JVM 面试题目

1.请简单描述下 JVM 运行时数据区包括哪些部分？

JVM 在执行 Java 程序的过程中会把它管理的内存分为若干个不同的区域，这些组成部分有些是线程私有的，有些则是线程共享的

线程私有的：程序计数器，虚拟机栈，本地方法栈

线程共享的：方法区，堆

2.JVM 中是怎么判断对象可回收的？

可达性分析算法

这个算法的基本思想就是通过一系列的称为 “GC Roots” 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的，可以被回收的。

3.方法区主要存储什么信息？

方法区是所有线程共享。主要用于存储类的信息、常量池、方法数据、方法代码等。方法区逻辑上属于堆的一部分，但是为了与堆进行区分，通常又叫“非堆”。

4.请简单描述下 JVM 出现的几种异常？

主要是两种：StackOverFlowError 和 OutOfMemoryError.

StackOverFlowError：

当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverFlowError 异常。

OutOfMemoryError：

1. 当线程请求栈时内存用完了，无法再动态扩展了，此时抛出 OutOfMemoryError 异常。

2. 堆内存或者永久代代/元空间不够，无法在分配对象或存放数据，同时堆空间或者永久代/元空间无法再拓展，，此时抛出 OutOfMemoryError 异常。

3. 垃圾回收器占用 JVM 中 98%的资源，同时回收效率不到 2%，JVM 会抛出 OutOfMemoryError，

5.在默认参数下，如果 Eden 区的大小为 80M，求堆空间总大小？

根据比例可以推算出（Eden:survivor1:survivor2 =8:1:1），两个 survivor 区各 10M，新生代 100M。老年代默认是年轻代的两倍，即 200M。那么堆总大小就是 3000M。

6.什么是程序计数器？为什么 JVM 需要它？

程序计数器记录当前线程正在执行的字节码的地址或行号。

JVM 中存在线程切换，主要作用是为了确保多线程情况下 JVM 程序的正常执行。

7.请描述下 JVM 中对象创建需要的几个步骤？

1）检查加载

先执行相应的类加载过程。如果没有，则进行类加载。

2）分配内存

分配方式两种：指针碰撞和空闲列表，Java 堆空间规整的话采用指针碰撞，不规整采用空闲列表

3）内存空间初始化

JVM 需要将分配到的内存空间都初始化为零值(如 int 值为 0,boolean 值为 false 等等)。

4）设置

虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象的对象头之中。

5）对象初始化

最后调用构造方法。

8.JVM 对象的访问定位有哪两种？HostSpot 版本中用哪种？

句柄和直接指针两种方式，

句柄： 如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

直接指针： 如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象的地址。

HotSpot 中使用直接指针，使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

9.JVM 中存在哪些引用？

1. 强引用

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

2. 软引用（SoftReference）

如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

3. 弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4. 虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。

在程序设计中除了强引用，使用软引用的情况较多，这是因为软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生

10.JVM 中垃圾收集有哪些算法，各自的特点？

1. 标记-清除算法

标记-清除算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：

1）效率问题

2）空间问题（标记清除后会产生大量不连续的碎片）

2. 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

3. 标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

11.HotSpot 中的堆为什么要分为新生代和老年代？

将 Java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集

12.JVM 中哪几种情况需要对类进行初始化？

虚拟机规范严格规定了有且只有五种情况必须立即对类进行“初始化”：

1。使用 new 关键字实例化对象的时候、读取或设置一个类的静态字段的时候，已经调用一个类的静态方法的时候。

2.使用 Java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有初始化，则需要先触发其初始化。

3.当初始化一个类的时候，如果发现其父类没有被初始化就会先初始化它的父类。

4.当虚拟机启动的时候，用户需要指定一个要执行的主类（就是包含 main()方法的那个类），虚拟机会先初始化这个类；

5.使用动态语言支持的时候，如果一个 Java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getstatic,REF_putstatic,REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

13.请简单描述下双亲委派模型？

双亲委派模型（Pattern Delegation Model）,要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器。这里父子关系通常是子类通过组合关系而不是继承关系来复用父加载器的代码。

双亲委派模型的工作过程： 如果一个类加载器收到了类加载的请求，先把这个请求委派给父类加载器去完成（所以所有的加载请求最终都应该传送到顶层的启动类加载器中），只有当父加载器反馈自己无法完成加载请求时，子加载器才会尝试自己去加载。

14. CMS 收集器有哪些特点？

总的来说，优点:并发收集,低停顿。缺点:CMS 收集器对 CPU 资源很敏感、CMS 收集器不能处理浮动垃圾。

CMS 收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为 4 个步骤，包括：

1、初始标记-短暂，仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快。

2、并发标记-和用户的应用程序同时进行，进行 GC Roots 追踪的过程，标记从 GCRoots 开始关联的所有对象开始遍历整个可达分析路径的对象。这个时间比较长，所以采用并发处理（垃圾回收器线程和用户线程同时工作）

3、重新标记-短暂，为了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，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

4、并发清除，由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

CMS 对处理器资源敏感，毕竟采用了并发的收集、当处理核心数不足 4 个时，CMS 对用户的影响较大。

浮动垃圾：由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在当次收集中处理掉它们，只好留待下一次 GC 时再清理掉。这一部分垃圾就称为“浮动垃圾”。

由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。在 1.6 的版本中老年代空间使用率阈值(92%)，如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。

15.G1 收集器有哪些特点？

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代,而G1不再是这样。使用 G1 收集器时，Java 堆的内存布局就与其他收集器有很大差别，它将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region（不需要连续）的集合。每一个区域可以通过参数-XX:G1HeapRegionSize=size 来设置。

Region 中还有一块特殊区域 Humongous 区域，专门用于存储大对象，一般只要认为一个对象超过了 Region 容量的一般可认为是大对象，如果对象超级大，那么使用连续的 N 个 Humongous 区域来存储。

并行与并发：G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 Stop-The-World 停顿的时间，部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 Java 程序继续执行。

分代收集：与其他收集器一样，分代概念在 G1 中依然得以保留。虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次 GC 的旧对象以获取更好的收集效果。

空间整合：与 CMS 的“标记—清理”算法不同，G1 从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着 G1 运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次 GC。

同时 G1 追求可预测的停顿时间，G1 尝试调整新生代和老年代的比例，堆大小，晋升年龄来达到这个目标时间。

一般在 G1 和 CMS 中间选择的话平衡点在 6~8G，只有内存比较大 G1 才能发挥优势。

16、JVM 类加载机制有哪几步，分别每一步做了什么工作？

包括:加载(Loading)、验证(Verification)、准备(Preparation)、解析(Resolution)、初始化（Initialization）阶段。其中验证、准备、解析 3 个部分统称为连接（Linking）。

加载

什么是需要开始类第一个阶段“加载”，虚拟机规范没有强制约束，这点交给虚拟机的具体实现来自由把控。

“加载 loading”阶段是整个类加载（class loading）过程的一个阶段。

虚拟机需要完成以下 3 件事情：

1）通过一个类的全限定名来获取定义此类的二进制字节流。

2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3）在内存中生成一个代表这个类的 Java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

验证

是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。但从整体上看，验证阶段大致上会完成下面 4 个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

准备

是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先，这时候进行内存分配的仅包括类变量（被 static 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，

解析

是虚拟机将常量池内的符号引用替换为直接引用的过程。部分详细内容见解析

初始化

初始化阶段，虚拟机规范则是严格规定了有且只有 6 种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

1）遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：

使用 new 关键字实例化对象的时候。

读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候

调用一个类的静态方法的时候。

2）使用 Java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3）当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4）当虚拟机启动时，用户需要指定一个要执行的主类（包含 main（）方法的那个类），虚拟机会先初始化这个主类。

5）当使用 JDK 1.7 的动态语言支持时，如果一个 Java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

6)当一个接口中定义了 JDK1.8 新加入的默认方法(被 default 关键字修饰的接口方法)时，如果这个接口的实现类发生了初始化，那该接口要在其之前被初始化。

初始化是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器＜clinit＞（）方法的过程。＜clinit＞（）方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的。

＜clinit＞（）方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成＜clinit＞（）方法。

并发编程面试题目

1. 在 Java 中守护线程和用户线程的区别？

Java 中的线程分为两种：守护线程（Daemon）和用户线程（User）。

任何线程都可以设置为守护线程和用户线程，通过方法 Thread.setDaemon（bool on）；true 则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon（）必须在 Thread.start（）之前调用，否则运行时会抛出异常。

两者的区别：

唯一的区别是判断虚拟机（JVM）何时离开，Daemon 是为其他线程提供服务，如果全部的 User Thread 已经结束，Daemon 没有可服务的线程，JVM 关闭。

扩展：Thread Dump 打印出来的线程信息，含有 daemon 字样的线程即为守护进程

2. 线程与进程的区别

进程是操作系统配资源的最小单元，线程是操作系统调度的最小单元。

一个程序至少有一个进程,一个进程至少有一个线程。

3. 什么是多线程中的上下文切换

多线程会共同使用一组计算机上的 CPU，而线程数大于给程序分配的 CPU 数量时，为了让各个线程都有执行的机会，就需要轮转使用 CPU。不同的线程切换使用 CPU 发生的切换数据等就是上下文切换。

4. 死锁与活锁的区别，死锁与饥饿的区别？

死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的必要条件：

互斥条件：所谓互斥就是进程在某一时间内独占资源。

请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

不剥夺条件:进程已获得资源，在末使用完之前，不能强行剥夺。

循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

5. synchronized 底层实现原理

synchronized （this）原理：涉及两条指令：monitorenter，monitorexit；再说同步方法，从同步方法反编译的结果来看，方法的同步并没有通过指令 monitorenter 和 monitorexit 来实现，相对于普通方法，其常量池中多了 ACC_SYNCHRONIZED 标示符。

JVM 就是根据该标示符来实现方法的同步的：当方法被调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取 monitor，获取成功之后才能执行方法体，方法执行完后再释放 monitor。在方法执行期间，其他任何线程都无法再获得同一个 monitor 对象。

注意，这个问题可能会接着追问，Java 对象头信息，偏向锁，轻量锁，重量级锁及其他们相互间转化。

6. 什么是线程组，为什么在 Java 中不推荐使用？

ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。

1.线程组 ThreadGroup 对象中比较有用的方法是 stop、resume、suspend 等方法，由于这几个方法会导致线程的安全问题（主要是死锁问题），已经被官方废弃掉了，所以线程组本身的应用价值就大打折扣了。

2.线程组 ThreadGroup 不是线程安全的，这在使用过程中获取的信息并不全是及时有效的，这就降低了它的统计使用价值。

7. 什么是 Executors 框架？为什么使用 Executor 框架？

Executor 框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

每次执行任务创建线程 new Thread（）比较消耗性能，创建一个线程是比较耗时、耗资源的。

调用 new Thread（）创建的线程缺乏管理，而且可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪，还有线程之间的频繁交替也会消耗很多系统资源。

接使用 new Thread（） 启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

8. 在 Java 中 Executor 和 Executors 的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。

Executor 接口对象能执行我们的线程任务。

ExecutorService 接口继承了 Executor 接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。

使用 ThreadPoolExecutor 可以创建自定义线程池。

9. 什么是原子操作？在 Java Concurrency API 中有哪些原子类（atomic classes）？

原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。CAS 操作——Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。Java.util.concurrent.atomic 下提供了大量的原子操作类，比如原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference ，原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray ，原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

10. Java Concurrency API 中的 Lock 接口（Lock interface）是什么？对比 synchronized 它有什么优势？

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。

他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：可以使锁更公平，可以使线程在等待锁的时候响应中断，可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间，可以在不同的范围，以不同的顺序获取和释放锁。

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的（tryLock 方法）、定时的（tryLock 带参方法）、可中断的（lockInterruptibly）、可多条件队列的（newCondition 方法）锁操作。另外 Lock 的实现类基本都支持非公平锁（默认）和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

11. 什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

JDK7 提供了 7 个阻塞队列。在实现上，主要是利用了 Condition 和 Lock 的等待通知模式。

12. 什么是 Callable 和 Future?

Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值,这个返回值可以被 Future 拿到,也就是说,Future 可以拿到异步执行任务的返回值。

可以认为是带有回调的 Runnable。

Future 接口表示异步任务，是还没有完成的任务给出的未来结果。所以说 Callable 用于产生结果，Future 用于获取结果。

13. 什么是 FutureTask?

在 Java 并发程序中 FutureTask 表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是调用了 Runnable 接口所以它可以提交给 Executor 来执行。

14. 什么是竞争条件？

当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件（race condition）。

15. 启动线程的方式有几种？

两种，启动线程的方式有：

1、X extends Thread;，然后 X.start

2、X implements Runnable；然后交给 Thread 运行

16. 为什么我们调用 start（）方法时会执行 run（）方法，为什么我们不能直接调用 run（）方法？

当你调用 start（）方法时你将创建新的线程，并且执行在 run（）方法里的代码。

但是如果你直接调用 run（）方法，它不会创建新的线程也不会执行调用线程的代码，只会把 run 方法当作普通方法去执行。

17. 执行两次 start 方法可以吗？

不行，程序会抛出 IlegelState 异常。

18. 在 Java 中 CycliBarriar 和 CountdownLatch 有什么区别？

CyclicBarrier 可以重复使用，而 CountdownLatch 不能重复使用。

19. 什么是不可变对象，它对写并发应用有什么帮助？

不可变对象（Immutable Objects）即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象（Mutable Objects）。

不可变对象的类即为不可变类（Immutable Class）。Java 平台类库中包含许多不可变类，如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。

不可变对象天生是线程安全的。它们的常量（域）是在构造函数中创建的。既然它们的状态无法修改，这些常量永远不会变。

不可变对象永远是线程安全的。

只有满足如下状态，一个对象才是不可变的；

它的状态不能在创建后再被修改；

所有域都是 final 类型；并且，

它被正确创建

20. notify（）和 notifyAll（）有什么区别？

当一个线程进入 wait 之后，就必须等其他线程 notify/notifyall,使用 notifyall,可以唤醒所有处于 wait 状态的线程,使其重新进入锁的争夺队列中,而 notify 只能唤醒一个。

如果没把握，建议 notifyAll，防止 notigy 因为信号丢失而造成程序异常。

21. 什么是可重入锁（ReentrantLock）？谈谈它的实现。

线程可以重复进入任何一个它已经拥有的锁所同步着的代码块，synchronized、ReentrantLock 都是可重入的锁。在实现上，就是线程每次获取锁时判定如果获得锁的线程

是它自己时，简单将计数器累积即可，每 释放一次锁，进行计数器累减，直到计算器归零，表示线程已经彻底释放锁。

22. 当一个线程进入某个对象的一个 synchronized 的实例方法后，其它线程是否可进入此对象的其它方法？

如果其他方法没有 synchronized 的话，其他线程是可以进入的。
所以要开放一个线程安全的对象时，得保证每个方法都是线程安全的。

23. 乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。Java 里面的同步原语 synchronized 关键字的实现是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。在 Java 中 j 原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

乐观锁的实现方式：

使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

Java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

24. 什么是 CAS 操作，缺点是什么？

CAS 的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。每一个 CAS 操作过程都包含三个运算符：一个内存地址 V，一个期望的值 A 和一个新值 B，操作的时候如果这个地址上存放的值等于这个期望的值 A，则将地址上的值赋为新值 B，否则不做任何操作。

CAS 缺点：

ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A,这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

25. SynchronizedMap 和 ConcurrentHashMap 有什么区别？

SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

ConcurrentHashMap 使用分段锁来保证在多线程下的性能。

26. 写时复制容器可以用于什么应用场景？

CopyOnWrite 并发容器用于对于绝大部分访问都是读，且只是偶尔写的并发场景。比如白名单，黑名单，商品类目的访问和更新场景。

透露的思想

读写分离，读和写分开

最终一致性

使用另外开辟空间的思路，来解决并发冲突

27. volatile 有什么用？能否用一句话说明下 volatile 的应用场景？

volatile 保证内存可见性和禁止指令重排。

volatile 用于多线程环境下的一写多读，或者无关联的多写。

28. 为什么代码会重排序？

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

在单线程环境下不能改变程序运行的结果；

存在数据依赖关系的不允许重排序

29. 在 Java 中 wait 和 sleep 方法的不同？

最大的不同是在等待时 wait 会释放锁，而 sleep 一直持有锁。Wait 通常被用于线程间

交互，sleep 通常被用于暂停执行。

30. 一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler 是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候 JVM 会使用 Thread.getUncaughtExceptionHandler（）来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的 uncaughtException（）方法进行处理。

31. 为什么 wait, notify 和 notifyAll 这些方法不在 thread 类里面？

JAVA 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的 wait（）方法就有意义了。如果 wait（）方法定义在 Thread 类中，线程正在等待的是哪个锁就不明显了。简单的说，由于 wait，notify 和 notifyAll 都是锁级别的操作，所以把他们定义在 Object 类中因为锁属于对象。

32. 什么是 ThreadLocal 变量？

ThreadLocal 是 Java 里一种特殊的变量。每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。

其实现原理是，在 Thread 中有一个 ThreadLocalMap 类型的成员变量，线程的变量副本就放在这个 ThreadLocalMap 中，这个 ThreadLocalMap 就是以 ThreadLocal 为 key，线程的变量副本为 value。

33. Java 中 interrupted 和 isInterrupted 方法的区别？

interrupted（） 和 isInterrupted（）的主要区别是前者会将中断状态清除而后者不会。Java 多线程的中断机制是用内部标识来实现的，调用 Thread.interrupt（）来中断一个线程就会设置中断标识为 true。当中断线程调用静态方法 Thread.interrupted（）来检查中断状态时，中断状态会被清零。而非静态方法 isInterrupted（）用来查询其它线程的中断状态且不会改变中断状态标识。

34. 为什么 wait 和 notify 方法要在同步块中调用？

主要是因为 Java API 强制要求这样做，如果你不这么做，你的代码会抛出 IllegalMonitorStateException 异常。

35. 为什么你应该在循环中检查等待条件?

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在 notify（）方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用 wait（）方法效果更好的原因

36. 怎么检测一个线程是否拥有锁？

在 Java.lang.Thread 中有一个方法叫 holdsLock（），它返回 true 如果当且仅当当前线程拥有某个具体对象的锁。

37. 你如何在 Java 中获取线程堆栈信息？

kill -3 [Java pid]

不会在当前终端输出，它会输出到代码执行的或指定的地方去。比如，kill -3 tomcat pid, 输出堆栈到 log 目录下。

Jstack [Java pid]

这个比较简单，在当前终端显示，也可以重定向到指定文件中。

或者使用 Java 提供的拟机线程系统的管理接口 ManagementFactory.getThreadMXBean（）。

38. Java 线程池中 submit（） 和 execute（）方法有什么区别？

两个方法都可以向线程池提交任务，execute（）方法的返回类型是 void，它定义在 Executor 接口中。

而 submit（）方法可以返回持有计算结果的 Future 对象，它定义在 ExecutorService 接口中，它扩展了 Executor 接口

39. 你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的（OS dependent）。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量（从 1-10），1 代表最低优先级，10 代表最高优先级。

Java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

40. 你如何确保 main（）方法所在的线程是 Java 程序最后结束的线程？

可以使用 Thread 类的 join（）方法（或者 CountDownLatch 工具类）来确保所有程序创建的线程在 main（）方法退出前结束。

41. 为什么 Thread 类的 sleep（）和 yield （）方法是静态的？

Thread 类的 sleep（）和 yield（）方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

42. 现在有 T1、T2、T3 三个线程，你怎样保证 T2 在 T1 执行完后执行，T3 在 T2 执行完后执行？

可以用 join 方法实现。

43. 用 Java 写代码来解决生产者——消费者问题。

阻塞队列实现即可，也可以用 wait 和 notify 来解决这个问题，或者用 Semaphore

44. Java 中如何停止一个线程？

使用共享变量的方式

在这种方式中，之所以引入共享变量，是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

使用 interrupt 方法终止线程

如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？比如当一个线程由于需要等候键盘输入而被阻塞，或者调用 Thread.join()方法，或者 Thread.sleep()方法，在网络中调用 ServerSocket.accept()方法，或者调用了 DatagramSocket.receive()方法时，都有可能导致线程阻塞，使线程处于处于不可运行状态时，即使主程序中将该线程的共享变量设置为 true，但该线程此时根本无法检查循环标志，当然也就无法立即中断。所以应该尽量使用 Thread 提供的 interrupt（）方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态。

45. JVM中哪个参数是用来控制线程的栈堆栈大小的

-Xss

46. 如果同步块内的线程抛出异常锁会释放吗？

会

47. 单例模式的双重检查实现是什么？为什么并不安全？如何在Java中创建线程安全的Singleton？

不安全的根本原因是重排序会导致未初始化完成的对象可以被其他线程看见而导致错误。创建安全的单例模式有：延迟占位模式、在声明的时候就new这个类的实例、枚举

48. 请概述线程池的创建参数，怎么样合理配置一个线程池的参数？

线程的构造方法如下：

public ThreadPoolExecutor（ int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler）

corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的prestartAllCoreThreads（）方法，线程池会提前创建并启动所有核心线程。

maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize

keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于 corePoolSize 时才有用

TimeUnit

keepAliveTime 的时间单位

workQueue

用于保存等待执行的任务的阻塞队列，一般来说，我们应该尽量使用有界队列，因为使用无界队列作为工作队列会对线程池带来如下影响。

1）当线程池中的线程数达到 corePoolSize 后，新任务将在无界队列中等待，因此线程池中的线程数不会超过 corePoolSize。

2）由于 1，使用无界队列时 maximumPoolSize 将是一个无效参数。

3）由于 1 和 2，使用无界队列时 keepAliveTime 将是一个无效参数。

4）更重要的，使用无界 queue 可能会耗尽系统资源，有界队列则有助于防止资源耗尽，同时即使使用有界队列，也要尽量控制队列的大小在一个合适的范围。

所以我们一般会使用，ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue。

threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名，当然还可以更加自由的对线程做更多的设置，比如设置所有的线程为守护线程。

Executors 静态工厂里默认的 threadFactory，线程的命名规则是“pool-数字-thread-数字”。

RejectedExecutionHandler

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了 4 种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现 RejectedExecutionHandler 接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

线程池的工作机制总结

1）如果当前运行的线程少于 corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

2）如果运行的线程等于或多于 corePoolSize，则将任务加入 BlockingQueue。

3）如果无法将任务加入 BlockingQueue（队列已满），则创建新的线程来处理任务。

4）如果创建新线程将使当前运行的线程超出 maximumPoolSize，任务将被拒绝，并调用 RejectedExecutionHandler.rejectedExecution（）方法。

合理地配置线程池

要想合理地配置线程池，就必须首先分析任务特性

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

•任务的性质：CPU 密集型任务、IO 密集型任务和混合型任务。

•任务的优先级：高、中和低。

•任务的执行时间：长、中和短。

•任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。

CPU 密集型任务应配置尽可能小的线程，如配置 Ncpu+1 个线程的线程池。由于 IO 密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如 2*Ncpu。

混合型的任务，如果可以拆分，将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。可以通过 Runtime.getRuntime（）.availableProcessors（）方法获得当前设备的 CPU 个数。

对于 IO 型的任务的最佳线程数，有个公式可以计算

Nthreads = NCPU * UCPU * （1 + W/C）

其中：

❑NCPU 是处理器的核的数目

❑UCPU 是期望的 CPU 利用率（该值应该介于 0 和 1 之间）

❑W/C 是等待时间与计算时间的比率

等待时间与计算时间我们在 Linux 下使用相关的 vmstat 命令或者 top 命令查看。

优先级不同的任务可以使用优先级队列 PriorityBlockingQueue 来处理。它可以让优先级高的任务先执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，等待的时间越长，则 CPU 空闲时间就越长，那么线程数应该设置得越大，这样才能更好地利用 CPU。

建议使用有界队列。有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点儿，比如几千。

假设，我们现在有一个 Web 系统，里面使用了线程池来处理业务，在某些情况下，系统里后台任务线程池的队列和线程池全满了，不断抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行 SQL 变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞，任务积压在线程池里。

如果当时我们设置成无界队列，那么线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。

49. 请概述锁的公平和非公平，JDK 内部是如何实现的。

公平锁是指所有试图获得锁的线程按照获取锁的顺序依次获得锁，而非公平锁则是当前的锁状态没有被占用时,当前线程可以直接占用,而不需要等待。在实现上，非公平锁逻辑基本跟公平锁一致，唯一的区别是，当前线程不需要判断同步队列中是否有等待线程。

非公平锁性能高于公平锁性能。首先，在恢复一个被挂起的线程与该线程真正运行之间存在着严重的延迟。而且，非公平锁能更充分的利用 cpu 的时间片，尽量的减少 cpu 空闲的状态时间。

使用场景的话呢，其实还是和他们的属性一一相关，比如：如果业务中线程占用（处理）时间要远长于线程等待，那用非公平锁其实效率并不明显，但是用公平锁可以保证不会有线程被饿死。

50. 请概述 AQS

是用来构建锁或者其他同步组件的基础框架，比如 ReentrantLock、ReentrantReadWriteLock 和 CountDownLatch 就是基于 AQS 实现的。它使用了一个 int 成员变量表示同步状态，通过内置的 FIFO 队列来完成资源获取线程的排队工作。它是 CLH 队列锁的一种变体实现。它可以实现 2 种同步方式：独占式，共享式。

AQS 的主要使用方式是继承，子类通过继承 AQS 并实现它的抽象方法来管理同步状态，同步器的设计基于模板方法模式,所以如果要实现我们自己的同步工具类就需要覆盖其中几个可重写的方法，如 tryAcquire、tryReleaseShared 等等。

这样设计的目的是同步组件（比如锁）是面向使用者的，它定义了使用者与同步组件交互的接口（比如可以允许两个线程并行访问），隐藏了实现细节；同步器面向的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。这样就很好地隔离了使用者和实现者所需关注的领域。

在内部，AQS 维护一个共享资源 state，通过内置的 FIFO 来完成获取资源线程的排队工作。该队列由一个一个的 Node 结点组成,每个 Node 结点维护一个 prev 引用和 next 引用，分别指向自己的前驱和后继结点，构成一个双端双向链表。

同时与 Condition 相关的等待队列，节点类型也是 Node，构成一个单向链表。

51. 请概述 volatile

volatile 关键字的作用主要有两点：

多线程主要围绕可见性和原子性两个特性而展开，使用 volatile 关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到 volatile 变量，一定是最新的数据。但是 volatile 不能保证操作的原子，对任意单个 volatile 变量的读/写具有原子性，但类似于++这种复合操作不具有原子性。。

代码底层在执行时为了获取更好的性能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用 volatile 则会对禁止重排序，当然这也一定程度上降低了代码执行效率。

同时在内存语义上，当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值刷新到主内存，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。

在 Java 中对于 volatile 修饰的变量，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序问题、强制刷新和读取。

在具体实现上，volatile 关键字修饰的变量会存在一个“lock:”的前缀。它不是一种内存屏障，但是它能完成类似内存屏障的功能。Lock 会对 CPU 总线和高速缓存加锁，可以理解为 CPU 指令级的一种锁。

同时该指令会将当前处理器缓存行的数据直接写会到系统内存中，且这个写回内存的操作会使在其他 CPU 里缓存了该地址的数据无效。

52. HashMap 和 HashTable 有什么区别？

A：①、HashMap 是线程不安全的，HashTable 是线程安全的；

②、由于线程安全，所以 HashTable 的效率比不上 HashMap；

③、HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null，而 HashTable 不允许；

④、HashMap 默认初始化数组的大小为 16，HashTable 为 11，前者扩容时，扩大两倍，后者扩大两倍+1；

⑤、HashMap 需要重新计算 hash 值，而 HashTable 直接使用对象的 hashCode

53. Java 中的另一个线程安全的与 HashMap 极其类似的类是什么？同样是线程安全，它与 HashTable 在线程同步上有什么不同？

A：ConcurrentHashMap 类（是 Java 并发包 Java.util.concurrent 中提供的一个线程安全且高效的 HashMap 实现）。

HashTable 是使用 synchronize 关键字加锁的原理（就是对对象加锁）；

而针对 ConcurrentHashMap，在 JDK 1.7 中采用分段锁的方式；JDK 1.8 中直接采用了 CAS（无锁算法）+ synchronized，也采用分段锁的方式并大大缩小了锁的粒度。

54. HashMap & ConcurrentHashMap 的区别？

A：除了加锁，原理上无太大区别。

另外，HashMap 的键值对允许有 null，但是 ConCurrentHashMap 都不允许。

在数据结构上，红黑树相关的节点类

55. 为什么 ConcurrentHashMap 比 HashTable 效率要高？

A：HashTable 使用一把锁（锁住整个链表结构）处理并发问题，多个线程竞争一把锁，容易阻塞；

ConcurrentHashMap

JDK 1.7 中使用分段锁（ReentrantLock + Segment + HashEntry），相当于把一个 HashMap 分成多个段，每段分配一把锁，这样支持多线程访问。锁粒度：基于 Segment，包含多个 HashEntry。

JDK 1.8 中使用 CAS + synchronized + Node + 红黑树。锁粒度：Node（首结点）（实现 Map.Entry<K,V>）。锁粒度降低了。

56. 针对 ConcurrentHashMap 锁机制具体分析（JDK 1.7 VS JDK 1.8）？

JDK 1.7 中，采用分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构，包括两个核心静态内部类 Segment 和 HashEntry。

①、Segment 继承 ReentrantLock（重入锁） 用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶；

②、HashEntry 用来封装映射表的键-值对；

③、每个桶是由若干个 HashEntry 对象链接起来的链表。

JDK 1.8 中，采用 Node + CAS + Synchronized 来保证并发安全。取消类 Segment，直接用 table 数组存储键值对；当 HashEntry 对象组成的链表长度超过 TREEIFY_THRESHOLD 时，链表转换为红黑树，提升性能。底层变更为数组 + 链表 + 红黑树。

57. ConcurrentHashMap 在 JDK 1.8 中，为什么要使用内置锁 synchronized 来代替重入锁 ReentrantLock？

1、JVM 开发团队在 1.8 中对 synchronized 做了大量性能上的优化，而且基于 JVM 的 synchronized 优化空间更大，更加自然。

2、在大量的数据操作下，对于 JVM 的内存压力，基于 API 的 ReentrantLock 会开销更多的内存。

58. ConcurrentHashMap 简单介绍？

①、重要的常量：

private transient volatile int sizeCtl;

当为负数时，-1 表示正在初始化，-N 表示 N - 1 个线程正在进行扩容；

当为 0 时，表示 table 还没有初始化；

当为其他正数时，表示初始化或者下一次进行扩容的大小。

②、数据结构：

Node 是存储结构的基本单元，继承 HashMap 中的 Entry，用于存储数据；

TreeNode 继承 Node，但是数据结构换成了二叉树结构，是红黑树的存储结构，用于红黑树中存储数据；

TreeBin 是封装 TreeNode 的容器，提供转换红黑树的一些条件和锁的控制。

③、存储对象时（put（） 方法）：

1.如果没有初始化，就调用 initTable（） 方法来进行初始化；

2.如果没有 hash 冲突就直接 CAS 无锁插入；

3.如果需要扩容，就先进行扩容；

4.如果存在 hash 冲突，就加锁来保证线程安全，两种情况：一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入；

5.如果该链表的数量大于阀值 8，就要先转换成红黑树的结构，break 再一次进入循环

6.如果添加成功就调用 addCount（） 方法统计 size，并且检查是否需要扩容。

④、扩容方法 transfer（）：默认容量为 16，扩容时，容量变为原来的两倍。

helpTransfer（）：调用多个工作线程一起帮助进行扩容，这样的效率就会更高。

⑤、获取对象时（get（）方法）：

1.计算 hash 值，定位到该 table 索引位置，如果是首结点符合就返回；

2.如果遇到扩容时，会调用标记正在扩容结点 ForwardingNode.find（）方法，查找该结点，匹配就返回；

3.以上都不符合的话，就往下遍历结点，匹配就返回，否则最后就返回 null。

59. ConcurrentHashMap 的并发度是什么？

1.7 中程序运行时能够同时更新 ConccurentHashMap 且不产生锁竞争的最大线程数。默认为 16，且可以在构造函数中设置。当用户设置并发度时，ConcurrentHashMap 会使用大于等于该值的最小 2 幂指数作为实际并发度（假如用户设置并发度为 17，实际并发度则为 32）。

1.8 中并发度则无太大的实际意义了，主要用处就是当设置的初始容量小于并发度，将初始容量提升至并发度大小。

60. 什么是锁消除和锁粗化？

锁消除：指虚拟机即时编译器在运行时，对一些代码上要求同步，但被检测到不可能

存在共享数据竞争的锁进行消除。主要根据逃逸分析。

锁粗化：原则上，同步块的作用范围要尽量小。但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作在循环体内，频繁地进行互斥同步操作也会导致不必要的性能损耗。锁粗化就是增大锁的作用域。

61. 除了 ReetrantLock， JUC 下还有哪些并发工具？

通常所说的并发包（JUC）也就是 Java.util.concurrent 及其子包，集中了 Java 并发的各种基础工具类，具体主要包括几个方面：

提供了 CountDownLatch、CyclicBarrier、Semaphore 等，比 Synchronized 更加高级，可以实现更加丰富多线程操作的同步结构。

提供了 ConcurrentHashMap、有序的 ConcunrrentSkipListMap，或者通过类似快照机制实现线程安全的动态数组 CopyOnWriteArrayList 等，各种线程安全的容器。

提供了 ArrayBlockingQueue 、 SynchorousQueue 或针对特定场景的 PriorityBlockingQueue 等，各种阻塞队列实现。

强大的 Executor 框架，可以创建各种不同类型的线程池，调度任务运行等。

62. 什么是 Java 的内存模型，Java 中各个线程是怎么彼此看到对方的变量的？

Java 的内存模型定义了程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出这样的底层细节。

此处的变量包括实例字段、静态字段和构成数组对象的元素，但是不包括局部变量和方法参数，因为这些是线程私有的，不会被共享，所以不存在竞争问题。

Java 中各个线程是怎么彼此看到对方的变量的呢？Java 中定义了主内存与工作内存的概念：

所有的变量都存储在主内存，每条线程还有自己的工作内存，保存了被该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递需要通过主内存。

Spring 面试题目

1. 你为什么使用 Spring?

