leo儒猿群分享

大家好，自我介绍一下：10年经验，普本毕业，坐标北京，这次跳槽进入了阿里。分享一下这次面试经验以及平时学习的积累
我的工作年限算是比较长，都有中年危机了，跟着石衫老师的架构课学习了两年，做技术一路走过只有脚踏实地的学习总结还有多积累、多思考才能有所进步，本次跳槽其实我是整整准备了一年半，充分利用周末和休假的时间学习提高，看石衫老师的课程的同时一定同步的做笔记，重要部分标红，我还看了很多相关书籍，书籍里的例子也是每个都必须敲一遍，看书的同时也做笔记把重要的记下来并标红，面试前一周做突击用

一：面试了哪些公司?
 阿里巴巴 快手 滴滴 京东数科，拿到了哪些公司的offer：阿里巴巴 快手。由于已经拿到了心仪的offer，就没有继续约其他大厂的面试了

二：面试前的准备：
java基础，代表的有原生的List、Map、并发和线程池、TCP、网络等
知识点对应的老师的课程：
1:java架构课程的JDK源码剖析系列，还有架构课程里面的其他专题，
2:互联网Java工程师面试突击（第一二三季）
3: 儒猿技术窝上面的所有专栏
这个一集不漏的需要看完看懂，老师画的图看自己再手动默写几遍理解原理，这些基础知识太重要，必问！

三：面试官提问的部分问题： 这些问题我都会结合文字+流程图/原理图 做非常深入的解答
问题：简述HashMap的底层原理
（1）hash算法：为什么要高位和低位做异或运算？答：让高位也参与hash寻址运算，降低hash冲突
（2）hash寻址：为什么是hash值和数组.length – 1进行与运算？答：因为取余算法效率很低，按位与运算效率高

（3）hash冲突的机制：链表，超过8个以后，红黑树（数组的容量大于等于64）

（4）扩容机制：数组2倍扩容，重新寻址（rehash），hash & n – 1，判断二进制结果中是否多出一个bit的1，如果没多，那么就是原来的index，如果多了出来，那么就是index + oldCap，通过这个方式。就避免了rehash的时候，用每个hash对新数组.length取模，取模性能不高，位运算的性能比较高
JDK 1.8以后，优化了一下，如果一个链表的长度超过了8，就会自动将链表转换为红黑树，查找的性能，是O(logn)，这个性能是比O(n)要高的

（5）红黑树是二叉查找树，左小右大，根据这个规则可以快速查找某个值
（6）但是普通的二叉查找树，是有可能出现瘸子的情况，只有一条腿，不平衡了，导致查询性能变成O(n)，线性查询了
（7）红黑树，红色和黑色两种节点，有一大堆的条件限制，尽可能保证树是平衡的，不会出现瘸腿的情况
（8）如果插入节点的时候破坏了红黑树的规则和平衡，会自动重新平衡，变色（红 <-> 黑），旋转，左旋转，右旋转

问题：volatile关键字底层原理，volatile关键字是否可以禁止指令重排以及如何底层如何实现的指令重排
（1）这里贴下石杉老师在讲volatile关键字底层原理画的图：硬件级别的原理:
http://note.youdao.com/s/Mr2SnBoK
下面是我根据老师的思路学习的笔记

（2）主动从内存模型开始讲起，原子性、可见性、有序性的理解，volatile关键字的原理
java内存模型：http://note.youdao.com/s/MKm6vAP8

（3）可见性：一个线程修改了变量，其他线程能马上读取到该变量的最新值
read（从主存读取），load（将主存读取到的值写入工作内存），use（从工作内存读取数据来计算），assign（将计算好的值重新赋值到工作内存中），
store（将工作内存数据写入主存），write（将store过去的变量值赋值给主存中的变量）
这个是流程图：http://note.youdao.com/s/XazdAWWu

（4）volatile读的内存语义如下：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

这个是流程图：http://note.youdao.com/s/GBzGnrOH

（4-1）当读flag变量后，本地内存B包含的值已经被置为无效。此时，线程B必须从主内存中读取共享变量，线程B的读取操作将导致本地内存B与主内存
中的共享变量的值变成一致。

