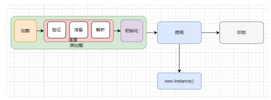
JVM结构

类加载子系统（加载类文件到内存中生成Class对象）

类的生命周期

一个类从诞生到卸载，大体分为如下几步：



类加载过程（由类加载器或者双亲委派）

- 加载
  - 查找字节码并加载到内存
    - 字节码-本质是class文件，通过一个类的全限定名获取一个类的二进制字节流，内存中生成一个代表这个类的java.lang.class对象作为方法区代表这个类的访问入口
      - JVM会为静态变量分配内存
- 链接
  - 验证
    - 确保被加载的类符合Java语言规范
      - 定义：验证是确保被加载的类或接口符合JVM规范的过程。这个阶段的目的是确保Class文件的数据格式是正确的，没有被篡改，且不会对JVM造成安全风险。
      - 内容：包括文件格式验证、元数据验证、字节码验证和符号引用验证等多个阶段。
      - 目的：确保类的结构正确无误，比如方法的签名是否合法、类中的字段和方法是否有冲突等。
  - 准备
    - 分配内存给变量赋初始值（0值或引用类型的null）
      - 还会给其他值（特殊情况），如果一个属性是被static+final修饰的类型，会在.class文件中会记录这个值，此时直接赋给他
  - 解析
    - 将符号引用解析为直接引用，从符号来描述变为直接指向目标的指针、偏移量、句柄。
- 初始化
  - 定义：在初始化阶段，JVM执行类构造器<clinit>()方法的过程。这个方法是由编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并产生的。
  - 触发时机：当创建类的实例、访问类的静态变量，或者调用类的静态方法时，如果类还未被初始化，就会先进行初始化。
  - 执行类的初始化代码
    - 主动引用：当直接使用到类的时候才会触发初始化，比如实例化对象、访问静态字段、调用静态方法等。
    - 被动引用：通过子类引用父类的静态字段不会导致子类初始化。通过数组定义来引用类、常量在编译阶段会存入调用类的常量池中，这些情况都不会触发该类的初始化。

如何加载类？

- 通过类加载器
  - 引导类加载器
  - 自定义类加载器
    - 写一个类去继承jdk提供的对象类，默认请款下是extclassloader，appclassloader，webappclassloader
- 双亲委派
  - 类加载器在尝试加载类时，会先委托给其父加载器去尝试加载这个类。这种委派机制一直向上递归，直到达到最顶层的类加载器引导类加载器（Bootstrap ClassLoader）
    - 如果父加载器不能加载该类（因为它不在父加载器的搜索范围内），则由子加载器自己来尝试加载这个类
  - 优点
    - 防止内存中出现多份同样的字节码
      - 由于在整个委派链中最多只会有一个加载器能成功加载一个类，这就避免了同一个类被多个加载器重复加载到内存中
    - 保护程序安全，防止核心API被随意篡改
  - 缺点
    - 不同的应用需要使用不同版本的同一个类库，这时候双亲委派模型就可能会导致问题。
      - 为了解决这种问题，Java提供了一种叫作上下文类加载器（Context ClassLoader）的机制，允许开发者在运行时使用不同的类加载器来加载类。
  - 某些特定场景下，可能需要打破双亲委派模型
    - 自定义网络加载
      - 当需要从网络或其他非标准来源动态加载类时，可能需要绕过JVM核心库的加载过程，使用自定义的类加载器直接加载类
    - 热部署
      - 比如Tomcat，需要实现类的热替换功能，即在不重启服务器的情况下更新类定义
    - OSGi框架
      - 在OSGi中，每个模块（或者称为Bundle）都有自己的类加载器
- 什么时候会加载
  1. 当创建类的实例时，如果该类还没有被加载，则会触发类的加载。例如，通过关键字new创建一个类的对象时，JVM会检查该类是否已经加载，如果没有加载，则会调用类加载器进行加载。
  2. 当使用类的静态变量或静态方法时，如果该类还没有被加载，则会触发类的加载。例如，当调用某个类的静态方法时，JVM会检查该类是否已经加载，如果没有加载，则会调用类加载器进行加载。
  3. 当使用反射机制访问类时，如果该类还没有被加载，则会触发类的加载。例如，当使用Class.forName()方法加载某个类时，JVM会检查该类是否已经加载，如果没有加载，则会调用类加载器进行加载。
  4. 当JVM启动时，会自动加载一些基础类，例如java.lang.Object类和java.lang.Class类等。

