# 面试经验

# 常见场景题：

## 1、你们项⽬如何排查JVM问题

对于还在正常运⾏的系统：

1. 可以使⽤jmap来查看JVM中各个区域的使⽤情况

2. 可以通过jstack来查看线程的运⾏情况，⽐如哪些线程阻塞、是否出现了死锁

3. 可以通过jstat命令来查看垃圾回收的情况，特别是fullgc，如果发现fullgc⽐较频繁，那么就得进⾏调优了

4. 通过各个命令的结果，或者jvisualvm等⼯具来进⾏分析

5. ⾸先，初步猜测频繁发送fullgc的原因，如果频繁发⽣fullgc但是⼜⼀直没有出现内存溢出，那么表示 fullgc实际上是回收了很多对象了，所以这些对象最好能在younggc过程中就直接回收掉，避免这些对象进⼊到⽼年代，对于这种情况，就要考虑这些存活时间不⻓的对象是不是⽐较⼤，导致年轻代放不下，直接进⼊到了⽼年代，尝试加⼤年轻代的⼤⼩，如果改完之后，fullgc减少，则证明修改有效

6. 同时，还可以找到占⽤CPU最多的线程，定位到具体的⽅法，优化这个⽅法的执⾏，看是否能避免某些对象的创建，从⽽节省内存

对于已经发⽣了OOM的系统：

1. ⼀般⽣产系统中都会设置当系统发⽣了OOM时，⽣成当时的dump⽂件（-

XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/base）

2. 我们可以利⽤jsisualvm等⼯具来分析dump⽂件

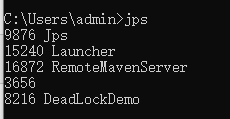
3. 根据dump⽂件找到异常的实例对象，和异常的线程（占⽤CPU⾼），定位到具体的代码

