**大公司面试**

Table of Contents

[大公司面试题 3](#_Toc16260442)

[考拉面经1： 3](#_Toc16260443)

[4、（没问题）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 3](#_Toc16260444)

[5、多线程实现同步的方式，互斥同步，非阻塞同步， 4](#_Toc16260445)

[6、reenterlocker和sychronize的区别？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 4](#_Toc16260446)

[1. （没问题）说一下 synchronized 底层实现原理？ 4](#_Toc16260447)

[2.（没问题）多线程中 synchronized 锁升级的原理是什么？ 6](#_Toc16260448)

[4.（没问题）reenterlocker和sychronize的区别？ 8](#_Toc16260449)

[5.为什么ConcurrentHashMap 1.8为什么要使用CAS+Synchronized取代Segment+ReentrantLock 13](#_Toc16260450)

[7、（没问题）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 13](#_Toc16260451)

[8、指令重排序相关及其happen-before时，数据依赖等 15](#_Toc16260452)

[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 15](#_Toc16260453)

[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 15](#_Toc16260454)

[11、NIO了解吗？   这个。。没答好，属于知识空白 15](#_Toc16260455)

[12、阐述一下快排基本原理。easy 15](#_Toc16260456)

[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 15](#_Toc16260457)

[14、JMM内存模型，如何保证缓存一致性。（重要）（需要搞清楚） 15](#_Toc16260458)

[15.reenterlocker和sychronize的区别？ 17](#_Toc16260459)

[1、自我介绍 17](#_Toc16260460)

[7、JVM了解吗，介绍一下，讲了垃圾回收机制 17](#_Toc16260461)

[8、Full GC和Minor GC区别，及各自的触发条件 17](#_Toc16260462)

[9、说说CMS垃圾回收器，及其适用场景 17](#_Toc16260463)

[考拉面经2： 17](#_Toc16260464)

[自我介绍 17](#_Toc16260465)

[2、spring java反射 17](#_Toc16260466)

[4、班级表和学生表，利用sql 语句查询每个班级的学生人数 17](#_Toc16260467)

[4、i++原子操作 18](#_Toc16260468)

[5、hashmap和hashtable区别，以及concurrenthashmap 18](#_Toc16260469)

[6、zookeeper 实现分布式锁 18](#_Toc16260470)

[7、团购系统的表设计 20](#_Toc16260471)

[8、（没问题）悲观锁 和乐观锁 以及sql如何实现 20](#_Toc16260472)

[什么是AbstractQueuedSynchronizer？ 21](#_Toc16260473)

[JVM优化经验（重要重要重要）（需要搞清楚） 22](#_Toc16260474)

[360面试总结 23](#_Toc16260475)

[为什么netty的线程池中，线程的个数是cpu的2倍？ 23](#_Toc16260476)

[工作线程数”的设置依据是什么，到底设置为多少能够最大化CPU性能。 23](#_Toc16260477)

[hdfs中的读写操作？ 24](#_Toc16260478)

[为什么三次握手 四次挥手？ 24](#_Toc16260479)

[接口和抽象类的区别？接口中是否可以放属性？ 26](#_Toc16260480)

[hashmap中key有什么要求？ 26](#_Toc16260481)

[hashmap的get的流程？ 26](#_Toc16260482)

[equal 和等号的区别 26](#_Toc16260483)

[java8 9的新功能 26](#_Toc16260484)

[一个线程接着一个线程执行 有什么方法？ 26](#_Toc16260485)

[创建线程的方式 28](#_Toc16260486)

[如何停止一个线程？ 28](#_Toc16260487)

[缓存一致性 28](#_Toc16260488)

大公司面试题

考拉面经1：

考拉海购Java岗  两轮技术面+HR面试

一面：

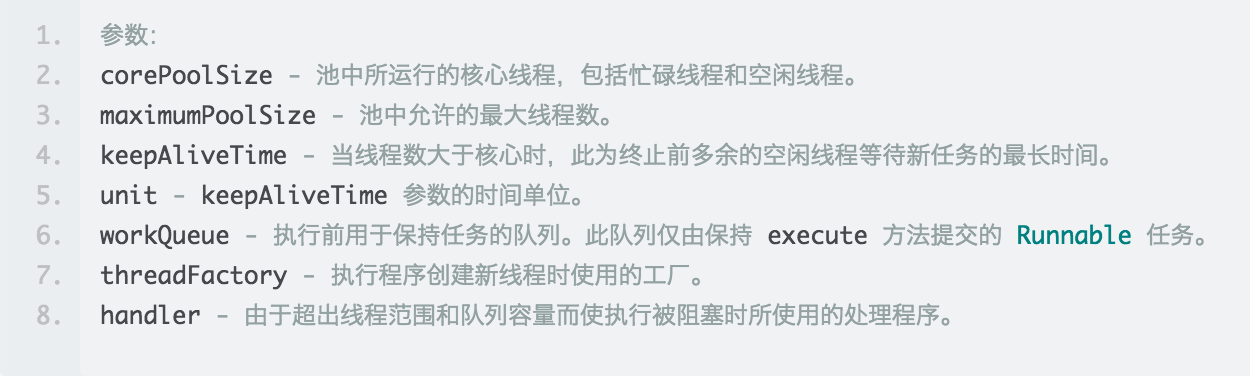
1、自我介绍

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、（没问题）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

