**小滴面试题**

目录

[集合 8](#_Toc126708654)

[java集合List 8](#_Toc126708655)

[说下vector和ArrayList LinkedList 的联系和区别？分别的使用场景？ 8](#_Toc126708656)

[如果要保证线程安全，arrayList 应该怎么做，用几种方式？ 8](#_Toc126708657)

[接上提问，CopyOnWriteArrayList和Collections.synchronizedList的区别，使用场景， 8](#_Toc126708658)

[考点：list集合扩充机制 8](#_Toc126708659)

[编程语言集合框架ConcurrentHashMap 9](#_Toc126708660)

[了解concurrentHashMap吗？为什么性能比HashTable高，说下原理 9](#_Toc126708661)

[jdjk1.7和1.8里面concurrentHashMap实现的区别有没有了解。 9](#_Toc126708662)

[Java集合hashMap框架 - map 基础 9](#_Toc126708663)

[了解MAP吗？用过哪些Map的实现 9](#_Toc126708664)

[说下HashMap和HashTable的区别 9](#_Toc126708665)

[介绍下对象的hashCode（）和equals（）,使用场景 9](#_Toc126708666)

[set和map的关系 9](#_Toc126708667)

[常见的map的排序规则是怎么样的？ 10](#_Toc126708668)

[考查map的横向和纵向知识点 10](#_Toc126708669)

[如果需要线程安全的，且高效的map，应该怎么实现？ 10](#_Toc126708670)

[为什么collections.synchronizedMap是线程安全? 10](#_Toc126708671)

[看过HashMap源码吗？介绍你了解的hashMap 10](#_Toc126708672)

[能否给解释一下什么是hash碰撞，常见的解决方法有哪些方法，hashmap采用那种方法？ 10](#_Toc126708673)

[你说下hashmap底层为什么使用数组+链表+红黑树，为什么要用这几类结构呢？ 10](#_Toc126708674)

[为啥选择红黑树而不用其他树，比如二叉查找树，为啥不一开始就用红黑树，而是到8的长度才会变换。 10](#_Toc126708675)

[并发编程 11](#_Toc126708676)

[考核是否掌握线程、进程、协程的区别 11](#_Toc126708677)

[能否解释一下什么是进程、线程、协程，他们之间的关系是怎么样的 11](#_Toc126708678)

[并发和并行 11](#_Toc126708679)

[平时业务代码里面有使用过多线程吗，能举例几个多线程的业务场景吗 11](#_Toc126708680)

[能举几个线程不安全的数据结构吗 11](#_Toc126708681)

[java中可以有哪些方法来保证线程安全 11](#_Toc126708682)

[java里实现多线程有几种方式，有什么不同，比较那种常用 12](#_Toc126708683)

[thread 12](#_Toc126708684)

[runable 12](#_Toc126708685)

[callable 12](#_Toc126708686)

[通过线程池创建线程 12](#_Toc126708687)

[线程的状态 13](#_Toc126708688)

[常见的状态有哪些，这些状态分别是做什么的？ 13](#_Toc126708689)

[常见线程的api操作 13](#_Toc126708690)

[了解volatile 关键字不？能否解释吗？然后这和synchronized有什么大的区别 13](#_Toc126708691)

[说下volatitle可以避免指令重排，能否解释一下什么是指令重排。 14](#_Toc126708692)

[知道happens-before吗，能否简单解释一下， 14](#_Toc126708693)

[并发编程三要素是否知道，能否分别解释一下，举个简单的例子， 14](#_Toc126708694)

[并发编程三要素 原子性 可见性 有序性 概述 14](#_Toc126708695)

[高性能的Compare and Swap 你知道多少 也就是CAS 15](#_Toc126708696)

[全称Compare and Swap 15](#_Toc126708697)

[CAS会存在比较严重的问题？ 15](#_Toc126708698)

[AQS知道AQS吗？能否介绍下，它的核心思想是什么？ 15](#_Toc126708699)

[AQS简述 15](#_Toc126708700)

[AQS的作用 16](#_Toc126708701)

[锁 17](#_Toc126708702)

[java多线程锁 17](#_Toc126708703)

[悲观锁和乐观锁: 17](#_Toc126708704)

[非公平锁性能高于公平锁，更能重复利用CPU的时间。 17](#_Toc126708705)

[可重入式锁能一定程度上避免死锁，synchronized reentrantLoc 17](#_Toc126708706)

[自旋锁 共享锁 互斥锁 死锁 18](#_Toc126708707)

[偏向锁 轻量级锁 重量级锁 18](#_Toc126708708)

[多线程死锁 19](#_Toc126708709)

[死锁的4个必要条件 19](#_Toc126708710)

[解决死锁，进行优化 19](#_Toc126708711)

[多线程使用的遵循标准实践 19](#_Toc126708712)

[不可重入锁 ReentrantLoc 19](#_Toc126708713)

[可重入式锁 ReentrantLoc 20](#_Toc126708714)

[synchronized 20](#_Toc126708715)

[对synchronized 了解吗？能否介绍synchronized 的理解 20](#_Toc126708716)

[jdk1.6后进行了优化。你知道哪些大的变化吗 20](#_Toc126708717)

[springCloud 21](#_Toc126708718)

[CAP 21](#_Toc126708719)

[CAP理论：指的是在一个分布式系统中，consistency(一致性)，Availaility(可用性)，Parttion tolerance（分区容错性）三者不可同时获得 21](#_Toc126708720)

[常见的注册中心：zk、eureka、nacos。应该怎样选择 21](#_Toc126708721)

[分布式CAP的权衡结果BASE理论 22](#_Toc126708722)

[什么是BASE理论 22](#_Toc126708723)

[不同数据一致性模型 22](#_Toc126708724)

[为什么要使用alibabaCloud和springCloud 的区别 22](#_Toc126708725)

[微服务的熔断降级 23](#_Toc126708726)

[分布式容错框架 23](#_Toc126708727)

[熔断 23](#_Toc126708728)

[降级 23](#_Toc126708729)

[springBoot和SpringCloud 的关系 23](#_Toc126708730)

[dubbo是如何做到系统交互的 23](#_Toc126708731)

[分布式事务 23](#_Toc126708732)

[2PC 23](#_Toc126708733)

[3PC 24](#_Toc126708734)

[TCC 24](#_Toc126708735)

[SpringBoot 26](#_Toc126708736)

[springBoot启动原理 26](#_Toc126708737)

[缓存 – redis 27](#_Toc126708738)

[redis 基础 27](#_Toc126708739)

[为什么要用redis,为啥不用其他缓存，比如memcache呢？ 27](#_Toc126708740)

[你用redis 哪些数据结构，说下这些结构的使用场景有哪些？ 27](#_Toc126708741)

[redis 是单线程，为什么这么快？ 27](#_Toc126708742)

[redis 有哪些持久化方式，分别说下他们的区别。 28](#_Toc126708743)

[*支持AOF和RDB持久化* 28](#_Toc126708744)

[AOF 28](#_Toc126708745)

[RDB 28](#_Toc126708746)

[缓存空间不够 28](#_Toc126708747)

[一般会使用淘汰策略,常见的淘汰策略有FIFO/LRU LFU 28](#_Toc126708748)

[能否说下FIFO/LRU LFU 这些策略不同点 28](#_Toc126708749)

[缓存击穿 穿透 雪崩的区别 28](#_Toc126708750)

[缓存击穿 28](#_Toc126708751)

[缓存穿透 28](#_Toc126708752)

[缓存雪崩（多个热点key都过期） 29](#_Toc126708753)

[redis事务 29](#_Toc126708754)

[redis事务命令 29](#_Toc126708755)

[reids 集群方案 redis提供了三种集群策略 29](#_Toc126708756)

[第一种主从模式， 29](#_Toc126708757)

[哨兵模式 29](#_Toc126708758)

[cluster模式 29](#_Toc126708759)

[Spring 30](#_Toc126708760)

[spring核心理念 30](#_Toc126708761)

[能否介绍下spring框架的di和ioc 30](#_Toc126708762)

[spring里面的单例和多例，scope作用域。 30](#_Toc126708763)

[bean实例化过程 30](#_Toc126708764)

[spring三级缓存 30](#_Toc126708765)

[springAOP 31](#_Toc126708766)

[概念 31](#_Toc126708767)

[什么是动态代理 sprimgAop 使用什么代理 31](#_Toc126708768)

[能否说下JDK动态代理和CGLIB动态代理的区别 31](#_Toc126708769)

[说下spring aop 中的代理使用的默认策略 31](#_Toc126708770)

[BeanPostProcessor的作用 31](#_Toc126708771)

[spring源码 32](#_Toc126708772)

[spring事务传播机制 32](#_Toc126708773)

[spring组件实例化讲解 32](#_Toc126708774)

[bean的生命周期 32](#_Toc126708775)

[bean的创建过程: 33](#_Toc126708776)

[@autoWired是如何工作的 33](#_Toc126708777)

[@beanFactory和Application的区别 33](#_Toc126708778)

[相同 33](#_Toc126708779)

[不同 33](#_Toc126708780)

[Mybatis 34](#_Toc126708781)

[jdbc连接数据库的开发步骤 34](#_Toc126708782)

[简单说下mybatis加载的流程 34](#_Toc126708783)

[http协议 35](#_Toc126708784)

[常见的http method 有哪些，使用场景 35](#_Toc126708785)

[http1.0 定义了三种 35](#_Toc126708786)

[http 2.0 35](#_Toc126708787)

[常见的http 状态码 35](#_Toc126708788)

[http协议无状态解决方案cookie和session 35](#_Toc126708789)

[单机情况下http无状态解决方案，cookie和session知识点 35](#_Toc126708790)

[你们公司C端业务登录时怎么样做的 35](#_Toc126708791)

[JWT 36](#_Toc126708792)

[简述 36](#_Toc126708793)

[JWT格式组成头部、负载、签名 36](#_Toc126708794)

[JWT为啥要使用这个，它的优点 缺点 36](#_Toc126708795)

[生成的token,在客户端或者游览器怎么存储的 36](#_Toc126708796)

[游览器的同源策略和跨域知识点 36](#_Toc126708797)

[同源策略 36](#_Toc126708798)

[为什么出现跨域，有什么常见的解决办法 37](#_Toc126708799)

[Mysql 38](#_Toc126708800)

[事务的ACID是什么 38](#_Toc126708801)

[原子性（atomicity）: 38](#_Toc126708802)

[一致性Consistency: 38](#_Toc126708803)

[隔离性Isolation: 38](#_Toc126708804)

[持久性（durability）: 38](#_Toc126708805)

[数据库的隔离级别 38](#_Toc126708806)

[能否简单解释一下脏读、不可重复读、幻读的意思 38](#_Toc126708807)

[常见的隔离级别由低到高有哪几种,mysql默认是哪种 38](#_Toc126708808)

[索引 39](#_Toc126708809)

[索引的好处 39](#_Toc126708810)

[索引的缺点 39](#_Toc126708811)

[索引优化实践 39](#_Toc126708812)

[数据库设计查询和上线里面的坑,数据库查询关键词执行顺序 40](#_Toc126708813)

[mysql中的datetime和datestmap有什么区别 40](#_Toc126708814)

[mysql 常见日志考查 40](#_Toc126708815)

[消息中间件 42](#_Toc126708816)

[优缺点: 42](#_Toc126708817)

[主流消息队列Apache activeMQ、kafka、rabbitMq、rocketMq 42](#_Toc126708818)

[apacheMq 42](#_Toc126708819)

[kafka 42](#_Toc126708820)

[rabbitmq 42](#_Toc126708821)

[rocketMq 42](#_Toc126708822)

[mq分布式事务 43](#_Toc126708823)

[rocketMq事务消息： 43](#_Toc126708824)

[整体交互流程 43](#_Toc126708825)

[rocketmq事务消息的状态 43](#_Toc126708826)

[消息队列怎么避免重复消费 43](#_Toc126708827)

[rocketmq如何保证消息的可靠性传输 44](#_Toc126708828)

[消息队列发生大量堆积应该怎么处理 44](#_Toc126708829)

[如何保证消息队列里消息生成和消费的顺序性 44](#_Toc126708830)

[JAVA基础 46](#_Toc126708831)

[Java基础 46](#_Toc126708832)

[2、== 和 equals 的区别是什么？ 46](#_Toc126708833)

[3、final 在 java 中有什么作用？ 46](#_Toc126708834)

[6、String str="i"与 String str=new String(“i”)一样吗？ 46](#_Toc126708835)

[12、普通类和抽象类有哪些区别？ 47](#_Toc126708836)

[13、接口和抽象类有什么区别？ 47](#_Toc126708837)

[17、什么是反射？ 47](#_Toc126708838)

[19、为什么要使用克隆？如何实现对象克隆？深拷贝和浅拷贝区别是什么？ 47](#_Toc126708839)

[20、throw 和 throws 的区别？ 48](#_Toc126708840)

[21、final、finally、finalize 有什么区别？ 48](#_Toc126708841)

[23、常见的异常类有哪些？ 48](#_Toc126708842)

[24、hashcode是什么？有什么作用？ 49](#_Toc126708843)

[30、什么是 Java 的内存模型? 49](#_Toc126708844)

[46、迭代器 Iterator 是什么？ 50](#_Toc126708845)

[47、Iterator 怎么使用？有什么特点？ 50](#_Toc126708846)

[48、Iterator 和 ListIterator 有什么区别？ 50](#_Toc126708847)

[56、线程的run()和start()有什么区别？ 51](#_Toc126708848)

[57、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？ 51](#_Toc126708849)

[95、cookie、session、token 51](#_Toc126708850)

[96、说一下 session 的工作原理？ 53](#_Toc126708851)

[105、websocket应用的是哪个协议 53](#_Toc126708852)

[107、请列举出在 JDK 中几个常用的设计模式？ 53](#_Toc126708853)

[108、什么是设计模式？你是否在你的代码里面使用过任何设计模式？ 55](#_Toc126708854)

[109、Java 中什么叫单例设计模式？请用 Java 写出线程安全的单例模式 56](#_Toc126708855)

[110、在 Java 中，什么叫观察者设计模式（observer design pattern）？ 56](#_Toc126708856)

[111、使用工厂模式最主要的好处是什么？在哪里使用？ 57](#_Toc126708857)

[112、请解释自动装配模式的区别？ 57](#_Toc126708858)

[113、举一个用 Java 实现的装饰模式(decorator design pattern)？它是作用于对象层次还是类层次？ 58](#_Toc126708859)

[118、BeanFactory 和 ApplicationContext 有什么区别？ 58](#_Toc126708860)

[132、Spring常用注解 59](#_Toc126708861)

[133、项目中是如何实现权限验证的，权限验证需要几张表 59](#_Toc126708862)

[135、如何防止表单重复提交 60](#_Toc126708863)

[136、Spring中都应用了哪些设计模式 60](#_Toc126708864)

[138、mybatis 中 #{}和 ${}的区别是什么？ 62](#_Toc126708865)

[175、mysql 的内连接、左连接、右连接有什么区别？ 62](#_Toc126708866)

[176、RabbitMQ的使用场景有哪些？ 62](#_Toc126708867)

[177、RabbitMQ有哪些重要的角色？有哪些重要的组件？ 63](#_Toc126708868)

[178、RabbitMQ中 vhost 的作用是什么？ 64](#_Toc126708869)

[179、说一下 jvm 的主要组成部分？及其作用？ 64](#_Toc126708870)

[180、说一下 jvm 运行时数据区？ 64](#_Toc126708871)

[181、什么是类加载器，类加载器有哪些？ 66](#_Toc126708872)

[182、说一下类加载的执行过程？ 67](#_Toc126708873)

[183、JVM的类加载机制是什么？ 68](#_Toc126708874)

[184、什么是双亲委派模型？ 68](#_Toc126708875)

[185、怎么判断对象是否可以被回收？ 68](#_Toc126708876)

[186、说一下 jvm 有哪些垃圾回收算法？ 69](#_Toc126708877)

[基本数据类型 ： 70](#_Toc126708878)

[成员变量和局部变量和静态变量 70](#_Toc126708879)

[异常: 71](#_Toc126708880)

[Throwable 71](#_Toc126708881)

[Error: NoClassDeFoundError 71](#_Toc126708882)

[Exception:异常 由代码抛出 71](#_Toc126708883)

[运行时异常和非运行时异常。 72](#_Toc126708884)

[反射 72](#_Toc126708885)

[IO 72](#_Toc126708886)

[IO模型 BIO ，NIO和AIO 72](#_Toc126708887)

[BIO全称是Blocking IO，同步阻塞式IO，是JDK1.4之前的传统IO模型。 72](#_Toc126708888)

[什么是NIO 同步非阻塞 73](#_Toc126708889)

[什么是AIO 73](#_Toc126708890)

[字节流和字符流 74](#_Toc126708891)

[序列化 74](#_Toc126708892)

[什么是 Java 序列化 74](#_Toc126708893)

[如何序列化 74](#_Toc126708894)

[问题 1) Java 中的可序列化接口和可外部接口之间的区别是什么？ 75](#_Toc126708895)

[问题 2) 可序列化的方法有多少？如果没有方法,那么可序列化接口的用途是什么？ 75](#_Toc126708896)

[问题 3) 什么是 serialVersionUID ？如果你不定义这个, 会发生什么？ 75](#_Toc126708897)

[问题 4) 序列化时,你希望某些成员不要序列化？你如何实现它？ 75](#_Toc126708898)

[问题 5) 如果类中的一个成员未实现可序列化接口，会发生什么情况？ 75](#_Toc126708899)

[问题 6) 如果类是可序列化的, 但其超类不是, 则反序列化后从超级类继承的实例变量的状态如何？ 76](#_Toc126708900)

[问题 7) 是否可以自定义序列化过程, 或者是否可以覆盖 Java 中的默认序列化过程？ 76](#_Toc126708901)

[问题 8) 假设新类的超级类实现可序列化接口, 如何避免新类被序列化？ 76](#_Toc126708902)

[问题 9) 在 Java 中的序列化和反序列化过程中使用哪些方法？ 77](#_Toc126708903)

[问题 10) 假设你有一个类,它序列化并存储在持久性中， 然后修改了该类以添加新字段。如果对已序列化的对象进行反序列化, 会发生什么情况？ 77](#_Toc126708904)

[11) Java序列化机制中的兼容更改和不兼容更改是什么？ 77](#_Toc126708905)

[12) 我们可以通过网络传输一个序列化的对象吗？ 77](#_Toc126708906)

[13) 在 Java 序列化期间,哪些变量未序列化？ 78](#_Toc126708907)

[数据库 79](#_Toc126708908)

[一、索引相关 79](#_Toc126708909)

[（1）什么是索引? 79](#_Toc126708910)

[（2）索引是个什么样的数据结构呢? 79](#_Toc126708911)

[（3）为什么使用索引？ 79](#_Toc126708912)

[（4）Innodb为什么要用自增id作为主键？ 79](#_Toc126708913)

[（5）Hash 索引和 B+ 树索引有什么区别或者说优劣呢? 79](#_Toc126708914)

[（6）什么是聚簇索引 80](#_Toc126708915)

[（7）说一说索引的底层实现？ 80](#_Toc126708916)

[（8）索引有哪些优缺点？ 81](#_Toc126708917)

[（9）聚簇索引和非聚簇索引的区别 81](#_Toc126708918)

[（11）MySQL中有几种索引类型，可以简单说说吗？ 81](#_Toc126708919)

[（14）联合索引是什么?为什么需要注意联合索引中的顺序? 82](#_Toc126708920)

[（15）创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因? 82](#_Toc126708921)

[（16）那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢? 82](#_Toc126708922)

[（17）为什么Mysql用B+树做索引而不用B-树或红黑树、二叉树 82](#_Toc126708923)

[（18）MySQL索引种类 82](#_Toc126708924)

[（19）索引在什么情况下遵循最左前缀的规则？ 83](#_Toc126708925)

[**二、事务相关** 83](#_Toc126708926)

[（1）什么是事务? 83](#_Toc126708927)

[（2）ACID是什么?可以详细说一下吗? 83](#_Toc126708928)

[（3）MySQL中为什么要有事务回滚机制？ 83](#_Toc126708929)

[（4）数据库并发事务会带来哪些问题？ 83](#_Toc126708930)

[（5）怎么解决这些问题呢?MySQL 的事务隔离级别了解吗? 86](#_Toc126708931)

[（6）Innodb使用的是哪种隔离级别呢? 86](#_Toc126708932)

[（7）不可重复读和幻读区别是什么？可以举个例子吗？ 87](#_Toc126708933)

[**三、锁相关** 87](#_Toc126708934)

[（1）对 MySQL 的锁了解吗? 87](#_Toc126708935)

[（2）MySQL 锁的分类 87](#_Toc126708936)

[（3）行级锁、表级锁、页级锁的描述与特点 87](#_Toc126708937)

[（4）共享锁 、 排他锁的描述 88](#_Toc126708938)

[（8）InnoDB 存储引擎有几种锁算法？ 89](#_Toc126708939)

[（15）什么是全局锁？它的应用场景有哪些？ 89](#_Toc126708940)

[（17）优化锁方面你有什么建议？ 89](#_Toc126708941)

[（4）说一下MySQL是如何执行一条SQL的？具体步骤有哪些？ 89](#_Toc126708942)

[（5）SQL 的执行顺序？ 90](#_Toc126708943)

[（6）简述触发器、函数、视图、存储过程？ 92](#_Toc126708944)

[（6）听说过视图吗？那游标呢？ 92](#_Toc126708945)

[**五、表结构相关** 92](#_Toc126708946)

[（1）为什么要尽量设定一个主键? 92](#_Toc126708947)

[（2）主键使用自增 ID 还是 UUID? 92](#_Toc126708948)

[（3）字段为什么要求定义为not null? 92](#_Toc126708949)

[（4）如果要存储用户的密码散列,应该使用什么字段进行存储? 92](#_Toc126708950)

[（5）说一说Drop、Delete与Truncate的共同点和区别? 93](#_Toc126708951)

[**六、其他问题** 93](#_Toc126708952)

[（1）MySQL 中的 varchar 和 char 有什么区别? 93](#_Toc126708953)

[**（2）varchar(10) 和 int(10) 代表什么含义?** 93](#_Toc126708954)

[（6）left join、right join以及inner join的区别? 93](#_Toc126708955)

[（7）什么是数据库约束,常见的约束有哪几种? 93](#_Toc126708956)

[（9）简述数据库的读写分离？ 94](#_Toc126708957)

[（13）count(1)、count(\*)与count(列名)的执行区别 94](#_Toc126708958)

[（14）sql 语句中where 1=1的作用 94](#_Toc126708959)

[（15）sql中null与空值的区别 94](#_Toc126708960)

[（1）日常工作中你是怎么优化SQL的？ 95](#_Toc126708961)

# 集合

## java集合List

### 说下vector和ArrayList LinkedList 的联系和区别？分别的使用场景？

* ArrayList:底层实现是数组，线程不安全，查询和修改非常快，但是增加和删除慢
* linkedList:底层是双向链表，线程不安全，查询和修改慢，增加和删除快。
* vector:底层是数组，线程安全，对元素操作时使用synchronized进行加锁
* 使用场景：

vector:已经很少用了

增加和删除多就用linkedlist

修改和查询多就用arraylist

### 如果要保证线程安全，arrayList 应该怎么做，用几种方式？

方式1：自己写个包装类，根据业务一般是add update remove 进行加锁

方式2: List<String> strings = Collections.synchronizedList(new ArrayList<String>());synchronized

方式3：CopyOnWriteArrayList<List<String>> lists = new CopyOnWriteArrayList<>(); 使用reentrantlock加锁，可重入式锁。

### 接上提问，CopyOnWriteArrayList和Collections.synchronizedList的区别，使用场景，

* CopyOnWriteArrayList：执行修改操作，会拷贝一份新的数据（add/set/remove）,代价昂贵，修改后会将原来的集合指向新的集合来完成操作，使用reentrantLock来保证不会有多个线程同时修改（可重入式锁，默认是非公平锁，该线程没有执行完成其线程在此请求优先获取锁）。

场景：适合读操作远大于写的操作的场景，（读操作时不需要加锁，直接获取，但是删除和增加需要锁读多写少）

copyonWriteArrayList的设计思想是怎么样的，有什么缺点？

答案：设计模式：读写分离+最终一直性，缺点：内存占用高，由于写时复制，内存里同时存在两个对象占用内存，如果大对象则容易发生YongGC和FULLGC

* Collections.synchronizedList ：线程安全的原因就是几乎每个方法都是使用synchronized加锁同步锁。

场景：写操作比copyonWriteArrayList,但是读操作并不如copyonWriteArrayList.