4. 然后再进⾏详细的分析和调试

## 2、[如何查看线程死锁？](https://www.cnblogs.com/lay2017/p/11343311.html)（重点）

1、jps命令获取java进程

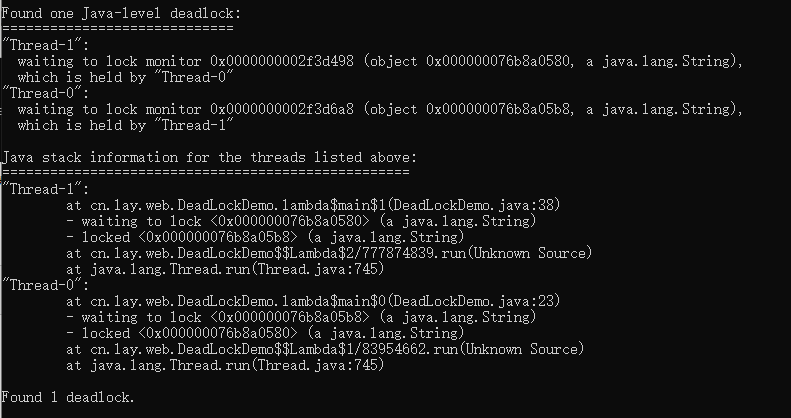
命令：jps



获得DeadLockDemo的进程为8216

2、查看带锁信息的堆栈

命令：jstack -l 8216



这里显示：

1）Thread-1和Thread-0相互等待，造成死锁

2）Thread-1在DeadLockDemo的38行等待

3）Thread-0在DeadLockDemo的23行等待

检查代码即可

# 补充基础题

## 1、Innodb是如何实现事务的

Innodb通过Buffer Pool，LogBuffer，Redo Log，Undo Log来实现事务，以⼀个update语句为例：

1. Innodb在收到⼀个update语句后，会先根据条件找到数据所在的⻚，并将该⻚缓存在Buffer Pool中

2. 执⾏update语句，修改Buffer Pool中的数据，也就是内存中的数据

3. 针对update语句⽣成⼀个RedoLog对象，并存⼊LogBuffer中

4. 针对update语句⽣成undolog⽇志，⽤于事务回滚

5. 如果事务提交，那么则把RedoLog对象进⾏持久化，后续还有其他机制将Buffer Pool中所修改的数据⻚持久化到磁盘中

## 2、Mysql数据库中，什么情况下设置了索引但⽆法使⽤?

1. 没有符合最左前缀原则

2. 字段进⾏了隐式数据类型转化

3. ⾛索引没有全表扫描效率⾼

## 3、Java死锁如何避免？

造成死锁的⼏个原因：

1. ⼀个资源每次只能被⼀个线程使⽤

2. ⼀个线程在阻塞等待某个资源时，不释放已占有资源

3. ⼀个线程已经获得的资源，在未使⽤完之前，不能被强⾏剥夺

4. 若⼲线程形成头尾相接的循环等待资源关系

这是造成死锁必须要达到的4个条件，如果要避免死锁，只需要不满⾜其中某⼀个条件即可。⽽其中前3个条件是作为锁要符合的条件，所以要避免死锁就需要打破第4个条件，不出现循环等待锁的关系。

在开发过程中：

1. 要注意加锁顺序，保证每个线程按同样的顺序进⾏加锁

2. 要注意加锁时限，可以针对所设置⼀个超时时间

3. 要注意死锁检查，这是⼀种预防机制，确保在第⼀时间发现死锁并进⾏解决

## 4、volatile关键字，他是如何保证可⻅性，有序性

1. 对于加了volatile关键字的成员变量，在对这个变量进⾏修改时，会直接将CPU⾼级缓存中的数据写回到主内存，对这个变量的读取也会直接从主内存中读取，从⽽保证了可⻅性

2. 在对volatile修饰的成员变量进⾏读写时，会插⼊内存屏障，⽽内存屏障可以达到禁⽌重排序的效果，

# 吉利外包面试

## 1、说电商项目的实现；

## 2、== 和equals

== 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；而 equals 默认情况下是引用比较，只是很多类重新了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等。

Int a=1; interge b=1

a==b true

Interge c=new Interge(1);

B==c falas

## 3、arrayList扩容

arraylist扩容就是根据旧数组和计算出的新长度copyof出新数组返回给elementData，arraylist初始是final修饰空数组长度0，当添加第一个元素时默认创建长度10的数组，之后扩容就是1.5倍进行扩容。

ArrayList相当于在没指定initialCapacity时就是会使用延迟分配对象数组空间，当第一次插入元素时才分配10（默认）个对象空间。假如有20个数据需要添加，那么会分别在第一次的时候，将ArrayList的容量变为10 (如下图一)；之后扩容会按照1.5倍增长。也就是当添加第11个数据的时候，Arraylist继续扩容变为10\*1.5=15(如下图二)；当添加第16个数据时，继续扩容变为15 \* 1.5 =22个

## 4、线程池参数

核心线程数、最大线程数、最大线程数存货时间、任务队列、ThreadFactory、拒绝策略

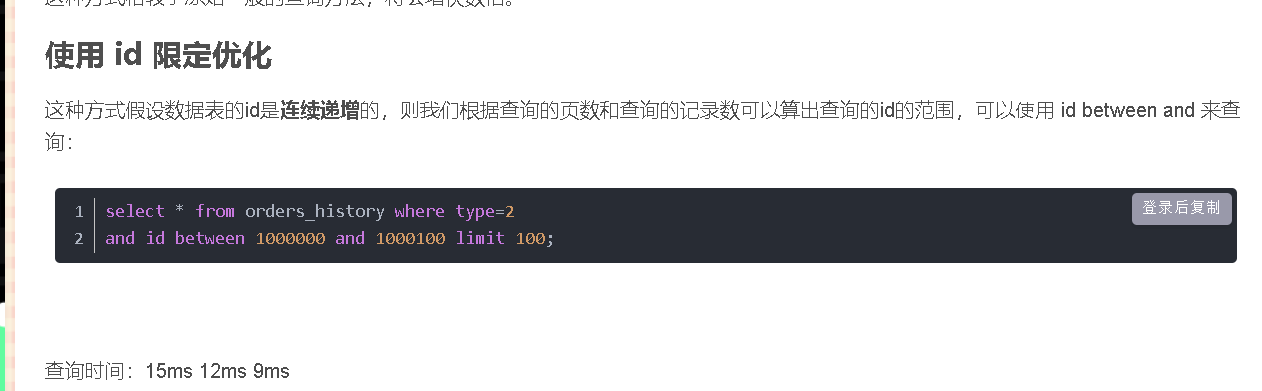
## 5、数据库乐观锁、悲观锁

## 6、 一张表分页查询慢的解决

（1）如果没有索引加索引，索引失效检查失效原因

（2）如果数据太多就分表

（3）优化查询语句，如下：



1. 表id为非数字型

利用表的覆盖索引来加速分页查询  
我们都知道，利用了索引查询的语句中如果只包含了那个索引列（覆盖索引），那么这种情况会查询很快。

因为利用索引查找有优化算法，且数据就在查询索引上面，不用再去找相关的数据地址了，这样节省了很多时间。另外Mysql中也有相关的索引缓存，在并发高的时候利用缓存就效果更好了。

