作者：寻筝

链接：https://www.nowcoder.com/discuss/56924

来源：牛客网

一面（23min）

**自我介绍**

**项目中最自豪的部分**

也没什么太自豪的，就是在移动端开发的时候不存在cookie和session，然后用redis存了一下验证码感觉还不错。

**讲一讲ArrayList和LinkedList**

ArrayList底层实现是数组，并且每次扩容扩容1.5倍，常用在查询较多的场景中。而LinkedList底层实现是链表常用在增删比较多的场景

**你说你对锁有了解，说一说你最熟悉的两个锁**

当时太紧张就随口说了两个，乐观锁，悲观锁。乐观锁：每次执行事务的时候都是先执行在检查是否有其他是否在执行。而悲观锁如行锁，表锁，都是先锁定再执行事务。通常在并发量比较大的情况下用悲观锁，并发量小的情况下用乐观锁

1. 乐观锁（optimistic locking）表现出大胆、务实的态度。使用乐观锁的前提是， 实际应用当中，发生**冲突的概率比较低**。他的设计和实现直接而简洁。 目前Web应用中，乐观锁的使用占有绝对优势。从本质上来讲，乐观锁并没有像悲观锁那样使用数据库的锁机制。乐观锁通过在表中增加一个计数字段，来表示当前记录被修改的次数（版本号）。然后在更新、删除前通过比对版本号来实现乐观锁。
2. 正如其名字，悲观锁（pessimistic locking）体现了一种谨慎的处事态度。其流程如下：1）在对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。2）如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 具体响应方式由开发者根据实际需要决定。3）如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。4）其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。5）悲观锁确实很严谨，有效保证了数据的一致性，在C/S应用上有诸多成熟方案。 但是他的缺点与优点一样的明显：6）悲观锁适用于可靠的持续性连接，诸如C/S应用。 对于Web应用的HTTP连接，先天不适用。7）锁的使用意味着性能的损耗，在高并发、锁定持续时间长的情况下，尤其严重。 Web应用的性能瓶颈多在数据库处，使用悲观锁，进一步收紧了瓶颈。8）非正常中止情况下的解锁机制，设计和实现起来很麻烦，成本还很高。9）不够严谨的设计下，可能产生莫名其妙的，不易被发现的， 让人头疼到想把键盘一巴掌碎的死锁问题。
3. 总体来看，悲观锁不大适应于Web应用，Yii团队也认为悲观锁的实现过于麻烦， 因此，ActiveRecord也没有提供悲观锁。

**说一下synchronied和ReentrantLock的区别**

synchronied是JVM级别的，而ReentrantLock是api级别的，JVM会对synchronied做出相应的优化，锁消除：当JVM判定该资源不会被其他线程争夺的时候就会消除该锁，还有锁自旋，自适应锁（面试官速度很快，还没等我来得及解释ReentrantLock就跳下一个问题了，也许是这个超底层的回答比较满意吧）

**讲一下ThreadLoacal吧**

ThreadLocal底层实现是一个Map结构的表，key是Thread.currentThread（）,而Value则是我们想要保存的对象

**它（ThreadLocal）有什么用，可以举一个例子吗？（Spring企业实战开发 P300）**

可以在每个线程都拥有一个各自的副本，封装起来不被影响，作用的话，可以做一个类似卖票的系统，每个售票厅（线程）只能出售100张票，不能多不能少

**用过exchange（）吗**

不好意思，只敲过一次例子，不太了解

**说一说JVM内存模型吧**

我：栈区：每次方法的执行都会有一次栈帧的生成，并且其中有局部变量表。堆区：每次new出来的对象都保存在这儿。本地方法栈：通过C/C++调用系统接口的一些方法，方法区：类的信息 静态变量都在这儿，还有程序计数器

**说一说AOP吧**

AOP就是面向切面编程，并解释了下列术语，joinpoint，pointcut，aspect，advice，并说了一下实现的原理，若实现了接口就是JDK动态代理否则就用CGLIB

