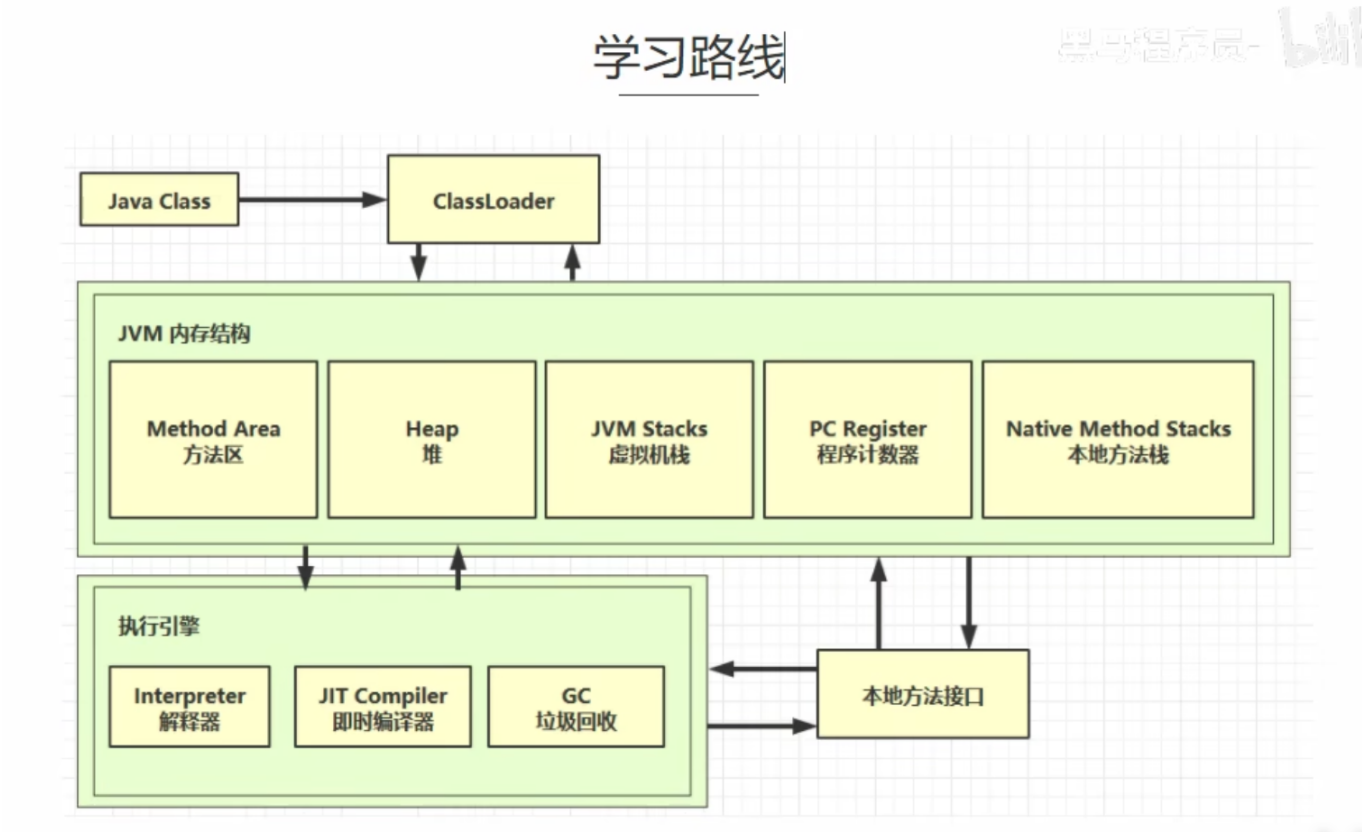
JVM

JVM 知识点：

- jvm的组成
- java类的加载过程，类加载的策略（双亲委派）
- jvm内存布局
- java垃圾回收，如何确定类可回收、垃圾回收算法、垃圾回收器、垃圾回收器能收集的区域
- jvm内存模型，变量的可见性、happens before规则、synchronized的实现原理、锁膨胀的机制
- jdk中自带的jvm监控、诊断工具
- jvm的问题诊断（内存泄漏、OOM、死锁、死循环、创建大量线程等、频繁GC、逃逸分析）、调优(调优的目的，优化内存占用、提高响应时间、提高吞吐量)等

学习路线



JVM 基础概念及调优

1. JVM组成

> 类加载子系统 加载字节码文件放入方法区（元空间） （loading(加载) -> linking（链接） ->Intialization（初始化）） 运行时数据区（方法区、java堆、java栈（线程独享）、本地方法栈、程序计数器（线程独享）） 字节码执行引擎 执行字节码、写程序计数器、执行垃圾收集线程进行垃圾回收

java类加载机制

java类加载机制就是将java类字节码加载到内存中，并转化为运行时数据结构，主要涉及到三个步骤：加载、连接、初始化

类加载阶段：

- 加载loading： 查找字节码文件，将其载入到内存中，这一阶段就是将.class文件字节流加载到jvm中
- 连接（包含以下三个步骤）：
    - 验证verification：对字节码进行验证，确保其符合jvm规范，不会威胁jvm
    - 准备preparation：这个阶段为jvm的类的静态变量分配内存空间，为其初始化默认值
    - 解析Resolution: 将符号引用替换为直接引用的过程
    - 符号引用就是一个类中（当然不仅是类，还包括类的其他部分，比如方法，字段等），引入了其他的类，可是JVM并不知道引入的其他类在哪里，所以就用唯一符号来代替，等到类加载器去解析的时候，就把符号引用换成那个引用类的地址，这个地址也就是直接引用
- 初始化Initialization：对类的静态变量进行初始化赋值，执行静态代码块。类被真正使用之前的最后一道屏障

使用final static修饰的变量，如果是基本数据类型或String类型，会在链接的准备阶段进行赋值，当为其他引用类型时，在初始化阶段的`<clinit>`中进行赋值

类初始化的触发时机：

- 虚拟机启动，触发用户指定的主类
- 遇到new指令时，触发初始化
- 调用静态方法的指令时，初始化静态方法所在的类
- 遇到访问静态字段的指令时，初始化改静态字段所在的类
- 子类初始化会触发父类初始化
- 使用反射API对某个类进行反射调用时，初始化该类

类加载器的种类：

- 引导类加载器，负责加载支持JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar,charsets.jar等
- 扩展类加载器：负责加载支撑JVM运行的位于jre的lib目录下的ext扩展目录中的jar包
- 应用程序类加载器：负责加载ClassPath路径下的类包，主要是加载自己编写的应用程序类
- 自定义加载器：负责加载用户自定义路径下的类包

jdk9之后引入模块化的概念，扩展类加载器改名为平台类加载器，java.base等由启动类加载器加载，其他模块都由平台类加载器加载

双亲委派模式:

当一个类加载器收到加载类的请求时，优先将这个请求委派给父类加载器处理，当父类加载器无法处理时子类加载器才会去尝试加载类，这种模型保证类的一致性，避免类的重复加载

双亲委派模式的破坏： 破坏双亲委派模式，以提供动态更新某个类的能力

类加载机制与OOP设计原则 类加载机制与面向对象设计原则中的开闭原则（对扩展开放，对修改关闭）有着紧密的联系。通过合理利用类加载机制，我们可以使系统更容易扩展，遵循OOP设计原则，降低系统的耦合度。

类加载机制的安全性考虑 由于类加载机制直接涉及字节码的加载和执行，因此在设计和使用自定义类加载器时，需要特别注意安全性问题 防止恶意代码注入、确保类加载的一致性和合法性是保障系统安全性的重要环

节

Java模块化与类加载机制 随着Java 9的推出，引入了模块化系统，这对类加载机制带来了一些新的变化。模块化系统通过模块路径（Module Path）的概念，更好地隔离了不同模块的类。这对于大型项目的架构和维护带来了便利。

jvm运行时数据区

2. 程序计数器

- java的二进制字节码（即JVM指令，所有平台都是一致的）无法被操作系统直接执行，需要经过java的解释器解释成机器码，然后才能由CPU执行
- 作用：程序计数器会记住下一条jvm指令的执行地址，通过寄存器实现
- 特点
    - 线程私有
    - 不会存在内存溢出
- 每个线程都会分配一个程序计数器，存储线程当前执行的位置，当线程出现切换时，程序计数器用于记住当前先线程执行的地址，一旦指令执行，程序计数器将更新到下一条指令，另外在线程恢复执行时指示当前线程执行的位置，程序计数器的值是由字节码执行引擎来修改的，用来保存当前线程执行指令的地址

2. java栈（先进后出）

存放局部变量数据，分配给线程使用。一个方法对应一块栈帧内存空间 栈帧结构

> 局部变量 方法中的局部变量 操作数栈 临时存放JVM指令操作数据的栈 动态链接 符号引用转化为直接引用 方法出口 方法返回的位置 另外栈中的引用对象的真实地址在堆中 栈大小设置(-Xss 2MB) linux 下默认是1024KB 栈溢出 出现没有出口的递归(栈帧过多)，抛出StackOverflowError 方法的执行超出了栈的大小（栈帧过大），抛出StackOverflowError

线程安全问题： 方法内的局部变量如果没有出现逃逸的情况不会有线程安全的问题

本地方法栈（native method stacks）

- 为本地方法执行提供内存空间
- 线程私有

4. 元空间（方法区，jdk1.7之前存在，hotspot 1.8之后是元空间，使用的是操作系统的本地内存）--存储类相关的信息

- 存放类元信息，方法元信息，常量池
- 线程共享
- 使用系统的直接内存，随着需要会进行扩容 虚拟机启动时创建 存放类的元数据信息、运行时常量值、静态变量（可能存在对象，在方法中存放的也是变量的引用，真正的变量放在堆中）、字段和方法数据，以及方法和构造函数的代码，包括用于类和实例初始化及接口初始化的特殊方法
- 运行时常量池 Stringtable，不在操作系统本地内存中，在堆（heap）中

- 设置的元空间大小(jdk8，默认没有大小限制) -XX:MaxMetaspaceSize=100M - XX: -UseCompressedOops 抛出OutOfMemoryError:metaspace(jdk1.6是 PermGen space)
- 原空间默认使用的是本地内存
- 方法区溢出，OutOfMemoryError

chatgpt的方法区的回答： Java 8 中的方法区（Method Area）主要包含以下内容：

- 已加载的类信息：包括类的全限定名、访问修饰符、父类信息、接口信息等。
- 常量池（Constant Pool）：存放编译期生成的字面量和符号引用，供 JVM 在运行时使用。
- 字段信息：包括字段的名称、类型、访问修饰符等。
- 方法信息：包括方法的名称、参数列表、返回值类型、访问修饰符等。
- 运行时常量池（Runtime Constant Pool）：为每个类和接口创建一个运行时常量池，用于存放运行时生成的常量。
- 动态生成的代理类信息：如通过 Java 动态代理创建的代理类。
- 其他一些与类加载、动态代理、反射等相关的数据结构和信息。 需要注意的是，方法区和堆一样，也是 JVM 运行时数据区的一部分，但它所包含的内容与堆有很大不同。方法区通常是 JVM 进程内存中最小的一块区域，但是如果应用程序中使用了大量反射、动态代理等技术，方法区的空间可能会很快被耗尽。在 Java 8 中，方法区已经被移除，取而代之的是元空间（Metaspace），元空间的实现不再使用固定大小的方法区，而是使用本地内存，可以动态地调整大小

