JVM53 问

1 入门部分

1.1 为什么要学习 JVM？

第一：为了面试。（企业招聘战略在升级，业务和技术问的越来越身）

第二：为了更好的理解 JAVA。

第三：为了更好的解决线上问题。

1）实现线上软件升级。(热替换)

2）更好防止内存泄漏，提高内存的有效使用率。

3）更好提高系统的吞吐量。

1.2 你了解哪些 JVM 产品？

Oracle 公司的 HotSpot。

IBM 公司的 J9。

阿里公司的 TaobaoVM。

1.3 JVM 的构成有哪几部分？

第一：类加载子系统(负责将类读到内存，校验类的合法性，对类进行初始化)

第二：运行时数据区(负责存储类信息，对象信息，执行逻辑)

第三：执行引擎(负责从指定地址对应的内存中数据然后解释执行以及 GC 操作)

第四：本地库接口(负责实现 JAVA 语言与其它编程语言之间的协同)

2 类加载部分

2.1 你知道哪些类加载器？

第一：BootStrapClassLoader

第二：ExtClassLoader

第三：AppClassLoader

第四：自定义 ClassLoader

2.2 为什么需要多个类加载器？

每个类加载器都有自己的加载职责，负责从不同位置加载我们所需要的类，同时可以基于需求进行懒加载(按需加载)，例如：

1) 加载基础类库(核心类库)。

2) 扩展类库。

3) 三方类（MyBatis，Spring,…..）。

4) 自己的类。

2.3 什么是双亲委派类加载模型？

所谓双亲委派模型可以简单理解为向上询问、向下委托。当我们的类在被加载时，首先会询问类加载器对象的 parent 对象(两者之间不是继承关系)，是否已经加载过此类，假如当前 parent 没有加载过此类，则会继续向上询问它的 parent，依次递归。如果当前父加载器可以完成类加载则直接加载，假如不可以则委托给下一层类加载器去加载（可以理解为逐层分配任务）。

2.4 双亲委派方式加载类有什么优势、劣势？

通过双亲委派类加载机制，保证同一个类只能被加载一次，同时也是对类资源的一种保护。例如我们自己也写了一个 java.lang.Object 类，为了保证 Java 官方的 java.lang.Object 类加载后不再加载我们的 Object 就可以使用双亲委派机制。但是这里也有一个缺陷，例如我们同一个 JVM 下有多个项目，但是不同项目中有包名类名都相同的类(类中的内容是不同的)，此时只能有一个项目中的类会被加载，其它项目则无法加载。还有这种双亲委派模型可能会因为向上询问和向下委托，多少会影响一些性能。

2.5 描述一下类加载时候的基本步骤是怎样的？

第一：查找类(例如通过指定路径+类全名找到指定类)
第二：读取类(通过字节输入流读取类到内存，并将类信息存储到字节数组)
第三：对字节数组中的信息流进行校验分析以及初始化并将其结构内容存储到方法区。
第四： 创建字节码对象(java.lang.Class)，基于此对象封装类信息的引用，基于这些引用获取方法区类信息。

2.6 什么情况下会触发类的加载？

我们可以将类的加载分为显式加载和隐式加载，显式加载是通过类加载器的 loadClass 方法或 Class 类的 forName 方法
直接对类进行加载。隐式加载一般指构建对象、访问类中属性或方法时触发的类加载。

2.7 类加载时静态代码块一定会执行吗？

不一定，静态代码块是否会执行，取决于类加载时，是否会执行类初始化。

2.8 如何理解类的主动加载和被动加载？

主动加载：访问本类成员或方法时触发的加载。
被动加载：访问本类(当前类)对应的父类属性时，本类(当前类)属于被动加载。被动加载不会触发当前类的初始化。

2.9 为什么要自己定义类加载器，如何定义？

当系统提供的类加载器不能完全满足我们需求时，我们可以考虑自定义类加载器，例如：

1）指定加载源头？(系统提供的类加载器已经确定了从哪些位置加载对应类，假如我们的类不在指定范围内呢？)
2）保证类安全？(可以在类编译时对字节码进行加密，类加载时对字节码进行解密)
3）打破双亲委派模型？(同一个 JVM 下有多个项目时，假如不同项目中有相同名字的类，这些类都需要加载。)

