1.HBase的WAL(Write-Ahead Logging)为什么能提升写性能?

答：之所以能够提升写性能，是因为**WAL将一次随机写转化为了一次顺序写加一次内存写**。

2.为什么HBase表设计的时候尽量减少ColumnFamily的个数?

答：**MemStore的最小flush单元是HRegion而不是单个MemStore**。可想而知，如果一个HRegion中Memstore过多，每次flush的开销必然会很大，因此我们也建议在进行表设计的时候尽量减少ColumnFamily的个数。

3.知道HBase的LSM结构吗?

答：**LSM-Tree主题思想为划分成不同等级的树。可以想象一份索引由两棵树组成：一个存在于内存（可以使其他树结构），一个存在于磁盘**。将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘，不过读取的时候稍微麻烦，需要合并磁盘中历史数据和内存中最近修改操作，所以写入性能大大提升，读取时可能需要先看是否命中内存，否则需要访问较多的磁盘文件。极端的说，**基于LSM树实现的HBase的写性能比Mysql高了一个数量级，读性能低了一个数量级**。LSM树原理把一棵大树拆分成N棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会flush到磁盘中，磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。

参考：

Hbase中LSM索引思想.note

4.mapreduce的map进程和reducer进程的jvm垃圾回收器怎么选择可以提高吞吐量?

答：Parallel Scavenge + Parallel Old

5.写一个算法判断一个图是不是DAG?

答：拓扑排序

6.数据倾斜怎么处理?

答：**如果是reduceByKey等聚合类算子引起的倾斜**，给所有的key增加随机字符串前缀再进行聚合。**如果是join算子引起的倾斜**，这就要分情况了，①**有一个表很小**：使用mapside join，将小表发到每个map端，这样就不用shuffle了，彻底解决了数据倾斜。②**两个表都很大，但是导致倾斜的key的数量很少**：采样倾斜key并分拆join操作，对表进行采样将导致数据进行的key的数据过滤出来，左表的key打上随机0-N前缀，右表同时进行过滤并扩充N倍，然后两表join并union上原表剩余数据的join结果。③**两个表都很大，导致倾斜的key数量很多**：对左表所有key打上随机前缀0-N，右表扩容N倍，进行join。

7.数据库中的事务，是什么意思，一般在哪用?

答：**将多条语句一起提交，要么成功要么失败**；一般我们再执行多条语句的一候使用，如delete一条语句条语句，再insert一条语句，要保证两条语句都执行成功，结果才正确，这样就可以用事务，要么都执行成功，只要有一条没成功就回滚。

8.RDD的弹性(容错)表现?

答：

1、弹性之一：自动的进行内存和磁盘数据存储的切换；

2、弹性之二：基于Lineage的高效容错（第n个节点出错，会从第n-1个节点恢复，血统容错）；

3、弹性之三：Task如果失败会自动进行特定次数的重试（默认4次）；

4、弹性之四：Stage如果失败会自动进行特定次数的重试（可以只运行计算失败的阶段）；只计算失败的数据分片；

9.RDD的sortBy是内存+磁盘排序，scala的sortBy是内存排序

10.spark streaming连接kafka的两种方式有什么区别(直连和receiver)?

答：①createStream会使用 Receiver；而createDirectStream不会；②createStream使用的 Receiver 会分发到某个 executor 上去启动并接受数据（存在内存中），而createDirectStream直接在 driver 上接收数据；③createStream使用 Receiver 源源不断的接收数据并把数据交给 ReceiverSupervisor 处理最终存储为 blocks 作为 RDD 的输入，从 kafka 拉取数据与计算消费数据相互独立；而createDirectStream会在每个 batch 拉取数据并就地消费，到下个 batch 再次拉取消费，周而复始，从 kafka 拉取数据与计算消费数据是连续的，没有独立开（每次触发action在进行拉取）；④createStream中创建的KafkaInputDStream 每个 batch 所对应的 RDD 的 partition 不与 Kafka partition 一一对应（需要使用union提高并发度）；而createDirectStream中创建的DirectKafkaInputDStream 每个 batch 所对应的 RDD 的 partition 与 Kafka partition 一一对应。

11.Spark的 shuffle和MapReduce的 shuffle的区别吗?