轻量级：Spring 在大小和透明性方面绝对属于轻量级的，基础版本的 Spring 框架大约只有 2MB。

控制反转(IOC)：Spring 使用控制反转技术实现了松耦合。依赖被注入到对象，而不是创建或寻找依赖对象。

面向切面编程(AOP)： Spring 支持面向切面编程，同时把应用的业务逻辑与系统的服务分离开来。

容器：Spring 包含并管理应用程序对象的配置及生命周期。

MVC 框架：Spring 的 web 框架是一个设计优良的 web MVC 框架，很好的取代了一些 web 框架。

事务管理：Spring 对下至本地业务上至全局业务(JAT)提供了统一的事务管理接口。

异常处理:Spring 提供一个方便的 API 将特定技术的异常(由 JDBC, Hibernate, 或 JDO 抛出)转化为一致的、Unchecked 异常。

2. Spring 支持几种 bean 的作用域?

1)singleton：默认，每个容器中只有一个 bean 的实例，单例的模式由 BeanFactory 自身来维护。

2)prototype：为每一个 bean 请求提供一个实例。

3)request：为每一个网络请求创建一个实例，在请求完成以后，bean 会失效并被垃圾回收器回收。

4)session:与 request 范围类似,确保每个 session 中有一个 bean 的实例,在 session 过期后，bean 会随之失效。

5)global-session：全局作用域，global-session 和 Portlet 应用相关。当你的应用部署在 Portlet 容器中工作时，它包含很多 portlet。如果你想要声明让所有的 portlet 共用全局的存储变量的话，那么这全局变量需要存储在 global-session 中。全局作用域与 Servlet 中的 session 作用域效果相同。

3. 请问 Spring 有几种自动装配模式？

自动装配提供五种不同的模式供 Spring 容器用来自动装配 beans 之间的依赖注入:

no：默认的方式是不进行自动装配，通过手工设置 ref 属性来进行装配 bean。

byName：通过参数名自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byName。之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。

byType：通过参数的数据类型自动自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byType。之后容器试图匹配和装配和该 bean 的属性类型一样的 bean。如果有多个 bean 符合条件，则抛出错误。

constructor：这个同 byType 类似，不过是应用于构造函数的参数。如果在 BeanFactory 中不是恰好有一个 bean 与构造函数参数相同类型，则抛出一个严重的错误。

autodetect：如果有默认的构造方法，通过 construct 的方式自动装配，否则使用 byType 的方式自动装配。

4. 对 Java 接口代理模式的实现原理的理解?

答：

1、字符串拼凑出代理类
2、用流的方式写到.Java 文件
3、动态编译.Java 文件
4、自定义类加载器把.class 文件加载到 jvm
5、返回代理类实例
其实核心就是根据接口反射出接口中的所有方法，然后通过拼凑或者 cglib 字节码的方式把代理类反射出来，而代理类在内存中就会对某种类型的类进行调用，invocationHandler（Jdk）或者 MethodInterceptor（cglib）

5. 怎么理解面向切面编程的切面?

答：

切面指的是一类功能，比如事务，缓存，日志等。切面的作用是在目标对象方法执行前或者后执行切面方法

6. 什么是 IOC 容器？

答：

Spring IOC 负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理

这些对象的生命周期。

Ioc 是把对象的控制权较给框架或容器，容器中存储了众多我们需要的对象，然后我们就无需再手动的在代码中创建对象。需要什么对象就直接告诉容器我们需要什么对象，容器会把对象根据一定的方式注入到我们的代码中。注入的过程被称为 DI。有时候需要动态的指定我们需要什么对象，这个时候要让容器在众多对象中去寻找，容器寻找需要对象的过程，称为 DL(Dependency Lookup, 依赖查找)。按照上面的理解，那么 IOC 包含了 DI 与 DL，并且多了对象注册的过程。

7. 为什么需要代理模式?

答：

代理其实是对开闭原则和里氏代换原则的一种实现，它强调在不修改老代码的基础上扩展功能，代理模式大大提高了代码的灵活性，同时也使代码的可读性变差，同时动态代理也加大了内存溢出的风险。

8. 讲讲静态代理模式的优点及其瓶颈?

答：

静态代理一种傻瓜式的对目标类的增强，其核心就两点，1、跟目标类实现同一接口 2、持有目标类的引用。 如果需要对目标类方法进行增强，就只要调用代理类的方法，在方法里面写增强功能然后调用目标类方法即可。优点就是代码直观且执行速度快，缺点就是不够灵活，代码量比较大，每一种类型的目标类都需要单独写一个代理类。

9. 讲解 Spring 框架中基于 Schema 的 AOP 实现原理?

答：

这个是基于 xml 配置方式的 aop

<aop:config proxy-target-class="true">

<aop:pointcut expression="execution(public void org.aop.Object.Student.display())" id="pointcut"/> <!-- 切点的声明定义-->

<aop:aspect ref="advice"> <!-- 关联通知类-->
<aop:before method="before" pointcut-ref="pointcut"/> <!-- 前置通知-->
<aop:after-returning method="after" pointcut-ref="pointcut" /><!-- 后置通知-->
<aop:after-throwing method="exception" pointcut-ref="pointcut" throwing="x"/><!-- 异常通知-->
<aop:around method="around" pointcut-ref="pointcut"/><!-- 环绕通知-->
</aop:aspect>
</aop:config>

AOP 实现流程

1、aop:config 自定义标签解析

2、自定义标签解析时封装对应的 aop 入口类，类的类型就是 BeanPostProcessor 接口类型

3、Bean 实现化过程中会执行到 aop 入口类中

4、在 aop 入口类中，判断当前正在实例化的类是否在 pointcut 中，pointcut 可以理解为一个模糊匹配，是一个 joinpoint 的集合

5、如果当前正在实例化的类在 pointcut 中，则返回该 bean 的代理，同时把所有配置的 advice 封装成 MethodInterceptor 对象加入到容器中，封装成一个过滤器链

代理对象调用，jdk 调到 invocationHandler 中，cglib 调到 MethodInterceptor 的 callback 类中，然后在 invoke 方法中执行过滤器链。

10. 讲解 Spring 框架中如何基于 AOP 实现的事务管理?

答:

事务管理，是一个切面。在 aop 环节中，其他环节都一样，事务管理就是有自己单独的 advice，有单独的通知，是 TransactionInterceptor，它一样的会在拦截器链中被执行到，这个 TransactionInterceptor 拦截器类是通过解析<tx:advice>自定义标签得到的。

11. Spring 在 Bean 创建过程中是如何解决循环依赖的?

答:

循环依赖只会存在在单例实例中，多例循环依赖直接报错。

A 类实例化后，把实例放 map 容器中，A 类中有一个 B 类属性，A 类实例化要进行 IOC 依赖注入，这时候 B 类需要实例化，B 类实例化跟 A 类一样，实例化后方 map 容器中。B 类中有一个 A 类属性，接着 B 类的 IOC 过程，又去实例化 A 类，这时候实例化 A 类过程中从

map 容器发现 A 类已经在容器中了，就直接返回了 A 的实例，依赖注入到 B 类中 A 属性中，B 类 IOC 完成后，B 实例化就完全完成了，就返回给 A 类的 IOC 过程。这就是循环依赖的解决。

12. Spring 中 Bean 是如何管理的？

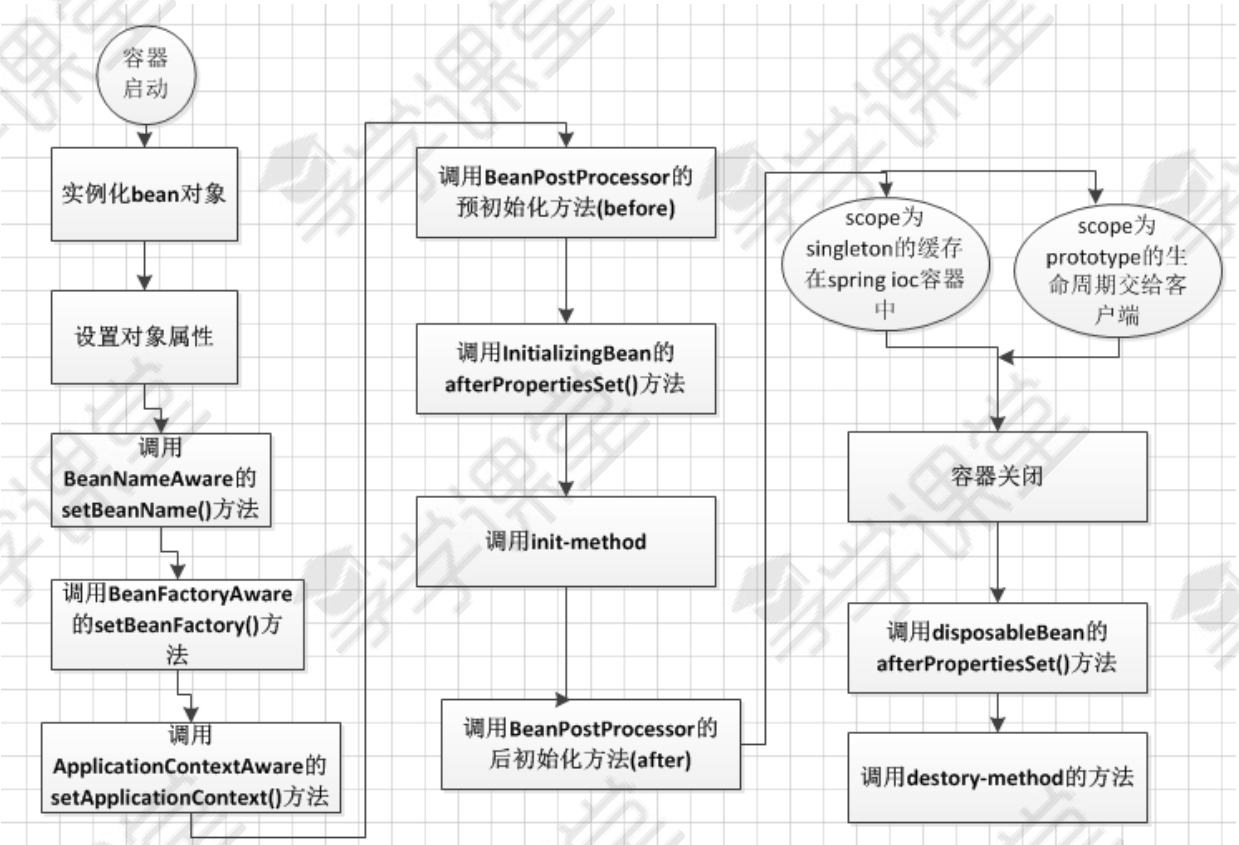
答：

在 Spring 框架中，一旦把一个 bean 纳入到 Spring IoC 容器之中，这个 bean 的生命周期就会交由容器进行管理，一般担当管理者角色的是 BeanFactory 或 ApplicationContext。认识一下 Bean 的生命周期活动，对更好的利用它有很大的帮助。

概括来说主要有四个阶段：实例化，初始化，使用，销毁。

13. 请具体描述 IOC 容器对 Bean 的生命周期控制流程：

通过构造器或工厂方法创建 Bean 实例
. 为 Bean 的属性设置值和对其他 Bean 的引用
. 将 Bean 实例传递给 Bean 后置处理器的 postProcessBeforeInitialization 方法
. 调用 Bean 的初始化方法(init-method)
. 将 Bean 实例传递给 Bean 后置处理器的 postProcessAfterInitialization 方法
. Bean 可以使用了
. 当容器关闭时，调用 Bean 的销毁方法(destroy-method)



14. BeanFactory 和 ApplicationContext 有什么区别?

1)BeanFactory 是 Spring 里面最底层的接口，包含了各种 Bean 的定义，读取 bean 配置文档，管理 bean 的加载、实例化，控制 bean 的生命周期，维护 bean 之间的依赖关系。ApplicationContext 接口作为 BeanFactory 的派生，除了提供 BeanFactory 所具有的功能外，还提供了更完整的框架功能。

2)BeanFactroy 采用的是延迟加载形式来注入 Bean 的,即只有在使用到某个 Bean 时(调用 getBean())，才对该 Bean 进行加载实例化。ApplicationContext 是在容器启动时，一次性创建了所有的 Bean。这样在容器启动时，我们就可以发现 Spring 中存在的配置错误，这样有利于检查所依赖属性是否注。

3)BeanFactory 通常以编程的方式被创建，ApplicationContext 还能以声明的方式创建，如使用 ContextLoader。

4)BeanFactory 和 ApplicationContext 都支持 BeanPostProcessor 、BeanFactoryPostProcessor 的使用，但两者之间的区别是：BeanFactory 需要手动注册，而 ApplicationContext 则是自动注册。

15. 谈谈 Spring Bean 创建过程中的设计模式?

策略模式、代理模式、适配器模式

16. Spring MVC 运行流程

答:

第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求 HandlerMapping 查找 Handler（ 可以根据 xml 配置、注解进行查找）

第三步：处理器映射器 HandlerMapping 向前端控制器返回 Handler

第四步：前端控制器调用处理器适配器去执行 Handler

第五步：处理器适配器去执行 Handler

第六步：Handler 执行完成给适配器返回 ModelAndView

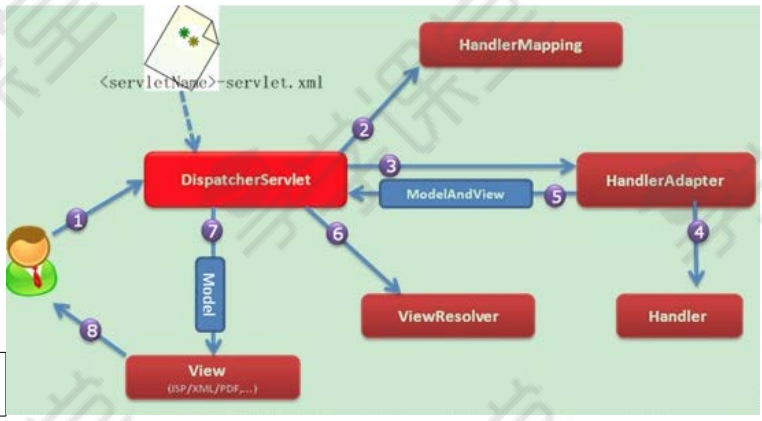
第七步：处理器适配器向前端控制器返回 ModelAndView（ModelAndView 是 springmvc 框架的一个底层对象，包括 Model 和 view）

第八步：前端控制器请求视图解析器去进行视图解析（根据逻辑视图名解析成真正的视图(jsp)）

第九步：视图解析器向前端控制器返回 View

第十步：前端控制器进行视图渲染（ 视图渲染将模型数据(在 ModelAndView 对象中)填充到 request 域）

第十一步：前端控制器向用户响应结果



17. Spring 框架中的单例 bean 是线程安全的吗?

一般的 Web 应用划分为展现层、服务层和持久层三个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程

ThreadLocal 是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal 比直接使用 synchronized 同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

线程安全问题都是由全局变量及静态变量引起的。

无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象 .不能保存数据，是不变类，是线程安全的。

当然也可以通过加 sync 锁的方法来解决线程安全，这种以时间换空间的场景在高并发场景下显然是不实际的。

18. 解释 Spring 支持的几种 bean 的作用域

Spring 框架支持以下五种 bean 的作用域：

singleton:bean 在每个 Springioc 容器中只有一个实例。

prototype：一个 bean 的定义可以有多个实例。

request：每次 http 请求都会创建一个 bean，该作用域仅在基于 web 的 SpringApplicationContext 情形下有效。

session：在一个 HTTPSession 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 SpringApplicationContext 情形下有效。

global-session：在一个全局的 HTTPSession 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 SpringApplicationContext 情形下有效。

缺省的 Springbean 的作用域是 Singleton.

18.解释 Spring 框架中 bean 的生命周期

Spring 容器从 XML 文件中读取 bean 的定义，并实例化 bean。

Spring 根据 bean 的定义填充所有的属性。

如果 bean 实现了 BeanNameAware 接口，Spring 传递 bean 的 ID 到 setBeanName 方法。

如果 Bean 实现了 BeanFactoryAware 接口，Spring 传递 beanfactory 给 setBeanFactory 方法。

如果有任何与 bean 相关联的 BeanPostProcessors，Spring 会在 postProcesserBeforeInitialization()方法内调用它们。

如果 bean 实现 IntializingBean 了，调用它的 afterPropertySet 方法，

如果 bean 声明了初始化方法，调用此初始化方法。

如果有 BeanPostProcessors 和 bean 关联，这些 bean 的 postProcessAfterInitialization()方法将被调用。

如果 bean 实现了 DisposableBean，它将调用 destroy()方法。

19.哪些是重要的 bean 生命周期方法？你能重载它们吗？

有两个重要的 bean 生命周期方法，第一个是 setup，它是在容器加载 bean 的

时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。Thebean 标签有两个重要的属性（init-method 和 destroy-method）。用它

们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct

和@PreDestroy）。

SpringBoot 面试题目

1.什么是 Spring Boot？

多年来，随着新功能的增加，spring 变得越来越复杂。只需访问 https://spring.io/projects 页面，我们就会看到可以在我们的应用程序中使用的所有 Spring 项目的不同功能。如果必须启动一个新的 Spring 项目，我们必须添加构建路径或添加 Maven 依赖关系，配置应用程序服务器，添加 spring 配置。因此，开始一个新的 spring 项目需要很多努力，因为我们现在必须从头开始做所有事情。

Spring Boot 是解决这个问题的方法。Spring Boot 已经建立在现有 spring 框架之上。使用 spring 启动，我们避免了之前我们必须做的所有样板代码和配置。因此，Spring Boot 可以帮助我们以最少的工作量，更加健壮地使用现有的 Spring 功能。

2.Spring Boot 有哪些优点？

Spring Boot 的优点有：

减少开发，测试时间和努力。

使用 JavaConfig 有助于避免使用 XML。

避免大量的 Maven 导入和各种版本冲突。

提供意见发展方法。

通过提供默认值快速开始开发。

没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat，Glassfish 或其他任何东西。

需要更少的配置 因为没有 web.xml 文件。只需添加用@Configuration 注释的类，然后添加用@Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将@Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。

基于环境的配置使用这些属性，您可以将您正在使用的环境传递到应用程序：-Dspring.profiles.active=
{enviornment}。在加载主应用程序属性文件后，Spring 将在（application{environment} .properties）中加载后续的应用程序属性文件。

3.什么是 JavaConfig？

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯 Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。

减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将 JavaConfig 与 XML 混合匹配是理想的。类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring 容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

4. 如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式 tomcat 将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

org.springframework.boot spring-boot-devtools true

5. Spring Boot 中的监视器是什么？

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的 REST 端点来检查状态。

6. 如何在 Spring Boot 中禁用 Actuator 端点安全性？

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。 我们可以使用 management.security.enabled = false 来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

7. 如何在自定义端口上运行 Spring Boot 应用程序？

为了在自定义端口上运行 Spring Boot 应用程序，您可以在 application.properties 中指定端口。

server.port = 8090

8. 什么是 YAML？

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。

与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

9. 如何实现 Spring Boot 应用程序的安全性？

为了实现 Spring Boot 的安全性，我们使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展 WebSecurityConfigurerAdapter 并覆盖其方法。

10. 如何集成 Spring Boot 和 ActiveMQ？

对于集成 Spring Boot 和 ActiveMQ，我们使用 spring-boot-starter-activemq 依赖关系。 它只需要很少的配置，并且不需要样板代码。

11. 如何使用 Spring Boot 实现分页和排序？

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的 org.springframework.data.domain.Pageable 传递给存储库方法。

12. 什么是 Swagger？你用 Spring Boot 实现了它吗？

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。

13. 什么是 Spring Profiles？

Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION 中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

14. 什么是 Spring Batch？

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

15. 什么是 FreeMarker 模板？

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。最后使用 freemarker 可以将这些结合起来，给出最终的输出页面。

16. 如何使用 Spring Boot 实现异常处理？

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

17. 您使用了哪些 starter maven 依赖项？

使用了下面的一些依赖项 spring-boot-starter-activemq
spring-boot-starter-security
spring-boot-starter-web
这有助于增加更少的依赖关系，并减少版本的冲突。

18. 什么是 CSRF 攻击？

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的 Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

19. 什么是 WebSockets？

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。
WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。
WebSocket 是全双工的 -客户端和服务器通信是相互独立的。
单个 TCP 连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信
Light -与 http 相比，WebSocket 消息数据交换要轻得多。

20. 什么是 AOP？

在软件开发过程中，跨越应用程序多个点的功能称为交叉问题。这些交叉问题与应用程序的主要业务逻辑不同。因此，将这些横切关注与业务逻辑分开是面向方面编程（AOP）的地方。

21. 什么是 Apache Kafka？

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

22. 我们如何监视所有 Spring Boot 微服务？

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。

SpringCloud 面试题目

1. 什么是 Spring Cloud?

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序，提供与外部系统的集成。

Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

2. 使用 Spring Cloud 有什么优势？

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。

服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。

冗余-分布式系统中的冗余问题。

负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。

性能-问题 由于各种运营开销导致的性能问题。

部署复杂性-Devops 技能的要求。

3. 服务注册和发现是什么意思？Spring Cloud 如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

4. 负载平衡的意义什么？

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

5. 什么是 Hystrix？它如何实现容错？

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务

假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.这是 hystrix 出现的地方

我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-consumer 公开的服务。

简化图如下所示

现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

6. 什么是 Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。

如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firtsPage 方法，并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会 。

7. 什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 Java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

使用功能区进行负载平衡。

获取服务实例，然后获取基本 URL。

利用 REST 模板来使用服务。 前面的代码如下

```
1. @Controller
2. public class ConsumerControllerClient {
3.
4. @Autowired
5. private LoadBalancerClient loadBalancer;
6.
7. public void getEmployee() throws RestClientException, IOException {
8.
9. ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");
10.
11. System.out.println(serviceInstance.getUri());
12.
13. String baseUrl=serviceInstance.getUri().toString();
14.
15. baseUrl=baseUrl+"/employee";
16.
17. RestTemplate restTemplate = new RestTemplate();
18. ResponseEntity<String> response=null;
```

```
19. try{
20. response=restTemplate.exchange(baseUrl,
21. HttpMethod.GET, getHeaders(),String.class);
22. }catch (Exception ex)
23. {
24. System.out.println(ex);
25. }
26. System.out.println(response.getBody());
27. }
```

之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

8.什么是 Spring Cloud Bus？我们需要它吗？

考虑以下情况：我们有多个应用程序使用 Spring Cloud Config 读取属性，而 Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 http：// localhost：8080 / refresh。同样对于 Employee Producer2 http：// localhost：8081 / refresh 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听。

Mybatis 面试题目

1. 什么是 MyBatis？

MyBatis 是一个可以自定义 SQL、存储过程和高级映射的持久层框架。

2. 讲下 MyBatis 的缓存

答:MyBatis 的缓存分为一级缓存和二级缓存,一级缓存放在 session 里面,默认就有,二级缓存放在它的命名空间里,默认是不打开的,使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache/>

3. Mybatis 是如何进行分页的？分页插件的原理是什么？

1）Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 实现分页，也可以使用

Mybatis 的分页插件。

2）分页插件的原理：实现 Mybatis 提供的接口，实现自定义插件，在插件的拦截方法内拦

截待执行的 sql，然后重写 sql。

举例：select * from student，拦截 sql 后重写为：select t.* from （select * from student）t limit 0，10

4. 简述 Mybatis 的插件运行原理，以及如何编写一个插件？

1）Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 通过动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

2）实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

5. Mybatis 动态 sql 是做什么的？都有哪些动态 sql？能简述一下动态 sql 的执行原理不？

1）Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能。

2）Mybatis 提供了 9 种动态 sql 标签：

trim|where|set|foreach|if|choose|when|otherwise|bind。

3）其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

6. #{}和${}的区别是什么？

1）#{}是预编译处理，${}是字符串替换。

2）Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；

3）Mybatis 在处理${}时，就是把${}替换成变量的值。

4）使用#{}可以有效的防止 SQL 注入，提高系统安全性。

7.为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

8.Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

1）Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

2）它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke()方法发现 a.getB()是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 a.getB().getName()方法的调用。这就是延迟加载的基本原理。

9.MyBatis 与 Hibernate 有哪些不同？

1）Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句，不过 mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将 Java 对象和 sql 语句映射生成最终执行的 sql，最后将 sql 执行的结果再映射生成 Java 对象。

2）Mybatis 学习门槛低，简单易学，程序员直接编写原生态 sql，可严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套 sql 映射文件，工作量大。

3）Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用 hibernate 开发可以节省很多代码，提高效率。但是 Hibernate 的缺点是学习门槛高，要精通门槛更高，而且怎么设计 O/R 映射，在性能和对象模型之间如何权衡，以及怎样用好 Hibernate 需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

10. MyBatis 的好处是什么？

1）MyBatis 把 sql 语句从 Java 源程序中独立出来，放在单独的 XML 文件中编写，给程序的维护带来了很大便利。

2)MyBatis 封装了底层 JDBC API 的调用细节,并能自动将结果集转换成 Java Bean 对象，大大简化了 Java 数据库编程的重复工作。

3）因为 MyBatis 需要程序员自己去编写 sql 语句，程序员可以结合数据库自身的特点灵活控制 sql 语句，因此能够实现比 Hibernate 等全自动 orm 框架更高的查询效率，能够完成复杂查询。

11. 简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系？

Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在 Xml 映射文件中，<parameterMap>标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象。<resultMap>标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。每一个<select>、<insert>、<update>、<delete>标签均会被解析为 MappedStatement 对象，标签内的 sql 会被解析为 BoundSql 对象。

12. 什么是 MyBatis 的接口绑定,有什么好处？

接口映射就是在 MyBatis 中任意定义接口,然后把接口里面的方法和 SQL 语句绑定,我们直接调用接口方法就可以,这样比起原来了 SqlSession 提供的方法我们可以有更加灵活的选择和设置.

13. 接口绑定有几种实现方式,分别是怎么实现的?

接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上 @Select@Update 等注解里面包含 Sql 语句来绑定,另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名.

14. 什么情况下用注解绑定,什么情况下用 xml 绑定?