常量池

二进制字节码的组成： 类基本信息，常量池，类的方法定义，包含虚拟机的指令 常量池的作用：为指令提供常量服务，以查表的方式查找到类名、方法名、参数类型、字面量等信息 常量池是.class文件中的，当该类被加载时，它的常量池信息会放入运行时常量池，并将里面的符号地址变为真实的地址 使用javap -v HelloWorld.class 查看类反编译后的详细信息

串池StringTable 运行时会将字符串常量放入StringTable中，运行时是懒加载的，只有用到时才会将串放入串池

```
//字符串会放入StringTable串池中，在常量池中
String a = "a";
String b = "b";

String ab = "ab";
//创建c时首先会创建一个StringBuilder类并进行初始化，然后将a、b append到StringBuilder中，然后调用StringBuilder的toString方法，toString方法实际上是又创建了一个新的String对象,这时c是放在堆中的
String c = a +b;
//返回false，因为c在堆中，而ab在常量池中
c==ab
//会直接使用常量池中的“ab”串，是javac在编译期的优化，结果在编译期确定为“ab”
String d = "a"+ "b";
//返回true，都在常量池中
ab == d;
```

特性：

- 常量池中的字符串仅是符号，第一次用到时才会变为对象
- 利用串池的机制，来避免重复创建字符串对象
- 字符串变量拼接的原理是StringBuilder（jdk1.8）
- 字符串常量拼接的原理是编译期优化
- 可以使用intern方法，主动将串池中还没有的字符串放入常量池中 jdk 1.8 尝试将字符串对象放入串池，如果没有则放入，如果有则不放入，然后将串池中的对象返回 jdk1.6 在串池中不存在串时会将字符串对象复制到串池中，

```
String s = new String("a") + new String("b");
//返回false，s在堆中，而StringTable中没有“ab”
s == "ab"
//jdk1.8将字符串对象尝试放入串池中，如果有就不放入，没有则放入（不入池则s还是在堆中），同时将串池的对象返回，这时s已经在串池中
//jdk1.6将会拷贝字符串到常量池中，s还是在堆中
s2 = s.intern();
//StringTable中已经存在“ab”，返回true
s2 == "ab"
//s还在堆中，返回false
s == "ab"
```

- jdk7之前StringTable是放在永久代中，在jdk8中StringTable是放在堆中的，放在堆中是为了保证在垃圾回收时StringTable能及时被垃圾回收，而不需要在full GC时才被回收
- 在jdk8中配置垃圾回收规则： -XX：+UseGCOverheadLimit （默认开启，可以使用-XX：-UseGCOverheadLimit关闭） 当jvm大量的时间花在垃圾回收（98%的时间）而只有少于2%的堆内存被垃圾回收，则jvm认为程序异常，jvm进程退出
- StringTable是可以被垃圾回收的
- StringTable是类似于hashTable的实现
- StringTable调优，调整HashTable的桶的个数， -XX:StringTableSize=20000 -XX:+PrintStringTableStatistics,当字符串数量太多时，可以将StringTable的个数调高点
- 将堆内存中的String对象放入到StringTable中避免大量的内存占用 （类似于设计模式中的享元模式）

配置jvm打印垃圾回收的详情： -Xmx10m -XX:+PrintStringTableStatistics -XX:+PrintGCDetails -verbose:gc

直接内存

- 常见于NIO操作，用于数据缓冲区
- 分配回收成本较高，读写性能高
- 不受JVM内存回收管理
- NIO读写（使用直接内存）时效率高的原因：
    - 直接在操作系统的内存中划出一块内存，jvm可以直接访问，磁盘文件读取数据到内存时避免从操作系统内存向jvm内存中进行拷贝的操作，提高读写性能

- 直接内存会有内存溢出异常：OutOfMemoryError:Direct buffer memory
- 直接内存释放回收的原理（不是使用java的gc来进行释放的）：

```
//分配直接内存，获取到内存分配的地址
  long base=unsafe.allocateMemory(_1GB);
  unsafe.setMemory(base,_1GB,(byte)0);
  //对内存进行释放
  unsafe.freeMemory(base);
```

- ByteBuffer原理：
  - 使用unsafe.allocateMemory(size)方法分配内存
  - 使用unsafe.freeMemory(address);回收内存（当ByteBuffer被垃圾回收收集时，出发unsafe.freeMemory(address)）

显示的禁用GC - XX:DisableExplicitGC ， 可以使System.gc()失效，这是由于不会显示出发jvm 的GC，可能导致直接内存的垃圾回收的失效，此时直接内存必须要等到jvm自动的垃圾回收才能回收 如果禁用了显式的垃圾回收，可以使用直接使用unsafe来对直接内存进行垃圾回收

堆

- 存放对象的内存区域
- 线程共享的一块内存区域，虚拟机启动时创建，垃圾回收器管理的主要区域 新生代 分为一个eden区、两个survivor区

> eden 区，对象出生的区域 survivor 经过minior gc之后存活的类会从eden区移动到survivor区 老年代 存放超过某一年龄代的对象，或是大对象直接分配到老年代 设置初始堆 -Xms 设置最大堆 -Xmx 堆内存溢出OutOfMemoryError 死循环导致堆内存溢出 内存大小不够导致内存溢出

JVM诊断工具(jdk自带的一些常用的诊断工具) jps 查看java进程 `jps 【选项 】 [hostid]` jmap -heap pid 检测某一时刻的堆内存使用情况 jmap -dump:format=b,live,file=test1.bin pid --抓取堆内存dump，抓取前进行垃圾回收，堆内存保存格式是二进制，保存文件名为test1.bin jconsole 图形化工具，查看堆内存占用，线程数量，类加载数量，cpu占用率 jstat 统计jvm内存信息 `jstat 【选项】 【进程ID】 [间隔时间 ] [查询次数]` jstat -gc -h3 2114 250 10 -gc 参数的输出内容 S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 S0C S1C S0U S1U EC EU OC OU MC

MU CCSC CCSU YGC YGCT FGC FGCT GCT 512.0 512.0 256.0 0.0 50688.0 12184.0 529920.0 476480.1 170240.0 163104.4 16896.0 15220.7 1978 19.427 31 18.441 37.868 S0C: 当前幸存者区0的容量 (kB). S1C: 当前幸存者区1的容量(kB). S0U: 幸存者区0已用内存 (kB). S1U: 幸存者区1已用内存 (kB). EC: 伊甸园区容量 (kB). EU: 伊甸园区已用内存 (kB). OC: 当前老旧区容量 (kB). OU: 老旧区已用内存 (kB). MC: 元数据区容量 (kB). MU: 元数据区已用内存 (kB). CCSC: 类压缩区容量 (kB). CCSU: 类压缩区已用内存 (kB). YGC: 新生垃圾回收事件数量. YGCT: 新生垃圾回收时间. FGC: 垃圾回收事件总和. FGCT: 完整的一次垃圾回收时间. GCT: 所有的垃圾回收时间.

jinfo JVM参数查看和修改 `jinfo 【选项】【具体选项参数名】【进程ID】` jstack JVM线程信息监控 `jstack [ 选项 ] [进程ID]远程IP`

VisualVM 通过安装各种插件实现对jvm的各种运行状态的实时监控 可以通过dump内存快照（堆转储）查看大对象占用内存的情况

arthas 阿里提供的jdk诊断工具，涵盖了几乎所有的jdk自带的诊断工具，同时可以很方便的查看死锁线程、修改应用中的某个类代码并动态编译和加载类到JVM

- 启动： java -jar arthas-boot.jar
- 查看全局信息（实时刷新）： dashboard
- 查看线程状态： thread
- 查看某一线程信息： tread 14

java 内存模型JMM

关于对象的可见性，遵循happens-brefore原则 "Happens before"原则是并发计算中的一个概念，用于描述事件之间的顺序关系。它是指在多线程或多进程的计算环境中，如果事件 A 在时间上先于事件 B 发生，那么事件 A 就被认为是在事件 B 之前发生的。 happends before 是一种内存可见性模型 多线程下由于指令重排序导致数据可见性的问题 A线程修改共享变量的值可能对B线程不可见 JMM通过happends beofore提供了一个跨越线程的可见性的保障，如果操作A在时间上必然先于B发生，则A就认为在B之前发生 happends before不表示指令执行的先后顺序，只要对最终的结果没有影响，jvm是允许指令重排序的

垃圾回收

如何判定对象可以回收 -- 引用计数法、可达性分析

引用计数

当出现A、B两个对象相互之间引用，则A、B的引用计数无法归0，则A、B两个对象都无法回收

8.可达性分析

- GC Root根节点 --肯定无法被垃圾回收的对象，线程本地变量、静态变量本地方法栈变量
- 从GC root开始搜索，搜索到的对象为非垃圾对象、未搜索到的为垃圾对象，全部回收 堆分析工具 MAT （eclipise提供的堆内存分析工具）
- 四类GC root对象
    - System Class 系统类
    - Native Stack jvm调用操作系统方法时引用的一些java对象
    - Busy monitor 被加锁的对象
    - Thread 活动线程中引用的局部变量

四种引用

- 强引用 一般通过new 的方式创建的对象是强引用对象，不可被回收
- 软引用（SoftReference） - 没有被强引用对象引用的对象，可能在发生gc时如果回收后内存还是不够则会将软引用对象回收
    - 软引用对象（这里指SoftReference本身）的清理，需要配合使用引用队列ReferenceQueue来回收
    - 当软引用所引用的对象被回收时，软引用对象会被放入ReferenceQueue，从queue中取出软引用然后置空移除
- 弱引用(WeakReference) - 没有被强引用对象引用的对象，在发生gc时直接将弱引用对象回收
    - 和软引用对象类似，需要使用ReferenceQueue来回收WeakReference本身
- 虚引用 - 创建ByteBuffer时会使用Cleaner分配直接内存地址 --配合引用队列使用
- 终结器引用 - 对象重写finalize（）方法时，在进行垃圾回收时会自动调用finalize（）方法
    - 通过引用队列的配合使用可以将软引用、弱引用、虚引用、终结器引用对象进行回收

