Map 接口有哪些实现
hashmap、linkedhashmap、treemap
hashmap 线程安全吗，会有什么问题。怎么实现一个线程安全的 map
不安全，jdk1.8 以前因为头插法可能导致循环链表，jdk1.8 之后虽然没有循环链表了但是仍然会导致修改丢失等问题。
hashtable、concurrenthashmap、collections.synchronizedMap
linkedhashmap 的应用场景
LRU
线程池的参数，他的流程是怎么样的
核心线程、最大线程、等待队列、拒绝策略、非核心线程存活时间
先请求过来创建核心线程，等核心线程创建达到指定值将请求放入等待队列，如果等待队列满了就创建非核心线程，等线程数达到最大线程就执行拒绝策略，等线程执行完毕，非核心线程空闲到指定时间就删除非核心线程
synchronized 和 reentranklock 的实现和区别
synchronized 是 JVM 层面的实现，锁类的话是 monitorenter，monitorexit 两个指令，锁方法的话是用 ACC_synchronized 标识来实现，在 JDK1.6 对 synchronized 进行了优化，有锁消除、锁粗化、轻量级锁、偏向锁、重量级锁等实现。
偏向锁的话就是在 MarkWord 上有一个当前持有线程的 id 和锁标识位，先检查锁标识位是不是偏向锁，然后是否指向自己线程，这样就少去了加锁步骤的开销，不是的话等持有线程运行到安全点会进行锁的膨胀，在 MarkWord 中会有一个指针指向持有线程栈帧中的对象，虽然有此时再有第三个线程过来或者自旋一定次数仍未获得轻量级锁，此时就要膨胀为重量级锁了，此时指针会指向 monitor 对象，其他线程想获得这个锁会阻塞。
reentranklock 的底层是 AQS，而 AQS 是 CLH 队列的增强，CLH 底层实现了一个虚拟节点的双向链表，每个节点会自旋查询前面一个节点的情况。AQS 在此基础上做了增强，支持可重入、可中断、不是一直自旋支持阻塞、支持非公平、支持独占和共享。reentranklock 默认是非公平可重入锁。
（这里我面试的时候直接就奔原理去了，上来就说了 synchronized 的原理，等说到 reentranklock 还没开始面试官就让我停了，我后来复盘才意识到，他仅仅是让我说区别罢了，那么补充下区别）
synchronized 是 JVM 的实现，是非公平可重入锁，代码执行完毕会在自动 放弃锁。
reentranklock 是 java 类的实现，默认是非公平可重入锁，但是可以实现公平锁，选择性通知。需要在 finally 去 unlock
怎么实现一个不可重入的锁
在 reentranklock 里面获取锁 state 会+1，,并判断是否是当前线程拿到的锁，那么直接把这个判断去了就行，如果 state>0，就不能再获取锁。
mybatis 的缓存实现，如果数据库进行修改了缓存怎么办
sqlsession 一级缓存和二级缓存。
就一级缓存而言，数据库进行修改会导致一级缓存失效，
就二级缓存而言，因为他的作用域是 namespace，如果有连表查询，而这个连表查询是写在某一个 namespace 里的，会导致脏数据的产生。
数据库和 redis 的一致性问题怎么解决的，如何保证强一致性
先更新数据库，再删除缓存。
那会出现一致性问题，
(1)缓存刚好失效

(2)请求 A 查询数据库，得一个旧值
(3)请求 B 将新值写入数据库
(4)请求 B 删除缓存
(5)请求 A 将查到的旧值写入缓存
怎么解决？ 答：异步双删，在请求 B 写入数据库后，开启一个异步线程，此线程等待一会后，再去删除缓存。这个等待时间 一般是查请求的执行时间加上一定的数值。
再问：仍然会有一致性问题，在 B 写入而缓存还没删除的时候。再答：用分布式锁，将写操作和缓存失效的操作变为原子操作。写数据时，首先尝试加锁。此锁锁定 1 秒时间。如果加锁成功，开始更新数据库。无论更新失败或者成功都解锁。如果成功，同时使缓存过期。

你项目中是怎么用 redis 的，为什么这么用，解决了什么问题，
发布/订阅来实现消息的发送
在访问主页的时候从数据库加载值时放到缓存中并设置失效时间，用作缓存，并且再将这个数据放到布隆过滤器里来解决缓存穿透的问题。
缓存击穿是怎么实现的？
分布式锁、我项目里的话只是简单的对查询加了一个排它锁。

秒杀系统怎么解决超卖问题，还有其他实现方式吗
我项目里的话只是用了 Spring 的声明式事务，相当于在数据库层面加排它锁来实现。
其他实现方式的话比如乐观锁，但是冲突会比较大。
还有就是引入 redis 的 String，借助 redis 串行化的特点，比如有 100 件商品，设置值为 100，每次请求进来判断是否大于 0，如果大于 0 把这个值-1，去访问数据库减库存。
可以把剩下一部分请求（比如 50 个）加入到消息队列中，剩下一部分全部拒绝掉。消息队列中的这部分请求用来处理可能出现的问题（因为你秒杀肯定是一个订单支付的操作，然后再去扣库存嘛，先是订单支付，你去操作数据库了，这个时候哪 100 个里有有一个人钱不够了，导致订单支付失败，然后后续操作都没法进行，这个时候我们消息队列里的 50 个请求就有用了。我是这么理解的，不知道具体情况怎么样。。）