（4-2）volatile写和volatile读的内存语义总结：

线程A写一个volatile变量，实质上是线程A向接下来将要读这个volatile变量的某个线程发出了（其对共享变量所做修改的）消息。

线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。

线程A写一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息。

（5）锁的释放和获取的内存语义：当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。

当线程获取锁时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须从主内存中读取变量。

（6）有序性：基于happens-before原则来看volatile关键字如何保证有序性

这个是流程图：http://note.youdao.com/s/BPU2J7te

happens-before规则

（6-1）程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

（6-2）监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

（6-3）volatile变量规则：对一个volatile变量域的写，happens-before于任意后续对这个volatile域的读

（6-4）传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

（6-5）start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。

（6-6）join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before与线程A从ThreadB.join()操作成功返回。

（7） 原子性：volatile关键字不能保证原子性，唯一的场景就是在32位虚拟机，对long/double变量的赋值写是原子的，volatile关键字底层原理，lock指令以及内存屏，
（8）lock指令：volatile实现的两条原则
（8-1）Lock前缀指令会引起处理器缓存回写到内存。
（8-2）一个处理器的缓存回写到内存会导致其他处理器的缓存失效。
（8-3）缓存一致性协议：http://note.youdao.com/s/NDbe0gMB

问题：线程有几种状态，状态之间的变化是怎样的?
Java线程在运行的声明周期中可能处于6种不同的状态，在给定的一个时刻，线程只能处于其中的一个状态
这里我弄了几张图：
http://note.youdao.com/s/ODYruIR9
http://note.youdao.com/s/cm4ARggj

问题：简述线程池的原理，自定义线程池的参数以及每个参数的意思，线程池有哪几种，分别的应用场景举例
大家先看下这个构造图：http://note.youdao.com/s/AO4EncTZ
corePoolSize：线程池里应该有多少个线程

---

maximumPoolSize：如果线程池里的线程不够用了，等待队列还塞满了，此时有可能根据不同的线程池的类型，可能会增加一些线程出来，但是最多把线程数量增加到maximumPoolSize指定的数量
keepAliveTime + TimeUnit：如果你的线程数量超出了corePoolSize的话，超出corePoolSize指定数量的线程，就会在空闲keepAliveTime毫秒之后，就会自动被释放掉
workQueue：你的线程池的等待队列是什么队列

---

threadFactory：在线程池里创建线程的时候，你可以自己指定一个线程工厂，按照自己的方式创建线程出来

RejectedExecutionHandler：如果线程池里的线程都在执行任务，然后等待队列满了，此时增加额外线程也达到了maximumPoolSize指定的数量了，这个时候实在无法承载更多的任务了，此时就会执行这个东西（拒绝策略）

上面的基本参数的意义以外。我还推荐大家看下 美团技术团队写的《Java线程池实现原理及其在美团业务中的实践》

https://tech.meituan.com/2020/04/02/java-pooling-pratice-in-meituan.html 这篇文章，写的非常干。

问题：简述OSI七层网络模型，TCP/IP四层网络模型

OSI七层网络模型，网络的七层加工从下到上主要包括物理层，数据链路层，网络层，传输层，会话层，表示层，应用层

这个是OSI七层网络模型：http://note.youdao.com/s/AstSGIs7

问题：简述TCP三次握手以及四次挥手

TCP三次握手的过程如下：

（1）客户端发送SYN(seq=x)报文给服务器端，进入SYN_SEND状态。

（2）服务器端收到SYN报文，回应一个SYN(seq=y)和ACK(ack = x+1)报文，进入SYN_RECV状态。

（3）客户端收到服务器端的SYN报文，回应一个ACK(ack=y+1)报文，进入Established状态。

TCP三次握手的过程图：http://note.youdao.com/s/biib0eAF

TCP四次挥手的过程如下：http://note.youdao.com/s/HCXT8KMR

学习资料：石杉老师在架构班讲的：《讲给Java工程师听的大白话网络课程》

推荐书籍： 《网络是怎样连接的》《图解TCP/IP》 《图解网络硬件》 《图解HTTP》

问题：CMS垃圾回收的过程

这个是JVM内存划分的图：http://note.youdao.com/s/9MR4PLq7

这里援引下儒猿群群友根据《从 0 开始带你成为JVM实战高手》专栏 总结出来的图，分享给大家，https://www.processon.com/view/link/5e69db12e4b055496ae4a673