SELECT a.\* FROM table\_name a JOIN (SELECT id FROM table\_name LIMIT 866613, 20) b WHERE a.id=b.id

## 7、为什么innerdb推荐使用自增id作为主键

如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页

## 8、spring事务什么时候会失效

1、发生自调用，类里面使用this调用本类的方法（this通常省略），此时这个this对象不是代理类，而是UserService对象本身！

解决方法很简单，让那个this变成UserService的代理类即可！

2、方法不是public的

@Transactional 只能用于 public 的方法上，否则事务不会失效，如果要用在非 public 方法上，可以开启 AspectJ 代理模式。

3、数据库不支持事务

4、没有被spring管理

5、异常被吃掉，事务不会回滚(或者抛出的异常没有被定义，默认为RuntimeException)

## 9、spring使用了哪些设计模式

### 简单工厂：

由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是 在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

### 工厂方法：

实现了FactoryBean接口的bean是一类叫做factory的bean。其特点是，spring会在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法，所以返回的不是factory这个bean，而是这个bean.getOjbect()方法的返回值

### 单例模式：

保证一个类仅有一个实例，并提供一个访问它的全局访问点

spring对单例的实现： spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是任意的java对象。

### 适配器模式：

Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替 controller执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC 的扩展了。

### 装饰器模式：

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

### 动态代理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象创建动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。

织入：把切面应用到目标对象并创建新的代理对象的过程。

### 观察者模式：

spring的事件驱动模型使用的是 观察者模式 ，Spring中Observer模式常用的地方是listener的实现。

### 策略模式：

Spring框架的资源访问Resource接口。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了Resource 接口来访问底层资源。

### 模板方法：

父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现。

最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。

## 10、实际get请求，前端传为post请求，怎么实现统一打印请求日志

我的回答是自己实现一个请求过滤器fillter,拦截请求打印请求信息。

Aop实现被否决

## 11、springmvc工作流程

## 12、mysql的innerdb数据结构

## 13、介绍rabbitmq

重要组件：

几种广播类型

作用：

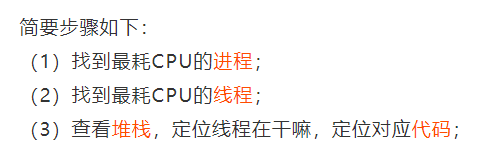
## 14、怎么保证mq高可用

我回答部署集群，面试官不满意。应该再说如何解决消息丢失问题（消息发送确认，消息持久化，消费者确认）

## 15、redis分布式锁、redission分布式锁实现原理

## 16、服务长期cpu100%，是什么原因，怎么解决（重点）

<https://developer.51cto.com/art/202004/614863.htm>



https://mp.weixin.qq.com/s/YGW7eJZVEHJaJqOlK3gMUg

## 微服务使用了哪些组件

# 视洞科技（二面）

## 开始A对象调B对象，在同一个springboot项目里。现在将B对象拆到另外一个项目，问在A不改代码的情况下，怎么让程序不报错，正确执行。采用的分布式dobble

## maven在引用同一个包fastjson，不同版本，程序会不会报错。打包会怎么样。

答：程序编译可以通过，但是执行fastjson相关代码会报错，常见报错：NoSuchMethod；程序打包以maven-compiler-plugin所在的POM文件引用的版本为准。

根据maven依赖处理的机制，引用路径最短的fastjson-？会真正作为应用最终的依赖，其它版本的fastjson则会被忽略

参考：https://blog.csdn.net/weixin\_37968613/article/details/108950774

## 100个人，每个人随机发放黑白不同颜色的各50帽子，每个人说自己帽子的颜色，怎么保证说对的正确率最高。

原理：

最后一个人可以看到前面99人的帽子颜色，从而说出自己帽子颜色，颜色奇数即为自己的颜色。  
同理第99人在知道第100人颜色后，再知道前98人颜色，即可推理出+第100颜色，奇数颜色为自己颜色。