**了解asm吗？**

我反应了半天才想起这是一个jar包，不过从没了解过就说不知道了

**String str1=new String("abc"),String str2=new String("abc"); str1==str2返回什么**

当时太紧张，注意力一直放在abc字符串上了忽略了这是new了一个对象，一直回答true，反问我很久我还是没反应过来，有点没发挥好

（补充：String str1=new String("abc")创建了几个对象？答案两个，一个在常量池中，一个在堆中）

**知道如何让线程同步吗？**

wait，notify，notifyAll，await，signal，siganlAll，countdownLatch，cyclicbarrer

**说一说countdownlatch，cyclicbarrer**

都可以等几个线程就绪后再执行后续操作，区别就是cyclicbarrer是可以复用的，而countdownlatch不可以

**（补充**：CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。

字面意思回环栅栏，可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。我们暂且把这个状态就叫做barrier，当调用await()方法之后，线程就处于barrier了。**）**

举个例子？

<br>可以假设以下场景，等9个跑步选手（线程）同时在起跑线就绪了再开始起跑

可以实习多久

你的职业规划

你有什么想问我的吗？<br>

最后面试官简单评价了一下：看得出来你平时看了很多书，不过前面有一两个答错了，你开始很紧张，后来好了一些。

二面（14min）

自我介绍

**讲一下hashmap吧**

hashmap底层是一个key-value结构的**entry数组+链表**，在put的时候根据key的hashcode的值在hash一次，然后根据得到 的hash值再和hashmap的size取余操作定位到要插入的哈希桶，如果为null则插入，否则遍历哈希桶后面的链表如果有key相等的则覆盖，否则添加到尾部，然后检查是否达到treshold需要扩容，并且每次扩容都是扩大两倍

**hashmap可以插入null值吗**

可以

**为什么呢？你不是说根据key的hashcode插入吗**

<br>不好意思，我只是知道可以，具体情况不太清楚

你说你会MySQL调优SQL语句，那你一般是怎么做的呢？

一般是先查看慢SQL日志，然后explain一下该查询语句，看是否索引失效或者是没建索引

**我们都知道hashmap是线程不安全的，那么为什么线程不安全呢？**

因为当两个线程同时对hasmap扩容的时候,会导致循环链表,使cpu达到100%，甚至宕机

**你说你对多线程有所了解，那你说一下线程池的实现原理吧**

好的，在线程池内部一共有这么几个参数，corepoolsize：核心线程数，maxpoolsize: 最大线程数,workquee:一个放预备线程的阻塞队列，time:允许的空闲时间，handler:异常处理器。当向线程池加入一个线程的时候，如果运行的线程数小于corpoolsize，则直接执行此线程,如果等于了corepoolsize则提交给阻塞队列阻塞等待，如果阻塞队列已满，则执行器继续增加新的线程来执行任务，如果达到了maxpoolsize则执行handler的异常处理方法，有以下几种方法抛出异常，忽略，提交线程者运行，删除等待最久的线程

**说一说垃圾回收的时候，如何判定一个对象是否需要回收？**

<br>一共有两种算法，一种是引用计数算法，每被引用一次计数器就+1，当引用为0就回收，不过这种算法有一个弊端就是，当对象之间互相引用的时候就永远不会回收，所以用的最多的还是GC ROOTS可达性分析，看对象是否被GC ROOTS引用，那么哪些可以作为GC ROOTS呢？

有以下四种：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象

说一说你的项目中登录是怎么做的

大概就是redis做验证码缓存，MD5加密密码（有点长。。省略）

说说cookie和session吧

session来自于cookie，session可以放在cookie里，并且cookie存在客户端，session存在服务器端（简单回答）

你说你用过redis，那你知道redis可以做持久化吗？

<br>可以

哪两种？

<br>RDB 和 AOF

总结

一面有点紧张，没发挥好，不过能从底层回答的都从底层回答了，问的浅，答得较深

二面运气不错，恰好问到了擅长的部分，问的是要比一面更难,不过还算是答的比较好

最后感谢程涛学长提供的内推机会，以及一直以来在Java学习上的帮助！

面经2

问题  
自己实现IOC容器，怎么实现  
某个场景下，事务会回滚么  
exception类有看过么？继承了什么类？  
联合索引，最左优化  
**秒杀场景怎么实现**  
设计思路

将请求拦截在系统上游，降低下游压力：秒杀系统特点是并发量极大，但实际秒杀成功的请求数量却很少，所以如果不在**前端拦截**很可能造成数据库读写锁冲突，甚至导致死锁，最终请求超时。