CMS的工作机制相对复杂，垃圾回收过程包含如下4个步骤

（1）初始标记：只标记和GC Roots直接关联的对象，速度很快，需要暂停所有工作线程。

（2）并发标记：和用户线程一起工作，执行GC Roots跟踪标记过程，不需要暂停工作线程。

（3）重新标记：在并发标记过程中用户线程继续运行，导致在垃圾回收过程中部分对象的状态发生变化，为了确保这部分对象的状态正确性，需要对其重新标记并暂停工作线程。

（4）并发清除：和用户线程一起工作，执行清除GC Roots不可达对象的任务，不需要暂停工作线程。

问题：G1与CMS的区别，你们公司使用的是哪个，为什么？（这个需要结合自己的业务场景回答）

相对于CMS垃圾收集器，G1垃圾收集器两个突出的改进。

（1）基于标记整理算法，不产生内存碎片。

（2）可以精确地控制停顿时间，在不牺牲吞吐量的前提下实现短停顿垃圾回收。

问题：JVM参数举例，讲讲为什么这么设置，为了避免fullGC的停顿对系统的影响，有哪些解决方案?

由于文本不方便贴代码，贴在在了有道云笔记里面：http://note.youdao.com/s/X4Qmucr0

为解决应用在午高峰发生 full gc 而影响系统响应时间问题, 考虑低峰期主动进行 full gc 对 old 区进行释放.

确保启动参数中 -XX:+DisableExplicitGC 项被删除, 该参数作用是禁止 System.gc() 调用. (启动参数一般配在 start 脚本中)

在启动参数中加入 -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses, 该参数的作用是主动 System.gc() 时调用 CMS 算法进行 gc 操作.

问题：内存模型以及分区，需要详细到每个区放什么

JVM 分为堆区和栈区，还有方法区，初始化的对象放在堆里面，引用放在栈里面， class 类信息常量池(static 常量和 static 变量)等放在方法区

（1） 方法区:主要是存储类信息，常量池(static 常量和 static 变量)，编译后的代码(字 节码)等数据

（2） 堆:初始化的对象，成员变量 (那种非 static 的变量)，所有的对象实例和数组都要 在堆上分配

（3） 栈:栈的结构是栈帧组成的，调用一个方法就压入一帧，帧上面存储局部变量表，操 作数栈，方法出口等信息，局部变量表存放的是 8 大基础类型加上一个应用类型，所 以还是一个指向地址的指针

（4） 本地方法栈:主要为 Native 方法服务

（5） 程序计数器:记录当前线程执行的行号

问题：JVM内存分那几个区，每个区的作用是什么?

java 虚拟机主要分为以下一个区:

方法区:

1. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生 GC，在这里 进行的 GC 主要是对方法区里的常量池和对类型的卸载

---

2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后 的代码等数据。

3. 该区域是被线程共享的。

4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池 具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量 池中。

虚拟机栈:

1. 虚拟机栈也就是我们平常所称的栈内存,它为 java 方法服务，每个方法在执行的时候都
会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。

2. 虚拟机栈是线程私有的，它的生命周期与线程相同。

3. 局部变量表里存储的是基本数据类型、returnAddress 类型(指向一条字节码指令的地 址)和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表 对象的句柄或者与对象相关联的位

置。局部变量所需的内存空间在编译器间确定

4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索 引来访问，而是压栈和出栈的方式 5.每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了 支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接 引用。

本地方法栈和虚拟机栈类似，只不过本地方法栈为 Native 方法服务。
堆：
java 堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这 里创建，因此该区域经常发生垃圾回收操作。
程序计数器 内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码 指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内 存区域是唯一一个 java 虚拟机规范没有规定任何 OOM 情况的区域。

问题：堆里面的分区：Eden，survival (from+ to)，老年代，各自的特点。

堆里面分为新生代和老生代(java8 取消了永久代，采用了 Metaspace)，新生代包 含 Eden+Survivor 区，
survivor 区里面分为 from 和 to 区，内存回收时，如果用的是复 制算法，
从 from 复制到 to，当经过一次或者多次 GC 之后，存活下来的对象会被移动 到老年区，当 JVM 内存不够用的时候，
会触发 Full GC，清理 JVM 老年区 当新生区满了之后会触发 YGC,先把存活的对象放到其中一个 Survice 区，然后进行垃圾清理。
因为如果仅仅清理需要删除的对象，这样会导致内存碎 片，因此一般会把 Eden 进行完全的清理，然后整理内存。那么下次 GC 的时候，
就会使用下一个 Survive，这样循环使用。如果有特别大的对象，新生代放不下， 就会使用老年代的担保，直接放到老年代里面。因为 JVM 认为，一般大对象的存 活时间一般比较久远。