答：①**从执行角度讲，两者有差别。**MapReduce的shuffle会经历map()，spill，merge，shuffle，sort，reduce()，是按照流程顺次执行的，**属于push类型**。但是，Spark不一样，因为Spark的Shuffle过程是算子驱动的，具有懒执行的特点，**属于pull类型**。正因为是算子驱动的，**Spark的Shuffle主要是两个阶段：Shuffle Write和Shuffle Read**。

②**从数据流角度讲，两者有差别。**MapReduce 只能从一个 Map Stage shuffle 数据，Spark 可以从多个 Map Stages shuffle 数据（这是 DAG 型数据流的优势，可以表达复杂的数据流操作，参见 CoGroup(), join() 等操作的数据流图 SparkInternals/4-shuffleDetails.md at master · JerryLead/SparkInternals · GitHub）。
③**从map角度讲，两者有差别。**MapReduce的map阶段的数据存在一个环形缓冲区中，缓冲区大小默认是100MB，达到80%后会溢写磁盘同时排序。Spark的shuffle write阶段是将数据写入了一个AppendOnlyMap或者pairBuffer结构中，可以选择不排序bypass模式。
④**从reduce角度讲，两者有差别。**MapReduce shuffle阶段就是边fetch边使用combine()进行处理，但是combine()处理的是部分数据。MapReduce不能做到边fetch边reduce处理，因为MapReduce为了让进入reduce()的records有序，必须等到全部数据都shuffle-sort后再开始reduce()。然而，Spark不要求shuffle后的数据全局有序，因此没必要等到全部数据shuffle完成后再处理。为了实现边shuffle边处理，而且流入的records是无序的可以用aggregate的数据结构，比如HashMap。

12.解释一下Spark的RDD

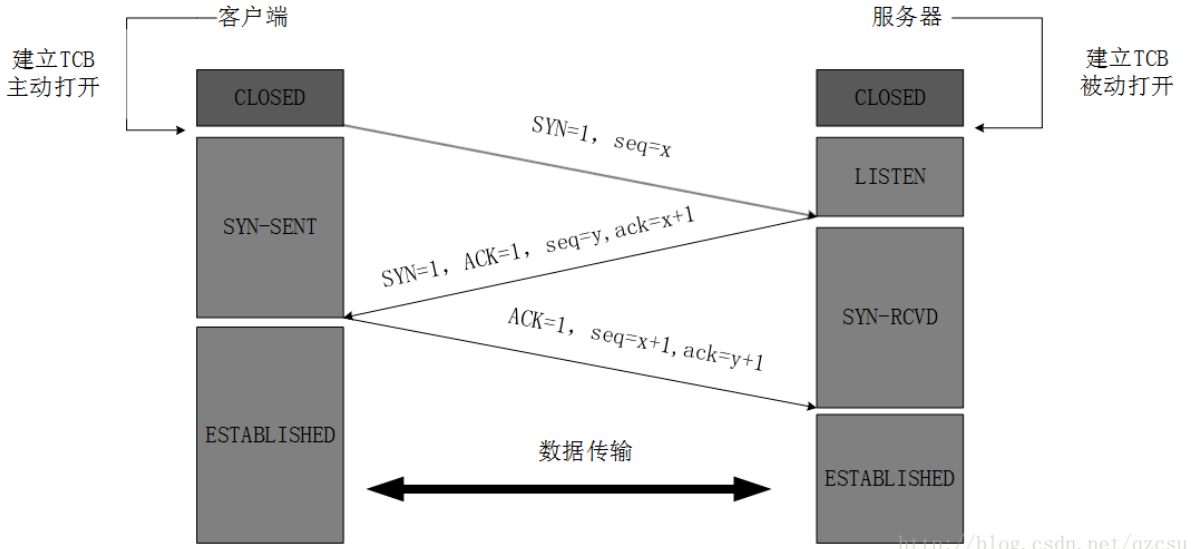
答：①一系列分区；②会有一个函数作用在每个切片上；③RDD和RDD之间存在依赖关系；④（可选）如果是RDD中装的是KV类型的，那么Shuffle时会有一个分区器。默认是HashPartitioner；⑤（可选）如果只从HDFS中读取数据，会感知数据的位置，将Executor启动在数据所在的机器上。