当 Sql 语句比较简单时候,用注解绑定;当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多

15. MyBatis 实现一对一有几种方式?具体怎么操作的?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成;嵌套查询是先查一个表,根据这个表里面的结果的外键 id,去再另外一个表里面查询数据,也是通过 association 配置,但另外一个表的查询通过 select 属性配置。

16. Mybatis 能执行一对一、一对多的关联查询吗?都有哪些实现方式,以及它们之间的区别?

能,Mybatis 不仅可以执行一对一、一对多的关联查询,还可以执行多对一,多对多的关联查询,多对一查询,其实就是一对一查询,只需要把 selectOne()修改为 selectList()即可;多对多查询,其实就是一对多查询,只需要把 selectOne()修改为 selectList()即可。

关联对象查询,有两种实现方式,一种是单独发送一个 sql 去查询关联对象,赋给主对象,然后返回主对象。另一种是使用嵌套查询,嵌套查询的含义为使用 join 查询,一部分列是 A 对象的属性值,另外一部分列是关联对象 B 的属性值,好处是只发一个 sql 查询,就可以把主对象和其关联对象查出来。

17. MyBatis 里面的动态 Sql 是怎么设定的?用什么语法?

MyBatis 里面的动态 Sql 一般是通过 if 节点来实现,通过 OGNL 语法来实现,但是如果要写的完整,必须配合 where,trim 节点,where 节点是判断包含节点有内容就插入 where,否则不插入,trim 节点是用来判断如果动态语句是以 and 或 or 开始,那么会自动把这个 and 或者 or 取掉。

18. Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用<resultMap>标签，逐一定义列名和对象属性名之间的映射关系。

第二种是使用 sql 列的别名功能，将列别名书写为对象属性名，比如 T_NAME AS NAME，对象属性名一般是 name，小写，但是列名不区分大小写，Mybatis 会忽略列名大小写，智能找到与之对应对象属性名，你甚至可以写成 T_NAME AS NaMe，Mybatis 一样可以正常工作。 有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

19. Xml 映射文件中，除了常见的 select|insert|updae|delete 标签之外，还有哪些标签？

还有很多其他的标签，<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态 sql 的 9 个标签，trim|where|set|foreach|if|choose|when|otherwise|bind 等，其中<sql>为 sql 片段标签，通过<include>标签引入 sql 片段，<selectKey>为不支持自增的主键生成策略标签。

20. 当实体类中的属性名和表中的字段名不一样，如果将查询的结果封装到指定 pojo？

1）通过在查询的 sql 语句中定义字段名的别名。

2）通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。

21. 模糊查询 like 语句该怎么写

1）在 Java 中拼接通配符，通过#{}赋值

2）在 Sql 语句中拼接通配符 （不安全 会引起 Sql 注入）

22. 通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应，Dao 的工作原理，是否可以重载？

不能重载，因为通过 Dao 寻找 Xml 对应的 sql 的时候全限名+方法名的保存和寻找策略。接口工作原理为 jdk 动态代理原理，运行时会为 dao 生成 proxy，代理对象会拦截接口方法，去执行对应的 sql 返回数据。

23. Mybatis 映射文件中，如果 A 标签通过 include 引用了 B 标签的内容，请问，B 标签能否定义在 A 标签的后面，还是说必须定义在 A 标签的前面？

虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的，但是，被引用的 B 标签依然可以定义在任何地方，Mybatis 都可以正确识别。原理是，Mybatis 解析 A 标签，发现 A 标签引用了 B 标签，但是 B 标签尚未解析到，尚不存在，此时，Mybatis 会将 A 标签标记为未解析状态，然后继续解析余下的标签，包含 B 标签，待所有标签解析完毕，Mybatis 会重新解析那些被标记为未解析的标签，此时再解析 A 标签时，B 标签已经存在，A 标签也就可以正常解析完成了。

24. Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复；毕竟 namespace 不是必须的，只是最佳实践而已。原因就是 namespace+id 是作为 Map<String,MappedStatement>的 key 使用的，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

25. Mybatis 中如何执行批处理？

使用 BatchExecutor 完成批处理。

26. Mybatis 都有哪些 Executor 执行器？它们之间的区别是什么？

Mybatis 有三种基本的 Executor 执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。1)SimpleExecutor:每执行一次 update 或 select,就开启一个 Statement 对象，用完立刻关闭 Statement 对象。2)ReuseExecutor:执行 update 或 select，以 sql 作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完后，不关闭 Statement 对象，而是放置于 Map3）BatchExecutor：完成批处理。

27. Mybatis 中如何指定使用哪一种 Executor 执行器？

在 Mybatis 配置文件中，可以指定默认的 ExecutorType 执行器类型，也可以手动给 DefaultSqlSessionFactory 的创建 SqlSession 的方法传递 ExecutorType 类型参数。

28. Mybatis 执行批量插入，能返回数据库主键列表吗？

能，JDBC 都能，Mybatis 当然也能。

29. Mybatis 是否可以映射 Enum 枚举类？

Mybatis 可以映射枚举类，不单可以映射枚举类，Mybatis 可以映射任何对象到表的一列上。映射方式为自定义一个 TypeHandler，实现 TypeHandler 的 setParameter()和 getResult()接口方法。TypeHandler 有两个作用，一是完成从 JavaType 至 jdbcType 的转换，二是完成 jdbcType 至 JavaType 的转换，体现为 setParameter()和 getResult()两个方法，分别代表设置 sql 问号占位符参数和获取列查询结果。

30. 如何获取自动生成的(主)键值？

配置文件设置 usegeneratedkeys 为 true

31. 在 mapper 中如何传递多个参数？

1）直接在方法中传递参数，xml 文件用#{0} #{1}来获取
2）使用 @param 注解:这样可以直接在 xml 文件中通过#{name}来获取

32. 32、resultType resultMap 的区别？

1）类的名字和数据库相同时，可以直接设置 resultType 参数为 Pojo 类
2）若不同，需要设置 resultMap 将结果名字和 Pojo 名字进行转换

33. 使用 MyBatis 的 mapper 接口调用时有哪些要求？

1）Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同
2）Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的
类型相同
3)Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同
Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

34. Mybatis 比 IBatis 比较大的几个改进是什么？

1）有接口绑定,包括注解绑定 sql 和 xml 绑定 Sql
2）动态 sql 由原来的节点配置变成 OGNL 表达式 3）
在一对一,一对多的时候引进了 association,在一对多的时候引入了 collection 节点,不过都是在 resultMap 里面配置

35. IBatis 和 MyBatis 在核心处理类分别叫什么？

IBatis 里面的核心处理类交 SqlMapClient,MyBatis 里面的核心处理类叫做 SqlSession。

36. IBatis 和 MyBatis 在细节上的不同有哪些？

1）在 sql 里面变量命名有原来的#变量# 变成了#{变量}

2）原来的$变量$变成了${变量}

3）原来在 sql 节点里面的 class 都换名字交 type

4）原来的 queryForObject queryForList 变成了 selectOne selectList5）原来的别名设置在映射文件里面放在了核心配置文件里

Mysql 面试题目

1. 数据库三范式是什么？

1. 第一范式（1NF）：字段具有原子性,不可再分。(所有关系型数据库系统都满足第一范式数据库表中的字段都是单一属性的，不可再分)

2. 第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。要求数据库表中的每个实例或行必须可以被惟一地区分。通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键。

3. 满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。>所以第三范式具有如下特征： >>1. 每一列只有一个值 >>2. 每一行都能区分。 >>3. 每一个表都不包含其他表已经包含的非主关键字信息。

2.有哪些数据库优化方面的经验?

1. 用 PreparedStatement， 一般来说比 Statement 性能高：一个 sql 发给服务器去执行，涉及步骤：语法检查、语义分析， 编译，缓存。

2. 有外键约束会影响插入和删除性能，如果程序能够保证数据的完整性，那在设计数据库时就去掉外键。

3. 表中允许适当冗余，譬如，主题帖的回复数量和最后回复时间等

4. UNION ALL 要比 UNION 快很多，所以，如果可以确认合并的两个结果集中不包含重复数据且不需要排序时的话，那么就使用 UNION ALL。 >>UNION 和 UNION ALL 关键字都是将两个结果集合并为一个，但这两者从使用和效率上来说都有所不同。 >1. 对重复结果的处理：UNION 在进行表链接后会筛选掉重复的记录，Union All 不会去除重复记录。 >2. 对排序的处理：Union 将会按照字段的顺序进行排序；UNION ALL 只是简单的将两个结果合并后就返回。

3.请简述常用的索引有哪些种类?

1. 普通索引：即针对数据库表创建索引

2. 唯一索引：与普通索引类似，不同的就是：MySQL 数据库索引列的值必须唯一，但允许有空值

3. 主键索引：它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引

4. 组合索引：为了进一步榨取 MySQL 的效率，就要考虑建立组合索引。即将数据库表中的多个字段联合起来作为一个组合索引。

4.以及在 mysql 数据库中索引的工作机制是什么？

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树

5.MySQL 的基础操作命令:

1. MySQL 是否处于运行状态:Debian 上运行命令 service mysql status，在 RedHat 上运行命令 service mysqld status

2. 开启或停止 MySQL 服务 :运行命令 service mysqld start 开启服务；运行命令 service mysqld stop 停止服务

3. Shell 登入 MySQL：运行命令 mysql -u root -p

4. 列出所有数据库:运行命令 show databases;

5. 切换到某个数据库并在上面工作:运行命令 use databasename; 进入名为 databasename 的数据库

6. 列出某个数据库内所有表: show tables;

7. 获取表内所有 Field 对象的名称和类型 :describe table_name;

6. mysql 的复制原理以及流程。

Mysql 内建的复制功能是构建大型，高性能应用程序的基础。将 Mysql 的数据分布到多个系统上去，这种分布的机制，是通过将 Mysql 的某一台主机的数据复制到其它主机（slaves）上，并重新执行一遍来实现的。 * 复制过程中一个服务器充当主服务器，而一个或多个其它服务器充当从服务器。主服务器将更新写入二进制日志文件，并维护文件的一个索引以跟踪日志循环。这些日志可以记录发送到从服务器的更新。 当一个从服务器连接主服务器时，它通知主服务器在日志中读取的最后一次成功更新的位置。从服务器接收从那时起发生的任何更新，然后封锁并等待主服务器通知新的更新。

过程如下

1. 主服务器把更新记录到二进制日志文件中。
2. 从服务器把主服务器的二进制日志拷贝到自己的中继日志（replay log）中。
3. 从服务器重做中继日志中的时间，把更新应用到自己的数据库上。

7. mysql 支持的复制类型?

1. 基于语句的复制： 在主服务器上执行的 SQL 语句，在从服务器上执行同样的语句。MySQL 默认采用基于语句的复制，效率比较高。 一旦发现没法精确复制时，会自动选着基于行的复制。

2. 基于行的复制：把改变的内容复制过去，而不是把命令在从服务器上执行一遍. 从 mysql5.0 开始支持

3. 混合类型的复制: 默认采用基于语句的复制，一旦发现基于语句的无法精确的复制时，就会采用基于行的复制。

8. mysql 中 myisam 与 innodb 的区别?

1. 事务支持 >*MyISAM:强调的是性能,每次查询具有原子性,其执行数度比 InnoDB 类型更快，但是不提供事务支持。* >InnoDB：提供事务支持事务，外部键等高级数据库功能。具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

2. InnoDB 支持行级锁，而 MyISAM 支持表级锁. >> 用户在操作 myisam 表时，select，update，delete，insert 语句都会给表自动加锁，如果加锁以后的表满足 insert 并发的情况下，可以在表的尾部插入新的数据。

3. InnoDB 支持 MVCC, 而 MyISAM 不支持

4. InnoDB 支持外键，而 MyISAM 不支持

5. 表主键 >*MyISAM：允许没有任何索引和主键的表存在，索引都是保存行的地址。* > InnoDB：如果没有设定主键或者非空唯一索引，就会自动生成一个 6 字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

6. InnoDB 不支持全文索引，而 MyISAM 支持。

7. 可移植性、备份及恢复 > *MyISAM：数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。* > InnoDB：免费的方案可以是拷贝数据文件、备份

binlog，或者用 mysqldump，在数据量达到几十 G 的时候就相对痛苦了

8. 存储结构 >*MyISAM：每个 MyISAM 在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm 文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。* > InnoDB：所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB。

9. mysql 中 varchar 与 char 的区别以及 varchar(50)中的 50 代表的涵义？

1. varchar 与 char 的区别: char 是一种固定长度的类型，varchar 则是一种可变长度的类型.

2. varchar(50)中 50 的涵义 : 最多存放 50 个字节

3. int（20）中 20 的涵义: int(M)中的 M indicates the maximum display width (最大显示宽度)for integer types. The maximum legal display width is 255.

10. MySQL 中 InnoDB 支持的四种事务隔离级别名称，以及逐级之间的区别？

1. Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

2. Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是 MySQL 默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的 commit，所以同一 select 可能返回不同结果。

3. Repeatable Read（可重读）

这是 MySQL 的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（PhantomRead）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB 和 Falcon 存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control 间隙锁）机制解决了该问题。注：其实多版本只是解决不可重复读问题，而加上间隙锁（也就是它这里所谓的并发控制）才解决了幻读问题。

Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

11. 表中有大字段 X（例如：text 类型），且字段 X 不会经常更新，以读为主，将该字段拆成子表好处是什么？

如果字段里面有大字段（text,blob)类型的，而且这些字段的访问并不多，这时候放在一起就变成缺点了。MYSQL 数据库的记录存储是按行存储的,数据块大小又是固定的(16K)，每条记录越小，相同的块存储的记录就越多。此时应该把大字段拆走，这样应付大部分小字段的查询时，就能提高效率。当需要查询大字段时，此时的关联查询是不可避免的，但也是值得的。拆分开后，对字段的 UPDAE 就要 UPDATE 多个表了

12. MySQL 中 InnoDB 引擎的行锁是通过加在什么上完成（或称实现）的？

InnoDB 行锁是通过给索引上的索引项加锁来实现的，这一点 MySQL 与 Oracle 不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB 这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁！

13. 若一张表中只有一个字段 VARCHAR(N)类型，utf8 编码，则 N 最大值为多少(精确到数量级即可)？

由于 utf8 的每个字符最多占用 3 个字节。而 MySQL 定义行的长度不能超过 65535，因此 N 的最大值计算方法为：(65535-1-2)/3。减去 1 的原因是实际存储从第二个字节开始，减去 2 的原因是因为要在列表长度存储实际的字符长度，除以 3 是因为 utf8 限制：每个字符最多占用 3 个字节。

14. [SELECT *] 和[SELECT 全部字段]的 2 种写法有何优缺点?

1. 前者要解析数据字典，后者不需要
2. 结果输出顺序，前者与建表列顺序相同，后者按指定字段顺序。
3. 表字段改名，前者不需要修改，后者需要改
4. 后者可以建立索引进行优化，前者无法优化
5. 后者的可读性比前者要高

15. HAVNG 子句 和 WHERE 的异同点?

1. 语法上：where 用表中列名，having 用 select 结果别名
2. 影响结果范围：where 从表读出数据的行数，having 返回客户端的行数
3. 索引：where 可以使用索引，having 不能使用索引，只能在临时结果集操作
4. where 后面不能使用聚集函数，having 是专门使用聚集函数的。

16. MySQL 当记录不存在时 insert,当记录存在时 update，语句怎么写?

```
INSERT INTO table (a,b,c) VALUES (1,2,3) ON DUPLICATE KEY
UPDATE c=c+1;
```

17. 一张表,里面有 ID 自增主键,当 insert 了 17 条记录之后，删除了第 15,16,17 条记录，再把 Mysql 重启，再 insert 一条记录，这条记录的 ID 是 18 还是 15 ？

(1)如果表的类型是 MyISAM，那么是 18

因为 MyISAM 表会把自增主键的最大 ID 记录到数据文件里，重启 MySQL 自增主键的最大 ID 也不会丢失

（2）如果表的类型是 InnoDB，那么是 15

InnoDB 表只是把自增主键的最大 ID 记录到内存中，所以重启数据库或者是对表进行 OPTIMIZE 操作，都会导致最大 ID 丢失

18. Mysql 的技术特点是什么？

Mysql 数据库软件是一个客户端或服务器系统，其中包括：支持各种客户端程序和库的多线程 SQL 服务器、不同的后端、广泛的应用程序编程接口和管理工具。

19. Heap 表是什么？

HEAP 表存在于内存中，用于临时高速存储。BLOB 或 TEXT 字段是不允许的,只能使用比较运算符=，<，>，=>，=< HEAP 表不支持 AUTO_INCREMENT 索引不可为 NULL

20. Mysql 服务器默认端口是什么？

Mysql 服务器的默认端口是 3306。

21. 与 Oracle 相比，Mysql 有什么优势？

Mysql 是开源软件，随时可用，无需付费。
Mysql 是便携式的
带有命令提示符的 GUI。
使用 Mysql 查询浏览器支持管理

22. 如何区分 FLOAT 和 DOUBLE？

以下是 FLOAT 和 DOUBLE 的区别：
浮点数以 8 位精度存储在 FLOAT 中，并且有四个字节。
浮点数存储在 DOUBLE 中，精度为 18 位，有八个字节。

23. 区分 CHAR_LENGTH 和 LENGTH？

CHAR_LENGTH 是字符数，而 LENGTH 是字节数。Latin 字符的这两个数据是相同的，但是对于 Unicode 和其他编码，它们是不同的。

24. 请简洁描述 Mysql 中 InnoDB 支持的四种事务隔离级别名称，以及逐级之间的区别？

SQL 标准定义的四个隔离级别为：
read uncommited ：读到未提交数据
read committed：脏读，不可重复读
repeatable read：可重读
serializable ：串行事物

25. 在 Mysql 中 ENUM 的用法是什么？

ENUM 是一个字符串对象，用于指定一组预定义的值，并可在创建表时使用。Create table size(name ENUM('Smail,'Medium','Large');

26. 如何定义 REGEXP？

REGEXP 是模式匹配，其中匹配模式在搜索值的任何位置。

27. CHAR 和 VARCHAR 的区别？

以下是 CHAR 和 VARCHAR 的区别：
CHAR 和 VARCHAR 类型在存储和检索方面有所不同
CHAR 列长度固定为创建表时声明的长度，长度值范围是 1 到 255
当 CHAR 值被存储时，它们被用空格填充到特定长度，检索 CHAR 值时需删除尾随空格。

28. 列的字符串类型可以是什么？

字符串类型是：
SET
BLOB
ENUM
CHAR
TEXT
VARCHAR

29. 如何获取当前的 Mysql 版本？

SELECT VERSION();用于获取当前 Mysql 的版本。

30. Mysql 中使用什么存储引擎？

存储引擎称为表类型，数据使用各种技术存储在文件中。
技术涉及：
Storage mechanism

Locking levels
Indexing
Capabilities and functions.

31. Mysql 驱动程序是什么？

以下是 Mysql 中可用的驱动程序：
PHP 驱动程序
JDBC 驱动程序
ODBC 驱动程序
CWRAPPER
PYTHON 驱动程序
PERL 驱动程序
RUBY 驱动程序
CAP11PHP 驱动程序
Ado.net5.mxj

32. TIMESTAMP 在 UPDATE CURRENT_TIMESTAMP 数据类型上做什么？

创建表时 TIMESTAMP 列用 Zero 更新。只要表中的其他字段发生更改，UPDATE CURRENT_TIMESTAMP 修饰符就将时间戳字段更新为当前时间。

33. 主键和候选键有什么区别？

表格的每一行都由主键唯一标识,一个表只有一个主键。主键也是候选键。按照惯例，候选键可以被指定为主键，并且可以用于任何外键引用。

34. 如何使用 Unix shell 登录 Mysql？

我们可以通过以下命令登录：
[mysql dir]/bin/mysql -h hostname -u

35. myisamchk 是用来做什么的？

它用来压缩 MyISAM 表，这减少了磁盘或内存使用。

36. 如何控制 HEAP 表的最大尺寸？

Heal 表的大小可通过称为 max_heap_table_size 的 Mysql 配置变量来控制。

37. MyISAM Static 和 MyISAM Dynamic 有什么区别？

在 MyISAM Static 上的所有字段有固定宽度。动态 MyISAM 表将具有像 TEXT，BLOB 等字段，以适应不同长度的数据类型。MyISAM Static 在受损情况下更容易恢复。

38. federated 表是什么？

federated 表，允许访问位于其他服务器数据库上的表。

39. 如果一个表有一列定义为 TIMESTAMP，将发生什么？

每当行被更改时，时间戳字段将获取当前时间戳。

40. 列设置为 AUTO INCREMENT 时，如果在表中达到最大值，会发生什么情况？

它会停止递增，任何进一步的插入都将产生错误，因为密钥已被使用。

41. 怎样才能找出最后一次插入时分配了哪个自动增量？

LAST_INSERT_ID 将返回由 Auto_increment 分配的最后一个值，并且不需要指定表名称。

42. 你怎么看到为表格定义的所有索引？

索引是通过以下方式为表格定义的：
SHOW INDEX FROM

43. LIKE 声明中的%和_是什么意思？

%对应于 0 个或更多字符，_只是 LIKE 语句中的一个字符。

44. 如何在 Unix 和 Mysql 时间戳之间进行转换？

UNIX_TIMESTAMP 是从 Mysql 时间戳转换为 Unix 时间戳的命令
FROM_UNIXTIME 是从 Unix 时间戳转换为 Mysql 时间戳的命令

45. 列对比运算符是什么？

在 SELECT 语句的列比较中使用=，<>，<=，<，>=，>，<<，>>，<=>，AND，OR 或 LIKE 运算符。

46. 我们如何得到受查询影响的行数？

行数可以通过以下代码获得：
SELECT COUNT(user_id)FROM users;

47. Mysql 查询是否区分大小写？

不区分

```
SELECT VERSION(), CURRENT_DATE;
SeLect version(), current_date;
seleCt vErSiOn(), current_DATE;
```

所有这些例子都是一样的，Mysql 不区分大小写。

48. LIKE 和 REGEXP 操作有什么区别？

LIKE 和 REGEXP 运算符用于表示^和%。

```
SELECT * FROM employee WHERE emp_name REGEXP "^b";
SELECT * FROM employee WHERE emp_name LIKE "%b";
```

49. BLOB 和 TEXT 有什么区别？

BLOB 是一个二进制对象，可以容纳可变数量的数据。有四种类型的 BLOB -

TINYBLOB

BLOB

MEDIUMBLOB 和

LONGBLOB

它们只能在所能容纳价值的最大长度上有所不同。

TEXT 是一个不区分大小写的 BLOB。四种 TEXT 类型

TINYTEXT

TEXT

MEDIUMTEXT 和

LONGTEXT

它们对应于四种 BLOB 类型，并具有相同的最大长度和存储要求。BLOB 和 TEXT 类型之间的唯一区别在于对 BLOB 值进行排序和比较时区分大小写，对 TEXT 值不区分大小写。

50. mysql_fetch_array 和 mysql_fetch_object 的区别是什么？

以下是 mysql_fetch_array 和 mysql_fetch_object 的区别：

mysql_fetch_array（） - 将结果行作为关联数组或来自数据库的常规数组返回。

mysql_fetch_object - 从数据库返回结果行作为对象。

51. 我们如何在 mysql 中运行批处理模式？

以下命令用于在批处理模式下运行：
mysql;
mysql mysql.out

52. MyISAM 表格将在哪里存储，并且还提供其存储格式？

每个 MyISAM 表格以三种格式存储在磁盘上：
• “.frm”文件存储表定义
• 数据文件具有“.MYD”（MYData）扩展名
索引文件具有“.MYI”（MYIndex）扩展名

53. Mysql 中有哪些不同的表格？

共有 5 种类型的表格：
MyISAM
Heap
Merge
INNODB
ISAM
MyISAM 是 Mysql 的默认存储引擎。

54. ISAM 是什么？

ISAM 简称为索引顺序访问方法。它是由 IBM 开发的，用于在磁带等辅助存储系统上存储和检索数据。

55. InnoDB 是什么？

lnnoDB 是一个由 Oracle 公司开发的 Innobase Oy 事务安全存储引擎。

56. Mysql 如何优化 DISTINCT？

DISTINCT 在所有列上转换为 GROUP BY，并与 ORDER BY 子句结合使用。

```
1
SELECT DISTINCT t1.a FROM t1,t2 where t1.a=t2.a;
```

57. 如何输入字符为十六进制数字？

如果想输入字符为十六进制数字，可以输入带有单引号的十六进制数字和前缀（X），或者只用（0x）前缀输入十六进制数字。

如果表达式上下文是字符串，则十六进制数字串将自动转换为字符串。

58. 如何显示前 50 行？

在 Mysql 中，使用以下代码查询显示前 50 行：

```
SELECT*FROM
LIMIT 0,50;
```

59. 可以使用多少列创建索引？

任何标准表最多可以创建 16 个索引列。

60. NOW（）和 CURRENT_DATE（）有什么区别？

NOW（）命令用于显示当前年份，月份，日期，小时，分钟和秒。

CURRENT_DATE（）仅显示当前年份，月份和日期。

61. 什么样的对象可以使用 CREATE 语句创建？

以下对象是使用 CREATE 语句创建的：

```
DATABASE
EVENT
FUNCTION
INDEXPROCEDURE
TABLE
```

TRIGGER
USER
VIEW

62. Mysql 表中允许有多少个 TRIGGERS？

在 Mysql 表中允许有六个触发器，如下：
BEFORE INSERT
AFTER INSERT
BEFORE UPDATE
AFTER UPDATE
BEFORE DELETE
AFTER DELETE

63. 什么是非标准字符串类型？

以下是非标准字符串类型：
TINYTEXT
TEXT
MEDIUMTEXT
LONGTEXT

64. 什么是通用 SQL 函数？

CONCAT(A, B) - 连接两个字符串值以创建单个字符串输出。通常用于将两个或多个字段合并为一个字段。
FORMAT(X, D)- 格式化数字 X 到 D 有效数字。
CURRDATE(), CURRTIME()- 返回当前日期或时间。
NOW（） - 将当前日期和时间作为一个值返回。
MONTH（），DAY（），YEAR（），WEEK（），WEEKDAY（） - 从日期值中提取给定数据。
HOUR（），MINUTE（），SECOND（） - 从时间值中提取给定数据。
DATEDIFF（A，B） - 确定两个日期之间的差异，通常用于计算年龄
SUBTIMES（A，B） - 确定两次之间的差异。
FROMDAYS（INT） - 将整数天数转换为日期值。

65. 解释访问控制列表

ACL（访问控制列表）是与对象关联的权限列表。这个列表是 Mysql 服务器安全模型的基础，它有助于排除用户无法连接的问题。

Mysql 将 ACL(也称为授权表)缓存在内存中。当用户尝试认证或运行命令时，Mysql 会按照预定的顺序检查 ACL 的认证信息和权限。

66. Mysql 支持事务吗？

在缺省模式下，MYSQL 是 autocommit 模式的，所有的数据库更新操作都会即时提交，所以在缺省情况下，mysql 是不支持事务的。

但是如果你的 MYSQL 表类型是使用 InnoDB Tables 或 BDB tables 的话，你的 MYSQL 就可以使用事务处理,使用 SET AUTOCOMMIT=0 就可以使 MYSQL 允许在非 autocommit 模式，在非 autocommit 模式下，你必须使用 COMMIT 来提交你的更改，或者用 ROLLBACK 来回滚你的更改。

示例如下：

一

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summmary=@A WHERE type=1;
COMMIT;
```

67. Mysql 里记录货币用什么字段类型好

NUMERIC 和 DECIMAL 类型被 Mysql 实现为同样的类型，这在 SQL92 标准允许。他们被用于保存值，该值的准确精度是极其重要的值，例如与金钱有关的数据。当声明一个类是这些类型之一时，精度和规模的能被(并且通常是)指定；

例如：