问题：如何判断一个对象是否存活？（或者GC对象的判定方法）
判断一个对象是否存活有两种方法:
1. 引用计数法 所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，
就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说 明此对象没有被引用，
也就是“死对象”,将会被垃圾回收. 引用计数法有一个缺陷就是无法解决循环引用问题，
也就是说当对象 A 引用对象 B，对象 B 又引用者对象 A，那么此时 A,B
对象的引用计数器都不为零，也就造成无法完成垃圾回 收，所以主流的虚拟机都没有采用这种算法。

2.可达性算法(引用链法)
该算法的思想是:从一个被称为 GC Roots 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。
在 java 中可以作为 GC Roots 的对象有以下几种: • 虚拟机栈中引用的对象
方法区类静态属性引用的对象 • 方法区常量池引用的对象

本地方法栈 JNI 引用的对象 虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比不一 定会被回收。当一个对象不可达 GC Root 时，这个对象并 不会立马被回收，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记 如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行 一次筛选，筛选的条件是是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法
或者已被虚拟机调用过，那么就认为是没必要的。

如果该对象有必要执行 finalize()方法，那么这个对象将会放在一个称为 F-Queue 的对队 列中，虚拟机会触发一个 Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承 诺一直等待它运行完，这是因为如果 finalize()执行缓慢或者发生了死锁，那么就会造成 F- Queue 队列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行 第二次被标记，这时，该对象将被移除”即将回收”集合，等待回收。

问题：服务类加载过多引发的OOM问题如何排查

如果服务出现无法调用接口假死的情况，首先要考虑的是两种问题
（1）第一种问题，这个服务可能使用了大量的内存，内存始终无法释放，因此导致了频繁GC问题。
也许每秒都执行一次Full GC，结果每次都回收不了多少，最终导致系统因为频繁GC，频繁Stop the World，接口调用出现频繁假死的问题
（2）第二种问题，可能是这台机器的CPU负载太高了，也许是某个进程耗尽了CPU资源，导致你这个服务的线程始终无法得到CPU资源去执行，
也就无法响应接口调用的请求。这也是一种情况。

在内存使用这么高的情况下会发生什么?
第一种，是内存使用率居高不下，导致频繁的进行Full GC，gc带来的stop the world问题影响了服务。
第二种，是内存使用率过多，导致JVM自己发生OOM。
第三种，是内存使用率过高，也许有的时候会导致这个进程因为申请内存不足，直接被操作系统把这个进程给杀掉了

问题：如何在JVM内存溢出的时候自动dump内存快照?
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/usr/local/app/oom
第一个参数意思是在OOM的时候，自动dump内存快照出来，第二个参数是说把内存快照放到哪去
自己阅读的书籍举例：《实战Java虚拟机：JVM故障诊断与性能优化（第2版）》

Netty知识点对应的老师的课程：《Netty核心功能精讲以及核心源码剖析》
问题：NIO开发的话为什么选择netty
不选择Java原生NIO编程的原因
（1）NIO的类库和API的繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。
（2）需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网络编程非常熟悉，才能写出高质量的NIO程序。

（3）可靠性能力补齐，工作量和难度都非常大。例如客户端面临重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理的问题，NIO编程的特点就是功能开发相对
容易，但是可靠性能力补齐工作量和难度都非常大
（4）JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%

为什么选择Netty
（1）API使用简单，开发门槛低；
（2）功能强大，预置了多种编解码弄能，支持多种主流协议；
（3）定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；

（4）性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；
（5）成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；
（6）社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；
（7）经历了大规模的商业应用考验，质量得到验证。

问题：简述TCP粘包拆包以及解决方案
开局一个图：http://note.youdao.com/s/2KZBtMrc
假设客户端分别发送了两个数据包D1和D2给服务端，由于服务端一次读取到的字节数是不确定的，故可能存在以下4种情况

（1）服务端分两次读取到了两个独立的数据包，分别是D1和D2，没有粘包和拆包；
（2）服务端一次接收到了两个数据包，D1和D2粘合在一起，被称为TCP粘包；
（3）服务端分两次读取到了两个数据包，第一次读取到了完整的D1包和D2包的部分内容，第二次读取到了D2包的剩余内容，这被称为TCP拆包；
（4）服务端分两次读取到了两个数据包，第一次读取到了D1包的部分内容D1_1，第二次读取到了D1包的剩余内容D1_2和D2包的整包。