## 现在有一个树，遍历树，怎么对jvm调优让吞吐量更大。

# 吉利正式（一面）

## 1、[Java内存溢出定位和解决方案（new）](https://www.cnblogs.com/snowwhite/p/9471710.html)（重点）

引起内存溢出的原因有很多种，列举一下常见的有以下几种：

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；  
2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；  
3.代码中存在死循环或循环产生过多重复的对象实体；  
4.使用的第三方软件中的BUG；  
5.启动参数内存值设定的过小

6、大量使用静态字段  
7、未关闭资源导致内存泄漏（数据库连接）  
8、不当的equal和hashcode会导致内存泄漏从而导致内存溢出  
9、ThreadLocal

内存溢出的解决方案：

第一步，修改JVM启动参数，直接增加内存。(-Xms，-Xmx参数一定不要忘记加。)

第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。

第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

重点排查以下几点：  
1.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

2.检查代码中是否有死循环或递归调用。

3.检查是否有大循环重复产生新对象实体。

4.检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

第四步，使用内存查看工具动态查看内存使用情况

内存溢出定位：

在启动JVM的时候加上以下两个参数：

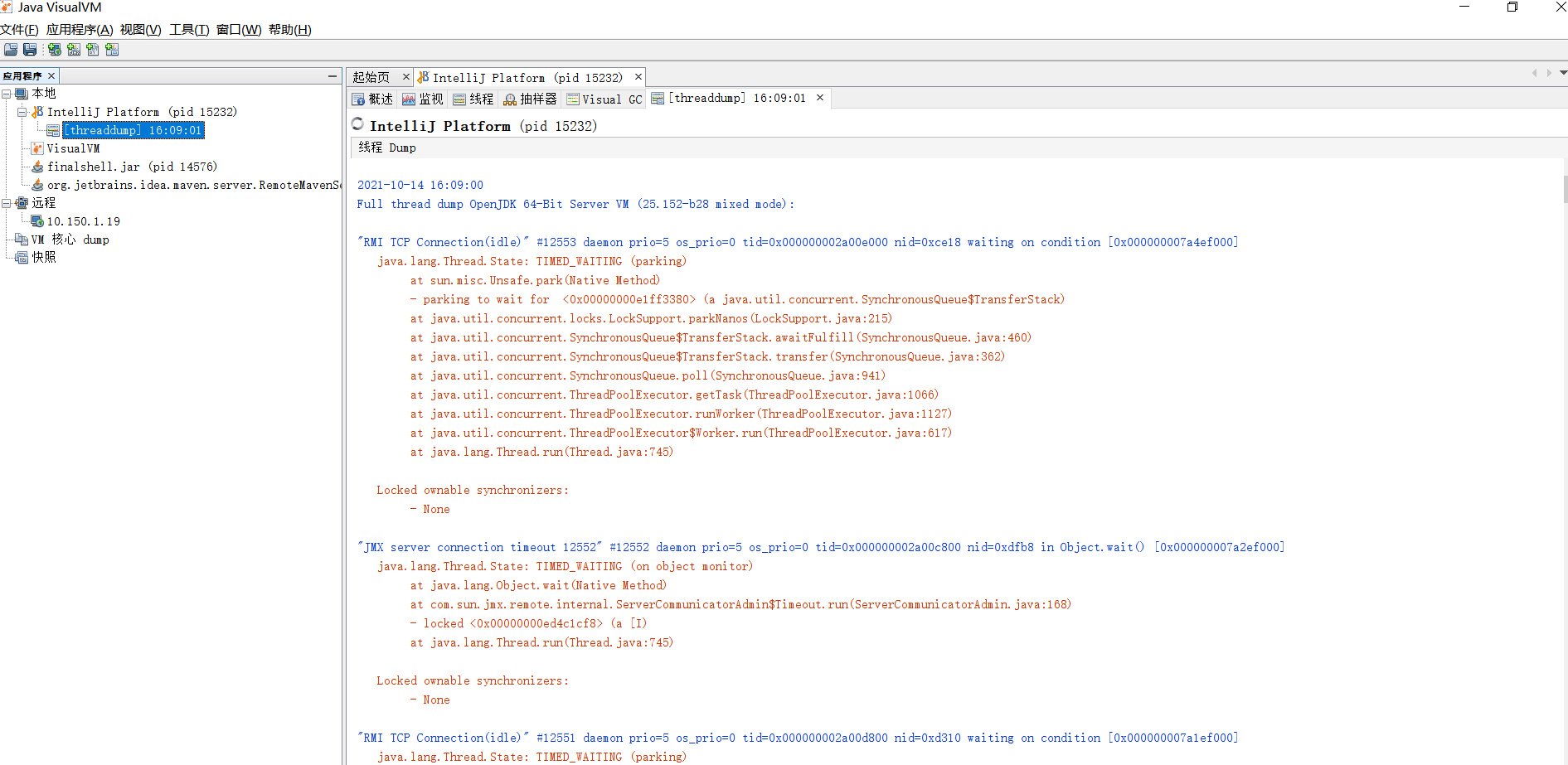
-XX:HeapDumpPath=./dumpfile.hprof

-XX:+HeapDumpOnOutOfMemoryError

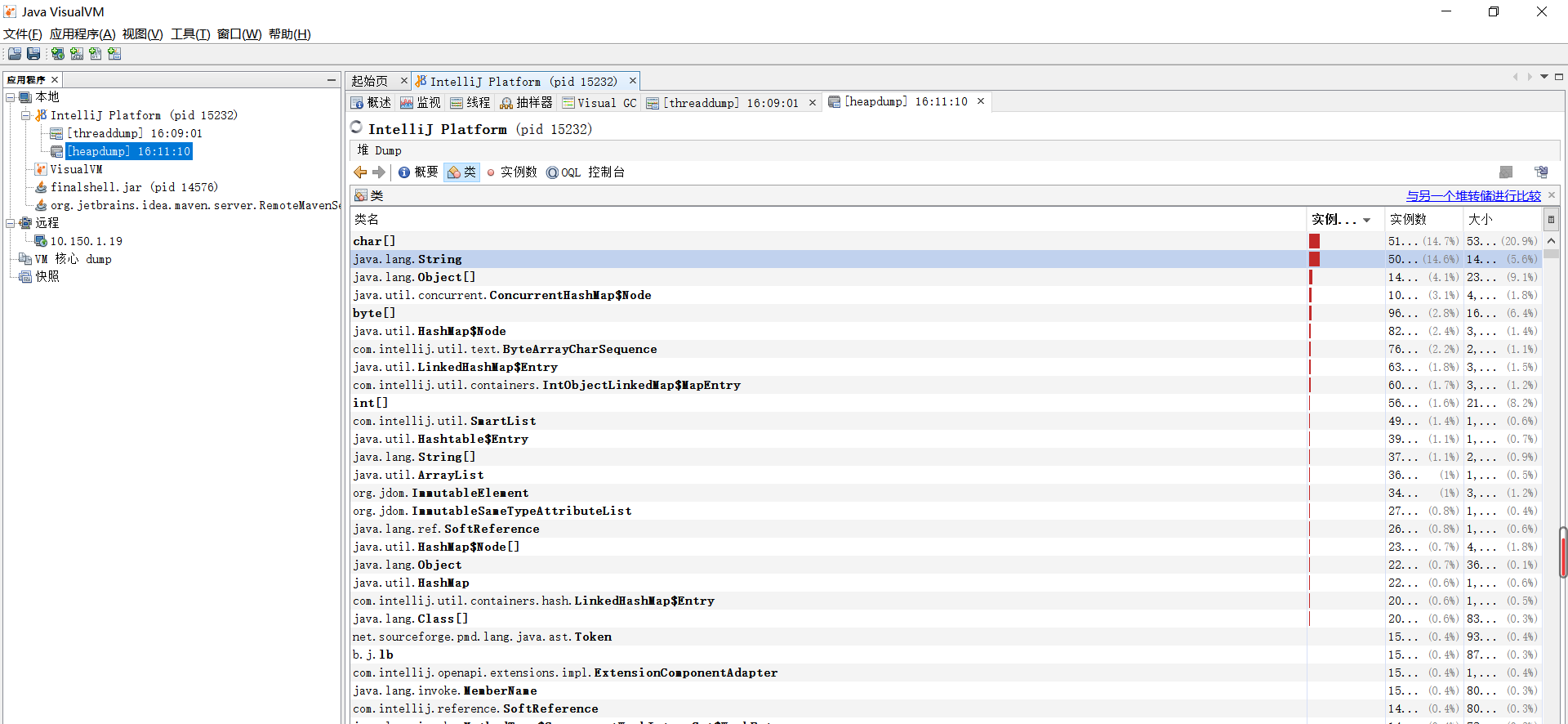
表示出现OOM错误后，将堆信息dump出来。不过这个参数也有个问题，在实际使用的时候发现在windows平台上，如果程序是以TOMCAT服务的方式运行，出现OOM错误后堆信息也dump不出来。

