目录

[一: JAVA基础: 6](#_Toc9786)

[1. JDK 和 JRE 有什么区别？ 6](#_Toc20596)

[2. Java 内存区域 6](#_Toc10915)

[3. == 和 equals 的区别是什么？ 7](#_Toc25137)

[4. final 在 Java 中有什么作用？ 7](#_Toc14119)

[5. String 类的常用方法都有那些？ 7](#_Toc7061)

[6. Java 容器都有哪些？ 7](#_Toc456)

[7. 数组与链表的区别 9](#_Toc7270)

[8. Map集合几种遍历方式? 9](#_Toc3547)

[9. List集合的特性 9](#_Toc15037)

[10. 如何实现数组和 List 之间的转换？ 10](#_Toc8388)

[11. ArrayList 和 LinkedList 有什么区别 10](#_Toc5978)

[12. List、set和map的区别 10](#_Toc11651)

[13. HashMap的特性 11](#_Toc24950)

[14. 说一下 HashMap 的实现原理？ 11](#_Toc30148)

[15. 说一下 HashSet 的实现原理？ 11](#_Toc11801)

[16. CurrentHashMap了解吗? 11](#_Toc21581)

[17. String,StringBuilder,StringBuffer三者的区别 11](#_Toc10057)

[18. 抽象类与接口区别 11](#_Toc1023)

[19. 类的实例化方法调用顺序 12](#_Toc14733)

[20. JAVA反射机制提供了什么功能 12](#_Toc4041)

[21. Stream常用方法： 12](#_Toc23183)

[22. 常见的异常类有哪些？ 13](#_Toc15163)

[23. Java中异常处理机制 13](#_Toc6641)

[24. 线程和进程的区别？ 13](#_Toc28821)

[25. 创建线程的方式 13](#_Toc30084)

[26. 线程的状态 14](#_Toc26770)

[27. Java中wait方法和sleep方法的不同之处 14](#_Toc29831)

[28. 线程池的种类 14](#_Toc8505)

[29. 线程池都有哪些状态？ 15](#_Toc27037)

[30. 线程同步 15](#_Toc9334)

[31. 多线程中 synchronized 锁升级的原理是什么？ 15](#_Toc31118)

[32. volatile与synchronized的区别，底层实现 15](#_Toc25680)

[33. 什么是死锁？ 16](#_Toc13596)

[34. 怎么防止死锁？ 16](#_Toc25075)

[35. IO对比总结 16](#_Toc18031)

[36. 反射，泛型项目中有没有使用 16](#_Toc15849)

[37. Java虚拟机有哪几块内存空间 16](#_Toc30433)

[38. JAVA中垃圾回收机制 16](#_Toc820)

[39. 类加载过程 17](#_Toc14391)

[40. 对象创建过程 17](#_Toc19500)

[41. 方法区堆栈溢出怎么处理 17](#_Toc29827)

[42. 单例设计模式中懒汉式和饿汉式的区别 17](#_Toc5961)

[43. 常见的基本排序 18](#_Toc16969)

[44. 常见设计模式 19](#_Toc4929)

[二:WEB: 19](#_Toc18079)

[1. cookie和session的区别与联系 19](#_Toc9511)

[2. 如果客户端禁止 cookie 能实现 session 还能用吗？ 19](#_Toc7509)

[3. Get方法和Post方法区别 19](#_Toc16493)

[4. 过滤器有哪些作用，以及过滤器的生命周期？ 20](#_Toc8048)

[5. servlet的生命周期及常用方法 20](#_Toc10224)

[6. 转发和重定向的区别 20](#_Toc25220)

[7. ajax书写方式及内部主要参数 20](#_Toc10366)

[8. Jquery常用选择器 21](#_Toc24480)

[9. JSP和Servlet的区别 21](#_Toc8309)

[10. JSP常用的标签 21](#_Toc26357)

[11. jsp九大内置对象及作用 22](#_Toc1386)

[12. JSP四大作用域及请求范围 22](#_Toc27998)

[13. 如何防止表单重复提交 22](#_Toc24570)

[14. 常见的http返回状态码 22](#_Toc27501)

[15. TCP和UDP的区别,HTTP协议 23](#_Toc14131)

[16. tcp为什么要三次握手，两次不行吗？为什么？ 23](#_Toc433)

[17. 说一下 tcp 粘包是怎么产生的？ 23](#_Toc28729)

[18. json的数据格式 23](#_Toc29770)

[19. 如何实现跨域？ 24](#_Toc3085)

[20. Tomcat体系结构讲解 24](#_Toc24588)

[21. tomcat 如何调优，涉及哪些参数 25](#_Toc15529)

[三:数据库: 26](#_Toc23993)

[1. 事务的特性和隔离级别 26](#_Toc21443)

[2. Mysql的体系结构 26](#_Toc11298)

[3. InnoDB，MyISAM存储引擎特性 27](#_Toc24831)

[4. CHAR和VARCHAR的区别： 27](#_Toc41)

[5. delete、drop、truncate区别 27](#_Toc16983)

[6. mysql 中 in 和 exists 区别 27](#_Toc20987)

[7. 数据库的三范式是什么？ 28](#_Toc16629)

[8. Sql优化 28](#_Toc30976)

[9. 分库分表方案 29](#_Toc26089)

[10. MySQL 索引是怎么实现的？ 29](#_Toc13384)

[11. Mysql支持的索引类型 29](#_Toc19787)

[12. 索引失效问题 30](#_Toc27532)

[13. 左连接 ，右连接，内连接和全外连接的4者区别 30](#_Toc13273)

[14. 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 MySQL 数据库，又插入了一条数据，此时 id 是几？ 30](#_Toc5364)

[15. 说一下 MySQL 的行锁和表锁？ 30](#_Toc6668)

[16. 说一下乐观锁和悲观锁？ 30](#_Toc20028)

[17. 高并发下，如何做到安全的修改同一行数据。 30](#_Toc28976)

[18. 数据库会死锁吗，举一个死锁的例子 31](#_Toc27643)

[19. 聚集索引和非聚集索引的区别 31](#_Toc10771)

[四:框架: 31](#_Toc6160)

[1. Spring 31](#_Toc31292)

[2. spring 有哪些主要模块？ 32](#_Toc8319)

[3. Spring AOP 32](#_Toc5069)

[4. IOC和DI 32](#_Toc25166)

[5. Spring中Bean的作用域 32](#_Toc11643)

[6. Spring框架实现实例化和依赖注入的方式 33](#_Toc22553)

[7. springmvc执行流程 33](#_Toc25409)

[8. Spring以及SpringMVC常用注解 33](#_Toc21111)

[9. springmvc获取表单的几种方式 34](#_Toc32187)

[10. Spring MVC的异常处理 ？ 34](#_Toc20589)

[11. 为什么要用 spring boot？ 34](#_Toc23885)

[12. Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？ 34](#_Toc24697)

[13. spring boot 有哪些方式可以实现热部署？ 34](#_Toc28542)

[14. SpringBoot 的常用注解有哪些？ 35](#_Toc30465)

[15. SpringBoot 有哪几种读取配置的方式？ 35](#_Toc2098)

[16. SpringBoot 配置加载顺序？ 35](#_Toc19750)

[17. Spring Boot 如何定义多套不同环境配置 36](#_Toc1443)

[18. Mybatis中使用#和$书写占位符有什么区别 36](#_Toc13135)

[19. 动态SQL 36](#_Toc7507)

[20. Mapper动态代理规范 36](#_Toc23781)

[21. Mybatis常用注解 37](#_Toc24899)

[22. springcloud如何实现服务的注册和发现 37](#_Toc5912)

[23. ribbon和feign区别 37](#_Toc18489)

[24. springcloud断路器的作用（Hystrix） 37](#_Toc10919)

[项目技术点: 38](#_Toc6362)

[五:MQ： 38](#_Toc26876)

[1.为什么使用MQ? 38](#_Toc22235)

[2.你了解哪些MQ技术，ActiveMQ,RabbitMQ,RocketMQ,kafka 38](#_Toc10013)

[3.RabbitMQ 的使用场景有哪些？ 39](#_Toc2727)

[4.如何保证MQ的高可用? 39](#_Toc15688)

[5.如何保证消息不被重复消费(如何保证消息消费时的幂等性)? 39](#_Toc16156)

[6.如何保证消息不丢失 39](#_Toc6185)

[7.如何保证消息的顺序性 40](#_Toc11861)

[8.消息大量积压怎么解决？ 40](#_Toc4367)

[9.RabbitMQ 节点的类型有哪些？ 40](#_Toc7574)

[10.rabbitmq如何确认消息一定发送到了消息中间件中呢? 40](#_Toc11410)

[11.rabbitmq的集群 40](#_Toc10714)

[六:Redis： 40](#_Toc7036)

[1. 使用redis缓存的好处 40](#_Toc4502)

[2. Redis的key和value的存储大小有限制吗？ 41](#_Toc30715)

[3. redis存储什么数据类型 41](#_Toc11680)

[4. 使用redis缓存的弊端 41](#_Toc18790)

[5. 为什么redis是单线程的但是还可以支撑高并发? 42](#_Toc6759)

[6. Redis为什么效率高? 42](#_Toc27414)

[7. Redis由哪些数据类型,分别在哪些场景下使用? 42](#_Toc23522)

[8. Redis过期策略 42](#_Toc5925)

[9. 怎样做redis支撑高并发(读多写少用缓存,读少写多用队列) 43](#_Toc22996)

[10. redis的持久化 43](#_Toc7189)

[11. redis怎么设置缓存大小 44](#_Toc16218)

[12. redis集群主从数据如何同步 44](#_Toc15524)

[13. Redis缓存穿透和redis雪崩处理 45](#_Toc24552)

[七:Elasticsearch 45](#_Toc29537)

[1.Elasticsearch是如何实现Master选举的 45](#_Toc5401)

[2.在并发情况下，Elasticsearch如果保证读写一致？ 45](#_Toc24504)

[3.Elasticsearch索引文档的过程 46](#_Toc25592)

[4.Elasticsearch更新和删除文档的过程 46](#_Toc4239)

[5.Elasticsearch搜索的过程 46](#_Toc28086)

[6.ElasticSearch中的集群、节点、索引、文档、类型是什么？ 46](#_Toc16560)

[7.elasticsearch 索引数据多了怎么办，如何调优 47](#_Toc3812)

[八:Dubbo和Zookeeper： 47](#_Toc31632)

[1. dubbo的工作流程 47](#_Toc27050)

[2. Dubbo的通信原理？ 47](#_Toc5422)

[3. dubbo负载均衡策略有哪些? 47](#_Toc15001)

[4.zookeeper注册中心挂了，dubbo可以继续使用吗？ 47](#_Toc30247)

[5.zookeeper 都有哪些功能？ 47](#_Toc5371)

[6.Zookeeper的watch机制（通知机制） 48](#_Toc1333)

[7.zookeeper 怎么保证主从节点的状态同步？ 48](#_Toc19162)

[8.什么是 zab 协议 48](#_Toc416)

[9.如果zookeeper服务挂了怎么办？ 48](#_Toc9752)

[九:分布式系统： 48](#_Toc32386)

[1. 怎么实现远程通信 48](#_Toc833)

[2. 分布式事务： 49](#_Toc27840)

[3. 系统的高并发问题是怎么解决的 49](#_Toc9220)

[4. 分布式锁的问题 50](#_Toc21024)

[电话面试: 50](#_Toc32384)

[1. Mq 干嘛的 50](#_Toc32248)

[2. Cas登录 51](#_Toc14263)

[3. Redis使用场景 52](#_Toc20535)

[石山码农第一季 52](#_Toc10989)

[1、分布式系统 52](#_Toc3549)

[1.1 为什么要进行系统拆分？ 52](#_Toc11458)

[1.2 分布式服务框架 52](#_Toc22053)

[1.3 分布式锁 53](#_Toc28734)

[1.4 分布式事务 53](#_Toc9693)

[1.5 分布式会话 53](#_Toc25905)

[2、高并发架构 53](#_Toc719)

[2.1 如何设计一个高并发系统？ 53](#_Toc25232)

[2.2 消息队列 53](#_Toc30506)

[（1）为什么使用消息队列啊？消息队列有什么优点和缺点啊？kafka、activemq、rabbitmq、rocketmq都有什么优点和缺点啊？ 53](#_Toc17589)

[（2） 如何保证消息队列的高可用啊？ 60](#_Toc1337)

[（3）如何保证消息不被重复消费啊（如何进行消息队列的幂等性问题）？ 63](#_Toc9903)

[（4）如何保证消息的可靠性传输（如何处理消息丢失的问题）？ 65](#_Toc15867)

[（5）如何保证消息的顺序性？ 69](#_Toc22275)

[（6）如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？ 71](#_Toc14677)

[（7） 如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路 72](#_Toc27537)

[（8） 总结一下消息队列相关问题的面试技巧 72](#_Toc25844)

[2.3 搜索引擎 74](#_Toc13687)

[（1）es的分布式架构原理能说一下么（es是如何实现分布式的啊）？ 74](#_Toc1817)

[（2）es写入数据的工作原理是什么啊？es查询数据的工作原理是什么啊？底层的lucene介绍一下呗？倒排索引了解吗？ 77](#_Toc7459)

[（3）es在数据量很大的情况下（数十亿级别）如何提高查询效率啊？ 79](#_Toc31970)

[（4） es生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？ 84](#_Toc28669)

[2.4 缓存 86](#_Toc14292)

[（1）在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？ 86](#_Toc13385)

[（2）redis和memcached有什么区别？redis的线程模型是什么？为什么单线程的redis比多线程的memcached效率要高得多？ 88](#_Toc23724)

[（3）redis都有哪些数据类型？分别在哪些场景下使用比较合适？ 91](#_Toc32049)

[（5）redis的过期策略都有哪些？手写一下LRU代码实现？ 93](#_Toc13762)

[（6）如何保证Redis高并发、高可用、持久化？redis的主从复制原理能介绍一下么？redis的哨兵原理能介绍一下么？ 96](#_Toc18463)

[（7） redis的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？ 113](#_Toc32135)

[（8） redis集群模式的工作原理能说一下么？在集群模式下，redis的key是如何寻址的？分布式寻址都有哪些算法？了解一致性hash算法吗？如何动态增加和删除一个节点？ 122](#_Toc30149)

[（9） 了解什么是redis的雪崩和穿透？redis崩溃之后会怎么样？系统该如何应对这种情况？如何处理redis的穿透？ 131](#_Toc27102)

[（10） 如何保证缓存与数据库的双写一致性？ 133](#_Toc17542)

[（11） redis的并发竞争问题是什么？如何解决这个问题？了解Redis事务的CAS方案吗？ 141](#_Toc26472)

[（12） 生产环境中的redis是怎么部署的？ 142](#_Toc7971)

[2.5 分库分表 142](#_Toc11255)

[1. 为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？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？ 142](#_Toc27100)

[2.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？ 147](#_Toc16831)

[3.如何设计可以动态扩容缩容的分库分表方案？ 150](#_Toc2388)

[4.分库分表之后，id主键如何处理？ 152](#_Toc30982)

[2.6 读写分离 158](#_Toc31242)

[（1）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？ 158](#_Toc23650)

[3、高可用架构 162](#_Toc17441)

[3.1 如何设计一个高可用系统？ 162](#_Toc25535)

[3.2 限流 162](#_Toc14020)

[（1） 如何限流？在工作中是怎么做的？说一下具体的实现？ 162](#_Toc17607)

[3.3 熔断 162](#_Toc28536)

[（1）如何进行熔断？熔断框架都有哪些？具体实现原理知道吗？ 162](#_Toc29542)

[3.4 降级 162](#_Toc5388)

[（1）如何进行降级？ 162](#_Toc15223)

[4\_如何设计高可用系统架构？限流？熔断？降级？什么鬼！ 162](#_Toc25828)

[Hadoop 185](#_Toc18936)

[1.1.1 数据采集与展示 185](#_Toc7615)

[告警中心: 186](#_Toc10576)

[组件监控: 186](#_Toc538)

[主机监控: 187](#_Toc5302)

**一: JAVA基础:**

1. **JDK 和 JRE 有什么区别？**

JDK：Java Development Kit 的简称，Java 开发工具包，提供了 Java 的开发环境和运行环境。

JRE：Java Runtime Environment 的简称，Java 运行环境，为 Java 的运行提供了所需环境。

具体来说 JDK 其实包含了 JRE，同时还包含了编译 Java 源码的编译器 Javac，还包含了很多 Java 程序调试和分析的工具。简单来说：如果你需要运行 Java 程序，只需安装 JRE 就可以了，如果你需要编写 Java 程序，需要安装 JDK。

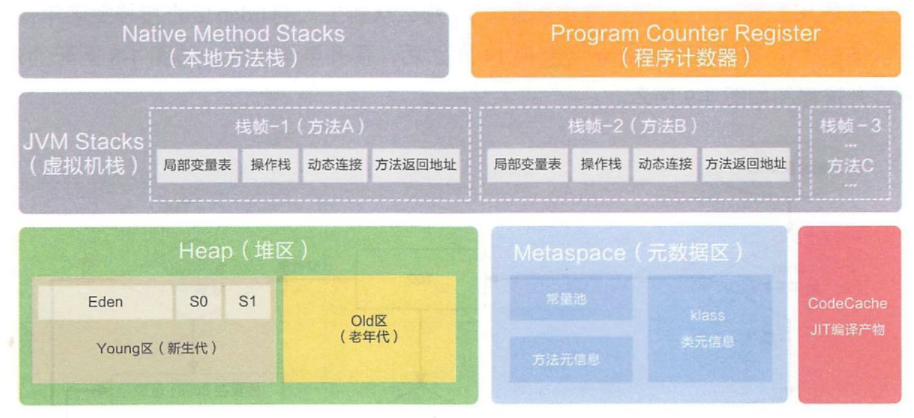
1. **Java 内存区域**

Java 虚拟机在执行 Java 程序的过程中会把他所管理的内存划分为若干个不同的数据区域。Java 虚拟机规范将 JVM 所管理的内存分为以下几个运行时数据区：程序计数器、Java 虚拟机栈、本地方法栈、Java 堆、元数据区。

JDK8 之前，Hotspot 中方法区的实现是永久代（Perm），JDK8 开始使用元空间（Metaspace），以前永久代所有内容的字符串常量移至堆内存，其他内容移至元空间，元空间直接在本地内存分配。

为什么要使用元空间取代永久代的实现？

1. 字符串存在永久代中，容易出现性能问题和内存溢出。
2. 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
3. 永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。
4. 将 HotSpot 与 JRockit 合二为一。



1. **== 和 equals 的区别是什么？**

== 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；

equals 默认情况下是引用比较，只是很多类重新了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等。

1. **final 在 Java 中有什么作用？**

final 修饰的类叫最终类，该类不能被继承。  
 final 修饰的方法不能被重写。  
 final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

1. **String 类的常用方法都有那些？**

indexOf()：返回指定字符的索引。

charAt()：返回指定索引处的字符。

replace()：字符串替换。

trim()：去除字符串两端空白。

split()：分割字符串，返回一个分割后的字符串数组。

getBytes()：返回字符串的 byte 类型数组。

length()：返回字符串长度。

toLowerCase()：将字符串转成小写字母。

toUpperCase()：将字符串转成大写字符。

substring()：截取字符串。

equals()：字符串比较。

1. **Java 容器都有哪些？**

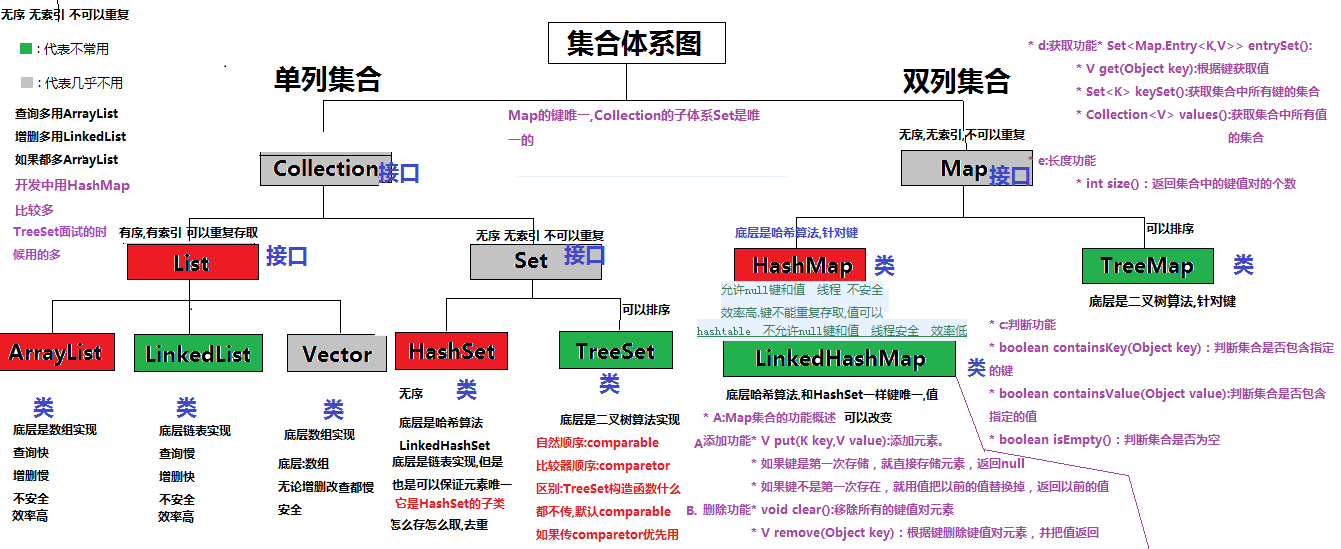
Java 容器分为 Collection 和 Map 两大类，其下又有很多子类，如下所示：

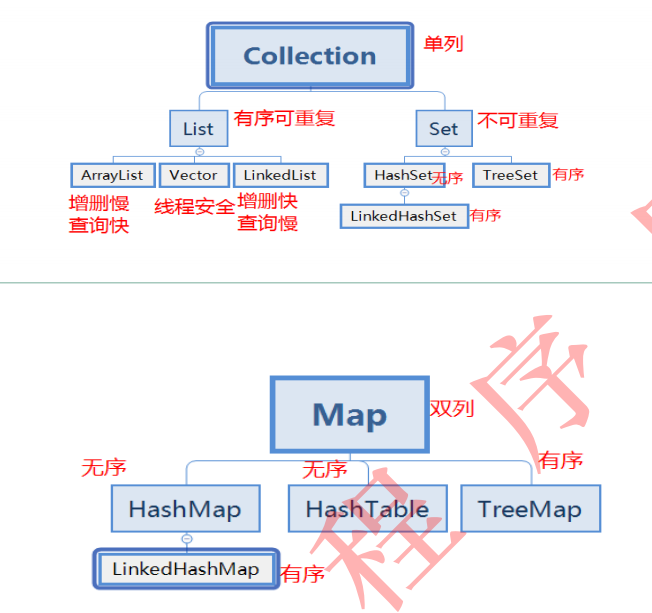
Collection

List、ArrayList、LinkedList、Vector、Stack、Set、HashSet、LinkedHashSet、TreeSet

Map

HashMap、LinkedHashMap、TreeMap、ConcurrentHashMap、Hashtable





1. **数组与链表的区别**

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；　arrayList和linkedList区别

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(1)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：   
　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；   
　　链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

1. **Map集合几种遍历方式?**

第一种:根据键找值方式遍历

第二种:获取所有的键值对对象集合,通过迭代器遍历

第三种:获取所有的键值对对象集合,通过增强for遍历

第四种:通过Map集合中values方法,拿到所有的值

1. **List集合的特性**

ArrayList

1.ArrayList是List接口可变数组的实现,允许存放null

2.底层是使用数组实现,无参构造函数默认初始化长度为10,数组扩容是会将原数组中的元素重新拷贝到新数组中,长度为原来的1.5+1(代扩容价高)

3.线程非同步.