2.10 内存中一个类的字节码对象可以有多个吗？

可以，即使是同一个类，但是它的类加载器不同，生成的字节码对象也不会相同。

3 JVM 运行内存部分

3.1 JVM 运行内存是如何划分的？

JVM 运行时内存有方法区(Method Area)、堆区(Heap)、Java 方法栈(Stack)、本地方法栈、程序计数器(寄存器)。

3.2 JVM 中的程序计数器用于做什么？

Java 中每个线程都有一个程序计数器，为线程私有，用于记录程序执行时的字节码指令地址。

3.3 JVM 虚拟机栈的结构是怎样的？

Java 中每个线程有一个虚拟机栈(Java 方法栈),每个方法的执行和退出会对应着一次入栈(Push)和出栈(Pop)操作。这个栈中的元素为一个一个的栈帧对象，这个栈帧有如下几部分构成：

1) 操作数栈(用于执行运算，例如两个变量值的加减)
2) 局部变量表(用于存储方法内的局部变量，对于实例方法，局部变量表的第 0 个位置为 this)
3) 方法返回值(记录调用方法的返回值)
4) 动态链接(方法中可以调用其它方法，如何找到要调用的方法？)
5) 其它信息

3.4 JVM 虚拟机栈中局部变量表的作用是什么？

局部变量标底层实现是一个数组，用于存储方法内的局部变量。对于 main 方法而言，方法中的 args 这个变量会存储在局部量表下标为 0 的位置。对于实例方法，局部变量下标为 0 位置存储的是 this。

3.5 JVM 虚拟机栈中操作数栈的做用是什么？

最核心的作用是进行计算。JVM 的执行引擎可以基于程序计数器中指令的地址找到具体指令，然后执行。在执行这些指令时，可以将指令对应的数据放到操作数栈、也可以从操作栈将数据取出存储到局部变量表，还可以将局部变量表中的数据取出，进行计算，将计算的结果再存储到操作数栈中。

3.6 JVM 堆的构成是怎样的？

JVM 堆主要用于存储我们创建 Java 对象，从由年轻代(Young 区)和老年代（Old 区）构成，年轻代又分伊甸园区和两个幸存区。

3.7 Java 对象分配内存的过程是怎样的？

1） 编译器通过逃逸分析（JDK8 已默认开启），确定对象是在栈上分配还是在堆上分配。

2） 如果是在堆上分配,则首先检测是否可在 TLAB（Thread Local Allocation Buffer）上直接分配。

3） 如果 TLAB 上无法直接分配则在 Eden 加锁区(CAS 算法进行加锁)进行分配(线程共享区)。

4） 如果 Eden 区无法存储对象，则执行 Yong GC（Minor Collection）。

5） 如果 Yong GC 之后 Eden 区仍然不足以存储对象，则直接分配在老年代。

3.8 JVM 年轻代幸存区设置的比较小会有什么问题？

伊甸园区被回收时，对象要拷贝到幸存区，假如幸存区比较小，拷贝的对象比较大，对象就会直接存储到老年代，这样会增加老年代 GC 的频率。而分代回收的思想就会被弱化。

3.9 JVM 年轻代伊甸园区设置的比例比较小会有什么问题？

我们程序中新创建的对象，大部分要存储到伊甸园区，假如伊甸园设置的比较小，会增加 GC 的频率，可能会导致 STW 的时间变长，进而影响系统性能。

3.10JVM 堆内存为什么要分成年轻代和老年代？

为了更好的实现垃圾回收，减少 GC 时长、提高其执行效率。（思考 GC 系统是扫描小块内存比较块还是扫描大块内存速度快）

3.11项目中最大堆和初始堆的大小为什么推荐设置为一样的？

我们在设置 JVM 初始化堆(-Xms)和最大堆(-Xmx)的大小为一样的目的是，避免程序运行过程中，因对象多少或 GC 后内存发生了变化而调整堆大小，带来的更大系统开销。在很多大厂的开发规范中都推荐初始堆和最大堆的大小是一样的。(例如阿里的开发手册)