TCP粘包/拆包发生的原因

（1）应用程序write写入的字节大小大于套接口发送缓冲区大小；

（2）进行MSS（Maxitum Segment Size 最大分段大小）大小的TCP分段；

（3）以太网帧的payload大于MTU（Maxitum Transmission Unit 最大传输单元）进行IP分片。

粘包问题的解决策略

（1）消息定长，例如每个报文的大小为固定长度200字节，如果不够，空位补空格；

（2）在包尾增加回车换换符进行分割，例如FTP协议；

（3）将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思想为消息头的一个字段使用int32来表示消息的总长度；

（4）更复杂的应用层协议。

问题：简述netty服务端和客户端创建的流程

看下这个图：http://note.youdao.com/s/CqMn1VPv 在面试的时候回答这个图里面的流程

问题：简述Netty的线程模型（这个最好画图，显示出自己思路清新）

现场画图：http://note.youdao.com/s/5SIaXNPB

问题：Netty解决了java原生NIO哪些问题（空轮询的bug，这个一定要说出来）

大家看下这个博客写的挺好的：https://blog.csdn.net/baiye_xing/article/details/73351330

问题：多路复用、零拷贝等原理

1. 传统数据传送

传统数据从Socket网络中传送，需要4次数据拷贝和4次上下文切换：

- 将磁盘文件，读取到操作系统内核缓冲区；
- 将内核缓冲区的数据，拷贝到用户空间的缓冲区；

- 数据从用户空间缓冲区拷贝到内核的socket网络发送缓冲区；
- 数据从内核的socket网络发送缓冲区拷贝到网卡接口（硬件）的缓冲区，由网卡进行网络传输。

这个是流程图：http://note.youdao.com/s/6wXq7f13

传统方式，读取磁盘文件并进行网络发送，经过的4次数据拷贝和4次上下文切换是非常繁琐的。实际IO读写，需要进行IO中断，
需要CPU响应中断(带来上下文切换)，尽管后来引入DMA来接管CPU的中断请求，但四次拷贝仍在存在不必要的环节。
http://note.youdao.com/s/QMdngSkf

---

2. 零拷贝实现原理
零拷贝的目的是为了减少IO流程中不必要的拷贝，以及减少用户进程地址空间和内核地址空间之间因为上下文切换而带来的开销。
由于虚拟机不能直接操作内核，因此它的实现需要操作系统OS的支持，也就是需要kernel内核暴漏API。

2.1 Netty中的零拷贝
1. Direct Buffers：Netty的接收和发送ByteBuffer采用直接缓冲区（Direct Buffer）实现零拷贝，直接在内存区域分配空间，避免了读写数据的二次内存拷贝，这就实现了读写Socket的零拷贝。
如果使用传统的堆内存缓冲区（Heap Buffer）进行Socket读写，JVM会将堆内存Buffer拷贝到直接内存中，然后才写入Socket中。相比堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

---

2. CompositeByteBuf：它可以将多个ByteBuf封装成ByteBuf，对外提供统一封装后的ByteBuf接口。CompositeByteBuf并没有真正将多个Buffer组合起来，而是保存了它们的引用，从而避免了数据的拷贝，实现了零拷贝。
传统的ByteBuffer，如果需要将两个ByteBuffer中的数据组合到一起，我们需要首先创建一个size=size1+size2大小的新的数组，然后将两个数组中的数据拷贝到新的数组中。但是使用Netty提供的组合ByteBuf，就可以避免这样的操作。

---

3. Netty的文件传输类DefaultFileRegion通过调用FileChannel.transferTo()方法实现零拷贝，文件缓冲区的数据会直接发送给目标Channel。底层调用Linux操作系统中的sendfile()实现的，数据从文件由DMA引擎拷贝到内核read缓冲区，；DMA从内核read缓冲区将数据拷贝到网卡接口（硬件）的缓冲区，由网卡进行网络传输。

问题：简述netty整体架构
来一张老师在讲netty的时候整体架构图：http://note.youdao.com/s/Zq4OixMr 把老师这张图熟记于心，这个流程最好能在面试中画出来。
自己阅读的书籍举例：《Netty权威指南（第2版）》