LinkedList

1.LinkedList是List接口双向链表的非同步实现,允许存放null

2.底层的数据结构是基于双向链表,数据结构是节点

3.双向链表中每个节点分为prev,next,item,其中prev中存放上一个节点的信息,next存放下一个节点的信息,item存放该节点的值

1. **如何实现数组和 List 之间的转换？**

数组转 List：使用 Arrays. asList(array) 进行转换。  
 List 转数组：使用 List 自带的 toArray() 方法。

1. **ArrayList 和 LinkedList 有什么区别**

ArrayList和LinkedList都实现了List接口，有以下的不同点：  
 1、ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。  
 2、相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。  
3、LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

1. **List、set和map的区别**

list和set是实现了collection接口的。

List：1.可以允许重复的对象。

　　 2.可以插入多个null元素。

    3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

Set： 1.不允许重复对象

　　 2. 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator  或者 Comparable 维护了一个排序顺序。

    3. 只允许一个 null 元素

4.Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

Map不是collection的子接口或者实现类。Map是一个接口。

1.Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。

2. TreeMap 也通过 Comparator  或者 Comparable 维护了一个排序顺序。

3.Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。

4.Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

1. **HashMap的特性**

* Map集合的特点

1. Map是一个双列集合,将键映射到值的对象.
2. Map集合的数据结构,只针对键有效,跟值没关系.
3. 一个映射不能包含重复的键,每个键最多只能映射一个值.

* HashMap的数据结构

1. 哈希表结构:数组+链表
2. 通过哈希表结构配合对象的hashCode和equals方法就可以确保键的唯一性.

* HashMap和Hashtable的区别

1. HashMap是jdk1.2版本出现的,允许存储null键和null值

不同步(线程不安全):效率高

1. Hashtable是jdk1.0版本出现的,不允许存储null键和null值

同步(线程安全):效率低

1. **说一下 HashMap 的实现原理？**

HashMap 基于 Hash 算法实现的，我们通过 put(key,value)存储，get(key)来获取。当传入 key 时，HashMap 会根据 key. hashCode() 计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。当计算出的 hash 值相同时，我们称之为 hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 hash 冲突的个数比较少时，使用链表否则使用红黑树

1. **说一下 HashSet 的实现原理？**

HashSet 是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，HashSet 不允许重复的值。

1. **CurrentHashMap了解吗?**

HashMap是线程不安全的 , 但是效率高 , HashTable是线程安全的 , 但是效率低.有没有一种对象是即是线程安全的 , 同时执行效率可以达到HashMap呢?

CurrentHashMap可以做到。底层实现通过分段加锁进行实现 , hashmap底层是数组加上链表实现的 , 那么一个线程来操作数据,只是操作数组中一个索引的数据. 如果此时对整个数组加锁,其他线程操作不了这个数组,所以效率低.其实线程也就操作数组的一个索引,对这个索引进行加锁 , 而锁对象就是这个索引所对应的值,其他线程来修改其他索引数据时,拿到的是其他索引的锁对象,从而提高了效率.

1. **String,StringBuilder,StringBuffer三者的区别**

* String和StringBuilder的本质区别

String是一个不可改变的字符序列.

StringBuilder是一个可以改变的字符序列.

* 常见的字符拼接,该选择谁

推荐使用StringBuilder,因为拼接的效率高

* StringBuilder和StringBuffer的区别

1. StringBuilder和StringBuffer的功能是完全一致的.
2. 不同点

StringBuffer是jdk1.0出现的,线程安全(同步):效率低.

StringBuilder是jdk1.5出现的,线程不安全(不同步):效率高.

1. **抽象类与接口区别**

* 抽象类可以有构造方法，接口中不能有构造方法
* 抽象类中可以有普通成员变量，接口中没有普通成员变量
* 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的
* 抽象类中的抽象方法的访问类型可以是public，protected，但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型
* 抽象类中可以包含静态方法，接口中不能包含静态方法
* 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型
* 一个类可以实现多个接口，但只能继承一个抽象类。
* 接口关注的是功能，抽象类关注的抽象现实中事物

1. **类的实例化方法调用顺序**

此题考察的是类加载器实例化时进行的操作步骤（加载–>连接->初始化）。  
 父类静态代变量、  
 父类静态代码块、  
 子类静态变量、  
 子类静态代码块、  
 父类非静态变量（父类实例成员变量）、  
 父类构造函数、  
 子类非静态变量（子类实例成员变量）、  
 子类构造函数。

1. **JAVA反射机制提供了什么功能**

Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

Java反射机制提供如下功能：

在运行时判断任意一个对象所属的类

在运行时构造任意一个类的对象

在运行时判段任意一个类所具有的成员变量和方法

在运行时调用任一个对象的方法

在运行时创建新类对象

在使用Java的反射功能时，基本首先都要获取类的Class对象，再通过Class对象获取其他的对象。

1. **Stream常用方法：**

* map: 用作类型转换 如把集合里面的字符串转为大写,或者一个对象的集合取几个字段转为新的对象集合
* filter: 过滤 符合条件的集合元素保存下来,不符合条件的去掉
* flatMap:合并集合,比如List<Album> Album里面有一LIst<Track> 对象,这个时候就能不通过循环的方式把 List<Album> 里的每一个元素的 trasks 对象组装成一个新的集合
* reduce: reduce可以做累加运算, .reduce(0, (a,b)-> a+b);
* count: count和size一样返回的是元素的个数
* max,min: 求最大值和最小值,这两个方法需要传入一个comparator比较器,Comparator比较器有一个comparing() 方法
* anyMatch表示，判断的条件里，任意一个元素成功，返回true
* allMatch表示，判断条件里的元素，所有的都是，返回true

1. **常见的异常类有哪些？**

* NullPointerException 空指针异常
* ClassNotFoundException 指定类不存在
* NumberFormatException 字符串转换为数字异常
* IndexOutOfBoundsException 数组下标越界异常
* ClassCastException 数据类型转换异常
* FileNotFoundException 文件未找到异常
* NoSuchMethodException 方法不存在异常
* IOException IO 异常
* SocketException Socket 异常

1. **Java中异常处理机制**

* 什么是异常

异常指的就是程序的不正常,简单理解就是程序所发生的错误.

* 异常的体系结构&分类

1. 分类
2. 编译时异常:指的就是编译期间,编译器检测到某段代码可能会发生某些问题,需要程序员提前给代码做出错误的解决方案,否则编译是不通过的.(例如FileReader)
3. 运行时异常:指的是编译通过了,但运行时出现的错误.
4. 体系结构

Throwable

Error：严重性错误

Exception：

RuntimeException:运行时异常

!RuntimeException:编译时异常

* 异常产生的原理

java对异常默认的处理方式,是将问题抛出给上一级

抛出之前,java会根据错误产生的异常类,创建出该类的对象,底层并通过throw关键字将异常抛出给上一级,不断向上抛出,直到抛给了JVM虚拟机,虚拟机拿到问题之后,就会将错误的原因和所在的位置,打印在控制台.

* 异常的处理方式

1. 问题可以自己处理掉的

try...catch处理方式:自己将问题处理掉,不会影响到后续代码的继续执行.

1. 问题自己处理不掉的

throws抛出处理方式：如果发现问题自己无法完美结局,就可以通过throw关键字,将异常对象抛出给调用者,但如果使用throw抛出异常对象,则方法上面必须进行throws的声明,告知调用者此方法存在异常.

细节:如果抛出的对象是RuntimeException,则方法上面无需throws声明.

1. **线程和进程的区别？**

* 进程：进程指正在运行的程序。确切的来说，当一个程序进入内存运行，即变成一个进程，进程是处于运行过程中的程序，并且具有一定独立功能。
* 线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。
* 简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程

车间 多流水线

1. **创建线程的方式**

* 继承Thread类

优点:代码简单

缺点:该类无法继承别的类

* 实现Runnable接口

优点:继承其他类.统一实现该接口的实例可以共享资源.

缺点:代码复杂.

* 实现Callable接口

Callable中的call()方法有返回值,其他和Runnable的run()方法一样.

* 线程池方式

优点:实现自动化装配,易于管理,循环利用资源.

1. **线程的状态**

* 就绪(Runnable):线程准备运行，不一定立马就能开始执行。
* 运行中(Running)：进程正在执行线程的代码。
* 等待中(Waiting):线程处于阻塞的状态，等待外部的处理结束。
* 睡眠中(Sleeping)：线程被强制睡眠。
* I/O阻塞(Blocked on I/O)：等待I/O操作完成。
* 同步阻塞(Blocked on Synchronization)：等待获取锁。
* 死亡(Dead)：线程完成了执行。

1. **Java中wait方法和sleep方法的不同之处**

* wait和sleep的基本使用:

wait:此方法来自于Object类,必须由锁对象进行调用

sleep:此方法来自于Thread类,是Thread类的静态方法,可以类名点调用

* wait方法和sleep方法的原理对比

sleep:让当前程序休眠xxx毫秒,休眠之后,程序继续执行.

wait:如果使用的wait方法是传入毫秒值参数的,产生的效果和sleep类似,但是wait方法较为麻烦一点,wait方法必须由锁对象调用,锁对象还必须放在同步当中.

* 总结:

wait方法:

1. 空参数:会让线程进入无限等待状态,进入无限等待状态后,必须由notify方法对其进行唤醒.

重点:

wait方法在等待的过程中,释放锁对象.

sleep方法在休眠的过程中,不会释放锁对象.

1. 有参数的:效果和sleep方法类似.
2. **线程池的种类、参数、方法**

* newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
* newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
* newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
* newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行

因为在项目中，线程的创建和销毁非常消耗资源，所以使用在多线程场景的时候会使用线程池，根据自己业务逻辑的需求，使用不同的线程池。

**参数**

corePoolSize ：核心线程数量

maximumPoolSize ：线程最大线程数

workQueue ：阻塞队列，存储等待执行的任务 很重要 会对线程池运行产生重大影响

keepAliveTime ：线程没有任务时最多保持多久时间终止

unit ：keepAliveTime的时间单位

threadFactory ：线程工厂，用来创建线程

rejectHandler ：当拒绝处理任务时的策略

**方法:**

execute（）：提交任务，交给线程池执行

submit（）：提交任务，能够返回执行结果 execute + Future

shutdown（）：关闭线程池，等待任务都执行完

shutdownNow（）：关闭线程池，不等待任务执行完

getTaskCount（）：线程池已执行和未执行的任务总数

getCompletedTaskCount（）：已完成的任务数量

getPoolSize（）：线程池当前的线程数量

getActiveCount（）：当前线程池中正在执行任务的线程数量

1. **线程池都有哪些状态？**

* RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
* SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
* STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
* TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
* TERMINATED：terminated()方法结束后，线程池的状态就会变成这个

1. **线程同步**

* 线程同步：实现共享数据的一致性，让多个线程有序的访问共享资源，而不是同时操作共享资源
* 方法同步：用关键字 synchonized 可将方法声明为同步
* 同步代码块：synchornized 获取的是参数中的对象锁，synchornized(obj){}
* 同步类的属性：在类中声明多个object对象，对每个代码块分别操作类的不同属性加锁的时候就分别使用object对象来作为锁，这样能保证多个线程同时运行，分别操作不同的对象
* synchronized 静态方法与非静态方法：非静态同步方法的锁对象是this，静态的同步方法的锁对象是当前类的字节码对象
* 当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本

1. **多线程中 synchronized 锁升级的原理是什么？**

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

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

1. **volatile与synchronized的区别，底层实现**

* 首先两者都是用来确保数据的一致性的，volatile它能够使变量在值发生改变时尽快让其他线程知道，为什么要这么做呢？编译器为了加快程序运行速度，对一些变量的写操作会现在寄存器或者是cpu缓存上进行，最后才写入内存，这个过程，变量的新值对其他线程是不可见的，而volatile的作用就是使它修饰的变量的读写操作都必须在内存中进行

区别：

* Volatile本质是告诉jvm当前变量在寄存器中的值是不安全的需要从内存中读取，sychronized则是锁定当前变量，只有当前线程可以访问到该变量其他线程被阻塞
* Volatile只能作用于变量，synchronized则是可以使用在变量和方法上
* Volatile仅能实现变量的修改可见性,但不具备原子特性,而synchronized则可以保证变量的修改可见性和原子性
* volatile不会造成线程的阻塞,而synchronized可能会造成线程的阻塞
* volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化

注意事项：

* 在使用volatile关键字时要慎重，并不是只要简单类型变量使用volatile修饰，对这个变量的所有操作都是原来操作，当变量的值由自身的上一个决定时，如n=n+1、n++ 等，volatile关键字将失效，只有当变量的值和自身上一个值无关时对该变量的操作才是原子级别的，如n = m + 1，这个就是原级别的。所以在使用volatile关键时一定要谨慎，如果自己没有把握，可以使用synchronized来代替volatile

1. **什么是死锁？**

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

1. **怎么防止死锁？**

* 尽量使用 tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用 Java. util. concurrent 并发类代替自己手写锁。
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。
* 尽量减少同步的代码块。

1. **IO对比总结**

* IO 的方式通常分为几种：同步阻塞的 BIO、同步非阻塞的 NIO、异步非阻塞的 AIO。
* 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。
* 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 开始支持。
* 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 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

举个例子：

* 同步阻塞：你到饭馆点餐，然后在那等着，啥都干不了，饭馆没做好，你就必须等着！
* 同步非阻塞：你在饭馆点完餐，就去玩儿了。不过玩一会儿，就回饭馆问一声：好了没 啊！
* 异步非阻塞：饭馆打电话说，我们知道您的位置，一会给你送过来，安心玩儿就可以了， 类似于现在的外卖。

1. **反射，泛型项目中有没有使用**

* 反射：spring框架，底层使用大量反射，但是我们直接使用IOC注入对象，所以并不直接使用。除非特殊情况下，自己封装一些工具类的时候使用到（最后一句可以不说）。
* 泛型：底层肯定也是大量使用的，写的工具类，返回给前端对象中，对外的参数List Map等可以使用泛型。

1. **Java虚拟机有哪几块内存空间**

* 程序计数器：可以看作是当前线程所执行的字节码文件（class）的行号指示器，它会记录执行痕迹，是每个线程私有的；
* 栈：栈是运行时创建的，是线程私有的，生命周期与线程相同，存储声明的变量的
* 本地方法栈：为native方法服务，native方法是一种由非java语言实现的java方法，为什么使用这种方法呢，与java环境外交互，或者与操作系统交互
* 堆：堆是所有线程共享的一块内存，是在java虚拟机启动时创建的，几乎所有对象实例都在此创建，所以经常发生垃圾回收操作；
* 方法区：主要存储已被虚拟机加载的类的信息，常量，静态变量和即时编译器编译后的代码等数据，该区域是被线程共享的，很少发生垃圾回收

1. **JAVA中垃圾回收机制**

* 什么样的对象会被当做垃圾回收

当一个对象的引用(地址)没有变量去记录的时候,该对象就会成为垃圾对象,并在垃圾回收器空闲的时候对其进行清扫.

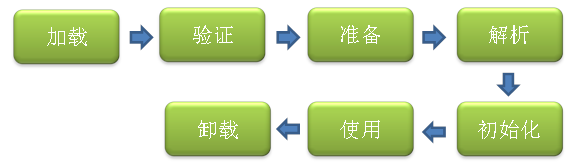
* 如何检验对象是否被回收

可以重写Object类中的finalize方法

这个方法在垃圾回收器执行的时候,被回收器自动调用执行的.

* 怎样通知垃圾回收器回收对象

可以调用system类的静态方法gc().通知垃圾回收器去清理垃圾.

1. **类加载过程**

* 加载 -> 验证 -> 准备 -> 解析 -> 初始化
* 加载:1.获取类的二进制字节流

2.将字节流代表的静态存储结构转化为方法区运行时数据结构

3.在队中生成class字节码对象

* 验证:连接过程的第一步,确保class文件的字节流中的信息符合当前虚拟机的要求,不会危害虚拟机的安全
* 准备:为类的静态变量分配内存并将其初始化为默认值
* 解析:虚拟机将常量池内符号引用替换成直接引用的过程
* 初始化:执行类构造器的init的过程

1. **对象创建过程**

* JVM会先去方法区下找有没有所创建对象的类存在，有就可以创建对象了，没有则把该类加载到方法区
* 在创建类的对象时，首先会先去堆内存中分配空间
* 当空间分配完后，加载对象中所有的非静态成员变量到该空间下
* 所有的非静态成员变量加载完成之后，对所有的非静态成员进行默认初始化
* 所有的非静态成员默认初始化完成之后，调用相应的构造方法到栈中
* 在栈中执行构造函数时，先执行隐式，再执行构造方法中书写的代码
* 执行顺序：静态代码库，构造代码块，构造方法
* 当整个构造方法全部执行完，此对象创建完成，并把堆内存中分配的空间地址赋给对象名（此时对象名就指向了该空间）

1. **方法区堆栈溢出怎么处理**

jdk1.7之前字符串常量池是方法区的一部分，方法区叫做“永久代”，在1.7之前无限的创建对象就会造成内存溢出

用jdk1.7之后，取消了永久代，添加了元数据区，就不会产生内存溢出。

在jdk1.7时出现内存溢出：

1. 需要查看代码中是否出现死循环。
2. 是否出现死锁现象。
3. 在jvm运行时，提高堆内存的大小。
4. **单例设计模式中懒汉式和饿汉式的区别**

* 饿汉式：

//饿汉式单例类.在类初始化时，已经自行实例化

public class Singleton1 {

private Singleton1() {}

private static final Singleton1 single = new Singleton1();

//静态工厂方法

public static Singleton1 getInstance() {

return single;

}

}

* 懒汉式：

//懒汉式单例类.在第一次调用的时候实例化自己

public class Singleton {

private Singleton() {}

private static Singleton single=null;

//静态工厂方法

public static Singleton getInstance() {

if (single == null) {

single = new Singleton();

}

return single;

}

}

* 饿汉式就是类一旦加载，就把单例初始化完成，保证getInstance()的时候，单例就已经存在。
* 懒汉式比较懒，只有当调用getInstance的时候，才会去初始化这个单例

区别：

* 饿汉式是线程安全的，懒汉式是线程不安全的（即一个进程内有多个线程在在同时使用时可能会产生多个实例，可创建个静态内部类，产生一个单例对象，通过静态内部类返回获取这个对象）

1. **常见的基本排序**

* 冒泡排序

public void bubbleSort(int[] arr) { //从小到大

int temp = 0;

for(int i = 0; i < arr.length -1; i++){ //控制趟数，到倒数第二个为止

for(int j = arr.length-1; j>i; j--){ //从最后一个值开始冒泡，将后面的小值与前面的大值进行交换，并且保证循环到前面已经排序完的索引为止

if(arr[j-1] > arr[j]){

temp = arr[j];

arr[j] = arr[j-1];

arr[j-1] = temp;

}

}

}

}

* 选择排序：

public void selectionSort(int[] arr){

int temp = 0;

int k = 0; //存储最小值的索引

for(int i = 0; i<arr.lengrh - 1; i++){ //控制趟数，到倒数第二个为止

k = i;

for(int j = i; j<arr.length;j++){ //将第一个数默认为最小值，将其索引赋值给k，从k索引开始，将后面每个数与k索引对应的值比较，如果值小了，就将其索引赋值给k

if(arr[j] < arr[k]){

k = j;

}

}

//遍历完后，k就指向了最小的值，将其与i对应的值交换(也可 以先做个判断，判断k的索引是否有变化，无变化可以不交换)

temp = arr[k];

arr[k] = arr[i];

arr[i] = temp;

}

}

1. **常见设计模式**

* 工厂模式：一个抽象接口的实现，多个抽象接口的实现类，spring的beanFactory就是工厂模式
* 单例模式：在内存中，保证对象的实例只有一个。
* 装饰者模式：对一个类进行装饰，增强其方法行为，如Java中的IO流就使用了装饰者模式
* 代理模式：比如sping AOP使用动态代理
* 适配器模式：io流，通过继承实现将一个接口适配到另一个接口，InputStreamReader类继承Reader接口，但要创建它们必须在构造函数中传入一个InputStream的实例，InputStreamReader的作用也就是将InputStream适配到Reader
* 状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

**二:WEB:**

1. **cookie和session的区别与联系**
2. cookie数据存放在客户的浏览器上,session数据存放在服务器上.
3. 很多浏览器限制站点最多保存20个cookie,单个cookie保存的数据不能超过4k.
4. cookie不是很安全,考虑安全应当使用session.
5. 可以考虑将登录信息等重要信息存放为session,其它信息如果需要保留,可以放在cookie中.
6. session会在一定时间内保存在服务器上.
7. session会在浏览器关闭或者一段时间内销毁,也可以通过setMaxInactiveInterval(int)方法进行设置,或是通过invalidate()方法强制结束当前会话.cookie可以通过setMaxAge(int)方法设置缓存在客户端的时间.
8. 一般情况下,session生成的sessionid都是保存在cookie中.

总结:

cookie:在客户端保存数据,不安全.只能保存字符串,且是少量数据.

session:在服务器端保存数据,安全.可以保存对象数据,数据无限制.