### 考点：list集合扩充机制

* 说下arrayList的扩容机制是怎么样的 注意：JDK1.7之前的arraylist的默认大小是10，jdk1.7之后是0，未指定容量，默认是0，若已经指定集合大小则为指定的大小，当集合第一次添加元素扩容大小为10，arrayList的元素个数大于10则为扩容大小=原始大小+原始大小/2. 也就是扩容因子是1.5倍。

## 编程语言集合框架ConcurrentHashMap

### 了解concurrentHashMap吗？为什么性能比HashTable高，说下原理

ConcurrentHashMap线程安全的Map,hashTalb类基本上所有的方法都采用了synchronized进行锁安全控制高并发情况下效率就低下。concurrentiHashMap是采用了分段锁的思想提高性能，锁力度更细化

### jdjk1.7和1.8里面concurrentHashMap实现的区别有没有了解。

* jdk8之前，concurrentHashMap使用锁分段技术，将数据分成一段段存储，每个数据配置一把锁，即segment类，这个类继承了reentrantLock来保证线程安全。技术点：sigment+HashEntry
* jdk8的版本取消segment这个分段数据结构，底层也是使用了node数组+链表+红黑树，从而实现了对每一段数据进行加锁，也减少了并发冲突概率，CAS（读)+synchronized(写) 技术点：node+cas+synchronized

## Java集合hashMap框架 - map 基础

### 了解MAP吗？用过哪些Map的实现

hashMap,LinkedMap、TreeMap.HashTable、ConcurrentHashMap

### 说下HashMap和HashTable的区别

HashMap底层基础数组+链表，非线程安全，允许有空的值和键，默认容量是16，扩容因子是0.75

hashTable：底层基于哈希表实现，线程安全（加了synchroized），默认容量是11，不允许有空的键和值。

hashcode和equals 对象的比较、排重hashcode和equals经常要重写，也是map和set里面常用知识

### 介绍下对象的hashCode（）和equals（）,使用场景

* HashCode

顶级类Object里面的方法，所有类都是继承Object,返回一个int类型的数。根据一定的hash规则（存储地址、字段、长度），映射一个值这个值就是哈希散列值

* equals

顶级类Object里面的方法，所有的类都是继承Object,返回的是一个布尔类型。

* 使用场景：

hashMap:散列桶（数组+链表），可以实现快速的存储和检索，但是确实包含无序的元素，适合在map中插入和定位元素

treeMap:s使用存储结构是一个平衡二叉树->红黑树。可以自定义排序规则，要实现comparator接口，能够便捷的实现内部元素的各种排序，但是一般性能比hashmap差，适用于安装自然排序或者自定义排序规则。

### set和map的关系

核心就是不保存重复的元素，存储一组唯一的对象set的每一种实现都是对应map里面的一种封装，hashset对应的就是hashmap，treeSet对应的就是treeMap

### 常见的map的排序规则是怎么样的？

按照添加顺序使用LinkedHashMap,按照自然排序就是treeMap,自定义排序TreeMap（Comparetor c）

## 考查map的横向和纵向知识点

### 如果需要线程安全的，且高效的map，应该怎么实现？

答案：多线程环境下可以用concurrent包下的concurrentHashMap,或者使用Collections.synchronizedMap,ConCurrentHashMap虽然是线程安全，但是它的效率比hashTable要高很多.

### 为什么collections.synchronizedMap是线程安全?

答案：使用collections.synchronizedMap包装后的Map就是加锁的。

### 看过HashMap源码吗？介绍你了解的hashMap

答案：看过源码 HashMap底层（数组+链表+红黑树 jdk1.8才有的红黑树） 数组中每一项是一个链表，即数组和链表的结合体 Node<key,value>[] table[] 是数组，数组的元素是entry（node继承entry）,entry元素是一个key-value的键值对，它持有一个向下个Entry得引用，table数组中的每个entry元素同时也作为entry链表的首节点，也指向了该链表下个entry元素。 在jdk1.8中，链表的长度大过8转换成红黑树。

### 能否给解释一下什么是hash碰撞，常见的解决方法有哪些方法，hashmap采用那种方法？

答：hash碰撞的意思就是不同的key得到的hash值相同，放到同一个bucket中。常见的解决办法：链表法，开放地址法，再哈希法。HashMap采用的是链表法。

### 你说下hashmap底层为什么使用数组+链表+红黑树，为什么要用这几类结构呢？

答案：数组node<k,v>[] table,根据对象的key的hash值进行在数组里面是哪个节点。链表的作用是解决hash冲突，将hash值一样的对象存在一个链表放在hash值对应的槽位。 红黑树 jdk8使用红黑树来代替8个节点的链表，主要是查询性能的提升，从原来的O（n）到O（long）,通过hash碰撞，让HashMap不断产生碰撞，那么相同的key的位置的链表就会不断的增长，当对整个hashMap的相应位置进行查询的时候，就会循环遍历这个超级大表，性能就会下降，所以改用红黑树。

### 为啥选择红黑树而不用其他树，比如二叉查找树，为啥不一开始就用红黑树，而是到8的长度才会变换。

二叉查找树在特殊的情况下，会变成一条线性结构，和原先的列表存在一样的深度遍历问题，查找性能会变慢，使用红黑树主要是提升查找数据的速度，红黑树是平衡二叉树的一种，插入新数据会通过左旋、右旋、变色等操作来保持平衡，解决单链表查询深度的问题，数据量少的时候操作数据，遍历线性表比红黑树所消耗的资源少，且前期数据少，平衡二叉树保持平衡是需要消耗资源，所以前期采用线性表，等到一定的数量之后在变换到红黑树。

# 并发编程

并发编程

## 考核是否掌握线程、进程、协程的区别

### 能否解释一下什么是进程、线程、协程，他们之间的关系是怎么样的

* 进程

本质上是一个独立执行的程序，进程是操作系统进行资源分配和调度的基本概念，操作系统进行资源分配和调度的一个独立单位

* 线程

是操作系统能够进行运算调度的最小单位，它被包含在进程中，是进程中实际运作单位，一个进程中可以并发多个线程，每个线程执行不同的任务，切换受系统控制。

* 协程

又称为微线程，是一种用户态的轻量级线程，协程不像线程和进程需要进行系统内核上的上下文切换，协程的上下文切换是由用户自己决定的，有自己的上下文，所以说是轻量级的线程，也称之为用户级别的线程就叫协程，一个线程有多个协程，线程进程都是同步机制，而协程则是异步，java的原生语法中并没有实现协程，目前python,lua和go等语言支持。

关系：一个线程有多个线程，它允许计算机同时运行两个或多个程序，一个进程至少有一个线程，线程是进程的最小执行单位，cpu调度切换的是进程和线程，进程和线程多了之后调度会消耗大量的cpu，线程是独立运行和独立调度的基本单位，CPU上真正运行的是线程，线程也可以对应多个协程。

### 并发和并行

并发：是指在单核的cpu上同时运行多个程序，并发不是真正意义上的“同时进行”，而时通过CPU去强时间片去执行。

并行：是指在多个cpu和或者一个多核CPU上同时运行多个程序，每个cpu或者多核CPu可以分别同时运行程序，两个程序不抢占CPU资源，可以真正做到同时进行，这种方式我们称之为并行。

### 平时业务代码里面有使用过多线程吗，能举例几个多线程的业务场景吗

异步任务 ：用户注册 记录日志 定时任务：定时备份日志 备份数据库 分布式计算：hadoop处理任务maperduce master-wark(单机进程) 服务编程：socket 网络编程，一个连接 一个线程

### 能举几个线程不安全的数据结构吗

arraylist linkedList hashMap

### java中可以有哪些方法来保证线程安全

加锁 synchronized /reentrantLock 使用volatile 声明变量，不能保证原子性，使用线程安全类 ，原子类 atomic 开头 并发容器 同步容器 copyonWriteArrayList /concurrentHashMap 等ThreadLocal 本地私有变量/信号量 semaphore等

## java里实现多线程有几种方式，有什么不同，比较那种常用

### thread

继承thread,重写run方法，创建实例，执行start,

优点，代码开发简单直接操作

缺点：单继承没有返回值，可扩展性差。

### runable

实现runabile，实现里面的run方法，创建thread类，使用runable接口的实现作为参数传递给thread对象，调用start方法

优点：线程可以实现多个接口，可以在继承一个类

缺点没有返回值，需要threa进行传递参数

### callable

创建callable接口实现类，并实现call方法，结合futureTask类包装callable对象，实现多线程

public class Mytask implements Callable<Object> {

@Override

public Object call() throws Exception {

System.out.println("通过callable实现多线程：" + Thread.currentThread().getName());

return "这是返回值";

}

public static void main(String[] args) {

Mytask mytask = new Mytask();

FutureTask<Object> futureTask = new FutureTask<>(mytask);

Thread thread = new Thread(futureTask);

thread.start();

System.out.println("主线程名称：" + Thread.currentThread().getName());

}

}

优点：有返回值，可扩展性高

缺点：jdk5之后才支持，需要重写call方法，结合多个类比如futureTask 和Thead

### 通过线程池创建线程

自定义runable接口，实现run方法，创建线程池，调用执行方法传入对象

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(

new Runnable() {

@Override

public void run() {

System.out.println("当前线程：" + Thread.currentThread().getName());

}

});

executorService.shutdown();

优点：安全高性能强，复用线程

缺点：jdk5后才支持，需要结合runable进行使用

## 线程的状态

### 常见的状态有哪些，这些状态分别是做什么的？

五种状态

创建new

就绪runable

运行runing

阻塞blocked

死亡dead

### 常见线程的api操作

#### 是否了解多线程开发里面常用的方法，sleep/yield/join wait/notify/notifyall,分别解释：

* sleep:属于线程Thread的方法让线程暂缓执行，等待预计的时间进行恢复，让渡cpu使用权，不会释放锁，进入阻塞状态，睡眠结束以后变成就绪状态。
* yield 属于线程thread的方法，t1/t2/t3/ 暂停当前线程的对象去执行其他线程，交出cpu使用权，不会释放锁，和sleep类似，作用：让相同优先权的线程进行执行，但不保证一定轮流。 注意：不会让线程进入阻塞状态，直接变为就绪状态，只需要重新获得CPU使用权。
* join 属于线程thread 的方法 在主线程上运行调用改方法，会让主线程休眠，不会释放已经持有的对象锁让调用join方法的线程先执行完成，在执行其他线程 类似 救护车 警车优先通过
* wait 属于object的方法 当前线程调用对象的wait方法，会释放锁，进入线程的等待队列需要依靠notify或者notifyall唤醒，或者wait(timout)时间自动唤醒。
* notify 属于object 的方法，唤醒在对象监视器上等待的单个线程，选择是任意的。
* notifyAll 属于object 的放大，唤醒在对象监视器上等待的全部线程。

## 了解volatile 关键字不？能否解释吗？然后这和synchronized有什么大的区别

* volatile是轻量级的synchronized，保证了共享变量的可见性，被volatitle关键字修饰的变量，如果发生了值变化，其他前线可立刻可见，避免出现脏读现象
* volatitle:保证可见性，但是不保证原子性，禁止指令重排 synchronized：保持可见性，也保持原子性
* 使用场景：不能修饰写入操作依赖当前值的变量，不如num++，num=num+1

### 说下volatitle可以避免指令重排，能否解释一下什么是指令重排。

* 指令重排分两类 编译器重排 和运行时重排序 Jvm在编译java代码或者CPU执行jvm字节码时，对现有的指令进行重排序，主要目的是优化运行效率（不改变程序结果的前提下）

int i=3/1 int b =4/2 int c=5/3 int h =a\*b\*c/4 定义顺序1 、2、3、4 计算顺序 1 、2、3 、 4 和2 、1 、3 、4，结果都是一样

* 虽然指令重排可以提高效率，但是多线程上可能会影响结果，有什么解决办法吗? 解决办法 内存屏障 解释：内存屏障是屏障指令，是CPU对屏障指令之前和之后的内存操作结果都是一样的

### 知道happens-before吗，能否简单解释一下，

* 涉及到并发，我们通常会想到两方面，一方面是使用多线程实现程序并发，另一方面是使用锁防止程序并发，不管是哪一方面都涉及到两大核心，三大性质，

两大核心分别是java内存模型、happer-before原则，三大原则，可见性 原子性 有序性

* 先行发生原则，volatitle 的内存可见性就体现了改原则之一 线程A操作 int k=1 线程b操作 int j=k 线程C 操作 ，int k =2
* 重排序确实可以提升性能，但是有时候会造成实际结果有我们所期望的结果不符，所以，JMM又提供了happers-before。而happens-before所做的事就是，每一条规则对应一个或多个编译器和处理器重排序规则，对于程序员而言，happens-before相当于对于重排序规则以及具体实现做了封装，方便程序员使用，程序员只需要理解happens-before规则就好，不需要去理解复杂的重排序相关的事儿。
* happer-before 八大原则

1、程序次序规则

2. 管程锁定规则

3、volatitle 变量规则

4、线程启动规则

5、线程中断规则

6、线程终止规则

7、对象终结规则

8、传递性

## 并发编程三要素是否知道，能否分别解释一下，举个简单的例子，

并发编程三要素 原子性 可见性 有序性 概述

* 原子性 ：一个不可在分割的颗粒，原子性指的是一个或多个操作要么全部成功要么全部失败，期间不能中断，也不存在上下文切换，线程切换会带来原子性的问题。

int num= 1; //原子性操作 num++;// 非原子性操作，从主内存中读取i的值到工作线程内存，进行i+1,在把i写到主内存，除非用原子类，即 java.util.concurrent.atomic 里面的原子类变量。

解决办法就是用synchronized 或者lock (比如 reentrantLock),来把这个多不操作变成原子操作，但是volatitle ，前面说过volatitle不能修饰有依赖值的情况。

* 有序性：程序执行的顺序按照代码的先后顺序执行，因为处理器可能会对指令进行重排，jvm在编译java代码或者CPU执行jvm字节码文件时，对现有的指令进行重新排序，主要的目的是优化运行效率（不改变程序结果的前提）

int a = 3/1 int b=4/2 int c=5/3 int h=a\*b\*c/4 执行顺序是 1 2 3 4 和2 3 1 4，结果都是一样，指令重排可以调高执行效率，但是多线程可能会影响结果。

* 可见性：一个线程A对共享变量的修改，另一个线程B可以立马看到

//线程A 执行

int num=0

//线程A执行

num++

//线程B执行

System.out.print("num==1"+num)

线程A执行i++之后在执行线程B，线程B可能会有两个结果，可能是0 和1；

因为i++ 在线程A中执行运算，并没有立刻更新到主内存中，而线程B从主内从中读取并打印，此时打印的就是0，也可能线程A执行完成更新到主内存中，线程B的值是1.所以需要保证线程的可见性， synchronized lock 和 volatitle 能够保证线程可见性。

## 高性能的Compare and Swap 你知道多少 也就是CAS

### 全称Compare and Swap

全称Compare and Swap即比较并交换，是实现并发应用到的一种技术 。底层通过unsafe类实现原子性操作，———内存地址 （v） 预期值 A ，新值B，如果内存中的地址和预期值一样，那么处理器自动将该位置值更新为新值，如果在第一轮的循环中，a线程获取地址被B线程更改，那么A线程会自旋，到下次才有可能机会执行。

CAS这个属于乐观锁，性能较悲观锁有很大的提高。 AtomicXX等原子类底层就是CAS实现，一定程度上比synchronized好，因为后者是悲观锁。

### CAS会存在比较严重的问题？

1、自旋时间长CPU利用增加，CAS里面是一个循环判断的过程，如果线程一直没有获取到线程，cpu资源会一直被占用。

2、存在ABA问题

## AQS知道AQS吗？能否介绍下，它的核心思想是什么？

## AQS简述

AQS的全称为（abstratQueuedSynchronized）这个类在java.util.concurrent.locks包下面，它是一个java提高的底层同步工具类，比如countDownLatch、reentrantLock、semaphore/futuretask等等都是基于aqs.