堆和栈的区别

- 内存分配和管理
  - 堆内存的分配和释放是动态的、手动的（或依赖于垃圾回收机制）
  - 栈内存的分配和释放是自动的
- 性能
  - 栈内存由于其管理方式简单（LIFO）访问速度比堆快
- 大小
  - 栈的大小一般比堆小，堆的大小受到虚拟内存大小的限制。
- 用途
  - 栈主要用于执行线程的临时数据存储，如函数的参数和局部变量
  - 堆用于存储需要跨函数调用或具有不确定生命周期的数据

线程私有

- 程序计数器
  - 记录了每个线程正在执行的字节码指令地址
  - 在切换线程时恢复执行位置
- 本地方法栈
  - Java中定义的方法，一般由其他语言实现（C++）
- 虚拟机栈（Java方法栈）
  - 每个线程创建时都会创建一个虚拟机栈，栈内保存一个个栈帧，每个栈帧对应一个方法。
    - 栈帧
      - 局部变量表
        - Slot相当于定义的一个一个局部变量
      - 操作数栈（过程需要记住吗？）
        - 执行字节码指令过程中进行计算
      - 方法返回地址
      - 动态链接
      - 附加信息
  - 存在内存溢出和栈溢出
    - 线程太多会出现OOM
      - **OOM一定会导致JVM 退出吗**
        - 都是可以被catch的，如果被catch掉之后，程序还是可以正常执行
    - 方法调用层次太多会出现SOFE. e.g.递归
  - 可以通过-XX设置虚拟机栈大小
    - java -Xss1M YourMainClass这个是设置为1M大小

执行引擎（执行字节码指令，翻译字节码为机器指令）

- 解释器（解释字节码指令为机器码）
- JIT编译器（将频繁执行的字节码转换为本地机器代码）
  - 可以把热点指令缓存起来加快执行效率
- 垃圾回收器（GC，在运行时管理内存，回收不再使用的对象，防止内存泄漏）

本地方法库（本地接口和本地方法）

内存泄露和内存溢出区别

- 内存泄漏指的是程序中分配的内存在不再需要时没有被正确释放或回收的情况
  - 这会导致程序持续占用内存，随着时间的推移，可用内存逐渐减少，最终可能导致程序性能下降或崩溃。
- 内存溢出指的是程序试图分配超过其可用内存的内存空间的情况
  - 这通常会直接导致Java程序崩溃。

常见版本？

- HotSpot JVM
  - 这是目前最流行的 JVM 版本，由 Oracle 提供。它是 Java Development Kit (JDK) 的一部分，并广泛用于生产环境。HotSpot JVM 以其优异的性能和广泛的优化功能而著称。
- OpenJDK JVM
  - 这是一个开源项目，Oracle 的 HotSpot JVM 就是基于这个项目的。OpenJDK 是许多其他自由和开源 Java 版本的基础
- GraalVM
  - 由 Oracle Labs 开发，GraalVM 不仅可以作为一个标准的 JVM 运行 Java 应用程序，还支持其他编程语言，如 JavaScript、Python、Ruby 等。它提供了一个高性能的多语言运行时环境

JVM常用的命令？

- 1.java
  - 功能：启动一个Java应用程序。
  - 示例：java -jar yourapp.jar 启动一个Jar包应用。
- 2. javac
  - 功能：Java编译器，用于将Java源代码文件编译成字节码文件。
  - 示例：javac HelloWorld.java 编译一个名为HelloWorld.java的Java源文件。
- 3. jps
  - 功能：Java Virtual Machine Process Status Tool，显示当前系统中所有的HotSpot虚拟机进程。
  - 示例：jps -l 显示Java进程的全路径信息。
- 4. jstat
  - 功能：用于收集HotSpot虚拟机的性能监控数据。
  - 示例：jstat -gcutil <pid> 1000 每1000毫秒监控一次指定进程的垃圾收集情况。
- 5. jstack
  - 功能：打印指定Java进程或核心文件的Java堆栈信息，常用于分析线程问题。
  - 示例：jstack -l <pid> 打印指定Java进程的线程堆栈和锁信息。
- 6. jmap
  - 功能：生成Java进程的内存映射（heap dump文件），用于内存分析。
  - 示例：jmap -heap <pid> 查看指定进程的堆配置信息。
  - 注意：在生产环境中使用jmap生成堆转储可能会显著影响性能。
- 7. jconsole
  - 功能：Java监视与管理控制台，提供了图形界面，用于监视Java应用程序的内存使用、线程使用、类加载情况等。
  - 使用方式：直接运行jconsole，然后连接到你想要监控的Java进程。
- 8. jvisualvm
  - 功能：集成了多个JDK命令行工具的图形工具，提供了强大的可视化界面，用于深入分析Java应用程序。
  - 使用方式：运行jvisualvm并通过其界面连接到各个Java进程。
- 9. jcmd
  - 功能：用于发送诊断命令请求到JVM，是一个多功能的工具，可以执行堆转储、生成垃圾收集报告、线程转储等。
  - 示例：jcmd <pid> GC.run 对指定的Java进程执行垃圾收集。