13.static关键字的特性

答：被类的所有对象所共享；随着类的加载而加载；优先于对象而存在；被static关键字所修饰的成员变量可以直接被类名所调用（也可以被对象调用）。

14.TCP三次握手和四次挥手

答：

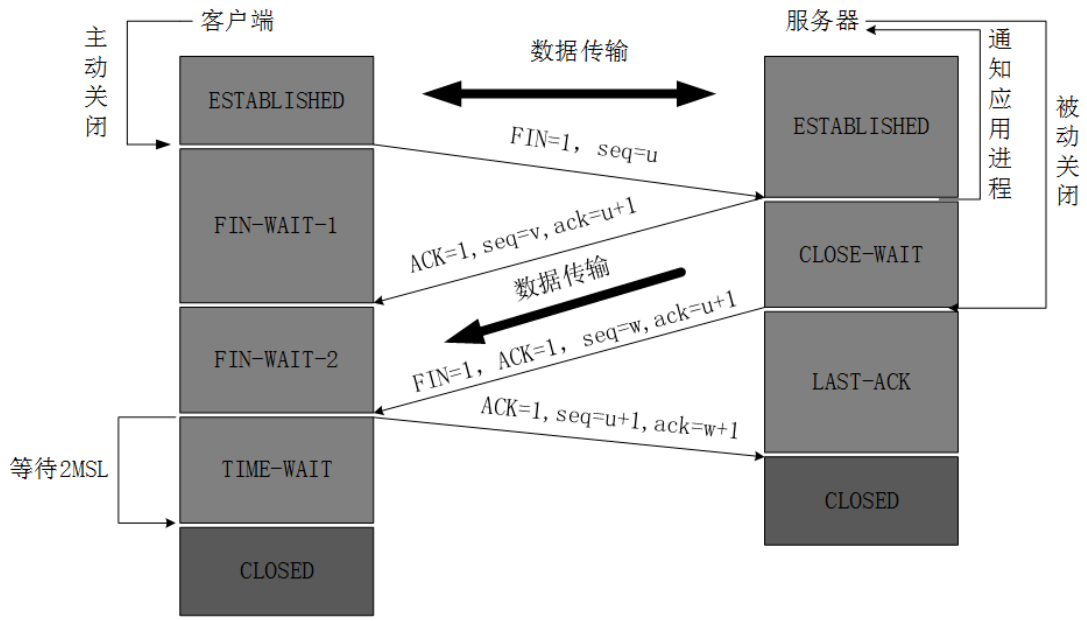


为什么是三次而不是两次?

一句话，主要防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

如果使用的是两次握手建立连接，假设有这样一种场景，**客户端发送了第一个请求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了**，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。



为什么客户端最后还要等待2MSL?

MSL（Maximum Segment Lifetime），最大报文段生成时间。

第一，保证TCP协议的全双工连接能够可靠关闭

第二，保证这次连接的重复数据段从网络中消失

为什么建立连接是三次握手，关闭连接确是四次挥手呢?