1. 强引用
    - 只有所有 GC Roots 对象都不通过【强引用】引用该对象，该对象才能被垃圾回收
2. 软引用（SoftReference）
    - 仅有软引用引用该对象时，在垃圾回收后，内存仍不足时会再次出发垃圾回收，回收软引用对象
    - 可以配合引用队列来释放软引用自身
3. 弱引用（WeakReference）
    - 仅有弱引用引用该对象时，在垃圾回收时，无论内存是否充足，都会回收弱引用对象
    - 可以配合引用队列来释放弱引用自身
4. 虚引用（PhantomReference）
    - 必须配合引用队列使用，主要配合 ByteBuffer 使用，被引用对象回收时，会将虚引用入队，由 Reference Handler 线程调用虚引用相关方法释放直接内存
5. 终结器引用（FinalReference）
    - 无需手动编码，但其内部配合引用队列使用，在垃圾回收时，终结器引用入队（被引用对象暂时没有被回收），再由 Finalizer 线程通过终结器引用找到被引用对象并调用它的 finalize 方法，第二次 GC 时才能回收被引用对象

回收算法

标记清除

- 先标记无用对象，然后清除
- 回收速度快
- 会产生内存碎片

标记整理

- 先标记无用对象，将可用对象整理移动到内存的一端，将垃圾对象移除
- 不会产生内存碎片
- 需要移动对象，效率较低

复制算法

- 将内存区域划分为大小相等的from，to两个区域，首先标记from中没有被引用的对象，然后将from中的有用对象复制到to区域，然后清除from空间，交换from、to区域
- 在存活对象较少时效率高
- 需要占用双倍的内存
- 不会有内存碎片

在jvm中三种回收算法都有使用

垃圾回收测试

```
/**
 * @Description: 垃圾回收测试
 * @CreateDate: Created in 2023/5/8 18:53
 * @Author: lijie3
 */
public class TestDemo1 {

    private static final int _7MB = 7*1024*1024;
    private static final int _8MB = 8*1024*1024;
    private static final int _512k = 512*1024;

    //-Xms20M -Xmx20M -Xmn10M -XX:+UseSerialGC -XX:+PrintGCDetails -
verbose:gc
    public static void main(String[] args) {
        ArrayList<byte[]> list = new ArrayList<>();
//        list.add(new byte[_7MB]);
//        list.add(new byte[_512k]);
//        list.add(new byte[_512k]);
        //直接进入老年代
        list.add(new byte[_8MB]);
        //heap内存溢出
        list.add(new byte[_8MB]);
    }
}
```

分代垃圾回收

- 将堆内存划分为新生代、老年代
- 新生代：存放朝生夕死的对象，对象的创建大多发生在新生代
    - 分为Eden、from、to区域
    - 一般对象都会在Eden中创建
    - 当Eden内存不够用，就会触发minor GC，使用复制算法将eden中的存活对象复制到to区域，并将存活对象的年龄加1，清空Eden，并交换from、to
    - 当触发了第二次Minor GC，会将eden、from中存活对象复制到to区域，交换from和to
    - 当from、to中的对象年龄超过某个值（默认15，记录在对象头中，使用4bit记录，最大为15），则将对象移动到老年代
    - 在minor gc时，会触发stop the world，暂停其他用户线程
- 老年代：长时间存活的对象
    - 当新生代from、to区域在进行垃圾回收to区域无法完全存放存活对象时，会将对象移动到老年代

- 当老年代空间不足，先尝试进行minor GC，如果minor GC后空间仍然不足，则触发FUll GC
- full gc也会引发stop the world，minor gc的stop the world会比full gc更长

大对象直接晋升到老年代 当新生代的内存已经不足的情况下，大对象创建时会直接放入老年代

一些参数配置解释

参数	含义
-Xms	初始堆大小
-Xmx	堆最大大小
-Xmn	新生代大小
-XX:InitialSurvivorRatio=ratio -XX:+UseAdaptiveSizePolicy	eden、from、to大小比例
-XX:SurvivorRatio	幸存区比例,默认8，指的是eden的占比，8:1:1
-XX:MaxTenuringThreshold=threshold	晋升老年代阈值
-XX:+PrintTenuringDistribution	打印晋升详情
-XX:PrintGCDetails -verbose:gc	gc详情
-XX:+ScavengeBeforeFullGC	FullGC 前minor GC

9. 一个对象在堆中的生命流转周期（一般情况下）

> 1. 对象创建以后放在新生代的eden区
> 2. 当eden区的空间无法再分配内存给新的对象时，触发一次minior GC，通过可达性分析判断eden区的对象是否被引用，如果不被引用，则直接回收，如果被引用，放入survivor区，对象的分代年龄加1，eden区再次分配空间给新的对象
> 3. 当eden区再次无法分配内存给新的对象，再次触发minior GC，通过可达性分析判断eden区和已经分配了对象的一个survivor区的可回收对象，对非垃圾对象的分代年龄再次加1，放入到另一个空闲的survivor区
> 4. 当内存不足时重复步骤3
> 5. 当某次minior gc 回收垃圾后空闲的survivor无法存放下存活的对象，则需要进行Major GC（Major GC的效率比minior GC慢很多），将对象放入老年代
> 6. 动态年龄判断：当某次minior GC之后，存活对象的总大小大于一块survivor 区域大小的50%，则直接将这次minior GC存活下来的对象直接放入老年代中 分代GC的理论基础：大多数的对象都是朝生夕死
> 7. 当老年代无法存放更多的对象时，会进行一次full gc
> 8. 当进行了full gc后任然无法存放新产生的对象，则JVM会抛出OOM异常

16.垃圾收集器

分类：

- 串行：单线程、适合堆内存小的场景

- 吞吐量优先：多线程，适合堆内存较大的场景，需要多核心cpu支持，让单位时间内stop the world的时间最短（并行收集，暂停用户线程，开启多个垃圾回收线程） 吞吐量计算： 用户线程执行时间/（用户线程执行时间+ 垃圾回收线程执行时间）
- 响应时间优先：多线程，适合堆内存较大的场景，需要多核心cpu支持，尽可能降低单次垃圾回收暂停时间（stop the world）（并发收集，做到最小的用户线程暂停时间）

串行垃圾回收器

开启： -XX:UseSerialGC = Serial(复制算法) + SerialOld（标记整理算法） 单线程回收，当垃圾回收线程执行时，要求用户线程阻塞 适用于单核CPU的情况

吞吐量优先的垃圾回收器

- 并行垃圾回收器
- 开启 ： -XX:+UseParallelGC (新生代，复制算法) -XX:+UseParallelOldGC （老年代，标记整理算法）
- 其他配置
  - -XX:ParallelGCThreads=10,指定垃圾回收的线程数量
  - -XX:+UseAdaptiveSizePolicy 自适应的调整新生代的比例大小
  - -XX:GCTimeRatio= ratio 调整吞吐量， 1/1+ratio，如ratio=99，则单位时间100分钟内内只有1/99+1 = 0.01 1分钟的垃圾回收的时间
  - -XX:MaxGCPauseMills=50ms 调整每次垃圾回收stop world time的时间

响应时间优先垃圾回收器

- 并发的垃圾回收器，垃圾回收时用户线程允许与垃圾回收线程并发运行
- -XX:UseParNewGC（新生代、并行、复制算法） -XX:+UseConcMarkSweepGC（标记清除，老年代）
- 开启： -XX:+UseConcMarkSweepGC（标记清除，老年代） +XX:+UseParNewGC（新生代、并行、复制算法）/+UseSerialOld
- 其他配置
  - -XX:ParallelGCTHreads=n(并行垃圾回收线程数) -XX:ConcGCThreads=threads（并发垃圾回收线程数，一般设置为并行线程数的1/4）
  - -XX:CMSintialingOccupancyFraction=percent（80），例如老年代的使用百分比达到80%时进行垃圾回收 执行cms垃圾回收的内存占比，主要是为了避免在并发清理过程中产生的新的垃圾导致的内存溢出问题
  - -XX:+CMSScavengeBeforeRemark 重新标记之前首先对新生代进行一次垃圾回收，减少重新标记阶段的压力
- 由于使用标记清除算法，可能会导致内存碎片过多，当碎片过多时，会退化为SerialOld垃圾回收，会导致垃圾回收的时间大大增加

32.CMS垃圾收集器收集过程--考虑尽量降低延迟的垃圾回收器

- cms是老年代的垃圾收集器

> 并发的垃圾收集器 实现了让垃圾收集器与用户线程基本上能同时工作 使用标记-清除算法

步骤：

- 初始标记：暂停所有其他线程，记录GC roots直接引用的对象（速度很快）

- 并发标记：从GC roots直接关联的对象开始遍历整个对象的过程，耗时较长，用户线程不需要停止，可能导致已标记对象发生变化
- 重新标记：修正并发标记期间因为用户程序继续运行导致标记变动的一部分对象标记记录，停止用户线程（解决少标记的问题，即产生了新的对象（不能被回收）没有被标记的问题）
- 并发清理：开启用户线程，GC线程开始对未标记区域清理，这一阶段新增的对象会被标记为不做任何处理的对象
- 并发重置：重置本次GC过程的标记数据

G1垃圾回收器 --优先收集垃圾最多的区域

- 同时注重吞吐量和低延迟，默认暂停目标是200ms
- 超大堆内存，将堆划分为多个大小相等的region，每个region都可以作为eden、from、to以及老年代区域
- 整体上是标记+整理算法，两个区域之间是复制算法
- 启动： -XX:+UseG1GC
- 配置region大小 -XX:G1HeapRegionSize=size 1,2,4,8等值
- 最大暂停时间 -XX:MaxGCPauseMills=200ms

回收阶段，从上到下依次循环

- young collection 新生代收集，会stop the world
- young collection + concurrent mark，新生代垃圾回收、初始标记（young GC会进行）、并发标记（老年代占用堆内存的比例达到阈值进行并发标记，不会stop the world）
- Mixed collection，对eden，from、to，老年代进行全面的垃圾回收，最终标记会stop the world，拷贝存活 也会stop the world，根据MaxGCPauseMills配置从老年代中选择回收价值最高的区域进行回收，达到小于最大暂停时间的目标

跨代引用 新生代回收的跨代引用，当老年代引用了新生代的对象，在老年代使用卡表记录哪些对象引用了新生带

jdk8字符串去重

- 让两个字符串引用同一个char数组（与String.intern不同，intern使用的是StringTable实现节省内存）
- 开启 -XX:UseStringDeduplication （默认是打开的）
- 优点：节省大量内存
- 缺点：略多占用cpu时间，新生代回收时间略微增加

jdk8并发标记类卸载 在并发标记后，能知道哪些类不再被使用，当一个类加载器的所有类都不在使用，则卸载它所加载的所有类 开启（默认是开启的）-XX:+ClassUnloadingWithConcurrentMark

jdk8巨型对象

- 一个对象大于Region的一半时，称为巨型对象
- G1不会对巨型对象进行拷贝
- 回收时会优先考虑

jdk9并发标记起始时间调整

- 并发标记必须在堆空间占满之前完成，否则会退化为FULLGC
- jdk9之前使用 -XX:InitiatingHeapOccupancyPercent = 45 配置