运行时数据区（存储运行时数据）

对象在堆中的生命周期

- 在 JVM 内存模型的堆中，堆被划分为新生代和老年代
- 新生代又被进一步划分为 Eden区 和 Survivor区，Survivor 区由 From Survivor 和 To Survivor 组成
- 当创建一个对象时，对象会被优先分配到新生代的 Eden 区 此时 JVM 会给对象定义一个对象年轻计数器（-XX:MaxTenuringThreshold）
- 当 Eden 空间不足时，JVM 将执行新生代的垃圾回收（Minor GC）
- JVM 会把存活的对象转移到 Survivor 中，并且对象年龄 +1
- 对象在 Survivor 中同样也会经历 Minor GC，每经历一次 Minor GC，对象年龄都会+1
- 如果分配的对象超过了-XX:PetenureSizeThreshold，对象会直接被分配到老年代

对象的分代晋升

- 触发新生代的GC(Young GC、MinorGC)，在新生代的GC过程中，没有被回收的对象会从Eden区被搬运到Survivor区，这个过程通常被称为“晋升”
- 对象也可能会晋升到老年代，触发条件主要看对象的大小和年龄
  - 躲过15次GC。
  - 动态对象年龄判断
    - 如果在Survivor空间中小于等于某个年龄的所有对象大小的总和大于Survivor空间的一半时，就把大于等于这个年龄的对象都晋升到老年代
  - 大对象直接进入老年代
    - -XX:PretenureSizeThreshold 来设置大对象的临界值

内存回收阶段

- **Eden满了之后会触发 YoungGC/MinorGC**，找到垃圾并回收，存活对象放S0，给一个数字标记存活次数
- 再次满了之后，经历相同过程，S0区也有对象被回收，存活对象放进S1，存活计数加1
- 数据超过15次（阈值）直接进入老年代，如果还没到阈值但是对象太多了会通过内存担保机制直接进入老年代
  - 大对象直接进入老年代
- **FullGC的触发条件**
  - 老年代空间不足
  - 空间分配担保失败
  - 代码中执行System.gc()

一些堆区的指令

- -Xms：ms 指定堆的初始化大小，等价于-XX：InitialHeapSize
- -Xmx：mx 指定堆最大内存大小，等价于-XX：MaxHeapSize
- 一般会把-Xms和Xmx设置为一样，这样就不用再GC后修改堆内存大小，提高效率
- 默认情况下
  - 初始化内存大小 = 物理内存大小/64
  - 最大内存大小 = 物理内存大小/4
- -XX:NewRatio
  - 配置新生代老年代比例，默认为2（新1老2），一般无需调整