建立连接的时候， 服务器在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。

而关闭连接时，服务器收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送，从而导致多了一次。

参考：

TCP协议详解7层和4层解析(美团面试，阿里面试) 尤其是三次握手，四次挥手 具体发送的报文和状态都要掌握.note

TCP的三次握手与四次挥手（详解+动图）.note

15.TCP与UDP的区别

答：**UDP，在传送数据前不需要先建立连接**，远地的主机在收到UDP报文后也不需要给出任何确认。虽然UDP不提供可靠交付，但是正是因为这样，省去和很多的开销，使得它的速度比较快，比如一些对实时性要求较高的服务，就常常使用的是UDP。**对应的应用层的**

协议主要有DNS,TFTP,DHCP,SNMP,NFS 等。**TCP，提供面向连接的服务**，在传送数据之前必须先建立连接，数据传送完成后要释放连接。因此TCP是一种可靠的的运输服务，但是正因为这样，不可避免的增加了许多的开销，比如确认，流量控制等。**对应的应用层的协议主要有** SMTP,TELNET,HTTP,FTP 等。

16.TCP滑动窗口以及拥塞控制方法

答：TCP这个协议是网络中使用的比较广泛，他是一个面向连接的可靠的传输协议。既然是一个可靠的传输协议就需要对数据进行确认。TCP协议里窗口机制有2种：**一种是固定的窗口大小**；**一种是滑动的窗口**。这个窗口大小就是我们一次传输几个数据。对所有数据帧按顺序赋予编号，**发送方在发送过程中始终保持着一个发送窗口，只有落在发送窗口内的帧才允许被发送**；**同时接收方也维持着一个接收窗口，只有落在接收窗口内的帧才允许接收**。这样通过调整发送方窗口和接收方窗口的大小可以实现流量控制。

TCP滑动窗口技术通过动态改变窗口大小来调节两台主机间数据传输。每个TCP/IP主机支持全双工数据传输，因此**TCP有两个滑动窗口：一个用于接收数据，另一个用于发送数据。TCP使用肯定确认技术，其确认号指的是下一个所期待的字节**。

拥塞控制方法：慢开始、拥塞避免、快重传、快恢复

参考：

TCP的滑动窗口机制.note

17.TCP报文重传

答：TCP每发送一个报文段，就对这个报文段设置一次**计时器**。当计时器超时而没有收到确认时，就重传该报文。

18.NIO、BIO、AIO、IO复用

答：**NIO(Non-blocked IO)：进程给CPU传达任务后，继续处理后续的操作，隔断时间再来询问之前的操作是否完成。这样的过程其实也叫轮询。**

BIO(Non-blocked IO)：进程给CPU传达一个任务之后，一直等待CPU处理完成，然后才执行后面的操作。

AIO(Asynchronous IO)：执行一个操作后，可以去执行其他的操作，然后等待通知再回来执行刚才没执行完的操作。

同步IO：NIO、BIO

异步IO：AIO

同步与异步的根本区别：(a)是数据通过网关到达内核，内核准备好数据，(b)数据从内核缓存写入用户缓存。

同步：不管是BIO,NIO,还是IO多路复用，第二步数据从内核缓存写入用户缓存一定是由 用户线程自行读取数据，处理数据。

异步：第二步数据是内核写入的，并放在了用户线程指定的缓存区，写入完毕后通知用户线程。

IO复用：此模型用到select和poll函数，这两个函数也会使进程阻塞，select先阻塞，有活动套接字才返回，但是和阻塞I/O不同的是，**这两个函数可以同时阻塞多个I/O操作**，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写（**就是监听多个socket**）。select被调用后，进程会被阻塞，内核监视所有select负责的socket，当有任何一个socket的数据准备好了，select就会返回套接字可读，我们就可以调用recvfrom处理数据。**正因为阻塞I/O只能阻塞一个I/O操作，而I/O复用模型能够阻塞多个I/O操作，所以才叫做多路复用。**

参考：

Java NIO：IO与NIO的区别 -阿里面试题.note

IO复用,AIO,BIO,NIO,同步，异步，阻塞和非阻塞 区别(百度).note

19.Java如何DUMP内存出来看看内存溢出呢?