- jdk9可以动态调整：
  - XX:InitiatingHeapOccupancyPercent = 45 设置初始值
  - jvm动态调整垃圾回收的时机

windows下查看虚拟机运行参数 -XX:PrintFlagsFinal -version|findstr GC

关于full gc的说明

- SerialGC、parallel GC、CMS GC 新生代不足发生的垃圾收集叫minor gc
- SerialGC、parallel GC 老年代内存不足发生的垃圾回收称为full gc
- CMS、G1 老年代垃圾回收不足，由于存在并发垃圾收集，当垃圾产生速度小于垃圾收集速度，暂停时间较短，不能称为fullGC，当垃圾产生速度大于收集速度，会退化为SerialOld，会出现full GC

GC调优

考虑的方向

- 内存
- 锁竞争
- cpu占用
- IO占用

是要求低延迟（互联网项目，CMS、G1、ZGC）、还是高吞吐量（科学计算，ParallelGC）

- 查看fullgc后前后内存占用，考虑以下的问题：
  - 数据是否太多
  - 数据是否太过臃肿
  - 是否存在内存泄漏

新生代的垃圾回收时间比老年代更小，回收价值更高

新生代调优 *适当的调大新生代的空间，oracle官方推荐新生代的空间控制在25%～50%（设置-Xmn指定新生代大小） 设置参考： -Xmn = 并发量*（请求/响应）的数据 survivor设置参考：

- 长时间存活的对象尽快晋升： -XX:MaxTenuringThreshold=threshold
- 打印survior晋升信息： -XX:PrintTenuringDistribution

- 老年代调优
  - CMS的老年代内存越大越好
  - 先尝试不做调优，如果没有FullGC，则不需要对老年代调优
  - 观察查发生FUll GC的老年代占用，将老年代内存预设调大1/4～1/3
  - 参数设置(老年代占用多少时进行垃圾回收) -XX:CMSInitiatingOccupancyFraction=percent

jdk中自带的监控与诊断工具

jps 查看java进程信息

jps 查看进程id及启动类名 jps -l 查看进程id及主类的全路径名 jps -q 只输出lvmid,即进程id jps -m 输出传递给main方法的参数 jps -v 输出jvm启动时显示指定的jvm参数

jstat 打印目标java进程的性能数据

jstat -class 10385 查看jvm的类加载情况

Loaded Bytes Unloaded Bytes Time
15544 30503.0 0 0.0 3.13

jstat -compiler 打印即时编译相关的数据 Compiled Failed Invalid Time FailedType FailedMethod 11726 0 0 2.61 0

jstat -gc -t 16593 1s 4 --打印垃圾回收相关数据，1s打印一次，打印4次,-t参数显示当前jvm运行的总时长，用来与垃圾回收的时间进行对比，判断jvm的垃圾回收是否太过频繁，是否存在内存泄漏的问题 S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT
0.0 22528.0 0.0 22370.4 176128.0 45056.0 155648.0 99759.0 78336.0 77508.5 9920.0 9476.9 20 0.134 0 0.000 0.134 0.0 22528.0 0.0 22370.4 176128.0 45056.0 155648.0 99759.0 78336.0 77508.5 9920.0 9476.9 20 0.134 0 0.000 0.134 0.0 22528.0 0.0 22370.4 176128.0 45056.0 155648.0 99759.0 78336.0 77508.5 9920.0 9476.9 20 0.134 0 0.000 0.134 0.0 22528.0 0.0 22370.4 176128.0 45056.0 155648.0 99759.0 78336.0 77508.5 9920.0 9476.9 20 0.134 0 0.000 0.134

jmap 在堆内存占用出现持续增长、垃圾回收时间越来越长，越来越频繁时，可以将jvm的堆内存进行dump

jmap -clstats 打印被加载的类信息 jmap -histo 统计各个实例数据以及占用内存，按照占用内存从多到少的排序 jmap -dump 导出java虚拟机堆快照 jmap -dump:live,file=test.bin 16593 导出文件到test.bin中

jinfo 查看jvm进程的参数，如-X参数，-XX参数，以及在java代码中通过System.getProperty获取的-D参数

jinfo 16593

jstack 打印目标java进程中各个线程的栈轨迹，以及线程持有的锁

jstack 16593 jstack可以查看线程的状态、持有的锁、以及正在请求锁的状态，还可以分析具体的死锁代码

jstack jvm线程信息监控 查看线程信息，生成线程快照 jstack -F 68963 正常输出的请求不被响应时，强制输出线程堆栈 jstack -l 除堆栈外，会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观察锁持有情况 jstack -m 如果调用到本地方法，可现实c/c++堆栈

jstack -l 68963 如果存在死锁则会有如下输出：

Found one Java-level deadlock:

"mythread2": waiting for ownable synchronizer 0x000000076aea4848, (a java.util.concurrent.locks.ReentrantLock$NonfairSync), which is held by "mythread1" "mythread1": waiting for ownable synchronizer 0x000000076aea4878, (a java.util.concurrent.locks.ReentrantLock$NonfairSync), which is held by "mythread2"

Java stack information for the threads listed above:

Found 1 deadlock.

jstack -l 68963 输出信息 2023-10-29 15:01:17 Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.291-b10 mixed mode):

"HikariPool-1 housekeeper" #76 daemon prio=5 os_prio=31 tid=0x00007f9d127d4000 nid=0xbc0f waiting on condition [0x0000000309a81000] java.lang.Thread.State: TIMED_WAITING (parking) at sun.misc.Unsafe.park(Native Method) - parking to wait for <0x00000006c4dd64f8> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject) at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:215) at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2078) at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:1093) at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:809) at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1074) at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1134) at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624) at java.lang.Thread.run(Thread.java:748)

Locked ownable synchronizers: - None

"DestroyJavaVM" #65 prio=5 os_prio=31 tid=0x00007f9d1238d800 nid=0x1303 waiting on condition [0x0000000000000000] java.lang.Thread.State: RUNNABLE

Locked ownable synchronizers: - None

"http-nio-7001-Acceptor" #63 daemon prio=5 os_prio=31 tid=0x00007f9d1279f800 nid=0xb503 runnable [0x0000000309675000] java.lang.Thread.State: RUNNABLE at sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method) at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:424) at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:252) - locked <0x00000006c4ddb118> (a java.lang.Object) at org.apache.tomcat.util.net.NioEndpoint.serverSocketAccept(NioEndpoint.java:574) at org.apache.tomcat.util.net.NioEndpoint.serverSocketAccept(NioEndpoint.java:80) at org.apache.tomcat.util.net.Acceptor.run(Acceptor.java:106) at java.lang.Thread.run(Thread.java:748)

Locked ownable synchronizers: - None

jcmd 是以上工具（除jstat）的合集

jcmd -l 查看jvm进程信息 jcmd 16593 GC.heap_info 查看堆及元数据信息 jcmd 16593 VM.uptime 查看jvm已启动的时长

eclipse MAT 对jmap dump出来的二进制快照进行解析

10. arthas 阿里开源的jvm调优工具

11. Java虚拟机调优 -- 减少full gc的次数，减少STW（stop the world）

为什么会有stop the world：如果不停止用户线程，线程之行过程中可能会不断的出现引用的变化，类似于现实中你妈打扫垃圾的过程中你还在不断的丢垃圾

12. jvm参数设置 针对jdk8的版本，主要是对堆空间分代大小进行设置，对元空间大小进行设置，分代年龄进行设置等等

类加载模块

类文件结构（class文件）

javac 编译成class javac -parameters -d . TestDemo1.java 查看字节码文件 od -t xC com/steven/gc/TestDemo1.class

class结构：

- 魔数(0-3字节)：标识文件类型：cafebabe
- 版本（4-7字节）：00 00 00 34 00 20 0a 00 07 00 19 07
- 常量池
    - 8-9字节：常量池中有多少项
    - 后面的包含方法引用信息、返回返回值类型、构造方法
- 访问修饰与继承信息
- 成员变量数量、成员变量信息
- method信息
- 附加属性

字节码指令

- aload_0 将局部变量表中slot 0 加载（加载this）
- invokespecial 预备调用构造方法
- getstatic 加载静态变量到操作数栈
- ldc 加载参数
- invokevirtual 执行普通成员方法
- b1 返回

反编译字节码信息： javap -v com/steven/gc/TestDemo1.class

方法字节码执行流程

栈帧中的内容：操作数栈、局部变量表 静态代码块的执行方法,类的构造方法 实例的构造方法

多态原理

invokevirtual指令调用的方法，使用vtable（虚方法表）查找多态时具体要执行的方法地址，使用缓存加快多态执行方法的效率 HSDB：查看内存分布工具 java -cp ./lib/sa-jdi.jar sun.jvm.hotspot.HSDB

异常处理

使用exception table监测异常代码，当代码出现异常时，查表进行异常捕获（try catch等处理） 示例：Exception table: from to target type 13 20 23 Class java/io/IOException finally的字节码： 将finally中的代码对应的字节码放在try块和catch块的后面，保证finally块代码的执行 **在finally中不要写return指令，如果出现了异常，则异常会被吞掉（与实际代码相符合，finally中不会捕获异常）**

```
public class TestTryCatch {

    public static void main(String[] args) {
        System.out.println(test1());
    }

    public static int test1(){
        int a = 10;
        int b = 0;
        try{
            int c = a/b;
            return a;
        }finally {
            //不会抛出异常
            return b;
        }
    }
}
```

关于finally的面试题（）：在try中return 之前，执行finally语句之前，会将a的当前值10进行暂存，然后执行finally，最后将暂存的10返回

```
public class TestTryCatch2 {

    public static void main(String[] args) {
        System.out.println(test1());
    }

    public static int test1(){
        int a = 10;

        try{
            return a;
        }finally {
            //不会抛出异常
            a = 20;
        }
    }
}
```

synchrinozed原理

使用monitorenter、monitorexit进行加锁解锁

编译期处理

- 默认构造器：当无构造器时，使用的是super的构造方法
- 自动拆装箱： jdk5之后的版本支持，在编译期自动生成对应的字节码

- 泛型集合取值：自动进行了泛型擦除，添加list.add（Object），获取，list.get()返回object，然后将object转化为泛型指定的类型
- 局部变量的泛型信息没有被擦除，方法的返回值、参数泛型类型是可以获取到的
- 可变参数列表
- hashcode：提高比较效率，equals和hashcode组合在HashMap中进行等值判断
- 枚举类字节码实现
- try-with-resources
- 子类重写父类方法重写允许子类方法返回值类型是父类返回值类型的子类
- 内部类的实现，匿名内部类引用外部变量必须是final类型