3.12什么情况下对象会存储到老年代？

第一：创建的对象比较大，年轻代没有空间存储。

第二：经过多次 GC，没有被回收的对象年龄在增加，默认 15 岁后会移动老年代。

3.13Java 中所有的对象创建都是在堆上分配内存的？

随着技术的升级，这个说法现在不准确了。对象还可以分配到栈上了。

3.14如何理解 JVM 方法区以及它的构成是怎样的？

方法区（Method Area）是 JVM 中的一种逻辑上规范，不同 JDK 对规范的落地会有不同，例如在 JDK8 的 HotSport 虚拟机中称之为 Metaspace（元空间）。方法区主要用于存储已被虚拟机加载的类信息、常 量、静态变量、即时编译后的代码等数据。

3.15JDK8 中 Hotsport 虚拟机的方法区内存在哪里？

JVM 堆外内存，严格来讲属于操作系统的一部分内存，也可以通过参数设置具体大小，假如没有设置，可以无限增大，直到操作系统内存不足。

3.16什么是逃逸分析以及可以解决什么问题？

逃逸分析一种数据分析算法，基于此算法可以检测对象是否发生了逃逸，未逃逸的小对象可以分配栈上，也可以进行标量替换，还可以实现锁消除。总之，可以有效减少 Java 对象在堆内存中的分配，可以减少线程阻塞，提高其执行效率。

3.17如何理解对象的标量替换，为什么要进行标量替换？

将未逃逸的小对象直接打散分配到栈上，减少堆中对象的创建次数。堆中对象创建的少了，GC 的频率就会降低，GC 频率降低了，系统正常业务的执行效率就会提高。

3.18什么是内存溢出以及导致内存溢出的原因？

内存中剩余的内存不足以分配给新的内存请求，此时就会出现内存溢出（OutOfMemoryError）。内存溢出可能直接导致系统崩溃。导致内存溢出的原因可能会有如下几种：

- 创建的对象太大导致堆内存溢出(内存中没有连续的内存空间可以存储你这个大对象)
- 创建的对象太多导致堆内存溢出(对象创建的太多，又不能及时回收这些对象)
- 方法出现了无限递归调用导致栈内存溢出(每次方法的调用都会对应这个一个栈帧对象的创建，同时将栈帧入栈)
- 方法区内存空间不足导致内存溢出。(将如内存中不断的加载新的类，类越来越多，此时可能出现内存溢出)
- 出现大量的内存泄漏

3.19什么是内存泄漏以及导致内存泄漏的原因？

程序运行时，动态分配的内存空间，在使用完毕后未得到释放，结果导致一直占用着内存单元，直到程序运行结束。这个现象称之为内存泄漏。导致内存泄漏的原因可能有如下几点：

- 大量使用静态变量(静态变量与程序生命周期一样)
- IO/连接资源用完没关闭(记得执行 close 操作)
- 内部类的使用方式存在问题(实力内部类或默认引用外部类对象)

- 缓存(Cache)应用不当(尽量不要使用强引用)
- ThreadLocal 应用不当(用完记得执行 remove 操作)

3.20JAVA 中的四大引用类型有什么特点？

Java 中为了更好控制对象的生命周期,提高对象对内存的敏感度,设计了四种类型的引用。按其在内存中的生命力强弱，可分为强引用、软引用、弱引用、虚引用。其中，"强引用"引用的对象生命力最强，其它引用引用的对象生命力依次递减。JVM 的 GC 系统被触发时，会因对象引用的不同，执行不同的对象回收逻辑。