堆内存的垃圾回收过程

- 标记（Mark）
  - 标记阶段
    - 识别所有从根集（root set）开始可达的对象。根集通常包括全局静态变量、活动线程的栈帧中的本地变量、活动 Java 方法的参数等。
  - 所有从根集可直接或间接访问到的对象都被视为活动对象，即这些对象还在被使用。垃圾回收器会遍历所有对象引用，标记所有可达的对象。
- 清除（Sweep）或压缩（Compact）
  - 清除阶段
    - 涉及到释放未被标记的对象所占用的内存。在简单的垃圾回收器中，这个过程可能仅仅是将未标记对象的内存标记为“空闲”，以便将来重新利用。
  - 在更复杂的垃圾回收器中，可能包括压缩步骤，即移动对象来消除堆中的碎片，这样可以连续地分配大块内存，提高内存分配的效率。
- 分代收集
  - 大多数现代的 Java 垃圾回收器采用分代垃圾收集 策略，将堆内存分为新生代（Young Generation）和老年代（Old Generation）。新生代通常包括 Eden 区域和两个幸存者（Survivor）区域。
  - 新对象首先在 Eden 区域分配。当 Eden 区域满时，进行一次 Minor GC，活动对象被转移到一个幸存者区域，非活动对象被清除。
  - 随着一些对象在多次 Minor GC 后仍然存活，它们会被晋升到老年代。老年代的对象较少经历垃圾回收，但当进行垃圾回收时，通常会触发 Major GC 或 Full GC，这个过程比 Minor GC 更耗时。

垃圾回收

为什么要进行垃圾回收？

- 垃圾是指在JVM中没有任何引用指向它的对象，如果不清理这些垃圾对象就会一直占用内存。没有足够的内存给其他对象使用，最终会导致内存泄漏。

垃圾回收的基本原理

- 找到垃圾
  - 引用计数法（不用这个缺点多）
    - 每个对象都保存在一个引用计数器属性，用户记录被引用对象的次数
      - 优点
        - 实现简单，计数器为0则为垃圾对象
      - 缺点：
        - 需要额外空间存储引用计数
        - 需要额外空间维护引用计数
        - 无法处理循环引用问题-比如相互引用，它们计数为1，但是是垃圾对象
  - 可达性分析法
    - 会以GC root为起点，逐层找到引用的对象，被找到就存活
    - GC roots是一组引用，包括很多
      - 虚拟机栈中的引用：这些是指在虚拟机栈（线程的栈帧）中的本地变量引用的对象。只要线程活动，这些引用就是活动的。
      - 方法区中的静态引用：这些是指类静态字段引用的对象。类一旦被加载到内存中，其静态字段就是 GC roots。
      - 方法区中的常量引用：这些是指常量池中引用的对象，如字符串常量、类常量等。
      - 本地方法栈中的引用：这些是指本地方法（native method）中引用的对象。
  - 如何判断变成垃圾了？
    - 四种引用
      - 强软弱虚
        - 强不会被回收
          - 如果一个对象具有强引用，垃圾回收器绝不会回收它
        - 软在不足时会被回收
          - 软引用对于垃圾回收器来说要比强引用更容易回收。只有在JVM即将抛出内存溢出异常之前，才会回收这些对象。软引用主要用于实现内存敏感的缓存。
        - 弱被发现就会被回收
          - 在垃圾回收器线程扫描其区域时，只要发现弱引用，不管当前内存空间足够与否，都会回收其对象。
        - 虚引用完全不会对对象的生存时间构成影响
          - 无法通过虚引用来获取对一个对象的实例。虚引用主要用来跟踪对象被垃圾回收的活动
          - 垃圾回收时会回收所有只存在虚引用的对象，但回收前会将引用加入到与之关联的引用队列中，以便程序可以跟踪对象的回收状态
- 垃圾回收

垃圾回收的方法

- 标记-清除
  - 标记阶段：从GCroot起遍历，找到可达对象并在对象头记录
  - 清除阶段：堆内存空间进行线性遍历，如果发现对象头无记录就清除
  - 优点-思路简单，不用移动对象
  - 缺点-效率低，速度慢，存在内存碎片
- 标记-整理
  - 标记阶段：从GCroot起遍历，找到可达对象并在对象头记录
  - 整理阶段：将所有存活对象移动到内存的一端，最后清理所有空间
  - 优点-不需要额外内存空间，无内存碎片
  - 缺点- 浪费时间（标记-清除-移动），也需要修改栈帧
- 复制算法
  - 将内存空间分为两块，每次只使用一块，在进行垃圾回收时将可达对象复制到另外一块没有被使用的内存中，再清除当前内存块中所有对象，后续按同样流程两个内存块交换来
  - 优点- 效率高，不会出现内存碎片
  - 缺点
    - 需要更多内存
    - 复制后对象的内存地址变化，需要额外时间修改栈帧中的引用地址
    - 可达对象多，垃圾对象少时，效率低