类加载

加载阶段

- 加载：将类的字节码载入方法区中，在堆中创建对应的class对象
- 链接：与加载交替运行
    - 验证：验证类是否符合jvm规范，安全性检查
    - 准备：为static变量分配空间，设置默认值（与class对象存储在一起，存储在堆中），static变量是在初始化阶段进行赋值，而static final变量的基本类型（不是通过new方式进行创建）即常量会在准备进行赋值 准备阶段做的工作如下：
        - 为类中的静态变量分配空间并设置默认的初始值
        - 为引用变量分配空间
        - 为常量（final static）分配初始值，并进行赋值

```
static int a = 10;//准备阶段仅分配空间
final static int b = 20; //准备阶段分配空间并赋值
static final Object o = new Object() //准备阶段分配空间不进行赋值
```

- 解析：将常量池中的符号引用解析为直接引用 ClassLoader.loadClass（）会导致类的加载和链接，不会导致类的解析和初始化,使用new()会导致类的解析和初始化

- 初始化：调用() V,虚拟机会保证这个类的构造方法的线程安全，初始化的时机包括以下几点：

    - main方法所在的类会先初始化
    - 首次访问类的静态变量或静态方法会导致初始化
    - 子类初始化，如果父类还未初始化，则先初始化父类
    - 子类访问父类静态变量，只会触发父类的初始化，子类不会初始化
    - Class.forName会导致初始化
    - new会导致初始化 不会导致初始化的情况： 类独享的.class不会触发初始化，访问static final静态常量不会出发初始化，创建类的数组不会出发初始化，调用ClassLoader.loadClass方法不会触发初始化，Class.forName的参数2为false不会触发初始化

- 懒汉式的单例模式就是是利用类的加载中初始化条件及线程安全性进行懒惰加载

类加载器

jdk8中类加载的层级： Bootstrap ClassLoader 加载java_home/jre/lib，打印出来的类加载器是null Extension ClassLoader 加载java_home/jre/lib/ext Application ClassLoader 加载classpath路径下的类加载器 自定义类加

载器 自定义要加载类的路径 使用双亲委托机制来加载

Bootstrap ClassLoader 可以指定虚拟机参数来让Bootstrap ClassLoader来加载指定的类 指定要加载的类路径 java -Xbootclasspath/a:. com.steven.Test -Xbootclasspath bootclasspath /a:. 将当前路径追加到类加载路径中

Extension ClassLoader

双亲委派模式 调用类加载器的loadClass方法时，首先由上级类加载器来加载，如果上级类加载器没有加载成功，则才向下查找类加载器来加载 类加载器之间并没有继承关系 类加载源码

```
  protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    long t1 = System.nanoTime();
                    c = findClass(name);

                    // this is the defining class loader; record the stats

sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }
```

线程上下文类加载器 jdbc的驱动加载 Class.forName("com.mysql.jdbc.Driver"); jdbc的驱动使用的是spi的方式来对驱动类进行加载: service provider interface 获取线程上下文类加载器：在线程启动时将应用程序类加载器赋值给线程上下文类加载器 Thread.currentThread().getContextClassLoader()

自定义类加载器 使用场景：

- 想加载自定义路径下的类文件
- 通过接口实现软件解耦
- 希望对类进行隔离，避免同名同包的类的冲突（tomcat中有使用）

示例：

```
public class MyClassLoader extends ClassLoader{

    @SneakyThrows
    @Override
    protected Class<?> findClass(String name) throws
ClassNotFoundException {
        String path =
"/Users/lijie3/Documents/code/javaBasicKnowledge/target/classes/com/steven
/jdk8/classloader/" + name+".class";
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        Files.copy(Paths.get(path),os);
        final byte[] bytes = os.toByteArray();
        return defineClass(name,bytes,0,bytes.length);
    }

    public static void main(String[] args) throws ClassNotFoundException {
        MyClassLoader myClassLoader = new MyClassLoader();
        final Class<?> testClass =
myClassLoader.loadClass("com.steven.jdk8.classloader.TestClass");
        final Class<?> testClass2 =
myClassLoader.loadClass("com.steven.jdk8.classloader.TestClass");
        System.out.println(testClass2 == testClass);
    }
}
```

判断两个类完全相等的条件 包名类名相同，且类加载器对象是同一个

运行期优化

热点代码 在运行期，jvm会对java代码进行优化 jvm将执行状态分为5层：

- 解释执行
- C1即时编译执行（不带profilling）
- C1即时编译执行（带基本profilling）
- C1即时编译执行（带完整profilling）
- 使用C2即时编译器

对将热点代码（hotspot）编译成机器码，提升执行效率

方法内联 当方法较短时，将方法内代码放入到调用的方法中 打印内联信息 -XX:+PrintInlining -X:+UnlockDiagnosticVMOptions

字段优化 将静态变量和常量进行优化

反射优化 循环调用方法达到膨胀阈值（默认15次）后，会触发生成动态MethodAccessor，然后接下来的调用是走普通的方法调用而不走反射调用

java内存模型 JMM java memory Model

定义了一套在多线程读写共享数据时，对数据可见性，有序性和原子性规则保证

synchronized 保证代码的原子性、有序性和可见性 **代码原子性** 内存模型

- jvm中将内存分为主内存和工作内存
- 共享的变量信息放在主内存中
- 线程数据信息放在工作内存中，共享变量需要先从主内存中读取到线程工作内存中，操作完成之后再写入主内存中，由于多线程下的指令重排可能导致数据的读写不安全的问题 **代码可见性** 观察下面这段代码： 这段代码执行后无法退出，原因： 线程从主内存中读取flag的值到工作内存后，由于jit即时编译器的优化，会导致线程不会再while判断时每次从主内存中获取flag，因此检测不到flag在主内存中由true变false ，因此无法退出

```
public class ThreadWhile {

    static boolean flag = true;
    public static void main(String[] args) throws InterruptedException {
        new Thread(()->{
            while (flag){
            }
        }).start();

        Thread.sleep(1000);
        flag = false;
    }
}
```

解决上面的问题，

1. 使用volatile，保证变量对主内存中共享数据的可见性

```
volatile static boolean flag = true;
```

volatile无法保证原子性，只适用于一个线程写，多个线程读 2. 在循环体中使用打印,System.out.println使用了synchronized,这里会让线程重新拿到更新后的flag

```
  static boolean flag = true;
    public static void main(String[] args) throws InterruptedException {
```

```
        new Thread(()->{
            while (flag){
                System.out.println(1);
            }
        }).start();

        Thread.sleep(1000);
        flag = false;
    }
```

有序性 下面的代码由于jit即时编译可能导致指令重排

```
public class CodeSort {

    int num = 0;
    boolean ready = false;

    public void actor1(Result r){
        if(ready){
            r.r1 = num +num;
        }else{
            r.r1 = 1;
        }
    }

    public void actor2(Result r){
        num = 2;
        ready = true;
    }

    public static void main(String[] args) throws InterruptedException {
        CodeSort c = new CodeSort();
        Result result = new Result();
        new Thread(() -> c.actor1(result)).start();
        new Thread(() -> c.actor2(result)).start();
        Thread.sleep(100);
        //由于指令重排可能导致出现 值为0的情况
        System.out.println(result.r1);
    }
}

class Result{
    int r1;
}
```

- 使用volatile可以解决指令重排的问题 **volatile可以保证代码的可见性，能禁止指令重排，法保证代码的原子性，所以volatile无法保证线程安全性** 使用volatile不能保证线程安全的情况：

```
    public static volatile int i = 6;

     public static void main(String[] args) throws InterruptedException {
         ThreadPoolExecutor threadPoolExecutor = new
ThreadPoolExecutor(8,8,
                 60, TimeUnit.SECONDS,new ArrayBlockingQueue<>(100),
                 new DefaultThreadFactory("test-pool"), new
ThreadPoolExecutor.AbortPolicy());
         for (int j = 0; j < 10; j++) {
             threadPoolExecutor.execute(()->{
                 for (;;){
                     if(i > 0){
                         try {
                             Thread.sleep(1000);
                         } catch (InterruptedException e) {
                             e.printStackTrace();
                         }
                         i--;
                         System.out.println(Thread.currentThread() + ":" +
i );
                     }else{
                         break;
                     }
                 }
             });
         }
         System.out.println(Thread.currentThread().getName() + i);
         threadPoolExecutor.shutdown();
     }
```

执行结果： main6 Thread[test-pool-1-5,5,main]:4 Thread[test-pool-1-6,5,main]:3 Thread[test-pool-1-4,5,main]:4 Thread[test-pool-1-2,5,main]:4 Thread[test-pool-1-8,5,main]:2 Thread[test-pool-1-7,5,main]:1 Thread[test-pool-1-1,5,main]:0 Thread[test-pool-1-3,5,main]:-1 Thread[test-pool-1-7,5,main]:-3 Thread[test-pool-1-8,5,main]:-2 Thread[test-pool-1-6,5,main]:-5 Thread[test-pool-1-4,5,main]:-6 Thread[test-pool-1-5,5,main]:-4 Thread[test-pool-1-2,5,main]:-7 分析：i>0处判断后进行了sleep，当睡眠线程都被唤醒时，都会进行i--操作，会出现i< 0的情况，导致计算结果错误

创建Singleton5的对象的指令对应了四个步骤： 0 new #2//分配空间 3 dup //复制栈顶 4 invkespecial #3 执行invokespecial，初始化实例对象,为成员变量赋值 7 putstatic #4 把这个对象的内存地址赋值给引用变量 4和7可能出现指令重排，可能将instance首先分配了空间，引用不为空，则某个线程正在执行构造方法（invokespecial），而另一个线程可能已经将instance进行了返回，这样其他线程拿到的就是未完全实例化只是分配了空间的instance

```
volatile boolean ready = false;
```

DCL（double checked locking）单例子模式必须使用volatile来保证线程安全

```

```

```
public class Singleton5 {
    private Singleton5() {
    }

    private static volatile Singleton5 instance;

    public static Singleton5 getInstance(){
        //第一次判断，如果instance不为null
        if(instance == null){
            synchronized (Singleton5.class){
                if(instance == null){
                    instance = new Singleton5();
                }
            }
        }
        return instance;
    }
}
```

happends-before "Happens before"原则是并发计算中的一个概念，用于描述事件之间的顺序关系。它是指在多线程或多进程的计算环境中，如果事件 A 在时间上先于事件 B 发生，那么事件 A 就被认为是在事件 B 之前发生的。 happends before 是一种内存可见性模型 多线程下由于指令重排序导致数据可见性的问题 A线程修改共享变量的值可能对B线程不可见 JMM通过happends beofore提供了一个跨越线程的可见性的保障，如果操作A在时间上必然先于B发生，则A就认为在B之前发生 happends before不表示指令执行的先后顺序，只要对最终的结果没有影响，jvm是允许指令重排序的