1. **如果客户端禁止 cookie 能实现 session 还能用吗？**

可以用，session 只是依赖 cookie 存储 sessionid，如果 cookie 被禁用了，可以使用 url 中添加 sessionid 的方式保证 session 能正常使用。

1. **Get方法和Post方法区别**

get:查询、请求参数 不安全,提交的数据量小

post:添加或更新数据、提交数据比较安全、提交的数据量无限制.

1. get方法用于信息获取,他是安全的,而post方法是用于修改服务器上资源的请求.
2. get请求的数据会附在url之后,而post方法提交的数据则放置在http报文实体的主体里,所以post方法的安全性比get方法要高.
3. get方法传输的数据量一般限制在2kb,post方法对于数据大小是无限制的.
4. **过滤器有哪些作用，以及过滤器的生命周期？**

生命周期：每个Filter在tomcat启动时进行初始化，每个Filter只有一个实例对象

* Init:在服务器启动时会创建Filter实例
* doFilto:这个方法会在用户每次访问“目标资源”时执行
* destroy():服务器关闭时销毁Filter对象

作用：

* 验证客户是否来自可信网络
* 对客户提交的数据进行重新编码
* 过滤掉客户的某些不应该出现的词汇
* 验证用户是否可以登录
* 验证客户的浏览器是否支持当前的应用
* 记录系统日志

1. **servlet的生命周期及常用方法**

* init()方法:在servlet的生命周期中,仅执行一次init()方法.
* service()方法:它是servlet的核心,每当客户请求一个httpservlet对象,该对象的service()方法就要调用,而且传递给这个方法一个”请求”对象和一个”响应”对象作为参数.
* destory()方法:仅执行一次,在服务器端停止且卸载servlet时执行该方法.

解决servlet线程安全

1. 继承SingleThreadModel,消耗服务器内存,降低性能.并且过时,不推荐.
2. 尽量避免使用全局变量,推荐.
3. 通过使用ThreadLocal.
4. **转发和重定向的区别**
5. 重定向是浏览器发送请求并收到响应以后再次向一个新地址发请求;转发是服务器收到请求后为了完成响应转到另一个资源.
6. 重定向中有两次请求对象,不共享数据;转发只产生一次请求对象且在组件间共享数据.
7. 重定向后地址栏地址改变,而转发不会.
8. 重定向的新地址可以是任意地址;转发必须是同一个应用内的某个资源.

获取servlet的转发和响应重定向的方式

* 转发的方法:

通过HttpServletRequest的getRequestDispatcher()方法获得

通过ServletContext的getRequestDispatcher()方法获得

* 重定向的方法:

HttpServletResponse的sendRedirect()方法.

1. **ajax书写方式及内部主要参数**

主要参数:

1. url:要求为String类型的参数,发送请求的地址.
2. data:要求为Object或String类型,发送到服务器的数据.
3. type:要求为String类型,请求方式get或post.
4. datatype:要求为String类型,预期服务器返回的类型.
5. timeout:要求为number类型,设置请求超时时间.
6. async:要求为boolean类型,异步为true(默认),同步为false.
7. cache:要求为boolean类型,默认为true,是否从浏览器缓存中加载信息.
8. beforesend:要求为Function类型的参数.例如添加自定义http头.

ajax的优缺点:

* 优点:减轻服务器的负担,按需取数据,最大程度的减少冗余请求,局部刷新页面,减少用户心理和实际的等待时间,带来更好的用户体验.
* 缺点:ajax大量的使用了JavaScript和ajax引擎,这些取决于浏览器的支持,在编写的时候考虑对浏览器的兼容性.ajax只是局部刷新,所以页面的后退按钮是没有用的.

1. **Jquery常用选择器**

Jquery选择器总共有四大类:

基本选择器、层级选择器、过滤选择器和表单选择器.

* 基本选择器是Jquery最常用的选择器,也是最简单的选择器,他通过元素id,class和标签名来查找dom元素.
* 层级选择器是通过dom元素间的层次关系来获取元素,主要层次关系包括父子、后代、相邻、兄弟关系.
* 过滤选择器主要是通过特定的过滤规则筛选出所需的dom元素,过滤规则与css中的伪类选择器语法相同,即选择器都以一个冒号开头.

1. Jquery基本过滤选择器
2. Jquery内容过滤选择器
3. Jquery可见性过滤选择器
4. Jquery属性过滤选择器
5. Jquery子元素过滤选择器
6. Jquery表单对象属性过滤选择器

* 表单选择器:我们可以极其方便的获取表单的某个或某类型的元素.

1. **JSP和Servlet的区别**

* 什么是JSP:Java Server Pages(Java服务器端页面)动态页面

它和servlet技术一样,都是SUN公司定义的一种用于开发动态web资源的技术.

jsp=html+java

* jap执行原理

.jsp(翻译)-->.java(编译)-->.class(执行)

servlet:服务器端的小应用程序.适合编写java逻辑代码.

jsp:适合编写输出动态内容,但不适合编写java逻辑.

1. **JSP常用的标签**

* 请求转发：<jsp:forward>
* 页面传递数据：<jsp:param>
* 输出标签：<c:out>
* 判读标签<c:if>
* 迭代标签<c:foreach>
* 多重判断标签<c:choose>

静态包含包含的是内容,动态包含包含的是结果.

静态包含不可以传递参数,而动态包含可以传递参数.

1. **jsp九大内置对象及作用**

* PageContext JSP的页面容器
* request 获取用户的请求信息
* response 服务器向客户端的回应信息
* session 用来保存每一个用户的信息
* application 表示所有用户的共享信息
* config 服务器配置信息，可以取得初始化参数
* out 页面输出
* page 但前页面对象，可以获取其他对象
* exception 异常对象

1. **JSP四大作用域及请求范围**

JSP四大作用域从小到大分别为:page,request,session,application.

* 第一个作用域是page,他只在当前页面有效,也就是用户的请求页面有效.
* 第二个作用域是request,他在当前第一次请求中有效.
* 第三个作用域是session,他在当前整个会话中有效.
* 第四个作用域是application,他在整个应用都有效.

PageContext:pageContext 存放的数据在当前页面有效.开发时使用较少.

ServletRequest:request 存放的数据在第一次请求(转发)中有效.使用非常多.

HttpSession:session 存放的数据在第一次会话中有效.使用的比较多.如存放登录信息,购物车

ServletContext:application 存放的数据在整个应用范围都有效.因为范围太大,应尽量少用.

1. **如何防止表单重复提交**

网络延迟时,重复点击提交按钮,有可能发生重复提交表单问题.

解决方案:

1. 数据库主键唯一.
2. 提交成功后重定向.
3. 使用JavaScript解决,使用标记位,提交后隐藏或不可用提交按钮.

使用session解决:

生成唯一的Token(uuid)给客户端,客户端第一次提交时带着这个Token,后台与session中的进行对比.

1. **常见的http返回状态码**

100:告诉客户端应继续发送请求

200:请求响应成功

202:请求已被受理还未做出响应

301：永久重定向

302：暂时重定向

400:请求无效,常见的情况是请求参数有误,http头构建错误

404:访问不到资源

500:服务器后端错误

1开头的状态码是消息类型的.

2开头的状态码表示成功.

3开头的状态码表示需要重定向.

4开头的状态码表示请求错误.

5开头的状态码表示服务器错误.

1. **TCP和UDP的区别,HTTP协议**

* TCP协议提供安全可靠的网络传输服务,它是一种面向连接的服务.类似于打电话,必须先拨号.双方建立一个传递信息的通道传输.
* UDP协议是一种数据报协议,它传输的数据是分组报文,它是无连接的,不需要和目标通信方建立连接,类似于写信,所以它的传输不保证安全可靠.但适合大数据量的传输.
* HTTP协议是超文本传输协议,是一种相对于TCP来说更细致的协议,TCP以及UDP协议规范的是网络设备之间的通信规范,HTTP实在TCP协议的基础上针对用户的协议,用户服务具体体现在应用程序之间的交互,比如javaweb中客户端服务端体系就要用http协议来规范通信.

TCP和UDP在开发中很少见到,但是网络底层都有他们的影子,正常的会话级别的服务:如客户端服务器体系底层就说基于TCP协议.而邮件发送,短信发送等底层使用的是UDP协议.

HTTP协议,客户端/服务器体系的程序都使用HTTP协议来规范通信.

1. **tcp为什么要三次握手，两次不行吗？为什么？**

如果采用两次握手，那么只要服务器发出确认数据包就会建立连接，但由于客户端此时并未响应服务器端的请求，那此时服务器端就会一直在等待客户端，这样服务器端就白白浪费了一定的资源。若采用三次握手，服务器端没有收到来自客户端的再此确认，则就会知道客户端并没有要求建立请求，就不会浪费服务器的资源。

1. **说一下 tcp 粘包是怎么产生的？**

tcp 粘包可能发生在发送端或者接收端，分别来看两端各种产生粘包的原因：

发送端粘包：发送端需要等缓冲区满才发送出去，造成粘包；  
 接收方粘包：接收方不及时接收缓冲区的包，造成多个包接收。

1. **json的数据格式**

* Json的最初出现是专门为JavaScript准备的,它是一种轻量级的数据交换格式.

1. 数据在名称/值对中;
2. 数据由逗号隔开;
3. 花括号保存对象

{”属性名”:”值”,”属性名”:”值”}

* Json对象数组:

[{”属性名”:”值”,”属性名”:”值”},{”属性名”:”值”,”属性名”:”值”}]

* 复杂格式:

{”属性名”:”值”,”属性名”:{”属性名”:”值”}}

json中的值是有限制的,对于简单类型来说,只能是字符串,数值(必须是十进制)、布尔值和null;对于复合类型来说,只能放数组或者对象,不能是正则、函数或者日期;

json在网络开发中有非常广泛的用途,但可以归纳为一句:可以用于接口开发及调用中使用的数据格式.一来用于服务端和JavaScript之间的数据交互,二来可以用于跨域传输数据的数据格式.

在项目中,前后端交互、接口开发中很多都使用json来作为数据传输格式.

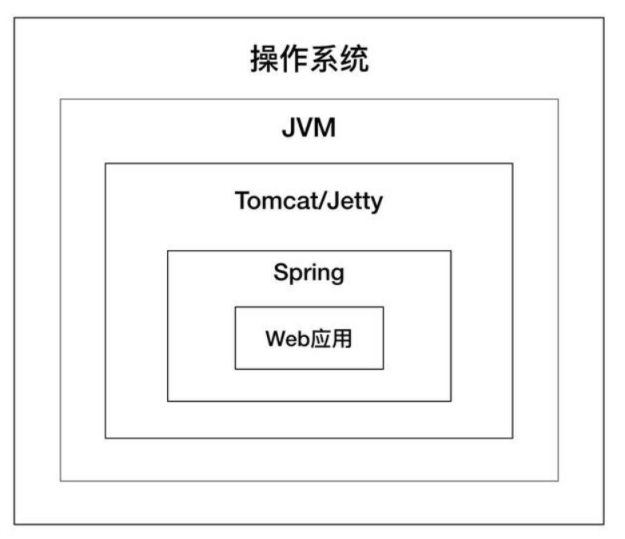
1. **如何实现跨域？**
   * 服务器端运行跨域 设置 CORS 等于 \*；
   * 在单个接口使用注解 @CrossOrigin 运行跨域；

* 使用 jsonp 跨域,JSONP 实现原理:

jsonp：JSON with Padding，它是利用script标签的 src 连接可以访问不同源的特性，加载远程返回的“JS 函数”来执行的。

1. **Tomcat体系结构讲解**

Tomcat 或者 Jetty 就是一个“HTTP 服务器 + Servlet 容器”，我们也叫它们 Web 容器。

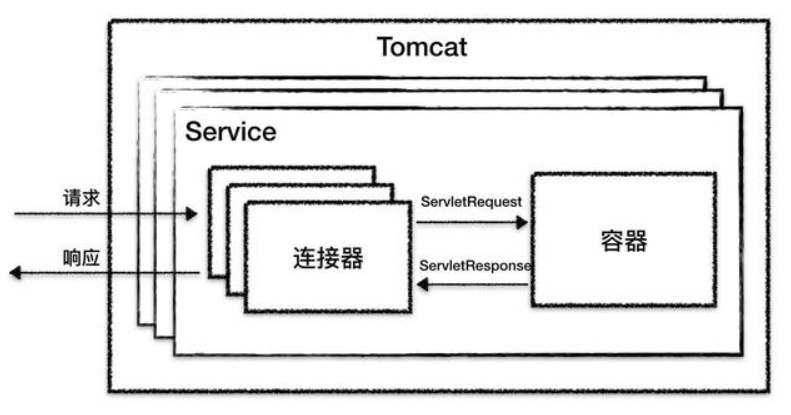


Web服务器需要实现两个核心功能：

* 处理 Socket 连接，负责网络字节流与 Request 和 Response 对象的转化。
* 加载和管理 Servlet，以及具体处理 Request 请求。

因此 Tomcat 设计了两个核心组件连接器（Connector）和容器（Container）来分别做这两件事情。连接器负责对外交流，容器负责内部处理。

可参考：http://www.sohu.com/a/321678611\_120176035



(1) Server  
 Server表示整个的Catalina Servlet容器。Tomcat提供了Server接口的一个默认实现，这通常不需要用户自己去实现。在Server容器中，可以包含一个或多个Service组件。  
(2) Service  
 Service是存活在Server内部的中间组件，它将一个或多个连接器（Connector）组件绑定到一个单独的引擎（Engine）上。在Server中，可以包含一个或多个Service组件。Service也很少由用户定制，Tomcat提供了Service接口的默认实现，而这种实现既简单又能满应用。  
(3) Connector  
 连接器（Connector）处理与客户端的通信，它负责接收客户请求，以及向客户返回响应结果。在Tomcat中，有多个连接器可以使用。  
(4) Engine  
 在Tomcat中，每个Service只能包含一个Servlet引擎（Engine）。引擎表示一个特定的Service的请求处理流水线。作为一个Service可以有多个连接器，引擎从连接器接收和处理所有的请求，将响应返回给适合的连接器，通过连接器传输给用户。用户允许通过实现Engine接口提供自定义的引擎，但通常不需要这么做。  
(5) Host  
 Host表示一个虚拟主机，一个引擎可以包含多个Host。用户通常不需要创建自定义的  
Host，因为Tomcat给出的Host接口的实现（类StandardHost）提供了重要的附加功能。  
(6) Context  
 一个Context表示了一个Web应用程序，运行在特定的虚拟主机中。什么是Web应用程序呢？在Sun公司发布的Java Servlet规范中，对Web应用程序做出了如下的定义：“一个Web应用程序是由一组Servlet、HTML页面、类，以及其他的资源组成的运行在Web服务器上的完整的应用程序。它可以在多个供应商提供的实现了Servlet规范的Web容器中运行”。一个Host可以包含多个Context（代表Web应用程序），每一个Context都有一个唯一的路径。用户 通 常 不 需 要 创 建 自 定 义 的 Context ， 因 为 Tomcat 给 出 的 Context 接 口 的 实 （ 类StandardContext）提供了重要的附加功能。凡是实现了Servlet规范的都可以成为Servlet容器

1. **tomcat 如何调优，涉及哪些参数**

硬件上选择，操作系统选择，版本选择，jdk选择，配置jvm参数，配置connector的线程数量，开启gzip压缩，trimSpaces，集群等  
可参考：[http://blog.csdn.net/lifetragedy/article/details/7708724](https://link.jianshu.com?t=http://blog.csdn.net/lifetragedy/article/details/7708724" \t "_blank)

作者：消失er  
链接：https://www.jianshu.com/p/97e210a6d95a  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**三:数据库:**

1. **事务的特性和隔离级别**

事务的特性：

* 原子性（Atomicity）

原子性指事务是不可分割的工作单位，事务中的操作要么都发生，要么都不发生

* 一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态 (转账总数不变)

* 隔离性（Isolation）

事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。(在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始)

* 持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。

隔离级别：

* 未提交读read uncommitted

会发生 脏读、不可重复读、虚读

* 已提交读read committed //Oracle SQL Server（系统事务）

解决脏读，但是不可重复读和虚读有可能发生

* 重复读repeatable read //Mysql

解决脏读和不可重复读，但是虚读有可能发生.

* 串行化serializable

避免脏读，不可重复读，虚读的发生

* 脏读：脏读是读到了别的事务回滚前的脏数据。比如事务B执行过程中修改了数据X，在未提交前，事务A读取了X，而事务B却回滚了，这样事务A就形成了脏读。
* 不可重复读：事务A首先读取了一条数据，然后执行逻辑的时候，事务B将这条数据改变了，然后事务A再次读取的时候，发现数据不匹配了，就是所谓的不可重复读了
* 幻读：事务A首先根据条件索引得到N条数据，然后事务B改变了这N条数据之外的M条或者增添了M条符合事务A搜索条件的数据，导致事务A再次搜索发现有N+M条数据了，就产生了幻读。

1. **Mysql的体系结构**

* 连接层
* 最上层是一些客户端和链接服务，包含本地sock 通信和大多数基于客户端/服务端工具实现的类似于 TCP/IP的通 信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证 它所具有的操作权限。
* 服务层
* 第二层架构主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化，部分内置函数的执行。所有跨存储引擎的功能也在这一层实现，如 过程、函数等。在该层，服务器会解析查询并创建相应的内部 解析树，并对其完成相应的优化如确定表的查询的顺序，是否利用索引等， 最后生成相应的执行操作。如果是 select语句，服务器还会查询内部的缓存，如果缓存空间足够大，这样在解决大量读操作的环境中能够很好的提升系统的性能。
* 引擎层
* 存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API和存储引擎进行通信。不同的存储引擎具有不同的功能，这样我们可以根据自己的需要，来选取合适的存储引擎。
* 存储层
* 数据存储层，主要是将数据存储在文件系统之上，并完成与存储引擎的交互。和其他数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎上，插件式的存储引擎架构，将查询处理和其他的系统任务以及数据的存储提取分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。

1. **InnoDB，MyISAM存储引擎特性**

* InnoDB
* InnoDB存储引擎是Mysql的默认存储引擎。InnoDB存储引擎提供了具有提交、回滚、崩溃恢复能力的事务安全。 但是对比MyISAM的存储引擎，InnoDB写的处理效率差一些，并且会占用更多的磁盘空间以保留数据和索引。
* MyISAM
* MyISAM 不支持事务、也不支持外键，其优势是访问的速度快，对事务的完整性没有要求或者以SELECT、INSERT 为主的应用基本上都可以使用这个引擎来创建表 。

1. **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。

1. **delete、drop、truncate区别**

* truncate 和 delete只删除数据，不删除表结构 ,drop删除表结构，并且释放所占的空间。
* 删除数据的速度，drop> truncate > delete
* delete属于DML语言，需要事务管理，commit之后才能生效。drop和truncate属于DDL语言，操作立刻生效，不可回滚。
* 使用场合：
* 当你不再需要该表时， 用 drop;
* 当你仍要保留该表，但要删除所有记录时， 用 truncate;
* 当你要删除部分记录时（always with a where clause), 用 delete.

1. **mysql 中 in 和 exists 区别**

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

如果查询的两个表大小相当，那么用in和exists差别不大。  
 如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in：  
not in 和not exists如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。  
 1.EXISTS只返回TRUE或FALSE，不会返回UNKNOWN。

2.IN当遇到包含NULL的情况，那么就会返回UNKNOWN。

1. **数据库的三范式是什么？**

* 第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项。

家庭信息(家庭人口,户籍)

第二范式：要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）产品数量、产品折扣、产品价格与“订单号”和“产品号”都相关，但是订单金额和订单时间仅与“订单号”相关，与“产品号”无关，

这样就不满足第二范式的要求，调整如下，需分成两个表

第三范式：任何非主属性不依赖于其它非主属性。数据表中的每一列数据都和主键直接相关，而不能间接相关。“班主任性别”和“班主任年龄”直接依赖的是“班主任姓名”，

而不是主键“学号”，

1. **Sql优化**

* 通过慢查询日志去寻找，哪些sql执行效率低
* 使用explain分析低效率的sql执行计划
* 针对低效率的sql执行计划分析

没有索引

索引失效

数据量太大

没有索引：

* 针对查询的列创建索引，提高查询效率，但是索引太多，mysql也会出现选择困难，所以建立索引要有效，无效的索引需要删除。

索引失效：

* 尽量选择较小的列
* 将where中用的比较频繁的字段建立索引
* select子句中避免使用‘\*’
* 避免在索引列上使用计算，not，in和<>等操作
* 当只需要一行数据的时候使用limit 1
* 保证表单数据不超过200w，适时分割表
* 针对查询较慢的语句，可以使用explain来分析该语句具体的执行情况
* 避免查询时判断null，否则可能会导致全表扫描，无法使用索引;
* 避免like查询，否则可能导致全表扫描，可以考虑使用全文索引
* 能用union all的时候就不用union，union过滤重复数据要耗费更多的CPU资源

数据量太大：

* 分页查询优化
* 在索引上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容。
* Select \* from table t where t.id in（select id from table order by id limit 100000，10） ;
* 对于主键自增的表，可以把limit查询缓存某个位置的查询
* Select \* from table where id > 100000 limit 10;

分库分表：

* 使用MyCat中间件实现。

全文索引技术：

* ElasticSearch ， solr

非关系型数据库：

* 不需要像关系型数据库一样维护表于表之间的关系，而是使用json这种灵活多变的形式，效率比Mysql提高很多

1. **分库分表方案**

* 一开始上来就是32个库，每个库32个表，1024张表这个分法，

基本上国内的互联网肯定都是够用了

无论是并发支撑还是数据量支撑都没问题

* 如果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。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都够了。
* 此方案最多可以扩展到32个数据库服务器，每个数据库服务器是一个库。如果还是不够？最多可以扩展到1024个数据库服务器，每个数据库服务器上面一个库一个表。因为最多是1024个表么。

服务器升级流程：

* 设定好几台数据库服务器，每台服务器上几个库，每个库多少个表，推荐是32库 \* 32表，对于大部分公司来说，可能几年都够了
* 路由的规则，orderId 模 32 = 库，orderId / 32 模 32 = 表
* 扩容的时候，申请增加更多的数据库服务器，装好mysql，倍数扩容，4台服务器，扩到8台服务器，16台服务器
* 由dba负责将原先数据库服务器的库，迁移到新的数据库服务器上去，很多工具，库迁移，比较便捷
* 我们这边就是修改一下配置，调整迁移的库所在数据库服务器的地址
* 重新发布系统，上线，原先的路由规则变都不用变，直接可以基于2倍的数据库服务器的资源，继续进行线上系统的提供服务