**充分利用缓存**：利用缓存可极大提高系统读写速度。

**消息队列**：消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

**前端方案**

浏览器端(js)：

页面静态化：将活动页面上的所有可以静态的元素**全部静态化**，并尽量**减少动态元素**。通过CDN来抗峰值。

禁止重复提交：用户提交之后按钮置灰，**禁止重复提交**

用户限流：在某**一时间段内只允许用户提交一次请求**，比如可以采取IP限流，或者在秒杀之前增加游戏或者问题环节进行消峰操作

那么后端的数据库在高并发和超卖下会遇到什么问题呢？主要会有如下3个问题：（**主要讨论写的问题**，读的问题通过增加cache可以很容易的解决）

　　I：　首先MySQL自身对于高并发的处理性能就会出现问题，一般来说，MySQL的处理性能会随着并发thread上升而上升，但是到了一定的并发度之后会出现明显的拐点，之后一路下降，最终甚至会比单thread的性能还要差。

　　II： 其次，超卖的根结在于减库存操作是一个事务操作，需要先select，然后insert，最后update -1。最后这个-1操作是不能出现负数的，但是当多用户在有库存的情况下并发操作，出现负数这是无法避免的。

　　III：最后，**当减库存和高并发碰到一起的时候，由于操作的库存数目在同一行，就会出现争抢InnoDB行锁的问题，导致出现互相等待甚至死锁，从而大大降低MySQL的处理性能，最终导致前端页面出现超时异常。**

**后端方案**

服务端控制器层(网关层)

限制uid（UserID）访问频率：我们上面拦截了浏览器访问的请求，但针对某些恶意攻击或其它插件，在服务端控制层需要**针对同一个访问uid，限制访问频率**。

服务层

上面只拦截了一部分访问请求，当秒杀的用户量很大时，即使每个用户只有一个请求，到服务层的请求数量还是很大。比如我们有100W用户同时抢100台手机，服务层并发请求压力至少为100W。

**采用消息队列缓存请求**：既然服务层知道库存只有100台手机，那完全没有必要把100W个请求都传递到数据库啊，那么可以先把这些请求都写到消息队列缓存一下，**数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束**。

**利用缓存应对读请求：**对类似于12306等购票业务，是典型的读多写少业务，大部分请求是查询请求，所以可以利用缓存分担数据库压力。

利用缓存应对写请求：缓存也是可以应对写请求的，比如我们就可以把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再**通过后台进程把Redis中的用户秒杀请求同步到数据库中**。

超卖问题：<http://blog.csdn.net/zhoudaxia/article/details/38067003>

**解决方案1：**

　　将**存库**从MySQL前移到Redis中，所有的写操作放到内存中，由于Redis中不存在锁故不会出现互相等待，并且由于Redis的写性能和读性能都远高于MySQL，这就解决了高并发下的性能问题。然后通过队列等异步手段，将变化的数据异步写入到DB中。

　　优点：解决性能问题

　　缺点：没有解决超卖问题，同时由于异步写入DB，存在某一时刻DB和Redis中数据不一致的风险。

**解决方案2：**

　　引入队列，然后将所有写DB操作在单队列中排队，完全串行处理。当达到库存阀值的时候就不在消费队列，并关闭购买功能。这就解决了超卖问题。

　　优点：解决超卖问题，略微提升性能。

　　缺点：性能受限于队列处理机处理性能和DB的写入性能中最短的那个，另外多商品同时抢购的时候需要准备多条队列。

**解决方案3：**

　　将写操作前移到Memcached中，同时利用Memcached的轻量级的锁机制CAS来实现减库存操作。

　　优点：读写在内存中，操作性能快，引入轻量级锁之后可以保证同一时刻只有一个写入成功，解决减库存问题。

　　缺点：没有实测，基于CAS的特性不知道高并发下是否会出现大量更新失败？不过加锁之后肯定对并发性能会有影响。

**解决方案4：**

　　将提交操作变成两段式，先申请后确认。然后利用Redis的原子自增操作（相比较MySQL的自增来说没有空洞），同时利用Redis的事务特性来发号，保证拿到小于等于库存阀值的号的人都可以成功提交订单。然后数据异步更新到DB中。

　　优点：解决超卖问题，库存读写都在内存中，故同时解决性能问题。

　　缺点：由于异步写入DB，可能存在数据不一致。另可能存在少买，也就是如果拿到号的人不真正下订单，可能库存减为0，但是订单数并没有达到库存阀值。

各种数据结构  
线程池，多线程  
  
  
**String设置不可变是为了什么**

1.字符串常量池的需要。

2.允许String对象缓存hashCode

3.安全性。String类将常用来作为参数，如URL，文件路径等

**String怎么实现不可变的**

String的成员变量都用private final修饰（字符数组，offset，count，hash只有private修饰），没有提供getter和setter方法，一旦初始化就无法对其进行修改