答：DUMP分为两种：

(a)内存dump是指通过jmap -dump <pid>输出的文件，

(b)而线程dump是指通过jstack <pid>输出的信息。

参考：

Java JVM- jstat查看jvm的GC情况.note

20.现在我的端口8080被占用了，如何找出这个进程来用哪个命令（windows）

答：netstat -aon | findstr "8080" -> 找到占用端口的pid -> tasklist | findstr "2016" -> 找到进程名字 -> taskkill /f /t /im xxoo.exe

参考：

如何在Windows下查看JAVA端口占用情况(阿里面试).note

21.Java如何查看GC情况

答：**jstat -gc 7964 5000** 每隔5秒钟查看进程7964的内存使用情况；

22.Java NIO和IO的主要区别

答：

IO	NIO
面向流	面向缓冲

阻塞IO	非阻塞IO
无	选择器

23.Java为什么覆盖equals时总要覆盖hashCode?

答： 一个很常见的错误根源在于没有覆盖hashCode方法。在每个覆盖了equals方法的类中，也必须覆盖hashCode方法。如果不这样做的话，就会违反Object.hashCode的通用约定，从而**导致该类无法结合所有基于散列的集合一起正常运作**，这样的集合包括HashMap、HashSet和Hashtable。

参考：

Java == ,equals 和 hashcode 的区别和联系(阿里面试).note

24.分布式和集群的区别是什么?

答：集群是个物理形态，分布式是个工作方式。分布式是指将不同的业务分布在不同的地方。 而集群指的是将几台服务器集中在一起，实现同一业务。**简单说，分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率**。

25.能否在加载类的时候，对字节码进行修改?

答：使用Java探针技术，能够实现

26.如果我想跟踪一个请求，从接收请求，处理到返回的整个流程，有没有好的办法?

答：ThreadLocal 可以做到传递参数。这是ThreadLocal的一个功能。很多人可能不知道，因为threadlocal是局部变量，只要线程不销毁，就会一直存在，因此可以使用threadlocal来跟踪传递参数；

ThreadLocal 适用于如下两种场景

- 每个线程需要有自己单独的实例
- 实例需要在多个方法中共享，但不希望被多线程共享

参考：

Java进阶（七）正确理解Thread Local的原理与适用场景.note

Java并发编程：深入剖析ThreadLocal.note

ThreadLocal 定义，以及是否可能引起的内存泄露(threadlocalMap的Key是弱引用，用线程池有可能泄露).note

Java学习记录--ThreadLocal使用案例.note

27.concurrentHashMap 在1.7与1.8底下的区别，

答：JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树

(1) 从1.7到1.8版本，由于HashEntry从链表 变成了红黑树所以 concurrentHashMap的时间复杂度从O(n)到O(log(n))

(2) HashEntry最小的容量为2

(3)Segment的初始化容量是16;

(4)HashEntry在1.8中称为Node,链表转红黑树的值是8 ,当Node链表的节点数大于8时Node会自动转化为TreeNode,会转换成红黑树的结构

ConcurrentHashMap与HashMap相比，有以下不同点

- ConcurrentHashMap线程安全，而HashMap非线程安全
- HashMap允许Key和Value为null，而ConcurrentHashMap不允许
- HashMap不允许通过Iterator遍历的同时通过HashMap修改，而ConcurrentHashMap允许该行为，并且该更新对后续的遍历可见

参考：

ConcurrentHashMap原理分析（1.7与1.8）-put和 get 需要执行两次Hash.note

28.Hashtable、HashMap、HashSet、TreeMap、TreeSet专题

答：**Hashtable:**

(1)Hashtable的存储结构 (数组+链表)

(2)Hashtable的扩容原理，扩容因子0.75，bucket的初始大小11.(扩容的函数为2N+1,hashMap的扩容函数是2N,之所以是2的倍数，是因为，Hashtable为了保证速度，扩容直接位移<<1这样就是2的倍数)

(3)使用链表的头插法，也就是新的键值对插在链表的头部，而不是链表的尾部

(4)index求法：int index = (hash & 0x7FFFFFFF) % tab.length;

HashMap(JDK1.7):

(1)Hashtable的存储结构 (数组+链表)