将快照文件导入jdk自带得分析图形化工具visualVM，可以查看堆内存、线程信息

线程快照：分析是哪个线程造成OOM



堆快照：分析是哪个对象造成OOM



## 2、多个版本的fastJson引用问题，假如，如jdk中存在fastjson，自己项目中存在fastjson包，maven存在fastjson包。

答：根据类加载双亲委派机制，可知，Appclassloarder类加载器，首先不会自己加载，会委托父类加载器extClassloarder，查看缓存中是否已经存在，若不存在，则继续向上委托给bootstrapClassloarder类加载器，该加载器已经加载fastjson,直接返回。

参考：<https://www.zhihu.com/question/46719811?sort=created>

## 3、mysq主从数据同步原理

mysql主从同步的过程：Mysql的主从复制中主要有三个线程： master（binlog dump thread）、slave（I/O thread 、SQL thread） ，Master一条线程和Slave中的两条线程。

主节点 binlog，主从复制的基础是主库记录数据库的所有变更记录到 binlog。binlog 是数据库服务器启动的那一刻起，保存所有修改数据库结构或内容的一个文件。

主节点 log dump 线程，当 binlog 有变动时，log dump 线程读取其内容并发送给从节点。

从节点 I/O线程接收 binlog 内容，并将其写入到 relay log 文件中。

从节点的SQL 线程读取 relay log 文件内容对数据更新进行重放，最终保证主从数据库的一致性。

注：主从节点使用 binglog 文件 + position 偏移量来定位主从同步的位置，从节点会保存其已接收到的 偏移量，如果从节点发生宕机重启，则会自动从 position 的位置发起同步。

由于mysql默认的复制方式是异步的，主库把日志发送给从库后不关心从库是否已经处理，这样会产生 一个问题就是假设主库挂了，从库处理失败了，这时候从库升为主库后，日志就丢失了。由此产生两个概念。