1. **MySQL 索引是怎么实现的？**

索引是满足某种特定查找算法的数据结构，而这些数据结构会以某种方式指向数据，从而实现高效查找数据。

具体来说 MySQL 中的索引，不同的数据引擎实现有所不同，但目前主流的数据库引擎的索引都是 B+ 树实现的，B+ 树的搜索效率，可以到达二分法的性能，找到数据区域之后就找到了完整的数据结构了，所有索引的性能也是更好的。

1. **Mysql支持的索引类型**

* index  ----  普通索引,数据可以重复，没有任何限制。
* unique   ---- 唯一索引,要求索引列的值必须唯一，但允许有空值；如果是组合索引，那么列值的组合必须唯一。
* primary key ---- 主键索引,是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值，一般是在创建表的同时创建主键索引。
* 组合索引 ----  在多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。
* fulltext ---- 全文索引,是对于大表的文本域：char，varchar，text列才能创建全文索引，主要用于查找文本中的关键字，并不是直接与索引中的值进行比较。

1. **索引失效问题**

7种引起索引失效的情境

* 如果条件中有or，即使其中有部分条件带索引也不会使用(这也是为什么尽量少用or的原因)，例子中user\_id无索引。注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引
* 对于复合索引，如果不使用前列，后续列也将无法使用，类电话簿。
* like查询是以%开头
* 存在索引列的数据类型隐形转换，则用不上索引，比如列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引
* where 子句里对索引列上有数学运算，用不上索引
* where 子句里对有索引列使用函数，用不上索引
* 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

1. **左连接 ，右连接，内连接和全外连接的4者区别**

* left join （左连接）：返回包括左表中的所有记录和右表中连接字段相等的记录。
* right join （右连接）：返回包括右表中的所有记录和左表中连接字段相等的记录。
* inner join （等值连接或者叫内连接）：只返回两个表中连接字段相等的行。
* full join （全外连接）：返回左右表中所有的记录和左右表中连接字段相等的记录。

1. **一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 MySQL 数据库，又插入了一条数据，此时 id 是几？**

* 表类型如果是 MyISAM ，那 id 就是 8。
* 表类型如果是 InnoDB，那 id 就是 6。  
  InnoDB 表只会把自增主键的最大 id 记录在内存中，所以重启之后会导致最大 id 丢失。

1. **说一下 MySQL 的行锁和表锁？**

MyISAM 只支持表锁，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。

1. **说一下乐观锁和悲观锁？**

乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交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。

悲观锁：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止，直到这个锁被释放。

数据库的乐观锁需要自己实现，在表里面添加一个 version 字段，每次修改成功值加 1，这样每次修改的时候先对比一下，自己拥有的 version 和数据库现在的 version 是否一致，如果不一致就不修改，这样就实现了乐观锁。

1. **高并发下，如何做到安全的修改同一行数据。**

使用悲观锁 悲观锁本质是当前只有一个线程执行操作，结束了唤醒其他线程进行处理。也可以缓存队列中锁定主键。

1. **数据库会死锁吗，举一个死锁的例子**

产生死锁的原因主要是：

（1）系统资源不足。  
 （2） 进程运行推进的顺序不合适。  
 （3）资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。  
 （2） 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。  
 （3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。  
 （4） 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。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

1. **聚集索引和非聚集索引的区别**

聚簇索引就是索引和记录紧密在一起。

聚簇索引:叶子节点就是对应的数据节点,逻辑顺序与磁盘上行的物理存储顺序相同

非聚集索引:叶节点仍然是索引节点,指针指向对应的数据块,二次查询,逻辑顺序与磁盘上行的物理存储顺序不同

聚集索引的叶节点就是最终的数据节点，而非聚集索引的叶节仍然是索引节点，但它有一个指向最终数据的指针。  
 非聚簇索引 索引文件和数据文件分开存放，索引文件的叶子页只保存了主键值，要定位记录还要去查找相应的数据块。

**四:框架:**

1. **Spring**

* spring是一个轻量级的,开源的,模块化的,一站式业务层框架,它能够整合其他主流框架
* spring的实质就是实现了工厂模式的工厂类,在配置文件中,通过添加<bean>标签,来创建实例对象
* spring的核心分别是Ioc和AOP

IoC是控制反转,即使将常见实力对象的权力交给spring框架去管理,通过依赖注入的方式去,去注入给调用者,解除bean与bean之间的耦合性

依赖注入的三种方式分别是:接口注入,构造注入,setter方法注入(<property name="" value或ref="">

AOP是指面向切面编程,就是在不修改代码的前体下,对程序进行加强,例如加入日志,权限判断,异常处理等等

AOP的底层使用的是代理技术,分别为jdkProxy动态代理和CGLIB动态代理,jdk动态代理是需要被代理的类实现接口的,而CGLIB则不需要被代理的类实现接口

* 关于动态代理,它的底层实现是利用java反射来实现.

Proxy主要使用 Proxy.newProxyInstance方法

CGLIB主要使用 Enhancer.create方法

MethodHandler主要使用 MethodHandles.lookup().findVirtual().bindTO()方法

静态代理:是实现已经编写好,在程序运行前代理类的.class文件就已经存在

动态代理:在程序运行时运用反射机制动态创建而成

静态代理虽然执行效率要比动态代理高,但需要实现编写好代理类的java文件,工程量大,由于代理类和委托类实现了相同接口,会出现大量重复代码,而且后期不易于维护;

而动态代理则是随着程序的运行而创建代理对象,比较灵活,但是动态代理的底层是反射机制,比较消耗资源

1. **spring 有哪些主要模块？**

* spring core：框架的最基础部分，提供 ioc 和依赖注入特性。
* spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框架式的对象访问方法。
* spring dao：Data Access Object 提供了JDBC的抽象层。
* spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
* spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。
* spring Web mvc：spring 中的 mvc 封装包提供了 Web 应用的 Model-View-Controller（MVC）的实现。

1. **Spring AOP**

Spring AOP的面向切面编程,是面向对象编程的一种补充,用于处理系统中分布的各个模块的横切关注点,比如事务管理、日志、缓存等.它是使用动态代理实现的,在内存中临时为增强某个方法生成一个AOP对象,这个对象包含目标对象的所有方法,在特定的切入点做了增强处理,并回调原来的方法.

Spring AOP的动态代理主要有两种方式实现,JDK动态代理和cglib动态代理.JDK动态代理通过反射来接受被代理的类,但是被代理的类必须实现接口,核心是InvacationHandler和Proxy类.cglib动态代理的类一般是没有实现接口的类,cglib是一个代码生成的类库,可以在运行时动态生成某个类的子类.所以,cglib是通过继承的方式做的动态代理,因此如果某个类被标记为final,那么它是无法使用cglib做动态代理的.

AOP能做什么:

* 降低模块的耦合度.
* 使系统容易扩展
* 避免修改业务代码,避免引入重复代码,更好的代码复用.

AOP怎么用:

* 前置通知:某方法调用前发出通知.
* 后置通知:某方法完成之后发出通知.
* 返回后通知:方法正常返回后,调用通知.在方法,正常退出发出通知.
* 异常通知:抛出异常后通知:在方法抛出异常退出时执行的通知.在方法调用时,异常退出发出通知.
* 环绕通知:通知包裹在被通知的方法的周围.

1. **IOC和DI**

* IOC(控制反转).将对在自身对象中的一个内置对象的控制反转,反转后不再由自己本身的对象进行控制这个内置对象的创建,而是由第三方系统去控制这个内置对象的创建.简单来说就是把本来在类内部控制的对象,反转到类外部进行创建后注入,不再由类本身进行控制,这就是IOC本质.
* DI(依赖注入).自身对象中的内置对象是通过注入的方式进行创建.
* IOC和DI的关系.

ioc就是容器,di就是注入这一行为,那么di确实就是ioc的具体功能的实现.而ioc则是di发挥的平台和空间.所以说,ioc和di即是相辅相成的搭档.最重要的是,他们都是为了实现解耦而服务的.

* DI是如何实现的.

依赖注入可以通过setter方法注入、构造注入、接口注入三种方法来实现,Spring支持setter注入和构造器注入,通常使用构造器注入来注入必须的依赖关系,对于可选的依赖关系,则setter注入是更好的选择,setter注入需要类提供无参构造器或者无参的静态工厂方法来创建对象.

1. **Spring中Bean的作用域**

Spring IOC容器在根据配置创建一个Bean对象实例时,可以为Bean指定实例的作用范围.

* singleton(单例模式)

IOC容器仅创建一个Bean实例,IOC容器每次返回的是同一个Bean实例.

* prototype(原型模式)

IOC容器仅创建多个Bean实例,IOC容器每次返回的是一个新的实例.

* request(HTTP请求)

该属性仅对HTTP请求产生作用,每次HTTP请求都会创建一个新的Bean,适用于WebApplicationContext.

* session(会话)

该属性仅用于HTTP Session,同一个session共享一个Bean实例.不同session使用不同的实例.

* global-session(全局会话,在spring5.x中已移除)

该属性仅用于HTTP Session,同session作用域不同的时候,所有session共享一个Bean实例.

Bean的生命周期

在实际开发中,我们一般常用的就是单例模式和原型模式

单例模式生命周期和容器相同.

原型模式生命周期,是每次使用时创建新的对象,用完等待垃圾回收器回收.

1. **Spring框架实现实例化和依赖注入的方式**

* 实例化:

1. 构造器实例化Bean
2. 静态工厂方式实例化Bean
3. 实例工厂方式实例化Bean

* 依赖注入:

1. 基于构造函数的注入
2. 基于set方法的注入
3. 基于自动装配的注入
4. 基于注解的依赖注入
5. **springmvc执行流程**

* 用户发送请求至前端控制器DispatcherServlet
* DispatcherServlet收到请求调用HandlerMapping处理器映射器。
* 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
* DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
* HandlerAdapter执行处理器(handler，也叫后端控制器)。
* Controller执行完成返回ModelAndView
* HandlerAdapter将handler执行结果ModelAndView返回给DispatcherServlet
* DispatcherServlet将ModelAndView传给ViewReslover视图解析器
* ViewReslover解析后返回具体View对象
* DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
* DispatcherServlet响应用户

1. **Spring以及SpringMVC常用注解**

* @Component:用于标记在一个类上,表示当前类是spring的一个组件,是ioc的一个容器.他有三个衍生注解:@Controller、@Service、@Repository
* @Controller：用于标记在一个类上,代表这个类是控制层组件.
* @Service:用于标记在一个类上,代表这个类是业务层组件.
* @Repository:用于标记在一个类上,代表这个类是数据访问层组件.
* @RequestMapping:是一个用于处理请求地址映射的注解,可用于类或方法上.用于类上,表示类中所有响应请求的方法都是以该地址作为父路径
* @RequestParam:用于将指定的请求参数赋给方法中的形参.
* @PathVariable:可以获取URL中的动态参数.
* @RequestBody:用于读取request请求的body部分数据.
* @ResponseBody:用于将controller方法返回的对象,用流响应给客户端.
* @RestController:@Controller+@ResponseBody,用于标记在一个类上.
* @Transactional:写在类上用于指定当前类中的方法支持事务,写在方法上表示当前的方法支持事务

1. **springmvc获取表单的几种方式**

* 借助原始ServletAPI的HttpServletRequest对象.

Controller的方法,

添加HttpServletRequest类型入参,

通过HttpServletRequest.getParameter()获取请求数据

* 借助控制器方法的形参

controller的方法,添加参数来接收表单的数据.

接收的类型包括:

1. 基本类型或String
2. 实体类类型
3. 实体类关联对象

* Controller方法,接收实体类类型参数,而表单提交的是json数据时,可以使用@RequestBody注解获取全部请求体,配合jackson开源组件可以实现转换成实体类.要求json数据的key必须和实体类属性保持一致.

1. **Spring MVC的异常处理 ？**

在 Spring MVC的Web应用程序中，可以存在多个实现了HandlerExceptionResolver的异常处理类，他们的执行顺序，由其order属性决定, order值越小，越是优先执行, 在执行到第一个返回不是null的ModelAndView 的Resolver时，不再执行后续的尚未执行的Resolver的异常处理方法。

1. **为什么要用 spring boot？**

配置简单  
独立运行  
自动装配  
无代码生成和 xml 配置  
提供应用监控  
易上手  
提升开发效率

1. **Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？**

1. 配置变更

2. JDK 版本升级

3. 第三方类库升级

4. 响应式 Spring 编程支持

5. HTTP/2 支持

6. 配置属性绑定

1. **spring boot 有哪些方式可以实现热部署？**

使用 devtools 启动热部署，添加 devtools 库，在配置文件中把 spring. devtools. restart. enabled 设置为 true；  
 使用 Intellij Idea 编辑器，勾上自动编译或手动重新编译。

1. **SpringBoot 的常用注解有哪些？**

@SpringBootApplication:

包含@Configuration、@EnableAutoConfiguration、@ComponentScan通常用在主类上；

@ComponentScan:组件扫描。个人理解相当于，如果扫描到有@Component @Controller @Service等这些注解的类，则把这些类注册为bean\*；  
@Configuration:指出该类是 Bean 配置的信息源，相当于XML中的，一般加在主类上；

@Bean:相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理；

@Import：用来导入其他配置类。

@ImportResource：用来加载xml配置文件。

@ControllerAdvice：包含@Component。可以被扫描到。统一处理异常。

@ExceptionHandler（Exception.class）：用在方法上面表示遇到这个异常就执行以下方法。

@SpringBootApplication： 包含了@ComponentScan、@Configuration和@EnableAutoConfiguration注解。其中@ComponentScan让spring Boot扫描到Configuration类并把它加入到程序上下文。

@Configuration 等同于spring的XML配置文件；使用Java代码可以检查类型安全。

@EnableAutoConfiguration 自动配置。

@ComponentScan 组件扫描，可自动发现和装配一些Bean。

@Component可配合CommandLineRunner使用，在程序启动后执行一些基础任务。

@RestController注解是@Controller和@ResponseBody的合集,表示这是个控制器bean,并且是将函数的返回值直 接填入HTTP响应体中,是REST风格的控制器。

@Autowired自动导入。

@PathVariable获取参数。

@JsonBackReference解决嵌套外链问题。

@RepositoryRestResourcepublic配合spring-boot-starter-data-rest使用。

1. **SpringBoot 有哪几种读取配置的方式？**

方式一： 通过@Value("${spring.datasource.url}")这样的方式读取  
 方式二：通过@ConfigurationProperties(prefix = "spring.datasource")这种写法系统会依据prefix前缀自动注入配置数据到数据实体变量,这种方式不错,但是存在缺陷,我们编写的PropertyPlaceholderConfigurer扩展字段会无效,所以如果只是单纯的读取配置而不需要额外操作时可使用这方式最简单

方式三： 我们可以直接注入Environment对象示例并读取properties对象属性environment.getProperty("spring.datasource.database");与方式一的本质差不多,我们不需要编写对应字段的模型对象,但是对于程序可阅读性不好友,复用率不高

方式四： 通过系统启动时候初始化Listener,使用PropertiesLoaderUtils工具类读取指定配置文件并获得Properties配置对象,我们可以随时随地使用该对象的属性,这种方式比较少用,针对比较自定义的配置数据可使用该方式

1. **SpringBoot 配置加载顺序？**

在不指定要被加载文件时，默认的加载顺序：由里向外加载，所以最外层的最后被加载，会覆盖里层的属性，加载顺序依次为：

* 位于与jar包同级目录下的config文件夹，
* 位于与jar包同级目录下
* idea 环境下，resource文件夹下的config文件夹
* idea 环境下，resource文件夹下  （1->4, 外->里）

1. **Spring Boot 如何定义多套不同环境配置**

一、Spring Boot 环境设置机制

spring.profiles.active 属性可以为我们指定当前设置的环境，以此来选择我们的配置文件。例如我们有配置文件

application.yml

application-dev.yml

application-test.yml

application-prod.yml

当执行 java -jar xxx.jar --spring.profiles.actvie=test 此时，系统将启用 application.yml 和 application-test.yml配置文件。

当执行 java -jar xxx.jar --spring.profiles.actvie=prod 此时，系统将启用 application.yml 和 application-prod.yml 配置文件。

二、配置多环境

正如 第一 点所述，我们配置不同的配置文件

application.yml

application-dev.yml（开发环境）

application-test.yml（测试环境）

application-uat.yml（预发布环境）

application-prod.yml（生产环境）

三、指定环境

1 在 cmd 命令中指定

java -jar xxx.jar --spring.profiles.actvie=dev

2 在 application.yml 中指定

spring: profiles: active: dev

1. **Mybatis中使用#和$书写占位符有什么区别**

#{}传参能防止sql注入.

${}传参是字符串拼接.

1. **动态SQL**

* 所谓SQL的动态和静态,是指SQL语句在何时被编译和执行,二者都是用在SQL嵌入式编程中的.
* SQL语句的主体结构,在编译时尚无法确定,只有等到程序运行起来,在执行的过程中才能确定,这种SQL叫做动态SQL.
* 静态SQL语句的编译是在应用程序运行前进行的,编译的结果会存储在数据库内部.
* 程序运行时,数据库将直接执行编译好的SQL语句,降低运行时的开销.
* MyBatis中用于实现动态SQL的元素主要有:if、where、foreach.

1. **Mapper动态代理规范**

* xml映射文件中的namespace与mapper接口的全类名相同.
* mapper接口方法名和xml映射文件中定义的每个statement的id相同.
* mapper接口方法的输入参数类型和xml映射文件中定义的每个sql的parameterType的类型相同.
* mapper接口方法的输出参数类型和xml映射文件中定义的每个sql的resultType的类型相同.
* mybatis中的mapper动态代理是不支持方法重载的dao接口里的方法,因为是全类名+方法名的保存和寻找策略.
* mapper接口的工作原理是JDK动态代理,mybatis运行时就是用JDK动态代理为mapper接口生成代理proxy对象,代理对象proxy会拦截接口方法,转而执行mappedStatement所代表的sql,然后将sql执行结果返回.

1. **Mybatis常用注解**

@Insert ： 插入sql , 和xml insert sql语法完全一样

@Select ： 查询sql, 和xml select sql语法完全一样

@Update ： 更新sql, 和xml update sql语法完全一样

@Delete ： 删除sql, 和xml delete sql语法完全一样

@Param ： 当映射器方法需要多个参数时，指定参数的名字

@Results ： 结果集合

@Result ： 结果

@One ：复杂类型的单独属性值映射

@Many ：复杂类型的己合属性映射

1. **springcloud如何实现服务的注册和发现**

服务在发布时 指定对应的服务名（服务名包括了IP地址和端口） 将服务注册到注册中心（eureka或者zookeeper）

这一过程是springcloud自动实现 只需要在main方法添加@EnableDisscoveryClient  同一个服务修改端口就可以启动多个实例

调用方法：传递服务名称通过注册中心获取所有的可用实例 通过负载均衡策略调用（ribbon和feign）对应的服务

1. **ribbon和feign区别**

Ribbon添加maven依赖 spring-starter-ribbon 使用@RibbonClient(value="服务名称") 使用RestTemplate调用远程服务对应的方法

feign添加maven依赖 spring-starter-feign 服务提供方提供对外接口 调用方使用 在接口上使用@FeignClient("指定服务名")

Ribbon和Feign的区别：Ribbon和Feign都是用于调用其他服务的，不过方式不同。

1.启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

2.服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

3.调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

Feign则是在Ribbon的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

1. **springcloud断路器的作用（**Hystrix**）**

当一个服务调用另一个服务由于网络原因或者自身原因出现问题时 调用者就会等待被调用者的响应 当更多的服务请求到这些资源时

导致更多的请求等待，这样就会发生连锁效应（雪崩效应）断路器就是解决这一问题

断路器有：

  完全打开：一定时间内达到一定的次数无法调用，并且多次检测没有恢复的迹象，断路器完全打开，那么下次请求就不会请求到该服务

  半开：短时间内有恢复迹象断路器会将部分请求发给该服务，当能正常调用时断路器关闭

  关闭：当服务一直处于正常状态能正常调用断路器关闭

**项目技术点:**

**五:MQ：**

**1.为什么使用MQ?**

* 使用MQ的好处:解耦，异步，削峰平谷;
* 解耦：
* 当A系统生产关键数据,而且B,C,D系统需要A系统给它们发送数据,来进行下一步操作,此时A系统和BCD系统产生了严重的耦合,所有的操作和维护都要在A系统中进行,如果将A系统产生的数据放到MQ当中,让BCD系统需要的时候去消费,此时就解放了A系统,不用考虑调用成功,失败超时等情况,同时ABCD系统独立运行,后续新添加系统需要A系统的数据,也不需要去修改A系统的代码,达到了解耦的效果
* 异步：
* 一般互联网类企业,对用户的直接操作,一般要求每个请求在200ms以内完成。对于一个系统调用多个系统,在不适用mq的情况下,它执行完返回的耗时,是执行完所有系统所需时间的总和;使用mq进行优化后,执行的耗时,则是执行主系统的耗时,以及加上主系统发送数据到消息队列的耗时,大幅度提升高延时接口的性能,提升了用户体验.
* 削峰平谷：
* 一般MySQL的每秒请求最高在2000左右，用户访问量高峰期的时候涌入的大量请求,很可能将MySQL给打死,然后系统就挂掉,但是高峰期过了,请求量可能远远低于2000,所以这种情况去增加服务器就不值得,如果使用mq的情况,将用户的请求全部放到mq中,然后让系统去消费用户的请求,不要超过系统所能承受的最大请求数量,保证系统不会再高峰期挂掉,但此时可能有几十万或几百万请求积压在mq中,但是高峰期一过,系统还是按照最大请求数量进行处理请求,很快就能将积压请求处理完

* 使用MQ的缺陷:
* 系统可用性降低:以前只要担心系统的问题,现在还要考虑mq如果挂掉的问题,因为mq一旦挂掉,所关联的系统,就会统统挂掉
* 系统复杂性变高:考虑的问题变多,要考虑消息丢失,消息重复消费的情况
* 一致性问题:比如A系统调用BCD系统,BCD同时成功才能执行成功,返回数据,现在BC执行成功,D发生异常的情况,并没有执行成功,但是A给用户返回的是成功

**2.你了解哪些MQ技术，ActiveMQ,RabbitMQ,RocketMQ,kafka**

* ActiveMQ它可以支持万级的吞吐量,它是一个比较成熟完善的中间件,但是它存在有少量信息丢失的情况,而且目前官方对他的更新迭代不是很即时,社区的活跃度不是很高
* RabbitMQ是一款用Erlang语言开发的消息中间件,它延时低,唯一一款达到微妙级延时的消息中间件,而且社区活跃度高,对于bug问题的修复很及时,而且提供了很友善的后台界面,唯一的劣势是就是我们搞java开发的很少有人能够阅读它的源码,对于问题的修复仅仅依靠社区。
* RocketMQ是阿里旗下的一款品质优秀的MQ,它可以达到十万级的吞吐量,而且它还是支持分布式事务,应用于分布式架构,它的维护基本上都是靠阿里,如果哪天阿里宣布放弃这个项目的维护,除非你们公司有能力继续来维护RocketMQ的使用。
* Kafka也是一款分布式的中间件,最大优点就是其吞吐量高,一般运用于大数据系统的实时运算和日志采集的场景,功能简单,可靠性高,扩展性高,唯一的缺点是可能导致重复消费,但是这点轻微的影响在大数据场景下可以忽略。

**3.RabbitMQ 的使用场景有哪些？**

抢购活动，削峰填谷，防止系统崩塌。

延迟信息处理，比如 10 分钟之后给下单未付款的用户发送邮件提醒。

解耦系统，对于新增的功能可以单独写模块扩展，比如用户确认评价之后，新增了给用户返积分的功能，这个时候不用在业务代码里添加新增积分的功能，只需要把新增积分的接口订阅确认评价的消息队列即可，后面再添加任何功能只需要订阅对应的消息队列即可。

**4.如何保证MQ的高可用?**

* 非分布式的MQ的高可用:
* 普通集群模式:实际queue保存在一个节点上,其他节点保存该queue的元数据,消费者连接某个节点,该节点去调用实际存放queue的节点,返回数据,说白了只是简单的提高了吞吐量而已.