* 只要搞懂了AQS,那么juc中的绝大部分的api都能轻松掌握。
* 简单来说：是用一个int类型的变量表示同步状态，并提供了一系列的CAS操作来管理这个同步状态对象，一个state（用于计算器，类似GC的回收计算器），一个线程标记（当前线程是谁加的锁）一个是阻塞队列（用于存放其他为拿到锁的线程）
* 例子：线程A调用了Lock（）方法，通过CAS将state的值赋值为1，然后将该线程标记为A加锁，如果线程A还未释放锁时，线程B来了，会查询标记的状态，因为当前的锁标记为A，线程B未能匹配上，所以线程B会加入阻塞的队列中，知道线程A触发了unlock方法，这时线程B才有机会去拿到锁，但是不一定肯定会拿到。

### AQS的作用

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 都是基于AQS实现的，他使用了一个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、他是clh队列锁的一种辩题实现，他可以实现2中同步方式：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，同步器的设计基础模板方法模式，所以如果要实现我们自己的同步工具就需要覆盖其中的几个可重写方法，比如tryacquire/ tryreleaseshared等等

# 锁

锁

## java多线程锁

### 悲观锁和乐观锁:

悲观锁适合多写少读的场景，乐观锁适合多读少写的场景，乐观锁的吞吐量要高于悲观锁。

#### 悲观锁

当线程去操作数据的时候，总认为别的线程会进行修改操作，所以它每次拿到数据后就会上锁，别的线程去那数据的时候就会阻塞，比如synchronized

#### 乐观锁

每次拿数据的时候都认为别人不会修改，更新的时候会判断别人是否进行更新操作，通过版本号来判断识别，如果数据库被修改了版本号就拒绝修改，比如CAS就是乐观锁，但严格意义上来说并不是锁，通过原子性来保证数据的同步，比如数据库的乐观锁，通过版本控制来实现，CAS不会保证线程同步，乐观的认为在数据更新期间梅雨其他线程影响

### 非公平锁性能高于公平锁，更能重复利用CPU的时间。

#### 公平锁

指多个线程按照申请锁的顺序来获取锁，简单来说，如果一个线程组里，能保证每个线程都能拿到锁，比如reentrantLock （底层是同步队列FIFO:FIRST INPUT FIRST OUTPUT 来实现）

#### 非公平锁

获取锁的方法是随机获取的，保证不了每个线程都能拿到锁，有就是存在有线程饿死，一直拿不到锁，比如synchronized reentrantLock

### 可重入式锁能一定程度上避免死锁，synchronized reentrantLoc

#### 可重入锁

也叫递归锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁。

#### 不可重入锁

若当前线程执行某个方法已经获得了该所，那么在方法中尝试再次获取锁时，就会获取不到被阻塞

### 自旋锁 共享锁 互斥锁 死锁

#### 自旋锁

一个线程在获取锁的时候，如果锁已经被其他线程获取，该线程将循环等待，然后不断的判断锁是否能够被成功获取，知道获取到锁才会退出循环，任何时刻最多只能有一个执行单元获得锁 常见的自旋锁 ticketLock clhlock msclock

会发生线程的状态切换一直处于用户态，减少了线程上下文切换了消耗。缺点是不断的循环消耗CPU

#### 共享锁

要叫S锁/读锁，能查看但无法进行修改和删除的一种数据锁，加锁后其他用户可以并发读取、查询数据，但不能修改，增加，删除数据，改锁可被多个线程所持有，用于资源共享

#### 互斥锁

也叫X锁、排它锁、写锁、独享锁、 该锁每一次被一个线程所持有，加锁后任何线程识图再次加锁额线程就会被阻塞，知道当前线程解锁，例子：如果线程A 对data1加排它锁后，则其他线程不能再对data加任何类型的锁，获得互斥锁的线程技能读数据又能修改数据。

#### 死锁

两个或多个以上的线程在执行的过程中，由于竞争资源或者由于批次通信而造成的一种阻塞现象，若无外力作用下，他们都将无法让程序进行下去。

### 偏向锁 轻量级锁 重量级锁

下面三种锁时jvm为了提高锁的效率而坐的优化，可针对synchronized 的锁升级，锁定状态通过对象的监视器再对象投中的字段来表明，是不可逆的过程。

#### 偏向锁

一段同步代码块一直被一个线程所访问，那么改线程会自动获取锁，获取锁的代价更低。

#### 轻量级锁

当锁时是偏向锁的时候，被其他线程访问，偏向锁就会自动升级为轻量锁，其他线程会通过自旋的方式尝试获取锁，但不会阻塞，且性能会高点。

#### 重量级锁

当锁为轻量级锁的时候，其他线程虽然是自旋，但资源不会一直持续下去，当自旋一定次数的时候且还没有获取到锁，就会进入阻塞，改锁升级为重量级锁，重量级锁会让其他申请的线程进入阻塞，性能也会降低。

#### 分段锁

#### 行锁

#### 表锁

## 多线程死锁

### 死锁的4个必要条件

要发生死锁，下面的条件都成立，只要一个不满足，就不会发生死锁

* 互斥条件

资源不能共享，只能由一个线程进行使用

* 请求与保持条件：

线程已经获得一些资源，但因为请求其他的资源发生阻塞，对已经获得的资源保持不释放。

* 不可抢占：

有些资源是不可抢占的，当某个线程获得后，系统不能强制回收，只能由线程使用完成自己释放

* 循环

多个线程形成环形链，每个都占用对方申请的下个资源。

### 解决死锁，进行优化

常见的解决方法：

调整申请锁的范围

调整申请锁的顺序

### 多线程使用**的遵循**标准实践

* 给不同的线程起名称，方便后续排查问题
* 使用不同代码块或者同步方法的时候，尽量减小同步的范围
* 多用并发集合少用同步集合

都是支持线程安全

同步集合：hashTable vector 同步工具类包装Collections.syncXXX

并发集合 concurrentHashMap 使用的是分段加锁 copyonWriteArrayList

* 线上业务需要使用多线程，优先考虑线程池是否更加合适，然后判断那种线程池比较好，最后创建单一线程。

## 不可重入锁 ReentrantLoc

不可重入式锁：若当前线程某个方法已经获取了改锁，那么在方法中再次获取锁时，就会获取不到被阻塞。

## 可重入式锁 ReentrantLoc

可重入式锁。也叫递归锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁。

## synchronized

### 对synchronized 了解吗？能否介绍synchronized 的理解

synchronized是解决线程安全的问题，常用在同步普通方法，静态方法、代码块中。特点是非公平，可重入式锁。

非公平、可重入 每个对象中有一个锁和一个等待队列，锁只能被一个线程所持有，其他的线程需要锁则要进行阻塞等待。锁被释放后，对象会从队列中取出一个并唤醒，唤醒哪个线程是不确定的，不能保证公平性。

### jdk1.6后进行了优化。你知道哪些大的变化吗

有得到锁的资源进入block状态，涉及到操作系统用户模式和内核模式切换，代价是比较高，JDK6进行了优化，增加了从偏向锁到轻量锁在到重量级锁的过渡，但是中最终转换为重量级锁，性能比较低。

# springCloud

SpringCloud

## CAP

### CAP理论：指的是在一个分布式系统中，consistency(一致性)，Availaility(可用性)，Parttion tolerance（分区容错性）三者不可同时获得

* 一致性（C）：所有节点都可以访问到最新的数据
* 可用性（A）每个请求都是可以得到响应的，不管请求是成功还是失败
* 分区容错性（P）：除了全部整体网络故障，其他故障都不能导致服务系统都不可用
* CAP理论就是说在分布式存储系统中，最多只能实现上面的两点，而由于当前的网络硬件肯定会出现延迟丢包等问题，所以区分容忍性是我们必须需要实现的。所以 我们只能在一致性和可用性之间进行权衡。
* CA 如果被要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但是放弃P的同时也就以为着放弃了系统的扩展性，也就是分布式节点受限，没有办法部署子节点，这是违背分布式系统设计的初衷，
* CP:如果不要求A（可用性）,每个请求都需要在服务器之间保持强一致性，而P（分区）会导致同步时间无限延长（也就是等待数据同步完才能正常访问服务），一旦发生网络故障或者消息丢失情况，就要牺牲用户的体验，等待所欲数据全部一致了之后再让用户访问系统。
* AP：要高可用并允许分区，则需放弃一致性。一旦分区放生，节点之间可能就会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局的数据的不一致性。

## 常见的注册中心：zk、eureka、nacos。应该怎样选择

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | nacos | eureka | consul | zookeeper |
| 一致性 | CP+CA | AP | CP | cp |
| 健康检查 | TCP/HTTP/MYSQL/Client Beat | 心跳 | TCP/HTTP/GRPC/长连接 | KEEP alive长连接 |
| 雪崩保护 | 有 | 有 | 无 | 无 |
| 访问协议 | http/dns | http | http/dns | tcp |
| springcloud集成 | 支持 | 支持 | 支持 | 支持 |

* Zookeeper ：CP设计，保证了一致性，集群搭建的时候，某个节点失败，则会进行选举的leader，或者半数以上的节点不可用，则无法提供服务，因此可用性没法满足。
* Eureka：AP 无主从节点，一个节点挂了，自动切换其他节点可以使用，去中心化。
* 结论：

分布式系统中P,肯定要满足，所以只能在CA中二选一，

没有最好的选择，最好选择是根据业务场景来进行架构设计

如果要求一致性，则选择zookeeper/nacos，如金 融行业cp

如果要去可用性，则Eureka,如电商系统。

CP:适合支付、交易类，要求数据强一致性，宁可业务不可用，也不能出现脏数据。

AP:互联网业务，比如信息流架构，不要求数据强一致性，但要求服务可用性。

## 分布式CAP的权衡结果BASE理论

### 什么是BASE理论

* CAP中的一致性和可用性进行一个权衡的结果，

核心思想就是：我们无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方式使系统达到最终一致性，来自Ebay的架构师提出。

* Basically Available(基本可用)

假设系统，出现了不可预知的故障，但还是能用，可能会有性能或者功能上的影响。

* Soft state(软状态)

允许系统中的数据存在中间状态，并认为该状态不影响整体可用性，即允许系统在多个不同节点的数据副本存在数据延迟时。

* Eventually consistent(最终一致性)

系统能够保证在没有其他的更新操作的情况下，数据最终一定能达到一致的状态，因此所有客户端对系统的数据访问最终都能获取到最终的值。

### 不同数据一致性模型

一般来说，数据一致性模型可以分为强一致性和弱一致性，强一致性也叫线性一致性，除此之外，所有其他的一致性都是弱一致性的特殊情况。弱一致性根据不同的业务场景，又可以分解为更细的模型，不同一致性模型又有不同的应用场景。

在互联网领域的绝大多数场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往值需要保证“最终一致性”，只要在这个最终时间是在用户可以接受的范围内即可。

* 强一致性

当数据更新完成之后，任何多个后续进程的访问都会返回最新的更新过的值，这种是对用户最友好的，就是用户上一次写入什么，下一次就保证读到什么，根据CAP理论，这种实现需要牺牲可用性。

* 弱一致性

系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久可以读到。用户读到某一操作对系统数据的更新需要一段时间，我们称这段时间为“不一致窗口”

* 最终一致性

最终一致性是弱一致性的特例，强调的是所有的数据副本在经过一段时间的同步之后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统的强一致性。

达到强一致性的时间，就是不一致的窗口时间，在没有故障发生的前提下，不一致窗口的时间主要受到通信延迟，系统负载和副本的个数影响。

最终一致性模型根据其提供的不同保证可以划分为更多的模型，包括因果一致性和会话一致性等。

## 为什么要使用alibabaCloud和springCloud 的区别

* SpringCloud和AlibabaCloud 组件存在很大交集，互相配合。
* SpringCloud 很多组件是基于第三方整合，目前多个已经不进行更新了，比如zuul、eureka、hystrix等
* AlibabaCloud提供了一站式微服务解决方法，已经和SpringCloud进行整合，组件互相支持。

## 微服务的熔断降级

### 分布式容错框架

阻止故障的连锁反应 快速失败并迅速恢复 回退并优雅降级 提供近实时的监控与告警

### 熔断

熔断就是为了防止异常扩散，保证系统的稳定性

如果调用某个服务报错（或者挂了），就对该服务熔断，在5分钟内请求此服务直接返回一个默认值，不需要每次都卡几秒，这个过程，就是所谓的熔断。

### 降级

编写好调用失败的补救逻辑，然后对服务直接停止运行，这样这些接口就无法调用，但又不至于直接报错，知识服务水平下降。

但是熔断了之后就会少调用一个服务，此时需要做下标记，标记本来不需要做什么业务，但是因为服务挂了，暂时没有事可做，等该服务恢复了，就可以手工处理这些业务，这个过程，就是所谓的降级。

## springBoot和SpringCloud 的关系

* SpringBoot可以离开SpringCloud单独使用开发项目，但是SpringCloud 离不开springBoot，属于依赖关系
* SpringBoot 专注于快速方便的开发个体微服务，SpringCloud 关注全局的微服务治理框架。
* SpringCloud 是关注全局的微服务协调整理治理框架，它将SpringBoot 开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、微服务发现、断路器、路由、微代理、事件总线、全局锁、决策竟选、分布式会话等等集成服务。

## dubbo是如何做到系统交互的

dubbo底层是通过rpc来完成服务和服务之间的调用的，那dubbo支持很多协议，比如默认的dubbo协议、http协议、rest协议都是支持的。他们底层使用的技术是不太一样的，dubbo协议底层使用的是netty也可以使用mina,http协议底层使用的tomcat或者jetty,服务消费者在调用某个服务时，会将当前所调用的一个服务接口信息，以及方法信息，执行方法所传入的入参数信息等组装成一个invocation对象，然后不同的协议，又通过不同的数据组织方式和传输方式，传送给服务提供者，服务提供者接收到这个对象后，找到对应的一个服务实现利用反射执行对应的方法，得到方法结果后在通过网络响应给服务消费者，duboo在调用的过程中做了很多其他的设计，比如说服务容错、负载均衡、filter机制动态路由机制等等，让dubbo能够处理企业中更多的需求

## 分布式事务

### 2PC

两阶段提交将参与者操作的成败通知协调者，在由协调者根据所有参与者反馈情况决定各参与者是否提交操作，还是中止操作。

#### 第一阶段 准备阶段

事务管理器给每个参与者发送prepare(破派欧)消息，每个参与者要么直接返回失败，要么在本地执行事务，写在本地的redo和undo日志，但不提交，达到一种准备状态。

redo log 是记录数据变更的日志

undo log 保存的是数据的历史版本

1.协调者节点向所有参与者节点询问是否可以执行提交操作，并开启等待各参与者节点的响应。

2.参与者节点执行询问发起为止的所有事务操作，并将undo信息和redo信息写入日志

3.各参与者节点响应协调者节点发起的询问，如果参与者节点的事务操作实际执行成功，则它返回一个同意消息，如果参与者节点的事务操作执行失败，则它返回一个终止消息。

#### 第二阶段 提交回滚阶段

如果事务管理器收到了参与者的失败消息或者超时消息，直接给每个参与者发送回滚消息，否则 发送commit消息。

1.协调者节点向所有参与节点发出回滚操作的请求

2.参与者节点利用之前写入的undo信息执行回滚，并释放在整个事务期间内占用的资源。

3.参与者节点向协调者发送回滚完成消息。

4.协调者节点受到所有参与节点反馈的 回滚完成消息后，取消事务。

#### 2PC协议缺点

1.同步阻塞 1pc准备阶段，只执行sql,不提交数据，并且占用数据库连接资源

2.单点故障，一旦协调者发生故障，参与者会一直阻塞下去。

3.数据不一致，在二阶段提交的阶段中，当协调者向参与者发送commit请求后，发生了网络异常或者commit的过程中发生故障，导致只有一部分数据提交。

## 3PC

三阶段提交，也叫三阶段提交协议，是二阶段提交的改进版，与两阶段提交不同的是，三阶段提交有两个改动点，一、引入超时机制，同时在协调者和参与者中都引入超时机制。二、在第一阶段和第二阶段中插入了一个准备阶段，保证了在最后提交阶段之前各参与阶段的状态是一致的。也就是说，除了引入超时机制之外，3pc把2pc的准备阶段再次一分为二，这样三阶段提交就有了cancommit、Precommit、doCommit三个阶段。

### TCC

* + 将事务提交分为

Try 完成所有业务检查(一致性)，预留必须业务资源（准隔离性）

Confirm：对业务系统做确认提交，默认confirm阶段不会出错的，即只要try成功，confirm就一定会成功

cancel业务执行错误，需要回滚的状态下执行业务取消，预留资源释放，进行补偿。

* + 优点

它把事务运行过程分为try confirm/cancel两个阶段，每个阶段由业务代码控制，这样事务锁力度可以完全自由控制，不存在资源阻塞的问题，每个方法都直接进行事务的提交。

* + 缺点

在业务层编写代码实现的两阶段提交，原本一个方法，现在却需要三个方法支持。对业务的侵入性很强，不能很好的复用。

# SpringBoot

## springBoot启动原理

* SpringApplication.run入口
* 获取实现了SpringApplicationRunListener接口的实现类，通过SPI机制加载, META-INF/spring.factories文件下的类
* 第三方jar包里面的类，commponentScan这个东西就不能扫描到了
* SpringBoot工程自动配置的实现原理是什么

StringBoot，通过SPI加载配置文件的方式，把类加载到Spring容器完成实例化。

SpringBoot常用注解以及实现

第一个@springBootApplication注解，这个标识了这是一个springboot工程，它实际上是另外三个注解的组合，这个三个注@springBootConfiguration,这个注解实际上就是一个@configuration表示启动类也是一个配置类第二个@EnableAutoConfiguration就是向spring容器中导入了一个selector用来加载ClassPath路径下meta-info下面的Spring.factorys中的所定义的一个自动配置类，将这些类自动加载为配置bean，还有第三个@componentScan就是标识扫描路径默认是没有配置这个实际扫描路径springBoot扫描的路径就是启动类所在的一个当前目录，那么还有一些注解比如@bean注解就是用来定义Bean类似于xml中的bean标签，spring在启动时会对加了这个@bean注解的方法进行解析，然后将方法的名字作为beanName并通过执行方法得到bean对象，还有很多其他的注解@controller @service@requestbody @autowired

# 缓存 – redis

## redis 基础

### 为什么要用redis,为啥不用其他缓存，比如memcache呢？

* + redis的数据结构比member更丰富，基本完全可以替代。
  + redis社区比较活跃，性能强大，基本完全可以替代。
  + 也和业务结合，比如电商系统的热销产品，需要用到zset 所以要用redis

## 你用redis 哪些数据结构，说下这些结构的使用场景有哪些？

String

比较简单的k v 场景

设置值 set a 123 取值 get a

hash

存储对象，一个key 多个值

设置值 hmset b age 10 grade 0

取值hgetall b

list

列表型数据，消息队列

设置值 lpush c 张三 wanwu

取值 lrange c 0 3

set

无序集合、去重、交集、并集，比如共同好友，在社交关系方便，数据排重等可以使用

设置值 sadd d java

取值 smembers d

sorted set

有序集合 去重 做榜单

添加值 zadd e 0 wangwu

取值 zrange e 0 10 withscores

## redis 是单线程，为什么这么快？

* + 基于内存，绝大部分请求是存粹的内存操作，cpu不是redis的瓶颈
  + 避免了不必要的CPU上下文切换和其他的竞争条件，比如锁的操作
  + 底层使用的是io复用模型，非阻塞io
  + redis6之后支持多线程，默认不开启

## redis 有哪些持久化方式，分别说下他们的区别。

支持AOF和RDB持久化

### AOF

以日志的形式记录服务器所处理的每一个写 删除操作，查询操作不会记录，以文本的方式记录

支持秒级持久化、兼容性好，对于相同数量的数据集而言，AOF文件通常大于RDB，恢复时间也比RDB慢

### RDB

在指定的时间间隔内将内存中的数据集快照写入磁盘，可以指定时间归档数据，但不能做到实时持久化，

文件紧凑，体积小，对于灾难恢复而言，RDB是非常不错的选择，相比AOF机制，如果数据集很大，RDB在恢复大数据集时的速度比AOF的恢复速度要快。

## 缓存空间不够

### 一般会使用淘汰策略,常见的淘汰策略有FIFO/LRU LFU

### 能否说下FIFO/LRU LFU 这些策略不同点

* 先进先出 FIRST IN FIRST OUT

新访问的数据插入FIFO队列尾部，数据在FIFO队列顺序移动，淘汰FIFO队列头部的数据

* 最近最少使用LAST recently used

根据数据的历史记录来进行数据淘汰，如果数据最近被访问多次，那么将来被访问的频率也更高

把数据加入到链表中，按频次排序，一个数据被访问过，把它的频次+1，发生淘汰的时候，把频次低的淘汰掉。

## 缓存击穿 穿透 雪崩的区别

### 缓存击穿

缓存中没有但是数据库中有的数据，假如是热点数据，那KEY在缓存过期的一刻，同时与大量的请求，这些请求都会击穿到DB，造成瞬间的DB请求量大，压力增大。

和缓存雪崩的区别在于这里针对某一keu缓存，后者则是很多key.

预防：设置热点过期数据不过期，定时任务定时更新缓存，或者设置互斥锁。

### 缓存穿透

查询一个不存在的数据，由于缓存不命中的，并且处于容错考虑，如发起为id为-1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 每次请求都要到存储层去查询，失去了缓存的 意义，存在大量查询 不到的数据，可能 db就 挂掉，这也是 黑客利用 不存在 的key频繁将KEY-value 对而成key-null，设置断点的过期时间 ，防止 同个key一直被攻击。

### 缓存雪崩（多个热点key都过期）

大量的key设置了相同的过期时间，导致 在缓存 在 同一时刻全部失效，造成短瞬时db请求量大、压力骤增，引起雪崩。

预防：存数据的 过期时间设置 随机，不要 在 某一时间 内同时失效，设置热点数据永不 过期，定时任务定期更新。

## redis事务

* redis事务就是一次性、顺序性的执行一个队列中的一系列的命令，说白了就是一个连贯操作。
* 重点：单条命令是原子性执行的，但事务不保证原子性，且没有自动回滚的功能，事务中任意命令失败，其余的命令仍会被执行。

### redis事务命令

* multi:标记一个事务开始，相当于mysql的begin transmanger
* exec:执行所有事务块的命令
* discard：取消事务，放弃事务块中的所有命令

## reids 集群方案 redis提供了三种集群策略

如果说存储的数据量不大就用哨兵模式，如果说数据量存的特别大并且持续的扩容就选择cluster模式

### 第一种主从模式，

这种模式比较简单，主库可以读写，并且和从库进行一个数据同步的操作，这种模式下，客户端直接连接主库或者某个从库，但是当主库或者从库宕机后，客户端需要自己手动的去修改IP,另外这种模式比较难以进行扩容，整个集群受到了这个机器的内存容量的限制。所以不可能支持这种特大数据量，所以就有了第二种哨兵模式

### 哨兵模式

这种模式在主从的基础上，新增了一个哨兵节点，就是当我们的主库节点宕机之后，哨兵会发现主节点宕机后，然后就在从库中自动选择一个库作为一个新的主库，另外哨兵也可以做集群，就是保证某一个哨兵节点宕机后，还有其他哨兵可以进行工作，这种模式比较好的保证redis集群的一个高可用，但是仍然不能很好的去解决redis容量的上限问题。然后就有了第三种模式cluster模式

### cluster模式

就是用的比较多的模式，他支持多主多从，这种模式会按照key进行槽位的分配可以得到不同的key分散到不同的一个主节点上利用这种模式可以使得整个集群支持更大的一个数据容量，同时每个主节点也可以拥有自己的多个从节点，如果某个主节点宕机后会从其他的从节点中选举出一个新的成为主节点这就是三种集群模式。

# Spring

## spring核心理念

### 能否介绍下spring框架的di和ioc

子主题 1

IOC控制反转，指的是将对象的创建权交给spring容器进行管理。

DI依赖注入，指spring创建对象的过程中，将对象依赖属性通过配置进行注入，不能单独存在，需要在ioc的基础上进行实现。

依赖注入DI和控制反转是从不同的角度描述同一件事情，通过引入ioc容器，利用依赖关系注入的方式，实现对象之前的调用

### spring里面的单例和多例，scope作用域。

singleton:单例，spring默认bean的实例是单例，调用getBean 方法返回是同一个对象会被缓存起来，效率比较高，当一个bean被标识为singleton时候，spring的ioc容器中只会存在一个实例。

prototype:多例，调用getBean 方法创建不同的对象，会频繁的创建和销毁对象造成很大的开销。

其他少用，作用域只在webapplicationcontext

request 每个http请求都会创建一个新的bean

session 每个http session请求都会创建一个新的bean

global session 基本不用

### bean实例化过程

1、实例化--》普通对象

2、填充属性BService --->单例池--》创建BService

1.实例化---》普通对象

2.填充AService ---》单例池Map-创建BserVice

3填充其他属性.