**答：**



### 各参数之间的关系-corePoolSize，maximumPoolSize，workQueue

**当新任务在方法 execute(java.lang.Runnable) 中提交时，**

1. 当运行的线程数少于corePoolSize核心线程数，则创建线程来处理任务，即使核心线程中有空闲的线程，也照样创建新的线程来处理任务。
2. 当运行的线程数大于corePoolSize核心线程数而少于maximumPoolSize最大线程数，且任务队列未满时，将任务放入任务队列。
3. 当运行的线程大于corePoolSize核心线程数，且任务队列已满

1.若运行的线程数小于最大线程数，创建线程。

2.若运行的线程数等于最大线程数，则抛出异常。

**拒绝任务-(RejectedExecutionHandler)handler**

1.当ThreadPoolExecutor对象调用了shutdown()或者shutdownNow()，即线程池关闭的时，

2.当线程池的任务队列已满，并且线程池中的线程数目达到maxmumpoolSize时

以下两种情况，将会导致线程池拒绝执行任务，在ThreadPoolExecutor类中的execute方法调用RejectedExecutionHandler接口中的rejectedExecution方法。

**下面提供了四种预定义的处理策略**

1.默认的使用ThreadPoolExecutor.AbortPolicy，这种处理是一旦拒绝接接受，丢弃任务，将会抛出一个RejectedExecutionException的异常。

2.ThreadPoolExecutor.DiscardPolicy，也是丢弃任务，但是不抛出异常。

3.ThreadPoolExecutor.CallerRunsPolicy，由调用executor方法自己的线程运行任务。这提供了一个简单的反馈控制机制，可以减慢新任务提交的速度。

4.ThreadPoolExecutor.DiscardOldestPolicy：如果executor还没有shutdown，则处在任务队列头的任务将丢弃，则重新尝试执行任务（如果再次失败，继续重复此过程）

5、多线程实现同步的方式，互斥同步，非阻塞同步，

6、reenterlocker和sychronize的区别？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

**答：**

1. （没问题）说一下 synchronized 底层实现原理？

<http://bigdatadecode.club/JavaSynchronizedTheory.html>

synchronized 语句块是由一对 monitorenter/monitorexit 指令来显示获取monitor对象和释放monitor对象来实现的，

sychronnized方法是通过判断是否设置常量池中ACC\_SYNCHRONIZED标示符来隐式地获得monitor对象和是否monitor对象来实现的，

不管是显示还是隐式的获取monitor，本质没有区别。

说到底synchronized是通过一个获取和释放monitor对象来实现的。每个对象都有一个监视器锁(monitor)与之对应。当monitor被占用时就会处于锁定状态，例如monitorenter/monitorexit：

* 线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：
  1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
  2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
  3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。
* 执行monitorexit的线程必须是objectref所对应的monitor的所有者。
  1. 指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个monitor的所有权。

通过这两个指令我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

什么是Monitor对象？它是一种同步机制，monitor是由ObjectMonitor实现的，其主要数据结构有四个属性，如下：

* \_count用来记录该线程获取锁的次数
* \_WaitSet存放处于wait状态的线程队列
* \_EntryList存放处于等待获取锁block状态的线程队列，即被阻塞的线程
* \_owner指向持有ObjectMonitor对象的线程

当多个线程同时访问一段同步代码时，首先会进入\_EntryList队列中，当某个线程获取到对象的monitor后进入\_Owner区域并把monitor中的\_owner变量设置为当前线程，同时monitor中的计数器\_count加1，若线程调用wait()方法，将释放当前持有的monitor，\_owner变量恢复为null，\_count自减1，同时该线程进入\_WaitSet集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。如下图所示



在 Java 6 之前，synchronized是重量级锁，因为monitor对象的实现依赖于底层的操作系统的互斥锁来实现的，而操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了偏向锁、轻量级锁和自旋锁等概念，大大改进了性能。

2.（没问题）多线程中 synchronized 锁升级的原理是什么？

**为什么优化？**

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒会耗性能。

**为什么要引入偏向锁？**

因为经过HotSpot的作者大量的研究发现，大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

**为什么要引入轻量级锁？**

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。*因为阻塞线程需要CPU从用户态转到内核态，代价较大*，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

**偏向锁，轻量级锁，重量级锁，这些锁是如何存储的？**

这些锁的存储，就要说到Java对象头

下面重点说下java对象头。

众所周知Java中万物皆对象，那对象在内存中是怎么存储的呢？

每个对象分为三块区域:对象头、实例数据和对齐填充。

* 对象头包含两部分，第一部分是Mark Word，用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等，这一部分占一个字节。第二部分是Klass Pointer（类型指针），是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，这部分也占一个字节。(如果对象是数组类型的，则需要3个字节来存储对象头，因为还需要一个字节存储数组的长度)
* 实例数据存放的是类属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。
* 填充数据是因为虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐。