```
salary DECIMAL(9,2)
```

在这个例子中，9(precision)代表将被用于存储值的总的小数位数，而 2(scale)代表将被用于存储小数点后的位数。

因此，在这种情况下，能被存储在 salary 列中的值的范围是从-9999999.99 到 9999999.99。

在 ANSI/ISO SQL92 中，句法 DECIMAL(p)等价于 DECIMAL(p,0)。

同样，句法 DECIMAL 等价于 DECIMAL(p,0)，这里实现被允许决定值 p。Mysql 当前不支持 DECIMAL/NUMERIC 数据类型的这些变种形式的任一种。

这一般说来不是一个严重的问题,因为这些类型的主要益处得自于明显地控制精度和规模的能力。DECIMAL 和 NUMERIC 值作为字符串存储，而不是作为二进制浮点数，以便保存那些值的小数精度。

一个字符用于值的每一位、小数点(如果 scale>0)和“-”符号(对于负值)。如果 scale 是 0，DECIMAL 和 NUMERIC 值不包含小数点或小数部分。DECIMAL 和 NUMERIC 值得最大的范围与 DOUBLE 一样，但是对于一个给定的 DECIMAL 或

NUMERIC 列，实际的范围可由制由给定列的 precision 或 scale 限制。

当这样的列赋给了小数点后面的位超过指定 scale 所允许的位的值，该值根据 scale 四舍五入。

当一个 DECIMAL 或 NUMERIC 列被赋给了其大小超过指定(或缺省的）precision 和 scale 隐含的范围的值，Mysql 存储表示那个范围的相应的端点值。

我希望本文可以帮助你提升技术水平。那些，感觉学的好难，甚至会令你沮丧的人，别担

心，我认为，如果你愿意试一试本文介绍的几点，会向前迈进，克服这种感觉。这些要点

也许对你不适用，但你会明确一个重要的道理：接受自己觉得受困这个事实是摆脱这个困

境的第一步。

68. Mysql 数据表在什么情况下容易损坏？

服务器突然断电导致数据文件损坏。

强制关机，没有先关闭 mysql 服务等。

69. Mysql 有关权限的表都有哪几个？

Mysql 服务器通过权限表来控制用户对数据库的访问，权限表存放在 mysql 数据库里，由 mysql_install_db 脚本初始化。这些权限表分别 user，db，table_priv，columns_priv 和 host。

70. Mysql 中有哪几种锁？

MyISAM 支持表锁，InnoDB 支持表锁和行锁，默认为行锁

表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低

行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高

Dubbo 面试题目

1. Dubbo 支持哪些协议，每种协议的应用场景，优缺点？

dubbo： 单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；

rmi： 采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 Java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，Java 序列化存在安全漏洞；

webservice： 基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；

http： 基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；

hessian： 集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

memcache： 基于 memcached 实现的 RPC 协议

redis： 基于 redis 实现的 RPC 协议

2. Dubbo 超时时间怎样设置？

Dubbo 超时时间设置有两种方式：

服务提供者端设置超时时间，在 Dubbo 的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。

服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告。

3. Dubbo 有些哪些注册中心？

Multicast 注册中心： Multicast 注册中心不需要任何中心节点，只要广播地址，就能进行服务注册和发现。基于网络中组播传输实现；

Zookeeper 注册中心： 基于分布式协调系统 Zookeeper 实现，采用 Zookeeper 的 watch 机制实现数据变更；

redis 注册中心： 基于 redis 实现，采用 key/Map 存储，住 key 存储服务名和类型，Map 中 key 存储服务 URL，value 服务过期时间。基于 redis 的发布/订阅模式通知数据变更；

Simple 注册中心

Dubbo 集群的负载均衡有哪些策略 Dubbo 提供了常见的集群策略实现，并预扩展点予以自行实现。

Random LoadBalance：随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀；

RoundRobin LoadBalance：轮循选取提供者策略，平均分布，但是存在请求累积的问题；

LeastActive LoadBalance：最少活跃调用策略，解决慢提供者接收更少的请求；

ConstantHash LoadBalance：一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动；

4. Dubbo 是什么？

Dubbo 是一个分布式、高性能、透明化的 RPC 服务框架，提供服务自动注册、自动发现等高效服务治理方案， 可以和 Spring 框架无缝集成。

5. Dubbo 的主要应用场景？

透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何 API 侵入。

软负载均衡及容错机制，可在内网替代 F5 等硬件负载均衡器，降低成本，减少单点。

服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的 IP 地址，并且能够平滑添加或删除服务提供者。

6. Dubbo 的核心功能？

主要就是如下 3 个核心功能：

Remoting：网络通信框架，提供对多种 NIO 框架抽象封装，包括“同步转异步”和“请求-响应”模式的信息交换方式。

Cluster：服务框架，提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。

Registry：服务注册，基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

□

7. Dubbo 服务注册与发现的流程？流程说明：

Provider(提供者)绑定指定端口并启动服务

指供者连接注册中心，并发本机 IP、端口、应用信息和提供服务信息发送至注册中心存储

Consumer(消费者），连接注册中心 ，并发送应用信息、所求服务信息至注册中心

注册中心根据 消费 者所求服务信息匹配对应的提供者列表发送至 Consumer 应用缓存。

Consumer 在发起远程调用时基于缓存的消费者列表择其一发起调用。

Provider 状态变更会实时通知注册中心、在由注册中心实时推送至 Consumer

设计的原因：

Consumer 与 Provider 解偶，双方都可以横向增减节点数。

注册中心对本身可做对等集群，可动态增减节点，并且任意一台宕掉后，将自动切换到另一台

去中心化，双方不直接依懒注册中心，即使注册中心全部宕机短时间内也不会影响服务的调用

服务提供者无状态，任意一台宕掉后，不影响使用

8. Dubbo 的架构设计？

Dubbo 框架设计一共划分了 10 个层：

服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。

配置层（Config）：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心。

服务代理层（Proxy）：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton。

服务注册层（Registry）：封装服务地址的注册与发现，以服务 URL 为中心。

集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心。

监控层（Monitor）：RPC 调用次数和调用时间监控。

远程调用层（Protocol）：封将 RPC 调用，以 Invocation 和 Result 为中心，扩展接口为 Protocol、Invoker 和 Exporter。

信息交换层（Exchange）：封装请求响应模式，同步转异步，以 Request 和 Response 为中心。

网络传输层（Transport）：抽象 mina 和 netty 为统一接口，以 Message 为中心。

9. Dubbo 的服务调用流程？Dubbo 支持哪些协议，每种协议的应用场景，优缺点？

dubbo： 单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；

rmi： 采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 Java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。 多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，Java 序列化存在安全漏洞；

webservice： 基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；

http： 基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；

hessian： 集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现,提供与 Hession 服务互操作。多个短连接,同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

memcache： 基于 memcached 实现的 RPC 协议

redis： 基于 redis 实现的 RPC 协议

10. dubbo 推荐用什么协议？

默认使用 dubbo 协议

11. Dubbo 有些哪些注册中心？

Multicast 注册中心： Multicast 注册中心不需要任何中心节点，只要广播地址，就能进行服务注册和发现。基于网络中组播传输实现；

Zookeeper 注册中心： 基于分布式协调系统 Zookeeper 实现，采用 Zookeeper 的 watch 机制实现数据变更；

redis 注册中心： 基于 redis 实现，采用 key/Map 存储，住 key 存储服务名和类型，Map 中 key 存储服务 URL，value 服务过期时间。基于 redis 的发布/订阅模式通知数据变更；

Simple 注册中心

12. Dubbo 默认采用注册中心？

采用 Zookeeper

13. 为什么需要服务治理？

过多的服务 URL 配置困难

负载均衡分配节点压力过大的情况下也需要部署集群

服务依赖混乱，启动顺序不清晰

过多服务导致性能指标分析难度较大，需要监控

14. Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么？

可以的,启动 dubbo 时,消费者会从 zookeeper 拉取注册的生产者的地址接口等数据,缓存在本地。每次调用时，按照本地存储的地址进行调用。

15. Dubbo 与 Spring 的关系？

Dubbo 采用全 Spring 配置方式，透明化接入应用，对应用没有任何 API 侵入，只需用 Spring 加载 Dubbo 的配置即可，Dubbo 基于 Spring 的 Schema 扩展进行加载。

16. Dubbo 使用的是什么通信框架?

默认使用 NIO Netty 框架

17. Dubbo 集群提供了哪些负载均衡策略?

Random LoadBalance: 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀;

RoundRobin LoadBalance: 轮循选取提供者策略，平均分布，但是存在请求累积的问题; □ LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收更少的请求;

ConstantHash LoadBalance: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动;缺省时为 Random 随机调用

18. Dubbo 的集群容错方案有哪些?

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错 。通常用于通知所有提供者更新缓存或日志等本地资源信息。

19. Dubbo 的默认集群容错方案?

Failover Cluster

20. Dubbo 支持哪些序列化方式?

默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。

21. Dubbo 超时时间怎样设置?

Dubbo 超时时间设置有两种方式:

服务提供者端设置超时时间，在 Dubbo 的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。

服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告。

22. 服务调用超时问题怎么解决？

dubbo 在调用服务不成功时，默认是会重试两次的。

23. Dubbo 在安全机制方面是如何解决？

Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo 还提供服务黑白名单，来控制服务所允许的调用方。

24. Dubbo 和 Dubbox 之间的区别？

dubbox 基于 dubbo 上做了一些扩展，如加了服务可 restful 调用，更新了开源组件等。

25. Dubbo 和 Spring Cloud 的关系？

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spirng、Spirng Boot 的优势之上，两个框架在开始目标就不一致，Dubbo 定位服务治理、Spirng Cloud 是一个生态。

26. Dubbo 和 Spring Cloud 的区别？

最大的区别：Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。

而 SpringCloud 是基于 Http 协议+Rest 接口调用远程过程的通信，相对来说，Http 请求会有更大的报文，占的带宽也会更多。但是 REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契

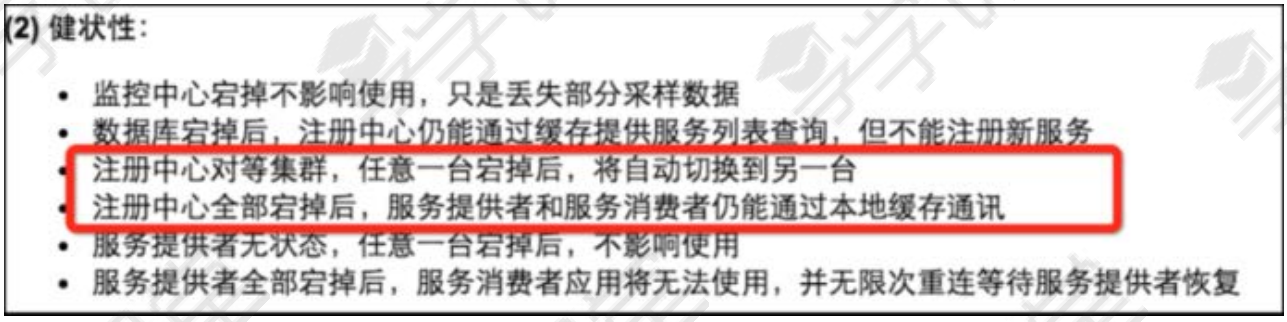
27. Dubbo 中 zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么？

可以通信的，启动 dubbo 时，消费者会从 zk 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用；

注册中心对等集群，任意一台宕机后，将会切换到另一台；注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通讯。服务提供者无状态，任一台 宕机后，不影响使用；服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复；

挂掉是不要紧的，但前提是你没有增加新的服务，如果你要调用新的服务，则是不能办到的。

附文档截图：



28. dubbo 服务负载均衡策略？

l Random LoadBalance

随机，按权重设置随机概率。在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比

较均匀，有利于动态调整提供者权重。(权重可以在 dubbo 管控台配置)

l RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。存在慢的提供者累积请求问题，比如：第二台机器很慢，但没挂，当请求调

到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

l LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。使慢的提供者收到更少请求，因为越慢的提供者的

调用前后计数差会越大。

l ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节

点，平摊到其它提供者，不会引起剧烈变动。缺省只对第一个参数 Hash

29. Dubbo 在安全机制方面是如何解决的

Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo 还提供服务黑白名单，来控制服务所允许的调用方。

30. dubbo 连接注册中心和直连的区别

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直联方式，将以服务接口为单位，忽略注册中心的提供者列表，服务注册中心，动态的注册和发现服务，使服务的位置透明，并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover， 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外，注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

31. dubbo 服务集群配置（集群容错模式）

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。可以自行扩展集群容错策略

l Failover Cluster(默认)

失败自动切换，当出现失败，重试其它服务器。(缺省)通常用于读操作，但重试会带来更长延迟。可通过 retries="2"来设置重试次数(不含第一次)。

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2"来设置最大并行数。

32. dubbo 通信协议 dubbo 协议为什么要消费者比提供者个数多：

因 dubbo 协议采用单一长连接，假设网络为千兆网卡(1024Mbit=128MByte)，根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。

33. dubbo 通信协议 dubbo 协议为什么不能传大包：

因 dubbo 协议采用单一长连接，如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡(1024Mbit=128MByte)，每条连接最大 7MByte(不同的环境可能不一样，供参考)，单个服务提供者的 TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

34. dubbo 通信协议 dubbo 协议为什么采用异步单一长连接：

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步 IO，复用线程池，防止 C10K 问题。

35. dubbo 通信协议 dubbo 协议适用范围和适用场景

适用范围：传入传出参数数据包较小（建议小于 100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。

适用场景：常规远程服务方法调用

dubbo 协议补充：

连接个数：单连接

连接方式：长连接

传输协议：TCP

传输方式：NIO 异步传输

序列化：Hessian 二进制序列化

5. **RMI 协议**

RMI 协议采用 JDK 标准的 Java.rmi.*实现，采用阻塞式短连接和 JDK 标准序列化方式，Java 标准的远程调用协议。

连接个数：多连接

连接方式：短连接

传输协议：TCP

传输方式：同步传输

序列化：Java 标准二进制序列化

适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。

适用场景：常规远程服务方法调用，与原生 RMI 服务互操作

6. **Hessian 协议**

Hessian 协议用于集成 Hessian 的服务，Hessian 底层采用 Http 通讯，采用 Servlet 暴露服务，Dubbo 缺省内嵌 Jetty 作为服务器实现基于 Hessian 的远程调用协议。

连接个数：多连接

连接方式：短连接

传输协议：HTTP

传输方式：同步传输

序列化：Hessian 二进制序列化适用范围：传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。

适用场景：页面传输，文件传输，或与原生 hessian 服务互操作

7. **http**

采用 Spring 的 HttpInvoker 实现基于 http 表单的远程调用协议。

连接个数：多连接

连接方式：短连接

传输协议：HTTP

传输方式：同步传输

序列化：表单序列化（JSON）

适用范围：传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或 URL 传入参数，暂不支持传文件。

适用场景：需同时给应用程序和浏览器 JS 使用的服务。

8. Webservice

基于 CXF 的 frontend-simple 和 transports-http 实现基于 WebService 的远程调用协议。

连接个数：多连接

连接方式：短连接

传输协议：HTTP

传输方式：同步传输

序列化：SOAP 文本序列化

适用场景：系统集成，跨语言调用。

9. Thrif

Thrift 是 Facebook 捐给 Apache 的一个 RPC 框架，当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展，在原生协议的基础上添加了一些额外的头信息，比如 service name，magic number 等

MongoDB 面试题目

1. 你说的 NoSQL 数据库是什么意思?NoSQL 与 RDBMS 直接有什么区别?为什么要使用和不使用 NoSQL 数据库?说一说 NoSQL 数据库的几个优点?

NoSQL 是非关系型数据库，NoSQL = Not Only SQL。

关系型数据库采用的结构化的数据，NoSQL 采用的是键值对的方式存储数据。

在处理非结构化/半结构化的大数据时；在水平方向上进行扩展时；随时应对动态增加的数据项时可以优

先考虑使用 NoSQL 数据库。

在考虑数据库的成熟度；支持；分析和商业智能；管理及专业性等问题时，应优先考虑关系型数据库。

2. NoSQL 数据库有哪些类型?

NoSQL 数据库的类型

例如：MongoDB, Cassandra, CouchDB, Hypertable, Redis, Riak, Neo4j, HBASE, Couchbase,MemcacheDB, RevenDB and Voldemort are the examples of NoSQL databases. 详细阅读。

3. MySQL 与 MongoDB 之间最基本的差别是什么?

MySQL 和 MongoDB 两者都是免费开源的数据库。MySQL 和 MongoDB 有许多基本差别包括数据的表示(data representation)，查询，关系，事务，schema 的设计和定义，标准化(normalization)，速度和性能。

通过比较 MySQL 和 MongoDB，实际上我们是在比较关系型和非关系型数据库，即数据存储结构不同。

详细阅读

4. 你怎么比较 MongoDB、CouchDB 及 CouchBase?

MongoDB 和 CouchDB 都是面向文档的数据库。MongoDB 和 CouchDB 都是开源 NoSQL 数据库的最典型代表。 除了都以文档形式存储外它们没有其他的共同点。MongoDB 和 CouchDB 在数据模型实现、接口、对象存储以及复制方法等方面有很多不同。

细节可以参见下面的链接：
MongDB vs CouchDB
CouchDB vs CouchBase

5. MongoDB 成为最好 NoSQL 数据库的原因是什么？

以下特点使得 MongoDB 成为最好的 NoSQL 数据库：
面向文件的
高性能
高可用性
易扩展性
丰富的查询语言

6. 32 位系统上有什么细微差别？

journaling 会激活额外的内存映射文件。这将进一步抑制 32 位版本上的数据库大小。因此，现在 journaling 在 32 位系统上默认是禁用的。

7. journal 回放在条目(entry)不完整时(比如恰巧有一个中途故障了)会遇到问题吗？

每个 journal (group)的写操作都是一致的，除非它是完整的否则在恢复过程中它不会回放。

8. 分析器在 MongoDB 中的作用是什么？

MongoDB 中包括了一个可以显示数据库中每个操作性能特点的数据库分析器。通过这个分析器你可以找到比预期慢的查询(或写操作);利用这一信息，比如，可以确定是否需要添加索引。

9. 名字空间(namespace)是什么？

MongoDB 存储 BSON 对象在丛集(collection)中。数据库名字和丛集名字以句点连结起来叫做名字空间(namespace)。

10. 如果用户移除对象的属性，该属性是否从存储层中删除？

是的，用户移除属性然后对象会重新保存(re-save())。

11. 能否使用日志特征进行安全备份?

是的。

12. 允许空值 null 吗?

对于对象成员而言，是的。然而用户不能够添加空值(null)到数据库丛集(collection)因为空值不是对象。

然而用户能够添加空对象{}。

13. 更新操作立刻 fsync 到磁盘?

不会，磁盘写操作默认是延迟执行的。写操作可能在两三秒(默认在 60 秒内)后到达磁盘。例如，如果一秒内数据库收到一千个对一个对象递增的操作，仅刷新磁盘一次。(注意，尽管 fsync 选项在命令行和经过 getLastError_old 是有效的).

14. 如何执行事务/加锁?

MongoDB 没有使用传统的锁或者复杂的带回滚的事务，因为它设计的宗旨是轻量，快速以及可预计的高性能。可以把它类比成 MySQL MylSAM 的自动提交模式。通过精简对事务的支持，性能得到了提升，特别是在一个可能会穿过多个服务器的系统里。

15. 为什么我的数据文件如此庞大?

MongoDB 会积极的预分配预留空间来防止文件系统碎片。

16. 启用备份故障恢复需要多久?

从备份数据库声明主数据库宕机到选出一个备份数据库作为新的主数据库将花费 10 到 30 秒时间。这期间在主数据库上的操作将会失败--包括写入和强一致性读取(strong consistent read)操作。然而，你还能在第二数据库上执行最终一致性查询(eventually consistent query)(在 slaveOk 模式下)，即使在这段时间里。

17. 什么是 master 或 primary?

它是当前备份集群(replica set)中负责处理所有写入操作的主要节点/成员。在一个备份集群中，当失效备援(failover)事件发生时，一个另外的成员会变成 primary。

18. 什么是 secondary 或 slave?

Seconday 从当前的 primary 上复制相应的操作。它是通过跟踪复制 oplog(local.oplog.rs)做到的。

19. 我必须调用 getLastError 来确保写操作生效了么?

不用。不管你有没有调用 getLastError(又叫"Safe Mode")服务器做的操作都一样。调用 getLastError 只是为了确认写操作成功提交了。当然，你经常想得到确认，但是写操作的安全性和是否生效不是由这个决定的。

20. 我应该启动一个集群分片(sharded)还是一个非集群分片的 MongoDB 环境?

为开发便捷起见，我们建议以非集群分片(unsharded)方式开始一个 MongoDB 环境，除非一台服务器不足以存放你的初始数据集。从非集群分片升级到集群分片(sharding)是无缝的，所以在你的数据集还不是很大的时候没必要考虑集群分片(sharding)。

21. 分片(sharding)和复制(replication)是怎样工作的?

每一个分片(shard)是一个分区数据的逻辑集合。分片可能由单一服务器或者集群组成，我们推荐为每一个分片(shard)使用集群。

22. 数据在什么时候才会扩展到多个分片(shard)里?

MongoDB 分片是基于区域(range)的。所以一个集合(collection)中的所有的对象都被存放到一个块(chunk)中。只有当存在多余一个块的时候，才会有多个分片获取数据的选项。现在，每个默认块的大小是 64Mb，所以你需要至少 64 Mb 空间才可以实施一个迁移。

23. 当我试图更新一个正在被迁移的块(chunk)上的文档时会发生什么?

更新操作会立即发生在旧的分片(shard)上，然后更改才会在所有权转移(ownership transfers)前复制到新的分片上。

24. 如果在一个分片(shard)停止或者很慢的时候，我发起一个查询会怎样?

如果一个分片(shard)停止了，除非查询设置了“Partial”选项，否则查询会返回一个错误。如果一个分片(shard)响应很慢，MongoDB 则会等待它的响应。

25. 我可以把 moveChunk 目录里的旧文件删除吗?

没问题，这些文件是在分片(shard)进行均衡操作(balancing)的时候产生的临时文件。一旦这些操作已经完成，相关的临时文件也应该被删除掉。但目前清理工作是需要手动的，所以请小心地考虑再释放这些文件的空间。

26. 我怎么查看 Mongo 正在使用的链接?