**String对象真的不可变吗？**

从上文可知String的成员变量是private final 的，也就是初始化之后不可改变。那么在这几个成员中， value比较特殊，因为他是一个引用变量，而不是真正的对象。value是final修饰的，也就是说final不能再指向其他数组对象，那么我能改变value指向的数组吗？ 比如将数组中的某个位置上的字符变为下划线“\_”。 至少在我们自己写的普通代码中不能够做到，因为我们根本不能够访问到这个value引用，更不能通过这个引用去修改数组。

那么用什么方式可以访问私有成员呢？ 没错，用反射， 可以反射出String对象中的value属性， 进而改变通过获得的value引用改变数组的结构。

在这个过程中，s始终引用的同一个String对象，但是再反射前后，这个String对象发生了变化， 也就是说，通过反射是可以修改所谓的“不可变”对象的。但是一般我们不这么做。这个反射的实例还可以说明一个问题：如果一个对象，他组合的其他对象的状态是可以改变的，那么这个对象很可能不是不可变对象。例如一个Car对象，它组合了一个Wheel对象，虽然这个Wheel对象声明成了private final 的，但是这个Wheel对象内部的状态可以改变， 那么就不能很好的保证Car对象不可变。

**http怎么传cookie的**

cookie和session，参考博客：<http://blog.csdn.net/fangaoxin/article/details/6952954/>

作者：知乎用户

链接：https://www.zhihu.com/question/19786827/answer/28752144

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

1. 由于HTTP协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session.典型的场景比如购物车，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个Session是保存在服务端的，有一个唯一标识。在服务端保存Session的方法很多，内存、数据库、文件都有。集群的时候也要考虑Session的转移，在大型的网站，一般会有专门的Session服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如Memcached之类的来放 Session。2. 思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。3. Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。所以，总结一下：Session是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

如何实现只用缓存，cache-only  
**abcd比例1234，返回基于比例的随机数**

根据生成的随机数映射成不同的值，比如要1:1生成1和2，那么可以让生成的随机数为0-0.5输出1,0.5-1输出2

**为什么要等待2msl时间（时间MSL叫做最长报文段寿命）**

**MSL：maximum augment lifetime**

第一：为了保证A发送的最后一个ACK报文段能够到达B。这个ACK报文段有可能丢失，因而使处在LAST-ACK状态的B收不到对己发送的FIN+ACK报文段的确认。

第二，防止“已失效的连接请求报文段”出现在本连接中。A在发送完最后一个ACK报文段后，再经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段

**总的来说：**

1。防止上一次连接中的包，迷路后重新出现，影响新连接（经过2MSL，上一次连接中所有的重复包都会消失）

2。可靠的关闭TCP连接

**301和302之间的区别**

一．官方说法

301，302 都是HTTP状态的编码，都代表着某个URL发生了转移，不同之处在于：

301 redirect: 301 代表永久性转移(Permanently Moved)。

302 redirect: 302 代表暂时性转移(Temporarily Moved )。

这是很官方的说法，那么它们的区别到底是什么呢？

二．现实中的差异

2.1.对于用户

301，302对用户来说没有区别，他们看到效果只是一个跳转，浏览器中旧的URL变成了新的URL。页面跳到了这个新的url指向的地方。

2.2.对于引擎及站长

2.2.1 302

302转向可能会有URL规范化及网址劫持的问题。可能被搜索引擎判为可疑转向，甚至认为是作弊。

网址规范化

请参见：http://www.chinamyhosting.com/seoblog/2006/04/10/url-canonicalization/

网址劫持

302重定向和网址劫持（URL hijacking）有什么关系呢？这要从搜索引擎如何处理302转向说起。从定义来说，从网址A做一个302重定向到网址B时，主机服务器的隐含意思是网址A随时有可能改主意，重新显示本身的内容或转向其他的地方。大部分的搜索引擎在大部分情况下，当收到302重定向时，一般只要去抓取目标网址就可以了，也就是说网址B。