Redis知识点对应的老师的课程：亿级流量电商详情页系统实战
面试官提问的部分问题：
问题：分别介绍下redis的内存模型和线程模型
https://gitee.com/shishan100/Java-Interview-Advanced/blob/master/docs/high-concurrency/redis-single-thread-model.md 老师的面试训练营

问题：缓存雪崩以及穿透的解决方案?
缓存雪崩发生的现象
缓存雪崩的事前事中事后的解决方案
事前：redis高可用，主从+哨兵，redis cluster，避免全盘崩溃
事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL被打死
事后：redis持久化，快速恢复缓存数据

缓存雪崩现象图：http://note.youdao.com/s/XMyKXr25
如何解决缓存血崩 ：http://note.youdao.com/s/53UbmTKV

缓存穿透现象以及解决方案： http://note.youdao.com/s/1oONTfsh

简述redis分布式锁的原理
https://gitee.com/shishan100/Java-Interview-Advanced/blob/master/docs/distributed-system/distributed-lock-redis-vs-zookeeper.md 老师的面试训练营
自己阅读的书籍举例：《Redis设计与实现》

ZooKeeper
知识点对应的老师的课程：《08_ZooKeeper顶尖高手课程：从实战到源码》
问题：2PC与3PC是什么，两者的流程，以及优缺点

2PC， 即二阶段提交， 为了是基于分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性而设计的一种算法。
通常，二阶段提交协议也被认为是一种一致性协议，用来保证分布式系统数据的一致性。目前绝大部分的关系型数据库都是采用二阶段提交协议，来完成分布式处理的，利用该协议能够非常方便地完成
所有分布式事务参与者的协调，统一决定事务的提交或回滚，从而能够有效地保证分布式数据一致性，因此二阶段提交协议被广泛地应用在许多分布式系统中。

阶段一：提交事务请求
（1）事务询问
（2）执行事务。
（3）各参与者向协调者反馈事务询问的响应。
阶段二：执行事务提交
（1）发送提交请求
（2）事务提交
（3）反馈事务提交结果
（4）完成事务

中断事务
（1）发送回滚请求
（2）事务回滚
（3）反馈事务回滚结果
二阶段提交协议的优点：原理简单，实现方便
二阶段提交协议的缺点：同步阻塞，单点问题，脑裂，太过保守
这个是流程图：http://note.youdao.com/s/aUlgviGc

3PC，即三阶段提交，是2PC的改进版，其将二阶段提交协议的提交事务请求过程一分为二，形成了由CanCommit、PreCommit和doCommit三个阶段组成的事务处理协议
阶段一：CanCommit
1、事务询问
2、各参与者向协调者反馈事务询问的响应。
阶段二：PreCommit
执行事务预提交
1、发送预提交请求
2、事务预提交
3、各参与者向协调者反馈事务执行的响应。

中断事务
1、发送中断请求。
2、中断事务
阶段三：doCommit

1、发送提交请求
2、事务提交
3、反馈事务提交结果
4、完成事务

中断事务
1、发送中断请求
2、事务回滚
3、反馈事务回滚结果
4、中断事务

三阶段提交协议的优点：相较于二阶段提交协议，三阶段提交协议最大的优点就是降低了参与者的阻塞范围，并能够在出现单点故障后继续达成一致。
三阶段提交协议的缺点：三阶段提交协议在去除阻塞的同时也引入了新的问题，那就是在参与者接收到PreCommit消息后，如果网络出现分区，此时协调者所在的节点和参与者
无法进行正常的网络通信，在这种情况下，该参与者依然会进行事务提交，这必然出现数据的不一致。

这个是流程图：http://note.youdao.com/s/JIlRPDgf

问题：简述ZAB协议
类似于一个两阶段提交。
（1）群首向所有追随者发送一个PROPOSAL消息p。
（2）当一个追随者接收到消息p后，会响应群首一个ACK消息，通知群首其已接受提案（proposal）
（3）当收到仲裁数量的服务器发送的确认消息后（该仲裁数包括群首自己），群首就会发送消息通知追随者进行提交（COMMIT）操作。