为什么要采用分代采集

- 不同对象存活时间不一样，可以针对不同对象采取不同算法

-XX：+PrintCommandLineFlags 查看使用的垃圾回收器

垃圾回收器

新生代和老年代的垃圾回收器有何区别

- 新生代收集器有Serial、ParNew、Parallel Scavenge；
- 老年代收集器有Serial Old、Parallel Old、CMS。
- 整堆收集器有G1、ZGC

常见的垃圾回收器

- 串行垃圾回收器
  - Serial GC
    - Serial是单线程的串行垃圾回收器，主要采用标记-复制算法进行垃圾回收
  - Serial Old
    - Serial Old是Serial的老年代版本，也是个单线程收集器，适用于老年代，使用的是标记-整理算法
- **并行**垃圾回收器
  - Parallel Scavenge
    - Parallel Scavenge 也是一个新生代的垃圾回收器，和ParNew一样，他也是多线程并行执行的，标记复制
  - Parallel Old
    - Parallel 是 Parallel Scavenge的老年代版本，同样是一个关注吞吐量的并行垃圾收集器，标记整理
  - ParNew
    - ParNew其实就是Serial的多线程版本，用标记-复制算法回收
  - 并行回收器主要的关注目标是吞吐量
- **并发**标记扫描垃圾回收器（CMS ）
  - 并发回收器主要关注的目标是STW的时长
- G1垃圾回收器
- ZGC垃圾回收器

垃圾回收器

CMS垃圾回收器是如何工作的（标记清理）

- 指定CMS垃圾回收器：-XX:+UseConcMarkSweepGC
- 特点
  - 短暂停，STW时间变短，但步骤变长
  - STW是用户暂停时间，可以提升用户体验
  - 但吞吐量更低
- 过程
  - 初始标记
    - 暂停所有工作线程，STW
  - 并发标记
    - 标记出所有可达对象，无需STW
  - 重新标记
    - 修正上阶段的偏差，STW但很短
  - 并发清理
    - 删除垃圾对象，无需移动对象，无需STW
  - 并发重置
    - 重置CMS内部结构，无需STW
- 可能存在的问题
  - 在并发标记，并发清理过程中，因用户线程同时进行，此时如果有新对象要进入老年代但空间不足够，会导致concurrent mode failure
    - 此时会采用serial old做一次垃圾收集，做一次全局STW
  - 并发清理可能产生新的垃圾
    - 下一次GC清理
  - 采用标记清除算法会有内存碎片
    - -XX:+UseCMSCompactAtFullCollection，启用该选项后会在执行FULL GC时执行碎片清理。
    - -XX: CMSFULLGCsBeforeCompaction，指定多少次GC后整理

G1垃圾回收器如何工作（分代标记-复制）

- 将堆的内存分成2048个Region，每个大小 = 堆内存 / 2048，也分了Eden，S0，S1，老年代，只是空间可以不连续
- Region类型
  - Eden
  - Survivor
  - Old
  - Humongous（专门放大对象，即超过region的%50）
- 过程
  - 初始标记
    - 暂停所有工作线程，STW
  - 并发标记
    - 标记出所有可达对象，无需STW
  - 重新标记
    - 修正上阶段的偏差，STW但很短
  - 筛选回收
    - 需要STW来清除对象
    - 可通过-XX：MaxGCPauseMills指定GC的STW时间
    - 可能不会回收所有对象
    - 采用复制算法不会产生碎片
- 主要回收方法
  - YoungGC
    - Eden区满触发YongGC
  - MixedGC
    - 老年占用率到了 -XX：InitiatingHeapOccupancyPercent指定的百分比，回收所有新生代和部分老年代以及大对象区
  - Full GC
    - 在进行MixedGC过程中，采用复制算法，如果复制过程中内存不够就会触发FullGC，STW并使用单线程标记整理算法进行GC。

什么是三色标记算法

- 三色标记算法是一种JVM中垃圾标记的算法，他可以减少JVM在GC过程中的STW时长，他是CMS、G1等垃圾收集器中主要使用的标记算法
  - 三色标记法的标记过程可以分为三个阶段：初始标记（Initial Marking）、并发标记（Concurrent Marking）和重新标记（Remark）