实际上如果搜索引擎在遇到302转向时，百分之百的都抓取目标网址B的话，就不用担心网址URL劫持了。问题就在于，有的时候搜索引擎，尤其是Google，并不能总是抓取目标网址。为什么呢？比如说，有的时候A网址很短，但是它做了一个302重定向到B网址，而B网址是一个很长的乱七八糟的URL网址，甚至还有可能包含一些问号之类的参数。很自然的，A网址更加用户友好，而B网址既难看，又不用户友好。这时Google很有可能会仍然显示网址A。

由于搜索引擎排名算法只是程序而不是人，在遇到302重定向的时候，并不能像人一样的去准确判定哪一个网址更适当，这就造成了网址URL劫持的可能性。也就是说，一个不道德的人在他自己的网址A做一个302重定向到你的网址B，出于某种原因， Google搜索结果所显示的仍然是网址A，但是所用的网页内容却是你的网址B上的内容，这种情况就叫做网址URL劫持。你辛辛苦苦所写的内容就这样被别人偷走了。

**如果服务器端有太多time-wait状态怎么办**

通过linux配置

**想要在程序退出时输出一句话怎么实现，runtime类，或者shutdown hook，**

对于简单的应用系统，我们直接可以在System.exit(0)代码执行前，添加需要在应用程序退出前需要完成的工作，如：关闭网络连接，关闭数据库连接等。  
然而，对于比较复杂的多线程应用，线程运行的状态较复杂，我们就很难预料程序何时结束，如何能在应用程序结束事件到来时，处理我们要做的工作呢？这就用到了**Java对应用程序的退出的事件处理机制**。  
对当前应用程序对象的获得，Java通过Runtime静态方法：Runtime.getRuntime()通过Runtime的 void addShutdownHook(Thread hook) 法**向Java虚拟机注册一个shutdown钩子事件**，这样**一旦程序结束事件到来时，就运行线程hook**，我们在实际应用时候，只要将程序需要完成之前做的一些工作直接通过线程hook来完成。

**说到了一个优雅停机的概念  
Springmvc与Spring的关系，谁来加载需要的Controller**

SpringMVC是建立在Spring应用平台之上的MVC模型

又问了一遍线程池，逻辑  
有没有自己写过注解  
有没有用过枚举类  
分布式事务

作者：低调...  
链接：<https://www.nowcoder.com/discuss/38082>  
来源：牛客网

自三月以来，从春招实习，一路走到现在。很累，也很有收获，分享一波面经。

总结起来就是一句话：我的学校在211里可能不占优势，但是最终的结果，取决于你自己想成为什么样的人

网易：

考拉海购Java岗  两轮技术面+HR面试

一面：

1、自我介绍

2、实习期间项目相关，然后转到Spring

3、Spring 在项目中用了哪些特性，IOC，及Kafka监听接口相关实现。

4、ThreadPool，Jdk原生线程池，四个参数详细解释原理，当线程池中poolSize达到corePoolSize且阻塞队列已满，再来一个任务，如何处理

**5、多线程实现同步的方式，互斥同步，非阻塞同步，**

1. 同步方法。用synchronized关键字修饰方法
2. 同步代码块。用synchronized关键字修饰代码块
3. 使用volatile实现线程的同步，注意，不能保证原子性，只能保证可见性
4. 使用ReentrantLock
5. 使用ThreadLocal管理变量，即每一个使用该变量的线程都获得该变量的副本

6、解释Synchronize关键字的锁优化技术，偏向锁，轻量级锁，重量级锁，这些锁是如何存储的，偏向锁撤销升级为轻量级锁的过程，结合源码解释

7、volatile关键字语义，内存屏障如何实现，JMM对内存屏障做了哪些优化，volatile的语义增强

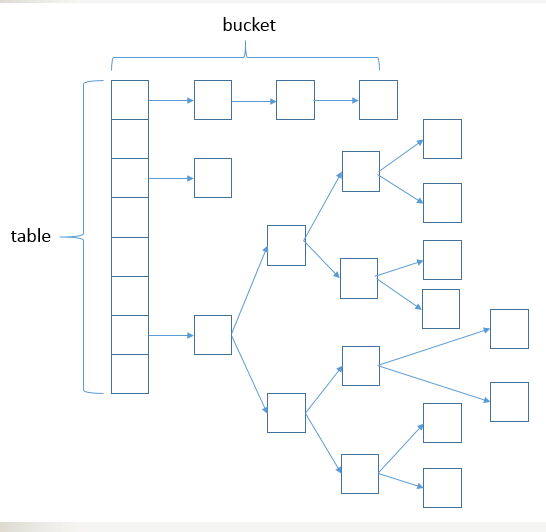
8、指令重排序相关及其happen-before时，数据依赖等

9、HashMap底层数据结构，如何处理hash冲突，为何HashMap的大小要设置为2的n次幂，为什么IndexFor方法里，需要hash&length-1，为什么HashMap允许null值，resize()过程，多线程下resize为什么会出现死循环，详细解释，结合源码