3.21项目中的哪些地方用到了缓存？

1) 数据库内置的缓存？(例如 mysql 的查询缓存)
2) 数据层缓存(一般由持久层框架提供，例如 MyBatis)
3) 业务层缓存(基于 map 等实现的本缓存，分布式缓存-例如 redis)
4) CPU 缓存(高速缓冲区)
5) 浏览器内置缓存？

3.22假如让你设计一个缓存你会考虑哪些问题？

1) 存储结构(使用什么结构存储数据效率会更高?-散列表)
2) 淘汰算法(缓存容量有限-LRU/FIFO/LFU)
3) 任务调度(定期刷新缓存，缓存失效时间)
4) 并发安全(缓存并发访问时的线程安全)
5) 日志记录(缓存是否命中，命中率是多少)
6) 序列化(存对象时序列化、取对象时反序列化)

4 字节码增强部分

4.1 什么是字节码？

符合 JVM 虚拟机规范的操作指令，可以被 JVM 执行引擎解释执行。

4.2 为何要学习字节码？

更好的理解 JAVA 代码，例如对于 Integer a=100，你是怎么知道 100 如何封装为 Integer 类型的，你如何知道注解本质上也是一个接口的，你怎么知道你写的枚举类型默认都继承 Enum，你如何知道 synchronized 是如何进行加锁的等。当我们掌握了字节码的基本特征后，就可以直接基于字节码对类功能进行增强，同时还可以创作出类似 JAVA 语言的编程语言（例如 Scala 语言），只要这个语言编译完成后生成的字节码符合 JVM 规范即可。

4.3 如何解读字节码内容？

我们解读字节码时通常会借助如下几种方式：
方式一：借助 Hex-Editor 插件可以查看字节码的 16 进制形式
方式二：借助 javap 指令直接查看字节码指令
方式三：借助 jclasslib 插件查看字节码指令

4.4 字节码内容由哪几部分构成？

魔数+版本号+常量池+类访问标识+父类引用+接口数+成员变量信息+方法信息+其它属性

4.5 什么是字节码增强？

在类加载或类运行时对类的字节码进行修改或生成新的字节码，这个过程我们称之为字节码

增强。

4.6 为什么要进行字节码增强？

通过这种方式可以实现类功能的增强，同时还可以简化部分重复代码的编写。例如 AOP 的设计、热部署等技术实现都用到字节码增强技术。

4.7 你了解哪些字节码增强技术？

ASM 技术、Javassist 技术等，我们可以基于这些技术修改类的字节码，创建新的接口、类、添加属性、方法都可以。

4.8 什么是热替换以及如何实现？

热替换我们可以理解为一种在线升级技术，就是在服务运行过程中，不重启服务就可以完成系统的在线升级。目前在 java 中实现热替换（热部署），可以基于 Java Agent 技术，此技术可以侵入正在运行的 JVM 应用程序，并借助 Asm 或 javassist 技术修改或扩展目标类型，并通过新的目标类型字节码替换正在运行的 JVM 字节码，以达到热部署的目的(在线升级)。