```
db._adminCommand("connPoolStats");
```

27. 如果块移动操作(moveChunk)失败了，我需要手动清除部分转移的文档吗?

不需要，移动操作是一致(consistent)并且是确定性的(deterministic);一次失败后，移动操作会不断重试;

当完成后，数据只会出现在新的分片里(shard)。

28. 如果我在使用复制技术(replication)，可以一部分使用日志(journaling)而其他部分则不使用吗?

可以。

29. 当更新一个正在被迁移的块（Chunk）上的文档时会发生什么？

更新操作会立即发生在旧的块（Chunk）上，然后更改才会在所有权转移前复制到新的分片上。

30.MongoDB 在 A:{B,C}上建立索引，查询 A:{B,C}和 A:{C,B}都会使用索引吗？

不会，只会在 A:{B,C}上使用索引。

31.如果一个分片（Shard）停止或很慢的时候，发起一个查询会怎样？

如果一个分片停止了，除非查询设置了“Partial”选项，否则查询会返回一个错误。如果一个分片响应很慢，MongoDB 会等待它的响应。

32. MongoDB 支持存储过程吗？如果支持的话，怎么用？

MongoDB 支持存储过程，它是 Javascript 写的，保存在 db.system.js 表中。

33.如何理解 MongoDB 中的 GridFS 机制，MongoDB 为何使用 GridFS 来存储文件？

GridFS 是一种将大型文件存储在 MongoDB 中的文件规范。使用 GridFS 可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了 BSON 对象有限制的问题。

分库分表面试题目

1.为什么要分库分表

跟着你的公司业务发展走的，你公司业务发展越好，用户就越多，数据量越大，请求量越大，那你单个数据库一定扛不住。

比如你单表都几千万数据了，你确定你能抗住么？绝对不行，单表数据量太大，会极大影响你的 sql 执行的性能，到了后面你的 sql 可能就跑的很慢了。一般来说，就以我的经验来看，单表到几百万的时候，性能就会相对差一些了，你就得分表了。

分表是啥意思？就是把一个表的数据放到多个表中，然后查询的时候你就查一个表。比如按照用户 id 来分表，将一个用户的数据就放在一个表中。然后操作的时候你对一个用户就操作那个表就好了。这样可以控制每个表的数据量在可控的范围内，比如每个表就固定在 200 万以内。

分库是啥意思？就是你一个库一般我们经验而言，最多支撑到并发 2000，一定要扩容了，而且一个健康的单库并发值你最好保持在每秒 1000 左右，不要太大。那么你可以将一个库的数据拆分到多个库中，访问的时候就访问一个库好了。

2.用过哪些分库分表中间件？

比较常见的包括：cobar、TDDL、atlas、sharding-jdbc、mycat

3.不同的分库分表中间件都有什么优点和缺点？

sharding-jdbc 这种 client 层方案的优点在于不用部署，运维成本低，不需要代理层的二次转发请求，性能很高，但是如果遇到升级啥的需要各个系统都重新升级版本再发布，各个系统都需要耦合 sharding-jdbc 的依赖；

mycat 这种 proxy 层方案的缺点在于需要部署，自己及运维一套中间件，运维成本高，但是好处在于对于各个项目是透明的，如果遇到升级之类的都是自己中间件那里搞就行了。

4.你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

水平拆分的意思，就是把一个表的数据给弄到多个库的多个表里去，但是每个库的表结构都一样，只不过每个库表放的数据是不同的，所有库表的数据加起来就是全部数据。水平拆分的意义，就是将数据均匀放更多的库里，然后用多个库来抗更高的并发，还有就是用多个库的存储容量来进行扩容。

垂直拆分的意思，就是把一个有很多字段的表给拆分成多个表，或者是多个库上去。

每个库表的结构都不一样，每个库表都包含部分字段。一般来说，会将较少的访问频率很高的字段放到一个表里去，然后将较多的访问频率很低的字段放到另外一个表里去。因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。

你的项目里该如何分库分表？一般来说，垂直拆分，你可以在表层面来做，对一些字段特别多的表做一下拆分；水平拆分，你可以说是并发承载不了，或者是数据量太大，容量承载不了，你给拆了，按什么字段来拆，你自己想好；分表，你考虑一下，你如果哪怕是拆到每个库里去，并发和容量都 ok 了，但是每个库的表还是太大了，那么你就分表，将这个表分开，保证每个表的数据量并不是很大。

而且这儿还有两种分库分表的方式，一种是按照 range 来分，就是每个库一段连续的数据，这个一般是按比如时间范围来的，但是这种一般较少用，因为很容易产生热点问题，大量的流量都打在最新的数据上了；或者是按照某个字段 hash 一下均匀分散，这个较为常用。

5. 现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？

简单来说，就是在线上系统里面，之前所有写库的地方，增删改操作，都除了对老库增删改，都加上对新库的增删改，这就是所谓双写，同时写俩库，老库和新库。

然后系统部署之后，新库数据差太远，用之前说的导数工具，跑起来读老库数据写新库，写的时候要根据 gmt_modified 这类字段判断这条数据最后修改的时间，除非是读出来的数据在新库里没有，或者是比新库的数据新才会写。

接着导万一轮之后，有可能数据还是存在不一致，那么就程序自动做一轮校验，比对新老库每个表的每条数据，接着如果有不一样的，就针对那些不一样的，从老库读数据再次写。反复循环，直到两个库每个表的数据都完全一致为止。

接着当数据完全一致了，就 ok 了，基于仅仅使用分库分表的最新代码，重新部署一次，不就仅仅基于分库分表在操作了么，还没有几个小时的停机时间，很稳。所以现在基本玩儿数据迁移之类的，都是这么干了。

6. 如何设计可以动态扩容缩容的分库分表方案？

一开始上来就是 32 个库，每个库 32 个表，1024 张表

我可以告诉各位同学说，这个分法，第一，基本上国内的互联网肯定都是够用了，第二，无论是并发支撑还是数据量支撑都没问题, 每个库正常承载的写入并发量是 1000，那么 32 个库就可以承载 32 * 1000 = 32000 的写并发，如果每个库承载 1500 的写并发，32 * 1500 = 48000 的写并发，接近 5 万/s 的写入并发，前面再加一个 MQ，削峰，每秒写入 MQ 8 万条数据，每秒消费 5 万条数据。

有些除非是国内排名非常靠前的这些公司，他们的最核心的系统的数据库，可能会出现几百台数据库的这么一个规模，128 个库，256 个库，512 个库 1024 张表，假设每个表放

500 万数据，在 MySQL 里可以放 50 亿条数据 每秒的 5 万写并发，总共 50 亿条数据，对于国内大部分的互联网公司来说，其实一般来说都够了 谈分库分表的扩容，第一次分库分表，就一次性给他分个够，32 个库，1024 张表，可能对大部分的中小型互联网公司来说，已经可以支撑好几年了 一个实践是利用 32 * 32 来分库分表，即分为 32 个库，每个库里一个表分为 32 张表。一共就是 1024 张表。根据某个 id 先根据 32 取模路由到库，再根据 32 取模路由到库里的表。

刚开始的时候，这个库可能就是逻辑库，建在一个数据库上的，就是一个 mysql 服务器可能建了 n 个库，比如 16 个库。后面如果要拆分，就是不断在库和 mysql 服务器之间做迁移就可以了。然后系统配合改一下配置即可。

7.你们有没有做 MySQL 读写分离？如何实现 mysql 的读写分离？MySQL 主从复制原理的是啥？如何解决 mysql 主从同步的延时问题？

其实很简单，就是基于主从复制架构，简单来说，就搞一个主库，挂多个从库，然后我们就单单只是写主库，然后主库会自动把数据给同步到从库上去。

主库将变更写 binlog 日志，然后从库连接到主库之后，从库有一个 IO 线程，将主库的 binlog 日志拷贝到自己本地，写入一个中继日志中。接着从库中有一个 SQL 线程会从中继日志读取 binlog，然后执行 binlog 日志中的内容，也就是在自己本地再次执行一遍 SQL，这样就可以保证自己跟主库的数据是一样的。

这里有一个非常重要的一点，就是从库同步主库数据的过程是串行化的，也就是说主库上并行的操作，在从库上会串行执行。所以这就是一个非常重要的点了，由于从库从主库拷贝日志以及串行执行 SQL 的特点，在高并发场景下，从库的数据一定会比主库慢一些，是有延时的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。

要考虑好应该在什么场景下来用这个 mysql 主从同步，建议是一般在读远远多于写，而且读的时候一般对数据时效性要求没那么高的时候，用 mysql 主从同步。通常来说，我们会对于那种写了之后立马就要保证可以查到的场景，采用强制读主库的方式。如果主从延迟较为严重，分库，将一个主库拆分为 4 个主库，每个主库的写并发就 500/s，此时主从延迟可以忽略不计；打开 mysql 支持的并行复制，多个库并行复制

8.分库分表之后，id 主键如何处理？

数据库自增 id，

这个就是说你的系统里每次得到一个 id，都是往一个库的一个表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个 id。拿到这个 id 之后再往对应的分库分表里去写入。

这个方案的好处就是方便简单，谁都会用；缺点就是单库生成自增 id，要是高并发的话，就会有瓶颈的；如果你硬是要改进一下，那么就专门开一个服务出来，这个服务每次就拿到当前 id 最大值，然后自己递增几个 id，一次性返回一批 id，然后再把当前最大 id 值修改成递增几个 id 之后的一个值；但是无论如何都是基于单个数据库。

UUID

好处就是本地生成，不要基于数据库来了；不好之处就是，UUID 太长了，作为主键性能太差了，另外 UUID 不具有有序性，会造成 B+ 树索引在写的时候有过多的随机写操作，频繁修改树结构，从而导致性能下降。

snowflake 算法

网络通讯面试题目

1. BIO 与 NIO 的区别

1、bio 同步阻塞 io：在此种方式下，用户进程在发起一个 IO 操作以后，必须等待 IO 操作的完成，只有当真正完成了了 IO 操作以后，用户进程才能运行。JAVA 传统的 IO 模型属于此种方式。

2、nio 同步非非阻塞式 I/O：Java NIO 采用用了了双向通道进行行行数据传输，在通道上我们可以注册我们感兴趣的事件：连接事件、读写事件；

NIO 主要有三大核心部分：Channel(通道)，Buffer(缓冲区)，Selector。传统 IO 基于字节流和字符流进行行行操作，而而 NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。

2. select 与 poll 的区别

A、io 多路复用：

1、概念：IO 多路路复用是指内核一旦发现进程指定的一个或者多个 IO 条件准备读取，它就通知该进程。

2、优势：与多进程和多线程技术相比，I/O 多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

3、系统：目前支持 I/O 多路路复用的系统调用有 select，pselect，poll，epoll。

B、select：

select 目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

C、poll：

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

a. 大量的 fd 的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不

是有意义。

b. poll 还有一个特点是“水平触发”，如果报告了了 fd 后，没有被处理，那么下次 poll 时会再次报告该 fd。

3. 请概述 OSI 网络模型

OSI 网络模型共有 7 层，自上而下分别是：

应用层（数据）：确定进程之间通信的性质以满足用户需要以及提供网络与用户应用。

表示层（数据）：主要解决用户信息的语法表示问题，如加密解密。在表示层进行代码/编码转换。

会话层(数据)：提供包括访问验证和会话管理在内的建立和维护应用之间通信的机制，如服务器验证用户登录便是由会话层完成的。在会话层封装会话控制参数。

传输层（段）：实现网络不同主机上用户进程之间的数据通信，可靠与不可靠的传输，传输层的错误检测，流量控制等。在传输层封装传输控制。

网络层（包）：提供逻辑地址（IP）、选路，数据从源端到目的端的传输。在网络层加上逻辑寻址地址。

数据链路层（帧）：将上层数据封装成帧，用 MAC 地址访问媒介，错误检测与修正。在数据链路层封装基于 MAC 的信息。

物理层（比特流）：设备之间比特流的传输，物理接口，电气特性等。在物理层连接到线缆系统进行实际传递。

4. TCP 和 UDP 的区别

TCP 和 UDP 都属于传输层协议，它们之间的区别在于：

TCP 是面向连接的；UDP 是无连接的。

TCP 是可靠的；UDP 是不可靠的。

TCP 只支持点对点通信；UDP 支持一对一、一对多、多对一、多对多的通信模式。

TCP 是面向字节流的；UDP 是面向报文的。

TCP 有拥塞控制机制；UDP 没有拥塞控制，适合媒体通信。

TCP 首部开销(20 个字节)，比 UDP 的首部开销(8 个字节)要大。

5. 请概述 TCP 的三次握手四次挥手机制

1. TCP 建立连接的过程。

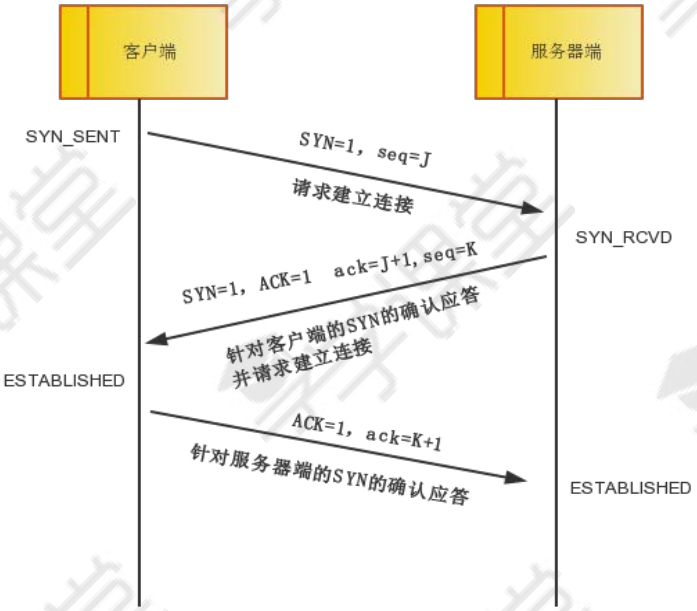
三次握手：

1. 第一次握手(客户端发送 syn 包到服务器端)：客户端发送 syn 包到服务器端，进入 syn_send 状态，等待服务器端的确认；

2. 第二次握手(服务器返回 syn+ack 包给客户端)：服务器端收到客户端的 syn 包，发送 syn+ack 包给客户端，进入 syn_recv 状态；

3. 第三次握手(客服端返回 ack 包给服务端)：客户端收到服务器端的 syn+ack 包，发送个 ack 包到服务器端，至此，客户端与服务器端进入 established 状态；

4. 握手过程中传送的包不包含任何数据，连接建立后才会开始传送数据，理想状态下，TCP 连接一旦建立，在通信双方的任何一方主动关闭连接前，TCP 连接都会一直保持下去。



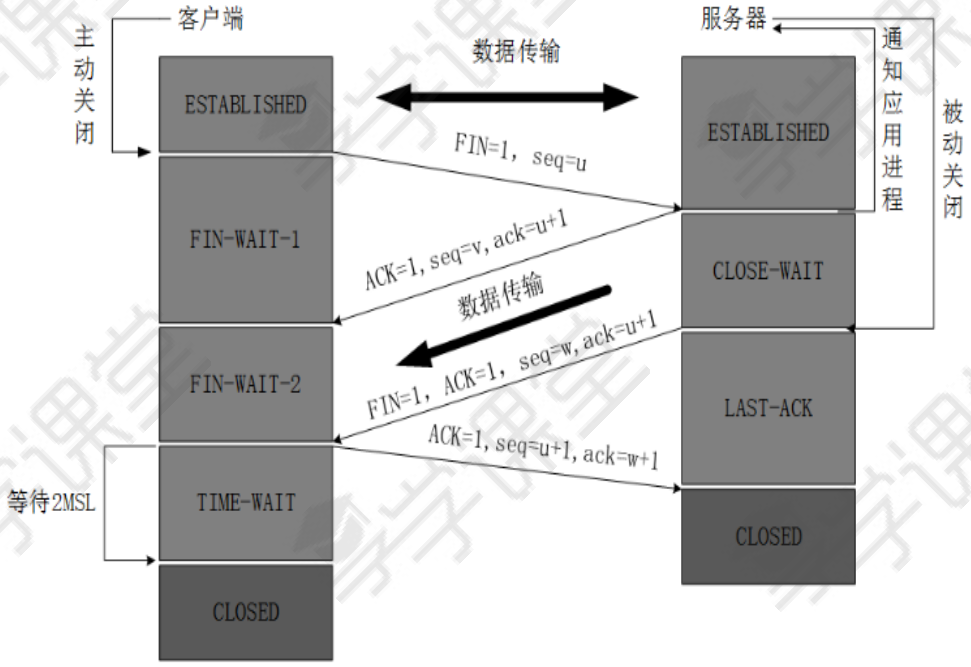
四次挥手：

1. 第一次挥手：主动关闭方发送 fin 包到被动关闭方，告诉被动关闭方我不会再给你发送数据了；

2. 第二次挥手：被动关闭方收到 syn 包，发送 ack 给对方，确认序号为收到序号+1；

3. 第三次挥手：被动关闭方也也发送 fin 包给主动关闭方，告诉对方我也不会给你发送数据了；

4. 第四次挥手：主动关闭方收到 syn 包，发送 ack 给对方，至此，完成四次挥手；



6. 为什么 TCP 握手需要三次?

TCP 是可靠的传输控制协议，三次握手能保证数据可靠传输又能提高传输效率。

如果 TCP 的握手是两次：

<1>若建立连接只需两次握手，客户端并没有太大的变化，仍然需要获得服务端的应答后才进入 ESTABLISHED 状态，而服务端在收到连接请求后就进入 ESTABLISHED 状态。此时如果网络拥塞，客户端发送的连接请求迟迟到不了服务端，客户端便超时重发请求，如果服务端正确接收并确认应答，双方便开始通信，通信结束后释放连接。此时，如果那个失效的连接请求抵达了服务端，由于只有两次握手，服务端收到请求就会进入 ESTABLISHED 状态，等待发送数据或主动发送数据。但此时的客户端早已进入 CLOSED 状态，服务端将会一直等待下去，这样浪费服务端连接资源。

如果 TCP 的握手是四次：

- 1. client 给 server 发送 SYN 同步报文；
- 2. server 收到 SYN 后，给 client 回复 ACK 确认报文；
- 3. server 给 client 发送 SYN 同步报文；
- 4. client 给 server 发送 ACK 确认报文。

第 2.3 步之间，server 和 client 没有任何的数据交互，分开发送相当于多发了一次 TCP 报文段，SYN 和 ACK 标识只是 TCP 报头的一个标识位。很明显，这两步可以合并，从而提高连接的速度和效率。

7. 为什么 TCP 的挥手需要四次？

因为 TCP 连接是全双工的网络协议，允许同时通信的双方同时进行数据的收发，同样也允许收发两个方向的连接被独立关闭，以避免 client 数据发送完毕，向 server 发送 FIN

关闭连接，而 server 还有发送到 client 的数据没有发送完毕的情况。所以关闭 TCP 连接需要进行四次握手，每次关闭一个方向上的连接需要 FIN 和 ACK 两次握手。

8. 什么是 DDOS 攻击

DDOS 攻击利用合理的服务请求占用过多的服务资源，使正常用户的请求无法得到相应。

常见的 DDOS 攻击有计算机网络带宽攻击和连通性攻击。

带宽攻击指以极大的通信量冲击网络，使得所有可用网络资源都被消耗殆尽，最后导致合法的用户请求无法通过。

连通性攻击指用大量的连接请求冲击计算机，使得所有可用的操作系统资源都被消耗殆尽，最终计算机无法再处理合法用户的请求。

9. 什么是 SYN 洪水攻击

SYN 洪水攻击属于 DOS 攻击的一种，它利用 TCP 协议缺陷，通过发送大量的半连接请求，耗费 CPU 和内存资源。

客户端在短时间内伪造大量不存在的 IP 地址，向服务器不断地发送 SYN 报文，服务器回复 ACK 确认报文，并等待客户的确认，由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的 SYN 报文被丢弃，目标系统运行缓慢，严重者引起网络堵塞甚至系统瘫痪。

10. HTTP1.0 和 HTTP1.1 的区别

主要是如下的 8 点：

可拓展性。

缓存。

带宽优化，带来了分块传输。

长连接，HTTP1.1 支持长连接（默认开启 Connect: keep-alive）和请求的流水线处理，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。

消息传递。

Host 头域。

错误提示。

内容协商。

11. Https 原理是什么？

HTTPS 协议就是基于 SSL 的 HTTP 协议，HTTPS 使用与 HTTP 不同的端口（HTTP80 ，HTTPS443）

提供了身份验证与加密通信方法，被广泛用于互联网上安全敏感的通信。

1、客户端请求 SSL 连接，并将自己支持的加密规则发给网站。

2、服务器端将自己己的身份信息以证书形式发回给客户端。证书里面包含了网站地址，加密公钥，以及证书的颁发机构。

3、获得证书后，客户要做以下工作：验证证书合法性，如果证书受信任，客户端会生成一串随机数的密码，并用证书提供的公钥进行加密。将加密好的随机数发给服务器。

4、获得到客户端发的加密了的随机数之后，服务器用自己的私钥进行解密，得到这个随机数，把这个随机数作为对称加密的密钥。（利用非对称加密传输对称加密的密钥）

5、之后服务器与客户之间就可以用随机数对各自的信息进行加密，解密。

注意的是：证书是一个公钥，这个公钥是进行加密用的。而私钥是进行解密用的。公钥任何都知道，私钥只有自己知道。这是非对称加密。

而对称加密就是钥匙只有一把，我们都知道。之所以用到对称加密，是因为对称加密的速度更快。而非对称加密的可靠性更高。

12. 说说你知道的几种 HTTP 响应码。

HTTP 响应码主要分为五种：

1XX：请求处理中，请求已被接收，正在处理。

2XX：请求成功，请求被成功处理。比如 200，OK，表示客户端请求成功。

3XX：重定向，要完成请求必须进行进一步处理。比如 301，MovedPermanently，永久重定向，使用域名跳转；302，Found，临时重定向，未登录的用户访问用户中心重定向到登陆界面。

4XX：客户端错误，请求不合符。比如 400，Bad Request，客户端请求有语法错误，不能被服务器所理解；401，Unauthrized，请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用；403，Forbidden，服务器收到请求，但是拒绝提供服务；404，Not Found，请求资源不存在，输入了错误的 URL。

5XX：服务器端错误，服务器不能处理合法请求。比如 500，Internal Servel Error，服务器发生不可预期的错误；503，Server Unavailable，服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

13. 如何理解 HTTP 协议的无状态性。

无状态，是指协议对于事务处理没有记忆功能。HTTP 是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive 没能改变这个结果。无状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就很快。

无状态，更容易做服务的扩容，支撑更大的访问量。

14. Session 和 cookie 的区别。

Session 和 cookie 都是实现对话管理的方案。主要区别在于：

Session 在服务端，Cookie 存储在客户端。

Session 的运行依赖 Session ID，而 Session ID 是存在 Cookie 中的，也就是说，如果浏览器禁用了 Cookie，同时 Session 也会失效，但是，可以通过其他方式实现，比如在 url 参数中传递 Session ID。

Tomcat 中的 Session 是存在服务器内存中，不过也可以通过特殊的方式做持久化处理（memcache，redis），方便 Session 共享。

cookie 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗，考虑到安全应当使用 session。

15. 用户在浏览器输入一个 URL 并回车，这个过程涉及到哪些网络协议。

浏览器输入一个 URL 并回车：

1. 首先进行域名解析，浏览器搜索自己的 DNS 缓存，缓存中维护一张域名与 IP 地址的对应表。若没有，则搜索操作系统的 DNS 缓存；若没有，则将域名发送至本地域名服务器（递归查询方式），本地域名服务器查询自己的 DNS 缓存，查找成功则返回结果，

否则，本地的 DNS 服务器向根域名服务器发出查询请求，根域名服务器告知该域名的一级域名服务器，然后本地服务器给该一级域名服务器发送查询请求，然后依次类推直到查询到该域名的 IP 地址。DNS 服务器是基于 UDP 的，因此会用到 UDP 协议。

2. 得到 IP 地址以后，浏览器就要与服务器建立一个 HTTP 连接，因此要用到 HTTP 协议。HTTP 生成一个 GET 请求报文。

3. 接下来到了传输层，选择传输协议，TCP 或者 UDP，TCP 是可靠的传输控制协议，对 HTTP 请求进行封装，加入了端口号等信息。

4. 然后到了网络层,通过 IP 协议将 IP 地址封装为 IP 数据报;然后此时会用到 ARP 协议，主机发送信息时将包含目标 IP 地址的 ARP 请求广播到网络上的所有主机，并接收返回消息，以此确定目标的物理地址，找到目的 MAC 地址。

5. 接下来到了数据链路层，把网络层交下来的 IP 数据报添加首部和尾部，封装为 MAC 帧，现在根据目的 mac 开始建立 TCP 连接，三次握手，接收端在收到物理层上交的比特流后，根据首尾的标记，识别帧的开始和结束，将中间的数据部分上交给网络层，然后层层向上传递到应用层。

6. 服务器响应请求并请求客户端要的资源，传回给客户端。

7. 断开 TCP 连接，浏览器对页面进行渲染呈现给客户端。

16. HTTP2.0 带来的变化。

HTTP2.0 和 HTTP1.x 相比的新特性为：

新的二进制格式,HTTP1.x 的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认 0 和 1 的组合。基于这种考虑 HTTP2.0 的协议解析决定采用二进制格式，实现方便且健壮。

多路复用，即连接共享，做到同一个连接并发处理多个请求，而且并发请求的数量比 HTTP1.1 大了好几个数量级。

多个请求可以同时在一个连接上并行执行，某个请求任务耗时严重，不会影响到其他连接的正常执行。这块和 HTTP1.1 的长连接有区别，长连接是串行化执行多个请求。

首部压缩,HTTP1.1 不支持 header 数据的压缩,HTTP2.0 使用 HPACK 算法对 header 的数据进行压缩，这样数据体积小了，在网络上传输就会更快。

服务器推送，当我们对支持 HTTP2.0 的 web server 请求数据的时候，服务器会顺便把一些客户端需要的资源（比如 css、js 文件）一起推送到客户端，免得客户端再次创建连接发送请求到服务器端获取。这种方式非常合适加载静态资源。还没有收到浏览器的请求，服务器就把各种资源推送给浏览器了。

17. Rest和Http什么关系?你对Rest风格如何理解?

Http 是一种协议，Rest 是一种软件架构风格。REST 架构风格并不是绑定在 HTTP 上，只不过目前 HTTP 是唯一与 REST 相关的实例。

REST (英文: Representational State Transfer, 简称 REST，表现层状态转化)，指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。

Rest 作为一种软件架构风格而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

Restful 架构中

(1) 每一个 URI 代表一种资源;

(2) 客户端和服务器之间，传递这种资源的某种表现层;

(3) 客户端通过四个 HTTP 动词(GET 用来获取资源，POST 用来新建资源(也可以用于更新资源)，PUT 用来更新资源，DELETE 用来删除资源。)，对服务器端资源进行操作，实现"表现层状态转化”。

18. 哪些应用比较适合用 udp 实现

多播的信息一定要用 udp 实现，因为 tcp 只支持一对一通信。

如果一个应用场景中大多是简短的信息，适合用 udp 实现，因为 udp 是基于报文段的，它直接对上层应用的数据封装成报文段，然后丢在网络中，如果信息量太大，会在链路层中被分片，影响传输效率。

如果一个应用场景重性能甚于重完整性和安全性，那么适合于 udp，比如多媒体应用，缺一两帧不影响用户体验，但是需要流媒体到达的速度快，因此比较适合用 udp

如果要求快速响应，那么 udp 听起来比较合适

如果又要利用 udp 的快速响应优点，又想可靠传输，那么只能考上层应用自己制定规则了。

常见的使用 udp 的例子：ICQ,QQ 的聊天模块。

19. Netty 的特点？

一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输的支持

使用更高效的 socket 底层，对 epoll 空轮询引起的 cpu 占用飙升在内部进行了处理，避免了直接使用 NIO 的陷阱，简化了 NIO 的处理方式。

采用多种 decoder/encoder 支持，对 TCP 粘包/分包进行自动化处理

可使用接受/处理线程池，提高连接效率，对重连、心跳检测的简单支持

可配置 IO 线程数、TCP 参数， TCP 接收和发送缓冲区使用直接内存代替堆内存，通过内存池的方式循环利用 ByteBuf

通过引用计数器及时申请释放不再引用的对象，降低了 GC 频率

使用单线程串行化的方式，高效的 Reactor 线程模型

大量使用了 volitale、使用了 CAS 和原子类、线程安全类的使用、读写锁的使用

20. Netty 的线程模型

Netty 通过 Reactor 模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss 线程池和 work 线程池，其中 boss 线程池的线程负责处理请求的 accept 事件，当接收到 accept 事件的请求时，把对应的 socket 封装到一个 NioSocketChannel 中，并交给 work 线程池，其中 work 线程池负责请求的 read 和 write 事件，由对应的 Handler 处理。

单线程模型：所有 I/O 操作都由一个线程完成，即多路复用、事件分发和处理都是在一个 Reactor 线程上完成的。既要接收客户端的连接请求,向服务端发起连接，又要发送/读取请求或应答/响应消息。一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高负载、大并发的应用场景不合适。

多线程模型：有一个 NIO 线程（Acceptor） 只负责监听服务端，接收客户端的 TCP 连接请求；NIO 线程池负责网络 IO 的操作，即消息的读取、解码、编码和发送；1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安全认证时，一个 Acceptor 线程可能会存在性能不足问题。

主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将 SocketChannel 从主线程池的 Reactor 线程的多路复用器上移除，重新注册到 Sub 线程池的线程上，用于处理 I/O 的读写等操作，从而保证 mainReactor 只负责接入认证、握手等操作；

21. TCP 粘包/拆包的原因及解决方法？

TCP 是以流的方式来处理数据，一个完整的包可能会被 TCP 拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP 粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象；

进行 MSS 大小的 TCP 分段，当 TCP 报文长度-TCP 头部长度>MSS 的时候将发生拆包

以太网帧的 payload（净荷）大于 MTU（1500 字节）进行 ip 分片。

解决方法

消息定长：FixedLengthFrameDecoder 类

包尾增加特殊字符分割：行分隔符类：LineBasedFrameDecoder 或自定义分隔符类 ：DelimiterBasedFrameDecoder

将消息分为消息头和消息体：LengthFieldBasedFrameDecoder 类。分为有头部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘包。

22. 请概要介绍下序列化

序列化（编码）是将对象序列化为二进制形式（字节数组），主要用于网络传输、数据持久化等；而反序列化（解码）则是将从网络、磁盘等读取的字节数组还原成原始对象，主要用于网络传输对象的解码，以便完成远程调用。

影响序列化性能的关键因素：序列化后的码流大小（网络带宽的占用）、序列化的性能（CPU 资源占用）；是否支持跨语言（异构系统的对接和开发语言切换）。

Java 默认提供的序列化：无法跨语言、序列化后的码流太大、序列化的性能差

XML，优点：人机可读性好，可指定元素或特性的名称。缺点：序列化数据只包含数据本身以及类的结构，不包括类型标识和程序集信息；只能序列化公共属性和字段；不能序列化方法；文件庞大，文件格式复杂，传输占带宽。适用场景：当做配置文件存储数据，实时数据转换。

JSON，是一种轻量级的数据交换格式，优点：兼容性高、数据格式比较简单，易于读写、序列化后数据较小，可扩展性好，兼容性好、与 XML 相比，其协议比较简单，解析速度比较快。缺点：数据的描述性比 XML 差、不适合性能要求为 ms 级别的情况、额外空间开销比较大。适用场景（可替代ＸＭＬ）：跨防火墙访问、可调式性要求高、基于 Web browser 的 Ajax 请求、传输数据量相对小，实时性要求相对低（例如秒级别）的服务。

Fastjson，采用一种“假定有序快速匹配”的算法。优点：接口简单易用、目前 Java 语言中最快的 json 库。缺点：过于注重快，而偏离了“标准”及功能性、代码质量不高，文档不全。适用场景：协议交互、Web 输出、Android 客户端

Thrift，不仅是序列化协议，还是一个 RPC 框架。优点：序列化后的体积小, 速度快、支持多种语言和丰富的数据类型、对于数据字段的增删具有较强的兼容性、支持二进制压缩编码。缺点：使用者较少、跨防火墙访问时，不安全、不具有可读性，调试代码时相对困难、不能与其他传输层协议共同使用（例如 HTTP）、无法支持向持久层直接读写数据，即不适合做数据持久化序列化协议。适用场景：分布式系统的 RPC 解决方案

Protobuf，将数据结构以.proto 文件进行描述，通过代码生成工具可以生成对应数据结构的 POJO 对象和 Protobuf 相关的方法和属性。优点：序列化后码流小，性能高、结构化数据存储格式（XML JSON 等）、通过标识字段的顺序，可以实现协议的前向兼容、结构化的文档更容易管理和维护。缺点：需要依赖于工具生成代码、支持的语言相对较少，官方只支持 Java 、C++ 、python。适用场景：对性能要求高的 RPC 调用、具有良好的跨防火墙的访问属性、适合应用层对象的持久化

其它

protostuff 基于 protobuf 协议，但不需要配置 proto 文件，直接导包即可

Jboss marshaling 可以直接序列化 Java 类， 无须实 Java.io.Serializable 接口

Message pack 一个高效的二进制序列化格式

Hessian 采用二进制协议的轻量级 remoting onhttp 工具

kryo 基于 protobuf 协议，只支持 Java 语言,需要注册（Registration），然后序列化（Output），反序列化（Input）

23. Netty 的零拷贝实现

Netty 的零拷贝主要包含三个方面：

Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行

Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。

Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

(高)Netty 是如何解决 JDK 中的 Selector BUG 的？

Selector BUG：若 Selector 的轮询结果为空，也没有 wakeup 或新消息处理，则发生空轮询，CPU 使用率 100%，

Netty 的解决办法：对 Selector 的 select 操作周期进行统计，每完成一次空的 select 操作进行一次计数，若在某个周期内连续发生 N 次空轮询，则触发了 epoll 死循环 bug。重建 Selector，判断是否是其他线程发起的重建请求，若不是则将原 SocketChannel 从旧的 Selector 上去除注册，重新注册到新的 Selector 上，并将原来的 Selector 关闭。

24. Netty 的优势有哪些？

使用简单：封装了 NIO 的很多细节，使用更简单。

功能强大：预置了多种编解码功能，支持多种主流协议。

定制能力强：可以通过 ChannelHandler 对通信框架进行灵活地扩展。

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。

稳定：Netty 修复了已经发现的所有 NIO 的 bug，让开发人员可以专注于业务本身。

社区活跃：Netty 是活跃的开源项目，版本迭代周期短，bug 修复速度快。

25. (高)Netty 高性能表现在哪些方面？

IO 线程模型：同步非阻塞，用最少的资源做更多的事。

内存零拷贝：尽量减少不必要的内存拷贝，实现了更高效率的传输。

内存池设计：申请的内存可以重用，主要指直接内存。内部实现是用一颗二叉查找树管理内存分配情况。

串形化处理读写：避免使用锁带来的性能开销。即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

高性能序列化协议：支持 protobuf 等高性能序列化协议。

高效并发编程的体现：volatile 的大量、正确使用；CAS 和原子类的广泛使用；线程安全容器的使用；通过读写锁提升并发性能。

26. Netty 中有哪些重要组件？

Channel：Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、connect、read、write 等。

EventLoop：主要是配合 Channel 处理 I/O 操作，用来处理连接的生命周期中所发生的事情。

ChannelFuture：Netty 框架中所有的 I/O 操作都为异步的，因此我们需要 ChannelFuture 的 addListener()注册一个 ChannelFutureListener 监听事件，当操作执行成功或者失败时，监听就会自动触发返回结果。

ChannelHandler：充当了所有处理入站和出站数据的逻辑容器。ChannelHandler 主要用来处理各种事件，这里的事件很广泛，比如可以是连接、数据接收、异常、数据转换等。

ChannelPipeline：为 ChannelHandler 链提供了容器，当 channel 创建时，就会被自动分配到它专属的 ChannelPipeline，这个关联是永久性的。

27. 如何让单机下 Netty 支持百万长连接？

单机下能不能让我们的网络应用支持百万连接？可以，但是有很多的工作要做。

操作系统

首先就是要突破操作系统的限制。

在 Linux 平台上，无论编写客户端程序还是服务端程序，在进行高并发 TCP 连接处理时，最高的并发数量都要受到系统对用户单一进程同时可打开文件数量的限制(这是因为系统为每个 TCP 连接都要创建一个 socket 句柄，每个 socket 句柄同时也是一个文件句柄）。

可使用 ulimit 命令查看系统允许当前用户进程打开的文件数限制：

```
$ ulimit -n
1024
```

这表示当前用户的每个进程最多允许同时打开 1024 个文件，这 1024 个文件中还得除去每个进程必然打开的标准输入，标准输出，标准错误，服务器监听 socket,进程间通讯的 unix 域 socket 等文件，那么剩下的可用于客户端 socket 连接的文件数就只有大概 1024-10=1014 个左右。也就是说缺省情况下，基于 Linux 的通讯程序最多允许同时 1014 个 TCP 并发连接。

对于想支持更高数量的 TCP 并发连接的通讯处理程序，就必须修改 Linux 对当前用户的进程同时打开的文件数量。

修改单个进程打开最大文件数限制的最简单的办法就是使用 ulimit 命令：

```
$ ulimit -n 1000000
```

如果系统回显类似于"Operation not permitted"之类的话，说明上述限制修改失败，实际上是因为在中指定的数值超过了 Linux 系统对该用户打开文件数的软限制或硬限制。因此，就需要修改 Linux 系统对用户的关于打开文件数的软限制和硬限制。

软限制（soft limit）:是指 Linux 在当前系统能够承受的范围内进一步限制用户同时打开的文件数；

硬限制（hardlimit）:是根据系统硬件资源状况（主要是系统内存）计算出来的系统最

多可同时打开的文件数量。

第一步，修改/etc/security/limits.conf 文件，在文件中添加如下行：

```
* soft nofile 1000000
* hard nofile 1000000
```

'*'号表示修改所有用户的限制；

soft 或 hard 指定要修改软限制还是硬限制；1000000 则指定了想要修改的新的限制值，即最大打开文件数（请注意软限制值要小于或等于硬限制）。修改完后保存文件。

第二步，修改/etc/pam.d/login 文件，在文件中添加如下行：

```
session required /lib/security/pam_limits.so
```

这是告诉 Linux 在用户完成系统登录后，应该调用 pam_limits.so 模块来设置系统对该用户可使用的各种资源数量的最大限制（包括用户可打开的最大文件数限制），而 pam_limits.so 模块就会从/etc/security/limits.conf 文件中读取配置来设置这些限制值。修改完后保存此文件。

第三步，查看 Linux 系统级的最大打开文件数限制，使用如下命令：

```
[speng@as4 ~]$ cat /proc/sys/fs/file-max
12158
```

这表明这台 Linux 系统最多允许同时打开(即包含所有用户打开文件数总和)12158 个文件，是 Linux 系统级硬限制，所有用户级的打开文件数限制都不应超过这个数值。通常这个系统级硬限制是 Linux 系统在启动时根据系统硬件资源状况计算出来的最佳的最大同时打开文件数限制，如果没有特殊需要，不应该修改此限制，除非想为用户级打开文件数限制设置超过此限制的值。

如何修改这个系统最大文件描述符的限制呢？修改 sysctl.conf 文件

