值

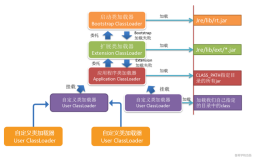
          如果被final修饰，会直接完成赋值
      - 解析（Resolution）
        - 将符号引用替换为直接引用

          符号引用：以一组符号来描述所引用的目标，如在编译期间是不知道Logger类的内存地址的，所以只能先用com.framework.Logger（假设是这个，实际上是由类似于

> CONSTANT_Class_info的常量来表示的）来表示Logger类的地址
> 直接引用：是直接指向目标的地址或句柄

- 初始化（Initialization）
  - 执行 `<clinit>()` ，初始化类变量、静态代码块
- 使用（Using）
- 卸载（Unloading）

- 类加载器



- 类加载器分类
  - 启动类加载器（Bootstrap ClassLoader）

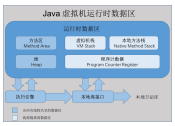
    由C++编写，JVM的一部分
    - ①加载JAVA_HOME\lib目录或者被-Xbootclasspath参数指定的部分类 ②加载main函数所在的类，启动扩展类加载器、应用类加载器
  - 扩展类加载器（Extension ClassLoader）
    - 由sun.misc.Launcher.ExtClassLoader实现，加载JAVA_HOME\lib\ext目录或者被java.ext.dirs系统变量指定的所有类库
  - 应用类加载器（Application ClassLoader）
    - 由sun.misc.Launcher.AppClassLoader实现，负责加载ClassPath路径上的类，如果没有自定义类加载器，应用类加载器就是程序默认的类加载器
  - 自定义类加载器（User ClassLoader）

    JVM提供的类加载器只能加载指定目录的类（jar和class），如果想从其他地方获取class文件，就需要自定义类加载器来实现，自定义类加载器主要都是**通过继承ClassLoader来实现**，但无论是通过继承ClassLoader还是它的子类，最终自定义类加载器的父加载器都是应用程序类加载器
    - 通过继承ClassLoader来实现
- 双亲委派

  如果一个类加载器收到了加载某个类的请求，则该类加载器并不会去加载该类，而是把这个请求**委派给父类加载器**。当父类加载器在其搜索范围内无法找到所需的类，并将该结果反馈给子类加载器，子类加载器会尝试去自己加载
  - 作用
    - 沙箱安全机制，防止核心API库被篡改
    - 避免类的重复加载，保证被加载类的唯一性
  - 打破双亲委派
    - **自定义类加载器**
      - 不委派、向下委派
    - SPI

        - 服务发现机制。通过在ClassPath路径下的META-INF/services文件夹查找文件，自动加载文件里所定义的类。这一机制为很多框架扩展提供了可能，比如在Dubbo、JDBC中都使用到了SPI机制

- 内存模型




    - 运行时数据区域
        - 程序计数器（Program Counter Register）
            - 可看做是当前线程执行字节码的行号指示器
        - 虚拟机栈（Java Virtual Machine Stack）
            - 由栈帧组成，每个栈帧中存储了局部变量表、操作数栈、动态链接、返回地址等信息，一个栈帧对应一个方法，一个方法从被调用到执行结束，就对应了一个栈帧从入栈到出栈的过程
        - 本地方法栈（Native Method Stack）
            - 为Native方法服务。在HotSpot中直接把本地方法栈和虚拟机栈合二为一
        - 堆（Heap）
            - 存放对象和数组
        - 方法区（Method Area）

            方法区是规范，永久代、元空间是具体实现。

            永久代：JDK8之前方法区的实现，在堆中
            元空间：使用本地内存，JDK8后取代永久代，存储类信息和编译代码

            - 存储虚拟机加载的类信息、常量、静态变量，以及编译器编译后的代码等数据
    - 对象的内存布局

        - 对象头（Header）
            - Mark Word：存储运行时数据，如HashCode、GC年龄分代、锁状态标识等
            - 类型指针：指向其类型元数据，通过这个指针确定这个对象的实例
        - 实例数据（Instance Data）
            - 对象存储的真正有效信息
        - 对齐填充（Padding）
    - 对象访问