规定对共享变量的写操作对其他线程读操作可见，是可见性与有序性的一套规则

- 线程解锁m之前对变量的写，对于接下来对m加锁的其他线程对变量的读可见
- 线程对volatile变量的写对接下来其他线程对变量的读可见
- 线程start之前对变量的写对该线程开始后对该变量的读可见
- 线程结束前对变量的写对其他线程得知它结束之后的读可见
- 线程1打断（interrupt）线程2前对变量的写对于其他线程得知线程2被打断后的变量的读可见
- 对变量默认值的写对其他线线程对该变量的读可见
- 传递性，volatile变量写入，写屏障会将写屏障之间的所有操作都同步到主存（即使写屏障之前的某个变量不是volatile变量）

CAS与原子类

- 乐观锁
- 使用cas和volatile可以实现无锁并发
- 适于竞争不激烈、多核CPU的场景
- 竞争激烈的情况下会导致重试频繁发生，反而影响效率

底层实现： 使用Unsafe直接调用操作系统底层的CAS指令 java中的锁 乐观锁：cas 悲观锁：synchronized

原子类操作： 利用cas+volatile实现无锁并发

synchronize的优化 jvm中对象的组成

> 对象头 * 对象的hashcode * markword * 对象的锁状态 * 对象当前使用的线程id * gc标志 * 对象元数据指针（指向方法区（元空间）） * 数组长度（数组对象才有）

> 对象数据 对齐填充

偏向锁 对象创建后有线程获取对象锁时，对象的markword锁标志会被设置为偏向锁标志，并将当前线程放入markword中 在没有竞争时，每次重入还是得执行cas操作，jdk6进一步优化只有第一次使用cas将线程id设置到对象的mark word头，之后发现这个id是自己的就表示没有竞争，不用重新进行cas操作 撤销偏向锁需要将持有锁的线程升级为轻量级锁，需要暂停所有线程STW

轻量级锁 如果一个对象虽然有很多线程访问，但线程访问的时间是错开的（没有竞争），那么可以使用轻量级锁来优化，使用CAS来获取锁记录，会将锁的标志位从01（无锁）变成00（轻量级锁） **锁膨胀** 在轻量级锁在进行CAS时经常失败（产生了竞争导致获取锁记录失败），则锁会升级成重量级锁，锁标志位从00 变成10（重量级锁） **重量级锁**

> 重量级锁通过monitorenter、monitorexit实现 重量级锁通过锁自旋实现优化，当线程获取锁失败时，首先进行自旋尝试，当一段时间后如果能获取到锁，则线程不需要进行阻塞 多核cpu才能使用闲置的核心进行自旋，单核自旋没有意义

其他

13. 对象的创建过程 jvm类加载子模块加载完类后对对象进行初始化，给对象的属性赋初始值--->执行底层的<init>方法，实际上是对对象的属性赋真实的值，和执行指定的类构造器方法

17.垃圾回收算法

18.对象的完整创建过程

> 类加载检查 类加载 分配内存（堆内存） 初始化属性的默认值（0值） 设置对象头（对象分代年龄、锁信息、gc标志等信息） 执行<init>方法（先对属性的真实值进行赋值，然后执行对象的构造方法）

19.对象的组成部分

20.jvm中的类加载器

- 引导类加载器，负责加载支持JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar,charsets.jar等
- 扩展类加载器：负责加载支撑JVM运行的位于jre的lib目录下的ext扩展目录中的jar包
- 应用程序类加载器：负责加载ClassPath路径下的类包，主要是加载自己编写的应用程序类
- 自定义加载器：负责加载用户自定义路径下的类包

```
//jdk 8下的运行结果如注释
public class ClassLoaderTest {

    public static void main(String[] args) {
        //引导类加载器，返回null
        System.out.println(String.class.getClassLoader());
        //应用程序类加载器，
jdk.internal.loader.ClassLoaders$AppClassLoader@251a69d7
```

```
        System.out.println(ClassLoaderTest.class.getClassLoader());
        //扩展类加载器 sun.misc.Launcher$ExtClassLoader@6d1e7682

System.out.println(ClassLoaderTest.class.getClassLoader().getParent());
        //引导类加载器  null

System.out.println(ClassLoaderTest.class.getClassLoader().getParent().getP
arent());
    }
}
```

```
//jdk 17下的运行结果如注释
public class ClassLoaderTest {

    public static void main(String[] args) {
        //引导类加载器，返回null
        System.out.println(String.class.getClassLoader());
        //应用程序类加载器,
jdk.internal.loader.ClassLoaders$AppClassLoader@251a69d7
        System.out.println(ClassLoaderTest.class.getClassLoader());
        //扩展类加载器
jdk.internal.loader.ClassLoaders$PlatformClassLoader@16b98e56

System.out.println(ClassLoaderTest.class.getClassLoader().getParent());
        //引导类加载器  null

System.out.println(ClassLoaderTest.class.getClassLoader().getParent().getP
arent());
    }
}
```

21.双亲委派机制

类加载器的亲子层级结构

> 顶层父类 引导类加载器 扩展类加载器 应用程序类加载器 自定义类加载器

程序员自己写了一个类，会首先找到应用程序类加载器，应用程序类加载器会委托给扩展类加载器，而扩展类加载器会继续向上委托给引导类加载器，如果引导类加载器能加载，则加载该类，如果不能加载，扩展类加载器尝试加载，如果扩展类加载器无法加载，则应用程序类加载器尝试自己加载

这样做原因 保证jre的核心jar包加载的安全性，防止jre核心的被篡改 避免类的重复加载

22.tomcat的类加载机制

> 因为tomcat中会部署不同的web应用，如果A应用里使用了spring3，而B应用里用了spring4，则两个应用中可能存在大量的spring重复类，而且重复类之间可能存在不同的逻辑，如果使用双亲委派机制，则由于同一classpath路径下的类不会被重复加载，这样可能导致了应用中类加载存在问题 tomcat自定义了WebAppClassLoader打破了双亲委派机制解决了上面的问题

23.指针碰撞与空闲列表

指针碰撞：这种情况将java堆的内存化为绝对规整的一块区域，所有用过的内存放在一边，空闲内存放在另一边，中间的指针作为分界点的指示器，分配内存就仅仅是吧指针指向空闲空间的那一边挪动一段与对象大小相等的距离 空闲列表：java堆内存不是绝对规整的，已经使用的内存与未使用的内存空间相互交错，虚拟机维护一个列表，记录那些内存时可以用的，在分配内存时从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录

24.对象分配内存时的并发问题解决CAS与TLAB

在并发情况下，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况 解决办法：

- CAS （比较并交换）虚拟机采用cas配上失败重试的方式保证更新操作的原子性来对分配内存空间的动作进行同步处理
- TLAB（thread local allocation buffer）本地线程分配缓冲，把内存分配的动作按照线程划分在不同的空间中进行，每个线程在java堆中预先分配一小块内存，通过-XX:+/-UserTLAB参数来设定，-XX:TLABSize 指定分配内存的大小

25. 对象占用内存的大小

> 使用jol-core包来分析对象的占用内存以及各个字节的含义

- 对象头（markword（32位4字节，64位8字节） + klass poniter（开启指针压缩占4字节，关闭指针压缩占8字节）+数组长度4字节（数组对象有值））
- 实例数据
- 对齐填充，保证对象的总的字节数是8的整数倍

26. 对象指针压缩

将对象的内存地址从8个字节压缩到4个字节 在jdk8下指针压缩默认开启 -XX:UseCompressedOops

27. 对象逃逸分析

分析对象的作用域，当一个对象在方法中被定义后，可能被外部方法所引用，例如作为调用参数传递到其他地方

```
//返回了方法内部使用的对象导致逃逸
 public ClassLoaderTest test(){
        return new ClassLoaderTest();
    }
```

开启逃逸分析 -XX:+DoEscapeAnalysis jdk7之后默认开启（-表示关闭） 开启逃逸分析+标量替换可以避免将局部变量分配在堆上，避免大量的gc

28.判断对象是否是垃圾的引用计数法有什么问题

当对象创建后每出现对对象的一次引用，引用计数加一，没失去一个引用，引用计数器减一，当计数器值为0时被判定为垃圾对象

> 出现了循环引用，导致引用计数引用永远都不会为0，出现内存泄露

29. GC可达性分析

将GC roots作为起点，从这些节点开始向下搜索引用的对象，找到的对象都被标记为非垃圾对象，其他对象都是垃圾对象 GC roots根节点：

> 线程栈的本地变量 静态变量 本地方法栈变量

30.什么样类能被回收

> 该类的所有对象实例都已经被回收，java堆中不存在该类的任何实例 加载该类的Classloader已经被回收 该类对应的java.lang.class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法

jvm内部的各种垃圾回收算法

标记清除算法 将内存中的垃圾对象（或非垃圾对象）做标记，然后清除垃圾对象

> 造成大量的空间碎片 **标记-复制算法** 将内存分为两块，其中一块作为备用的复制算法，一块作为当前使用的算法，当需要垃圾回收时，标记当前使用的内存中的垃圾对象，将非垃圾对象复制到备用内存中，然后将已经使用的内存清空，备用内存切换当前成使用内存 空间利用率变低 当存在大量存活对象时，需要进行大量的内存复制操作，效率比较低下

标记-整理算法 标记存活对象，将剩余的垃圾对象清除，再将存活对象移动到内存空间的一端

31.分代收集理论

大部分对象都是朝生夕死

34.CMS存在的问题

并发收集阶段再次触发full gc

> 在并发标记、并发清理的阶段如果用户线程产生了大量的对象，可能会触发full gc，则并发失败 此时会进入用户线程停止阶段，收集器切换到serial old垃圾收集器来回收垃圾

35.三色标记算法

黑色：垃圾收集器访问过的对象，它是安全存活的，如果有其他对象引用指向了黑色对象，无需重新扫描一遍.黑色对象不可能直接指向某个白色对象 灰色：表示对象已经被垃圾收集器访问，但这个对象上至少还有一个引用没有被扫描过 白色: 对象尚未被垃圾收集器访问过，可达性分析刚开始剪短，所有对象都是白色对象

36.G1垃圾回收器回收垃圾的过程

- 初始标记：暂停锁哟前台新城，并记录下gc roots直接能引用的对象，速度很快
- 并发标记：同cms的并发标记（不暂停用户线程）
- 最终标记：同cms，需要暂停用户线程
- 筛选回收：首先对各个region的回收价值成本进行排序，根据用户所期望的GC停顿STW时间（-XX:MaxGcPauseMills指定）来制定回收计划 回收使用的上方是标记复制算法，将一个region中存货的对象复制到另一个region中，不会像cms那样回收完因为很多内存碎片还需要整理一次，G1采用复制算法回收几乎不会有内存碎片

37.G1垃圾回收器最大停顿时间的实现

G1收集器后台维护一个有限列表，根据允许的收集时间，优先选择回收价值最大的region。这种使用region划分内存空间及有优先级的区域回收方式，保证g1在有限时间内可以尽可能提高收集效率