```
vi /etc/sysctl.conf
 # 在末尾添加
fs.file_max = 1000000
 # 立即生效
sysctl -p
```

Netty 调优

设置合理的线程数

对于线程池的调优,主要集中在用于接收海量设备 TCP 连接、TLS 握手的 Acceptor 线程池(Netty 通常叫 boss NioEventLoop Group)上,以及用于处理网络数据读写、心跳发送的 IO 工作线程池(Nety 通常叫 work Nio EventLoop Group)上。

对于 Nety 服务端,通常只需要启动一个监听端口用于端侧设备接入即可,但是如果服务端集群实例比较少,甚至是单机(或者双机冷备)部署,在端侧设备在短时间内大量接入时,需要对服务端的监听方式和线程模型做优化,以满足短时间内(例如 30s)百万级的端侧设备接入的需要。

服务端可以监听多个端口,利用主从 Reactor 线程模型做接入优化,前端通过 SLB 做 4 层门 7 层负载均衡。

主从 Reactor 线程模型特点如下:服务端用于接收客户端连接的不再是一个单独的 NO 线程,而是一个独立的 NIO 线程池; Acceptor 接收到客户端 TCP 连接请求并处理后(可能包

含接入认证等),将新创建的 Socketchanne 注册到 I/O 线程池(subReactor 线程池)的某个 IO 线程,由它负责 Socketchannel 的读写和编解码工作; Acceptor 线程池仅用于客户端的登录、握手和安全认证等,一旦链路建立成功,就将链路注册到后端 sub reactor 线程池的 IO 线程,由 IO 线程负责后续的 IO 操作。

对于 IO 工作线程池的优化,可以先采用系统默认值(即 CPU 内核数×2)进行性能测试,在性能测试过程中采集 IO 线程的 CPU 占用大小,看是否存在瓶颈对于 O 工作线程池的优化,可以先采用系统默认值(即 CPU 内核数×2)进行性能

测试,在性能测试过程中采集 IO 线程的 CPU 占用大小,看是否存在瓶颈, 具体可以观察线程堆栈, 如果连续采集几次进行对比,发现线程堆栈都停留在 Selectorlmpl. lock AndDoSelect,则说明 IO 线程比较空闲,无须对工作线程数做调整。

如果发现 IO 线程的热点停留在读或者写操作,或者停留在 Channelhandler 的执行处,则可以通过适当调大 Nio EventLoop 线程的个数来提升网络的读写性能。

心跳优化

针对海量设备接入的服务端,心跳优化策略如下。

(1)要能够及时检测失效的连接,并将其剔除,防止无效的连接句柄积压,导致 OOM 等问题

(2)设置合理的心跳周期,防止心跳定时任务积压,造成频繁的老年代 GC(新生代和老年代都有导致 STW 的 GC,不过耗时差异较大),导致应用暂停

(3)使用 Nety 提供的链路空闲检测机制,不要自己创建定时任务线程池,加重系统的负担,以及增加潜在的并发安全问题。

当设备突然掉电、连接被防火墙挡住、长时间 GC 或者通信线程发生非预期异常时,会导致链路不可用且不易被及时发现。特别是如果异常发生在凌晨业务低谷期间,当早晨业务高峰期到来时,由于链路不可用会导致瞬间大批量业务失败或者超时,这将对系统的可靠性产生重大的威胁。

从技术层面看,要解决链路的可靠性问题,必须周期性地对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。心跳检测机制分为三个层面

(1)TCP 层的心跳检测,即 TCP 的 Keep-Alive 机制,它的作用域是整个 TCP 协议栈。

(2)协议层的心跳检测,主要存在于长连接协议中,例如 MQTT。

(3)应用层的心跳检测,它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。

心跳检测的目的就是确认当前链路是否可用,对方是否活着并且能够正常接收和发送消息。作为高可靠的 NIO 框架,Nety 也提供了心跳检测机制。

一般的心跳检测策略如下。

(1)连续 N 次心跳检测都没有收到对方的 Pong 应答消息或者 Ping 请求消息,则认为链路已经发生逻辑失效,这被称为心跳超时。

(2)在读取和发送心跳消息的时候如果直接发生了 IO 异常,说明链路已经失效,这被称为心跳失败。无论发生心跳超时还是心跳失败,都需要关闭链路,由客户端发起重连操作,保证链路能够恢复正常。

Nety 提供了三种链路空闲检测机制,利用该机制可以轻松地实现心跳检测

(1)读空闲,链路持续时间 T 没有读取到任何消息。

(2)写空闲,链路持续时间 T 没有发送任何消息

(3)读写空闲,链路持续时间 T 没有接收或者发送任何消息

对于百万级的服务器，一般不建议很长的心跳周期和超时时长

接收和发送缓冲区调优

在一些场景下,端侧设备会周期性地上报数据和发送心跳,单个链路的消息收发量并不大,针对此类场景,可以通过调小TCP的接收和发送缓冲区来降低单个TCP连接的资源占用率

当然对于不同的应用场景,收发缓冲区的最优值可能不同,用户需要根据实际场景,结合性能测试数据进行针对性的调优

合理使用内存池

随着 JVM 虚拟机和 JT 即时编译技术的发展,对象的分配和回收是一个非常轻量级的工作。但是对于缓冲区 Buffer,情况却稍有不同,特别是堆外直接内存的分配和回收,是一个耗时的操作。

为了尽量重用缓冲区,Nety 提供了基于内存池的缓冲区重用机制。

在百万级的情况下,需要为每个接入的端侧设备至少分配一个接收和发送缓冲区对象,采用传统的非池模式,每次消息读写都需要创建和释放 ByteBuf 对象,如果有 100 万个连接,每秒上报一次数据或者心跳,就会有 100 万次/秒的 ByteBuf 对象申请和释放,即便服务端的内存可以满足要求,GC 的压力也会非常大。

以上问题最有效的解决方法就是使用内存池,每个 NioEventLoop 线程处理 N 个链路,在线程内部,链路的处理是串行的。假如 A 链路首先被处理,它会创建接收缓冲区等对象,待解码完成,构造的 POJO 对象被封装成任务后投递到后台的线程池中执行,然后接收缓冲区会被释放,每条消息的接收和处理都会重复接收缓冲区的创建和释放。如果使用内存池,则当 A 链路接收到新的数据报时,从 NioEventLoop 的内存池中申请空闲的 ByteBuf,解码后调用 release 将 ByteBuf 释放到内存池中,供后续的 B 链路使用。

Nety 内存池从实现上可以分为两类:堆外直接内存和堆内存。由于 Byte Buf 主要用于网络 IO 读写,因此采用堆外直接内存会减少一次从用户堆内存到内核态的字节数组拷贝,所以性能更高。由于 DirectByteBuf 的创建成本比较高,因此如果使用 DirectByteBuf,则需要配合内存池使用,否则性价比可能还不如 Heap Byte。

Netty 默认的 IO 读写操作采用的都是内存池的堆外直接内存模式,如果用户需要额外使用 ByteBuf,建议也采用内存池方式;如果不涉及网络 IO 操作(只是纯粹的内存操作),可以使用堆内存池,这样内存的创建效率会更高一些。

IO 线程和业务线程分离

如果服务端不做复杂的业务逻辑操作,仅是简单的内存操作和消息转发,则可以通过调大 NioEventLoop 工作线程池的方式,直接在 IO 线程中执行业务 Channelhandler,这样便减少了一次线程上下文切换,性能反而更高。

如果有复杂的业务逻辑操作,则建议 IO 线程和业务线程分离,对于 IO 线程,由于互相之间不存在锁竞争,可以创建一个大的 NioEvent Loop Group 线程组,所有 Channel 都共享同一个线程池。

对于后端的业务线程池,则建议创建多个小的业务线程池,线程池可以与 IO 线程绑定,

这样既减少了锁竞争,又提升了后端的处理性能。

针对端侧并发连接数的流控

无论服务端的性能优化到多少,都需要考虑流控功能。当资源成为瓶颈,或者遇到端侧设备的大量接入,需要通过流控对系统做保护。流控的策略有很多种，比如针对端侧连接数的流控。在 Nety 中,可以非常方便地实现流控功能:新增一个 FlowControlchannelhandler，然后添加到 ChannelPipeline 靠前的位置,覆盖 channelActiveO 方法,创建 TCP 链路后,执行流控逻辑,如果达到流控阈值,则拒绝该连接,调用 ChannelHandler Context 的 close(方法关闭连接。

JVM 层面相关性能优化

当客户端的并发连接数达到数十万或者数百万时,系统一个较小的抖动就会导致很严重的后果,例如服务端的 GC,导致应用暂停(STW)的 GC 持续几秒,就会导致海量的客户端设备掉线或者消息积压,一旦系统恢复,会有海量的设备接入或者海量的数据发送很可能瞬间就把服务端冲垮。

JVM 层面的调优主要涉及 GC 参数优化,GC 参数设置不当会导致频繁 GC,甚至 OOM 异常,对服务端的稳定运行产生重大影响。

确定 GC 优化目标

GC(垃圾收集)有三个主要指标。

(1)吞吐量:是评价 GC 能力的重要指标,在不考虑 GC 引起的停顿时间或内存消耗时,吞吐量是 GC 能支撑应用程序达到的最高性能指标。

(2)延迟:GC 能力的最重要指标之一,是由于 GC 引起的停顿时间,优化目标是缩短延迟时间或完全消除停顿(STW),避免应用程序在运行过程中发生抖动。

(3)内存占用:GC 正常时占用的内存量。

JVM GC 调优的三个基本原则如下。

(1) Minor go 回收原则:每次新生代 GC 回收尽可能多的内存,减少应用程序发生 Full gc 的频率。

2)GC 内存最大化原则:垃圾收集器能够使用的内存越大,垃圾收集效率越高,应用程序运行也越流畅。但是过大的内存一次 Full go 耗时可能较长,如果能够有效避免 FullGC,就需要做精细化调优。

(3)3 选 2 原则:吞吐量、延迟和内存占用不能兼得,无法同时做到吞吐量和暂停时间都最优,需要根据业务场景做选择。对于大多数应用,吞吐量优先,其次是延迟。当然对于时延敏感型的业务,需要调整次序。

2.确定服务端内存占用

在优化 GC 之前,需要确定应用程序的内存占用大小,以便为应用程序设置合适的内存,提升 GC 效率。内存占用与活跃数据有关,活跃数据指的是应用程序稳定运行时长时间存活的

Java 对象。活跃数据的计算方式:通过 GC 日志采集 GC 数据,获取应用程序稳定时老年代占用的 Java 堆大小,以及永久代(元数据区)占用的 Java 堆大小,两者之和就是活跃数据的内存占用大小。

GC 优化过程

1、GC 数据的采集和研读
2、设置合适的 JVM 堆大小
3、选择合适的垃圾回收器和回收策略

当然具体如何做，请参考 JVM 相关课程。而且 GC 调优会是一个需要多次调整的过程，期间不仅有参数的变化，更重要的是需要调整业务代码。

消息中间件面试题目

1. RabbitMQ 中的 broker 是指什么？cluster 又是指什么？

答：broker 是指一个或多个 erlang node 的逻辑分组，且 node 上运行着 RabbitMQ 应用程序。cluster 是在 broker 的基础之上，增加了 node 之间共享元数据的约束。

2. RabbitMQ 中 RAM node 和 disk node 的区别？

答：RAM node 仅将 fabric（即 queue、exchange 和 binding等 RabbitMQ基础构件）相关元数据保存到内存中，但 disk node 会在内存和磁盘中均进行存储。RAM node 上唯一会存储到磁盘上的元数据是 cluster 中使用的 disk node 的地址。要求在 RabbitMQ cluster 中至少存在一个 disk node 。

3. RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？

答：可以认为是无限制，因为限制取决于机器的内存，但是消息过多会导致处理效率的下降。

4. RabbitMQ 概念里的 channel、exchange 和 queue 这些东东是逻辑概念，还是对应着进程实体？它们分别起什么作用？

答：queue 具有自己的 erlang 进程；exchange 内部实现为保存 binding 关系的查找表；channel 是实际进行路由工作的实体，即负责按照 routing_key 将 message 投递给 queue 。由 AMQP 协议描述可知，channel 是真实 TCP 连接之上的虚拟连接，所有 AMQP 命令都是通过 channel 发送的，且每一个 channel 有唯一的 ID。一个 channel 只能被单独一个操作系统线程使用，故投递到特定 channel 上的 message 是有顺序的。但一个操作系统线程上允许使用多个 channel 。channel 号为 0 的 channel 用于处理所有对于当前 connection 全局有效的帧，而 1-65535 号 channel 用于处理和特定 channel 相关的帧。AMQP 协议给出的 channel 复用模型如下

其中每一个 channel 运行在一个独立的线程上，多线程共享同一个 socket。

5. RabbitMQ 中 vhost 是什么？起什么作用？

答：vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

6. 在单 node 系统和多 node 构成的 cluster 系统中声明 queue、exchange ，以及进行 binding 会有什么不同？

答：当你在单 node 上声明 queue 时，只要该 node 上相关元数据进行了变更，你就会得到 Queue.Declare-ok 回应；而在 cluster 上声明 queue ，则要求 cluster 上的全部 node 都要进行元数据成功更新，才会得到 Queue.Declare-ok 回应。另外，若 node 类型为 RAM node 则变更的数据仅保存在内存中，若类型为 disk node 则还要变更保存在磁盘上的数据。

7. 客户端连接到 cluster 中的任意 node 上是否都能正常工作？

答：是的。客户端感觉不到有何不同。

8. 能够在地理上分开的不同数据中心使用 RabbitMQ cluster 么？

答：不能。第一，你无法控制所创建的 queue 实际分布在 cluster 里的哪个 node 上（一般使用 HAProxy + cluster 模型时都是这样），这可能会导致各种跨地域访问时的常见问题；第二，Erlang 的 OTP 通信框架对延迟的容忍度有限，这可能会触发各种超时，导致业务疲于处理；第三，在广域网上的连接失效问题将导致经典的“脑裂”问题，而 RabbitMQ 目前无法处理（该问题主要是说 Mnesia）。

9. routing_key 和 binding_key 的最大长度是多少？

答：255 字节。

10. RabbitMQ 中 “dead letter”queue 有什么用途？

答：当消息被 RabbitMQ server 投递到 consumer 后，但 consumer 却通过 Basic.Reject 进行了拒绝时（同时设置 requeue=false），那么该消息会被放入“dead letter”queue 中。该 queue 可用于排查 message 被 reject 或 undeliver 的原因。

11. RabbitMQ 中 Basic.Reject 的用法是什么？

答：该信令可用于 consumer 对收到的 message 进行 reject 。若在该信令中设置 requeue=true，则当 RabbitMQ server 收到该拒绝信令后，会将该 message 重新发送到下一个处于消费状态的消费者处（理论上仍可能将该消息发送给当前 consumer）。若设置 requeue=false ，则 RabbitMQ server 在收到拒绝信令后，将直接将该 message 从 queue 中移除。

而 Basic.Nack 是对 Basic.Reject 的扩展，以支持一次拒绝多条 message 的能力。

12. 为什么不应该对所有的 message 都使用持久化机制？

答：首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现问题。

矛盾点在于，若消息设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的所属节点出现异常后，在未重建该queue前，发往该 queue 的 消息将被 blackholed；若 消息设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的所属节点异常且无法重启的情况下，则该 queue 无法在其他节点上重建，只能等待其所属节点重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。

所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务器），则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。

另外一种处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

13. RabbitMQ 中的 cluster、mirrored queue 机制分别用于解决什么问题？存在哪些问题？

答：cluster 是为了解决当 cluster 中的任意 node 失效后，producer 和 consumer 均可以通过其他 node 继续工作，即提高了可用性；另外可以通过增加 node 数量增加 cluster 的消息吞吐量的目的。cluster 本身不负责 message 的可靠性问题（该问题由 producer 通过各种机制自行解决）；cluster 无法解决跨数据中心的问题（即脑裂问题）。另外，在cluster 前使用 HAProxy 可以解决 node 的选择问题，即业务无需知道 cluster 中多个 node 的 ip 地址。可以利用 HAProxy 进行失效 node 的探测，可以作负载均衡。下图为 HAProxy + cluster 的模型。

Mirrored queue 是为了解决使用 cluster 时所创建的 queue 的完整信息仅存在于单一 node 上的问题，从另一个角度增加可用性。

14. 请概要说明 Kafka 的体系结构

Kafka将消息以topic为单位进行归纳
将向Kafka topic发布消息的程序成为producers.
将预订topics并消费消息的程序成为consumer.
Kafka以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个broker.
producers通过网络将消息发送到Kafka集群，集群向消费者提供消息

15. Kafa consumer 是否可以消费指定分区消息？

Kafa consumer消费消息时，向broker发出"fetch"请求去消费特定分区的消息，consumer指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息，customer拥有了offset的控制权，可以向后回滚去重新消费之前的消息。

16. Kafka 消息是采用 Pull 模式，还是 Push 模式？

Kafka最初考虑的问题是，customer应该从brokes拉取消息还是brokers将消息推送到consumer，也就是pull还push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer从broker拉取消息

采用push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式。

Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略

Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到达。为了避免这点，Kafka有个参数可以让consumer阻塞知道新消息到达。

17. Kafka 高效文件存储设计特点：

(1).Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

(2).通过索引信息可以快速定位message和确定response的最大大小。

(3).通过index元数据全部映射到memory，可以避免segment file的IO磁盘操作。

(4).通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。

18. Kafka 与传统消息系统之间有三个关键区别

(1).Kafka 持久化日志，这些日志可以被重复读取和无限期保留

(2).Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

(3).Kafka 支持实时的流式处理

19. Kafka 里 partition 的数据如何保存到硬盘

topic中的多个partition以文件夹的形式保存到broker，每个分区序号从0递增，且消息有序 。

Partition文件下有多个segment（xxx.index，xxx.log）

segment 文件里的 大小和配置文件大小一致可以根据要求修改 默认为1g

如果大小大于1g时，会滚动一个新的segment并且以上一个segment最后一条消息的偏移量命名

20. 请概述 kafka 的 ack 机制

request.required.acks有三个值 0、 1、 -1

0:生产者不会等待broker的ack，这个延迟最低但是存储的保证最弱当server挂掉的时候就会丢数据

1:服务端会等待ack值 leader副本确认接收到消息后发送ack但是如果leader挂掉后他

不确保是否复制完成新leader也会导致数据丢失

-1：同样在1的基础上 服务端会等所有的follower的副本受到数据后才会受到leader发出的ack，这样数据不会丢失

21. kafaka 生产数据时数据的分组策略

生产者决定数据产生到集群的哪个partition中，每一条消息都是以（key，value）格式，Key是由生产者发送数据传入。所以生产者（key）决定了数据产生到集群的哪个partition。

22. 数据传输的事务定义有哪三种？

数据传输的事务定义通常有以下三种级别：

（1）最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输

（2）最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

（3）精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

23. 为什么使用消息队列？

其实就是问问你消息队列都有哪些使用场景，然后你项目里具体是什么场景，说说你在这个场景里用消息队列是什么？

面试官问你这个问题，期望的一个回答是说，你们公司有个什么业务场景，这个业务场景有个什么技术挑战，如果不用 MQ 可能会很麻烦，但是你现在用了 MQ 之后带给了你很多的好处。消息队列的常见使用场景，其实场景有很多，但是比较核心的有 3 个：解耦、异步、削峰。

解耦：

A 系统发送个数据到 BCD 三个系统，接口调用发送，那如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？现在 A 系统又要发送第二种数据了呢？而且 A 系统要时时刻刻考虑 BCDE 四个系统如果挂了咋办？要不要重发？我要不要把消息存起来？

你需要去考虑一下你负责的系统中是否有类似的场景，就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给他异步化解耦，也是可以的，你就需要去考虑在你的项目里，是不是可以运用这个 MQ 去进行系统的解耦。

异步：

A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 30ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 30 + 300 + 450 + 200 = 980ms，接近 1s，异步后，BCD 三个系统分别写库的时间，A 系统就不再考虑了。

削峰：

每天 0 点到 16 点，A 系统风平浪静，每秒并发请求数量就 100 个。结果每次一到 16 点~23 点，每秒并发请求数量突然会暴增到 1 万条。但是系统最大的处理能力就只能是每秒钟处理 1000 个请求啊。怎么办？需要我们进行流量的削峰，让系统可以平缓的处理突增的请求。

24. 消息队列有什么优点和缺点?

优点上面已经说了，就是在特殊场景下有其对应的好处，解耦、异步、削峰。
缺点呢？

系统可用性降低

系统引入的外部依赖越多，越容易挂掉，本来你就是 A 系统调用 BCD 三个系统的接口就好了，ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了怎么办？MQ 挂了，整套系统崩溃了，业务也就停顿了。

系统复杂性提高

硬生生加个 MQ 进来，怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉。

25. 消息重复的原因和解决

第一类原因

消息发送端应用的消息重复发送,有以下几种情况。

- 消息发送端发送消息给消息中间件,消息中间件收到消息并成功存储,而这时消息中间件出现了问题,导致应用端没有收到消息发送成功的返回因而进行重试产生了重复。

- 消息中间件因为负载高响应变慢,成功把消息存储到消息存储中后,返回“成功”这个结果时超时。

- 消息中间件将消息成功写入消息存储,在返回结果时网络出现问题,导致应用发送端重试,而重试时网络恢复,由此导致重复。

可以看到,通过消息发送端产生消息重复的主要原因是消息成功进入消息存储后,因为各种原因使得消息发送端没有收到“成功”的返回结果,并且又有重试机制,因而导致重复。

第二类原因

消息到达了消息存储,由消息中间件进行向外的投递时产生重复，有以下几种情况。

消息被投递到消息接收者应用进行处理,处理完毕后应用出问题了,消息中间件不知道消息处理结果,会再次投递。

消息被投递到消息接收者应用进行处理,处理完毕后网络出现问题了,消息中间件没有收到消息处理结果,会再次投递。

消息被投递到消息接收者应用进行处理,处理时间比较长,消息中间件因为消息超时会再次投递。

消息被投递到消息接收者应用进行处理,处理完毕后消息中间件出问题了,没能收到消息结果并处理,会再次投递

消息被投递到消息接收者应用进行处理,处理完毕后消息中间件收到结果但是遇到消息存储故障,没能更新投递状态,会再次投递。

可以看到,在投递过程中产生的消息重复接收主要是因为消息接收者成功处理完消息后,消息中间件不能及时更新投递状态造成的。

如何解决重复消费

那么有什么办法可以解决呢?主要是要求消息接收者来处理这种重复的情况,也就是要求消息接收者的消息处理是幂等操作。

什么是幂等性?

对于消息接收端的情况,幂等的含义是采用同样的输入多次调用处理函数,得到同样的结果。例如，一个 SQL 操作