对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码(HashCode)，4Bits用于存储对象分代年龄，2Bits用于存储锁标志 位，1Bit固定为0，在其他状态(轻量级锁定、重量级锁定、GC标记、可偏向)下对象的存储内容如下表所示对象头存储结构。  


Synchronized通常被称为重量级锁，但是1.6之后对其进行优化，新增了轻量级锁和偏向锁，这里重点说下重量级锁，随后对Synchronized的优化简单介绍下。

从对象头的存储内容可以看出锁的状态都保存在对象头中，Synchronized也不例外，当其从轻量级锁膨胀为重量级锁时，锁标识位为10，其中指针指向的是monitor对象(也称为管程或监视器锁)的起始地址。

**synchronized 锁升级原理：**在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，或者第二个线程还在自旋中，这时又有一个线程3过来竞争这个锁对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

这几种锁的优缺点（偏向锁、轻量级锁、重量级锁）



4.（没问题）reenterlocker和sychronize的区别？

Synchronized和reenterlocker在基本用法上，很相似，都具有重入性，只是代码写法上有点区别，一个表现在API层面的互斥锁，一般在try{}finally{}中搭配使用的，另一个表现是原生语法层面的互斥锁，不过相比于synchronized，reenterlocker增加一些高级特性，主要是，等待可中断，可实现公平锁，以及锁可以绑定多个条件：

1. 等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。
2. 公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，可以防止饥饿，非公平锁则当锁释放的时候，其他线程随机抢占锁。Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。
3. 锁可以绑定多个条件：一个ReentrantLock对象可以同时绑定多个condition。而sychronized，锁对象的wait和notify或者notifyall方法可以实现一个隐含的条件，如果要和多于一个的条件关联的话，就不得不额外加一把锁，而reentrantlock只需要多次调用newcondition方法既可
   1. await(); 造成当前线程在被中断和释放前一直处于等待状态。
   2. signal();随机唤醒一个等待的线程。
   3. signAll();唤醒所有等待的线程

**性能方面：**

* ReentrantLock属于乐观锁思想,是非阻塞式同步的,是轻量级的,其内部的CAS就是基于乐观锁的思想实现的,
  + CAS指令需要有三个操作数，分别是内存位置（在java中可以简单理解为变量的内存地址用V表示），旧的预期值（用A表示）和新值（用B表示）。CAS指令执行时，当且仅当内存地址V等于预期值A时，处理器用新值B更新V的值，否知它就不执行更行，但是无论是否更行了V的值，都会返回V的值，并且这三个操作是原子性的，比锁的切换是要快很多的，并且如果则线程更新失败,失败的线程不会被挂起,通过自旋会重新准备再次尝试更新。
* syncthronized属于悲观锁方式,是阻塞式同步的,采用独占的方式,每次获取和释放锁频繁,上下文资源切换较大,存在短暂调度延迟。在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）,重量级锁后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞

**选择：**如果需要reenterlocker的高级特性，则优先选择reentrantlock。

syncthronized是基于这个关键字加锁的,ReentrantLock是基于其api方法加锁的,syncthronized是自动释放锁的,Reentrant需要手动释放,在业务编写中ReentrantLock一般在try{}finally{}中搭配使用的,避免锁的释放。

syncthronized属于悲观锁方式,是阻塞式的,采用独占的方式,每次获取和释放锁频繁,上下文资源切换较大,存在短暂调度延迟。

ReentrantLock属于乐观锁思想,是非阻塞式的,是轻量级的,其内部的CAS就是基于乐观锁的思想实现的,CAS是建立在硬件指令集中的,是通过指令切换来达到原子性,比锁的切换是要快很多的,CAS中有三个参数,内存变量地址A,预期值B,要更新的值C,当多个线程想要更新CAS的内存变量值时会进行判断,若当前CAS内存地址指向B,则将C更新为A,若当前A指向不是预期的B,则线程更新失败,失败的线程不会被挂起,会重新准备再次尝试更新。

|  |  |  |
| --- | --- | --- |
|  | Reenterlocker | Synchronized |
| 定义 | 是基于其api方法加锁的 | 基于这个关键字加锁的 |
| 可重入性 | 可重入的，可用在递归执行的方法上，重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样 | 可重入的，可用在递归执行的方法上，不用担心释放锁 |
| 便利性 | 自动释放锁 | Reentrant需要手动释放,在业务编写中ReentrantLock一般在try{}finally{}中搭配使用的,避免锁的释放。 |
| 性能区别 | ReentrantLock属于乐观锁思想,是非阻塞式的,是轻量级的,其内部的CAS就是基于乐观锁的思想实现的,CAS是建立在硬件指令集中的,是通过指令切换来达到原子性,比锁的切换是要快很多的,CAS中有三个参数,内存变量地址A,预期值B,要更新的值C,当多个线程想要更新CAS的内存变量值时会进行判断,若当前CAS内存地址指向B,则将C更新为A,若当前A指向不是预期的B,则线程更新失败,失败的线程不会被挂起,会重新准备再次尝试更新。 | syncthronized属于悲观锁方式,是阻塞式的,采用独占的方式,每次获取和释放锁频繁,上下文资源切换较大,存在短暂调度延迟。  在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）,重量级锁后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。 |
| 高级功能，reenterlocker的优势 | 主要有以下3项：          1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。          2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，可以防止饥饿，非公平锁则当锁释放的时候，其他线程随机抢占锁。Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。          3.锁可以绑定多个条件：一个ReentrantLock对象可以同时绑定多个condition。而sychronized，锁对象的wait和notify或者notifyall方法可以实现一个隐含的条件，如果要和多于一个的条件关联的话，就不得不额外加一把锁，而reentrantlock只需要多次调用newcondition方法既可  await(); 造成当前线程在被中断和释放前一直处于等待状态。  signal();随机唤醒一个等待的线程。  signAll();唤醒所有等待的线程。 |  |
|  |  |  |