缺陷在于:在集群内部产生大量的数据传输,高可用性没有,queue所在的节点宕机,服务就没法继续使用

* 镜像集群模式:每个节点上都保存了实际的queue的全部数据,消费者无论连接哪个节点,都能直接获取到数据.任何一个节点宕机,其他节点还存在queue的全部数据,消费者换个节点继续消费,这是高可用的模式
* 分布式MQ的高可用:
* 假设生产者往topic里写了三条数据,每条数据放在一个partition中,每个partition部署在一台机器中,每个partition都有一个或多个replica副本,其中有一个是leader,其他的是follower,只有leader能够向外提供数据的读写,生产者写入消息时,leader负责向follower同步消息,此时高可用架构就已经体现,如果一台leader宕机,会从它的follower中选举一个leader,继续向外提供服务

**5.如何保证消息不被重复消费(如何保证消息消费时的幂等性)?**

* 重复消费:
* 每个中间件它能够保证的是消息不丢失,但不能保证消息不被重复发送，所以接收方就要做幂等性的判断，防止消息重复消费，例如：支付金额的消息被消费了两次，那么所得到的钱数就是错误的，这种情况在系统中肯定是不允许存在的。
* 消费的幂等性（同样的操作，在一段时间内，只执行一次）:
* 如果是往数据库里写入数据,就根据主键查一下,如果数据已经存在,就update
* 可以在把数据先存在set或者redis中,消费前,先去里面查看,数据是否已存在,已存在就丢弃这数据,比如说我们的订单系统,生成订单也会用到mq,此时订单id就是全局唯一的id,在写入数据库之前,就可以先把数据去redis中查询,如果redis中已经存在,则不进行消费操作,如果redis中不存在的话,就存在redis中,然后进行下一步操作.
* 在数据库中设置唯一约束,就不会导致重复数据的多次插入

**6.如何保证消息不丢失**

* rabbitMQ:
* 消息丢失的情况:生产者写的消息在到中间件的网络传输过程中就丢了,或者是消息到了中间件,但是内部出错,消息没保存下来
* 中间件将消息保存下来,还没等到消费者消费完,就自己挂掉,导致消息丢失
* 消费者取到消息还没来得及消费就自己挂掉了,因为rabbitMQ消费者开启了autoAck,在消费数据还没成功时,就已经向中间件发送完成的信息,此时消费者挂掉,就会消息丢失
* 解决方案:
* 生产者消息丢失,可以通过开启事务功能,如果消息没有发送成功,发生异常就回滚,然后重试发送消息,发送成功就提交事务,这个的缺陷就是阻塞式的,降低吞吐量,耗费性能;如果是rabbitMQ可以开启confirm模式,它能给每次写的消息都分配一个唯一的id,如果写入到rabbitMQ中,rabbitMQ就会回传一个ack消息,如果没有就会会挑一个nack接口,告诉你消息接收失败,你可以重试,confrim机制是异步的,效率会高很多
* 关于中间件的数据丢失,可以开启中间件的持久化,将消息持久化磁盘中,中间件挂了恢复之后自动读取之前存储的数据.
* 消费者数据丢失,关闭rabbitMQ的autoACK机制,自己手动提交完成信息

**7.如何保证消息的顺序性**

* rabbitMQ为多个消费者开辟多个queue队列（先进先出）,将保证操作顺序的消息发布到同一个队列中去，操作这个队列的消费者会一个一个消息去处理，因为队列这种结构是先进先出的类型，所以保证的数据的顺序性。

**8.消息大量积压怎么解决？**

* 临时启动多个消息者，并发处理消息
* 临时启动多个消息者，接受消息之后，不处理。暂时把消息写到文件中。消息中间件中的消息处理的完了。关闭临时消费者。单独写个离线程序，处理文件中的消息
* 临时启动多个消息者，接受消息之后，直接丢弃。 可以让生产者的源头恢复数据

**9.RabbitMQ 节点的类型有哪些？**

* 磁盘节点：消息会存储到磁盘。
* 内存节点：消息都存储在内存中，重启服务器消息丢失，性能高于磁盘类型。

**10.rabbitmq如何确认消息一定发送到了消息中间件中呢?**

消息发送到server是先通过交换机,再到消息队列,只要数据到队列,那么此消息肯定是发送成功的.

第一步确定消息发送到了交换机.

使用发送方确认机制来判断消息是否发送成功.

第二步确认交换机把消息路由到消息队列.

使用失败回调来判断消息是否发送成功.

只有两步都成功,此消息才是发送成功的.

**11.rabbitmq的集群**

参考网站:docker中搭建rabbitmq集群-> https://www.cnblogs.com/vipstone/p/9362388.html

集群分为普通集群和镜像集群.

普通集群: 普通集群,他会把所有节点的交换机信息和队列的元数据分为两种(队列数据分为两种: 一种为队列里面的消息,另一种是队列本身的信息, 后者被称为元数据.)进行复制,确保所有的节点都有一份.

镜像集群: 在普通集群的基础上,把所有的队列数据完全同步(对性能有一定的影响)当对数据可靠性要求高时,可以使用镜像模式.

镜像集群实现由两种方式: 一种是直接在管理台控制,

一种是在声明队列的时候控制.

配置集群还设计到节点信息, 有内存节点和磁盘节点,如果对队列有修改的情况下,必须有磁盘节点,用来保存信息,内存节点断电后,信息就消失,无法保存. 默认就是磁盘节点, 设置 --ram为内存节点.

# **六:Redis：**

1. **使用redis缓存的好处**

* 提高并发量，不需要每次都访问数据库，提高并发量。
* 提高性能，redis运行在内存中，内存的执行效率，远远超过数据库。

1. **Redis的key和value的存储大小有限制吗？**

* Redis的key和value的存储大小都是有限制的，都是512M。

1. **redis存储什么数据类型**
2. 字符串（String）
3. 字符串列表（lists）
4. 字符串集合（sets）
5. 有序字符串集合（sorted sets）
6. 哈希（hash）
7. **使用redis缓存的弊端**

* 缓存穿透，大量的请求访问,查询的是数据库中不存在的数据,就会在缓存中无法命中,直接去访问数据库,数据库中也查询不到,自然无法将结果写入缓存,下一秒又有大量请求查询不存在的数据,导致数据库最终挂掉(要么系统出bug,要么黑客恶意攻击)

解决方案:

* 在数据库中没有查询到的数据,存储一个特定值到缓存中,这样下次恶意访问就可以去缓存中查询到数据,不会访问到数据库.注意的是这些特定的key,需要设置过期时间，避免黑客攻击时，大量的无效key把redis存满。
* 缓存雪崩，大规模的key失效，大量的请求通过key访问到数据库，导致数据库直接崩溃,然后系统直接瘫痪.

解决方案:

* 事前:保证redis集群的高可用性,redis cluster,主从机制;
* 事中:hystrix（熔断器）限流+降级,避免数据库被打死.高并发的访问量走hystrix限流组件,让访问量保持在数据库最大负载范围内,保证数据库不被打死,系统可以继续使用.剩余的请求走降级组件,返回一些默认的值或者是友情提示等
* 事后:redis持久化机制,尽快回复缓存集群,一旦重启,自动从磁盘上加载数据,回复内存中的数据
* 缓存与数据库不一致，修改数据时，数据库和缓存数据不一致。

最初级的缓存不一致:

出现场景：

* 先修改数据库,再删除缓存,如果修改缓存失败,就会导致数据库中式新数据,缓存中是旧数据,数据不一致

解决思路:

* 先删除缓存,再修改数据库,如果删除缓存成功,修改数据库失败,那么数据库中是旧数据,缓存中是空的,读的时候缓存中没有,去数据库中读旧数据,然后更新到缓存中去

高并发场景下数据库与缓存的数据不一致:

出现场景：

* 读写并发请求,导致数据库和缓存中的数据不一致,在写请求删除缓存,修改数据库库存还未成功时,查询库存的请求就发来,先去缓存中查询,发现是空,然后去数据库中查,然后将结果放入缓存中,然后修改库存的操作成功,导致数据库中是新数据,缓存中是旧数据

解决思路:

* 相同的商品id,进行哈希取值,再加上对内存队列的数量进行取模,每个商品都可以路由到某一个内存队列中,然后将去请求和写请求串行化,这样就可保证一定不会出现不一致的情况,但是会导致系统的吞吐量会大幅降低
* 缓存并发竞争，同一时间，多个线程来执行，操作同一个key。

解决思路:

* 方式一：分布式锁,（zookeeper分布式锁），确保同一时间,只能由一个系统实例在操作某个key,别的实例不被允许读和写
* 方式二：乐观锁 ，每次要写之前,先判断这个value的时间戳是否比缓存里的更新,如果新就允许写.

1. **为什么redis是单线程的但是还可以支撑高并发?**

* redis线程模型:redis里有个处理器叫做文件事件处理器(file event handler),这个文件事件处理器是单线程的执行的,redis所以是单线程的模型。

执行流程:

* redis启动初始化时,redis会将连接应答器和AE\_READABLE事件关联
* 客户端跟redis连接,会在serverSocket产生一个AE\_READABLE事件,然后由IO多路复用程序监听到事件,将serverSocket压入队列,然后连接应答处理器来处理事件,和客户端建立连接,创建对应的socket,同时将这个socket的AE\_READABLE事件跟命令请求处理器关联起来
* 客户端向redis发起请求时,首先会在socket中产生一个AE\_READABLE事件,然后被监听,被压入队列,然后由对应的命令请求处理器来处理
* redis准备好给客户端响应数据之后,就会将socket的AE\_WRITABLE命令跟回复处理器关联起来,当客户端准备好读取响应数据时,就会在socket上产生一个AE\_WRITABLE事件,会由对应的命令回复处理器来处理
* 命令回复处理器处理完之后,就会删除这个socket的AE\_WRITABLE事件和命令回复处理器的关联关系

1. **Redis为什么效率高?**

* 核心是基于非阻塞的IO多路复用机制,由监听程序轮询等待的事件,然后压入队列,可以达到一个线程同时处理多个io请求的目的
* 纯内存操作
* 单线程避免了多线程的频繁切换问题

1. **Redis由哪些数据类型,分别在哪些场景下使用?**

* String,存储基本的类型
* List,有序列表,比如之前项目中的广告缓存,用的就是list集合进行缓存的,它可以用于比如存储粉丝列表,文章的评论列表等等
* Set,无序列表,它最主要的特点就是去重,比如微博上查询两个明星的共同粉丝,就可以用set进行去重
* SortSet,它主要是在set的基础上加上了排序功能
* Hash,类似map的集合,一般可以用来存储对象,我们系统中的购物车对象就是以hash类型存储在redis中,key就是用户的id,value就是购物车对象

1. **Redis过期策略**

* 定期删除配合惰性删除
* 定期删除,指的是redis默认每个100ms就抽取一些设置了过期事件的key,检查是否过期,如果过期就删除.如果太多的key都设置了过期时间,每隔几百毫秒,就全部检查删除,cpu的负载会很高.实际上redis是每隔100ms随机抽取一些key来检查和删除.
* 惰性删除,就是当你获取key时,redis就会检查一下,如果key过期了,就删除,不会返回,如果没过期就返回
* 因为redis的过期策略会导致你设置了过期key,过期了很多过期key没有被删除,还堆积在内存中,此时就要设置内存淘汰机制来解决了
* 内存淘汰机制
* noevication:当内存不足以容纳写入新数据时,新写入操作就会报错
* allkeys-lru:当内存不足以容纳新写入数据时,在键空间中,移除最近最少使用的key
* allkeys-random:在键空间中,随机移除某个key
* volatile-lru:在设置了过期时间的键空间中,移除最近最少使用的key
* volatile-random:在设置了过期时间的键空间中,随机移除某个key
* volatile-ttl:在设置了过期时间的键空间中,由更早过期时间的key优先移除

1. **怎样做redis支撑高并发(读多写少用缓存,读少写多用队列)**

* 搭建redis集群,做成主从架构,一主多从,通过redis replication机制实现读写分离,主负责写,并且将数据同步复制到其他slave（从）节点上,从节点负责读,所有的读请求走从节点.好处,可支持水平扩容的读高并发架构
* redis replication的核心机制
* redis采用异步方式复制数据都slave节点上,slave节点会周期性的确认自己每次复制的数据量
* 一个master（主）节点可以配置多了slave节点
* slave节点也可以连接其他slave节点
* slave节点做复制的时候,也不会阻塞对自己的查询操作,它会用旧数据向外提供服务;复制完后,它就删除旧数据,加载新数据,这个时候向外停止服务(毫秒级别)
* slave节点主要用来横向扩容,做读写分离,提高吞吐量
* Master（主节点）持久化对于主从架构的安全保障的意义
* 如果仅仅用slave节点作为master节点的数据热备份,当master节点宕机后重启,数据清空,本地没有数据可以恢复,它会任为自己的数据就是空的,然后同步到slave节点上,就会导致数据百分百丢失
* 其次,如果开启了哨兵机制,在master节点挂掉之后,会迅速推选出一个slave节点去做master节点,但是当哨兵还没检测到master节点挂掉,master节点就自动重启了,还是会出现上面的情况

* redis主从架构的核心原理
* 当启动一个slave节点时,会发送一个PSYNC命令给master节点
* 如果这个slave节点第一次连接master节点,就会出发full resynchronization
* 开始full resynchronization的时候,master会开启一个后台线程,开始生成一份rdb快照文件,同时还还会继续接收客户端的写命令,存在内存中.RDB文件生成完毕后,master会将这个RDB发送给slave,slave先将RDB写入本地自盘,然后再加载到内存.然后master会将内存中缓存的写命令发送给slave,slave同步这些数据
* redis的高可用
* 不可用:虽然主从架构,读写分离可以保证缓存应对高并发场景,但是如果master节点挂掉,整个缓存架构就会挂掉,大量的流量涌入数据库,超过数据库的最大承载压力,数据库挂掉,整个系统就会挂掉不可用
* 高可用:故障转移,主备切换,当master故障时,自动检测(通过sentinal node 哨兵),将某个slave自动切换成master的过程交主备切换.这个机制实现了redis主从架构下的高可用性

1. **redis的持久化**

* 如果没有持久化,遇到灾难性故障时,就会丢失所有数据，如果开启持久化到磁盘,定期备份到云服务上,就能保证遇到灾难性故障,就不会丢失全部数据
* RDB
* 当redis需要做持久化时,redis会fork一个子进程,将数据写到磁盘写上一个临时RDB文件中,当子进程完成写临时文件后,将原来的RDB换掉
* 优势:

适合做冷备份,在最坏的情况下,恢复数据要比AOF快

对redis对外提供读写服务,影响服务非常小

RDB数据快照文件更少

* AOF
* 可以做到更精细的持久化,redis每执行一个修改数据的命令,都会将它添加到os cache中,一般会每个一秒执行一次fsync操作,保证将os cache中的数据写入磁盘中AOF日志文件中,AOF日志文件以append-only模式写入,文件不容易破损,在AOF日志过大时,就会出现后台重写
* 优势:

AOF可以更好的保护数据不丢失,丢失的数据更少

AOF日志文件以append-only模式写入,没有磁盘寻址的开销,写入性能非常高

AOF日志过大时,也不会影响客户端的读写

* RDB和AOF到底如何抉择
* 综合使用两种持久化机制,用AOF保证数据不丢失,作为数据恢复的第一选择,用RDB来做不同程度的冷备份,在AOF文件丢失或者损坏不可用的情况下,用RDB来进行快速的数据回复

1. **redis怎么设置缓存大小**

打开redis配置文件

示例：maxmemory 100mb

单位：mb,gb。

默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一点要设置。

设置maxmemory之后，配合的要设置缓存数据回收策略, 可以通过设置LRU算法来删除部分key，释放空间(LRU是Least Recently Used 近期最少使用算法。)。

就可以修改缓存大小为16gb.

1. **redis集群主从数据如何同步**

答1:

 Redis主从同步有两种方式：全同步和部分同步。主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。

答2:

1. 全量同步：  
   Redis全量复制一般发生在S-ave初始化阶段，这时S-ave需要将Master上的所有数据都复制一份。具体步骤如下：  
   　　1）从服务器连接主服务器，发送SYNC命令；  
   　　2）主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；  
   　　3）主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；  
   　　4）从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；  
   　　5）主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；  
   　　6）从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；  
   (2)增量同步：  
   Redis增量复制是指S-ave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。  
   增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令  
   (3)Redis主从同步策略：  
   主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，s-ave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。
2. **Redis缓存穿透和redis雪崩处理**

**(1)缓存穿透**

​ 一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，应该去后端系统查询（比如数据库）。如果key对应的value不存在，并且对key并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

**解决方案(晦涩)**  
对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。还有最常见的则是采用布隆过滤器，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，从而避免了对底层存储系统的查询压力。  
也可以采用一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

**(2)缓存雪崩**

​ 当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统（比如数据库）带来很大的压力。

​ 缓存雪崩是指在我们设置缓存是采用了相同的过期时间，导致缓存在某一时期同时失效，请求全部转发到DB，DB瞬间压力过大雪崩。

**​ 解决方案：**

​ 把缓存时间分散开，比如在原来的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。

****七:Elasticsearch****

****1.Elasticsearch是如何实现Master选举的****

* 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。

****2.在并发情况下，Elasticsearch如果保证读写一致？****

* 可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；
* 另外对于写操作，一致性级别支持quorum/one/all，默认为quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。
* 对于读操作，可以设置replication为sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置replication为async时，也可以通过设置搜索请求参数\_preference为primary来查询主分片，确保文档是最新版本。

****3.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；

****4.Elasticsearch更新和删除文档的过程****

删除和更新也都是写操作，但是Elasticsearch中的文档是不可变的，因此不能被删除或者改动以展示其变更；  
　　磁盘上的每个段都有一个相应的.del文件。当删除请求发送后，文档并没有真的被删除，而是在.del文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del文件中被标记为删除的文档将不会被写入新段。  
　　在新的文档被创建时，Elasticsearch会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

****5.Elasticsearch搜索的过程****

搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；  
　　在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。 每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。PS：在搜索的时候是会查询Filesystem Cache的，但是有部分数据还在Memory Buffer，所以搜索是近实时的。  
　　每个分片返回各自优先队列中 所有文档的 ID 和排序值 给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。  
　　接下来就是 取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。  
　　补充：Query Then Fetch的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch增加了一个预查询的处理，询问Term和Document frequency，这个评分更准确，但是性能会变差。

****6.ElasticSearch中的集群、节点、索引、文档、类型是什么？****

　　群集是一个或多个节点（服务器）的集合，它们共同保存您的整个数据，并提供跨所有节点的联合索引和搜索功能。群集由唯一名称标识，默认情况下为“elasticsearch”。此名称很重要，因为如果节点设置为按名称加入群集，则该节点只能是群集的一部分。

　　节点是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。

　　索引就像关系数据库中的“数据库”。它有一个定义多种类型的映射。索引是逻辑名称空间，映射到一个或多个主分片，并且可以有零个或多个副本分片。 MySQL =>数据库ElasticSearch =>索引

　　文档类似于关系数据库中的一行。不同之处在于索引中的每个文档可以具有不同的结构（字段），但是对于通用字段应该具有相同的数据类型。 MySQL => Databases =>Tables => Columns / Rows ElasticSearch => Indices => Types =>具有属性的文档

类型是索引的逻辑类别/分区，其语义完全取决于用户。

****7.elasticsearch 索引数据多了怎么办，如何调优****

使用bulk API  
 初次索引的时候，把 replica 设置为 0  
 增大 threadpool.index.queue\_size  
 增大 indices.memory.index\_buffer\_size  
 增大 index.translog.flush\_threshold\_ops  
 增大 index.translog.sync\_interval  
 增大 index.engine.robin.refresh\_interval

**八:Dubbo和Zookeeper：**

1. **dubbo的工作流程**

* provider向注册中心去注册自己为一个服务
* consumer去注册中心订阅服务,注册中心会通知consumer注册好的服务,consumer会将provider的地址等信息拉取到本地缓存
* consumer去调用provider
* consumer和provider都异步的通知监控中心

1. **Dubbo的通信原理？**

* Dubbo底层使用 hessain2进行二进制序列化进行远程调用
* Dubbo底层使用 netty框架进行异步通信。NIO

1. **dubbo负载均衡策略有哪些?**

* random loadbalance
* 默认情况下,dubbo是random loadbalance随机调用实现负载均衡,可以对provider不同实例设置不同的权重,会按照权重来负载均衡,权重大分配的流量高
* roundrobin loadbalance
* 默认就是均匀地将流量达到各个机器上,值得注意的是,要根据机器的性能,调整权重。
* leastactive loadbalance
* dubbo自动感知,如果某个机器性能越差,那么接收的请求就越少,给不活跃的机器更少的请求
* consistanthash loadbalance
* 一致性hash算法,相同参数的请求一定分发到同一个provider上,provider挂掉的时候,会基于虚拟机节点均匀分配甚于的流量,抖动不会太大.适用于订单,同一个订单的操作,分配到同一个机器上,这样就可以避免高并发场景下,数据库和缓存中数据不一致的情况

**4.zookeeper注册中心挂了，dubbo可以继续使用吗？**

可以,因为刚开始dubbo初始化的时候,消费者会将提供者的地址等信息缓存到dubbo,注册中心挂了dubbo可以继续通信

**5.zookeeper 都有哪些功能？**

* 集群管理：监控节点存活状态、运行请求等。
* 主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 zookeeper 可以协助完成这个过程。
* 分布式锁：zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。zookeeper可以对分布式锁进行控制。
* 命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

**6.Zookeeper的watch机制（通知机制）**

Zookeeper系统中其实类似于window系统中的文件夹，zookeeper可以建立普通节点和临时节点，但是每个节点只能有一个，例如：/a/b/c节点已经存在，那么再来申请/a/b/c的节点就会失败。此时可以设置watch此节点，当/a/b/c节点删除后，会通知watch此节点的所有线程，线程再来创建节点，完成设定的任务。（分布式锁，高可用性）

**7.zookeeper 怎么保证主从节点的状态同步？**

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，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。

**8.什么是 zab 协议**

ZAB 是 Zookeeper 原子广播协议的简称

整个ZAB协议主要包括消息广播和崩溃恢复两个过程，进一步可以分为三个阶段，分别是：

发现 Discovery  
 同步 Synchronization  
 广播 Broadcast

组成ZAB协议的每一个分布式进程，都会循环执行这三个阶段，将这样一个循环称为一个主进程周期。

**9.如果zookeeper服务挂了怎么办？**

注册中心对等集群，任意一台宕掉后，会自动切换到另一台

注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯

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

**九:分布式系统：**

1. **怎么实现远程通信**

什么是远程通信:简单来说，就是一个系统去调用另一个系统中的数据.