5 JVM 垃圾回收部分

5.1 何为 GC？

GC（Garbage Collection）称之为垃圾回收，是对内存中的垃圾对象采用一定的算法进行内存回收的一个动作。

5.2 为什么要 GC？

深入理解 GC 的工作机制，可以帮你写出更好的 Java 应用（例如避免内存泄漏，提高运行效率），提高开发效率。

5.3 如何判定对象是否为垃圾？

引用计数法（出现循环引用，对象无法回收，可考虑弱引用、软引用）；
可达性分析法（从 GC root 引用查找对象，假如访问不到则认为不可达）

5.4 你知道哪些 GC 算法？

标记清除、标记复制、标记整理算法

5.5 JVM 中有哪些垃圾回收器？

串行（Serial）、并行(Parallel)、并发(CMS)、G1(收集器)等

5.6 如何查看 JVM 默认的垃圾收集器？

```
-XX:+PrintCommandLineFlags -version
```

5.7 说出几个常用的 JVM 配置参数？

1）堆栈配置相关

-Xmx3550m： 最大堆大小为 3550m。

-Xms3550m： 设置初始堆大小为 3550m。

-Xmn2g： 设置年轻代大小为 2g。

-Xss128k： 每个线程的堆栈大小为 128k。

-XX:NewRatio=4: 设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去

持久 代)。

-XX:SurvivorRatio=4： 设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxTenuringThreshold=0： 设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不 经过 Survivor 区，直接进入年老代。

2）垃圾收集器相关

-XX:+UseParallelGC： 选择垃圾收集器为并行收集器。

-XX:ParallelGCThreads=20： 配置并行收集器的线程数

-XX:+UseConcMarkSweepGC： 设置年老代为并发收集。

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存 空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection： 打开对年老代的压缩。可能会影响性能，但是可以消除碎片

3）辅助信息相关 -XX:+PrintGC 输出形式

5.8 JAVA 中的堆区为什么要分代？

因为 GC 过程都触发 STW(stop the world),也就说可能要暂停正常业务的执行，影响执行效率。如果能够想办法缩短一次 GC 的时长,那我们是否可以只收集其中的一部分内存区域。基于这样的一种原因就产生分代设计思想。

5.9 服务频繁 fullgc，younggc 次数较少，可能原因？

1.经常有超过大对象阈值的对象进入老年代,可以通过-XX:PretenureSizeThreshold 设置，大于这个值的参数直接在老年代分配。

2.老年代参数设置不当，-XX:CMSInitiatingOccupancyFaction=92 设置不合理（阈值达

到多少才进行一次 CMS 垃圾回），导致频繁 FULLGC

3.FULLGC 之后没有整理老年代内存碎片，导致没有连续可用的内存地址，进入恶性循环，导致频繁老年代 GC，-XX:CMSFullGCsBeforeCompaction 可以设置

4.新生代过小，或者 e 区和 s 区比例不当，对象通过动态年龄判断机制频繁进入老年代。

5.不合理使用 System.gc()，造成频繁的 FullGC，-XX:+DisableExplicitGC 这个参数可以禁用 System.gc().

6.存在内存泄露，老年代中驻扎着大量不可回收的对象，一定程度上缩小了老年代的大小，造成对象一进入老年代就触发 FULLGC

7.Meatspace 不够用引发 fullgc,甚至无限 fullgc,这类问题常见于 tomcat 热部署，以及使用反射不当。

5.10你知道哪些 JVM 小工具？

- **Jps**

jps 主要用来输出 JVM 中运行的进程状态信息。语法格式如下：

```
jps [options] [hostid]
```

-q 不输出类名、Jar 名和传入 main 方法的参数

-m 输出传入 main 方法的参数

-l 输出 main 类或 Jar 的全限名

-v 输出传入 JVM 的参数

- **Jstack**

jstack 主要用来查看某个 Java 进程内的线程堆栈信息。语法格式如下：

```
jstack [option] pid
jstack [option] executable core
jstack [option] [server-id@]remote-hostname-or-ip
```

- **jmap**

jmap 导出堆内存，然后使用 jhat 来进行分析， jmap 语法格式如下：

```
jmap [option] pid
jmap [option] executable core
jmap [option] [server-id@]remote-hostname-or-ip
```

使用 jmap -heap pid 查看进程堆内存使用情况

- **jstat**

jstat 是 JVM 统计监测工具，看看各个区内存和 GC 的情况。

```
jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]
```

例如：

```
jstat -gc 21711 250 4
```

vmid 是 Java 虚拟机 ID，在 Linux/Unix 系统上一般就是进程 ID。interval 是采样时间间隔。count 是采样数目。比如下面输出的是 GC 信息，采样时间间隔为 250ms，采样数为 4