*相似点：*

1.这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的（操作系统需要在用户态与内核态之间来回切换，代价很高，不过可以通过对锁优化进行改善）。

2.ReentrantLock和synchronized都是可重入的。synchronized因为可重入因此可以放在被递归执行的方法上,且不用担心线程最后能否正确释放锁；而ReentrantLock在重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样，否则可能导致其他线程无法获得该锁

*功能区别：*

这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成

*便利性：*很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

*性能的区别*：

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）,重量级锁后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

*1.Synchronized实现原理*

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器就减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

2.ReentrantLock

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：

        1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。

        2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。公平锁和非公平锁的区别？

公平锁是指当锁可用时,在锁上等待时间最长的线程将获得锁的使用权。而非公平锁则随机分配这种使用权。和synchronized一样，默认的ReentrantLock实现是非公平锁,因为相比公平锁，非公平锁性能更好。当然公平锁能防止饥饿,某些情况下也很有用。在创建ReentrantLock的时候通过传进参数true创建公平锁,如果传入的是false或没传参数则创建的是非公平锁

        3.锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象。ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

*ReenTrantLock实现的原理：（？？？）*

之后还会总结一篇ReenTrantLock相关的原理底层原理分析，简单来说，ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

每一个ReentrantLock实例都有且只有一个AQS实例，一个AQS实例维护一个Sync Queue和多个Condition Queue（看你新建了多少condition）。所以说，当我们的业务代码中的多个线程对同一个ReentrantLock实例进行锁竞争操作时，其实际就是对同一个Sync Queue的队列进行入队、出队操作。Condition 仿照实现Object类的wait signal signallAll等函数功能的，

每个AQS里有x（视你newCondition几次）个Condition Queue，它的结点类也是AQS内部类Node。Node里有一个nextWaiter，指向下一个在同一Condition Queue里的Node。

 首先明确下是，condition.wait一定是在成功lock的线程里调用才有效，不然不符合逻辑，同时也会抛出IlleagleMornitorException。

 获取锁的线程处于Sync Queue的队首，当调用condition.wait时，该线程会释放锁（即将AQS的state置为0），同时唤醒后继结点，后继结点在acquire的循环里会成功获取锁，然后将自己所在结点置为队首，然后开始自己线程自己的业务代码。

* 当waiter\_1收到相应condition的signal后，在Condition Queue中的Node会从Condition Queue中出队，进入Sync Queue队列，开始它的锁竞争的过程。
* 这里可以看出来，即使是被signal了，被signal的线程也不是直接就开始跑，而是再次进入Sync Queue开始竞争锁而已。这里的这个逻辑，跟Object.wait Object.signal也是完全一样的。

*什么情况下使用ReenTrantLock：*

答案是，如果你需要实现ReenTrantLock的三个独有功能时。

5.为什么[ConcurrentHashMap 1.8为什么要使用CAS+Synchronized取代Segment+ReentrantLock](https://www.cnblogs.com/yangfeiORfeiyang/p/9694383.html)

<https://juejin.im/post/5bc87409f265da0ad701da35>

7、（没问题）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

内存屏障，又称内存栅栏，是一个CPU指令，它的作用有两个，*一是保证特定操作的执行顺序，二是保证某些变量的内存可见性*（利用该特性实现volatile的内存可见性）。由于编译器和处理器都能执行指令重排优化。如果在指令间插入一条Memory Barrier则会告诉编译器和CPU，不管什么指令都不能和这条Memory Barrier指令重排序，也就是说通过插入内存屏障禁止在内存屏障前后的指令执行重排序优化。Memory Barrier的另外一个作用是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读取到这些数据的最新版本。总之，volatile变量正是通过内存屏障实现其在内存中的语义，即可见性和禁止重排优化。

**内存屏障是什么**

* 硬件层的内存屏障分为两种：Load Barrier 和 Store Barrier即读屏障和写屏障。
* 内存屏障有两个作用：

1. 阻止屏障两侧的指令重排序；
2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。

* 对于Load Barrier（读屏障）来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据；
* 对于Store Barrier（写屏障）来说，在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

 java的内存屏障通常所谓的四种即LoadLoad,StoreStore,LoadStore,StoreLoad实际上也是上述两种的组合，完成一系列的屏障和数据同步功能。