常见的有三种方式：

(1)Webservice的方式：

    1)优点：跨语言跨平台

    2)缺点：它是基于soap协议的，使用http+xml的方式进行数据传输，http是应用层协议，传输效率不是很高，而且xml的解析也比 较费时，所以项目内部进行通信的时候，不建议使用Websservice的方式

(2)restful形式的服务：

   1)优点：restful本身就是http，使用的是http+json的方式进行数据传输，因为json数据本身是非常简洁的，所以它比webservice的 传输效率更高；手机app端一般都使用该方法，其他很多项目也是用这种方式

    2)缺点：如果服务太多的话，会出现服务之间调用关系混乱，此时就需要治理服务

(3)Dubbo：

     使用的是RPC协议进行远程调用，RPC协议是一个二进制协议，直接使用的socket进行通信，传输效率高，并且可以统计出系统 之间的调用关系和调用次数系统分布式SAO系统的内部通信推荐使用dubbo

1. **分布式事务：**

TCC方案(Try,Confirm,Cancel)

* Try:此阶段是对各个服务的资源做检测以及对资源进行锁定或者预留
* Confirm:此阶段是在各个服务中执行实际的操作
* Cancel:如果任何一个服务的业务执行出错,就要进行补偿,将已经执行成功的业务逻辑回滚操作
* 这种方案的事务回滚实际上是依赖于自己写代码来回滚和补偿,补偿代码巨大,除了一些严格保证分布式事务的场景,比如和资金,订单相关的业务,尽量少使用

本地消息表(ebay)

* A系统在自己本地一个事务里操作同时,插入一条数据到消息表
* A系统将这个消息发送到MQ中
* B系统接收到消息之后,在一个事务里,往自己本地消息表中插入一条数据,同时执行其他业务操作,如果这个消息已经被处理了,那么此时这个消息会回滚,这样保证不会重复处理消息
* B系统执行成功后,就会更新本地消息表的状态以及A系统消息表的状态
* 如果B系统处理失败了,就不会更新消息表状态,A系统会定时扫描自己的消息表,没有处理的消息,会再次发送到MQ中,让B处理
* 这个方案为了保证最终一致性,哪怕B事务失败了,A会不断重发消息,直到B成功为止;此外最大的问题是严重依赖数据库的消息表来管理事务,不适用高并发的场景,扩展性能差

可靠消息最终一致性方案

* 这个方案是基于阿里的rocketMQ(3.3.6之前)来实现,是目前比较国内比较常用的方案
* A系统发送一个prepared消息到mq,如果prepared消息发送失败,就直接取消操作
* 如果消息发送成功,就接着执行本地事务,如果成功就给mq发送确认消息,如果失败就告诉mq回滚消息
* 如果发送了确认消息,B系统就会接收到,然后执行本地的事务
* mq会定时轮询所有prepared消息,回调接口,所有没发送确认消息的,问你继续重试还是回滚,然后A系统就去查看本地事务的状态,如果回滚,告诉mq回滚消息,如果成功了就重新发送确认消息
* 如果B系统的事务失败,自动不断重试,直到成功,如果实在不行就让B系统本地回滚,然后想办法通知A系统也回滚,或者是发送警报由人工手动回滚和补偿

最大努力通知方案

* A系统本地事务执行完后,发送消息到MQ
* 有个服务会消费MQ中的消息,然后写入数据库记录下来,或者放入内存队列中,接着调用B系统的接口
* 要是B系统执行成功就ok,执行失败,就最大努力通知服务定时尝试重新调用B系统,反复N次,实在不行就放弃

1. **系统的高并发问题是怎么解决的**

并发问题高，这个问题的解决方案是一个系统性的，系统的每一层面都需要做优化：

1） 数据层

a) 集群

b) 分表分库

c) 开启索引

d) 开启缓存

e) 表设计优化

f) Sql语句优化

g) 缓存服务器（提高查询效率，减轻数据库压力）

h) 搜索服务器（提高查询效率，减轻数据库压力）

2）项目层

a) 采用面向服务分布式架构（分担服务器压力，提高并发能力）

b) 采用并发访问较高的详情系统采用静态页面

c) 使用页面缓存

d) 用 ActiveMQ使得业务进一步进行解耦，提高业务处理能力

e) 使用分布式文件系统存储海量文件

3） 应用层

a) Nginx服务器来做负载均衡

b) Lvs做二层负载

1. **分布式锁的问题**

针对分布式锁的实现，目前比较常用的有以下几种方案：

1.基于数据库实现分布式锁

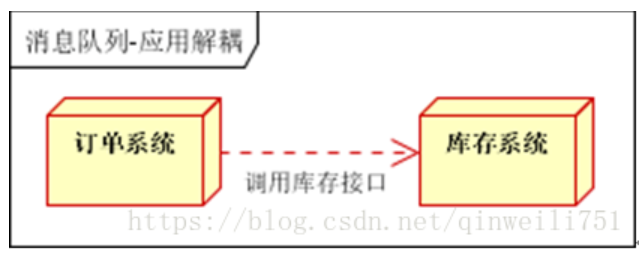
2.基于缓存（redis，memcached，tair）实现分布式锁

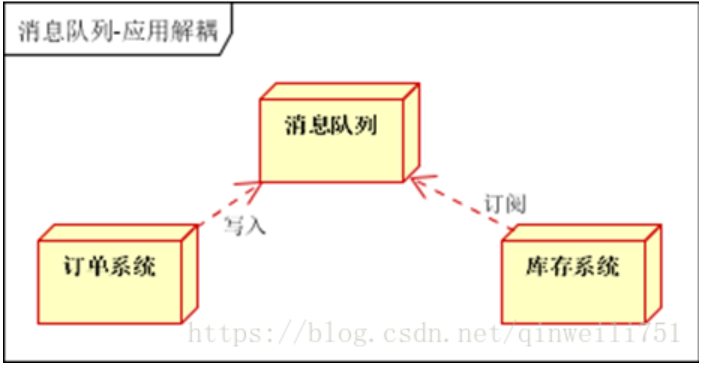
3.基于 zookeeper实现分布式锁

**电话面试:**

国睿

1. **Mq 干嘛的**
2. 时间 用户注册后，需要发注册邮件和注册短信。传统的做法有两种1.串行的方式；2.并行方式。
3. 应用解耦





在下单时库存系统不能正常使用。也不影响正常下单，

实现订单系统与库存系统的应用解耦。

3流量肖峰

可以控制活动的人数.

可以缓解短时间内高流量压垮应用；

1. **Cas登录**

1）.当客户端进行访问,filter将拦截所有的请求,首先从cookie当中获取token信息值（通过登录保存的前缀key获取）,然后通过token到redis查询是否有过用户的登录信息，如果查询数据（user为空），则携带当前的请求     地址跳转到统一的登录页面。

   2).当在统一登录页面进行登录时,将登录信息（user）查询数据库当前信息是否正确,

          如果不正确，重新跳转到登录页面,

          如果登录信息正确,基于用户信息生成一个token,key:雪花算法生成唯一token，value:为当前登录的用户信息保存在redis当中，并将此token保存在cookie当中，key：为任意前缀，value：            为token信息，任何重定向到一开始的请求地址.

采用jwt+rsa对称式加密算法来生成一个令牌，保存在浏览器，浏览器下次可以通过携带cookie来，我们用公钥对其解密，如果能够解出其中的信息，没那么证明这个令牌是正确的，这个人已经登陆。

1. **Redis使用场景**

热点数据缓存 搜索 点赞、好友等相互关系的存储set是可以自动排重 延时操作

排行榜 SortedSet (value: openid用户名信息 和score:点赞数)

主播分类 主播 礼物 主播

在奶茶活动中，我们需要展示各个部门的点赞排行榜， 所以我针对每个部门做了一个SortedSet,然后以用户的openid作为上面的username,以用户的点赞数作为上面的score, 然后针对每个用户做一个hash,通过zrangebyscore就可以按照点赞数获取排行榜，然后再根据username获取用户的hash信息，这个当时在实际运用中性能体验也蛮不错的。