问题：强一致性和最终一致性的区别，ZooKeeper的一致性是怎样的?
强一致性：只要写入一条数据，立马无论从zk哪台机器上都可以立马读到这条数据，强一致性，你的写入操作卡住，直到leader和全部follower都进行了commit之后，才能让写入操作返回，认为写入成功了
此时只要写入成功，无论你从哪个zk机器查询，都是能查到的，强一致性

ZAB协议机制，zk一定不是强一致性
最终一致性：写入一条数据，方法返回，告诉你写入成功了，此时有可能你立马去其他zk机器上查是查不到的，短暂时间是不一致的，但是过一会儿，最终一定会让其他机器同步这条数据，最终一定是可以查到的
研究了ZooKeeper的ZAB协议之后，你会发现，其实过半follower对事务proposal返回ack，就会发送commit给所有follower了，只要follower或者leader进行了commit，这个数据就会被客户端读取到了

那么有没有可能，此时有的follower已经commit了，但是有的follower还没有commit？绝对会的，所以有可能其实某个客户端连接到follower01，可以读取到刚commit的数据，但是有的客户端连接到follower02在这个时间还没法读取到
所以zk不是强一致的，不是说leader必须保证一条数据被全部follower都commit了才会让你读取到数据，而是过程中可能你会在不同的follower上读取到不一致的数据，但是最终一定会全部commit后一致，让你读到一致的数据的

---

zk官方给自己的定义：顺序一致性
因此zk是最终一致性的，但是其实他比最终一致性更好一点，出去要说是顺序一致性的，因为leader一定会保证所有的proposal同步到follower上都是按照顺序来走的，起码顺序不会乱
但是全部follower的数据一致确实是最终才能实现一致的
如果要求强一致性，可以手动调用zk的sync()操作

问题：羊群效应是什么，如何解决的？
zk在共享锁的获取和释放流程图：http://note.youdao.com/s/aKrNn9K9

在整个分布式锁的竞争过程中，大量的“Watcher通知”和“子节点列表获取”两个操作重复运行，
并且绝大多数的运行结果都是判断出自己并非是序号最小的节点，从而继续等待下一次通知，
这看起来显然不怎么科学。客户端无端地接收到过多和自己不相关的事件通知，如果在集群规模比较大的情况下，不仅会对ZooKeeper服务器造成巨大的性能影响和网络冲击，

更为严重的是，如果同一时间有多个节点对应的客户端完成事务或是事务中断引起节点消失，ZooKeeper服务器就会在短时间内向其余客户端发送大量的事件通知，这就是羊群效应。
这个ZooKeeper分布式锁实现中出现羊群效应的根源在于，没有找准客户端真正的关注点。我们再来回顾一下上面分布式锁的竞争过程。

它的核心逻辑在于：判断自己是否是所有子节点中序号最小的。于是很容易可以联想到，每个节点对应的客户端只需要关注比自己序号小的那个相关节点的变更情况就可以了，而不需要关注全局的子列表变更情况。
改进过的zk在共享锁的获取和释放流程图：http://note.youdao.com/s/YofrjZGB

小结：
我本次面试阿里，面试总共经历了6轮，前3轮技术面试都做了算法题，第四轮技术最终面试没有做算法题，聊项目和离职原因等
HR我面了2轮，第一轮HR面试主要聊入职阿里要做的产品以及我本人的一个职业发展规划，第二轮HR面试是HRBP面的，主要是谈薪资
和股票等

一些技术的问题大概就是上面列举出来的，重点是问我工作经历中做的项目，以及项目中的设计，遇到的问题以及解决方案，还有就是面试官会给出一个他们产品中遇到的问题让你通过你过往的工作经验给一个解决方案，也就是技术探讨。

我本次面试的重点说的项目是自研API网关，apollo配置中心的二次开发，运单系统的分库分表方案等
要想进入例如阿里这样的大厂，必须要自己有一些含金量比较高的项目拿出来给面试官讲解，并且要讲解细节，例如项目整体的架构，

整体流程，项目部署的机器配置，平时与活动的QPS峰值，流量估算经验，自定义扩展开发或者自研的原因，踩过哪些坑，以及解决的方案是什么
石衫老师的互联网java工程师面试突击训练第一季、第二季、第三季都要看完并且理解
力扣注册一个会员，刷题，我是用了1年半刷了140道题，题量不要求多，但要有代表性，例如：链表、递归、迭代等，然后充分理解解题思路即可，平时没事的时候，对着题能把代码写出来
再一个就是平时的积累和总结了。