4.做一些其他的事情（AOP）

5.添加到单例池

### spring三级缓存

第一级缓存singletonObjects

第二级缓存 earlySingletonObjects

第三季缓存 singletonFactories

## springAOP

### 概念

* aspect 切面

声明切面，类似于类定义，每个aspect可以包含多个pointcut 和advice

* pointcut 切点

用来筛选我们主要在哪些地方进行增强逻辑

* joinpoint 连接点

通过切点筛选出来的主要进行增加逻辑的地方

* advice 通知

我们对筛选出来的连接点肯定要进行额外的操作，这些额外的操作就是advice

@before

方法执行之前执行

@around

围绕着方法执行

@after

在方法执行后执行

@afterReturning

在方法返回之后执行

@afterThrowing

在方法抛出异常之后执行

### 什么是动态代理 sprimgAop 使用什么代理

在程序运行时，运用反射机制创建而成，无需手动编写代码

springAop 里面常用的是JDK动态代理，CGLIB动态代理。

### 能否说下JDK动态代理和CGLIB动态代理的区别

JDK动态代理，要求目标对象实现一个接口，但是有时候目标对象只是一个单独的对象，并没有实现任何接口，这个时候就可以用CGLIB动态代理

jdk动态代理是自带的，CGlib则需要引入第三方包

cglib动态代理，它是在内存中构建一个子类对象从而实现对目标对象功能的扩展。

CGlib动态代理基于继承来实现代理，所以无法对final prvate 和static 方法来实代理

### 说下spring aop 中的代理使用的默认策略

如果目标对象有实现接口，则默认使用jdk动态代理

如果目标对象没有实现接口，则采用cglib方式动态代理

如果目标对象实现了接口，程序里面依然可以指定cglib 进行代理。

## BeanPostProcessor的作用

.收集@Resource@Autowired@Value@PostConstruct，@PreDestroy 注解的方法和属性埋点

## spring源码

第一spring是一个快速开发框架，它是帮助程序员管理对象的，

第二spring的源码是非常优秀的，设计模式的应用，并发安全的实现，面向接口的设计（这部分无需背）

就是在创建spring容器，也就是启动spring时，首先进行扫描得到一个所有的beanDefinition对象，并且存在一个map中，然后筛选出非懒加载的单例BeanDefinition去创建bean，对于多例bean就是不需要在启动的过程中去进行创建，对于多例bean会在每次获取bean的时候利用beanDefinition去创建Bean,就是Bean的创建生命周期，这期间包括合并了beanDefintion,推断构造方法 属性填充 初始化前 初始化 初始化后等步骤，那么AOP就是放生在初始化后这一步骤，单例Bean创建完成之后，spring会触发一个容器的启动事件，到此spring就启动完成

## spring事务传播机制

多个事务互相调用的时候，事务如果在这些方法中如何传递，spring提供了7种不同的传播机制。

REQUIRED:默认的传播机制如果没有事务则新建一个事务，如果当前存在事务，则加入这个事务。

SUPPORTS:当前存在事务，则加入当前事务，如果没有事务，则以非事务的方式执行。

MANDATORY:当前存在事务，则加入事务，如果当前不存在事务，则抛异常。

REQUIRED\_NEW:创建一个新的事务，如果存在当前事务，则挂起事务。

NOT\_SUPPORTED:以非事务的方式执行，如果存在当前事务，则挂起当前事务。

NEVER:不使用事务，如果当前事务存在，则抛异常。

NESTED:如果当前事务存在，则在嵌套事务中执行，否则和REQUIRED一样。

## spring组件实例化讲解

### bean的生命周期

Bean生命周期描述的是spring中一个bean创建过程和销毁过程中锁经历的步骤，其中bean创建过程是重点，程序员可以利用bean生命周期机制对bean进行自定义加工。

1.beanDefinitionBean定义

beanDefinition表示bean定义，它定义了某个bean的类型，Spring就是利用BeanDefinition来创建bean的，比如需要利用BeanDefinition中的beanClass属性确定Bean的类型，从而实例化出来对象。

2.构造方法推断选出一个构造方法

一个bean中可以有多个构造方法，此时就需要spring来判断到底使用哪个构造方法，这个过程比较复杂的。通过构造方法推断之后确定一个构造方法后，就可以利用构造方法实例化得到一个对象。

3.实例化构造方法反射得到对象

通过构造方法反射得到一个实例化对象，在spring中，可以通过beanPostProcessor机制对实例化进行干预。

4.初始化后AOP生成代理对象

初始化后就是bean创建生命周期中最后一个步骤，我们常说的aop机制，就是在这个步骤中通过BeanPostProcesso机制实现的，初始化之后得到的对象才是真正的bean对象。

5.初始化 对其他属性赋值、校验

6.属性填充，给属性进行自动填充

实例化所得到的是不完整的对象，不完整的意思是该对象中的某些属性还没有进行属性填充，也就是spring还没有自动给某些属性赋值，属性填充就是我们通常说的自动注入，依赖注入。

### bean的创建过程:

UserService->无参构造方法-> 普通对象->依赖注入->初始化前@postConstruct->初始化(InitializingBean) ->初始化后(AOP) ->代理对象->放入单例池map singletonObjects->bean对象

### @autoWired是如何工作的

@autowired加在某个属性上，Spring在进行bean的生命周期过程中，在属性填充这一步，会基于实例化出来的对象，对该对象加上了@autowired的属性自动给属性赋值。

spring会先根据属性的类型去spring容器中找出该类型所有的Bean对象，如果找出来多个，则在根据属性的名字从多个中在确定一个，如果requied属性为true，并且根据属性信息找不到对象，则直接抛异常。

## @beanFactory和Application的区别

### 相同

Spring 提供了两种不同的ioc容器，一个是BeanFactory，另一个是ApplicationContext，他们都是java interFace，applicationContext继承于BeanFabory（applicatonContext 继承ListAbleBeanFactory）

他们都可以用来配置xml属性，也支持属性的自动注入。

而ListableBeanFactroy继承BeanFactroy，BeanFactory和ApplicatonContext，都提供了一种方式，使用getBean 获取bean

### 不同

当你调用getBean（）方法是，beanFactory仅实例化bean，而ApplicationContext在启动容器的时候实例化单例Bean,不会等待调用getBean（）方法是在实例化。

# Mybatis

## jdbc连接数据库的开发步骤

加载数据库驱动

获取数据库连接对象

获取语句对象

会话对象有两种statement和preparedStatement执行语句

preparedStatement 是预编译方式

效率高于statement,防止sql注入，

preparedStatement是占位符方式非拼接方式，提高了可读性。

处理结果集

关闭资源

## 简单说下mybatis加载的流程

每个基于mysqlbatis的应用都是以一个sqlsessonFactory的实例为核心

sqlSessionFactory的实例可以通过sqlSessionFactoryBuilder获得

sqlSessonFactoryBuilder可以从xml获取配置文件或者一个预先配置的configuration实例俩构建出sqlSessionFactory 实例

工厂设计模式里面需要获取SqlSession，里面提供了数据库执行Sql命令锁需的所有方法。

# http协议

## 常见的http method 有哪些，使用场景

### http1.0 定义了三种

GET 向游览器获取资源，比如常见的查询请求

POST 向服务器提交表单数据

HEAD 和get类似，返回响应中没有具体内容，用于获取请求头信息

### http 2.0

put 一般用来修改请求，比如个人信息，商品信息修改

patch PUT 的补充，更新指定资源 下的数据

Delte 用书删除请求

options 获取服务器支持的http请求方法，服务器性能和跨域检查等

cennect 方法的作用就是服务器作为跳板，让服务器代替用户去访问其他网页，之后把数据原原本本的返回给用户，网页开发基本不用这个方法，如果是http代理就会使用这个，让服务器用户去访问其他网页 资源信息。

tarce 回显服务器收到的请求，主要用于测试或诊断

## 常见的http 状态码

1XX 收到请求，需要请求者继续执行操作，比较少见

2XX 请求成功 常用的200

3XX 重定向，游览器在拿到服务器返回的这个状态码后会自动跳转到新的url地址，这个地址可以从响应的location首部中获取

4XX 客户端出错，请求包含语法错误和无法完成请求

5XX 服务端出错，服务器在处理请求的过程中发生了错误

## http协议无状态解决方案cookie和session

### 单机情况下http无状态解决方案，cookie和session知识点

cookie数据保存在客户端，session保存在服务端

cookie没有安全性容易泄密，不能直接存储信息

cookie的大小和数量有限制

### 你们公司C端业务登录时怎么样做的

部分采用redis替代本身的tomcat单机session（业务需要高度可控）

还有部分业务使用 JSON WEB TOKEN（C端普通业务）

## JWT

### 简述

JWT是一个开放标准，它定义了一种用于简洁，自包含的用于通信双方之间以JSON对象的形式安全传递信息的方法，可以使用hMAC算法或者是RAS的公钥秘钥对进行签名

### JWT格式组成头部、负载、签名

heear +payload+signtrue

头部

主要是描述签名算法

负载

主要描述是加密对象的信息，如用户的id等，也可以加些规范里面的东西，如iss签发者，exp过期时间，sub面向的用户

签名

主要是把前面两部分进行加密，防止别人拿到token进行base进行解密后篡改token

简单的来说就是通过一定的规范生成token，然后通过可以解密的算法逆向解密token，这样就可以获取用户信息

### JWT为啥要使用这个，它的优点 缺点

#### 优点

生产的token可以包含基本信息，比如id，用户昵称、头像等信息，避免再次查库

存储在客户端，不占用服务端的资源，使用加解密的方式进行校验，在分布式业务中能较好的提高性能和节省空间。

#### 缺点

token是经过base64编码，所以可以在解码，因此token加密前的对象不应该包含敏感信息，如用户权限，密码等

如果没有服务端存储，则不能做登录失效处理，除非服务端该秘钥

### 生成的token,在客户端或者游览器怎么存储的

可以存储到cookie localstorage 和sessionStorage里面

## 游览器的同源策略和跨域知识点

### 同源策略

是一种约定，它是游览器最核心也最基本的安全功能，如果缺少了同源策略，则游览器的正常功能可能就会受到影响。

由netspace剔除的一个著名安全策略，当一个游览器的两个tab页中分别打开来，百度和谷歌的页面，当游览器的百度tap页面执行一个脚本的时候会检查这个脚本属于那个页面的，即检查是否同源，只有和百度同源的脚本才会被执行。如果非同源，只有和百度同源的脚本才会被执行。如果非同源，那么在请求数据的时，游览器会在控制台报一个异常，提示拒绝访问。同源策略是游览器行为，是为了保护本地数据比被javaScript代码所获取的数据污染，因此拦着的是客户端发出来的请求回来的数据接收，即请求来了，服务器响应了，但是无法被游览器接收。

游览器同源策略1995年，同源政策由netspace公司引入游览器，目前，所有游览器都实行这个政策。最初，它的含义是指，A网页设置的cookie，B网页不能打开，除非这两个网页同源。所谓同源指的是三个相同

协议相同 https http\.

端口 80 88

域名相同 baidu.com google.com

### 为什么出现跨域，有什么常见的解决办法

不同源即为跨域

#### jsonp（只支持get请求，支持老的IE浏览器）适合加载不同域名的js、css，img等静态资源；

原理：就是利用<script>标签没有跨域限制，通过<script>标签src属性，发送带有callback参数的GET请求，服务端将接口返回数据拼凑到callback函数中，返回给浏览器，浏览器解析执行，从而前端拿到callback函数返回的数据。

#### 缺点：只能发送get一种请求。

#### CORS（支持所有类型的HTTP请求，但浏览器IE10以下不支持）适合做ajax各种跨域请求；

#### Nginx代理跨域和nodejs中间件跨域原理都相似，都是搭建一个服务器，直接在服务器端请求HTTP接口，这适合前后端分离的前端项目调后端接口。

# Mysql

## 事务的ACID是什么

事务的四大特性ACID

### 原子性（atomicity）:

一个事务必须分割成不可分割的最小工作单元，整个操作要么全部成功，要么全部失败，一般就是通过commit和rollback来控制。

### 一致性Consistency:

数据库总能从一个一致性状态转换到另一个一致性状态，比如小滴课堂付款成功后开通视屏播放权限，只要有一方发生异常另一方都不会提交事务。

### 隔离性Isolation:

一个事务相对于另一个事物是隔离的，一个事务所做的修改是在最终提交以前，对其他事务是不可见的。

### 持久性（durability）:

一旦事务提交，则其所做出的的修改，会永久保存到数据库中，此时即使是系统崩溃，修改的数据也不会丢失。

## 数据库的隔离级别

### 能否简单解释一下脏读、不可重复读、幻读的意思

* 脏读

脏读：事务中修改未提交的数据，别的事务能够读取到未提交的数据称为脏读。

* 不可重复读

当某个事务前后多次读取，不能读到相同的数据内容，中间另一个事务读到未提交的数据为脏读。

* 幻读：

某个事务查询某个范围，多次查询的结果不一致。

### 常见的隔离级别由低到高有哪几种,mysql默认是哪种

事务隔离级别越高，事务越安全，但是并发能力越差。

#### read uncomitted（未提交读，读取未提交内容）

事务中的修改即使没有提交，其他事务也能看见，事务可以读到未提交的数据称为脏读，也存在不可重复读幻读问题。

例子：小滴课堂运营配置了一个活动，原价500的课程，配置成了50，但是事务没有提交，你刚好看到了这个课程那么便宜准备购买，但是anna小姐姐马上回滚了事务，重新配置并提交了事务，你准备下单的时候价格恢复到了500元。

#### read committed 提交读 读取已提交的数据

一个事务开始后只能看见已经提交到的事务所做的修改，在事务中执行两次同样的查询可能得到不一样的结果，也叫做不可重复读（前后多次读取，不能读取到相同的数据内容），也存在幻读。

#### repeatable 可重复读，mysql默认的事务隔离级别

解决脏读、不可重复读的问题，存在幻读的问题，使用mmvc机制 实现可重复读

事例:老王在小滴课堂有1000积分，准备去兑换《面试专题》，查询数据库确实有1000积分，老王的女朋友同时也在别的地方兑换了《这个面试专题课程》，老王的事务提交的时候发现存在了，之前读取的没用了，出现幻读。

#### 幻读的问题：mysql 的innoDB引擎通过了mvcc自动帮我们解决，即是多版本并发控制。

## 索引

创建索引的时候主要考虑啥，使用索引的优缺点有哪些，使用应该注意什么？

### 索引的好处

快速定位到表的位置，减少服务器扫描的索引

有些索引存储了实际的值，特定情况下只要使用索引就能完成查询

### 索引的缺点

索引会浪费磁盘空间，不要创建非必要的索引

插入、更新、删除需要维护索引，带来额外的开销

索引过多，修改标的时候重构所以性能差

## 索引优化实践

前缀索引，特别是text和blog类型的字段，只检索前面几个字符，提高检索速度

尽量使用数据量少的索引，索引值过长查询速度会受到影响

选择合适的索引列顺序

内容变动少，且查询频繁，可以建立索引

内容变动频繁，谨慎创建索引

根据业务创建合适的索引类型，比如某个字段常用来作查询条件，则为这个字段建立索引提高查询速度

组合索引选择业务查询最相关的字段

## 数据库设计查询和上线里面的坑,数据库查询关键词执行顺序

数据查询的指令有多个，说下执行顺序select where from group having order by

from

来自那张表

where

初步过滤查询条件

group by

过滤后进行分组

having

对分组后的数据进行二次过滤

select

查看哪些结果集

order by

按照怎样的顺序进行排序返回

## mysql中的datetime和datestmap有什么区别

datetime

字节 8字节

范围 1000 -9000

存储与时区无关，不会发生改变

timestmap

字节 4字节

范围 1970-2038

存储的是与时区有关，随数据库的时区而发生改变

为什么timestmap只能到2038年

mysql的timestmap类型是4个字节，最大值是2的31次方减1，结果是2147483647，转换成北京时间就是2018-01-19 11:14:07

## mysql 常见日志考查

redo 重做日志

作用：确保事务的持久性，防止在发生故障，脏页未写入磁盘。重启数据库会进行redo log执行重做，到达事务一致性。

undo 回滚日志

作用：保证数据的原子性，记录事务发生之前的数据的一个版本，用户回滚。

innodb事务的可重复读和读取已提交 隔离级别就是通过mvcc \_undo实现。

errorlog 错误日志

作用：mysql 本身启动 停止 运行期间发生的错误信息

slow query log 慢查询日志

作用：记录执行时间过长的sql 时间阀值可以配置，只记录执行成功

binlog 二进制日志

作用： 用于主从复制 实现主从同步

relay log 中继日志

用于数据主从同步，将主库发送过来的binlog先保存在本地，然后从库进行回放

general log 普通日志

作用：记录数据库操作明细，默认关闭，开启会降低数据性能。

# 消息中间件

## 优缺点:

是否了解主流的消息队列，引入队列有啥缺点，对比其他消息中间件产品，选择这款的原因是啥。

优点：解耦，异步化，削峰填谷

缺点系统可用性降低、复杂度增高、维护成本增高

## 主流消息队列Apache activeMQ、kafka、rabbitMq、rocketMq

apacheMq:

Apache出品，历史悠久，支持多种语言的客户端和协议，多种语言支持java net c++等，基于JMS provider的实现。缺点吞吐量不高，多队列的时候性能下降，存在消息丢失的情况，比较少大规模的使用。

kafka

是由apache软件基金会开发的一个开源流处理平台，由scala和java编写，kafka是一种高吞吐量的分布式发布订阅消息系统，它可以处理大规模的网站中所有动作流数据（网页游览，搜索和其他用户的行动），副本集机制，实现数据冗余，保障数据尽量不丢失；知识多个生产者和消费者。

缺点：不支持批量和广播消息，运维难度大，文档比较少，需要掌握scala

### rabbitmq

是个一个开源的amqp实现，服务器端用的Erlang语言编写，支持多种客户端，如：python/ ruby .net java jms c勇于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面变现不错。

缺点:使用erlang开发，阅读和修改源码难度大

### rocketMq

阿里开源的一款消息中间件，纯java开发，具有高吞吐量，高性能、高可用、审核大规模分布式系统应用的特点，性能强劲（零拷贝技术），支持海量堆积，支持指定次数和时间净额的失败消息重发，支持consumer端tag过滤、延迟消息等，在阿里内部进行大规模使用，适合在电商，互联网金融等领域使用。

确定：成熟的资料相对不多，社区处于新生状态但是热度很高。

## mq分布式事务

### rocketMq事务消息：

#### rocketmq提供分布式事务功能，通过rocketmq事务消息能达到分布式事务的最终一致。

#### 半消息half message

暂不能投递的消息（暂不能消费），product已经将消息成功发送到了broker端，但是服务端未收到生产者对该消息的二次确认，此时该消息被标记成“暂不能y投递”,处于该种状态下的消息即半消息

#### 消息回查

由于网络闪断、生产者应用重启等原因导致某条事务消息的二次确认丢失，消息队列rocketmq服务端通过扫描发现某条消息长期处于“半消息”时，需要主动向生产者询问该消息的最终态（commit或是rollback），该过程即消息回查。

### 整体交互流程

* product向borker端发送消息。
* 服务端将消息持久化成功之后，向发送方返回ACK确认消息已经发送成功，此时消息为半消息。
* 发送方执行本地事务逻辑
* 发送方根据本地事务执行结果向服务端提交二次确认（commit或者rollback），服务端收到commit状态则将半消息标记为可投递，订阅方最终将收到该消息；服务端收到rollback状态则删除半消息，订阅方将不会消费该消息。
* 在断网或者应用重启的特殊情况下，上述步骤4提交的二次确认最终未达到服务端，经过固定时间后服务端将对该消息发起消息回查。
* 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
* 发送方根据检查得到的本地事务的最终状态再次提交二次确认，服务端仍按照步骤对消息进行操作。

### rocketmq事务消息的状态

commit\_message

提交事务消息，消费者可以消费此消息

rollback\_message

回顾事务消息，消息会在broker中删除，消费者不能消费。

unknow

borker需要回查确认消息的状态。

### 消息队列怎么避免重复消费

rocketMq不保证消息不重复，如果你的业务需要保证严格的不重复消费，需要你自己在业务端去重。

接口幂等性保障，消费端处理业务消息要保持幂等性。

#### redis

* senx（）做消息id去重java版本目前不支持设置过期时间。扩展（如果再用expire则不是原子操作，可以使用lua脚本实现）
* incr原子操作，key自增，大于0返回值大于0说明被消费过。
* 上述两种方式都可以，但是不能用于分布式锁，考虑原子性问题，但是排重可以不考虑原子性问题。数据量多则需要设置过期时间。