全同步复制

主库写入binlog后强制同步日志到从库，所有的从库都执行完成后才返回给客户端，但是很显然这个方式的话性能会受到严重影响。

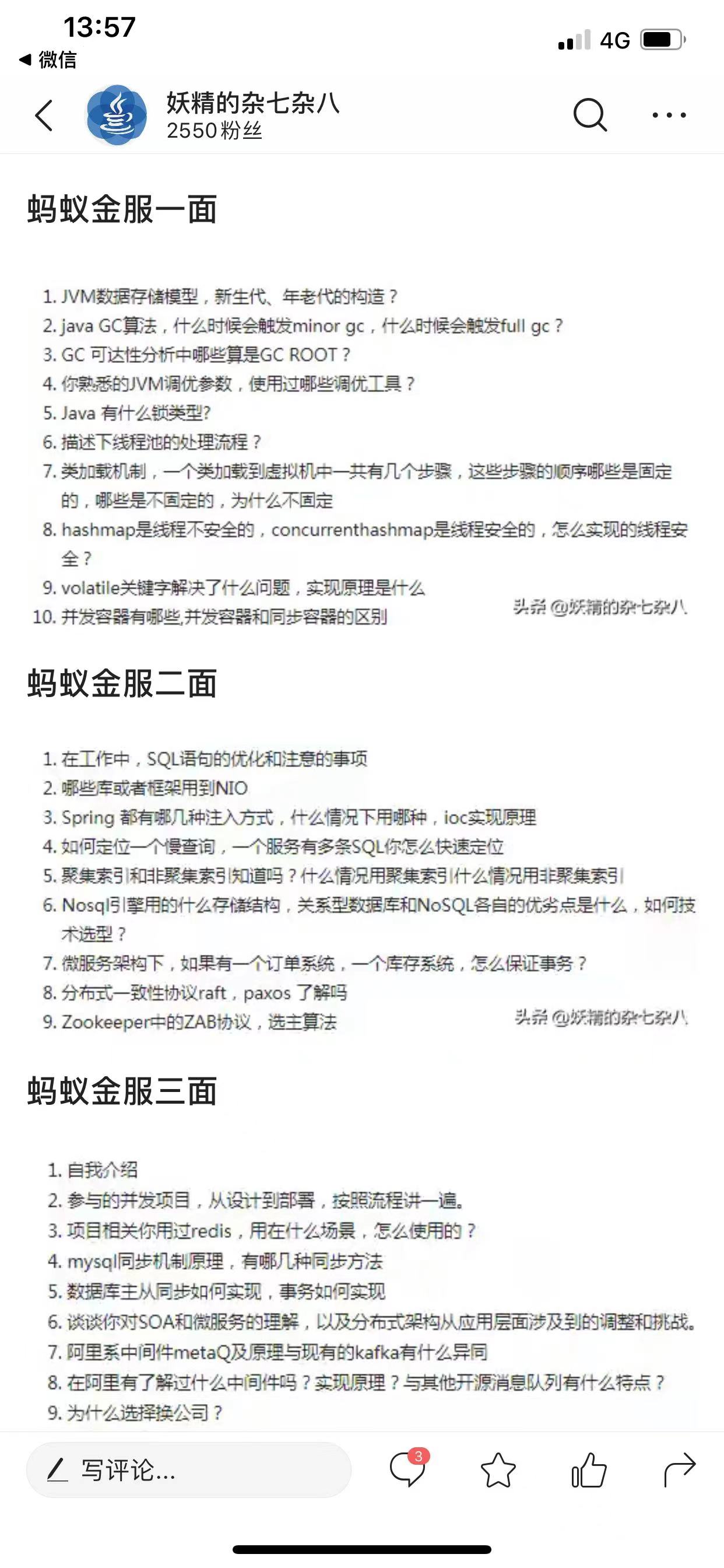
半同步复制

和全同步不同的是，半同步复制的逻辑是这样，从库写入日志成功后返回ACK确认给主库，主库收到至少一个从库的确认就认为写操作完成。

## 4、springboot的starter启动原理及自定义starter

参考：<https://blog.csdn.net/weixin_39565910/article/details/111169919>

# 大厂面试题

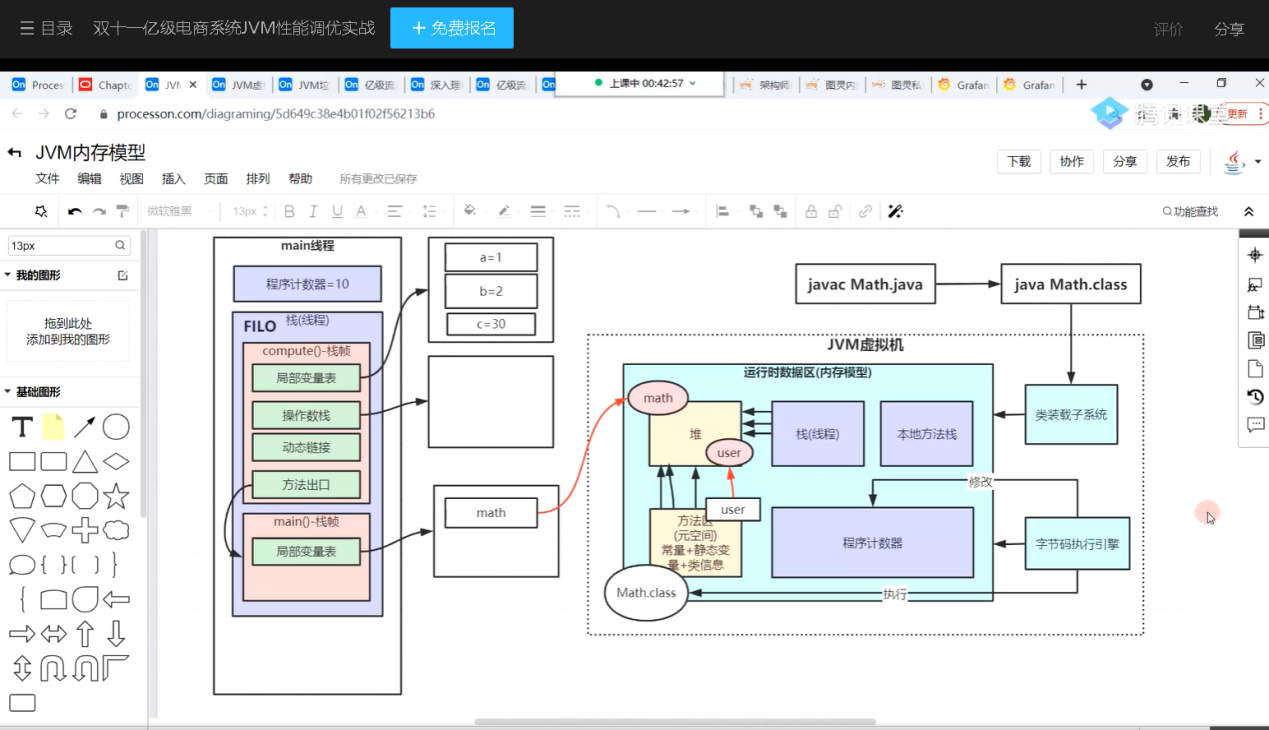


## 蚂蚁一面：

### 1、jvm数据存储模型，新生代/老年代构造

答：jvm数据存储模型：jvm内存分为线程私有和线程共享，其中线程私有分为栈、本地方法栈、程序计数器；线程共享分为堆、方法区（元空间）。

如图：



新生代、老年代构造：

新生代和老年代占比1：2。

年轻代分Eden区，Servior区。Servior区又分为From space区和To Space区。Eden区和Servior区的内存比为8:1。

应用程序只能使用一个新生代和复活代，当发生初级回收时，gc挂起程序，然后将新生代和复活代中的存货对象复制到另外一个复活代中，然后一次性消除新生代和复活代，将原来的非复活代标记成活动复活代。将在指定次数（默认15）回收后仍然存在的对象移动到老年代中，初级回收后，得到一个空的可用的新生代。

Full GC 是发生在老年代的垃圾收集动作，所采用的是标记-清除算法。

### 2、什么时候触发Minor GC ，Full GC

Minor GC ，Full GC 触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

### 3、gc可达性算法，哪些算gc root。

答：

1、栈中的对象引用；

2、本地方法栈中的对象引用；

3、方法区中常量引用的对象；