 LoadLoad屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。 **在后续读操作执行之前，****之前的读操作要读取的数据已被读取。**

 StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。**在后续写操作执行之前，****之前****的写操作已对其他处理器可见。**

 LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。 **在后续写操作执行之前，之前的读操作要读的数据已被读取**

 StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。**在后续读操作执行之前，之前的写操作已对其他处理器可见。*它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能***

***Volitle内存屏障规则：由于内存屏障的作用，避免了volatile变量和其它指令重排序、线程之间实现了通信，使得volatile表现出了锁的特性。***

内存屏障规则： SS,写操作,SL, 读操作,LL, 读操作，LS

（1）在每一个volatile写操作前面插入一个StoreStore屏障。这确保了在进行volatile写之前前面的所有普通的写操作都已经刷新到了内存。

（2）在每一个volatile写操作后面插入一个StoreLoad屏障。这样可以避免volatile写操作与后面可能存在的volatile读写操作发生重排序。

（3）在每一个volatile读操作后面插入一个LoadLoad屏障。这样可以避免volatile读操作和后面普通的读操作进行重排序。

（4）在每一个volatile读操作后面插入一个LoadStore屏障。这样可以避免volatile读操作和后面普通的写操作进行重排序。

*volatile的内存屏障插入策略非常保守，其实在实际中，只要不改变volatile写-读得内存语义，编译器可以根据具体情况优化，省略不必要的屏障。*

比如在volatile读操作后面，还是volatile读的话，那么前一个volatile读操作后面的LL，由于禁止下面所有的普通写与上面的volatile读重排序，但是由于存在第二个volatile读，那个普通的读根本无法越过第二个volatile读。所以可以省略。

再比如在volatile写操作后面，还是volatile写的话，那么前一个volatile写操作后面的SL，可以省略，因为后面跟的是volatile写，没有读操作。

再比如：在volatile读操作后面，是普通写的话，volatile读操作后面的LL就可以省了。因为下面不存在普通读了。

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

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，如何保证并发安全。（重要）（需要搞清楚）

15.reenterlocker和sychronize的区别？

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。

考拉面经2：

自我介绍

面试岗位：java研发

时间 ：2017-08-23

部门 ： 考拉海购

面试次数：2轮技术面+1面HR面

一面技术

1、介绍项目

2、spring java反射

3、数据结构题

4、班级表和学生表，利用sql 语句查询每个班级的学生人数

create table student(stu\_id int,name varchar(10),class\_id int);

insert into student values(1,'w',1);

insert into student values(2,'y',1);

insert into student values(3,'z',2);

insert into student values(4,'zx',3);

create table class(class\_id int, class\_name varchar(10));

insert into class value(1,'一班');

insert into class value(2,'二班');

insert into class value(3,'三班');

insert into class value(4,'四班');

select ifnull(s.class\_id,c.class\_id) class\_id,ifnull(count(s.class\_id),0) num from class c left join student s on c.class\_id=s.class\_id group by s.class\_id,c.class\_id order by num desc;

Select classId,count() from students group by classId

5、int和Integer区别

二面

1、自我介绍

2、情感摘要介绍

3、spring java反射

4、i++原子操作

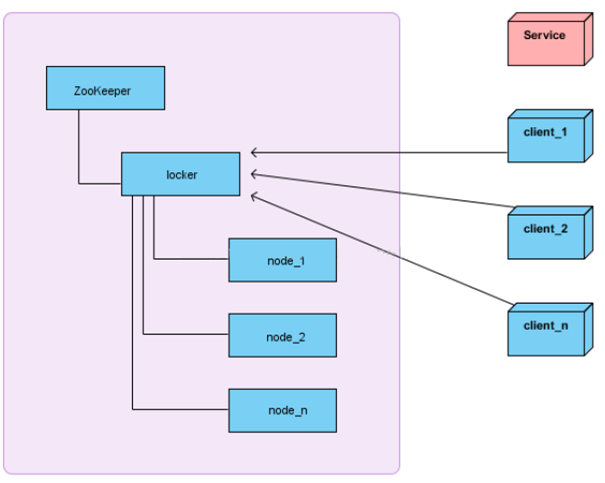
5、hashmap和hashtable区别，以及concurrenthashmap

6、zookeeper 实现分布式锁

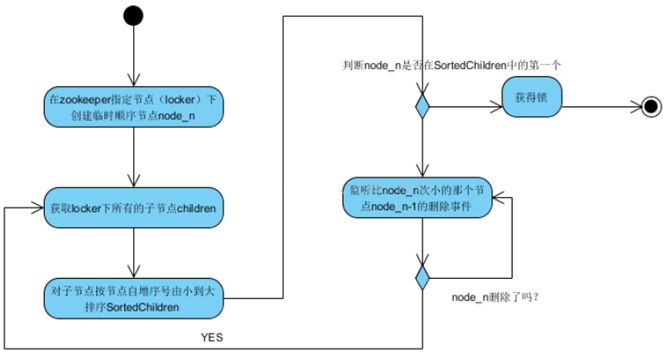
#### ****参考答案****：

有了zookeeper的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。 对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的distribute\_lock 节点就释放出锁。 对于第二类， /distribute\_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，编号最小的获得锁，用完删除，依次方便。