### rocketmq如何保证消息的可靠性传输

* product端

不采用oneway发送，使用同步或者异步方式发送，做好重试，但是重试的messagekey必须唯一

投递的日志需要保存，关键字、投递时间，投递状态，重试次数，请求体，响应体

* broker端

双主双从架构，nameServer需要多节点

同步双写、异步刷盘可靠性更高，但是性能差，根据业务选择

* consumer端

消息消费务必保留日志，即消息的元数据和消息体。

消费端务必做好幂等性处理

* 投递到broker端后

机器断电重启：异步刷盘，消息丢失；同步刷盘消息不丢失

硬件故障：可能存在丢失，看队列架构

### 消息队列发生大量堆积应该怎么处理

临时topic队列扩容，并提高消费者能力，但是如果增加consumer数量，但是堆积topic里面的messagequeue数量固定，过多的consumer不能分配到messagequeue

编写临时处理分发程序，从旧的topic快速读取到临时新topic中，新topic的queue数量扩容多倍，然后在启动多consumer进行临时新的topic里消费。

### 如何保证消息队列里消息生成和消费的顺序性

* 什么是顺序消息

消费的生产和消费顺序一致

* 全局顺序：

topic下面全部消息都要有序（少用），性能要求不高，所有的消息都严格按照fifo原则进行消息发布和消费的场景，并行度成为消息系统的瓶颈，吞吐量不高。

在证券处理中，以人民币兑换美元为例子，在价格相同的情况下，先出价者优先处理，则可以通过全局顺序的方式按照fifo的方式进行发布和消费。

* 局部顺序：

只要保证一组消息被顺序消费即可，性能要求高

使用场景：电商的订单创建，同一个订相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息、订单交易成功消息、都会按照先后顺序来发布和消费。

* 顺序发布：

对于指定的一个topic，客户端将按照一定的先后顺序发送消息：举例：订单的顺序流程是：创建、付款、物流、完成、订单号相同的消息会被先后发送到同一个队列中，根据messageQueueSelector里面自定义策略，根据同个业务id防止到同个queue里面，如果订单号取模运算在放到selector中，同一个模的值都会投递到同一个queue

* 顺序消费：

对于指定的一个topic，按照一定的先后顺序接受消息，即先发送的消息一定会被客户端所接收到。

举例：消费端在保证同个topic里的同个队列，不应该用messageLitenterConcurrently,应该使用messageListenterOrderLy,z自带单线程消费消息，不能再用consumer端再使用多线程去消费，消费端分配到的queue数量是固定的，集群消费会所在当前正在消费的队列集合的消息，所以会保证顺序消费。

* 注意：

顺序消费咱不支持广播模式

顺序消息不支持异步发送方式，否则将无法严格保证顺序。

# JAVA基础

## Java基础

### 2、== 和 [equals](https://so.csdn.net/so/search?q=equals&spm=1001.2101.3001.7020) 的区别是什么？

1. 对于基本类型，==比较的是值；
2. 对于引用类型，==比较的是地址；
3. equals不能用于基本类型的比较；
4. 如果没有重写equals，equals就相当于==；
5. 如果重写了equals方法，equals比较的是对象的内容；

### 3、final 在 java 中有什么作用？

（1）用来修饰一个引用

1. 如果引用为基本数据类型，则该引用为常量，该值无法修改；
2. 如果引用为引用数据类型，比如对象、数组，则该对象、数组本身可以修改，但指向该对象或数组的地址的引用不能修改。
3. 如果引用时类的成员变量，则必须当场赋值，否则编译会报错。

（2）用来修饰一个方法

当使用final修饰方法时，这个方法将成为最终方法，无法被子类重写。但是，该方法仍然可以被继承。

（3）用来修饰类

当用final修改类时，该类成为最终类，无法被继承。

 比如常用的String类就是最终类。

### 6、String str="i"与 String str=new String(“i”)一样吗？

String str="i"会将起分配到常量池中，常量池中没有重复的元素，如果常量池中存中i，就将i的地址赋给变量，如果没有就创建一个再赋给变量。

String str=new String(“i”)会将对象分配到堆中，即使内存一样，还是会重新创建一个新的对象。

### 12、普通类和抽象类有哪些区别？

抽象类不能被实例化；  
抽象类可以有抽象方法，只需申明，无须实现；  
有抽象方法的类一定是抽象类；  
抽象类的子类必须实现抽象类中的所有抽象方法，否则子类仍然是抽象类；  
抽象方法不能声明为静态、不能被static、final修饰。

### 13、接口和抽象类有什么区别？

（1）接口

接口使用interface修饰；  
接口不能实例化；  
类可以实现多个接口；

①java8之前，接口中的方法都是抽象方法，省略了public abstract。②java8之后；接口中可以定义静态方法，静态方法必须有方法体，普通方法没有方法体，需要被实现；

（2）抽象类

抽象类使用abstract修饰；  
抽象类不能被实例化；  
抽象类只能单继承；  
抽象类中可以包含抽象方法和非抽象方法，非抽象方法需要有方法体；  
如果一个类继承了抽象类，①如果实现了所有的抽象方法，子类可以不是抽象类；②如果没有实现所有的抽象方法，子类仍然是抽象类。

### 17、什么是反射？

所谓反射，是java在运行时进行自我观察的能力，通过class、constructor、field、method四个方法获取一个类的各个组成部分。

在Java运行时环境中，对任意一个类，可以知道类有哪些属性和方法。这种动态获取类的信息以及动态调用对象的方法的功能来自于反射机制。

### 19、为什么要使用克隆？如何实现对象克隆？深拷贝和浅拷贝区别是什么？

（1）什么要使用克隆？

想对一个对象进行复制，又想保留原有的对象进行接下来的操作，这个时候就需要克隆了。

（2）如何实现对象克隆？

实现Cloneable接口，重写clone方法；  
实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深克隆。  
BeanUtils，apache和Spring都提供了bean工具，只是这都是浅克隆。

（3）深拷贝和浅拷贝区别是什么？

浅拷贝：仅仅克隆基本类型变量，不克隆引用类型变量；  
深克隆：既克隆基本类型变量，又克隆引用类型变量；

### 20、throw 和 throws 的区别？

（1）throw

作用在方法内，表示抛出具体异常，由方法体内的语句处理；  
一定抛出了异常；

（2）throws

作用在方法的声明上，表示抛出异常，由调用者来进行异常处理；  
可能出现异常，不一定会发生异常；

### 21、final、finally、finalize 有什么区别？

final可以修饰类，变量，方法，修饰的类不能被继承，修饰的变量不能重新赋值，修饰的方法不能被重写

finally用于抛异常，finally代码块内语句无论是否发生异常，都会在执行finally，常用于一些流的关闭。

finalize方法用于垃圾回收。

一般情况下不需要我们实现finalize，当对象被回收的时候需要释放一些资源，比如socket链接，在对象初始化时创建，整个生命周期内有效，那么需要实现finalize方法，关闭这个链接。

但是当调用finalize方法后，并不意味着gc会立即回收该对象，所以有可能真正调用的时候，对象又不需要回收了，然后到了真正要回收的时候，因为之前调用过一次，这次又不会调用了，产生问题。所以，不推荐使用finalize方法。

### 23、常见的异常类有哪些？

1. NullPointerException：空指针异常；
2. SQLException：数据库相关的异常；
3. IndexOutOfBoundsException：数组下角标越界异常；
4. FileNotFoundException：打开文件失败时抛出；
5. IOException：当发生某种IO异常时抛出；
6. ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出此异常；
7. NoSuchMethodException：无法找到某一方法时，抛出；
8. ArrayStoreException：试图将错误类型的对象存储到一个对象数组时抛出的异常；
9. NumberFormatException：当试图将字符串转换成数字时，失败了，抛出；
10. IllegalArgumentException 抛出的异常表明向方法传递了一个不合法或不正确的参数。
11. ArithmeticException当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。

### 24、hashcode是什么？有什么作用？

Java中Object有一个方法：

public native int hashcode();

（1）hashcode()方法的作用

hashcode()方法主要配合基于散列的集合一起使用，比如HashSet、HashMap、HashTable。

当集合需要添加新的对象时，先调用这个对象的hashcode()方法，得到对应的hashcode值，实际上hashmap中会有一个table保存已经存进去的对象的hashcode值，如果table中没有改hashcode值，则直接存入，如果有，就调用equals方法与新元素进行比较，相同就不存了，不同就存入。

（3）重写equals方法时，一定要重写hashcode方法

### 30、什么是 Java 的内存模型?

在了解什么是 Java 内存模型之前，先了解一下为什么要提出 Java 内存模型。

之前提到过并发编程有三大问题

CPU 缓存，在多核 CPU 的情况下，带来了可见性问题  
操作系统对当前执行线程的切换，带来了原子性问题  
译器指令重排优化，带来了有序性问题  
为了解决并发编程的三大问题，提出了 JSR-133，新的 Java 内存模型，JDK 5 开始使用。

简单总结下

Java 内存模型是 JVM 的一种规范  
定义了共享内存在多线程程序中读写操作行为的规范  
屏蔽了各种硬件和操作系统的访问差异，保证了 Java 程序在各种平台下对内存的访问效果一致  
解决并发问题采用的方式：限制处理器优化和使用内存屏障  
增强了三个同步原语（synchronized、volatile、final）的内存语义  
定义了 happens-before 规则

### 46、迭代器 Iterator 是什么？

为了方便的处理集合中的元素,Java中出现了一个对象,该对象提供了一些方法专门处理集合中的元素.例如删除和获取集合中的元素.该对象就叫做迭代器(Iterator)。

### 47、Iterator 怎么使用？有什么特点？

Iterator 接口源码中的方法：

1. java.lang.Iterable 接口被 java.util.Collection 接口继承，java.util.Collection 接口的 iterator() 方法返回一个 Iterator 对象
2. next() 方法获得集合中的下一个元素
3. hasNext() 检查集合中是否还有元素
4. remove() 方法将迭代器新返回的元素删除

### 48、Iterator 和 ListIterator 有什么区别？

（1）ListIterator 继承 Iterator

（2）ListIterator 比 Iterator多方法

1. add(E e)  将指定的元素插入列表，插入位置为迭代器当前位置之前
2. set(E e)  迭代器返回的最后一个元素替换参数e
3. hasPrevious()  迭代器当前位置，反向遍历集合是否含有元素
4. previous()  迭代器当前位置，反向遍历集合，下一个元素
5. previousIndex()  迭代器当前位置，反向遍历集合，返回下一个元素的下标
6. nextIndex()  迭代器当前位置，返回下一个元素的下标

（3）使用范围不同，Iterator可以迭代所有集合；ListIterator 只能用于List及其子类

1. ListIterator 有 add 方法，可以向 List 中添加对象；Iterator 不能
2. ListIterator 有 hasPrevious() 和 previous() 方法，可以实现逆向遍历；Iterator不可以
3. ListIterator 有 nextIndex() 和previousIndex() 方法，可定位当前索引的位置；Iterator不可以
4. ListIterator 有 set()方法，可以实现对 List 的修改；Iterator 仅能遍历，不能修改。

### 56、线程的run()和start()有什么区别？

每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，run()方法称为线程体。通过调用Thread类的start()方法来启动一个线程。

start() 方法用于启动线程，run() 方法用于执行线程的运行时代码。run() 可以重复调用，而 start() 只能调用一次。

### 57、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？

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来！

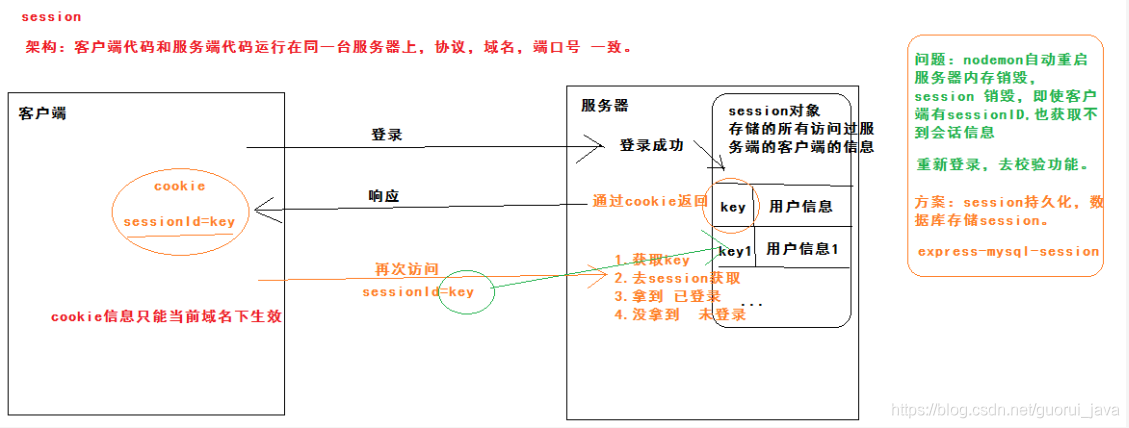
new 一个 Thread，线程进入了新建状态。调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。

而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结： 调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

### 95、cookie、session、token

1、session机制



session是服务端存储的一个对象，主要用来存储所有访问过该服务端的客户端的用户信息（也可以存储其他信息），从而实现保持用户会话状态。但是服务器重启时，内存会被销毁，存储的用户信息也就消失了。

不同的用户访问服务端的时候会在session对象中存储键值对，“键”用来存储开启这个用户信息的“钥匙”，在登录成功后，“钥匙”通过cookie返回给客户端，客户端存储为sessionId记录在cookie中。当客户端再次访问时，会默认携带cookie中的sessionId来实现会话机制。

（1）session是基于cookie的。

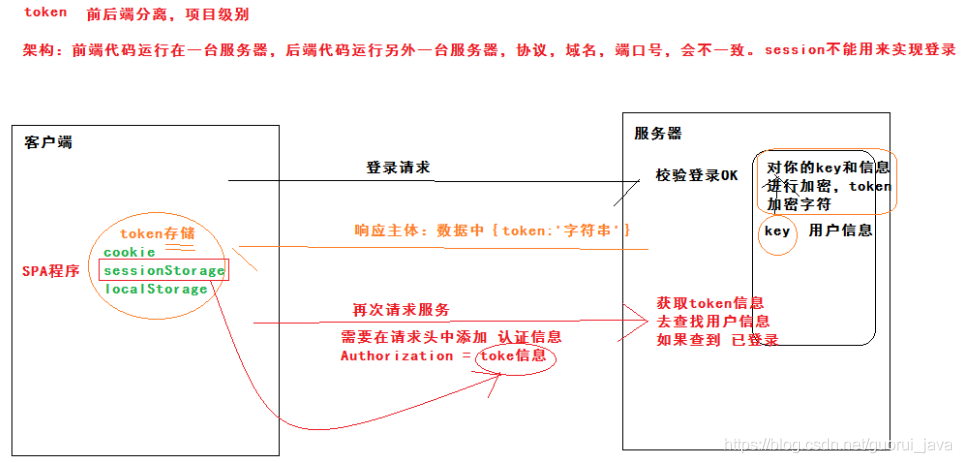
* cookie的数据4k左右；
* cookie存储数据的格式：字符串key=value
* cookie存储有效期：可以自行通过expires进行具体的日期设置，如果没设置，默认是关闭浏览器时失效。
* cookie有效范围：当前域名下有效。所以session这种会话存储方式方式只适用于客户端代码和服务端代码运行在同一台服务器上（前后端项目协议、域名、端口号都一致，即在一个项目下）

（2）session持久化

用于解决重启服务器后session消失的问题。在数据库中存储session，而不是存储在内存中。通过包：express-mysql-session。

当客户端存储的cookie失效后，服务端的session不会立即销毁，会有一个延时，服务端会定期清理无效session，不会造成无效数据占用存储空间的问题。

2、token机制



适用于前后端分离的项目（前后端代码运行在不同的服务器下）

请求登录时，token和sessionid原理相同，是对key和key对应的用户信息进行加密后的加密字符，登录成功后，会在响应主体中将{token：“字符串”}返回给客户端。

客户端通过cookie都可以进行存储。再次请求时不会默认携带，需要在请求拦截器位置给请求头中添加认证字段Authorization携带token信息，服务器就可以通过token信息查找用户登录状态。

### 96、说一下 session 的工作原理？

当客户端登录完成后，会在服务端产生一个session，此时服务端会将sessionid返回给客户端浏览器。客户端将sessionid储存在浏览器的cookie中，当用户再次登录时，会获得对应的sessionid，然后将sessionid发送到服务端请求登录，服务端在内存中找到对应的sessionid，完成登录，如果找不到，返回登录页面。

### 105、websocket应用的是哪个协议

WebSocket是一个允许Web应用程序(通常指浏览器)与服务器进行双向通信的协议。HTML5的WebSocket API主要是为浏览器端提供了一个基于TCP协议实现全双工通信的方法。

WebSocket优势： 浏览器和服务器只需要要做一个握手的动作，在建立连接之后，双方可以在任意时刻，相互推送信息。同时，服务器与客户端之间交换的头信息很小。

### 107、请列举出在 JDK 中几个常用的设计模式？

1、单例模式

作用：保证类只有一个实例。

JDK中体现：Runtime类。

2、静态工厂模式

作用：代替构造函数创建对象，方法名比构造函数清晰。

JDK中体现：Integer.valueOf、Class.forName

3、抽象工厂

作用：创建某一种类的对象。

JDK中体现：Java.sql包。

4、原型模式

clone()；

原型模式的本质是拷贝原型来创建新的对象，拷贝是比new更快的创建对象的方法，当需要大批量创建新对象而且都是同一个类的对象的时候考虑使用原型模式。

一般的克隆只是浅拷贝（对象的hash值不一样，但是对象里面的成员变量的hash值是一样的）。

有些场景需要深拷贝，这时我们就要重写clone方法，以ArrayList为例：

5、适配器模式

作用：使不兼容的接口相容。

JDK中体现：InputStream、OutputStream。

6、装饰器模式

作用：为类添加新的功能，防止类继承带来的类爆炸。

JDK中体现：io类、Collections、List。

7、外观模式

作用：封装一组交互类，一直对外提供接口。

JDK中体现：logging包。

8、享元模式

作用：共享对象、节省内存。

JDK中体现：Integer.valueOf、String常量池。

9、代理模式

作用：

（1）透明调用被代理对象，无须知道复杂实现细节；

（2）增加被代理类的功能；

JDK中体现：动态代理。

10、迭代器模式

作用：将集合的迭代和集合本身分离。

JDK中体现：Iterator

11、命令模式

作用：封装操作，使接口一致。

JDK中体现：Runable、Callable、ThreadPoolExecutor。

### 108、什么是设计模式？你是否在你的代码里面使用过任何设计模式？

1、什么是设计模式？

设计模式是解决软件开发某些特定问题而提出的一些解决方案，也可以理解为解决问题的一些固定思路。

通过设计模式可以帮助我们增强代码的可复用性、可扩展性、灵活性。

我们使用设计模式的最终目的是实现代码的高内聚、低耦合。

2、设计模式的七大原则

1. 单一职责原则
2. 接口隔离原则
3. 依赖倒转原则
4. 里式替换原则
5. 开闭原则
6. 迪米特法则
7. 合成复用原则

3、你是否在你的代码里面使用过任何设计模式？

（1）单例模式

JDK种的runtime，Spring种的singeton。

（2）简单工厂模式

Spring的BeanFactory，根据传入一个唯一标识来获得bean对象。

（3）原型模式

clone()

（4）代理模式

Spring的AOP中，Spring实现AOP功能的原理就是代理模式，①JDK动态代理。②CGLIB动态代理，使用Advice（通知）对类进行方法级别的切面增强。

（5）装饰器模式

为类添加新的功能，防止类爆炸；

IO流、数据源包装，Spring中用到的装饰器模式表现在Wrapper。

### 109、Java 中什么叫单例设计模式？请用 Java 写出线程安全的单例模式

1. 保证程序只有一个对象的实例，叫做单例模式；
2. 内部类的方式实现单例模式，是线程安全的；
3. 双重验证方式实现单例模式也是线程安全的；

### 110、在 Java 中，什么叫观察者设计模式（observer design pattern）？

1、观察者模式是一种一对多的依赖关系，让多个观察者同时监听某一主题对象。当这个主题对象发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

2、JAVA提供的对观察者模式的支持

在JAVA语言的java.util库里面，提供了一个Observable类以及一个Observer接口，构成JAVA语言对观察者模式的支持。

（1）Observer接口

这个接口只定义了一个方法，即update()方法，当被观察者对象的状态发生变化时，被观察者对象的notifyObservers()方法就会调用这一方法。

1. public interface Observer {
2. void update(Observable o, Object arg);
3. }

（2）Observable类

被观察者类都是java.util.Observable类的子类。java.util.Observable提供公开的方法支持观察者对象，这些方法中有两个对Observable的子类非常重要：一个是setChanged()，另一个是notifyObservers()。第一方法setChanged()被调用之后会设置一个内部标记变量，代表被观察者对象的状态发生了变化。第二个是notifyObservers()，这个方法被调用时，会调用所有登记过的观察者对象的update()方法，使这些观察者对象可以更新自己。

### 111、使用工厂模式最主要的好处是什么？在哪里使用？

1、工厂模式好处

* 良好的封装性、代码结构清晰；
* 扩展性好，如果想增加一个产品，只需扩展一个工厂类即可；
* 典型的解耦框架；

2、在哪里使用？

* 需要生成对象的地方；
* 不同数据库的访问；

### 112、请解释自动装配模式的区别？

有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式来进行依赖注入。

1、no

默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。第 402 页 共 485 页

2、byName

通过参数名 自动装配，Spring 容器在配置文件中发现 bean

的 autowire 属性被设置成 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。

3、byType:

通过参数类型自动装配，Spring 容器在配置文件中发现 bean

的 autowire 属性被设置成 byType，之后容器试图匹配、装配和该 bean 的属

性具有相同类型的 bean。如果有多个 bean 符合条件，则抛出错误。

4、constructor

这个方式类似于 byType， 但是要提供给构造器参数，如

果没有确定的带参数的构造器参数类型，将会抛出异常。

5、autodetect

首先尝试使用 constructor 来自动装配，如果无法工作，

则使用 byType 方式。

### 113、举一个用 Java 实现的装饰模式(decorator design pattern)？它是作用于对象层次还是类层次？

在Java IO中运用了装饰器模式，inputStream作为抽象类，其下有几个实现类，表示从不同的数据源输入：

1. byteArrayInputStream
2. fileInputStream
3. StringBufferInputStream
4. PipedInputStream，从管道产生输入；
5. SequenceInputStream，可将其他流收集合并到一个流内；

FilterInputStream作为装饰器在JDK中是一个普通类，其下面有多个具体装饰器比如BufferedInputStream、DataInputStream等。

FilterInputStream内部封装了基础构件：

protected volatile InputStream in;

而BufferedInputStream在调用其read()读取数据时会委托基础构件来进行更底层的操作，而它自己所起的装饰作用就是缓冲，在源码中可以很清楚的看到这一切。

### 118、BeanFactory 和 ApplicationContext 有什么区别？

1、BeanFactory是Spring的最底层接口，包含bean的定义，管理bean的加载，实例化，控制bean的生命周期，特点是每次获取对象时才会创建对象。

ApplicationContext是BeanFactory的子接口，拥有BeanFactory的全部功能，并且扩展了很多高级特性，每次容器启动时就会创建所有的对象。

1. ApplicationContext的额外功能：
2. 继承MessageSource，支持国际化；
3. 统一的资源文件访问方式；
4. 提供在监听器中注册bean；
5. 同时加载过个配置文件；
6. 载入多个（有继承关系）上下文，使得每个上下文都专注于一个特定的层次，比如应用的web层；

2、BeanFactory通常以编程的方式被创建，ApplicationContext可以以声明的方式创建，如使用ContextLoader。

3、BeanFactory 和 ApplicationContext都支持BeanPostProcessor，BeanFactoryPostProcessor，但BeanFactory需要手动注册，ApplicationContext则是自动注册。

### 132、Spring常用注解

[Spring常用注解（绝对经典）](https://blog.csdn.net/guorui_java/article/details/107347754)

https://blog.csdn.net/guorui\_java/article/details/107347754

### 133、项目中是如何实现权限验证的，权限验证需要几张表

通过了解，现在最普遍的权限管理模型就是RBAC（Role-Based Access Control）。

1、权限控制分类

菜单功能  
url控制（控制访问不同的控制器）  
2、RBAC的优缺点

（1）优点

简化了用户和权限的关系  
易扩展、易维护

（2）缺点

RBAC模型没有提供操作顺序的控制机制，这一缺陷使得RBAC模型很难适应哪些对操作次序有严格要求的系统。

3、RBAC支持的安全原则

（1）最小权限原则

RBAC可以将角色配置成其完成任务所需的最小权限集合。

（2）责任分离原则

可以通过调用相互独立互斥的角色来共同完成敏感的任务，例如要求一个记账员和财务管理员共同参与统一过账操作。

（3）数据抽象原则

可以通过权限的抽象来体现，例如财务操作用借款、存款等抽象权限，而不是使用典型的读写权限。

4、远古时代的权限控制

当时还没有RBAC，也没有这个概念，就是一堆程序员在那鼓捣，觉得登录这块该做点什么。

1、新建一个用户，对这个用户进行赋予权限。

2、但是一旦用户多了，权限复杂了，这工作量也是蛮大的。

5、RBAC

RBAC 1.0

### 135、如何防止表单重复提交

1、通过JavaScript屏蔽提交按钮（不推荐）

2、给数据库增加唯一键约束（简单粗暴）

3、利用Session防止表单重复提交（推荐）

4、使用AOP自定义切入实现

### 136、Spring中都应用了哪些设计模式

1、简单工厂模式

简单工厂模式的本质就是一个工厂类根据传入的参数，动态的决定实例化哪个类。

Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象。

2、工厂方法模式

应用程序将对象的创建及初始化职责交给工厂对象，工厂Bean。

定义工厂方法，然后通过config.xml配置文件，将其纳入Spring容器来管理，需要通过factory-method指定静态方法名称。

3、单例模式

Spring用的是双重判断加锁的单例模式，通过getSingleton方法从singletonObjects中获取bean。

1. */\*\**
2. *\* Return the (raw) singleton object registered under the given name.*
3. *\* <p>Checks already instantiated singletons and also allows for an early*
4. *\* reference to a currently created singleton (resolving a circular reference).*
5. *\* @param beanName the name of the bean to look for*
6. *\* @param allowEarlyReference whether early references should be created or not*
7. *\* @return the registered singleton object, or {@code null} if none found*
8. *\*/*
9. protected Object getSingleton(String beanName, boolean allowEarlyReference) {
10. Object singletonObject = this.singletonObjects.get(beanName);
11. if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
12. synchronized (this.singletonObjects) {
13. singletonObject = this.earlySingletonObjects.get(beanName);
14. if (singletonObject == null && allowEarlyReference) {
15. ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
16. if (singletonFactory != null) {
17. singletonObject = singletonFactory.getObject();
18. this.earlySingletonObjects.put(beanName, singletonObject);
19. this.singletonFactories.remove(beanName);
20. }
21. }
22. }
23. }
24. return (singletonObject != NULL\_OBJECT ? singletonObject : null);
25. }

4、代理模式

Spring的AOP中，使用的Advice（通知）来增强被代理类的功能。Spring实现AOP功能的原理就是代理模式（① JDK动态代理，② CGLIB字节码生成技术代理。）对类进行方法级别的切面增强。

5、装饰器模式

装饰器模式：动态的给一个对象添加一些额外的功能。

Spring的ApplicationContext中配置所有的DataSource。这些DataSource可能是不同的数据库，然后SessionFactory根据用户的每次请求，将DataSource设置成不同的数据源，以达到切换数据源的目的。

在Spring中有两种表现：

一种是类名中含有Wrapper，另一种是类名中含有Decorator。

6、观察者模式

定义对象间的一对多的关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。

Spring中观察者模式一般用在listener的实现。

7、策略模式

策略模式是行为性模式，调用不同的方法，适应行为的变化 ，强调父类的调用子类的特性 。

getHandler是HandlerMapping接口中的唯一方法，用于根据请求找到匹配的处理器。

8、模板方法模式

Spring JdbcTemplate的query方法总体结构是一个模板方法+回调函数，query方法中调用的execute()是一个模板方法，而预期的回调doInStatement(Statement state)方法也是一个模板方法。

### 138、mybatis 中 #{}和 ${}的区别是什么？

1. #{}带引号，${}不带引号；
2. #{}可以防止SQL注入；
3. ${}常用于数据库表名、order by子句；
4. 一般能用#{}就不要使用${}；

### 175、mysql 的内连接、左连接、右连接有什么区别？

1. 内连接,显示两个表中有联系的所有数据;
2. 左链接,以左表为参照,显示所有数据,右表中没有则以null显示
3. 右链接,以右表为参照显示数据，,左表中没有则以null显示

### 176、RabbitMQ的使用场景有哪些？

1、解决异步问题

例如用户注册，发送邮件和短信反馈注册成功，可以使用RabbitMQ消息队列，用户无需等待反馈。

2、服务间解耦

订单系统和库存系统，中间加入RabbitMQ消息队列，当库存系统出现问题时，订单系统依旧能正常使用，降低服务间耦合度。

3、秒杀系统

利用RabbitMQ的最大值，实现秒杀系统。

### 177、RabbitMQ有哪些重要的角色？有哪些重要的组件？

1、RabbitMQ有哪些重要的角色？

客户端、RabbitMQ、服务端。

2、有哪些重要的组件？

（1）connectionFactory（连接管理器）

应用程序与RabbitMQ之间建立连接的管理器。

（2）Channel（信道）

消息推送使用的信道。

（3）RoutingKey（路由键）

用于把生产者的数据分配到交换机上。

（4）Exchange（交换机）

用于接受和分配消息。

（5）BindKey（绑定键）

用于把交换机的消息绑定到队列上

（6）Queue（队列）

用于存储生产者消息。

### 178、RabbitMQ中 vhost 的作用是什么？

vhost可以理解为mini版的RabbitMQ，其内部均含有独立的交换机、绑定、队列，最重要的是拥有独立的权限系统，可以做到vhost范围内的用户控制。从RabbitMQ全局考虑，不同的应用可以跑在不同的vhost上，作为不同权限隔离的手段。

### 179、说一下 jvm 的主要组成部分？及其作用？

JVM包括类加载子系统、堆、方法区、栈、本地方法栈、程序计数器、直接内存、垃圾回收器、执行引擎。

1、类加载子系统

类加载子系统负责加载class信息，加载的类信息存放于方法区中。

2、直接内存

直接内存是在Java堆外的、直接向系统申请的内存空间。访问直接内存的速度会由于Java堆。出于性能的考虑，读写频繁的场合可能会考虑使用直接内存。

3、垃圾回收器

垃圾回收器可以对堆、方法区、直接内存进行回收。

4、执行引擎

执行引擎负责执行虚拟机的字节码，虚拟机会使用即时编译技术将方法编译成机器码后再执行。

### 180、说一下 jvm 运行时数据区？

运行时数据区包括堆、方法区、栈、本地方法栈、程序计数器。

1、堆

堆解决的是对象实例存储的问题，垃圾回收器管理的主要区域。

2、方法区

方法区可以认为是堆的一部分，用于存储已被虚拟机加载的信息，常量、静态变量、即时编译器编译后的代码。

3、栈

栈解决的是程序运行的问题，栈里面存的是栈帧，栈帧里面存的是局部变量表、操作数栈、动态链接、方法出口等信息。

（1）栈帧

每个方法从调用到执行的过程就是一个栈帧在虚拟机栈中入栈到出栈的过程。

（2）局部变量表

用于保存函数的参数和局部变量。

（3）操作数栈

操作数栈又称操作栈，大多数指令都是从这里弹出数据，执行运算，然后把结果压回操作数栈。

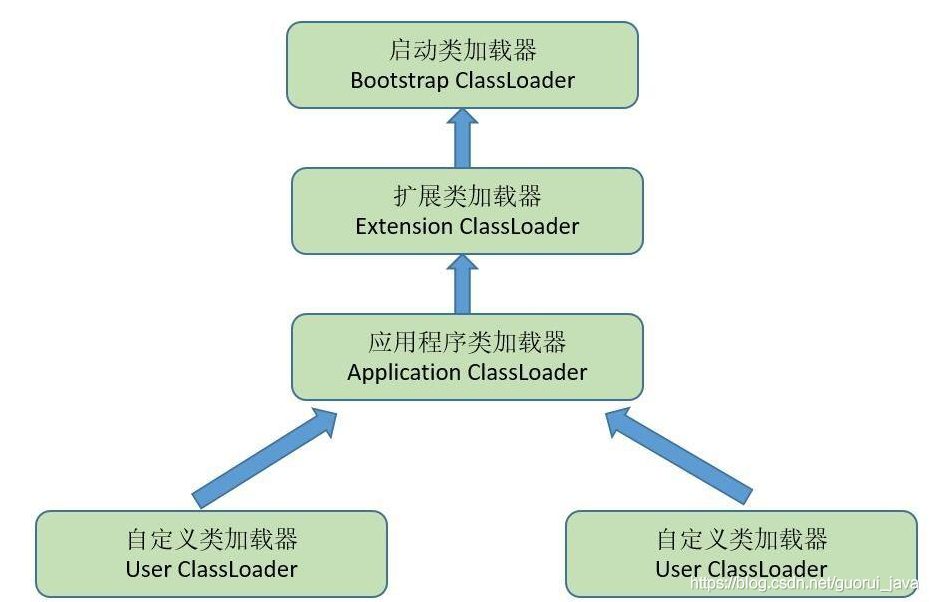
4、本地方法栈

与栈功能相同，本地方法栈执行的是本地方法，一个Java调用非Java代码的接口。

5、程序计数器（PC寄存器）

程序计数器中存放的是当前线程所执行的字节码的行数。JVM工作时就是通过改变这个计数器的值来选取下一个需要执行的字节码指令。

### 181、什么是类加载器，类加载器有哪些？



1、什么是类加载器？

类加载器负责加载所有的类，其为所有被载入内存的类生成一个java.lang.Class实例对象。

2、类加载器有哪些？

JVM有三种类加载器：

（1）启动类加载器

该类没有父加载器，用来加载Java的核心类，启动类加载器的实现依赖于底层操作系统，属于虚拟机实现的一部分，它并不继承自java.lang.classLoader。

（2）扩展类加载器

它的父类为启动类加载器，扩展类加载器是纯java类，是ClassLoader类的子类，负责加载JRE的扩展目录。

（3）应用程序类加载器

它的父类为扩展类加载器，它从环境变量classpath或者系统属性java.lang.path所指定的目录中加载类，它是自定义的类加载器的父加载器。

### 182、说一下类加载的执行过程？

当程序主动使用某个类时，如果该类还未被加载到内存中，JVM会通过加载、连接、初始化3个步骤对该类进行类加载。

1、加载

加载指的是将类的class文件读入到内存中，并为之创建一个java.lang.Class对象。

类的加载由类加载器完成，类加载器由JVM提供，开发者也可以通过继承ClassLoader基类来创建自己的类加载器。

通过使用不同的类加载器可以从不同来源加载类的二进制数据，通常有如下几种来源：

1. 从本地文件系统加载
2. 从jar包加载
3. 通过网络加载
4. 把一个Java源文件动态编译，并执行加载

2、连接

当类被加载之后，系统为之生成一个对应的Class对象，接着进入连接阶段，连接阶段负责将类的二进制数据合并到JRE中。

类连接又可分为三个阶段：

（1）验证

文件格式验证  
元数据验证  
字节码验证  
符号引用验证

（2）准备

为类的静态变量分配内存，并设置默认初始值。

（3）解析

将类的二进制数据中的符号引用替换成直接引用。

3、初始化

为类的静态变量赋予初始值。

### 183、JVM的类加载机制是什么？

JVM类加载机制主要有三种：

1、全盘负责

类加载器加载某个class时，该class所依赖的和引用其它的class也由该类加载器载入。

2、双亲委派

先让父加载器加载该class，父加载器无法加载时才考虑自己加载。

3、缓存机制

缓存机制保证所有加载过的class都会被缓存，当程序中需要某个class时，先从缓存区中搜索，如果不存在，才会读取该类对应的二进制数据，并将其转换成class对象，存入缓存区中。

这就是为什么修改了class后，必须重启JVM，程序所做的修改才会生效的原因。

### 184、什么是双亲委派模型？

如果一个类收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器执行，如果父加载器还存在其父加载器，则进一步向上委托，依次递归，请求将最终到达顶层的启动类加载器，如果父类加载器可以完成父加载任务，就成功返回，如果父加载器无法完成加载任务，子加载器才会尝试自己去加载，这就是双亲委派模型。

双亲委派模式的优势：

1. 避免重复加载；
2. 考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委派模式传递到启动加载器，而启动加载器在核心Java API中发现同名的类，发现该类已经被加载，就不会重新加载网络传递的Integer类，而直接返回已加载过的Integer.class，这样可以防止核心API库被随意篡改。

### 185、怎么判断对象是否可以被回收？

1、引用计数算法

（1）判断对象的引用数量

通过判断对象的引用数量来决定对象是否可以被回收；  
每个对象实例都有一个引用计数器，被引用+1，完成引用-1；  
任何引用计数为0的对象实例可以被当做垃圾回收；

（2）优缺点

优点：执行效率高，程序受影响较小；  
缺点：无法检测出循环引用的情况，导致内存泄漏；

2、可达性分析算法

通过判断对象的引用链是否可达来决定对象是否可以被回收。

如果程序无法再引用该对象，那么这个对象肯定可以被回收，这个状态称为不可达。

那么不可达状态如何判断呢？

答案是GC roots，也就是根对象，如果一个对象无法到达根对象的路径，或者说从根对象无法引用到该对象，该对象就是不可达的。

以下三种对象在JVM中被称为GC roots，来判断一个对象是否可以被回收。

（1）虚拟机栈的栈帧

每个方法在执行的时候，JVM都会创建一个相应的栈帧（操作数栈、局部变量表、运行时常量池的引用），当方法执行完，该栈帧就从栈中弹出，这样一来，方法中临时创建的独享就不存在了，或者说没有任何GC roots指向这些临时对象，这些对象在下一次GC的时候便会被回收。

（2）方法区中的静态属性

静态属性数据类属性，不属于任何实例，因此该属性自然会作为GC roots。这要这个class在，该引用指向的对象就一直存在，class也由被回收的时候。

class何时会被回收？

1. 堆中不存在该类的任何实例
2. 加载该类的classLoader已经被回收
3. 该类的java.lang.class对象没有在任何地方被引用，也就是说无法通过反射访问该类的信息

（3）本地方法栈引用的对象

### 186、说一下 jvm 有哪些垃圾回收算法？

1、对象是否已死算法

* 引用计数器算法
* 可达性分析算法

2、GC算法

（1）标记清除算法

如果对象被标记后进行清除，会带来一个新的问题--内存碎片化。如果下次有比较大的对象实例需要在堆上分配较大的内存空间时，可能会出现无法找到足够的连续内存而不得不再次触发垃圾回收。

（2）复制算法（Java堆中新生代的垃圾回收算法）

1. 先标记待回收内存和不用回收内存；
2. 将不用回收的内存复制到新的内存区域；
3. 就的内存区域就可以被全部回收了，而新的内存区域也是连续的；

缺点是损失部分系统内存，因为腾出部分内存进行复制。

（3）标记压缩算法（Java堆中老年代的垃圾回收算法）

对于新生代，大部分对象都不会存活，所以复制算法较高效，但对于老年代，大部分对象可能要继续存活，如果此时使用复制算法，效率会降低。

标记压缩算法首先还是标记，将不用回收的内存对象压缩到内存一端，此时即可清除边界处的内存，这样就能避免复制算法带来的效率问题，同时也能避免内存碎片化的问题。

老年代的垃圾回收算法称为“Major GC”。

## 基本数据类型 ：

byte short int long float double char Boolean

byte(字节型，占1字节)、short(短整型，占2字节)、int(整型，占4字节)、long(长整型，占8字节）

float(单精度[浮点数](https://so.csdn.net/so/search?q=%E6%B5%AE%E7%82%B9%E6%95%B0&spm=1001.2101.3001.7020)，占4字节)、double(双精度浮点数，占8字节)

**布尔型**：boolean(布尔型)，只有true 和false两个取值

**字符型**：char(字符型，占2个字节)  注：因字符类型表示Unicode字符，所以一个字符占2字节

## 成员变量和局部变量和静态变量

静态成员

变量值是所有类的实例共享的，所以调用的时候只需要  类名.属性名  就可调用（也只能这样调用）

在类加载的时候，类加载完成，就分配完空间；直到类被卸载时空间被回收

成员变量

定义在类中，整个类有效，存在堆内存，随该对象的消失而消失

局部变量

定义在方法，参数，语句上，只在该大括号内生效，存在栈中，大括号结束释放

全局变量

1. 全局变量:
2. 写在函数和大括号外部的变量, 我们称之为全局变量
3. 作用域: 从定义的那一行开始, 一直到文件末尾
4. 全局变量可以先定义在初始化, 也可以定义的同时初始化
5. 存储: 静态区
6. 程序一启动就会分配存储空间, 直到程序结束才会释放

## 异常:

### Throwable

的子类, 在java中只有[Throwable类](https://so.csdn.net/so/search?q=Throwable%E7%B1%BB&spm=1001.2101.3001.7020)型的实例才可以被抛出或者捕获,它是异常处理机制的基本类型.Error 错误,通常由jvm产生

### Error: NoClassDeFoundError

编写sh脚本时,有遇见

无法通过try(){}catch(exception)捕获

需要通过 try(){}catch(Throwable)捕获

### Exception:异常 由代码抛出

IOexception

Sqlexception

1. **非检查异常:**

* ArrayIndexOutOfBoundsException //用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引
* ArithmeticException //当出现异常的运算条件时，抛出此异常。（ 例如，一个整数“除以零”时，抛出此类的一个实例）
* IllegaArguementException //抛出的异常表明向方法传递了一个不合法或不正确的参数
* NullPointerException //空指针异常（调用 null 对象的实例方法等）
* ClassCastException //类转换异常
* ArrayStoreException //数据存储异常，操作数组时类型不一致

1. **检查异常**

除了RuntimeException与其子类，以及错误（Error）,都是检查异常

* ClassNotFoundException // 找不到具有指定名称的类的定义
* DataFormatException //数据格式异常
* IOException //输入输出异常
* SQLException //提供有关数据库访问错误或其他错误的信息的异常
* FileNotFoundException //当试图打开指定路径名表示的文件失败时，抛出此异常
* EOFException //当输入过程中意外到达文件或流的末尾时，抛出此异常
* IndexOutOfBoundsException：数组下角标越界异常;

### 运行时异常和非运行时异常。

刚才我们是从Exception和Error整体来划分的，现在我们可以对Exception异常进行划分，它可分为运行时异常和非运行时异常。

一、运行时异常

都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是非检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。

二、非运行时异常

是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不要自定义检查异常。

## 反射

动态加载一个指定的类，并获得该类中的所有内容，反射技术可以对一个类进行解剖

好处：大大增强了程序的扩展性

基本步骤：

1.获得Class对象，就是获取到指定的名称的字节码文件对象

2.实例化对象，获得类的属性，方法和构造函数

3.访问属性，调用方法，调用构造函数创建对象

## IO

## IO模型 BIO ，NIO和AIO

* 同步，一个任务的完成之前不能做其他操作，必须等待（等于在打电话）
* 异步，一个任务的完成之前，可以进行其他操作（等于在聊QQ）
* 阻塞，是相对于CPU来说的， 挂起当前线程，不能做其他操作只能等待
* 非阻塞,，无须挂起当前线程，可以去执行其他操作

### BIO全称是Blocking IO，同步阻塞式IO，是JDK1.4之前的传统IO模型。

* BIO：同步并阻塞，服务器实现一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，没处理完之前此线程不能做其他操作（如果是单线程的情况下，我传输的文件很大呢？），当然可以通过线程池机制改善。BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

### 什么是NIO 同步非阻塞

* NIO:同步非阻塞，服务器实现一个连接一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4之后开始支持。

 同步非阻塞，服务器实现模式为一个线程处理多个请求(连接)，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求就进行处理

* Channel(通道)

本地文件IO通道，用于读取、写入、映射和操作文件的通道

* Buffer(缓冲区)

缓冲区 Buffer 是 Java NIO 中一个核心概念，在NIO库中，所有数据都是用缓冲区处理的。

在读取数据时，它是直接读到缓冲区中的,在写入数据时，它也是写入到缓冲区中的,任何时候访问 NIO 中的数据，都是将它放到缓冲区中。

而在面向流I/O系统中，所有数据都是直接写入或者直接将数据读取到Stream对象中。

* Selector(选择器)

Selector类是NIO的核心类，Selector（选择器）选择器提供了选择已经就绪的任务的能力。

Selector会不断的轮询注册在上面的所有channel，如果某个channel为读写等事件做好准备，那么就处于就绪状态，通过Selector可以不断轮询发现出就绪的channel，进行后续的IO操作。

一个Selector能够同时轮询多个channel，这样，一个单独的线程就可以管理多个channel，从而管理多个网络连接，这样就不用为每一个连接都创建一个线程，同时也避免了多线程之间上下文切换导致的开销。

### 什么是AIO

AIO：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由操作系统先完成了再通知服务器应用去启动线程进行处理，AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用操作系统参与并发操作，编程比较复杂，JDK1.7之后开始支持。.