4、方法区中静态对象的引用。

### 4、你熟悉哪些jvm调优，用过哪些调优工具

答：熟悉jvm垃圾回收调优

### 5、java有什么锁类型。

答：

公平锁/非公平锁

可重入锁

独享锁/共享锁

互斥锁/读写锁

乐观锁/悲观锁

分段锁

偏向锁/轻量级锁/重量级锁

自旋锁

然后，在简单介绍各个锁的含义。

### 6、描述线程池的处理流程

1、线程池执行任务，查看核心线程是否已满，未满创建核心线程执行，已满执行第二步；

2、将任务放入任务队列中，查看任务队列是否已满，未满将任务加人队列，已满执行第三步；

3、查看最大线程数是否已满，未满创建临时线程执行，已满根据拒绝策略处理任务。

### 7、类加载机制，一个类加载到虚拟机经过哪几个步骤，这些步骤哪些是固定的，哪些是不固定的，为什么不固定。

答：

### 8、CurrentHashMap怎么实现的线程安全

在JDK1.7中ConcurrentHashMap，数据结构上，首先整体上是数组+链表+红黑树的结构与HashMap保持一致，其次取消了Segment分段锁的数据结构，取而代之的是Node，Node的value和next都是由volatile关键字进行修饰，可以保证可见性。将细化的粒度从段进一步降低到节点。线程安全实现上，采用CAS+Synchronized替代Segment分段锁。

ConcurrentHashMap会判断tabAt(tab, i = (n - 1) & hash)是不是 null，是的话就直接采用CAS进行插入，而如果不为空的话，则是synchronized锁住当前Node的首节点，这是因为当该Node不为空的时候，证明了此时出现了Hash碰撞，就会涉及到链表的尾节点新增或者红黑树的节点新增以及红黑树的平衡，这些操作自然都是非原子性的。从而导致无法使用CAS，当Node的当前下标为null的时候，由于只是涉及数组的新增，所以用CAS即可。

实在就是理解ConcurrentHashMap的put原理（源码）

参考：https://blog.csdn.net/Min\_Chander/article/details/104343528