        通过reference指向对象的引用

        - 句柄和直接指针
    - 内存溢出

内存泄露：内存没有被回收

- 除程序计数器外都有可能发生OutOfMemoryError
- 线程请求的栈深度大于虚拟机栈所允许的最大深度：StackOverflowError

- 垃圾回收
  - 对象已死
    - 引用计数算法（Reference Counting）
      - 在对象中添加一个引用计数器
      - 原理简单，判断效率高
      - 难以解决循环引用问题
    - 可达性分析算法（Reachability Analysis）
      - GC Roots
        - 虚拟机栈（栈帧中本地变量表）中引用的对象
        - 方法区中类静态属性引用对像
        - 方法区中常量引用的对象
        - 本地方法栈中JNI（native）引用对象
    - 引用
      - 强引用
        - new Object()
      - 软引用
        - OOM前回收
      - 弱引用
        - 生存到下一次GC
      - 虚引用
        - 回收时收到系统通知
    - finalize()
    - 回收方法区
      - 主要是对常量池的回收和对类的卸载
      - 为了避免内存溢出，在大量使用反射和动态代理的场景都需要虚拟机具备类卸载功能
  - 垃圾收集算法

    内存规整：指针碰撞（Bump The Pointer）
    内存不规整：空闲列表（Free List）
    - 分代收集理论

      弱分代假说（Weak Generational Hypothesis）：绝大多数对象都是朝生夕灭的
      强分代假说（Strong Generational Hypothesis）：熬过越多次垃圾收集过程的对象就越难以消亡
      跨代引用假说（Intergenerational Reference Hypothesis）：跨代引用相对于同代引用来说仅占极少数

        - **垃圾收集器设计原则：收集器应该将Java堆划分出不同的区域，然后根据对象的年龄分配到不同的区域中存储**
        - 通过记忆集解决跨代引用问题

          > 不应为了少量的跨代引用去扫描整个老年代，只需要在新生代上建立一个全局数据结构（记忆集，Remembered Set），把老年代划分多个区域，标记处哪一块区域存在跨代引用
    - 标记-清除算法（Mark-Sweep）

      1. 标记回收对象，回收标记对象
      2. 标记存活对象，回收未标记对象

        - 执行效率不稳定，适用于少量垃圾
        - 内存空间碎片化
    - 复制算法（Copying）

      半区复制

        - 实现简单，运行高效，适合对象存活率低时
        - 空间浪费
    - 标记-整理算法（Mark-Compact）

      先执行标记，然后让所有的存活对象都向一端移动，最后清理掉边界以外的内存

        - 不会产生内存碎片
        - 需要移动大量对象，处理效率比较低
        - **对象移动必须暂停用户应用程序**
- HotSpot算法实现
    - OopMap
        - 记录该类型的对象内什么偏移量上是什么类型的数据
    - 根节点枚举
        - 必须暂停用户线程
    - 安全点（Safepoint）
        - 记录OopMap的位置
        - 安全点和安全区域确保停顿用户线程，以进行根节点枚举
    - 安全区域（Safe Region）
        - 引用关系不会发生变化的代码片段，在该区域的垃圾收集是安全的
    - 记忆集（Remembered Set）
        - 记录从非收集区域指向收集区域的指针集合

          > 只需通过记忆集判断出某一块非收集区域是否存在有指向了收集区域的指针

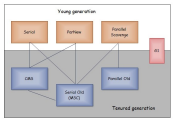
        - HotSpot通过卡表（Card Table）实现记忆集
            - 通过写屏障（Write Barrier）维护卡表状态

              > 写屏障：在引用对象赋值时产生一个环形通知

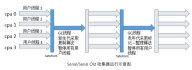
        - 解决跨代引用问题

- 三色标记（Tri-color Marking）
  - 并发标记“对象消失”问题：增量更新和原始快照

- 垃圾收集器




  - Young generation
    - Serial



Client 场景下的默认新生代收集器

      - 单线程，复制算法，简单高效，单核下没有线程交互开销

    - ParNew

Server 场景下默认的新生代收集器，除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用；激活CMS后的默认新生代收集器

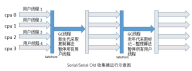
ParNew+CMS在JDK9之后不再是官方推荐的服务端模式下的收集器解决方案，被G1替代

      - Serial 收集器的多线程版本，复制算法

    - Parallel Scavenge
      - 目标是达到一个可控制的吞吐量，标记-整理算法

        吞吐量指 CPU 用于运行用户程序的时间占总时间的比值

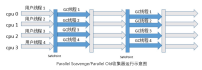
  - Tenured generation
    - Serial Old



在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用
作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用