AIO属于NIO包中的类实现，其实IO主要分为BIO和NIO，AIO只是附加品，解决IO不能异步的实现

在以前很少有Linux系统支持AIO，Windows的IOCP就是该AIO模型。但是现在的服务器一般都是支持AIO操作

AIO是java中IO模型的一种，作为NIO的改进和增强随JDK1.7版本更新被集成在JDK的nio包中，因此AIO也被称作是NIO2.0。区别于传统的BIO(Blocking IO,同步阻塞式模型,JDK1.4之前就存在于JDK中，NIO于JDK1.4版本发布更新)的阻塞式读写，AIO提供了从建立连接到读、写的全异步操作。AIO可用于异步的文件读写和网络通信。

### 字节流和字符流

字节流继承inputStream和OutputStream

字符流继承⾃InputSteamReader和OutputStreamWriter

## 序列化

### 什么是 Java 序列化

序列化是把对象改成可以存到磁盘或通过网络发送到其他运行中的 Java 虚拟机的二进制格式的过程, 并可以通过反序列化恢复对象状态。

Java 序列化API给开发人员提供了一个标准机制, 通过 java.io.Serializable 和 java.io.Externalizable 接口, ObjectInputStream 及ObjectOutputStream 处理对象序列化。

Java 程序员可自由选择基于类结构的标准序列化或是他们自定义的二进制格式, 通常认为后者才是最佳实践, 因为序列化的二进制文件格式成为类输出 API的一部分, 可能破坏 Java 中私有和包可见的属性的封装.

## 如何序列化

让 Java 中的类可以序列化很简单. 你的 Java 类只需要实现 java.io.Serializable 接口, JVM 就会把 Object 对象按默认格式序列化。让一个类是可序列化的需要有意为之。

推荐看下：关于Java序列化你应该知道的一切

类可序列会可能为是一个长期代价， 可能会因此而限制你修改或改变其实现. 当你通过实现添加接口来更改类的结构时， 添加或删除任何字段可能会破坏默认序列化， 这可以通过自定义二进制格式使不兼容的可能性最小化, 但仍需要大量的努力来确保向后兼容性。

序列化如何限制你更改类的能力的一个示例是 SerialVersionUID。如果不显式声明 SerialVersionUID, 则 JVM 会根据类结构生成其结构， 该结构依赖于类实现接口和可能更改的其他几个因素。

假设你新版本的类文件实现的另一个接口, JVM 将生成一个不同的 SerialVersionUID 的, 当你尝试加载旧版本的程序序列化的旧对象时, 你将获得无效类异常 InvalidClassException。

### 问题 1) Java 中的可序列化接口和可外部接口之间的区别是什么？

这是 Java 序列化访谈中最常问的问题。下面是我的版本 Externalizable 给我们提供 writeExternal() 和 readExternal() 方法, 这让我们灵活地控制 Java 序列化机制, 而不是依赖于 Java 的默认序列化。正确实现 Externalizable 接口可以显著提高应用程序的性能。

### 问题 2) 可序列化的方法有多少？如果没有方法,那么可序列化接口的用途是什么？

可序列化 Serializalbe 接口存在于java.io包中,构成了 Java 序列化机制的核心。它没有任何方法, 在 Java 中也称为标记接口。当类实现 java.io.Serializable 接口时, 它将在 Java 中变得可序列化, 并指示编译器使用 Java 序列化机制序列化此对象。

### 问题 3) 什么是 serialVersionUID ？如果你不定义这个, 会发生什么？

我最喜欢的关于Java序列化的问题面试问题之一。serialVersionUID 是一个 private static final long 型 ID, 当它被印在对象上时, 它通常是对象的哈希码,你可以使用 serialver 这个 JDK 工具来查看序列化对象的 serialVersionUID。

SerialVerionUID 用于对象的版本控制。也可以在类文件中指定 serialVersionUID。不指定 serialVersionUID的后果是,当你添加或修改类中的任何字段时, 则已序列化类将无法恢复, 因为为新类和旧序列化对象生成的 serialVersionUID 将有所不同。

Java 序列化过程依赖于正确的序列化对象恢复状态的, ,并在序列化对象序列版本不匹配的情况下引发 java.io.InvalidClassException 无效类异常。

### 问题 4) 序列化时,你希望某些成员不要序列化？你如何实现它？

另一个经常被问到的序列化面试问题。这也是一些时候也问， 如什么是瞬态 transient 变量， 瞬态和静态变量会不会得到序列化等，所以,如果你不希望任何字段是对象的状态的一部分, 然后声明它静态或瞬态根据你的需要, 这样就不会是在 Java 序列化过程中被包含在内。

### 问题 5) 如果类中的一个成员未实现可序列化接口，会发生什么情况？

关于Java 序列化过程的一个简单问题。如果尝试序列化实现了可序列化接口的类的对象，但该对象包含对不可序列化类的引用，则在运行时将引发不可序列化异常 NotSerializableException， 这就是为什么我始终将一个可序列化警报(在我的代码注释部分中)，作为代码注释最佳实践之一， 提示开发人员记住这一事实， 在可序列化类中添加新字段时要注意。

### 问题 6) 如果类是可序列化的, 但其超类不是, 则反序列化后从超级类继承的实例变量的状态如何？

Java 序列化过程仅在对象层级都是可序列化的类中继续， 即：实现了可序列化接口， 如果从超级类没有实现可序列化接口，则超级类继承的实例变量的值将通过调用构造函数初始化。

且一旦构造函数链启动, 就不可能停止， 因此， 即使层次结构中更高的类成员变量实现了可序列化接口， 也将通过执行构造函数创建，而不再是反序列化得到。如你所见， 这个序列化面试问题看起来非常不易回答， 但如果你熟悉关键概念， 则并不难。

### 问题 7) 是否可以自定义序列化过程, 或者是否可以覆盖 Java 中的默认序列化过程？

答案是肯定的, 你可以。我们都知道,对于序列化一个对象需调用 ObjectOutputStream.writeObject(saveThisObject), 并用 ObjectInputStream.readObject() 读取对象, 但 Java 虚拟机为你提供的还有一件事, 是定义这两个方法。

如果在类中定义这两种方法, 则 JVM 将调用这两种方法, 而不是应用默认序列化机制。你可以在此处通过执行任何类型的预处理或后处理任务来自定义对象序列化和反序列化的行为。

需要注意的重要一点是要声明这些方法为私有方法, 以避免被继承、重写或重载。由于只有 Java 虚拟机可以调用类的私有方法, 你的类的完整性会得到保留, 并且 Java 序列化将正常工作。

在我看来, 这是在任何 Java 序列化面试中可以问的最好问题之一, 一个很好的后续问题是, 为什么要为你的对象提供自定义序列化表单？

### 问题 8) 假设新类的超级类实现可序列化接口, 如何避免新类被序列化？

这是在 Java 序列化中不好回答的问题。如果类的 Super 类已经在 Java 中实现了可序列化接口， 那么它在 Java 中已经可以序列化， 因为你不能取消接口，它不可能真正使它无法序列化类, 但是有一种方法可以避免新类序列化。

为了避免 Java 序列化,你需要在类中实现 writeObject() 和 readObject() 方法, 并且需要从该方法引发不序列化异常NotSerializableException。这是自定义 Java 序列化过程的另一个好处, 如上述序列化面试问题中所述, 并且通常随着面试进度, 它作为后续问题提出。

### 问题 9) 在 Java 中的序列化和反序列化过程中使用哪些方法？

这是很常见的面试问题, 在序列化基本上面试官试图知道: 你是否熟悉 readObject() 的用法、writeObject()、readExternal() 和 writeExternal()。

Java 序列化由java.io.ObjectOutputStream类完成。该类是一个筛选器流, 它封装在较低级别的字节流中, 以处理序列化机制。要通过序列化机制存储任何对象, 我们调用 ObjectOutputStream.writeObject(savethisobject), 并反序列化该对象, 我们称之为 ObjectInputStream.readObject()方法。

调用以 writeObject() 方法在 java 中触发序列化过程。关于 readObject() 方法, 需要注意的一点很重要一点是, 它用于从持久性读取字节, 并从这些字节创建对象, 并返回一个对象, 该对象需要类型强制转换为正确的类型。

### 问题 10) 假设你有一个类,它序列化并存储在持久性中， 然后修改了该类以添加新字段。如果对已序列化的对象进行反序列化, 会发生什么情况？

这取决于类是否具有其自己的 serialVersionUID。正如我们从上面的问题知道, 如果我们不提供 serialVersionUID, 则 Java 编译器将生成它, 通常它等于对象的哈希代码。

通过添加任何新字段, 有可能为该类新版本生成的新 serialVersionUID 与已序列化的对象不同, 在这种情况下, Java 序列化 API 将引发 java.io.InvalidClassException, 因此建议在代码中拥有自己的 serialVersionUID, 并确保在单个类中始终保持不变。

### 11) Java序列化机制中的兼容更改和不兼容更改是什么？

真正的挑战在于通过添加任何字段、方法或删除任何字段或方法来更改类结构, 方法是使用已序列化的对象。

根据 Java 序列化规范, 添加任何字段或方法都面临兼容的更改和更改类层次结构或取消实现的可序列化接口, 有些接口在非兼容更改下。

对于兼容和非兼容更改的完整列表, 我建议阅读 Java 序列化规范。

### 12) 我们可以通过网络传输一个序列化的对象吗？

是的 ,你可以通过网络传输序列化对象, 因为 Java 序列化对象仍以字节的形式保留, 字节可以通过网络发送。你还可以将序列化对象存储在磁盘或数据库中作为 Blob。

### 13) 在 Java 序列化期间,哪些变量未序列化？

这个问题问得不同, 但目的还是一样的, Java开发人员是否知道静态和瞬态变量的细节。由于静态变量属于类, 而不是对象, 因此它们不是对象状态的一部分, 因此在 Java 序列化过程中不会保存它们。

# 数据库

## 一、索引相关

### （1）什么是索引?

索引是一种数据结构，可以帮助我们快速的进行数据的查找。

### （2）索引是个什么样的数据结构呢?

索引的数据结构和具体存储引擎的实现有关，在 MySQL 中使用较多的索引有 **Hash 索引，B+ 树索引**等，而我们经常使用的 InnoDB 存储引擎的默认索引实现为：B+ 树索引。

### （3）为什么使用索引？

* 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
* 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
* 帮助服务器避免排序和临时表。
* 将随机IO变为顺序IO。
* 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

### （4）Innodb为什么要用自增id作为主键？

如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置。如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，消耗更大。

### （5）[Hash](https://so.csdn.net/so/search?q=Hash&spm=1001.2101.3001.7020) 索引和 B+ 树索引有什么区别或者说优劣呢?

首先要知道 Hash 索引和 B+ 树索引的底层实现原理：

hash 索引底层就是 hash 表，进行查找时，调用一次 hash 函数就可以获取到相应的键值，之后进行回表查询获得实际数据。B+ 树底层实现是多路平衡查找树。对于每一次的查询都是从根节点出发，查找到叶子节点方可以获得所查键值，然后根据查询判断是否需要回表查询数据。

那么可以看出他们有以下的不同：

* hash 索引进行等值查询更快(一般情况下)，但是却无法进行范围查询。  
  因为在 hash 索引中经过 hash 函数建立索引之后，索引的顺序与原顺序无法保持一致，不能支持范围查询。而 B+ 树的的所有节点皆遵循(左节点小于父节点，右节点大于父节点，多叉树也类似)，天然支持范围。
* hash 索引不支持模糊查询以及多列索引的最左前缀匹配。原理也是因为 hash 函数的不可预测。
* hash 索引虽然在等值查询上较快，但是不稳定。性能不可预测，当某个键值存在大量重复的时候，发生 hash 碰撞，此时效率可能极差。而 B+ 树的查询效率比较稳定，对于所有的查询都是从根节点到叶子节点，且树的高度较低。

因此，在大多数情况下，直接选择 B+ 树索引可以获得稳定且较好的查询速度。而不需要使用 hash 索引。

### （6）什么是聚簇索引

聚簇索引就是按照每张表的 **主键** 构造一棵B+树，同时叶子节点中存放的就是整张表的行记录数据。

在 InnoDB 中，只有主键索引是聚簇索引，如果没有主键，则挑选一个唯一键建立聚簇索引。如果没有唯一键，则MySQL自动为InnoDB表生成一个隐含字段来建立聚簇索引，这个字段长度为6个字节，类型为长整形。

### （7）说一说索引的底层实现？

**Hash索引**

基于哈希表实现，只有精确匹配索引所有列的查询才有效，对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），并且Hash索引将所有的哈希码存储在索引中，同时在索引表中保存指向每个数据行的指针。

**B-Tree索引（MySQL使用B+Tree）**

B-Tree能加快数据的访问速度，因为存储引擎不再需要进行全表扫描来获取数据，数据分布在各个节点之中。

**B+Tree索引**

是B-Tree的改进版本，同时也是数据库索引所采用的存储结构。数据都在叶子节点上，并且增加了顺序访问指针，每个叶子节点都指向相邻的叶子节点的地址。相比B-Tree来说，进行范围查找时只需要查找两个节点，进行遍历即可。而B-Tree需要获取所有节点，相比之下B+Tree效率更高。

B+tree性质：

* n棵子tree的节点包含n个关键字，不用来保存数据而是保存数据的索引。
* 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
* 所有的非终端结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。
* B+ 树中，数据对象的插入和删除仅在叶节点上进行。
* B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

### （8）索引有哪些优缺点？

**索引的优点**

* 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
* 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

**索引的缺点**

* 时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；
* 空间方面：索引需要占物理空间。

### （9）聚簇索引和非聚簇索引的区别

* **聚簇索引的叶子节点存放的是主键值和数据行，支持覆盖索引**
* **非聚簇索引的叶子节点存放的是主键值或数据记录的地址（InnoDB辅助索引的data域存储相应记录主键的值，MyISAM辅助索引的data域保存数据记录的地址）**

### （11）MySQL中有几种索引类型，可以简单说说吗？

* **FULLTEXT** ：即为全文索引，目前只有MyISAM引擎支持。其可以在CREATE TABLE ，ALTER TABLE ，CREATE INDEX 使用，不过目前只有 CHAR、VARCHAR ，TEXT 列上可以创建全文索引。
* **HASH** ：由于HASH的唯一（几乎100%的唯一）及类似键值对的形式，很适合作为索引。HASH索引可以一次定位，不需要像树形索引那样逐层查找,因此具有极高的效率。但是，这种高效是有条件的，即只在“=”和“in”条件下高效，对于范围查询、排序及组合索引仍然效率不高。
* **BTREE** ：BTREE索引就是一种将索引值按一定的算法，存入一个树形的数据结构中（二叉树），每次查询都是从树的入口root开始，依次遍历node，获取leaf。这是MySQL里默认和最常用的索引类型。
* **RTREE** ：RTREE在MySQL很少使用，仅支持geometry数据类型，支持该类型的存储引擎只有MyISAM、BDb、InnoDb、NDb、Archive几种。相对于BTREE，RTREE的优势在于范围查找。

### （14）联合索引是什么?为什么需要注意联合索引中的顺序?

**MySQL 可以使用多个字段同时建立一个索引，叫做联合索引**。在联合索引中，如果想要命中索引，需要按照建立索引时的字段顺序挨个使用，否则无法命中索引。

具体原因为：

MySQL 使用索引时需要索引有序，假设现在建立了"name,age,school"的联合索引，那么索引的排序为：先按照name排序，如果 name 相同，则按照 age 排序，如果 age 的值也相等，则按照 school 进行排序。

当进行查询时，此时索引仅仅按照 name 严格有序，因此必须首先使用 name 字段进行等值查询，之后对于匹配到的列而言，其按照 age 字段严格有序，此时可以使用 age 字段用做索引查找，以此类推。因此在建立联合索引的时候应该注意索引列的顺序，一般情况下，将查询需求频繁或者字段选择性高的列放在前面。此外可以根据特例的查询或者表结构进行单独的调整。

### （15）创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

MySQL 提供了 explain 命令来查看语句的执行计划，MySQL 在执行某个语句之前，会将该语句过一遍查询优化器，之后会拿到对语句的分析，也就是执行计划，其中包含索引有关的信息来分析是否命中了索引

### （16）那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?

* 使用不等于查询
* 列参与了数学运算或者函数
* 在字符串 like 时左边是通配符。类似于’%aaa’。
* 当 mysql 分析全表扫描比使用索引快的时候不使用索引。
* 当使用联合索引，前面一个条件为范围查询，后面的即使符合最左前缀原则，也无法使用索引。

以上情况，MySQL无法使用索引。

### （17）为什么Mysql用B+树做索引而不用B-树或红黑树、二叉树

**主要原因：B+树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而B树只能中序遍历所有节点，效率太低。**

### （18）MySQL索引种类

普通索引、唯一索引(主键索引、唯一索引)、联合索引、全文索引、空间索引

### （19）索引在什么情况下遵循最左前缀的规则？

在建立了联合索引的前提条件下，数据库会一直从左向右的顺序依次查找，直到遇到了范围查询(>,<,between,like等)

**二、事务相关**

### （1）什么是事务?

事务是一系列的数据库操作，他们要符合 ACID 特性，事务是数据库应用的基本单位。MySQL 事务主要用于处理操作量大，复杂度高的数据。

### （2）ACID是什么?可以详细说一下吗?

* A=Atomicity：原子性，就是要么全部成功，要么全部失败。不可能只执行一部分操作。
* C=Consistency：一致性，系统(数据库)总是从一个一致性的状态转移到另一个一致性的状态，不会存在中间状态。
* I=Isolation：隔离性，通常来说：一个事务在完全提交之前，对其他事务是不可见的.注意前面的通常来说加了红色，意味着有例外情况。
* D=Durability：持久性，一旦事务提交，那么就永远是这样子了，哪怕系统崩溃也不会影响到这个事务的结果。

### （3）MySQL中为什么要有事务回滚机制？

而在 MySQL 中，**恢复机制是通过回滚日志（undo log）实现的**，所有事务进行的修改都会先记录到这个回滚日志中，然后在对数据库中的对应行进行写入。**当事务已经被提交之后，就无法再次回滚了。**

回滚日志作用：

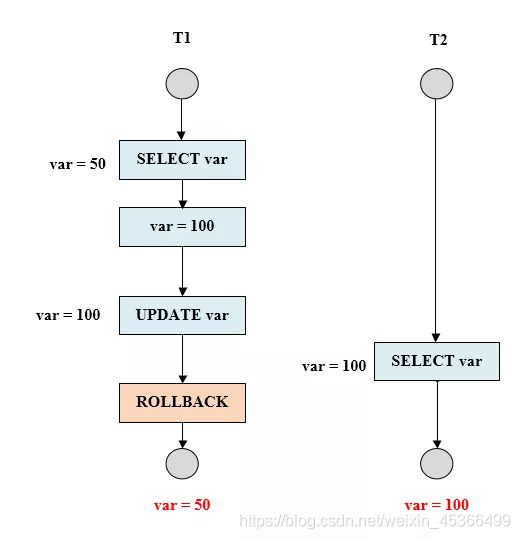
1. 能够在发生错误或者用户执行 ROLLBACK 时提供回滚相关的信息
2. 在整个系统发生崩溃、数据库进程直接被杀死后，当用户再次启动数据库进程时，还能够立刻通过查询回滚日志将之前未完成的事务进行回滚，这也就需要回滚日志必须先于数据持久化到磁盘上，是我们需要先写日志后写数据库的主要原因。