HashMap中indexFor方法，当找某个key的位置的时候，正常的思路是去余，但是设计者用的是位运算符，效率很高（return h&(length-1)）. 又是位运算，高帅富啊！这里h是通过K的hashCode最终计算出来的哈希值，并不是hashCode本身，而是在hashCode之上又经过一层运算的hash值，length是目前容量。这块的处理很有玄机，与容量一定为2的幂环环相扣，当容量一定是2^n时，h & (length - 1) == h % length，它俩是等价不等效的，位运算效率非常高，实际开发中，很多的数值运算以及逻辑判断都可以转换成位运算，但是位运算通常是难以理解的，因为其本身就是给电脑运算的，运算的是二进制，而不是给人类运算的，人类运算的是十进制，这也是位运算在普遍的开发者中间不太流行的原因(门槛太高)。这个等式实际上可以推理出来，2^n转换成二进制就是1+n个0，减1之后就是0+n个1，如16 -> 10000，15 -> 01111，那根据&位运算的规则，都为1(真)时，才为1，那0≤运算后的结果≤15，假设h <= 15，那么运算后的结果就是h本身，h >15，运算后的结果就是最后三位二进制做&运算后的值，最终，就是%运算后的余数，我想，这就是容量必须为2的幂的原因。HashTable中的实现对容量的大小没有规定，最终的bucketIndex是通过取余来运算的。注：我在考虑这样做是否真的是最好，规定容量大小一定为2的幂会浪费许多空间成本，而时间上的优化针对当今越来越牛逼的CPU是否还提升了许多，有些资料上说，CPU处理除法和取余运算是最慢的，这个应该取决于语言调用CPU指令的顺序和CPU底层实现的架构，也是有待验证，可以用Java写段程序测试一下，不过我想，CPU也是通过人来实现，人脑运算的速度应该是加法>减法>乘法>除法>取模(这里指的是正常的算法，不包括速算)，CPU也应是如此。

1. **void** transfer(Entry[] newTable) {
2. Entry[] src = table;
3. **int** newCapacity = newTable.length;
4. **for** (**int** j = 0; j < src.length; j++) {
5. Entry<K,V> e = src[j];
6. **if** (e != **null**) {
7. src[j] = **null**;
8. **do** {
9. Entry<K,V> next = e.next;
10. **int** i = indexFor(e.hash, newCapacity);
11. e.next = newTable[i];
12. newTable[i] = e;
13. e = next;
14. } **while** (e != **null**);
15. }
16. }
17. }

10、HashMap jdk1.8之后，为何转链式结构为红黑树，为何长度为8的时候，才转成红黑树



11、NIO了解吗？   这个。。没答好，属于知识空白

12、阐述一下快排基本原理。easy

13、手写单例模式。分别问了多线程情况如何处理，什么时候可以破坏单例模式，怎么防止单例模式被破坏

14、JMM内存模型，如何保证缓存一致性。

二面：

1、自我介绍

2、介绍实习项目

3、Kafka监听接口如何实现，为何采用Kafka而不是用其他消息队列

4、实现Kafka监听接口的时候，遇到哪些坑，采取了哪些措施或者解决方案

5、Kafka的基本概念，及特点和优势。offset+zk+持久化分区日志顺序存储之类的

6、MySQL了解吗，介绍一下InnoDB

7、JVM了解吗，介绍一下，讲了垃圾回收机制

8、Full GC和Minor GC区别，及各自的触发条件

9、说说CMS垃圾回收器，及其适用场景

HR面：

HR小姐姐，真的很漂亮=。=

HR面比较常规，相比于那些让讲笑话之类的，中规中矩，有没有实习，在哪儿实习，负责什么系统，为何让你独立负责，是否拿到offer，offer薪资多少，哪里人，以后打算在哪儿发展，是否接受入职前实习，是否接受996。