**获取分布式锁的流程**

[](https://github.com/0voice/interview_internal_reference/blob/master/12.Zookeeper%E7%AF%87/zk_distributed_locker1.png)

在获取分布式锁的时候在locker节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用createNode方法在locker下创建临时顺序节点， 然后调用getChildren(“locker”)来获取locker下面的所有子节点，注意此时不用设置任何Watcher。客户端获取到所有的子节点path之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非locker所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，然后对其调用exist()方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的Watcher会收到相应通知，此时再次判断自己创建的节点是否是locker子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。

[](https://github.com/0voice/interview_internal_reference/blob/master/12.Zookeeper%E7%AF%87/zk_distributed_locker2.png)

代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于BaseDistributedLock，实现了基于Zookeeper实现分布式锁的细节。

7、团购系统的表设计

8、（没问题）悲观锁 和乐观锁 以及sql如何实现

悲观锁、乐观锁的区别及使用场景

定义：

悲观锁（Pessimistic Lock）：

每次获取数据的时候，都会担心数据被修改，所以每次获取数据的时候都会进行加锁，确保在自己使用的过程中数据不会被别人修改，使用完成后进行数据解锁。由于数据进行加锁，期间对该数据进行读写的其他线程都会进行等待。

代表：sychronize，lock

实现：

乐观锁（Optimistic Lock）：

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

实现：CAS实现和version方式

version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

核心SQL代码：

|  |  |
| --- | --- |
| 1 | update table set x=x+1, version=version+1 where id=#{id} and version=#{version}; |

CAS操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

代表：

适用场景：

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

什么是AbstractQueuedSynchronizer？

AbstractQueuedSynchronizer是JUC并发包中锁的底层支持，AbstractQueuedSynchronizer是抽象同步队列，简称AQS，是实现同步器的基础组件，并发包中锁的实现底层就是使用AQS实现，当然大多数人不会直接用到AQS，但是学习这个类对并发包的底层理解还是有莫大的帮助的。

AQS中维持了一个单一的状态信息state，可以通过getState,setState,compareAndSetState 函数修改其值，AQS内部维持一个FIFO队列来完成资源获取线程的排队工作。对于ReentrantLock 的实现来说，state 可以用来表示当前线程获取锁的可重入次数；

对应读写锁ReentrantReadWriteLock 来说state 的高16位表示读状态，也就是获取该读锁的次数，低 16位 表示获取到写锁的线程的可重入次数；对于FuterTask 来说，state用来表示任务状态(例如，还没开始，运行，完成，取消）；

对应CountDownlatch 和CyclicBarrie 来说，state用来表示计数器当前的值。

AQS有个内部类ConditionObject 是用来结合锁实现线程同步，ConditionObject可以直接访问AQS对象内部的变量，比如 state 状态值 和AQS队列；

ConditionObject是条件变量，每个条件变量对应一个条件队列（单向链表队列)，用来存放调用条件变量的await（）方法后被阻塞的线程。

对于AQS 来说，线程同步的关键是对状态值state进行操作，根据state是否属于一个线程，操作state的方式分为独占模式和共享模式。

独占模式下获取和释放资源使用方法的源码如下：

void acquire(int arg)

void acquireInterruptibly(int arg)

boolean release(int arg)

共享模式下获取和释放资源方法的源码如下：

void acquireShared(int arg)

void acquireSharedInterruptibly(int arg)

boolean releaseShared(int arg)

另外还有个查询同步队列等待线程情况的方法如下：

Collection<Thread> getQueuedThreads()

对于独占锁方式获取的资源是与具体线程绑定的，也就是说如果一个线程获取到了资源，就会标记是那个线程获取到的，其他线程尝试操作state获取资源时候发现当前该资源不是自己持有，就会获取失败后被阻塞；

比如独占锁ReentrantLock的实现，当一个线程获取了ReentrantLock的锁后，AQS内部会首先使用CAS操作把state状态从0 变成 1，然后设置当前锁的持有者为当前线程，当该线程再次获取锁的时候，发现当前线程就是锁的持有者，则会把state状态值从1变成2，

也就是设置可重入次数，当另外一个线程获取锁的时候发现自己并不是该锁的持有者就会被放入AQS阻塞队列后挂起。

对于共享操作方式资源是与具体线程不相关的，多个线程去请求资源时候是通过CAS方式竞争获取资源，当一个线程获取到了资源后，另外一个线程再次获取时候，如果 当前资源还能满足它的需要，则当前线程只需要使用CAS方式进行获取即可，

共享模式下并不需要记录哪个线程获取了资源；比如 Semaphore 信号量，当一个线程通过acquire()方法获取一个信号量时候，会首先看当前信号两个数是否满足需要，不满足则把当前线程放入阻塞队列，如果满足则通过自旋CAS获取信号量。