```
update stat_table set count= 10 where id =1
```

这个操作多次执行,id 等于 1 的记录中的 count 字段的值都为 10,这个操作就是幂等的,我们不用担心这个操作被重复。

再来看另外一个 SQL 操作

update stat_table set count= count +1 where id= 1;

这样的 SQL 操作就不是幂等的,一旦重复,结果就会产生变化。

常见办法

因此应对消息重复的办法是,使消息接收端的处理是一个幂等操作。这样的做法降低了消息中间件的整体复杂性,不过也给使用消息中间件的消息接收端应用带来了一定的限制和门槛。

1. MVCC：

多版本并发控制，乐观锁的一种实现，在生产者发送消息时进行数据更新时需要带上数据的版本号,消费者去更新时需要去比较持有数据的版本号,版本号不一致的操作无法成功。例如博客点赞次数自动+1 的接口：

public boolean addCount(Long id, Long version);

update blogTable set count= count+1,version=version+1 where id=321 and version=123

每一个 version 只有一次执行成功的机会,一旦失败了生产者必须重新获取数据的最新版本号再次发起更新。

2. 去重表：

利用数据库表单的特性来实现幂等，常用的一个思路是在表上构建唯一性索引，保证某一类数据一旦执行完毕，后续同样的请求不再重复处理了（利用一张日志表来记录已经处理成功的消息的 ID，如果新到的消息 ID 已经在日志表中，那么就不再处理这条消息。）

以电商平台为例子，电商平台上的订单 id 就是最适合的 token。当用户下单时，会经历多个环节，比如生成订单，减库存，减优惠券等等。每一个环节执行时都先检测一下该订单 id 是否已经执行过这一步骤，对未执行的请求，执行操作并缓存结果，而对已经执行过的 id，则直接返回之前的执行结果，不做任何操作。这样可以在最大程度上避免操作的重复执行问题，缓存起来的执行结果也能用于事务的控制等。

26. 消息的可靠性传输

RabbitMQ

（1）生产者弄丢了数据

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启

RabbitMQ 事务（channel.txSelect），然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息;如果收到了消息,那么可以提交事务(channel.txCommit)。但是问题是,RabbitMQ 事务机制一搞，基本上吞吐量会下来，因为太耗性能。

所以一般来说，如果要确保 RabbitMQ 的消息别丢，可以开启 confirm 模式，在生产者那里设置开启 confirm 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中,RabbitMQ 会给你回传一个 ack 消息,告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你一个 nack 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和 cnofirm 机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是 confirm 机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用 confirm 机制的。

（2）RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须开启 RabbitMQ 的持久化，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建 queue 和交换器的时候将其设置为持久化的，这样就可以保证 RabbitMQ 持久化相关的元数据，但是不会持久化 queue 里的数据；第二个是发送消息的时候将消息的 deliveryMode 设置为 2，就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

而且持久化可以跟生产者那边的 confirm 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 ack 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 ack，你也是可以自己重发的。

哪怕是你给 RabbitMQ 开启了持久化机制,也有一种可能,就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据会丢失。

（3）消费端弄丢了数据

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 ack 机制，简单来说，就是你关闭 RabbitMQ 自动 ack，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再程序里 ack 一把。这样的话，如果你还没处理完，不就没有 ack？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。

Kafka

（1）消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了 offset，让 kafka 以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

大家都知道 kafka 会自动提交 offset，那么只要关闭自动提交 offset，在处理完之后自己手动提交 offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交 offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的 kafka 消费者消费到了数据之后是写到一个内存的 queue 里先缓冲一下，结果有的时候，你刚把消息写入内存 queue，然后消费者会自动提交 offset。

然后此时我们重启了系统，就会导致内存 queue 里还没来得及处理的数据就丢失了

（2）kafka 弄丢了数据

这块比较常见的一个场景，就是 kafka 某个 broker 宕机，然后重新选举 partiton 的 leader 时。大家想想，要是此时其他的 follower 刚好还有些数据没有同步，结果此时 leader 挂了，然后选举某个 follower 成 leader 之后，他不就少了一些数据？这就丢了一些数据啊。

所以此时一般是要求起码设置如下 4 个参数：

给这个 topic 设置 replication.factor 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。

在 kafka 服务端设置 min.insync.replicas 参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。

在 producer 端设置 acks=all：这个是要求每条数据，必须是写入所有 replica 之后，才能认为是写成功了。

在 producer 端设置 retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。

（3）生产者会不会弄丢数据

如果按照上述的思路设置了 ack=all，一定不会丢，要求是，你的 leader 接收到消息，所有的 follower 都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

27. 消息的顺序性

从根本上说，异步消息是不应该有顺序依赖的。在 MQ 上估计是没法解决。要实现严格的顺序消息，简单且可行的办法就是：保证生产者 - MQServer - 消费者是一对一对一的关系。

RabbitMQ

如果有顺序依赖的消息，要保证消息有一个 hashKey，类似于数据库表分区的的分区 key 列。保证对同一个 key 的消息发送到相同的队列。A 用户产生的消息（包括创建消息和删除消息）都按 A 的 hashKey 分发到同一个队列。只需要把强相关的两条消息基于相同的路由就行了，也就是说经过 m1 和 m2 的在路由表里的路由是一样的，那自然 m1 会优先于 m2 去投递。而且一个 queue 只对应一个 consumer。

Kafka

一个 topic，一个 partition，一个 consumer，内部单线程消费

ELK 面试题目

1.ELK 是什么？

ELK 其实并不是一款软件，而是一整套解决方案，是三个软件产品的首字母缩写
Elasticsearch：负责日志检索和储存
Logstash：负责日志的收集和分析、处理
Kibana：负责日志的可视化
这三款软件都是开源软件，通常是配合使用，而且又先后归于 Elastic.co 公司名下，故被简称为 ELK。加入 Beats 系列组件后，官方名称就变为了 Elastic Stack

2.ELK 能做什么？

ELK 组件在海量日志系统的运维中，可用于解决：
分布式日志数据集中式查询和管理、系统监控，包含系统硬件和应用各个组件的监控、故障排查、安全信息和事件管理、报表功能等等

3.简要概述 Elasticsearch？

ElasticSearch 是一个基于 Lucene 的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于 RESTful API 的 web 接口。
Elasticsearch 是用 Java 开发的，并作为 Apache 许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便

4.Elasticsearch 主要特点

1.实时分析
2.分布式实时文件存储，并将每一个字段都编入索引
3.文档导向，所有的对象全部是文档
4.高可用性，易扩展，支持集群（Cluster）、分片和复制（Shards 和 Replicas）
接口友好，支持 JSON

5.ES 相关概念

Node： 装有一个 ES 服务器的节点。
Cluster： 有多个 Node 组成的集群
Document： 一个可被搜素的基础信息单元
Index： 拥有相似特征的文档的集合

Type： 一个索引中可以定义一种或多种类型

Filed： 是 ES 的最小单位，相当于数据的某一列

Shards： 索引的分片，每一个分片就是一个 Shard

Replicas： 索引的拷贝

6.什么是分词器

分词是将文本转换成一系列单词（Term or Token）的过程，也可以叫文本分析，在 ES 里面称为 Analysis

分词器是 ES 中专门处理分词的组件，英文为 Analyzer，它的组成如下：

Character Filters：针对原始文本进行处理，比如去除 html 标签

Tokenizer：将原始文本按照一定规则切分为单词

Token Filters：针对 Tokenizer 处理的单词进行再加工，比如转小写、删除或增新等处理

ES 提供了一个可以测试分词的 API 接口，方便验证分词效果，endpoint 是_analyze

ES 也提供了很多内置的分析器。

7.elasticsearch 的倒排索引是什么？

正排索引是以文档的 ID 为关键字，表中记录文档中每个字的位置信息，查找时扫描表中每个文档中字的信息直到找出所有包含查询关键字的文档。

而倒排索引，是通过分词策略，形成了词和文章的映射关系表，这种词典+映射表即为倒排索引。

有了倒排索引，就能实现 o（1）时间复杂度的效率检索文章了，极大的提高了检索效率。

所以总的来说，正排索引是从文档到关键字的映射（已知文档求关键字），倒排索引是从关键字到文档的映射（已知关键字求文档）。

8.Elasticsearch 是如何实现 Master 选举的？

采用 Bully 算法，它假定所有节点都有一个唯一的 ID，使用该 ID 对节点进行排序。Elasticsearch 的选主是 ZenDiscovery 模块负责的，主要包含 Ping（节点之间通过这个 RPC 来发现彼此）和 Unicast（单播模块包含一个主机列表以控制哪些节点需要 ping 通）这两部分；

对所有可以成为 master 的节点（node.master: true）根据 nodeId 字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第 0 位）节点，暂且认为它是 master 节点。

如果对某个节点的投票数达到一定的值（可以成为 master 节点数 n/2+1）并且该节点自己也选举自己，那这个节点就是 master。否则重新选举一直到满足上述条件。

补充：master 节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；data 节点可以关闭 http 功能。

7.X 之后的 ES，采用一种新的选主算法，实际上是 Raft 的实现，但并非严格按照 Raft

论文实现，而是做了一些调整。Raft 是工程上使用较为广泛分布式共识协议，是多个节点对某个事情达成一致的看法，即使是在部分节点故障、网络延时、网络分区的情况下。

9.Elasticsearch 如何避免脑裂？

ES 集群中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办？

当集群 master 候选数量不小于 3 个时，可以通过设置最少投票通过数量（discovery.zen.minimum_master_nodes）超过所有候选节点一半以上来解决脑裂问题；

当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data 节点，避免脑裂问题。

在 Elasticsearch 7.0 里重新设计并重建了的集群协调子系统，移除 minimum_master_nodes 参数，转而由集群自主控制。

10.详细描述一下 Elasticsearch 索引文档的过程

协调节点默认使用文档 ID 参与计算（也支持通过 routing），以便为路由提供合适的分片。

shard = hash(document_id) % (num_of_primary_shards)

当分片所在的节点接收到来自协调节点的请求后，会将请求写入到 Memory Buffer，然后定时（默认是每隔 1 秒）写入到 Filesystem Cache，这个从 Momery Buffer 到 Filesystem Cache 的过程就叫做 refresh；

当然在某些情况下，存在 Momery Buffer 和 Filesystem Cache 的数据可能会丢失，ES 是通过 translog 的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到 translog 中，当 Filesystem cache 中的数据写入到磁盘中时，才会清除掉，这个过程叫做 flush；

在 flush 过程中，内存中的缓冲将被清除，内容被写入一个新段，段的 fsync 将创建一个新的提交点，并将内容刷新到磁盘，旧的 translog 将被删除并开始一个新的 translog。

flush 触发的时机是定时触发（默认 30 分钟）或者 translog 变得太大（默认为 512M）时；

11.请概述 Elasticsearch 搜索的过程？

搜索拆解为“query then fetch” 两个阶段。

query 阶段的目的：定位到位置，但不取。

步骤拆解如下：

1）假设一个索引数据有 5 主+1 副本 共 10 分片，一次请求会命中（主或者副本分片中）的一个。

2）每个分片在本地进行查询，结果返回到本地有序的优先队列中。

3）第 2）步骤的结果发送到协调节点，协调节点产生一个全局的排序列表。

fetch 阶段的目的：取数据。

路由节点获取所有文档，返回给客户端。

Redis 面试题目

1. 什么是 Redis？简述它的优缺点？

Redis 的全称是：Remote Dictionary.Server，本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘进行保存。因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，因此 Redis 可以用来实现很多有用的功能。 比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性 能消息队列服务，用他的 Set 可以做高性能的 tag 系统等等。

另外 Redis 也可以对存入的 Key-Value 设置 expire 时间，因此也可以被当作一 个功能加强版的

memcached 来用。 Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

2. Redis 与 memcached 相比有哪些优势？

1.memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类型

2.redis 的速度比 memcached 快很多 redis 的速度比 memcached 快很多

3.redis 可以持久化其数据 redis 可以持久化其数据

3. Redis 支持哪几种数据类型？

String、List、Set、Sorted Set、hash

4. Redis 主要消耗什么物理资源？

内存。

5. Redis 有哪几种数据淘汰策略？

1.noeviction:返回错误当内存限制达到，并且客户端尝试执行会让更多内存被使用的命令。

2.allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

3.volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

4.allkeys-random: 回收随机的键使得新添加的数据有空间存放。

5.volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

6.volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

6. Redis 官方为什么不提供 Windows 版本？

因为目前 Linux 版本已经相当稳定，且 Linux 操作系统自带的 epoll 相关函数，在高并发情况下性能一般比 windows 的 select 函数性能较好，为了高性能起见，Redis 官网不提供 windows 版本。

7. 一个字符串类型的值能存储最大容量是多少？

512M

8. 为什么 Redis 需要把所有数据放到内存中？

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。

所以 redis 具有快速和数据持久化的特征，如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。

在内存越来越便宜的今天，redis 将会越来越受欢迎， 如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

9. Redis 集群方案应该怎么做？都有哪些方案？

1.codis2.目前用的最多的集群方案，基本和 twemproxy 一致的效果，但它支持在节点数量改变情况下，旧节点数据可恢复到新 hash 节点。

2.redis cluster3.0 自带的集群，特点在于他的分布式算法不是一致性 hash，而是 hash 槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

3.在业务代码层实现，起几个毫无关联的 redis 实例，在代码层，对 key 进行 hash 计算，然后去对应的 redis 实例操作数据。这种方式对 hash 层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

10. Redis 集群方案什么情况下会导致整个集群不可用？

有 A，B，C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了，那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。

11. MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？

redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

12. Redis 有哪些适合的场景？

（1）会话缓存（Session Cache）

最常用的一种使用 Redis 的情景是会话缓存（session cache），用 Redis 缓存会话比其他存储（如

Memcached）的优势在于：Redis 提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的商业平台 Magento 也提供 Redis 的插件。

（2）全页缓存（FPC）

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。

再次以 Magento 为例，Magento 提供一个插件来使用 Redis 作为全页缓存后端。

此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）队列

Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。

如果你快速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。

排行榜/计数器 Redis 在内存中对数字进行递增或递减的操作实现的非常好。

集合（Set）和有序集合（SortedSet）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。

所以，我们要从排序集合中获取到排名最靠前的 10 个用户 - 我们称之为“user_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user_scores 0 10 WITHSCORES

Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

（5）发布/订阅

最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！

13. Redis 支持的 Java 客户端都有哪些？官方推荐用哪个？

Redisson、Jedis、lettuce 等等，官方推荐使用 Redisson。

14. Redis 和 Redisson 有什么关系？

Redisson 是一个高级的分布式协调 Redis 客服端，能帮助用户在分布式环境中轻松实现一些 Java 的对象 (Bloom filter, BitSet, Set, SetMultimap, ScoredSortedSet, SortedSet, Map, ConcurrentMap,

List, ListMultimap, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock,

ReadWriteLock, AtomicLong, CountDownLatch, Publish / Subscribe, HyperLogLog)。

15. Jedis 与 Redisson 对比有什么优缺点？

Jedis 是 Redis 的 Java 实现的客户端，其 API 提供了比较全面的 Redis 命令的支持；

Redisson 实现了分布式和可扩展的 Java 数据结构，和 Jedis 相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等 Redis 特性。Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

16. 说说 Redis 哈希槽的概念？

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

17. Redis 集群的主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型, 每个节点都会有 N-1 个复制品。

18. Redis 集群会有写操作丢失吗？为什么？

Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

19. Redis 集群之间是如何复制的？

异步复制

20. Redis 集群最大节点个数是多少？

16384 个

21. Redis 集群如何选择数据库？

Redis 集群目前无法做数据库选择，默认在 0 数据库。

22. Redis 中的管道有什么用？

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应，这样就可以将多个命令发送到服务 器，而不用等待回复，最后在一个步骤中读取该答复。

这就是管道（pipelining），是一种几十年来广泛使用的技术。例如许多 POP3 协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。23、怎么理解 Redis 事务？

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行，事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

23. Redis 事务相关的命令有哪几个？

MULTI、EXEC、DISCARD、WATCH

24. Redis key 的过期时间和永久有效分别怎么设置？

EXPIRE 和 PERSIST 命令

25. Redis 如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面。

26. Redis 回收进程如何工作的？

一个客户端运行了新的命令，添加了新的数据。

Redi 检查内存使用情况，如果大于 maxmemory 的限制，则根据设定好的策略进行回收。

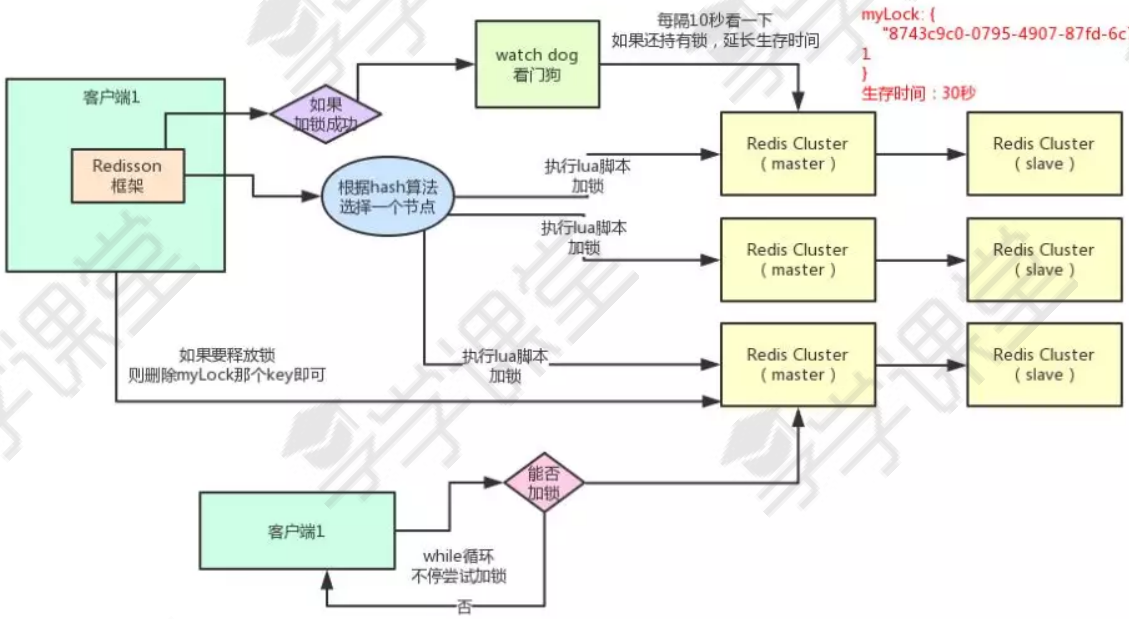
一个新的命令被执行，等等。

所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。

如果一个命令的结果导致大量内存被使用(例如很大的集合的交集保存到一个新的键)，不用多久内存限制就会被这个内存使用量超越。

27. 加锁机制

我们先看看 Redisson 这个开源框架对 Redis 分布式锁的实现原理



咱们来看上面那张图，现在某个客户端要加锁。如果该客户端面对的是一个 redis cluster 集群，他首先会根据 hash 节点选择一台机器。这里注意，仅仅只是选择一台机器！这点很关键！紧接着，就会发送一段 lua 脚本到 redis 上，那段 lua 脚本如下所示：

为啥要用 lua 脚本呢？因为一大坨复杂的业务逻辑，可以通过封装在 lua 脚本中发送给 redis，保证这段复杂业务逻辑执行的原子性。

那么，这段 lua 脚本是什么意思呢？这里 KEYS[1]代表的是你加锁的那个 key，比如说：RLock lock = redisson.getLock("myLock");这里你自己设置了加锁的那个锁 key 就是“myLock”。

ARGV[1]代表的就是锁 key 的默认生存时间，默认 30 秒。ARGV[2]代表的是加锁的客户端的 ID，类似于下面这样：8743c9c0-0795-4907-87fd-6c719a6b4586:1

给大家解释一下，第一段 if 判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁 key 不存在的话，你就进行加锁。如何加锁呢？很简单，用下面的命令：hset myLock8743c9c0-0795-4907-87fd-6c719a6b4586:1 1，通过这个命令设置一个 hash 数据结构，这行命令执行后，会出现一个类似下面的数据结构：

上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这

个锁 key 完成了加锁。接着会执行“pexpire myLock 30000”命令，设置 myLock 这个锁 key 的生存时间是 30 秒。好了，到此为止，ok，加锁完成了。

28. 锁互斥机制

那么在这个时候，如果客户端 2 来尝试加锁，执行了同样的一段 lua 脚本，会咋样呢？很简单，第一个 if 判断会执行“exists myLock”，发现 myLock 这个锁 key 已经存在了。接着第二个 if 判断，判断一下，myLock 锁 key 的 hash 数据结构中，是否包含客户端 2 的 ID，但是明显不是的，因为那里包含的是客户端 1 的 ID。

所以，客户端 2 会获取到 pttl myLock 返回的一个数字，这个数字代表了 myLock 这个锁 key 的剩余生存时间。比如还剩 15000 毫秒的生存时间。此时客户端 2 会进入一个 while 循环，不停的尝试加锁。

29. watch dog 自动延期机制

客户端 1 加锁的锁 key 默认生存时间才 30 秒，如果超过了 30 秒，客户端 1 还想一直持有这把锁，怎么办呢？

简单！只要客户端 1 一旦加锁成功，就会启动一个 watch dog 看门狗，他是一个后台线程，会每隔 10 秒检查一下，如果客户端 1 还持有锁 key，那么就会不断的延长锁 key 的生存时间。

30. 可重入加锁机制

那如果客户端 1 都已经持有了这把锁了，结果可重入的加锁会怎么样呢？比如下面这种代码：这时我们来分析一下上面那段 lua 脚本。第一个 if 判断肯定不成立，“exists myLock”会显示锁 key 已经存在了。第二个 if 判断会成立，因为 myLock 的 hash 数据结构中包含的那个 ID，就是客户端 1 的那个 ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”此时就会执行可重入加锁的逻辑，他会用：incrby myLock 8743c9c0-0795-4907-87fd-6c71a6b4586:1 1 ，通过这个命令，对客户端 1 的加锁次数，累加 1。此时 myLock 数据结构变为下面这样：大家看到了吧，那个 myLock 的 hash 数据结构中的那个客户端 ID，就对应着加锁的次数

31. 释放锁机制

如果执行 lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。其实说白了，就是每次都对 myLock 数据结构中的那个加锁次数减 1。如果发现加锁次数是 0 了，说明这个客户端已经不再持有锁了，此时就会用：“del myLock”命令，从 redis 里删除这个 key。

然后呢，另外的客户端 2 就可以尝试完成加锁了。这就是所谓的分布式锁的开源 Redisson 框架的实现机制。

一般我们在生产系统中，可以用 Redisson 框架提供的这个类库来基于 redis 进行分布式锁的加锁与释放锁。

32. 上述 Redis 分布式锁的缺点

其实上面那种方案最大的问题，就是如果你对某个 redis master 实例，写入了 myLock 这种锁 key 的 value，此时会异步复制给对应的 master slave 实例。但是这个过程中一旦发生 redis master 宕机，主备切换，redis slave 变为了 redis master。

接着就会导致，客户端 2 来尝试加锁的时候，在新的 redis master 上完成了加锁，而客户端 1 也以为自己成功加了锁。此时就会导致多个客户端对一个分布式锁完成了加锁。这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是 redis cluster，或者是 redis master-slave 架构的主从异步复制导致的 redis 分布式锁的最大缺陷：在 redis master 实例宕机的时候，可能导致多个客户端同时完成加锁。

33. 使用过 Redis 分布式锁么，它是怎么实现的？

先拿 setnx 来争抢锁，抢到之后，再用 expire 给锁加一个过期时间防止锁忘记了释放。如果在 setnx 之后执行 expire 之前进程意外 crash 或者要重启维护了，那会怎么样？set 指令有非常复杂的参数，这个应该是可以同时把 setnx 和 expire 合成一条指令来用的！

34. 使用过 Redis 做异步队列么，你是怎么用的？有什么缺点？

一般使用 list 结构作为队列，rpush 生产消息，lpop 消费消息。当 lpop 没有消息的时候，要适当 sleep 一会再重试。

缺点：

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如 rabbitmq 等。

能不能生产一次消费多次呢？

使用 pub/sub 主题订阅者模式，可以实现 1:N 的消息队列。

35. redis 和 memcached 什么区别？为什么高并发下有时单线程的 redis 比多线程的 memcached 效率要高？

区别：

1.mc 可缓存图片和视频。rd 支持除 k/v 更多的数据结构;

2.rd 可以使用虚拟内存，rd 可持久化和 aof 灾难恢复，rd 通过主从支持数据备份;

3.rd 可以做消息队列。

原因：mc 多线程模型引入了缓存一致性和锁，加锁带来了性能损耗。

36. redis 主从复制如何实现的？redis 的集群模式如何实现？redis 的 key 是如何寻址的？

主从复制实现：主节点将自己内存中的数据做一份快照，将快照发给从节点，从节点将数据恢复到内存中。之后再每次增加新数据的时候，主节点以类似于 mysql 的二进制日志方式将语句发送给从节点，从节点拿到主节点发送过来的语句进行重放。

分片方式：

-客户端分片

-基于代理的分片

- Twemproxy
- codis

-路由查询分片

- Redis-cluster（本身提供了自动将数据分散到 Redis Cluster 不同节点的能力，整个数据集合的某个数据子集存储在哪个节点对于用户来说是透明的)redis-cluster 分片原理：Cluster 中有一个 16384 长度的槽(虚拟槽)，编号分别为 0-16383。

每个 Master 节点都会负责一部分的槽，当有某个 key 被映射到某个 Master 负责的槽，那么这个 Master 负责为这个 key 提供服务，至于哪个 Master 节点负责哪个槽，可以由用户指定，也可以在初始化的时候自动生成，只有 Master 才拥有槽的所有权。Master 节点维护着一个 16384/8 字节的位序列，Master 节点用 bit 来标识对于某个槽自己是否拥有。比如对于编号为 1 的槽，Master 只要判断序列的第二位（索引从 0 开始）是不是为 1 即可。

这种结构很容易添加或者删除节点。比如如果我想新添加个节点 D，我需要从节点 A、B、C 中得部分槽到 D 上。

37. 使用 redis 如何设计分布式锁？说一下实现思路？使用 zk 可以吗？如何实现？这两种有什么区别？

Redis:

1.线程 A setnx(上锁的对象,超时时的时间戳 t1)，如果返回 true，获得锁。

2.线程 B 用 get 获取 t1,与当前时间戳比较,判断是是否超时,没超时 false,若超时执行第 3 步;

3.计算新的超时时间 t2,使用 getset 命令返回 t3(该值可能其他线程已经修改过)，如果

t1==t3，获得锁，如果 t1!=t3 说明锁被其他线程获取了。

4.获取锁后，处理完业务逻辑，再去判断锁是否超时，如果没超时删除锁，如果已超时，不用处理（防止删除其他线程的锁）。

Zookeeper:

1.客户端对某个方法加锁时，在 zk 上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点 node1;

2.客户端获取该路径下所有已经创建的子节点，如果发现自己创建的 node1 的序号是最小的，就认为这个客户端获得了锁。

3.如果发现 node1 不是最小的，则监听比自己创建节点序号小的最大的节点，进入等待。4.获取锁后，处理完逻辑，删除自己创建的 node1 即可。

区别： zk 性能差一些，开销大，实现简单。

38. 知道 redis 的持久化吗？底层如何实现的？有什么优点缺点？

RDB(Redis DataBase:在不同的时间点将 redis 的数据生成的快照同步到磁盘等介质上):内存到硬盘的快照，定期更新。缺点：耗时，耗性能(fork+io 操作)，易丢失数据。

AOF(Append Only File：将 redis 所执行过的所有指令都记录下来，在下次 redis 重启时，只需要执行指令就可以了):写日志。缺点：体积大，恢复速度慢。

bgsave 做镜像全量持久化，aof 做增量持久化。因为 bgsave 会消耗比较长的时间，不够实时，在停机的时候会导致大量的数据丢失，需要 aof 来配合，在 redis 实例重启时，优先使用 aof 来恢复内存的状态，如果没有 aof 日志，就会使用 rdb 文件来恢复。Redis 会定期做 aof 重写，压缩 aof 文件日志大小。Redis4.0 之后有了混合持久化的功能，将 bgsave 的全量和 aof 的增量做了融合处理,这样既保证了恢复的效率又兼顾了数据的安全性。bgsave 的原理,fork 和 cow,fork 是指 redis 通过创建子进程来进行 bgsave 操作，cow 指的是 copy onwrite，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

39. redis 过期策略都有哪些？LRU 算法知道吗？写一下 Java 代码实现？

过期策略:

定时过期(一 key 一定时器)，惰性过期：只有使用 key 时才判断 key 是否已过期，过期则清除。定期过期：前两者折中。

LRU:new LinkedHashMap<K, V>(capacity, DEFAULT_LOAD_FACTORY, true); //第三个参数置为 true，代表 linkedlist 按访问顺序排序，可作为 LRU 缓存；设为 false 代表按插入顺序排序，可作为 FIFO 缓存 LRU 算法实现：

1.通过双向链表来实现，新数据插入到链表头部；

2.每当缓存命中（即缓存数据被访问），则将数据移到链表头部；3.当链表满的时候，将链表尾部的数据丢弃。

LinkedHashMap：HashMap 和双向链表合二为一即是 LinkedHashMap。HashMap 是无序的，LinkedHashMap 通过维护一个额外的双向链表保证了迭代顺序。该迭代顺序可以是插入顺序（默认），也可以是访问顺序。

40. 缓存穿透、缓存击穿、缓存雪崩解决方案？

什么是缓存穿透?

指查询一个一定不存在的数据，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到 DB 去查询，可能导致 DB 挂掉。

穿透解决方案如下：

1.查询返回的数据为空，仍把这个空结果进行缓存，但过期时间会比较短；

2.布隆过滤器：将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对 DB 的查询。