,弹幕

游客购物车

**石山码农第一季**

因为java面试题是一个较为大的体系，所以我们会拆分为几季来讲解，预计是会推出3季，本次是第一季，第一季课程包含的内容如下：

# 1、分布式系统

## 1.1 为什么要进行系统拆分？

（1）为什么要进行系统拆分？如何进行系统拆分？拆分后不用dubbo可以吗？dubbo和thrift有什么区别呢？

## 1.2 分布式服务框架

（1）说一下的dubbo的工作原理？注册中心挂了可以继续通信吗？

（2）dubbo支持哪些序列化协议？说一下hessian的数据结构？PB知道吗？为什么PB的效率是最高的？

（3）dubbo负载均衡策略和高可用策略都有哪些？动态代理策略呢？

（4）dubbo的spi思想是什么？

（5）如何基于dubbo进行服务治理、服务降级、失败重试以及超时重试？

（6）分布式服务接口的幂等性如何设计（比如不能重复扣款）？

（7）分布式服务接口请求的顺序性如何保证？

（8）如何自己设计一个类似dubbo的rpc框架？

## 1.3 分布式锁

（1）使用redis如何设计分布式锁？使用zk来设计分布式锁可以吗？这两种分布式锁的实现方式哪种效率比较高？

## 1.4 分布式事务

（1）分布式事务了解吗？你们如何解决分布式事务问题的？TCC如果出现网络连不通怎么办？XA的一致性如何保证？

## 1.5 分布式会话

（1）集群部署时的分布式session如何实现？

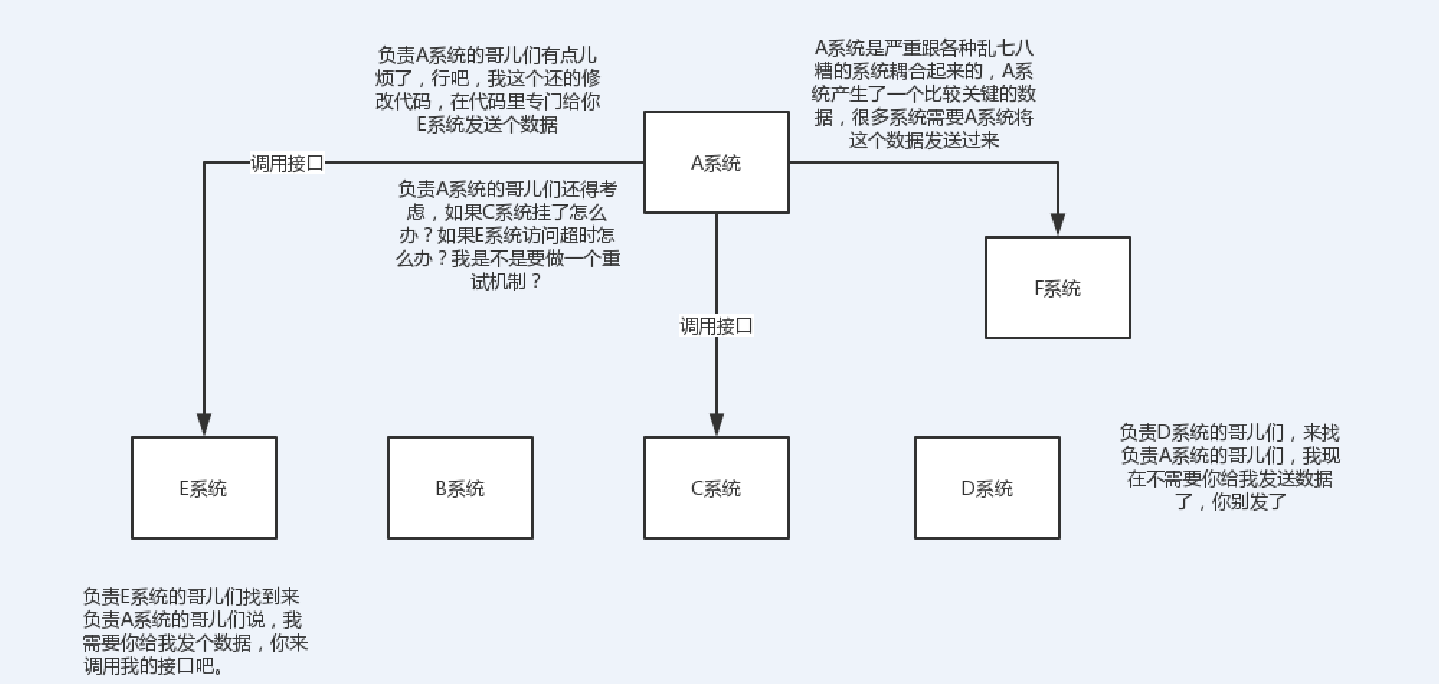
# 2、高并发架构

## 2.1 如何设计一个高并发系统？

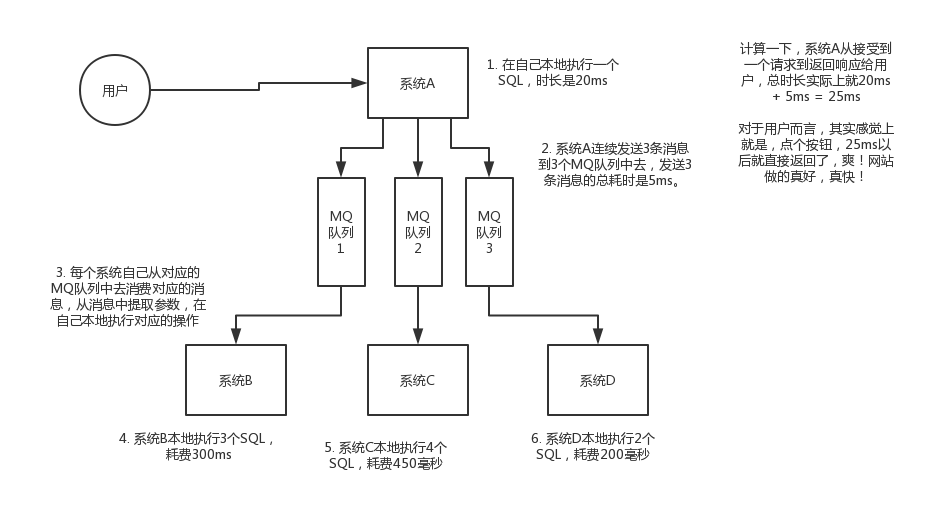
## 2.2 消息队列

### （1）为什么使用消息队列啊？消息队列有什么优点和缺点啊？kafka、activemq、rabbitmq、rocketmq都有什么优点和缺点啊？

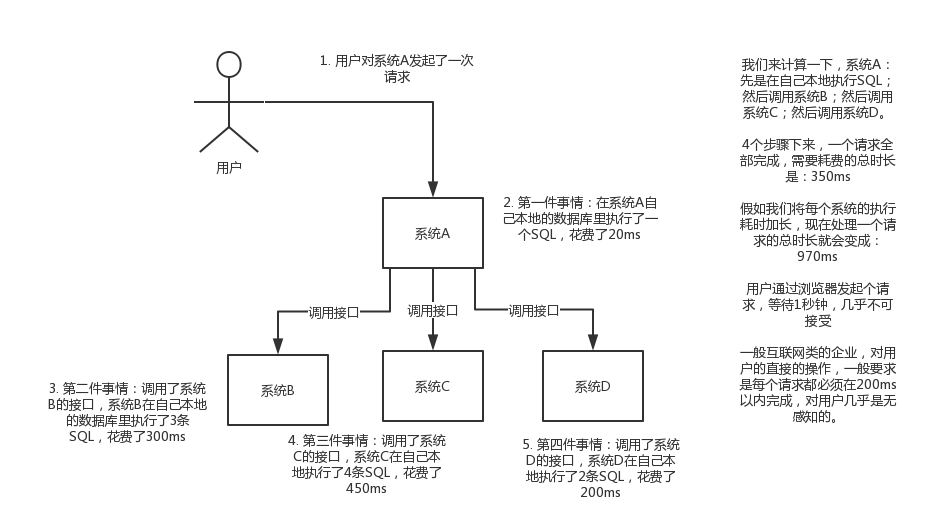
05-01



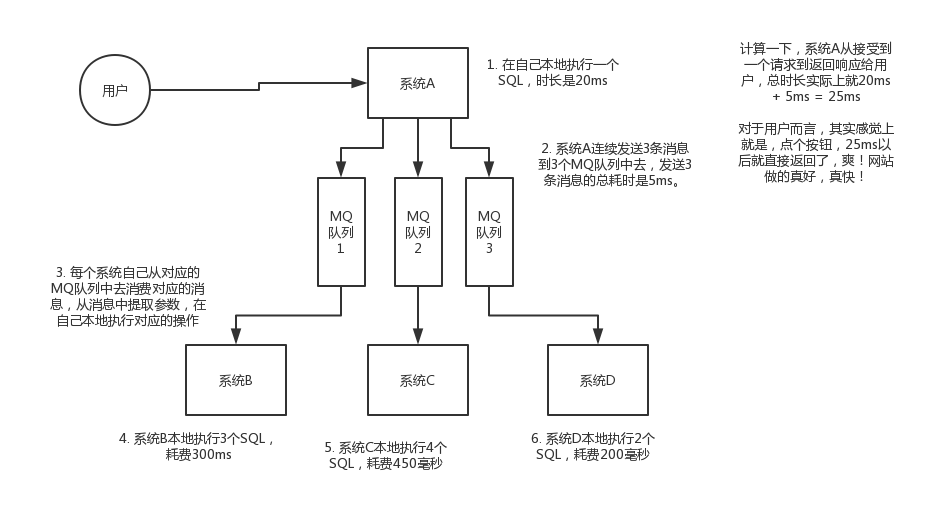
05-02



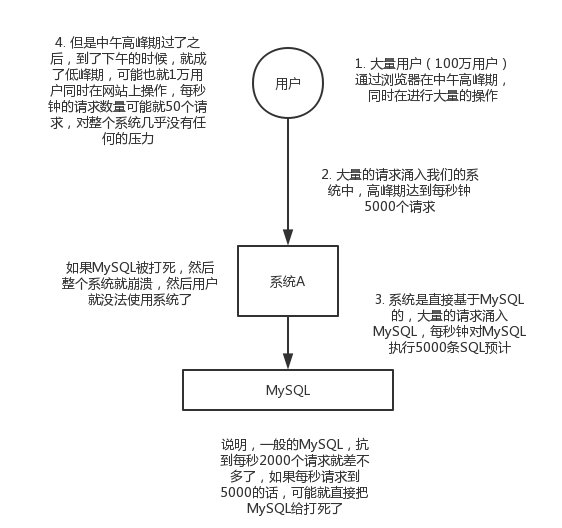
05-03

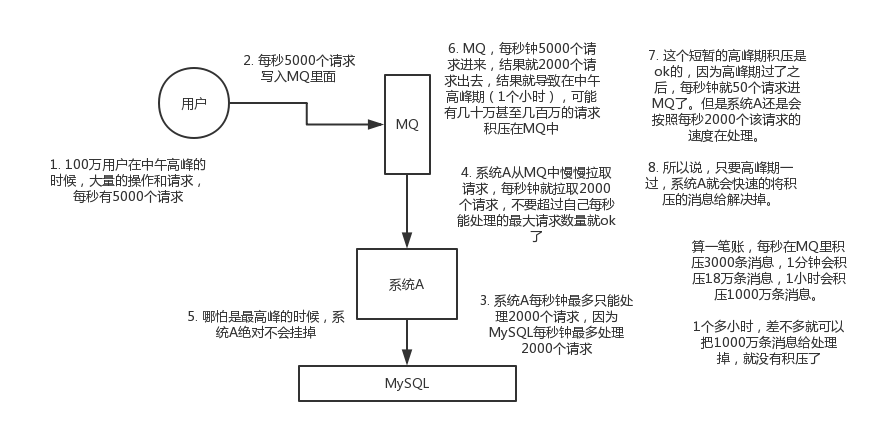


05-04

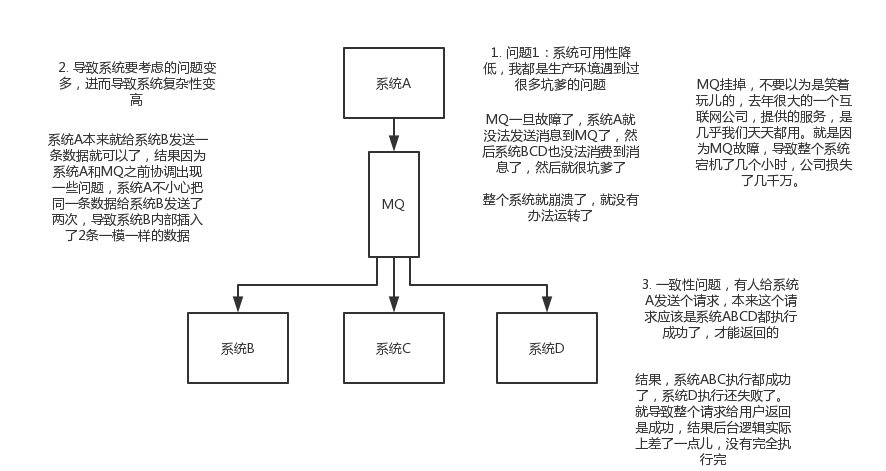


05-05

  
05-06



05-07



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特性 | ActiveMQ | RabbitMQ | RocketMQ | Kafka |
| 单机吞吐量 | 万级，吞吐量比RocketMQ和Kafka要低了一个数量级 | 万级，吞吐量比RocketMQ和Kafka要低了一个数量级 | 10万级，RocketMQ也是可以支撑高吞吐的一种MQ | 10万级别，这是kafka最大的优点，就是吞吐量高。  一般配合大数据类的系统来进行实时数据计算、日志采集等场景 |
| topic数量对吞吐量的影响 |  |  | topic可以达到几百，几千个的级别，吞吐量会有较小幅度的下降  这是RocketMQ的一大优势，在同等机器下，可以支撑大量的topic | topic从几十个到几百个的时候，吞吐量会大幅度下降  所以在同等机器下，kafka尽量保证topic数量不要过多。如果要支撑大规模topic，需要增加更多的机器资源 |
| 时效性 | ms级 | 微秒级，这是rabbitmq的一大特点，延迟是最低的 | ms级 | 延迟在ms级以内 |
| 可用性 | 高，基于主从架构实现高可用性 | 高，基于主从架构实现高可用性 | 非常高，分布式架构 | 非常高，kafka是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用 |
| 消息可靠性 | 有较低的概率丢失数据 |  | 经过参数优化配置，可以做到0丢失 | 经过参数优化配置，消息可以做到0丢失 |
| 功能支持 | MQ领域的功能极其完备 | 基于erlang开发，所以并发能力很强，性能极其好，延时很低 | MQ功能较为完善，还是分布式的，扩展性好 | 功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准 |
| 优劣势总结 | 非常成熟，功能强大，在业内大量的公司以及项目中都有应用  偶尔会有较低概率丢失消息  而且现在社区以及国内应用都越来越少，官方社区现在对ActiveMQ 5.x维护越来越少，几个月才发布一个版本  而且确实主要是基于解耦和异步来用的，较少在大规模吞吐的场景中使用 | erlang语言开发，性能极其好，延时很低；  吞吐量到万级，MQ功能比较完备  而且开源提供的管理界面非常棒，用起来很好用  社区相对比较活跃，几乎每个月都发布几个版本分  在国内一些互联网公司近几年用rabbitmq也比较多一些  但是问题也是显而易见的，RabbitMQ确实吞吐量会低一些，这是因为他做的实现机制比较重。  而且erlang开发，国内有几个公司有实力做erlang源码级别的研究和定制？如果说你没这个实力的话，确实偶尔会有一些问题，你很难去看懂源码，你公司对这个东西的掌控很弱，基本职能依赖于开源社区的快速维护和修复bug。  而且rabbitmq集群动态扩展会很麻烦，不过这个我觉得还好。其实主要是erlang语言本身带来的问题。很难读源码，很难定制和掌控。 | 接口简单易用，而且毕竟在阿里大规模应用过，有阿里品牌保障  日处理消息上百亿之多，可以做到大规模吞吐，性能也非常好，分布式扩展也很方便，社区维护还可以，可靠性和可用性都是ok的，还可以支撑大规模的topic数量，支持复杂MQ业务场景  而且一个很大的优势在于，阿里出品都是java系的，我们可以自己阅读源码，定制自己公司的MQ，可以掌控  社区活跃度相对较为一般，不过也还可以，文档相对来说简单一些，然后接口这块不是按照标准JMS规范走的有些系统要迁移需要修改大量代码  还有就是阿里出台的技术，你得做好这个技术万一被抛弃，社区黄掉的风险，那如果你们公司有技术实力我觉得用RocketMQ挺好的 | kafka的特点其实很明显，就是仅仅提供较少的核心功能，但是提供超高的吞吐量，ms级的延迟，极高的可用性以及可靠性，而且分布式可以任意扩展  同时kafka最好是支撑较少的topic数量即可，保证其超高吞吐量  而且kafka唯一的一点劣势是有可能消息重复消费，那么对数据准确性会造成极其轻微的影响，在大数据领域中以及日志采集中，这点轻微影响可以忽略  这个特性天然适合大数据实时计算以及日志收集 |

综上所述，各种对比之后，我个人倾向于是：

一般的业务系统要引入MQ，最早大家都用ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是算了吧，我个人不推荐用这个了；

后来大家开始用RabbitMQ，但是确实erlang语言阻止了大量的java工程师去深入研究和掌控他，对公司而言，几乎处于不可控的状态，但是确实人是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司，会去用RocketMQ，确实很不错，但是我提醒一下自己想好社区万一突然黄掉的风险，对自己公司技术实力有绝对自信的，我推荐用RocketMQ，否则回去老老实实用RabbitMQ吧，人是活跃开源社区，绝对不会黄

所以中小型公司，技术实力较为一般，技术挑战不是特别高，用RabbitMQ是不错的选择；大型公司，基础架构研发实力较强，用RocketMQ是很好的选择

如果是大数据领域的实时计算、日志采集等场景，用Kafka是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范

### 如何保证消息队列的高可用啊？

1、面试题

如何保证消息队列的高可用啊？

2、面试官心理分析

如果有人问到你MQ的知识，高可用是必问的，因为MQ的缺点，我刚才已经说过了，有好多，导致系统可用性降低，等等。所以只要你用了MQ，接下来问的一些要点肯定就是围绕着MQ的那些缺点怎么来解决了。

要是你傻乎乎的就干用了一个MQ，各种问题从来没考虑过，那你就杯具了，面试官对你的印象就是，只会简单实用一些技术，没任何思考，马上对你的印象就不太好了。这样的同学招进来要是做个20k薪资以内的普通小弟还凑合。如果招进来做薪资20多k的高工，那就惨了，让你设计个系统，里面肯定一堆坑，出了事故公司受损失，团队一起背锅。

去年的事儿，非常大的互联网公司，非常核心的系统，就是疏忽了MQ，没考虑MQ如何保证高可用，如果MQ挂了怎么办，导致几个小时系统不可用，公司损失几千万，team背锅，你闹的祸，你老大帮你一起背锅

3、面试题剖析

这个问题这么问是很好的，因为不能问你kafka的高可用性怎么保证啊？ActiveMQ的高可用性怎么保证啊？一个面试官要是这么问就显得很没水平，人家可能用的就是RabbitMQ，没用过Kafka，你上来问人家kafka干什么？这不是摆明了刁难人么。

所以有水平的面试官，问的是MQ的高可用性怎么保证？这样就是你用过哪个MQ，你就说说你对那个MQ的高可用性的理解。

（1）RabbitMQ的高可用性

RabbitMQ是比较有代表性的，因为是基于主从做高可用性的，我们就以他为例子讲解第一种MQ的高可用性怎么实现。

rabbitmq有三种模式：单机模式，普通集群模式，镜像集群模式

1）单机模式

就是demo级别的，一般就是你本地启动了玩玩儿的，没人生产用单机模式

2）普通集群模式

意思就是在多台机器上启动多个rabbitmq实例，每个机器启动一个。但是你创建的queue，只会放在一个rabbtimq实例上，但是每个实例都同步queue的元数据。完了你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从queue所在实例上拉取数据过来。

这种方式确实很麻烦，也不怎么好，没做到所谓的分布式，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个queue所在实例消费数据，前者有数据拉取的开销，后者导致单实例性能瓶颈。

而且如果那个放queue的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你开启了消息持久化，让rabbitmq落地存储消息的话，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个queue拉取数据。

所以这个事儿就比较尴尬了，这就没有什么所谓的高可用性可言了，这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个queue的读写操作。

3）镜像集群模式

这种模式，才是所谓的rabbitmq的高可用模式，跟普通集群模式不一样的是，你创建的queue，无论元数据还是queue里的消息都会存在于多个实例上，然后每次你写消息到queue的时候，都会自动把消息到多个实例的queue里进行消息同步。

这样的话，好处在于，你任何一个机器宕机了，没事儿，别的机器都可以用。坏处在于，第一，这个性能开销也太大了吧，消息同步所有机器，导致网络带宽压力和消耗很重！第二，这么玩儿，就没有扩展性可言了，如果某个queue负载很重，你加机器，新增的机器也包含了这个queue的所有数据，并没有办法线性扩展你的queue

那么怎么开启这个镜像集群模式呢？我这里简单说一下，避免面试人家问你你不知道，其实很简单rabbitmq有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候可以要求数据同步到所有节点的，也可以要求就同步到指定数量的节点，然后你再次创建queue的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。

（2）kafka的高可用性

kafka一个最基本的架构认识：多个broker组成，每个broker是一个节点；你创建一个topic，这个topic可以划分为多个partition，每个partition可以存在于不同的broker上，每个partition就放一部分数据。

这就是天然的分布式消息队列，就是说一个topic的数据，是分散放在多个机器上的，每个机器就放一部分数据。

实际上rabbitmq之类的，并不是分布式消息队列，他就是传统的消息队列，只不过提供了一些集群、HA的机制而已，因为无论怎么玩儿，rabbitmq一个queue的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个queue的完整数据。

kafka 0.8以前，是没有HA机制的，就是任何一个broker宕机了，那个broker上的partition就废了，没法写也没法读，没有什么高可用性可言。

kafka 0.8以后，提供了HA机制，就是replica副本机制。每个partition的数据都会同步到吉他机器上，形成自己的多个replica副本。然后所有replica会选举一个leader出来，那么生产和消费都跟这个leader打交道，然后其他replica就是follower。写的时候，leader会负责把数据同步到所有follower上去，读的时候就直接读leader上数据即可。只能读写leader？很简单，要是你可以随意读写每个follower，那么就要care数据一致性的问题，系统复杂度太高，很容易出问题。kafka会均匀的将一个partition的所有replica分布在不同的机器上，这样才可以提高容错性。

这么搞，就有所谓的高可用性了，因为如果某个broker宕机了，没事儿，那个broker上面的partition在其他机器上都有副本的，如果这上面有某个partition的leader，那么此时会重新选举一个新的leader出来，大家继续读写那个新的leader即可。这就有所谓的高可用性了。

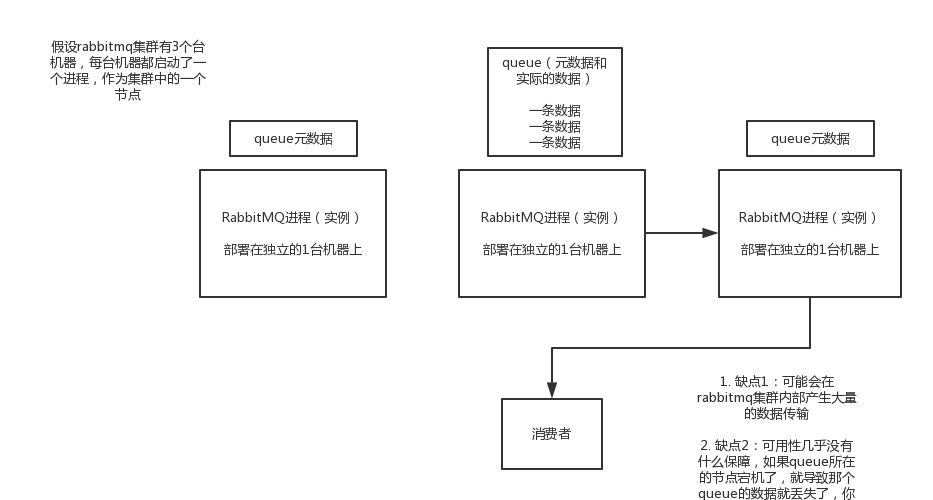
写数据的时候，生产者就写leader，然后leader将数据落地写本地磁盘，接着其他follower自己主动从leader来pull数据。一旦所有follower同步好数据了，就会发送ack给leader，leader收到所有follower的ack之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）

消费的时候，只会从leader去读，但是只有一个消息已经被所有follower都同步成功返回ack的时候，这个消息才会被消费者读到。

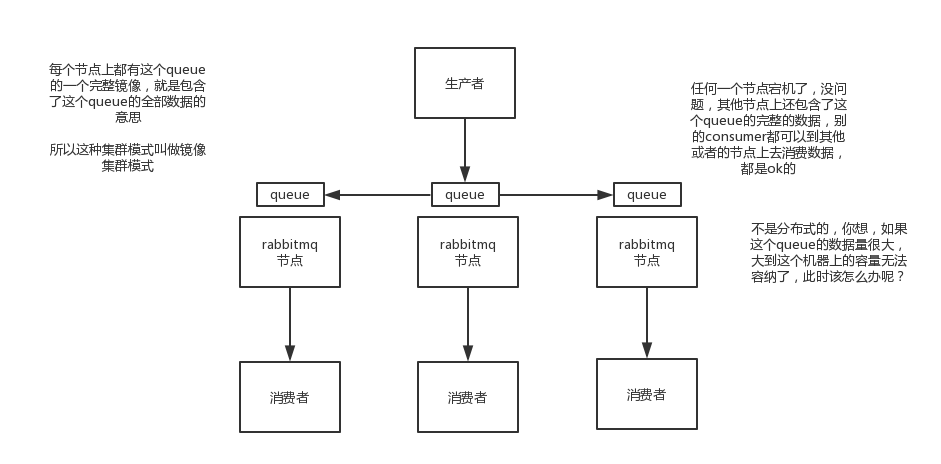
实际上这块机制，讲深了，是可以非常之深入的，但是我还是回到我们这个课程的主题和定位，聚焦面试，至少你听到这里大致明白了kafka是如何保证高可用机制的了，对吧？不至于一无所知，现场还能给面试官画画图。要遇上面试官确实是kafka高手，深挖了问，那你只能说不好意思，太深入的你没研究过。

但是大家一定要明白，这个事情是要权衡的，你现在是要快速突击常见面试题体系，而不是要深入学习kafka，要深入学习kafka，你是没那么多时间的。你只能确保，你之前也许压根儿不知道这块，但是现在你知道了，面试被问到，你大概可以说一说。然后很多其他的候选人，也许还不如你，没看过这个，被问到了压根儿答不出来，相比之下，你还能说点出来，大概就是这个意思了。

06-01

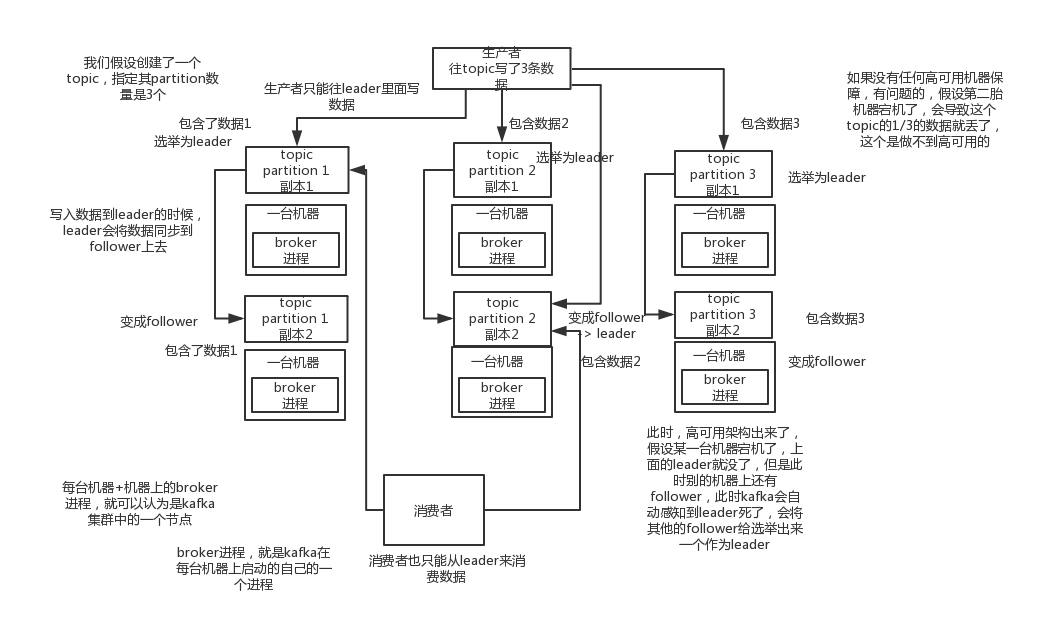


06-02



06-03

06-04



### （3）如何保证消息不被重复消费啊（如何进行消息队列的幂等性问题）？

1、面试题

如何保证消息不被重复消费啊（如何保证消息消费时的幂等性）？

2、面试官心里分析

其实这个很常见的一个问题，这俩问题基本可以连起来问。既然是消费消息，那肯定要考虑考虑会不会重复消费？能不能避免重复消费？或者重复消费了也别造成系统异常可以吗？这个是MQ领域的基本问题，其实本质上还是问你使用消息队列如何保证幂等性，这个是你架构里要考虑的一个问题。

面试官问你，肯定是必问的，这是你要考虑的实际生产上的系统设计问题。

3、面试题剖析

回答这个问题，首先你别听到重复消息这个事儿，就一无所知吧，你先大概说一说可能会有哪些重复消费的问题。

首先就是比如rabbitmq、rocketmq、kafka，都有可能会出现消费重复消费的问题，正常。因为这问题通常不是mq自己保证的，是给你保证的。然后我们挑一个kafka来举个例子，说说怎么重复消费吧。

kafka实际上有个offset的概念，就是每个消息写进去，都有一个offset，代表他的序号，然后consumer消费了数据之后，每隔一段时间，会把自己消费过的消息的offset提交一下，代表我已经消费过了，下次我要是重启啥的，你就让我继续从上次消费到的offset来继续消费吧。

但是凡事总有意外，比如我们之前生产经常遇到的，就是你有时候重启系统，看你怎么重启了，如果碰到点着急的，直接kill进程了，再重启。这会导致consumer有些消息处理了，但是没来得及提交offset，尴尬了。重启之后，少数消息会再次消费一次。

其实重复消费不可怕，可怕的是你没考虑到重复消费之后，怎么保证幂等性。

给你举个例子吧。假设你有个系统，消费一条往数据库里插入一条，要是你一个消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下已经消费过了，直接扔了，不就保留了一条数据？

一条数据重复出现两次，数据库里就只有一条数据，这就保证了系统的幂等性

幂等性，我通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

那所以第二个问题来了，怎么保证消息队列消费的幂等性？

其实还是得结合业务来思考，我这里给几个思路：

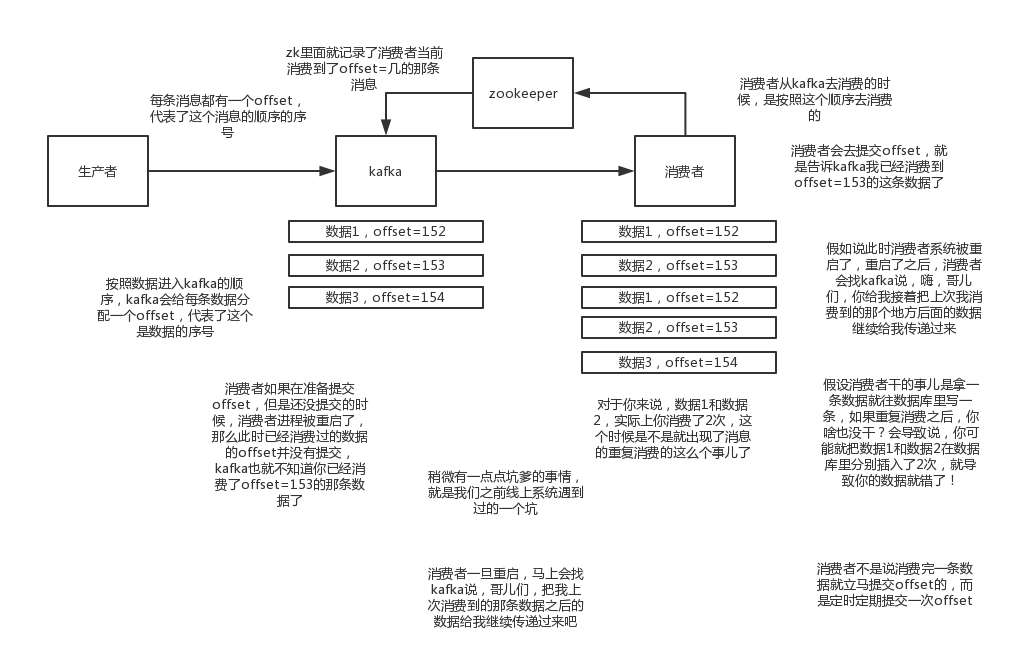
（1）比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update一下好吧

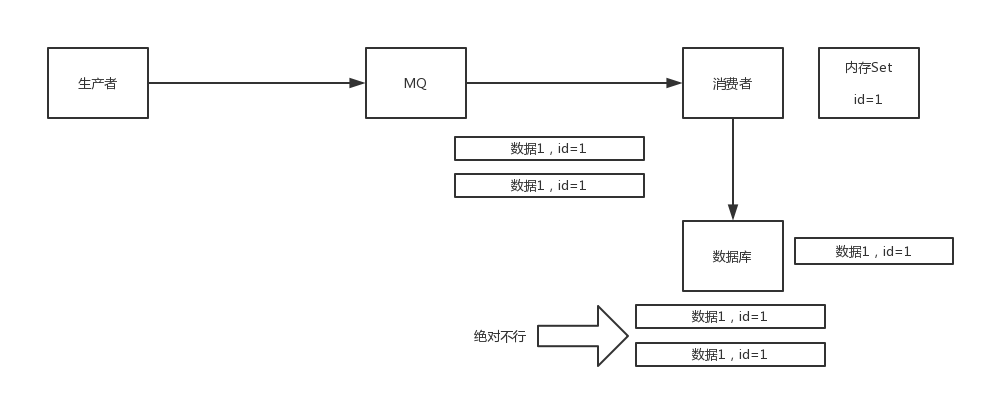
（2）比如你是写redis，那没问题了，反正每次都是set，天然幂等性

（3）比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的id，类似订单id之类的东西，然后你这里消费到了之后，先根据这个id去比如redis里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个id写redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