G1和CMS有什么区别

特性	CMS	G1
回收位置	老年代	整堆
GC算法	标记-清除算法	标记-复制算法回收年轻代 标记-整理算法回收老年代
垃圾识别算法	三色标记法——增量更新解决漏标	三色标记法——原始快照解决漏标
碎片产生	存在内存碎片	可防止内存碎片产生
可预测性	无法预测	G1的STW时长可预测
堆内存基本要求	一般要求不高	4G以上
自适应调优	不支持	支持

对象的分配过程

- new 的对象先放在伊甸园区，此区有大小限制
- 当伊甸园的空间填满时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收（Minor GC），将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
- 然后将伊甸园中的剩余对象移动到幸存者 0 区
- 如果再次触发垃圾回收，此时上次幸存下来的放到幸存者 0 区，如果没有回收，就会放到幸存者 1 区
- 如果再次经历垃圾回收，此时会重新放回幸存者 0 区，接着再去幸存者 1 区
- 什么时候才会去养老区呢？ 默认是 15 次回收标记
- 在养老区，相对悠闲。当养老区内存不足时，再次触发 Major GC，进行养老区的内存清理
- 若养老区执行了 Major GC 之后发现依然无法进行对象的保存，就会产生 OOM 异常

JVM调优（每天100w次登录请求，4核8G机器如何做）

1. 确定调优领域：是否需要调优，尝试年轻代调优，一般不会直接考虑老年代调优
   - 如果没有高峰期，虽然100万听上去挺多的，但是其实平均下来一秒钟的QPS也就10，这个量的话，其实根本不需要做什么特别的JVM优化
   - 假设登录业务存在高峰期，峰值时长大概持续1个小时，峰值的QPS可以达到200。那么需要做哪些优化
   - 堆内存设置
     - 我们一般建议都是把JVM的堆内存设置成操作系统内存的一半，也就是4G
2. 依据机器内存，机器中其它任务情况确认采用垃圾回收期，>4g可以考虑G1
   - 在新生代的垃圾收集器中，主要以Serial、ParNew、Parallel Scavenge以及支持整堆回收的G1了。
   - 因为新生代采用的都是复制算法，所以不太需要考虑碎片的问题，我们主要考虑吞吐量和STW的时长就行了。
   - 首先排除单线程的Serial，剩下ParNew是一个并发的收集器，Parallel Scavenge更加关注吞吐量，而G1作为JDK 9中默认垃圾收集器，他不仅同时具有低暂停时间和高吞吐量的优点，但是他对内存有要求，最小要4G，
   - 从使用门槛上来说，G1是可以用的，因为一般来说，内存要大于等于4G的话，才适合使用G1进行GC。
3. 各个区域大小设置
   - 各区大小设置
     - G1的内存划分是自适应的，它会根据堆的大小和使用情况来动态调整各个区域的大小和比例。但是，我们也可以通过一些JVM参数来手动设置G1的各个分代内存配置。
     - G1 中的分代和其他垃圾回收器不太一样，它不是严格按照年轻代和老年代划分的，而是通过划分各个区域的存活对象数量来实现垃圾回收的。因此，G1 中不需要像其他垃圾回收器那样设置新生代和老年代的大小比例，而是需要设置一些区域的内存配置。
     - -XX:G1NewSizePercent 和 -XX:G1MaxNewSizePercent分别用于设置年轻代的初始大小和最大大小，它们的默认值分别为 5% 和 60%。针对我们的业务场景，我们其实可以适当的调高一下年轻代的初始大小，5%的比例太小了，我们可以调整到30%。
4. 添加必要的日志：添加参数记录gc完整信息，堆内存溢出时输出dump日志，比如
   - -XX:MaxGCPauseMillis=100：最大 GC 暂停时间为 100 毫秒，可以根据实际情况调整；
   - -XX:+HeapDumpOnOutOfMemoryError：当出现内存溢出时，自动生成堆内存快照文件；
   - -XX: HeapDumpPath=/path/to/heap/dump/file.hprof：堆内存快照文件的存储路径；
   - -XX:+PrintGC：输出 GC 信息；
   - -XX:+PrintGCDateStamps：输出 GC 发生时间；
   - -XX:+PrintGCTimeStamps：输出 GC 发生时 JVM 的运行时间；
   - -XX:+PrintGCDetails：输出 GC 的详细信息；
   - -Xloggc:gc*:file=/path/to/gc.log:time,uptime:filecount=10,filesize=100M：将 GC 日志输出到指定文件中，可以根据需要调整日志文件路径、数量和大小