(2)Hashtable的扩容原理，扩容因子0.75，bucket的初始大小16.(扩容的函数为2N)

(3)使用链表的头插法，也就是新的键值对插在链表的头部，而不是链表的尾部

(4)index求法：h & (length-1)

(5)扩容后index求法：h & (length-1)

HashMap(JDK1.8):

(1)使用一个Node数组来存储数据，但这个Node可能是链表结构，也可能是红黑树结构。如果插入的key的hashcode相同，那么这些key也会被定位到Node数组的同一个格子里。如果同一个格子里的key不超过8个，使用链表结构存储。如果超过了8个，那么会调用treeifyBin函数，将链表转换为红黑树。那么即使hashcode完全相同，由于红黑树的特点，查找某个特定元素，也只需要O(log n)的开销也就是说put/get的操作的时间复杂度最差只有O(log n)。
(2)HashMap的扩容原理，扩容因子0.75，bucket的初始大小16.(扩容的函数为2N)
(3)index求法：h & (length-1)
(4)扩容后index求法：在扩充HashMap的时候，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap
重要：他们的扩容方法不同，hash函数也不同
元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”
参考：
HashMap的实现原理--链表散列.note
Hashtable数据存储结构-遍历规则，Hash类型的复杂度为啥都是O(1)-源码分析.note
Java 8系列之重新认识HashMap.note

29.多个线程一起put时候，currentHashMap如何操作(1.7)
答：在将数据插入指定的HashEntry位置时（链表的尾端），会通过继承ReentrantLock的tryLock（）方法尝试去获取锁，如果获取成功就直接插入相应的位置，如果已经有线程获取该Segment的锁，那当前线程会以**自旋的方式(如果不了解自旋锁，请参考：**自旋锁原理及java自旋锁**)**去继续的调用tryLock（）方法去获取锁，超过指定次数就挂起，等待唤醒。
1.8先hash查找，如果为null，用CAS的方式插入，如果不为null，则对该元素使用synchronized关键字申请锁，然后进行操作

30.currentHashMap在多个线程下如何确定size(1.7)
答：1、第一种方案他会使用不加锁的模式去尝试多次计算ConcurrentHashMap的size，最多三次，比较前后两次计算的结果，结果一致就认为当前没有元素加入，计算的结果是准确的
2、第二种方案是如果第一种方案不符合，他就会给每个Segment加上锁，然后计算ConcurrentHashMap的size返回(美团面试官的问题,多个线程下如何确定size)
在1.8中每次添加和删除会记录addCount，需要的时候直接返回即可

31.ReentrantLock 可重入锁，指的是什么可重入，举个例子

答：**可重入指的是：线程第一次进入加锁了之后，以后就不需要获取锁了。**比如同一个对象的两个synchronized 方法可以递归调用。

ReentrantLock 内部类NonfairSync和FairSync是AQS的实现类

lock流程

第一步，通过**CAS**尝试获取锁，判断state状态是否为0，若为0则将0置为1，获取锁成功，若获取失败则调用**tryAcquire**方法，检查state字段，若为0，表示锁未被占用，那么尝试占用，若不为0，检查当前锁是否被自己占用，若被自己占用，则更新state字段，表示重入锁的次数。如果以上两点都没有成功，则获取锁失败，返回false。

第二步，将没有获取锁的线程入队，首次入队的线程对队列进行初始化，利用**自旋+CAS**给队列设置空的头结点并将线程入队，线程是用一个Node对象来包装

第三步，挂起。入队线程相继执行acquireQueued(final Node node, int arg)。这个方法让已经入队的线程尝试获取锁，若失败则会被挂起。线程入队后能够挂起的前提是，它的前驱节点的状态为SIGNAL，它的含义是“Hi，前面的兄弟，如果你获取锁并且出队后，记得把我唤醒！”。所以shouldParkAfterFailedAcquire会先判断当前节点的前驱是否状态符合要求，若符合则返回true，然后调用parkAndCheckInterrupt，将自己挂起。如果不符合，再看前驱节点是否>0(CANCELLED)，若是那么向前遍历直到找到第一个符合要求的前驱，若不是则将前驱节点的状态设置为SIGNAL。整个流程中，如果前驱结点的状态不是SIGNAL，那么自己就不能安心挂起，需要去找个安心的挂起点，同时可以再尝试下看有没有机会去尝试竞争锁(if (p == head && tryAcquire(arg)) 当前节点是老二就可以尝试获得锁)。