redis 的分布式锁有了解过吗，怎么实现的
redission 就可以实现，原理的话就是用 setnx 来创建锁并设置锁的过期时间，然后有一个 watchdog 来对锁进行加时，方式锁到期请求还没处理完的情况，最后使用 lua 脚本将判断锁和删除锁写一起使其变成一个原子操作来完成锁的释放。
redission 是可重入的吗？如果要实现分布式锁的可重入怎么实现
是可重入的、可以通过请求带来一个 UUID，然后保存起来，下次再来请求先判断是不是这个 UUID，如果是话就重入。
dubbo 的流程，默认用的什么序列化方法，为什么这么用
服务提供方在容器中启动，将服务暴露到注册中心，消费者向注册中心订阅自己所需的服务，注册中心会提供消费者的地址列表，如果有变更，以 zookeeper 为例，也会较快收到变更的信息，从地址列表中，基于负载均衡算法，选一个提供者。消费者和提供者都会定时向监控中心返回方法的调用次数和调用时间。
默认序列化方法：hession2，为什么？不知道
数据库的隔离级别，用的什么引擎，他的 ACID，这四个特性每一个 MySQL 是怎么保证的
readuncommited
readcommited

repeatedtable
seriazable
用的 Innodb，持久性由 redolog 保证，就算断电也可以根据 redolog 进行未完成的事务，原子性由 undolog 保证，等 undolog 提供了回滚的特性，隔离性由 MVCC 机制保证，通过对每行数据加一个当前版本号和删除版本号的隐藏字段来保证事务间的隔离，一致性由前面三者保证，C 是目的，AID 是手段。
权限的实现，你自己的实现和 shiro，springsecurity 有什么区别，你会怎么优化
没了解过 shiro 和 springsecurity
遇到过最难的事情是什么
还有什么想问我的
了解到蘑菇街会有一个缓存中台，然后数据库都是单表查询，查出来之后再去拼接？是自己实现的嘛还是有现成的框架。
看业务，有些是这么实现的，是公司自己实现的。

这是跟我一起面试的同学的面经，我三点钟他五点钟。
1.上来就手撕，链表快排（不能把链表转为数组）
[快速排序——链表快排](https://blog.csdn.net/u012114090/article/details/81751259)
2.java 对象创建过程，引用和对象分别是什么时候被 gc 的

类加载检查-->分配内存-->初始化零值-->设置对象头-->执行 init 方法
强引用：不会被 gc
软引用：内存不足时候 gc
弱引用:每次垃圾回收时
虚引用：gc 时间未知，它的作用在于跟踪垃圾回收过程，在对象被收集器回收时收到一个系统通知。 当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在垃圾回收后，将这个虚引用加入引用队列，在其关联的虚引用出队前，不会彻底销毁该对象。 所以可以通过检查引用队列中是否有相应的虚引用来判断对象是否已经被回收了。
对象是什么时候被 gc 的？
由引用计数法或者可达性分析来判断对象是否可以被 gc

3.半初始化过程
emm？ 这是啥 我不知道
4.说说 volatile 关键字吧，有哪几种屏障？
volatile 主要保证的是可见性和防止指令重排。
一般可见性方面，volatile 可以使得本线程内的缓存失效，也就是读 volatile 变量的时候直接从内存中读，而写 volatile 变量的时候直接写入内存
volatile 提供了四个内存屏障，loadstore 读写、loadload 读读、storestore 写写、storeload 写读

来保证指令不会重排序
来看下重排序可能导致的危害：
线程 A：
context = loadContext();
inited = true;
线程 B：
while(!inited){ //根据线程 A 中对 inited 变量的修改决定是否使用 context 变量
sleep(100);
}
doSomethingwithconfig(context);

假设线程 A 中发生了指令重排序:
inited = true;
context = loadContext();
那么 B 中很可能就会拿到一个尚未初始化或尚未初始化完成的 context,从而引发程序错误。

还有可能比如不安全发布的问题（比如单例模式中不写 volatile 的话可能会拿到由于重排序导致的"半个单例"，比如以下情况导致的尚未初始化的对象。

memory = allocate(); //1：分配对象的内存空间
instance = memory; //3：设置 instance 指向刚分配的内存地址（此时对象还未初始化）
ctorInstance(memory); //2：初始化对象
）

5.gc 过程，为什么要分代
为什么分代：对于年轻代的对象，由于对象来的快去得快，垃圾收***比较频繁，因此执行时间一定要短，效率要高，因此要采用执行时间短，执行时间的长短只取决于对象个数的垃圾回收算法。但是这类回收器往往会比较浪费内存，比如 Copying GC，会浪费一半的内存，以空间换取了时间。
对于老年代的对象，由于本身对象的个数不多，垃圾收集的次数不多，因此可以采用对内存使用比较高效的算法。

gc 过程：
这里就简单说说 CMS 和 G1 的区别吧
CMS：
一开始先标记 GC Roots 能直接标记到的对象（会停顿）
然后开始延伸（不会停顿）
再之后修正并发标记期间,因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录（会停顿）
最后就是清理啦,因为不需要移动存活对象,所以这个阶段也是可以与用户线程同时并发的。（不会停顿，浮动垃圾也是这步产生的，cms 用的是标记清楚）
G1：先是标记 GC Roots 能直接关联到的对象（会停顿）
扫描整个堆里的对象图(并发标记，不停顿)
对用户线程进行暂停，处理并发标记的时候变动的对象（开始停顿）

筛选回收垃圾（根据设置的数值来控制回收的内存中垃圾大小）
可以看到最后一步 G1 是停顿的而 CMS 不会停顿，这也是 CMS 会产生浮动垃圾的原因

6.synchronized 和 lock 哪个效率高，为什么，底层实现
1.6 之后差不多，为什么？因为 1.6 之后也可以用 cas 来实现啦（如果没有膨胀到重量级锁的话）。我的面经已经讲过啦
7.hashmap 为什么在链表长度为 8 时转化为红黑树。
泊松分布