      - 单线程，标记-整理算法

    - Parallel Old



在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器

      - Parallel Scavenge 收集器的老年代版本，标记-整理算法

    - CMS(Concurrent Mark Sweep)

步骤

1. 初始标记：标记出GC Roots直接关联的对象，速度很快，**需要停顿**，单线程执行
2. 并发标记：进行 GC Roots Tracing 的过程，耗时较长，不需要停顿

3. 重新标记：修正在并发标记阶段因用户程序执行而产生变动的标记记录，**需要停顿**
4. 并发清除：不需要停顿

- 以最短回收停顿时间为目标，标记-清除算法
- 缺点
  - 低停顿时间以牺牲吞吐量为代价
  - 无法处理浮动垃圾，可能出现 Concurrent Mode Failure

    浮动垃圾是指并发标记和并发清除阶段产生的垃圾，需要到下一次 GC 时才能进行回收

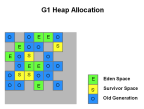
    CMS收集器需要预留出一部分内存存放浮动垃圾。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。
  - 标记-清除算法会产生空间碎片

- G1（Garbage First）

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

步骤：
1. 初始标记：标记出GC Roots直接关联的对象，速度很快，**需要停顿**，单线程执行
2. 并发标记：进行 GC Roots Tracing 的过程，耗时较长，不需要停顿
3. 最终标记：修正在并发标记阶段因用户程序执行而产生变动的标记记录，**需要停顿**
4. 筛选回收：对各个Region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。把决定回收的Region的存活对象复制到空Region中，**需要停顿**

- 基于Region内存布局，每个Region都可以被标记为Eden、Survivor、Old、Humongous（存储大对象）




- **内存分配**
  - 对象优先在Eden分配
  - 大对象直接进入老年代
  - 长期存活对象将进入老年代
  - 动态对象年龄判定

    相同年龄对象总和大于Survivor空间的一半，则大于或等于该年龄的对象直接进入老年代
  - 空间分配担保

    在进行Minor GC之前，会判断老年代最大连续可用空间是否大于新生代所有对象总空间，如果大于，说明Minor GC是安全的，否则会判断是否允许担保失败，如果允许，判断老年代最大连续可用空间是否大于历次晋升到老年代的对象的平均大小，如果大于，则执行Minor GC，否则执行Full GC
- GC
  - Minor GC：新生代垃圾收集

        - Major GC：老年代垃圾收集（目前只有CMS有单独收集老年代的行为）
        - Mixed GC：收集整个新生代以及部分老年代（目前只有G1有）
        - Full GC：收集整个堆和方法区
- 性能调优
    - 使用较小的内存占用来获得较高的吞吐量或者较低的延迟
    - 调优工具
        - jps（JVM process Status）：查看虚拟机启动的进程

          -l：执行主类名称
          -v：JVM启动参数
        - jstat（JVM Statistics Monitoring Tool）：监视虚拟机信息，显示虚拟机进程中的类信息、内存、垃圾收集、JIT 编译等运行数据

          `jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]`

          例：`jstat -gc -h3 31736 1000 10` 表示分析进程 id 为 31736 的 gc 情况，每隔 1000ms 打印一次记录，打印 10 次停止，每 3 行后打印指标头部
        - jmap（Memory Map for Java）：**生成堆转储快照**

          `jmap -heap pid` 打印堆信息
          `jmap -histo pid` 打印出当前堆中每个类的实例数量和内存占用
          `jmap -dump:format=b,file=xxx.hprof pid` 把当前堆内存的快照转储到二进制文件xxx.hprof中
        - jstack（Stack Trace for Java）：生成虚拟机当前时刻的线程快照

          `jstack <pid>|grep -A 10 4cd0` 打印线程堆栈信息中4cd0 (线程pid十六进制)这个线程所在行的后面10行
          `top -H -p <pid>` 显示进程pid中的线程占用
        - jinfo：实时查看和调整虚拟机各项参数
    - Arthas

      下载之后，java -jar运行即可
        - dashboard：查看进程的运行情况（线程、内存、GC、运行环境信息）
        - thread：查看线程详细情况

          thread <线程ID>：查看线程堆栈
          thread -b：可以查看线程死锁