什么是缓存击穿?

对于设置了过期时间的 key，缓存在某个时间点过期的时候，恰好这时间点对这个 Key 有大量的并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把 DB 压垮。

击穿解决方案如下:

1.使用互斥锁：当缓存失效时，不立即去 load db，先使用如 Redis 的 setnx 去设置一个互斥锁，当操作成功返回时再进行 load db 的操作并回设缓存，否则重试 get 缓存的方法。

2.永远不过期：物理不过期，但逻辑过期（后台异步线程去刷新）。

什么是缓存雪崩?

设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到 DB，DB 瞬时压力过重雪崩。与缓存击穿的区别：雪崩是很多 key，击穿是某一个 key 缓存。

雪崩解决方案如下:

将缓存失效时间分散开，比如可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

41. 在选择缓存时，什么时候选择 redis，什么时候选择 memcached

选择 redis 的情况：

1、复杂数据结构，value 的数据是哈希，列表，集合，有序集合等这种情况下，会选择 redis, 因为 memcache 无法满足这些数据结构，最典型的的使用场景是，用户订单列表，用户消息，帖子评论等。

2、需要进行数据的持久化功能，但是注意，不要把 redis 当成数据库使用，如果 redis 挂了，内存能够快速恢复热数据，不会将压力瞬间压在数据库上，没有 cache 预热的过程。对于只读和数据一致性要求不高的场景可以采用持久化存储

3、高可用，redis 支持集群，可以实现主动复制，读写分离，而对于 memcache 如果想要实现高可用，需要进行二次开发。

4、存储的内容比较大，memcache 存储的 value 最大为 1M .选择 memcache 的场景：

当纯 KV,数据量非常大的业务，使用 memcache 更合适，原因是如下:

a)memcache 的内存分配采用的是预分配内存池的管理方式，能够省去内存分配的时间，redis 是临时申请空间，可能导致碎片化。

b)虚拟内存使用，memcache 将所有的数据存储在物理内存里，redis 有自己的 vm 机制，理论上能够存储比物理内存更多的数据，当数据超量时，引发 swap,把冷数据刷新到磁盘上，从这点上，数据量大时，memcache 更快

c)网络模型，memcache 使用非阻塞的 IO 复用模型，redis 也是使用非阻塞的 IO 复用模型，但是 redis 还提供了一些非 KV 存储之外的排序，聚合功能，复杂的 CPU 计算，会阻塞整个 IO 调度，从这点上由于 redis 提供的功能较多，memcache 更快些

d) 线程模型，memcache 使用多线程，主线程监听，worker 子线程接受请求，执行读写，这个过程可能存在锁冲突。redis 使用的单线程，虽然无锁冲突，但是难以利用多核的特性提升吞吐量。

42. 缓存与数据库不一致怎么办

假设采用的主存分离，读写分离的数据库，

如果一个线程 A 先删除缓存数据，然后将数据写入到主库当中，这个时候，主库和从库同步没有完成，线程 B 从缓存当中读取数据失败，从从库当中读取到旧数据，然后更新至缓存，这个时候，缓存当中的就是旧的数据。

发生上述不一致的原因在于，主从库数据不一致问题，加入了缓存之后，主从不一致的时间被拉长了

处理思路：

在从库有数据更新之后，将缓存当中的数据也同时进行更新，即当从库发生了数据更新之后，向缓存发出删除，淘汰这段时间写入的旧数据。

43. 主从数据库不一致如何解决

场景描述，对于主从库，读写分离，如果主从库更新同步有时差，就会导致主从库数据的不一致

忽略这个数据不一致，在数据一致性要求不高的业务下，未必需要时时一致性

强制读主库，使用一个高可用的主库，数据库读写都在主库，添加一个缓存，提升数据读取的性能。

选择性读主库，添加一个缓存，用来记录必须读主库的数据，将哪个库，哪个表，哪个主键，作为缓存的 key,设置缓存失效的时间为主从库同步的时间，如果缓存当中有这个数据，直接读取主库，如果缓存当中没有这个主键，就到对应的从库中读取。

44. Redis 常见的性能问题和解决方案

1、master 最好不要做持久化工作，如 RDB 内存快照和 AOF 日志文件

2、如果数据比较重要，某个 slave 开启 AOF 备份，策略设置成每秒同步一次

3、为了主从复制的速度和连接的稳定性，master 和 Slave 最好在一个局域网内

4、尽量避免在压力大得主库上增加从库

5、主从复制不要采用网状结构，尽量是线性结构，Master<--Slave1<----Slave2....

45. Redis 的数据淘汰策略有哪些

voltile-lru 从已经设置过期时间的数据集中挑选最近最少使用的数据淘汰
voltile-ttl 从已经设置过期时间的数据库集当中挑选将要过期的数据
voltile-random 从已经设置过期时间的数据集任意选择淘汰数据
allkeys-lru 从数据集中挑选最近最少使用的数据淘汰
allkeys-random 从数据集中任意选择淘汰的数据
no-eviction 禁止驱逐数据

46. Redis 当中有哪些数据结构

字符串 String、字典 Hash、列表 List、集合 Set、有序集合 SortedSet。如果是高级用户，那么还会有，如果你是 Redis 中高级用户，还需要加上下面几种数据结构 HyperLogLog、Geo、Pub/Sub。

47. 假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用 keys 指令可以扫出指定模式的 key 列表。

对方接着追问：如果这个 redis 正在给线上的业务提供服务，那使用 keys 指令会有什么问题？

这个时候你要回答 redis 关键的一个特性：redis 的单线程的。keys 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 scan 指令，scan 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

48. 使用 Redis 做过异步队列吗，是如何实现的

使用 list 类型保存数据信息，rpush 生产消息，lpop 消费消息，当 lpop 没有消息时，可以 sleep 一段时间，然后再检查有没有信息，如果不想 sleep 的话，可以使用 blpop，在没有信息的时候，会一直阻塞，直到信息的到来。redis 可以通过 pub/sub 主题订阅模式实现一个生产者，多个消费者，当然也存在一定的缺点，当消费者下线时，生产的消息会丢失。

49. Redis 如何实现延时队列

使用 sortedset，使用时间戳做 score，消息内容作为 key，调用 zadd 来生产消息，消费者使用 zrangbyscore 获取 n 秒之前的数据做轮询处理。

Zookeeper 面试题目

1. ZooKeeper 是什么？

ZooKeeper 是一个*分布式*的，开放源码的分布式*应用程序协调服务*，是 Google 的 Chubby 一个开源的实现，它是*集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作*。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

客户端的*读请求*可以被集群中的*任意一台机器处理*，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于*写请求*，这些请求会同*时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功*。因此，随着 *zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降*。

有序性是 zookeeper 中非常重要的一个特性，所有的*更新都是全局有序的*，每个更新都有一个*唯一的时间戳*，这个时间戳称为 *zxid（Zookeeper Transaction Id）*。而*读请求只会相对于更新有序*，也就是读请求的返回结果中会带有这个 *zookeeper 最新的 zxid*。

2. ZooKeeper 提供了什么？

1、*文件系统*

2、*通知机制*

3. Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点*都可以设置关联的数据*，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper *不能用于存放大量的数据*，每个节点的存放数据上限为 *1M*。

4. 四种类型的 znode

1、*PERSISTENT-持久化目录节点*

客户端与 zookeeper 断开连接后，该节点依旧存在

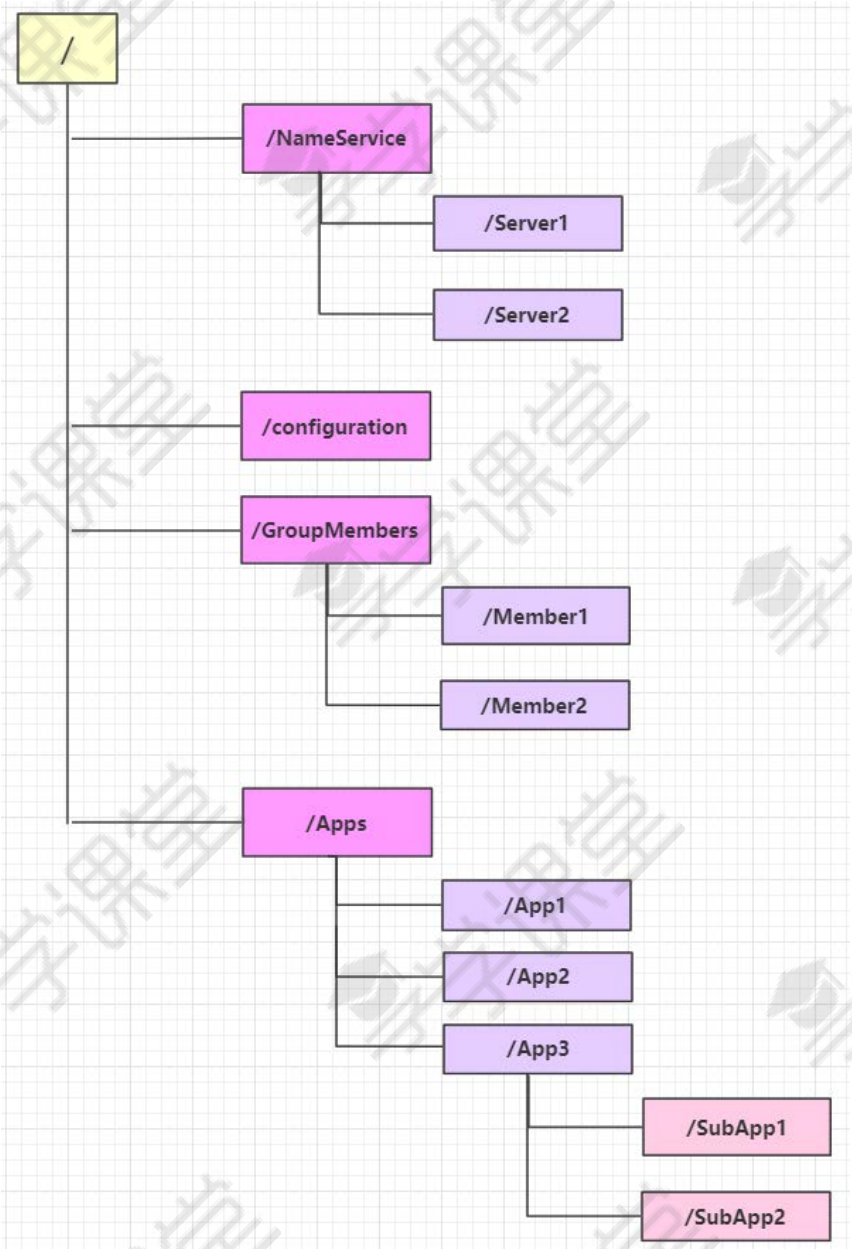
2、*PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点*

客户端与 zookeeper 断开连接后，该节点依旧存在，只是 Zookeeper 给该节点名称进行顺序编号

3、*EPHEMERAL-临时目录节点*

客户端与 zookeeper 断开连接后，该节点被删除

4、*EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点*客户端与 zookeeper 断开连接后，该节点被删除，只是 Zookeeper 给该节点名称进行顺序编号



5. Zookeeper 通知机制

client 端会对某个 znode 建立一个 *watcher 事件*，当该 znode 发生变化时，这些 client 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。

6. Zookeeper 做了什么？

1、命名服务
2、配置管理
3、集群管理
4、分布式锁
5、队列管理
7.zk 的命名服务（文件系统）

命名服务是指通过指定的名字来*获取资源*或者*服务的地址*，利用 zk 创建一个全局的路径，即是*唯*一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

7. zk 的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 *znode* 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 *watcher* 通知给各个客户端，从而更改配置。

8. Zookeeper 集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：*是否有机器退出和加入*、*选举 master*。

对于第一点，所有机器约定在父目录下*创建临时目录节点*，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，*所有其他机器都收到通知：某个兄弟目录被删除*，于是，所有人都知道：它上船了。

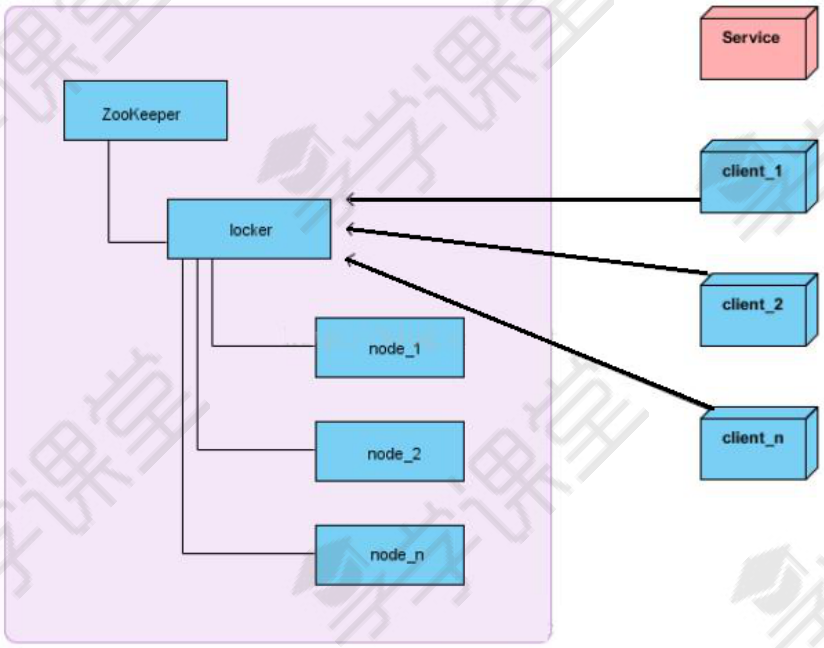
新机器加入也是类似，*所有机器收到通知：新兄弟目录加入*，highcount 又有了，对于第二点，我们稍微改变一下，*所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好*。

9. Zookeeper 分布式锁（文件系统、通知机制）

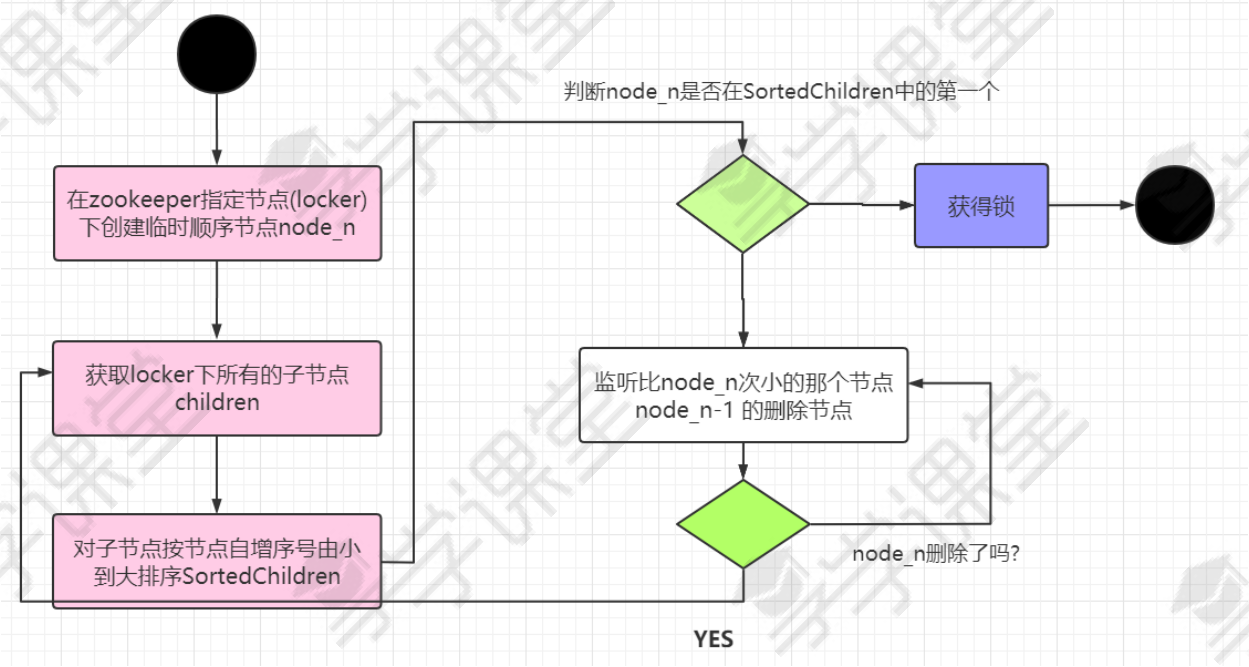
有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是*保持独占*，另一个是*控制时序*。

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。

10. 获取分布式锁的流程



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。



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。

11. Zookeeper 队列管理（文件系统、通知机制）

两种类型的队列：

1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下 Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

12. Zookeeper 数据复制 Zookeeper 作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处：

1、容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；

2、提高系统的扩展能力 ：把负载分布到多个节点上，或者增加节点来提高系统的负载能力；

3、提高性能：让*客户端本地访问就近的节点，提高用户访问速度*。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：

1、*写主*(WriteMaster) ：对数据的*修改提交给指定的节点*。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称*读写分离*;

2、*写任意*(Write Any)：对数据的*修改可提交给任意的节点*，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对 zookeeper 来说，它采用的方式是*写任意*。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立 observer 的原因），而响应能力则取决于具体实现方式，是*延迟复制保持最终一致性*，还是*立即复制快速响应*。

13. Zookeeper 工作原理

Zookeeper 的核心是*原子广播*，这个机制保证了*各个 Server 之间的同步*。实现这个机制的协议叫做 *Zab 协议*。Zab 协议有两种模式，它们分别是*恢复模式（选主）*和*广播模式（同步）*。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 Server 具有相同的系统状态。

14. zookeeper 是如何保证事务的顺序一致性的？

zookeeper 采用了*递增的事务 Id* 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid,zxid 实际上是一个 64 位的数字,高 32 位是 epoch(时期; 纪元; 世; 新时代）用来标识 leader 是否发生改变，如果有新的 leader 产生出来，epoch 会自增，*低 32 位用来递增计数*。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

15. Zookeeper 下 Server 工作状态每个 Server 在工作过程中有三种状态：

LOOKING：当前 Server *不知道 leader 是谁*，正在搜寻

LEADING：当前 Server 即为选举出来的 leader

FOLLOWING：leader 已经选举出来，当前 Server 与之同步

16. zookeeper 是如何选取主 leader 的？

当 leader 崩溃或者 leader 失去大多数的 follower，这时 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 Server 都恢复到一个正确的状态。Zk 的选举算法有两种：一种是基于 basic paxos 实现的，另外一种是基于 fast paxos 算法实现的。系统默认的选举算法为 *fast paxos*。

1、Zookeeper 选主流程(basic paxos)

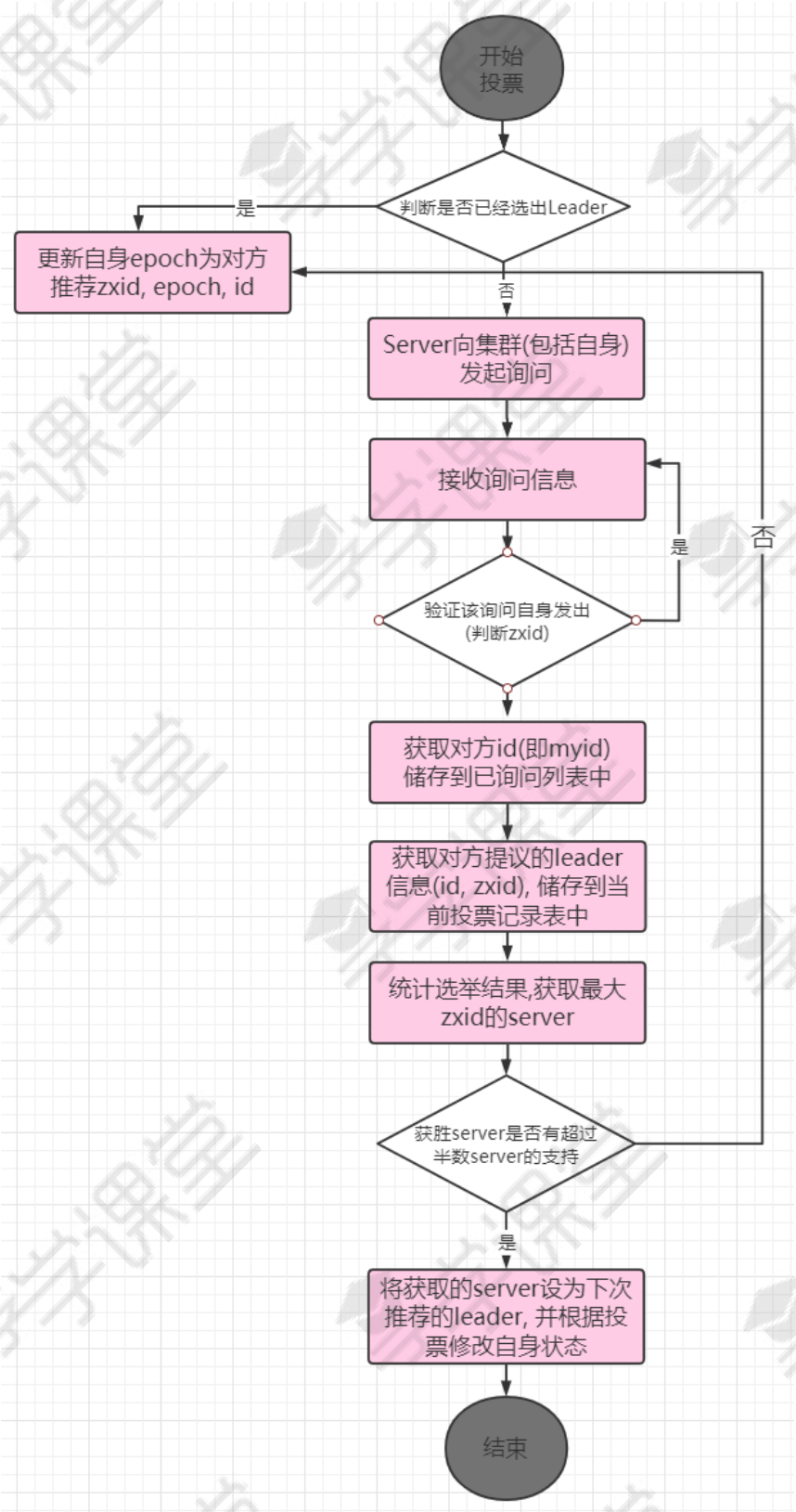
（1）选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的 Server；

（2）选举线程首先向所有 Server 发起一次询问(包括自己)；

（3）选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 id(myid)，并存储到当前询问对象列表中，最后获取对方提议的 leader 相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

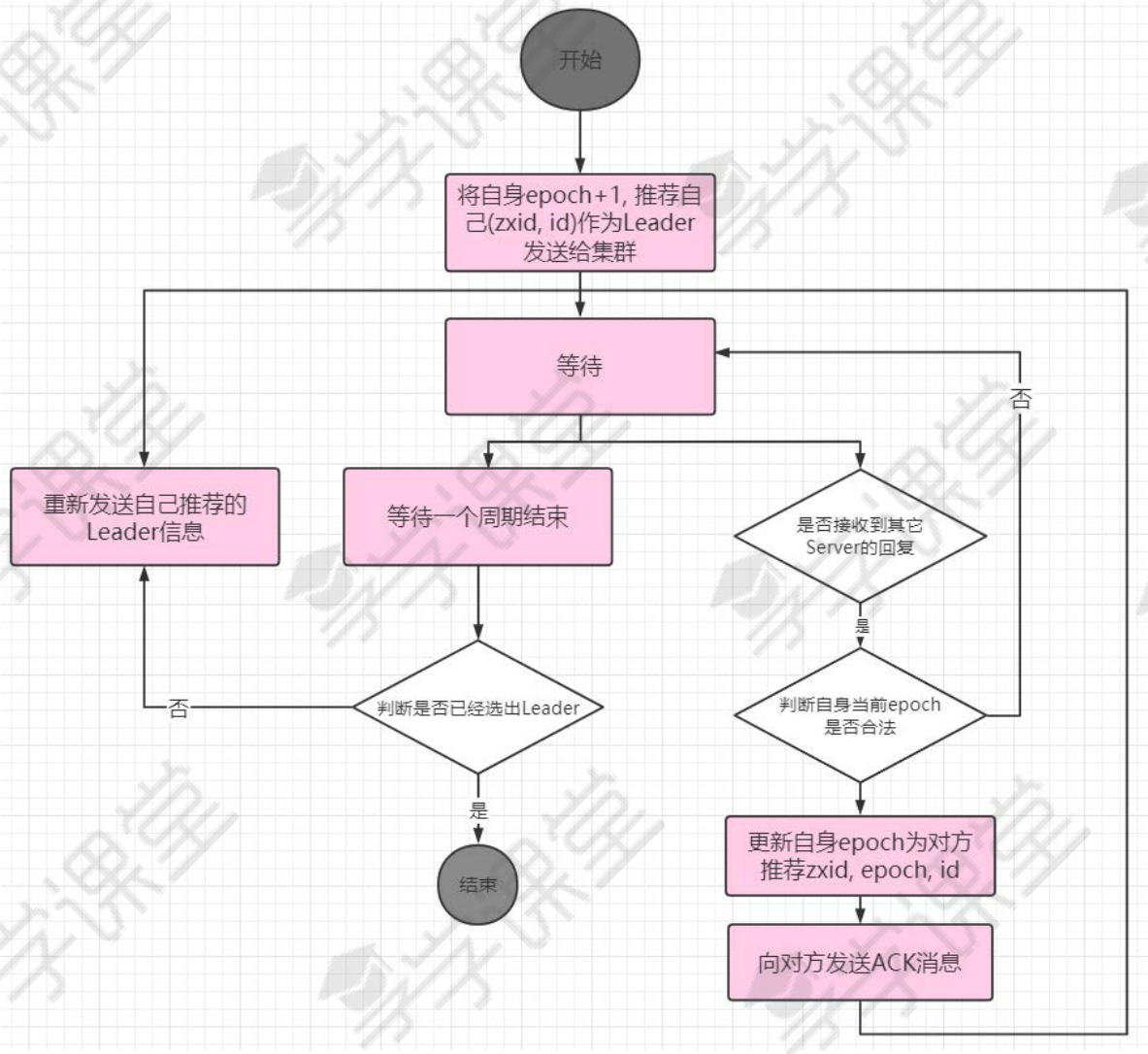
（4）收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server；

（5）线程将当前 zxid 最大的 Server 设置为当前 Server 要推荐的 Leader，如果此时获胜的 Server 获得 n/2 + 1 的 Server 票数，设置当前推荐的 leader 为获胜的 Server，将根据获胜的 Server 相关信息设置自己的状态，否则，继续这个过程，直到 leader 被选举出来。 通过流程分析我们可以得出：要使 Leader 获得多数 Server 的支持，则 Server 总数必须是奇数 2n+1，且存活的 Server 的数目不得少于 n+1. 每个 Server 启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的 server 还会从磁盘快照中恢复数据和会话信息，zk 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。



2、Zookeeper 选主流程(basic paxos)

fast paxos 流程是在选举过程中，某 Server 首先向所有 Server 提议自己要成为 leader，当其它 Server 收到提议以后，解决 epoch 和 zxid 的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出 Leader。



17. Zookeeper 同步流程

选完 Leader 以后，zk 就进入状态同步过程。

1、Leader 等待 server 连接；

2、Follower 连接 leader，将最大的 zxid 发送给 leader；

3、Leader 根据 follower 的 zxid 确定同步点；

4、完成同步后通知 follower 已经成为 uptodate 状态；

5、Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了。

18. 分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台*改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。*

对于执行情况汇报：每个工作进程都在某个目录下*创建一个临时节点。并携带工作的进度数据，*这样*汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。*

19. 机器中为什么会有 leader？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，*其他的机器可以共享这个结果，*这样可以大大*减少重复计算，提高性能，*于是就需要进行 leader 选举。

20. zk 节点宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；

如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)

2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)

21. zookeeper 负载均衡和 nginx 负载均衡区别

zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件；但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

22. zookeeper watch 机制

Watch 机制官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。

Zookeeper 机制的特点：

1、一次性触发数据发生改变时，一个 watcher event 会被发送到 client，但是 client *只会收到一次这样的信息。*

2、watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是*异步*的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于*网络延迟或其他因素导致客户端在不通的时刻监听到事件，*由于 Zookeeper 本身提供了 *ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。*所以我们使用

Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper *只能保证最终的一致性，而无法保证强一致性*。

3、数据监视 Zookeeper 有数据监视和子数据监视 getdata() and exists()设置数据监视，getchildren()设置了子节点监视。

4、注册 watcher *getData*、*exists*、*getChildren*

5、触发 watcher *create*、*delete*、*setData*

6、*setData()*会触发 znode 上设置的 data watch（如果 set 成功的话）。一个成功的 *create()* 操作会触发被创建的 znode 上的数据 watch，以及其父节点上的 child watch。而一个成功的 *delete()*操作将会同时触发一个 znode 的 data watch 和 child watch（因为这样就没有子节点了），同时也会触发其父节点的 child watch。

7、当一个客户端*连接到一个新的服务器上*时，watch 将会被以任意会话事件触发。当*与一个服务器失去连接*的时候，是无法接收到 watch 的。而当 client *重新连接*时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，*watch 可能会丢失*：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

8、Watch 是轻量级的，其实就是本地 JVM 的 *Callback*，服务器端只是存了是否有设置了 Watcher 的布尔类型

数据结构与算法面试题目

1. 说说常见的集合有哪些。

Map 接口和Collection 接口是所有集合类框架的父接口：

Collcetion 接口的子接口包括：Set 接口和List 接口。

Map 接口的实现类主要有：HashMap、HashTable、ConcurrentHashMap 以及Properties 等。

Set 接口的实现类主要有：HashSet、TreeSet、LinkedHashSet 等。

List 接口的实现类主要有：ArrayList、LinkedList、Stack 以及 Vector 等。

2. HashMap 与 HashTable 的区别。

主要有以下几点区别。

HashMap 没有考虑同步，是线程不安全的;HashTable 在关键方法（put、get、contains、size 等）上使用了Synchronized 关键字，是线程安全的。

HashMap 允许Key/Value 都为null，后者Key/Value 都不允许为null。

HashMap 继承自AbstractMap 类；而HashTable 继承自Dictionary 类。

在jdk1.8 中，HashMap 的底层结构是数组+链表+红黑树，而HashTable 的底层结构是数组+链表。

HashMap 对底层数组采取的懒加载，即当执行第一次put 操作时才会创建数组；而HashTable 在初始化时就创建了数组。

HashMap 中数组的默认初始容量是16，并且必须是2 的指数倍数，扩容时newCapacity=2*oldCapacity;而HashTable 中默认的初始容量是11,并且不要求必须是2 的指数倍数，扩容时newCapacity=2*oldCapacity+1。

在hash 取模计算时，HashTable 的模数一般为素数，简单的做除取模结果会更为均匀，int index = (hash & 0x7FFFFFFF) % tab.length;

HashMap 的模数为2 的幂，直接用位运算来得到结果，效率要大大高于做除法，i = (n - 1) & hash。

3. HashMap 是怎么解决哈希冲突的

哈希冲突：当两个不同的输入值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做哈希碰撞。

在Java 中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难。链表的特点是：寻址困难，但插入和删除容易。所以我们将数组和链表结合在一起，发挥两者的优势，使用一种叫做链地址法的方式来解决哈希冲突。这样我们就可以拥有相同哈希值的对象组织成的一个链表放在hash 值对应的bucket 下，但相比

Key.hashCode() 返回的int 类型，我们HashMap 初始的容量大小DEFAULT_INITIAL_CAPACITY = 1 << 4（即2 的四次方为16）要远小于int 类型的范围，所以我们如果只是单纯的使用hashcode 取余来获取对应位置的bucket，这将会大大增加哈希碰撞的几率，并且最坏情况下还会将HashMap 变成一个单链表。所以肯定要对hashCode 做一定优化。

来看HashMap 的hash()函数。上面提到的问题，主要是因为如果使用hashCode 取余，那么相当于参与运算的只有hashCode 的低位，高位是没有起到任何作用的，所以我们的思路就是让**hashCode 取值出的高位也参与运算，进一步降低hash 碰撞的概率，使得数据分布更平均，我们把这样的操作称为扰动，在JDK 1.8 中的hash()函数相比在JDK 1.7 中的4 次位运算，5 次异或运算（9 次扰动），在1.8 中，只进行了1 次位运算和1 次异或运算（2 次扰动），更为简洁了。两次扰动已经达到了高低位同时参与运算的目的，提高了对应数组存储下标位置的随机性和均匀性。

通过上面的链地址法（使用散列表）和扰动函数，数据分布更为均匀，哈希碰撞也减少了。但是当HashMap 中存在大量的数据时，假如某个bucket 下对应的链表中有n 个元素，那么遍历时间复杂度就变成了O(n)，针对这个问题，JDK 1.8 在HashMap 中新增了红黑树的数据结构，进一步使得遍历复杂度降低至O(logn)。

简单总结一下HashMap 是如何有效解决哈希碰撞的：

使用链地址法（散列表）来链接拥有相同hash 值的元素；

使用2 次扰动（hash 函数）来降低哈希冲突的概率，使得概率分布更为均匀；

引入红黑树进一步降低遍历的时间复杂度。

4. HashMap 中为什么数组长度要保证 2 的幂次方。

只有当数组长度为 2 的幂次方时，h&(length-1)才等价于 h%length，可以用位运算来代替做除取模运算，实现了 key 的定位，2 的幂次方也可以减少冲突次数，提高 HashMap 的查询效率；

当然，HashTable 就没有采用 2 的幂作为数组长度，而是采用素数。素数的话是用简单做除取模方法来获取下标 index，而不是位运算，效率低了不少，但分布也很均匀。

5. 什么是 Java 集合的快速失败机制“fail-fast”，以及安全失败“fail-safe”。

“fail-fast”是Java 集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast 机制。

例如：假设存在两个线程（线程1、线程2），线程1 通过Iterator 在遍历集合A 中的元素，在某个时候线程2 修改了集合A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出ConcurrentModificationException 异常，从而产生fail-fast 机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个modCount 变量。集合在被遍历期间如果内容发生变化，就会改变odCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount 变量是否为expectedmodCount

值，是的话就返回遍历；否则抛出异常ConcurrentModification，终止遍历。

Java.util 包下的集合类都是快速失败机制，不能在多线程下发生并修改（迭代过程中被修改）。

与“fail-fast”对应的是“fail-safe”。

采用安全失败机制的集合容器,在遍历时不是直接在集合内容上访问的,而是先copy 原有集合内容,在拷贝的集合上进行遍历。由于迭代时是对原集合的拷贝的值进行遍历,所以在遍历过程中对原集合所作的修改并不能被迭代器检测到,所以不会触发ConcurrentModificationException 异常。

基于拷贝内容的迭代虽然避免了ConcurrentModificationException 异常，但同样地，迭代器并不能访问到修改后的内容，简单来说，迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器行为是不知道的。

Java.util.concurrent 包下的容器都是安全失败的,可以在多线程下并发使用,并发修改。

6. ArrayList 和 LinkedList 的区别。

主要有以下几点区别：

LinkedList 实现了List 和Deque 接口,一般称为双向链表;ArrayList 实现了List 接口，是动态数组。

LinkedList 在插入和删除数据时效率更高,ArrayList 在查找某个index的数据时效率更高。

LinkedList 比ArrayList 需要更多内存。

7. HashSet 是如何保证数据不可重复的。

HashSet 的底层其实就是HashMap，只不过HashSet 是实现了Set 接口，并且把数据作为Key 值，而Value 值一直使用一个相同的虚值来保存。由于ashMap 的K 值本身就不允许重复，并且在HashMap 中如果K/V 相同时，会用新的V 覆盖掉旧的V，然后返回旧的V，那么在HashSet 中执行这一句话始终会返回一个false，导致插入失败，这样就保证了数据的不可重复性。

8. BlockingQueue 是什么。

Java.util.concurrent.BlockingQueue 是一个队列，在进行检索或移除一个元素的时候,它会等待队列变为非空;当添加一个元素时,它会等待队列中的可用空间。BlockingQueue 接口是Java 集合框架的一部分，主要用于实现生产者-消费者模式。这样我们就不需要担心等待生产者有可用的空间，以及消费者有可用的对象。因为它们都在BlockingQueue 的实现类中被处理了。

Java 提供了几种BlockingQueue 的实现，比如ArrayBlockingQueue 、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue 等。

9. JDK8 中 HashMap 为什么选用红黑树而不是 AVL 树

在平常我们用HashMap的时候，HashMap里面存储的key是具有良好的hash算法的key（比如String、Integer等包装类），冲突几率自然微乎其微，此时链表几乎不会转化为红黑树，但是当key为我们自定义的对象时，我们可能采用了不好的hash算法，使HashMap中key的冲突率极高，但是这时HashMap为了保证高速的查找效率，引入了红黑树来优化查询了。

因为从时间复杂度来说，链表的查询复杂度为o(n)；而红黑树的复杂度能达到o(logn)；比如若hash算法写的不好，一个桶中冲突1024个key，使用链表平均需要查询512次，但是红黑树仅仅10次，红黑树的引入保证了在大量hash冲突的情况下，HashMap还具有良好的查询性能。

红黑树相比avl树，在检索的时候效率其实差不多，都是通过平衡来二分查找。但对于插入删除等操作效率提高很多。红黑树不像avl树一样追求绝对的平衡，他允许局部很少的不完全平衡，这样对于效率影响不大，但省去了很多没有必要的调平衡操作，avl树调平衡有时候代价较大，所以效率不如红黑树。

10. JDK8 中 HashMap 链表转红黑树的阈值为什么选 8？

HashMap 在jdk1.8 之后引入了红黑树的概念，表示若桶中链表元素超过8 时，会自动转化成红黑树；若桶中元素小于等于6 时，树结构还原成链表形式。

HashMap源码作者通过泊松分布算出，当桶中结点个数为8时，出现的几率是亿分之6的，因此常见的情况是桶中个数小于8的情况，此时链表的查询性能和红黑树相差不多,因为红黑树的平均查找长度是log(n)，长度为8 的时候，平均查找长度为3，如果继续使用链表，平均查找长度为8/2=4，这才有转换为树的必要。链表长度如果是小于等于6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。

亿分之6这个几乎不可能的概率是建立在良好的hash算法情况下，例如String，Integer等包装类的hash算法，如果一旦发生桶中元素大于8，说明是不正常情况，可能采用了冲突较大的hash算法，此时桶中个数出现超过8的概率是非常大的，可能有n个key冲突在同一个桶中，这个时候就必要引入红黑树了。

另外，上下阈值选择6 和8 的情况下，中间有个差值7 可以防止链表和树之间频繁的转换。假设一下，如果设计成链表个数超过8 则链表转换成树结构，链表个数小于8 则树结构转换成链表，如果一个HashMap 不停的插入、删除元素，链表个数在8 左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

11. 说一下几种常见的排序算法和分别的复杂度

a. 快速排序：

i. 原理：快速排序采用用的是一种分治的思想,它先找一个基准数(一般选择第一个值),

然后将比这个基准数小的数字都放到它的左边,然后再递归调用,分别对左右两边快速排序,直到每一边只有一个数字.整个排序就完成了.

1.选定一个合适的值（理想情况中值最好，但实现中一般使用用数组第一个值）,称为“枢轴”(pivot)。

2.基于这个值，将数组分为两部分，较小的分在左边，较大的分在右边。

3.可以肯定，如此一轮下来，这个枢轴的位置一定在最终位置上。

4.对两个子数组分别重复上述过程，直到每个数组只有一个元素。

5.排序完成。

ii. 复杂度：O(n)

b. 冒泡排序：

i. 原理：冒泡排序其实就是逐一比较交换,进行里外两次循环,外层循环为遍历所有数字,逐个确定每个位置,里层循环为确定了了位置后,遍历所有后面没有确定位置的数字,与该位置的数字进行比较,只要比该位置的数字小,就和该位置的

数字进行交换.

ii. 复杂度：O(n^2)，最佳时间复杂度为O(n)

c. 直接插入入排序：

i. 原理：直接插入入排序是将从第二个数字开始,逐个拿出来,插入到之前排好序的数列列里.

ii. 复杂度：O(n^2)，最佳时间复杂度为O(n)

d. 直接选择排序：

i. 原理:直接选择排序是从第一个位置开始遍历位置,找到剩余未排序的数据里里里最小的,找到最小的后,再做交换。

ii. 复杂度：O(n^2)

12. Hashmap 什么时候进行扩容呢？

当 hashmap 中的元素个数超过数组大小 loadFactor 时，就会进行数组扩容，loadFactor 的默认值为 0.75，也就是说，默认情况下，数组大小为 16，那么当 hashmap 中元素个数超过 160.75=12 的时候，就把数组的大小扩展为 216=32，即扩大一倍，然后重新

计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知 hashmap 中元素的个数，那么预设元素的个数能够有效的提高 hashmap 的性能。

比如说，我们有 1000 个元素 new HashMap(1000)，但是理论上来讲 new HashMap(1024) 更合适，不过上面已经说过，即使是 1000，hashmap 也自动会将其设置为 1024。但是 new HashMap(1024) 还不是更合适的，因为 0.75*1000 < 1000，也就是说为了让 0.75 * size > 1000，我们必须这样 new HashMap(2048) 才最合适，既考虑了 & 的问题，也避免了 resize 的问题。

13. HashSet 和 TreeSet 有什么区别？

HashSet 是由一个 hash 表来实现的，因此，它的元素是无序的。

TreeSet 是由一个树形的结构来实现的，它里面的元素是有序的。

14. LinkedHashMap 的实现原理？

LinkedHashMap 也是基于 HashMap 实现的，不同的是它定义了一个 Entry header，这个 header 不是放在 Table 里，它是额外独立出来的。

LinkedHashMap 通过继承 hashMap 中的 Entry，并添加两个属性 Entry before,after，和 header 结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。LinkedHashMap 定义了排序模式 accessOrder，该属性为 boolean 型变量，对于访问顺序，为 true；对于插入顺序，则为 false。一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

15. 什么是迭代器 (Iterator)？

Iterator 接口提供了很多对集合元素进行迭代的方法。每一个集合类都包含了可以返回迭代 器实例的迭代方法。迭代器可以在迭代的过程中删除底层集合的元素，但是不可以直接调用集合的 remove(Object Obj) 删除，可以通过迭代器的 remove() 方法删除。

16. Iterator 和 ListIterator 的区别是什么？

下面列出了他们的区别：

Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只能用来遍历 List。

Iterator 对集合只能是前向遍历，ListIterator 既可以前向也可以后向。

ListIterator 实现了 Iterator 接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

17. Collection 和 Collections 的区别。

collection 是集合类的上级接口，继承与它的接口主要是 set 和 list。

collections 类是针对集合类的一个帮助类. 它提供一系列的静态方法对各种集合的搜索, 排序, 线程安全化等操作。