unLock流程

流程大致为先尝试释放锁，若释放成功，那么查看头结点的状态是否为SIGNAL，如果是则唤醒头结点的下个节点关联的线程，如果释放失败那么返回false表示解锁失败。

参考：

Java不可重入锁和可重入锁理解.note

ReentrantLock实现原理.note

32.公平锁与非公平锁

答：公平锁是指多个线程同时尝试获取同一把锁时，获取锁的顺序按照线程达到的顺序，而非公平锁则允许线程“插队”。synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁。

33.ReentrantLock可重入锁和synchronized的区别

参考：

ReenTrantLock可重入锁（和synchronized的区别）总结.note

34.阿里的面试官问了个问题，如果corepollSize=10，MaxPollSize=20，如果来了25个线程 怎么办?

答：**当线程数小于corePoolSize时,提交一个任务创建一个线程(即使这时有空闲线程)来执行该任务。**

当线程数大于等于corePoolSize，首选将任务添加等待队列workQueue中（这里的workQueue是上面的BlockingQueue），等有空闲线程时，让空闲线程从队列中取任务。当等待队列满时，如果线程数量小于maximumPoolSize则创建新的线程，否则使用拒绝线程处理器来处理提交的任务。

参考：

多线程之线程池-各个参数的含义- 阿里，美团，京东面试题目.note

理解线程池的原理.note

35.newFixedThreadPool和newSingleThreadExector使用的是LinkedBlockingQueue的无界模式(美团面试题目)。

36.拒绝处理器的拒绝策略

答：**首先创建一个线程池，然后根据任务的数量逐步将线程增大到corePoolSize，如果此时仍有任务增加，则放置到workQueue中，**直到workQueue爆满为止，然后继续增加池中的线程数量（增强处理能力），最终达到maxinumPoolSize。那如果此时还有任务要增加进来呢？这就需要handler来处理了，**或者丢弃新任务，或者抛出异常，或者挤占已有的任务中等待时间最久的，或者用当前线程来执行（拒绝策略，美团面试）**。在任务队列和线程池都饱和的情况下，一旦有线程处于等待（任务处理完毕，没有新任务）状态的时间超过keepAliveTime，则该线程终止，也就是说池中的线程数量会逐渐降低，直至为corePoolSize数量为止。

参考：

多线程之线程池-各个参数的含义- 阿里，美团，京东面试题目.note

线程池的拒绝策略.note

37.后台线程相关

参考：

Java多线程—后台线程(daemon).note

38.说说进程与线程的区别

答：

进程是资源分配的最小单位，线程是程序执行的最小单位。

进程有自己的独立地址空间，**每启动一个进程，系统就会为它分配地址空间**，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而**线程是共享进程中的数据的，使用相同的地址空间**，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。

线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。

但是**多进程程序更健壮**，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

39.关于线程中断

答：关于中断：它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为true）。**中断只会影响到wait状态、sleep状态和join状态**。被打断的线程会抛出InterruptedException。

Thread.interrupted()检查当前线程是否发生中断，返回boolean。**synchronized在获锁的过程中是不能被中断的**。

中断是一个状态！interrupt()方法只是将这个状态置为true而已。所以说正常运行的程序不去检测状态，就不会终止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted())，则同样可以在中断后离开代码体。

40.sleep、yield、wait、join的区别(阿里面试)

参考：

sleep与wait是否会占用cpu时间.note

sleep、yield、wait、join的区别(阿里面试).note