38.什么是内存泄漏，怎么快速排查

举例：在jvm中使用map进行缓存，当map越来越大，一直占用老年代空间，时间长了就导致full gc频繁，甚至导致OOM 使用jvm自带的jmap，jconsole、jstack等工具来排查 可以使用redis等缓存工具将jvm的缓存移动到redis数据库中

39. GC是什么时候做的

GC是需要代码运行到安全点或安全区域才能做 安全点指代码中的特定位置，线程运行到这些位置时他的状态是确定的，有以下几种安全点：

1. 方法执行返回之前
2. 调用某个方法之后
3. 抛出异常的位置
4. 循环的末尾 安全点是对正在执行的线程设定的，如果一个线程处于sleep状态或中断状态，它不能响应JVM的终端请求，再运行到安全点上。 安全区域：一段代码片段中，引用关系不会发生变化，这个区域内的任何地方开始GC都是安全的

40.字符串常量池

jdk7之后字符串常量池重永久代的运行时常量池分离到堆里 jdk8中，永久带被移除，运行时常量池在元空间，字符串常量池依然在堆里

JMX

JMX(Java Management Extensions),Java管理扩展,是一个为应用程序植入管理功能的框架

Notification MBean之间的通信是必不可少的，Notification起到了在MBean之间沟通桥梁的作用。JMX 的通知由四部分组成： 1、Notification这个相当于一个信息包，封装了需要传递的信息 2、Notification broadcaster这个相当于一个广播器，把消息广播出。 3、Notification listener 这是一个监听器，用于监听广播出来的通知信息。 4、Notification filiter 这个一个过滤器，过滤掉不需要的通知。这个一般很少使用

正则表达式

简单的正则表达式匹配

```
    public static void main(String[] args) {
        String input = "include      script.`57054`; select * from a; " +
                "include      script.`57080`;";
        String pattern = "include\\s+script\\.\\`(\\d+)\\`";
        Pattern r = Pattern.compile(pattern);
        Matcher matcher = r.matcher(input);
        while (matcher.find()){

            String values = matcher.group(1).replaceAll("\\s", "");
```

```
            System.out.println("Values: " + values);

        }
         String input2 = "valueMode = \"scriptId\"\n" +
                "    and values = \"61574\"";
//        String input2 = "values = \"61574\"\n" +
//                "    and valueMode = \"scriptId\"";
        //valueMode\s*=\s*"scriptId"\s+and\s+values\s*=\s*"(\d+)"
        String pattern2 =
"valueMode\\s*=\\s*\"scriptId\"\\s+.*and\\s+values\\s*=\\s*\"
(\\d+)|values\\s*=\\s*\"
(\\d+)\"\\s+.*and\\s+valueMode\\s*=\\s*\"scriptId\"";
        Pattern r2 = Pattern.compile(pattern2);
        Matcher matcher2 = r2.matcher(input2);
        while (matcher2.find()) {
            String values = matcher2.group(1);
            if(StringUtils.isEmpty(values)){
                values = matcher2.group(2);
            }
            System.out.println("Values: " + values);
        }
    }
```

排查 JVM 内存溢出问题通常需要一系列的步骤和工具，以下是一个示例：

错误信息和日志分析：

首先，查看应用程序的错误信息和日志。通常，JVM 内存溢出问题会导致 OutOfMemoryError 异常。查看异常堆栈跟踪以确定问题的源头。 内存快照：

使用 JVM 提供的工具（如 jmap）获取应用程序的堆内存快照。这将显示哪些对象正在占用大量内存。 例如，可以运行以下命令获取堆内存快照：jmap -dump:format=b,file=heapdump.hprof 。 内存分析工具：

使用内存分析工具（如 Eclipse Memory Analyzer 或 VisualVM）来分析堆内存快照文件。这些工具能够可视化显示哪些对象占用了大量内存，帮助你找到潜在的内存泄漏或过度使用内存的地方。 代码审查：

仔细审查应用程序的代码，特别是涉及内存管理的部分。查找可能导致内存泄漏或不必要的内存分配的代码。 确保你的代码在使用完对象后进行正确的资源释放，特别是在使用类似于文件流或数据库连接等资源时。 GC 日志分析：

启用 JVM 的垃圾回收日志（GC 日志），并分析它们以了解垃圾回收活动。GC 日志可以告诉你内存使用的趋势以及垃圾回收是否在工作。 你可以使用 JVM 参数来启用 GC 日志，例如 -XX:+PrintGCDetails 和 -XX:+PrintGCDateStamps。 堆大小和GC 参数调整：

根据你的应用程序的内存需求，可能需要调整堆大小和垃圾回收参数。过小的堆或不合适的 GC 配置都可能导致内存溢出问题。 使用内存分析工具进行实时监控：

一些工具允许你实时监控应用程序的内存（jstat）使用情况。这对于发现内存泄漏或内存使用异常的情况

在实际中，可以遇到以下一些排查JVM内存溢出问题的情况：

OutOfMemoryError: Java heap space：这是最常见的JVM内存溢出错误，表示Java堆空间不足。可以通过查看堆转储文件（heap dump）来了解哪些对象占用了大量的内存并进行分析。

OutOfMemoryError: PermGen space（Java 7之前）或 OutOfMemoryError: Metaspace（Java 8及更高版本）：这表示永久代或元空间不足。可以通过调整永久代（或元空间）的大小或增加它们的上限来解决。有可能由于反射或cglib机制的滥用导致了大量class信息存储于方法区，需要重点排查这方面的代码。调整jvm参数配置，-XX:MaxPermSize=256m，将元空间的大小设置更大

OutOfMemoryError: unable to create new native thread：这表示无法创建新的本地线程。可以通过减少线程的创建数量，或者增加操作系统线程的限制，如ulimit（Linux）来解决。

OutOfMemoryError: request bytes for . OutOfMemoryError: unable to create new native thread：这表示JVM试图为Java堆或线程堆栈分配更多的内存，但操作系统中的可用内存不足。可以通过增加操作系统的可用内存来解决。

OutOfMemoryError: GC overhead limit exceeded：这表示垃圾收集器花费了过多的时间来回收垃圾，但仍然无法释放足够的内存。可以通过调整垃圾收集器参数或增加堆内存来解决。

在排查这些问题时，可以使用各种工具和技术，例如使用jmap将内存快照dump下来，然后用堆转储文件分析工具（如MAT），JVM监控工具（如VisualVM），日志分析和代码审查等。

JVM内存溢出是否会导致JVM退出

在线程OOM发生时，java进程不一定会立即退出 在java中线程的异常都是有线程自身来处理，每个java线程都会有一个默认的异常处理器，通过这个异常处理器来处理线程异常 private void dispatchUncaughtException(Throwable e) { getUncaughtExceptionHandler().uncaughtException(this, e); } 而jvm退出的条件是：jvm虚拟机中不存在非守护线程。 线程发生未处理异常会导致线程结束，而与jvm的退出毫无关系 OOM与JVM退出 OOM的发生表示了此刻JVM堆内存告罄，不能分配出更多的资源，或者gc回收效率不可观。一个线程的OOM，在一定程度的并发下，若此时其他线程也需要申请堆内存，那么其他线程也会因为申请不到内存而OOM，甚至连锁反应导致整个JVM的退出。

一般情况下，出现OOM异常，JVM的GC会进行回收，是不会导致JVM进程退出的。要真说唯一导致退出的情况，那就是内存泄漏，由于内存占用越来越大，导致垃圾回收失败，导致申请不到内存从而导致线程异常退出，最终导致jvm退出 不过这种JVM的OOM导致的异常，很好排查。

当jdk出现OOM时的排查

当java进程已经退出时：

- 启动时在jvm启动参数中添加OOM的dump打印信息，当jvm进程退出时，内存溢出可以快速定位到内存溢出的问题 -XX:+HeapDumOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof
- 结合jvisualvm，载入dump的文件查看相信的类的信息

java进程退出的情况

- 使用jmap -dump:format=b,live,file=test1.bin pid 临时将当前的堆内存占用情况打印出来
- 使用 jmap histo:live pid 查看当前所有类的实例情况
- 解决jvisualvm加载dump文件进行调试判断

JVM退出的条件包括

JVM（Java虚拟机）进程退出的条件通常包括以下几种情况：

应用程序执行完成：当Java应用程序的main方法或主要执行线程完成其工作后，JVM会正常退出。这是正常情况下的退出方式。

显式调用System.exit()：如果应用程序调用System.exit()方法，JVM将立即退出，无论应用程序是否已完成其主要工作。

未捕获的异常：如果应用程序抛出了未捕获的异常，且该异常没有被捕获和处理，JVM将以异常的错误代码退出。例如，NullPointerException、ArrayIndexOutOfBoundsException等未捕获的异常可能会导致JVM退出。

内存溢出：当JVM遇到内存溢出（OutOfMemoryError）情况时，通常会导致JVM进程退出。这包括堆内存溢出、方法区溢出、栈溢出等。

用户中断：在一些操作系统中，用户可以使用CTRL+C等键盘快捷键来中断正在运行的Java进程，这将导致JVM退出。

操作系统信号：操作系统可以向JVM发送信号，强制其退出。例如，Unix/Linux系统上的kill命令可以发送信号来终止Java进程。

资源耗尽：如果操作系统在JVM运行期间耗尽了系统资源（如文件描述符、内存、线程等），JVM可能会因无法获得足够的资源而退出。

调用Runtime.getRuntime().halt()：与System.exit()不同，Runtime.getRuntime().halt(int status)方法会导致JVM立即退出，而不会执行清理操作。这是一种比较粗暴的退出方式，不建议常用。

总之，JVM进程会在上述情况下退出。对于正常的应用程序退出，通常是应用程序完成其主要任务后自行退出。而对于异常情况，JVM会记录相关信息，并在退出前尽量释放资源和执行清理工作，以确保应用程序的稳定性。在开发和部署应用程序时，需要注意处理异常情况，以避免不必要的JVM退出。

什么是Full GC，什么情况下会产生Full GC

Full GC是一次特殊的GC，这次GC会回收整个堆内存，包括老年代、新生代、metaspace等 在Full GC的过程中所有的用户线程处于暂停状态 什么情况下会产生full gc

- 调用System.gc()建议虚拟机执行FullGC，虚拟机不一定会真的执行
- 未指定老年代和新生代大小，堆的伸缩会产生full gc，一定要配置-Xmx、-Xms
- 老年代空间不足会触发full GC
- 空间分配担保失败，可能对象晋升的平均大小 > 老年代剩余空间，也可能是minor GC后的晋升的对象大小超过了老年代剩余的空间