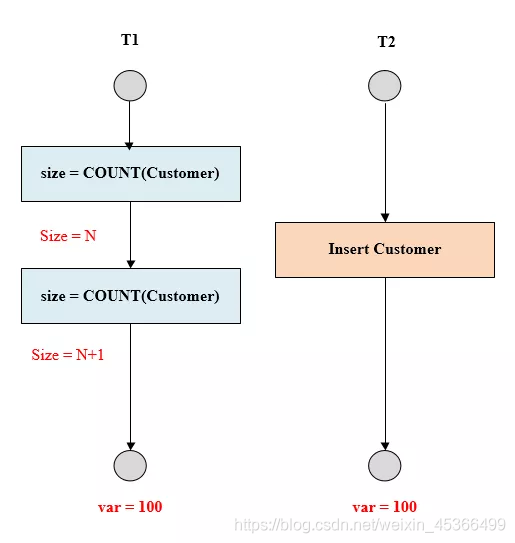
### （4）数据库并发事务会带来哪些问题？

数据库并发事务会带来 **脏读、幻读、丢弃更改、不可重复读** 这四个常见问题，其中：

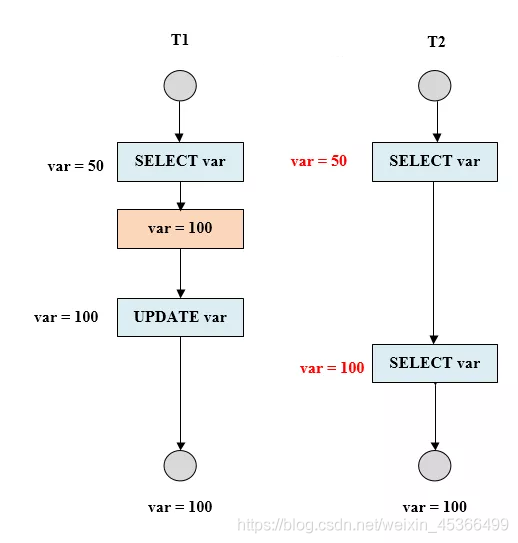
* **脏读**：A 事务读取到了 B 事务未提交的内容，但是之后B事务满足一致性等特性而做了回滚操作，那么读取事务得到的结果就是脏数据了。
* **幻读**：A 事务读取了一个范围的内容，而同时 B 事务在此期间插入（删除）了一条数据。造成"幻觉"。
* **丢弃修改**：两个写事务T1 T2同时对A=0进行递增操作，结果T2覆盖T1，导致最终结果是1 而不是2，事务被覆盖
* **不可重复读**：当设置T2事务只能读取 T1 事务已经提交的部分，T2 读取一个数据，然后T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。

当设置A事务只能读取 B 事务已经提交的部分，会造成在 A 事务内的两次查询，结果竟然不一样，因为在此期间 B 事务进行了提交操作。

**脏读图解：**  
  
第一个事务首先读取var变量为50，接着准备更新为100的时，并未提交，第二个事务已经读取var为100，此时第一个事务做了回滚。最终第二个事务读取的var和数据库的var不一样。

**幻读图解：**  
  
T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。

**不可重复读图解：**

  
T2 读取一个数据，T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。

### （5）怎么解决这些问题呢?MySQL 的事务隔离级别了解吗?

MySQL 的四种隔离级别如下：

* 未提交读(READ UNCOMMITTED)：事务中发生了修改，即使没有提交，其他事务也是可见的，比如对于一个数A原来50修改为100，但是我还没有提交修改，另一个事务看到这个修改，而这个时候原事务发生了回滚，这时候A还是50，但是另一个事务看到的A是100.**可能会导致脏读、幻读或不可重复读**
* 已提交读(READ COMMITTED)：对于一个事务从开始直到提交之前，所做的任何修改是其他事务不可见的，举例就是对于一个数A原来是50，然后提交修改成100，这个时候另一个事务在A提交修改之前，读取的A是50，刚读取完，A就被修改成100，这个时候另一个事务再进行读取发现A就突然变成100了；**可以阻止脏读，但是幻读或不可重复读仍有可能发生**
* 可重复读(REPEATABLE READ)：就是对一个记录读取多次的记录是相同的，比如对于一个数A读取的话一直是A，前后两次读取的A是一致的；**可以阻止脏读和不可重复读，但幻读仍有可能发生**
* 可串行化(SERIALIZABLE)：在并发情况下，和串行化的读取的结果是一致的，没有什么不同，比如不会发生脏读和幻读；**该级别可以防止脏读、不可重复读以及幻读**

### （6）Innodb使用的是哪种隔离级别呢?

**MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重复读）**

**InnoDB 存储引擎在分布式事务 的情况下一般会用到SERIALIZABLE(可串行化)隔离级别。**

### （7）不可重复读和幻读区别是什么？可以举个例子吗？

**不可重复读的重点是修改，幻读的重点在于新增或者删除。**

* 例1（同样的条件, 你读取过的数据, 再次读取出来发现值不一样了 ）：事务1中的A先生读取自己的工资为 1000的操作还没完成，事务2中的B先生就修改了A的工资为2000，导致A再读自己的工资时工资变为 2000；这就是不可重复读。
* 例2（同样的条件, 第1次和第2次读出来的记录数不一样 ）：假某工资单表中工资大于3000的有4人，事务1读取了所有工资大于3000的人，共查到4条记录，这时事务2 又插入了一条工资大于3000的记录，事务1再次读取时查到的记 录就变为了5条，这样就导致了幻读。

**三、锁相关**

### （1）对 MySQL 的锁了解吗?

**当数据库有并发事务的时候,可能会产生数据的不一致,这时候需要一些机制来保证访问的次序,锁机制就是这样的一个机制.**

### （2）MySQL 锁的分类

Mysql中锁的分类按照不同类型的划分可以分成不同的锁：

按照 **锁的粒度** 划分可以分成：

* 行锁
* 表锁
* 页锁

按照 **使用的方式** 划分可以分为：

* 共享锁
* 排它锁

按照 **思想** 的划分：

* 乐观锁
* 悲观锁

### （3）行级锁、表级锁、页级锁的描述与特点

**行级锁：**

* **描述**：行级锁是mysql中锁定粒度最细的一种锁。**表示只针对当前操作的行进行加锁**。行级锁能大大减少数据库操作的冲突，其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁和排他锁
* **特点**：开销大，加锁慢，会出现死锁。发生锁冲突的概率最低，并发度也最高。

**表级锁：**

* **描述**：表级锁是mysql中锁定粒度最大的一种锁，**表示对当前操作的整张表加锁**，它实现简单，资源消耗较少，被大部分mysql引擎支持。最常使用的MyISAM与InnoDB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）
* **特点**： 开销小，加锁快，不会出现死锁。发生锁冲突的概率最高，并发度也最低。

**页级锁：**

* **描述**：**页级锁是 MySQL 中锁定粒度介于行级锁和表级锁中间的一种锁**。表级锁速度快，但冲突多，行级冲突少，但速度慢。因此，采取了折衷的页级锁，一次锁定相邻的一组记录。BDB 支持页级锁。
* **特点**：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

### （4）共享锁 、 排他锁的描述

**共享锁：**

* **描述**：
  + 共享锁又称**读锁**，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。
  + 如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。
* **用法**：
  + SELECT … LOCK IN SHARE MODE;
  + 在查询语句后面增加LOCK IN SHARE MODE，MySQL 就会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

**排他锁：**

* **描述**：
  + 排他锁又称写锁、独占锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。
* **用法**：
  + SELECT … FOR UPDATE;
  + 在查询语句后面增加FOR UPDATE，MySQL 就会对查询结果中的每行都加排他锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请排他锁，否则会被阻塞。

用上面的例子来说就是用户的行为有两种,一种是来看房,多个用户一起看房是可以接受的. 一种是真正的入住一晚,在这期间,无论是想入住的还是想看房的都不可以.

* **InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁**

### （8）InnoDB 存储引擎有几种锁算法？

* **Record Lock** — 单个行记录上的锁；
* **Gap Lock** — 间隙锁，锁定一个范围，不包括记录本身；
* **Next-Key Lock** — 锁定一个范围，包括记录本身。

### （15）什么是全局锁？它的应用场景有哪些？

全局锁就是对整个数据库实例加锁，它的典型使用场景就是做全库逻辑备份，这个命令可以使用整个库处于只读状态，使用该命令之后，数据更新语句，数据定义语句，更新类事务的提交语句等操作都会被阻塞。

### （17）优化锁方面你有什么建议？

* 尽量使用较低的隔离级别。
* 精心设计索引， 并尽量使用索引访问数据， 使加锁更精确， 从而减少锁冲突的机会。
* 选择合理的事务大小，小事务发生锁冲突的几率也更小。
* 给记录集显示加锁时，最好一次性请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁。
* 不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大大减少死锁的机会。
* 尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响。
* 不要申请超过实际需要的锁级别。
* 除非必须，查询时不要显示加锁。 MySQL 的 MVCC 可以实现事务中的查询不用加锁，优化事务性能；MVCC 只在 COMMITTED READ（读提交）和 REPEATABLE READ（可重复读）两种隔离级别下工作。
* 对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能。

### （4）说一下MySQL是如何执行一条SQL的？具体步骤有哪些？

**Server层按顺序执行sql的步骤为：**

1.客户端请求->

2.连接器（验证用户身份，给予权限） ->

3.查询缓存（存在缓存则直接返回，不存在则执行后续操作）->

4.分析器（对SQL进行词法分析和语法分析操作） ->

5.优化器（主要对执行的sql优化选择最优的执行方案方法） ->

6.执行器（执行时会先看用户是否有执行权限，有才去使用这个引擎提供的接口）->

7.去引擎层获取数据返回（如果开启查询缓存则会缓存查询结果）

简单概括：

* **连接器**：管理连接、权限验证；
* **查询缓存**：命中缓存则直接返回结果；
* **分析器**：对SQL进行词法分析、语法分析；（判断查询的SQL字段是否存在也是在这步）
* **优化器**：执行计划生成、选择索引；
* **执行器**：操作引擎、返回结果；
* **存储引擎**：存储数据、提供读写接口。

### （5）SQL 的执行顺序？

SELECT DISTINCT

< select\_list >

FROM

< left\_table > < join\_type >

JOIN < right\_table > ON < join\_condition >

WHERE

< where\_condition >

GROUP BY

< group\_by\_list >

HAVING

< having\_condition >

ORDER BY

< order\_by\_condition >

LIMIT < limit\_number >

* 1它的执行顺序你知道吗？这道题就给你一个回答。

**FROM 连接**

首先，对 SELECT 语句执行查询时，对FROM 关键字两边的表执行连接，会形成笛卡尔积，这时候会产生一个虚表VT1(virtual table)

**ON 过滤**

然后对 FROM 连接的结果进行 ON 筛选，创建 VT2，把符合记录的条件存在 VT2 中。

**JOIN 连接**

第三步，如果是 OUTER JOIN(left join、right join) ，那么这一步就将添加外部行，如果是 left join 就把 ON 过滤条件的左表添加进来，如果是 right join ，就把右表添加进来，从而生成新的虚拟表 VT3。

**WHERE 过滤**

第四步，是执行 WHERE 过滤器，对上一步生产的虚拟表引用 WHERE 筛选，生成虚拟表 VT4。

**GROUP BY**

根据 group by 字句中的列，会对 VT4 中的记录进行分组操作，产生虚拟机表 VT5。果应用了group by，那么后面的所有步骤都只能得到的 VT5 的列或者是聚合函数（count、sum、avg等）。

**HAVING**

紧跟着 GROUP BY 字句后面的是 HAVING，使用 HAVING 过滤，会把符合条件的放在 VT6

**SELECT**

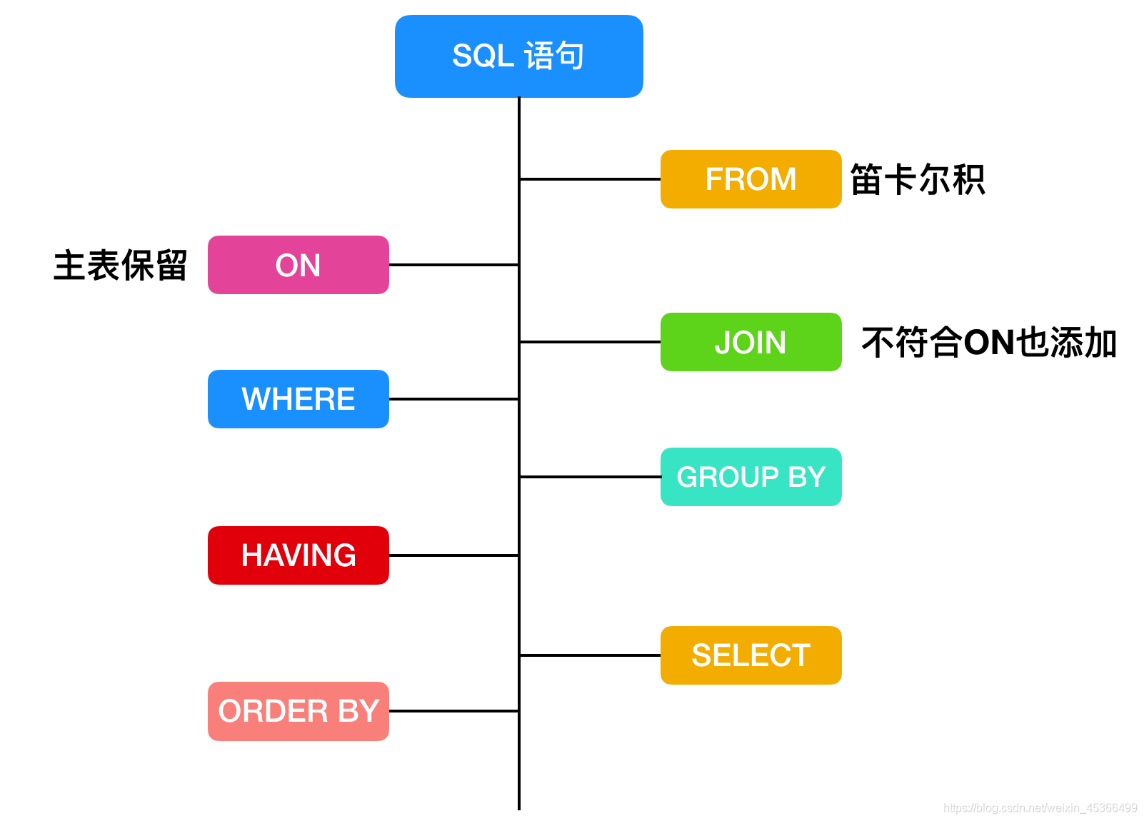
第七步才会执行 SELECT 语句，将 VT6 中的结果按照 SELECT 进行刷选，生成 VT7

**DISTINCT**

在第八步中，会对 TV7 生成的记录进行去重操作，生成 VT8。事实上如果应用了 group by 子句那么 distinct 是多余的，原因同样在于，分组的时候是将列中唯一的值分成一组，同时只为每一组返回一行记录，那么所以的记录都将是不相同的。

**ORDER BY**

应用 order by 子句。按照 order\_by\_condition 排序 VT8，此时返回的一个游标，而不是虚拟表。sql 是基于集合的理论的，集合不会预先对他的行排序，它只是成员的逻辑集合，成员的顺序是无关紧要的。

SQL 语句执行的过程如下：  


### （6）简述触发器、函数、视图、存储过程？

* **触发器**：使用触发器可以定制用户对表进行【增、删、改】操作时前后的行为,触发器无法由用户直接调用，而知由于对表的【增/删/改】操作被动引发的
* **函数**：是MySQL数据库提供的内部函数(当然也可以自定义函数)。这些内部函数可以帮助用户更加方便-的处理表中的数据
* **视图**：视图是虚拟表或逻辑表，它被定义为具有连接的SQL SELECT查询语句。
* **存储过程**：存储过程是存储在数据库目录中的一坨的声明性SQL语句，数据库中的一个重要对象,有效提高了程序的性能

### （6）听说过视图吗？那游标呢？

* 视图是一种**虚拟的表**，通常是有一个表或者多个表的行或列的子集，具有和物理表相同的功能 游标是对查询出来的结果集作为一个单元来有效的处理。
* 一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

**五、表结构相关**

### （1）为什么要尽量设定一个主键?

主键是数据库确保数据行在整张表唯一性的保障,即使业务上本张表没有主键,也建议添加一个自增长的 ID 列作为主键.设定了主键之后,在后续的删改查的时候可能更加快速以及确保操作数据范围安全。

### （2）主键使用自增 ID 还是 UUID?

**推荐使用自增ID，不要使用 UUID。**

因为在 InnoDB存储引擎中,主键索引是作为聚簇索引存在的,也就是说,主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序),如果主键索引是自增ID,那么只需要不断向后排列即可,如果是UUID,由于到来的ID与原来的大小不确定,会造成非常多的数据插入,数据移动,然后导致产生很多的内存碎片,进而造成插入性能的下降.

总之，在数据量大一些的情况下，用自增主键性能会好一些。

### （3）字段为什么要求定义为not null?

null 值会占用更多的字节，且会在程序中造成很多与预期不符的情况。

### （4）如果要存储用户的密码散列,应该使用什么字段进行存储?

密码散列，用户身份证号等固定长度的字符串应该使用 char 而不是 varchar 来存储，这样可以节省空间且提高检索效率。

### （5）说一说Drop、Delete与Truncate的共同点和区别?

* Drop直接删掉表;
* Truncate删除表中数据，再插入时自增长id又从1开始 ;
* Delete删除表中数据，可以加where字句。

**六、其他问题**

### （1）MySQL 中的 varchar 和 char 有什么区别?

char 是一个定长字段，假如申请了char(10)的空间，那么无论实际存储多少内容。该字段都占用 10 个字符，而 varchar 是变长的，也就是说申请的只是最大长度，占用的空间为实际字符长度 +1，最后一个字符存储使用了多长的空间

在检索效率上来讲，char > varchar，因此在使用中，如果确定某个字段的值的长度，可以使用 char，否则应该尽量使用 varchar。例如存储用户 MD5 加密后的密码，则应该使用 char。

**（2）varchar(10) 和 int(10) 代表什么含义?**

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.也就是说,varchar(10) 和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示.

### （6）left join、right join以及inner join的区别?

* left join：左关联，主表在左边，右边为从表。如果左侧的主表中没有关联字段，会用null 填满
* right join：右关联 主表在右边和letf join相反
* inner join： 内关联只会显示主表和从表相关联的字段，不会出现null

### （7）什么是数据库约束,常见的约束有哪几种?

**数据库约束用于保证数据库、表数据的完整性（正确性和一致性）**。  
可以通过定义约束\索引\触发器来保证数据的完整性。总体来讲,约束可以分为:

* 主键约束：primary key；
* 外键约束：foreign key；
* 唯一约束：unique；
* 检查约束：check；
* 空值约束：not null；
* 默认值约束：default；

### （9）简述数据库的读写分离？

读写分离为了确保数据库产品的稳数据定性，很多数据库拥有双机热备功能。也就是，第一台数据库服务器，是对外提供增删改业务的生产服务器；第二台数据库服务器，主要进行读的操作。

### （13）count(1)、count(\*)与count(列名)的执行区别

**执行效果上 ：**

* count(\*)：包括了所有的列，相当于行数，在统计结果的时候， **不会忽略列值为NULL**
* count(1)：包括了忽略所有列，用1代表代码行，在统计结果的时候， **不会忽略列值为NULL**
* count(列名)：只包括列名那一列，在统计结果的时候，**会忽略列值为空（这里的空不是只空字符串或者0，而是表示null）的计数， 即某个字段值为NULL时，不统计。**

**执行效率上：**

* 列名为主键，count(列名)会比count(1)快
* 列名不为主键，count(1)会比count(列名)快
* 如果表多个列并且没有主键，则 count（1） 的执行效率优于 count（\*）
* 如果有主键，则 select count（主键）的执行效率是最优的
* 如果表只有一个字段，则 select count（\*）最优。

### （14）sql 语句中where 1=1的作用

* **SQL注入**
* **语法规范**
* **拷贝表**
* **复制表结构**
* **1=1的坏处**

### （15）sql中null与空值的区别

* **1.占用空间区别**：空值(’’)的长度是0，是不占用空间的；而的NULL长度是NULL，是占用空间的
* **2.插入/查询方式区别**：NULL值查询使用is null/is not null查询，而空值(’’)可以使用=或者!=、<、>等算术运算符。
* **3.COUNT 和 IFNULL函数**：使用 COUNT(字段) 统计会过滤掉 NULL 值，但是不会过滤掉空值。
* **4.索引字段说明**：在有NULL值的字段上使用常用的索引，如普通索引、复合索引、全文索引等不会使索引失效。在官网查看在空间索引的情况下，说明了 索引列必须为NOT NULL。

## （1）日常工作中你是怎么优化SQL的？

（1.1）表结构优化

1. 尽量使用数字型字段

若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

1. 尽可能的使用 varchar 代替 char

变长字段存储空间小，可以节省存储空间。

1. 当索引列大量重复数据时，可以把索引删除掉

比如有一列是性别，几乎只有男、女、未知，这样的索引是无效的。

（1.2）查询优化

1. 应尽量避免在 where 子句中使用!=或<>操作符
2. 应尽量避免在 where 子句中使用 or 来连接条件
3. 任何查询也不要出现select \*
4. 避免在 where 子句中对字段进行 null 值判断

（1.3）索引优化

1. 对作为查询条件和 order by的字段建立索引
2. 避免建立过多的索引，多使用组合索引

（1.4）慢查询优化

1. 分析语句，是否加载了不必要的字段/数据
2. 分析 SQL 执行句话，是否命中索引等
3. 如果 SQL 很复杂，优化 SQL 结构
4. 如果表数据量太大，考虑分表