JVM优化经验（重要重要重要）（需要搞清楚）

HR面

1、自我介绍

2、研究生的学习情况

3、意向工作地点

4、碰到的问题，解决方法，以及实现了什么效果

5、担任职务

6、实习情况

360面试总结

为什么netty的线程池中，线程的个数是cpu的2倍？

IO优化中，这样的估算公式可能更适合：

线程执行计算，占用CPU的时间，

线程等待时间，不占用CPU的时间，执行IO

最佳线程数目 = （（线程等待时间/线程执行时间）+ 1）\* CPU数目

有个属性叫ioratio,作用是在一个eventloop中花在IO事件上的时间比例，默认值是50，设置50的目的是尝试保证在在eventloop中花在IO事件上的时间和非IO事件(CPU计算时间)的时间比例是1:1.

因为很显然，线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

*Sets the percentage of the desired amount of time spent for I/O in the event loop. The default value is  
\* {****@code*** *50}, which means the event loop will try to spend the same amount of time for I/O as for non-I/O tasks.*

在netty中的eventloopthread的run方法是eventloopthread线程的主体，做的事情分为对三个步骤不断循环：

1. 首先轮询注册到reactor线程对用的selector上的所有的channel的IO事件

2. 处理产生网络IO事件的channel

3. 处理任务队列

在第二步和第三步开始前加了个判断,

* 如果ioratio等于100，直接处理所有IO时间和处理所有的任务队列，这会导致如果eventloop线程在处理任务队列的停留的时间过长，那么将积攒许多的IO事件无法处理，最终导致大量客户端请求阻塞。这条路一般不会走。
* 如果ioratio不等于100，默认是50，则会保证处理io事件的时间和处理任务队列的时间是相等的，第一，不会阻塞IO事件，第二，在IO优化中，有这么个公式，最佳线程数目 = （（线程等待时间/线程执行时间）+ 1）\* CPU数目，这样应用的性能会最优。所以netty进行了这个设计。­
  + 线程执行计算，占用CPU的时间，
  + 线程等待时间，不占用CPU的时间，执行IO

工作线程数”的设置依据是什么，到底设置为多少能够最大化CPU性能。

问：工作线程数是不是设置的越大越好？

答：肯定不是的

服务器CPU核数有限，能够同时并发的线程数有限，单核CPU设置10000个工作线程没有意义

线程切换是有开销的，如果线程切换过于频繁，反而会使性能降低

问：调用sleep()函数的时候，线程是否一直占用CPU？

答：不占用，等待时会把CPU让出来，给其他需要CPU资源的线程使用。

不止sleep()函数，在进行一些阻塞调用时，例如网络编程中的：

阻塞accept()，等待客户端连接

阻塞recv()，等待下游回包

都不占用CPU资源。

问：单核CPU，设置多线程有意义么，是否能提高并发性能？

答：即使是单核，使用多线程也是有意义的，大多数情况也能提高并发

hdfs中的读写操作？

为什么三次握手 四次挥手？

**TCP连接建立过程：**

1、客户端向服务器发送SYN，其中seq=x。

2、服务器收到SYN报文段后，发送SYN+ACK，其中seq=y，确认号=x+1。

3、客户端收到SYN+ACK报文段后，发送ACK，确认号=y+1。服务器收到ACK报文段后，连接建立。

**为什么连接需要三次握手？**

三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误”

假设连接时时2次握手，正常情况下，客户端发送请求，服务器返回确认，这样是没问题的，但是出现另外一种异常情况，比如客户端发送请求，因为网络延时，客户端发出的请求延误到连接释放以后的某个时间才到达服务器，本来是一个早已失效的报文段，但是服务器接受到之后，误认为客户端有发出了一次新的连接，服务器端返回确认，新连接就创建了，由于客户端没有发出请求，并不会理睬服务器的确认，所以不会向服务器发送数据，但是服务器确认为连接已经建立，并一直等待客户端客户端发来数据，服务器的许多资源就这样白白浪费。

由于三次握手之后，双方发送和接受正常。所以四次没必要

**TCP连接断开过程：**

1、客户端TCP模块在收到应用程序的通知后，发送FIN，seq=x。

2、服务器收到FIN报文段，发送ACK，确认号=x+1，并且通知应用程序客户端关闭了连接。客户端收到ACK报文段。

3、服务器端的应用程序通知TCP关闭连接，服务器端TCP发送FIN+ACK，seq=y，确认号=x+1（这里ACK只是一般性的捎带ACK，TCP总是这样，以增强健壮性，反正也不费力气，从原理上说，对连接断开不是必须的）。

4、客户端收到FIN+ACK报文段后，发送ACK，确认号y+1。服务器收到ACK报文段后，连接断开。

**为什么断开需要4次握手？**

Ps: 全双工传输：是指交换机在发送数据的同时也能够接收数据，两者同步进行，所以连接关闭的时候，通信双方都需要独立关闭自己的通信通道，也就是半关闭。

TCP的连接释放的四次握手，可以看成2个2次握手

client先发送FIN告知对方我已经完成数据发送了，server回复ack来确定我知道了。这样一个流程，就关闭了client到服务端的连接。但是还可以接收来自server方的数据。

server此时已经知道接收不到client的数据了，但是还可以给它发送数据。如果server也没有啥数据要发送给对方了，server也会以FIN标志位发送一个信息给client，client接到后，也会传递一个ack表示知道了。这样子，双方都完成了关闭。