公司jvm配置实例

java -XX:+UseContainerSupport -XX:MaxRAMPercentage=75.0 -XX:InitialRAMPercentage=75.0 -XX:MinRAMPercentage=75.0 \ -javaagent:/opt/skywalking_agent/agent/skywalking-agent.jar -Dskywalking.agent.service_name=https://skywalking.dxy.cn -jar target/odep-web-1.0-SNAPSHOT.jar

-XX:+UseContainerSupport 启动容器支持，让 JVM 可以根据容器的资源限制来自动调整内存大小 -XX:MaxRAMPercentage=75.0 jvm运行时使用的最大内存量占用主机可用内存的百分比为75% -XX:InitialRAMPercentage=75.0 JVM 最初分配的内存量占主机可用内存的百分比，这里设置为 75% -

XX:MinRAMPercentage=75.0 JVM运行时使用的最小内存量占主机内存的百分比，这里设置为75% -javaagent:/opt/skywalking_agent/agent/skywalking-agent.jar Java应用程序中的Java代理器路径 -Dskywalking.agent.service_name=https://skywalking.dxy.cn 设置jvm系统属性

如何查看jvm使用的垃圾回收器

1. 查看jvm默认的垃圾回收器可以使用以下命令： Java -XX:+PrintCommandLineFlags -version

jdk8 默认使用的是 parallel scanvenge + parallel old 默认开启指针压缩

-XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC java version "1.8.0_291" Java(TM) SE Runtime Environment (build 1.8.0_291-b10) Java HotSpot(TM) 64-Bit Server VM (build 25.291-b10, mixed mode) 2. 查看某个jvm进程实例使用的垃圾回收器 首先使用`jps`命令查看对应的jvm进程 然后使用`jinfo -flag UseG1GC pid`命令查看指定的参数是否在命令行中被使用 例如查看jvm是否使用了cms垃圾回收器 jinfo -flag UseConcMarkSweepGC 55376 -XX:+UseConcMarkSweepGC

关于GC日志的参数

jdk8中开启GC日志的方法： -verbose:gc 与 -XX:+PrintGC 等价，-XX:+PrintGC被标记成了deprecated -XX:+PrintGCDetails 开启详细日志模式，在这种模式下，日志格式和所使用的GC算法有关 -XX:+PrintGCDateStamps 在gc日志中加上时间戳信息 -Xloggc:/path/to/gc.log 将gc日志输出到指定的文件中 -XX: +HeapDumpOnOutOfMemoryError 让虚拟机在堆内存溢出时自动生成堆转储快照文件 -XX: +HeapDumpOnCtrlBreak 使用ctrl + break键让虚拟机生成堆转储快照文件

指定垃圾回收器的参数

-XX:+UseConcMarkSweepGC 启用cms进行老年代回收 cms垃圾回收器只能与ParNew的年轻代垃圾回收器配合使用，所以这个参数默认启用了 -XX:+UseParNewGC;当CMS 老年代出现了很多内存碎片时，它还会使用SerialOld进行老年代回收 所以：-XX:+UseConcMarkSweepGC=ParNew+CMS+Serial Old -XX:+UseG1GC 启用G1垃圾回收器 -XX:+UseParallelGC 使用并行垃圾回收器（parallel scavenge + parallel old 使用并行垃圾回收器） 所以-XX:+UseParallelGC=Parallel Scavenge+Parallel Old -XX:+UseSerialGC 使用单线程串行垃圾回收器，在单核cpu，小内存的情况下使用比较合适 -XX:+UseSerialGC=Serial New+Serial Old

hotspot参数分类

标准参数 以`-`开头，所有的hotspot虚拟机都支持 非标准参数，以`-X`开头，特定版本hotspot支持 不稳定参数：以`-XX`开头，下个版本可能取消

JVM 自带工具使用

jstat JVM内存信息统计 查看各个功能和区域的统计信息

- -class 查看类加载信息
- -gc 监视jvm的堆状况，包括一个Eden区，两个survivor 区，老年代、永久代等容量、占用情况、垃圾合计等信息
- -gccapacity 监视内容与 -gc相同，输出堆区域使用到的最大、最小空间
- -gcutil 监视内容与-gc基本相同，输出主要关注已经使用空间占总空间的百分比
- -gccasuse 与-gcutil功能一样，额外输出导致上一次垃圾手机产生的原因
- -gcnew 监视新生代垃圾收集状况

jstat -gc 68963 1000 10 S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT 8704.0 8704.0 0.0 8704.0 69952.0 3400.2 174784.0 44053.7 62632.0 58544.8 8368.0 7587.4 27 0.317 4 0.029 0.346 8704.0 8704.0 0.0 8704.0 69952.0 3400.2 174784.0 44053.7 62632.0 58544.8 8368.0 7587.4 27 0.317 4 0.029 0.346 8704.0 8704.0 0.0 8704.0 69952.0 3400.2 174784.0 44053.7 62632.0 58544.8 8368.0 7587.4 27 0.317 4 0.029 0.346 8704.0 8704.0 0.0 8704.0 69952.0 3400.2 174784.0 44053.7 62632.0 58544.8 8368.0 7587.4 27 0.317 4 0.029 0.346 S0C：表示幸存区0的当前容量。 S1C：表示幸存区1的当前容量。 S0U：表示幸存区0的使用容量。 S1U：表示幸存区1的使用容量。 EC：表示伊甸园区（Eden）的当前容量。 EU：表示伊甸园区的使用容量。 OC：表示老年代（Old）当前容量。 OU：表示老年代的使用容量。 MC：表示元数据区（Metaspace）的当前容量。 MU：表示元数据区的使用容量。 CCSC：表示压缩类空间的当前容量。 CCSU：表示压缩类空间的使用容量。 YGC：表示从应用程序启动到采样时发生的年轻代垃圾收集次数。 YGCT：表示从应用程序启动到采样时年轻代垃圾收集所用的时间（单位秒）。 FGC：表示从应用程序启动到采样时发生的完全垃圾收集次数。 FGCT：表示从应用程序启动到采样时完全垃圾收集所用的时间（单位秒）。 GCT：表示从应用程序启动到采样时垃圾收集总时间（单位秒）。

jinfo JVM参数查看和修改 查看和调整jvm启动和运行参数 查看正在运行java应用程序的扩展参数 jinfo 68963 查看某一具体参数是否被设置 jinfo -flag PrintGCDetails 68963 输出 -XX:-PrintGCDetails 打印日志参数的参数并未被设置

开启或关闭某个参数 jinfo -flag [+ | -]name pid 修改指定的参数值 jinfo -flag name=value pid jinfo -flag +PrintGCDetails 68963 开启打印gc详情参数

查看老年代大小 jinfo -flag OldSize 68963
输出：-XX:OldSize=178978816

jmap jvm内存占用信息和快照 监控堆内存的使用情况，生成堆内存快照文件，查看堆内存区域配置信息 jmap -dump 生成java堆转储快照，格式为-dump:[live,]format=b,file=dump.bin jmap -heap 显示java堆详情信息，使用了什么垃圾回收器，参数设置、分代情况等,新生代，老年代的内存占用情况 jmap -histo 现实堆中的对象统计信息，包括类、实例数量、合集容量 jmap -dump -F -dump 选项不响应时，强制生成dump快照

jmap -dump:live,format=b,file=/home/myheapdump.hprof 68963 生成堆的dump文件

jhat 分析jvm堆快照工具 使用jmap -dump生成内存快照文件后，可以使用jhat将dump文件转成html形式，然后通过在浏览器中查看堆的情况 jmap -dump:live,format=b,file=/home/myheapdump.hprof 68963 jhat /home/myheapdump.hprof jhat会启动一个http服务，端口号是7000 在浏览器中直接访问 http://localhost:7000 可以查看系统中的所有对象情况，内存溢出情况

VisualVM 可以离线分析dump文件

jconsole 监控jvm运行状态、内存使用情况、线程状态等，提供对jvm垃圾回收器和类加载器的配置选项

线上问题排查思路：

出现OOM问题时进行排查

1. 出现问题时，视项目的重要程度，如果项目比较重要，访问变慢或者访问宕机影响比较大，则优先重启服务，然后再次观察线上的服务器（如果允许，可以使用-XX: +HeapDumpOnOutOfMemoryError 参数让jvm在出现内存溢出时将内存快照dump下来）

2. 从日志文件中检索JVM的OOM错误信息，确认是不是由OOM导致的问题，如果是OOM问题，需要区分是堆内存溢出还是栈内存溢出
3. 用前面dump出来的文件使用MAT工具进行分析（或是在出现OOM的节点使用jmap -dump命令将内存快照dump下来）
4. 使用top命令观察cpu占用较高的java进程，然后使用top -Hp 1232(pid)定位具体的占用CPU高的线程
5. 查看dump文件中存活类实例，分析实例中实例对象较多的类，定位到代码中具体的类的引用，观察是不是存在内存泄漏的情况
6. 修改代码在本地进行模拟压力测试，确认问题解决重新发布线上

线程诊断与排查问题--cpu飙高的问题排查

原因分析：可能是死锁导致线程被阻塞，cpu花费更多的时间在等待资源上；可能是死循环导致cpu飙高，也有可能是因为jvm内存分配不合理导致垃圾回收频繁导致cpu占用率高

- 使用top命令定位哪个进程对CPU的占用过高
- top -Hp pid 查看某个进程中的线程占用情况
- ps H -eo pid,tid,%cpu|grep pid 进一步定位哪个线程引起cpu占用过高
- 使用jstack pid 查看线程详情，根据线程id找到有问题的线程，（可能是死锁，可能是死循环等问题），定位到线程行数
- 可以使用jstack检测进程中出现的死锁

jvm调优

优化的方向：

- 减少垃圾回收的时间，尽量避免full gc
- 对于并发量高的应用，应该减少单次垃圾回收的用户线程停顿时间，选择低延迟的垃圾回收器（CMS（Concurrent Mark-Sweep）和G1（Garbage-First））
- 对于吞吐量高的应用，应该减少单位时间内的垃圾回收的停顿时间，选择吞吐量有限的垃圾回收器（Parallel GC）

触发fullgc的几个条件：

1. minor gc之后，老年代没有足够的空间来存放晋升老年代的对象，会触发full gc
2. 显示调用了system.gc()
3. 元空间不足 元空间是存放 类元信息、方法元信息、常量池（包括运行时常量池、字符串常量池、类文件常量池，字符串常量池在堆中，运行时常量池在元空间中），使用系统的直接内存，会在jvm运行过程中动态调整大小 元空间的设置参数： -XX:MetaspaceSize 设置元空间初始大小，当占用率达到该值会触发full gc垃圾回收进行类型卸载， -XX：MaxMetaspaceSize 元空间最大值，默认无限制 -XX：MinMetaspaceFreeRatio gc之后，元空间剩余容量百分比 -XX:MaxMetaspaceFreeRatio gc之后，元空间最大剩余容量百分比，这两个参数是为了动态调整元空间大小减少垃圾回收时间