**为什么客户端A在time-wait状态必须等待2MSL的时间？**

**1.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，因而使得处于LAST-ACK状态的B收不到已发送的FIN+ACK的确认。如果超时，B会重传FIN+ACK，A就能在2MSL之内收到B的重传，接着A重传最后一个ACK，重新启动2MSL计时器。最后A和B都正常进入到CLOSED。如果A发送完最后一个ACK之后，不等待2MSL时间立即释放的话，那么就无法收到B的重传的FIN+ACK，因为也不会再发送一次ACK了，这样B就无法按照正常步骤进入CLOSED状态.**

**2.防止连接三次握手时提到的 “已失效的连接请求报文段出现在“ 出现在本连接中。A在发送最后一个ACK报文之后，在经过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消失，这样就可以使下一个新连接中，不会出现这种旧的连接请求。**

接口和抽象类的区别？接口中是否可以放属性？

hashmap中key有什么要求？

<https://blog.csdn.net/weixin_42054155/article/details/80715750>

1、HashMap 的 key 和 value 都可以是 null,

2、Map 的 key 和 value 都 不允许 是 基本数据类型

3、HashMap 的 key 可以是 任意对象，意思都要继承Object类的hashcode()方法，但如果 对象的 hashCode 改变了，那么将找不到原来该 key 所对应的 value, 所以最好Key用final修饰。

hashmap的get的流程？

* Get方法的工作原理：通过对key的hashcode做hash操作得出一个hash值，即是对key的hashcode做高16位于低16位做一个异或，然后计算数组下标（n-1）&hash值,如果产生碰撞，则利用key.equals()方法去链表中查找对应的节点。

equal 和等号的区别

java8 9的新功能

一个线程接着一个线程执行 有什么方法？

<https://blog.csdn.net/qq_35571554/article/details/82834486>

public static void main(String[] args) throws InterruptedException {  
  
 final Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 add();  
 }  
 });  
 final Thread t2 = new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 //引用t1线程，等待t1线程执行完  
 t1.start();  
 t1.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 add();  
 }  
 });  
 Thread t3 = new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 //引用t2线程，等待t2线程执行完,一定要在此处t2.start()  
 //在主线程中调用t2.start()执行顺序不确定，因为若t2.join()先于t2.start()运行  
 //导致结果不确定  
 t2.start();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 add();  
 }  
 });  
 t3.start();  
// t2.start();  
// t1.start();  
 }

使用thread.join方法， 首先join() 是一个synchronized方法， 里面调用了wait()，

当t1调用join方法时，相当于在外部线程中写了一个同步同步代码块，

t1.start()

sychronized(t1) {

while(true)

t1.wait() // 当前线程等待

}

}

add()

当t1线程执行完之后，也就是执行了t1.notofyall(), 也就相当于唤醒当前线程，然后当前线程继续执行add()。

创建线程的方式

如何停止一个线程？

缓存一致性

由于cpu的处理速度和内存的读写速度有几个数量级的差距，所以计算机不得不加入高速缓存作为cpu和内存之间的缓冲：将运算要使用的数据复制到缓存中，让运算能快速进行，当运算结束之后，在将缓存同步到内存中，这样cpu就无需等待缓慢的内存读写了。

但是这样会引入一个问题：缓存一致性问题是由于在多核系统中，由于cpu缓存数据更新时没有及时刷新到主内存中，这时就会导致不同处理器缓存的数据不一致的问题，为了解决缓存不一致的问题，就提出了对总线加锁或者缓存一致性协议，例如MESI

解决方案有：

1.通过在总线加LOCK#锁的方式

原理是通过锁住总线，保证每次只有一个cpu核能访问内存，其他cpu全部停止执行，只有该cpu完全执行完之后，其他cpu才能从内存中读取共享变量，但是在锁住总线期间，其他cpu无法访问内存，会导致效率低下。

2.通过缓存一致性协议

例如：MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本（浅拷贝），会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从主内存重新读取。

思考：既然cpu自己能够做到通过缓存一致性机制，保证各个处理器中的各自缓存和内存一致，那就不存在并发问题了，那Java就没有必要做内存模型去控制并发问题了？

思考答案：其实缓存一致性并不能保证立即的内存可见性，CPU为了提高效率，不会时时刻刻去刷新缓存，而是在合适的时机才会刷新缓存，只有在JVM声明了锁操作的时候（也就是Lock#信号或者内存屏障），才会立即刷新缓存。

喜马拉雅面试总结

为什么想换工作？

为了成长

内核态和用户态的区别?为什么要这么分？

InputStream和inputRead的区别？应用场景？

字符和字节的区别？

http和tcp的区别？

Mysql的索引为什么B+树？什么是B树，B-树，B+树？

Fileio.close主要关的是什么？