还有比如基于数据库的唯一键来保证重复数据不会重复插入多条，我们之前线上系统就有这个问题，就是拿到数据的时候，每次重启可能会有重复，因为kafka消费者还没来得及提交offset，重复数据拿到了以后我们插入的时候，因为有唯一键约束了，所以重复数据只会插入报错，不会导致数据库中出现脏数据

如何保证MQ的消费是幂等性的，需要结合具体的业务来看





### （4）如何保证消息的可靠性传输（如何处理消息丢失的问题）？

1、面试题

如何保证消息的可靠性传输（如何处理消息丢失的问题）？

2、面试官心里分析

这个是肯定的，用mq有个基本原则，就是数据不能多一条，也不能少一条，不能多，就是刚才说的重复消费和幂等性问题。不能少，就是说这数据别搞丢了。那这个问题你必须得考虑一下。

如果说你这个是用mq来传递非常核心的消息，比如说计费，扣费的一些消息，因为我以前设计和研发过一个公司非常核心的广告平台，计费系统，计费系统是很重的一个业务，操作是很耗时的。所以说广告系统整体的架构里面，实际上是将计费做成异步化的，然后中间就是加了一个MQ。

我们当时为了确保说这个MQ传递过程中绝对不会把计费消息给弄丢，花了很多的精力。广告主投放了一个广告，明明说好了，用户点击一次扣费1块钱。结果要是用户动不动点击了一次，扣费的时候搞的消息丢了，我们公司就会不断的少几块钱，几块钱，积少成多，这个就对公司是一个很大的损失。

3、面试题剖析

这个丢数据，mq一般分为两种，要么是mq自己弄丢了，要么是我们消费的时候弄丢了。咱们从rabbitmq和kafka分别来分析一下吧

rabbitmq这种mq，一般来说都是承载公司的核心业务的，数据是绝对不能弄丢的

（1）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，如果你要确保说写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的时候将其设置为持久化的，这样就可以保证rabbitmq持久化queue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。

（2）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。

这不是一样么，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。

生产环境也遇到过，我们也是，之前kafka的leader机器宕机了，将follower切换为leader之后，就会发现说这个数据就丢了

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

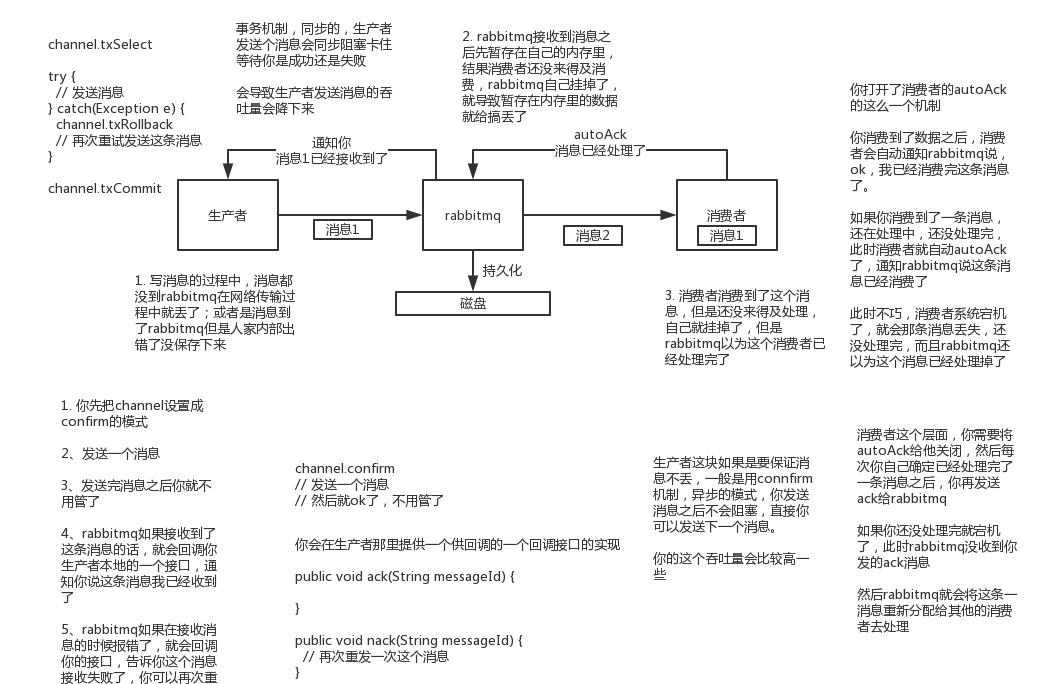
在producer端设置acks=all：这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了

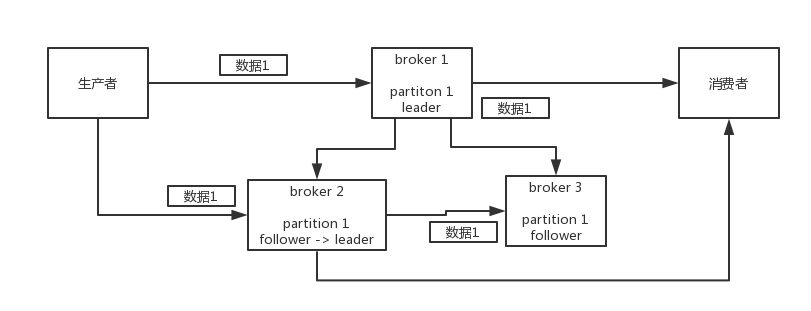
在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了

我们生产环境就是按照上述要求配置的，这样配置之后，至少在kafka broker端就可以保证在leader所在broker发生故障，进行leader切换时，数据不会丢失

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。





### （5）如何保证消息的顺序性？

1、面试题

如何保证消息的顺序性？

2、面试官心里分析

其实这个也是用MQ的时候必问的话题，第一看看你了解不了解顺序这个事儿？第二看看你有没有办法保证消息是有顺序的？这个生产系统中常见的问题。

3、面试题剖析

我举个例子，我们以前做过一个mysql binlog同步的系统，压力还是非常大的，日同步数据要达到上亿。mysql -> mysql，常见的一点在于说大数据team，就需要同步一个mysql库过来，对公司的业务系统的数据做各种复杂的操作。

你在mysql里增删改一条数据，对应出来了增删改3条binlog，接着这三条binlog发送到MQ里面，到消费出来依次执行，起码得保证人家是按照顺序来的吧？不然本来是：增加、修改、删除；你楞是换了顺序给执行成删除、修改、增加，不全错了么。

本来这个数据同步过来，应该最后这个数据被删除了；结果你搞错了这个顺序，最后这个数据保留下来了，数据同步就出错了。

先看看顺序会错乱的俩场景

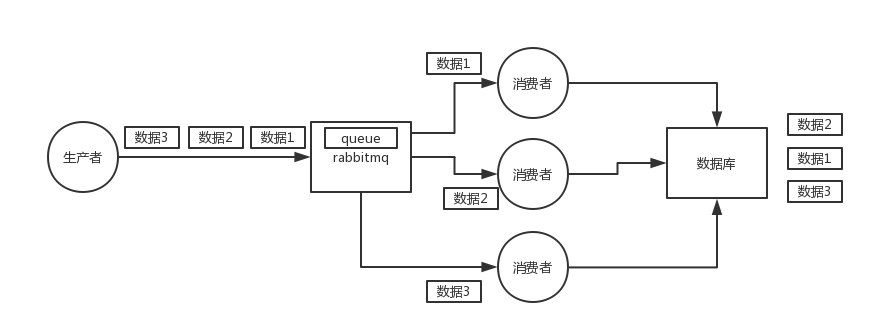
（1）rabbitmq：一个queue，多个consumer，这不明显乱了

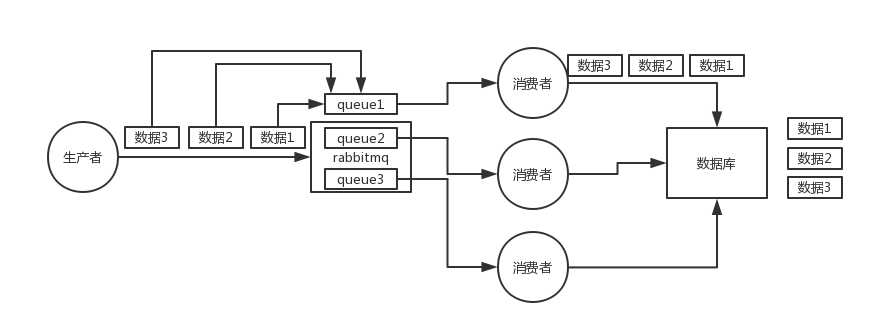
（2）kafka：一个topic，一个partition，一个consumer，内部多线程，这不也明显乱了

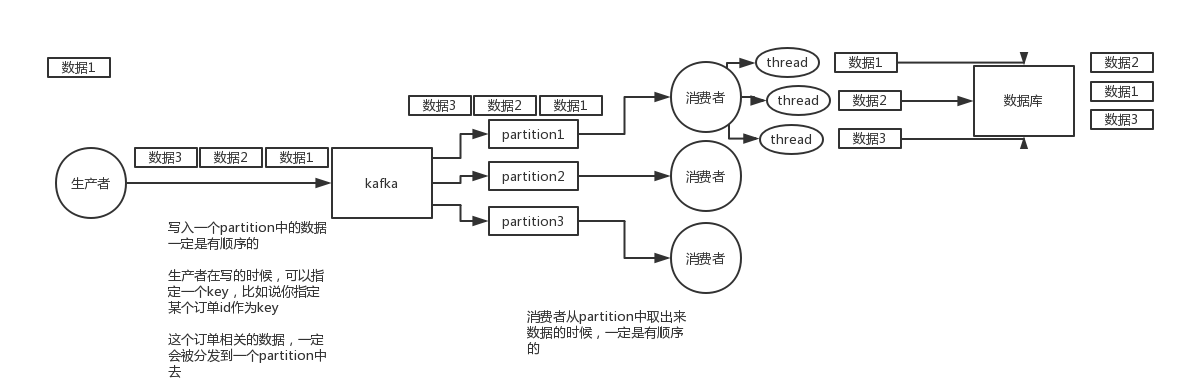
那如何保证消息的顺序性呢？简单简单

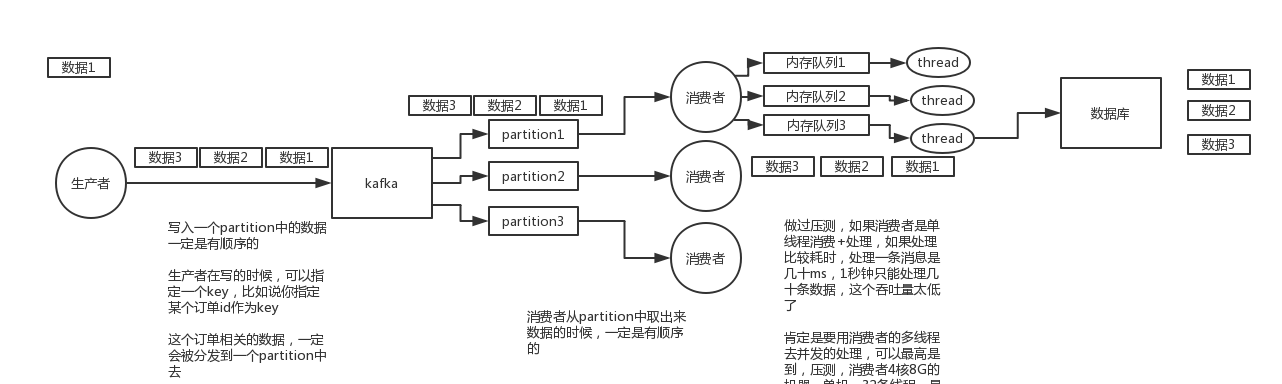
（1）rabbitmq：拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理

（2）kafka：一个topic，一个partition，一个consumer，内部单线程消费，写N个内存queue，然后N个线程分别消费一个内存queue即可









### （6）如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

1、面试题

如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

2、面试官心里分析

你看这问法，其实本质针对的场景，都是说，可能你的消费端出了问题，不消费了，或者消费的极其极其慢。接着就坑爹了，可能你的消息队列集群的磁盘都快写满了，都没人消费，这个时候怎么办？或者是整个这就积压了几个小时，你这个时候怎么办？或者是你积压的时间太长了，导致比如rabbitmq设置了消息过期时间后就没了怎么办？

所以就这事儿，其实线上挺常见的，一般不出，一出就是大case，一般常见于，举个例子，消费端每次消费之后要写mysql，结果mysql挂了，消费端hang那儿了，不动了。或者是消费端出了个什么叉子，导致消费速度极其慢。

3、面试题分析

关于这个事儿，我们一个一个来梳理吧，先假设一个场景，我们现在消费端出故障了，然后大量消息在mq里积压，现在事故了，慌了

（1）大量消息在mq里积压了几个小时了还没解决

几千万条数据在MQ里积压了七八个小时，从下午4点多，积压到了晚上很晚，10点多，11点多

这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复consumer的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条

所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

1）先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉

2）新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量

3）然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的10倍数量的queue

4）接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据

5）这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据

6）等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息

（2）这里我们假设再来第二个坑

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。

假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次

（3）然后我们再来假设第三个坑

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。



### 如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路

### 总结一下消息队列相关问题的面试技巧

一般而言，如果一个面试官水平还算不错，会沿着从浅入深的环节深入挖一个点。比如我吧，其实按照这个思路可以一直问下去，除了这里的7个问题之外，甚至能挑着你熟悉的一个mq一直问到源码级别非常底层。我还可能会结合项目来仔细问，我可能会先让你给我详细说说你的业务细节，然后将你的业务跟这些mq的问题场景结合起来，看看你每个细节是怎么处理的。

但是确实因为我们这个是面试突击型课程，不是什么kafka源码剖析课，也不是什么RocketMQ高并发架构项目实战课程，所以只能讲到这个程度。

所以我们这个课程只能让你从大面儿上，基本常见问题可以回答出来。基本上mq这块你能答到这个程度，你基本知识面儿是有了，但是深度是绝对没有的。所以如果一个面试官就问问这些问题，感觉你面儿上过的去了，那就恭喜你了。但是如果碰到我这种难缠的面试官，喜欢深挖底层，细扣项目细节的，那可能确实是不行的。

如果你碰到人家在7个问题之外还死扣着你问的，那你最好是认一下怂，就说你确实没研究那么深过，如果你面的就是个一般的职位，那可能就过去了。就我而言，如果招聘的就是个普通职位，而你能答到这个程度，那么就觉得说的过去了。毕竟说实话，相当大比例的程序员出去面java职位的时候，mq这块还回答不到这个程度呢。你能答好这些，至少比之前一无所知的你好了一些，也比很多没准备过的程序员都好了很多。

最后说一个技巧，要是确实碰一个面试官连这7个问题都没问满，只要他提到mq，你自己就和盘托出一整套的东西，你就说，mq你们之前遇到过什么问题，巴拉巴拉，你们的方案是什么，自己突出自己会的东西

1、面试题

如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路

2、面试官心里分析

其实聊到这个问题，一般面试官要考察两块：

（1）你有没有对某一个消息队列做过较为深入的原理的了解，或者从整体了解把握住一个mq的架构原理

（2）看看你的设计能力，给你一个常见的系统，就是消息队列系统，看看你能不能从全局把握一下整体架构设计，给出一些关键点出来

说实话，我一般面类似问题的时候，大部分人基本都会蒙，因为平时从来没有思考过类似的问题，大多数人就是平时埋头用，从来不去思考背后的一些东西。类似的问题，我经常问的还有，如果让你来设计一个spring框架你会怎么做？如果让你来设计一个dubbo框架你会怎么做？如果让你来设计一个mybatis框架你会怎么做？

3、面试题剖析

其实回答这类问题，说白了，起码不求你看过那技术的源码，起码你大概知道那个技术的基本原理，核心组成部分，基本架构构成，然后参照一些开源的技术把一个系统设计出来的思路说一下就好

比如说这个消息队列系统，我们来从以下几个角度来考虑一下

（1）首先这个mq得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下kafka的设计理念，broker -> topic -> partition，每个partition放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给topic增加partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

（2）其次你得考虑一下这个mq的数据要不要落地磁盘吧？那肯定要了，落磁盘，才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是kafka的思路。

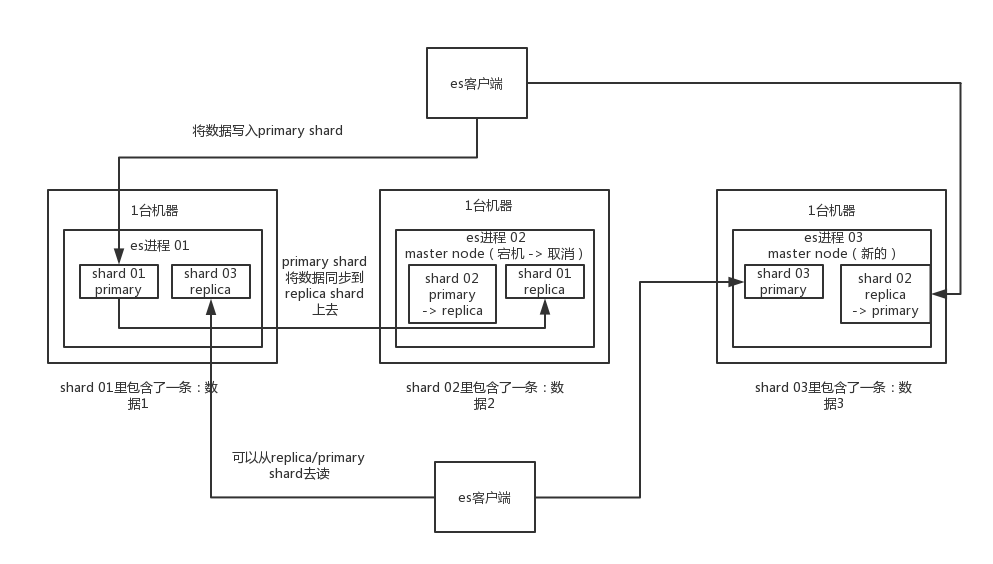
1. 其次你考虑一下你的mq的可用性啊？这个事儿，具体参考我们之前可用性那个环节讲解的kafka的高可用保障机制。多副本 -> leader & follower -> broker挂了重新选举leader即可对外服务。

（4）能不能支持数据0丢失啊？可以的，参考我们之前说的那个kafka数据零丢失方案

其实一个mq肯定是很复杂的，面试官问你这个问题，其实是个开放题，他就是看看你有没有从架构角度整体构思和设计的思维以及能力。确实这个问题可以刷掉一大批人，因为大部分人平时不思考这些东西。

## 2.3 搜索引擎

### （1）es的分布式架构原理能说一下么（es是如何实现分布式的啊）？



1、面试题

es的分布式架构原理能说一下么（es是如何实现分布式的啊）？

2、面试官心里分析

在搜索这块，lucene是最流行的搜索库。几年前业内一般都问，你了解lucene吗？你知道倒排索引的原理吗？现在早已经out了，因为现在很多项目都是直接用基于lucene的分布式搜索引擎——elasticsearch，简称为es。

而现在分布式搜索基本已经成为大部分互联网行业的java系统的标配，其中尤为流行的就是es，前几年es没火的时候，大家一般用solr。但是这两年基本大部分企业和项目都开始转向es了。

所以互联网面试，肯定会跟你聊聊分布式搜索引擎，也就一定会聊聊es，如果你确实不知道，那你真的就out了。

如果面试官问你第一个问题，确实一般都会问你es的分布式架构设计能介绍一下么？就看看你对分布式搜索引擎架构的一个基本理解。

3、额外的友情提示

同学啊，如果你看到这里发现自己对es一无所知，没事儿，保持淡定，暂停一下课程。然后上百度搜一下es是啥？本机启动个es？然后写个es的hello world感受一下？然后搜个帖子把es常见的几个操作都执行一遍（聚合、常见搜索语法之类的）？ok了，1~2小时熟悉足够了，回来吧，继续看我们的课程。

4、面试题剖析

elasticsearch设计的理念就是分布式搜索引擎，底层其实还是基于lucene的。

核心思想就是在多台机器上启动多个es进程实例，组成了一个es集群。

es中存储数据的基本单位是索引，比如说你现在要在es中存储一些订单数据，你就应该在es中创建一个索引，order\_idx，所有的订单数据就都写到这个索引里面去，一个索引差不多就是相当于是mysql里的一张表。index -> type -> mapping -> document -> field。

index：mysql里的一张表

type：没法跟mysql里去对比，一个index里可以有多个type，每个type的字段都是差不多的，但是有一些略微的差别。

好比说，有一个index，是订单index，里面专门是放订单数据的。就好比说你在mysql中建表，有些订单是实物商品的订单，就好比说一件衣服，一双鞋子；有些订单是虚拟商品的订单，就好比说游戏点卡，话费充值。就两种订单大部分字段是一样的，但是少部分字段可能有略微的一些差别。

所以就会在订单index里，建两个type，一个是实物商品订单type，一个是虚拟商品订单type，这两个type大部分字段是一样的，少部分字段是不一样的。

很多情况下，一个index里可能就一个type，但是确实如果说是一个index里有多个type的情况，你可以认为index是一个类别的表，具体的每个type代表了具体的一个mysql中的表

每个type有一个mapping，如果你认为一个type是一个具体的一个表，index代表了多个type的同属于的一个类型，mapping就是这个type的表结构定义，你在mysql中创建一个表，肯定是要定义表结构的，里面有哪些字段，每个字段是什么类型。。。

mapping就代表了这个type的表结构的定义，定义了这个type中每个字段名称，字段是什么类型的，然后还有这个字段的各种配置

实际上你往index里的一个type里面写的一条数据，叫做一条document，一条document就代表了mysql中某个表里的一行给，每个document有多个field，每个field就代表了这个document中的一个字段的值

接着你搞一个索引，这个索引可以拆分成多个shard，每个shard存储部分数据。

接着就是这个shard的数据实际是有多个备份，就是说每个shard都有一个primary shard，负责写入数据，但是还有几个replica shard。primary shard写入数据之后，会将数据同步到其他几个replica shard上去。

通过这个replica的方案，每个shard的数据都有多个备份，如果某个机器宕机了，没关系啊，还有别的数据副本在别的机器上呢。高可用了吧。

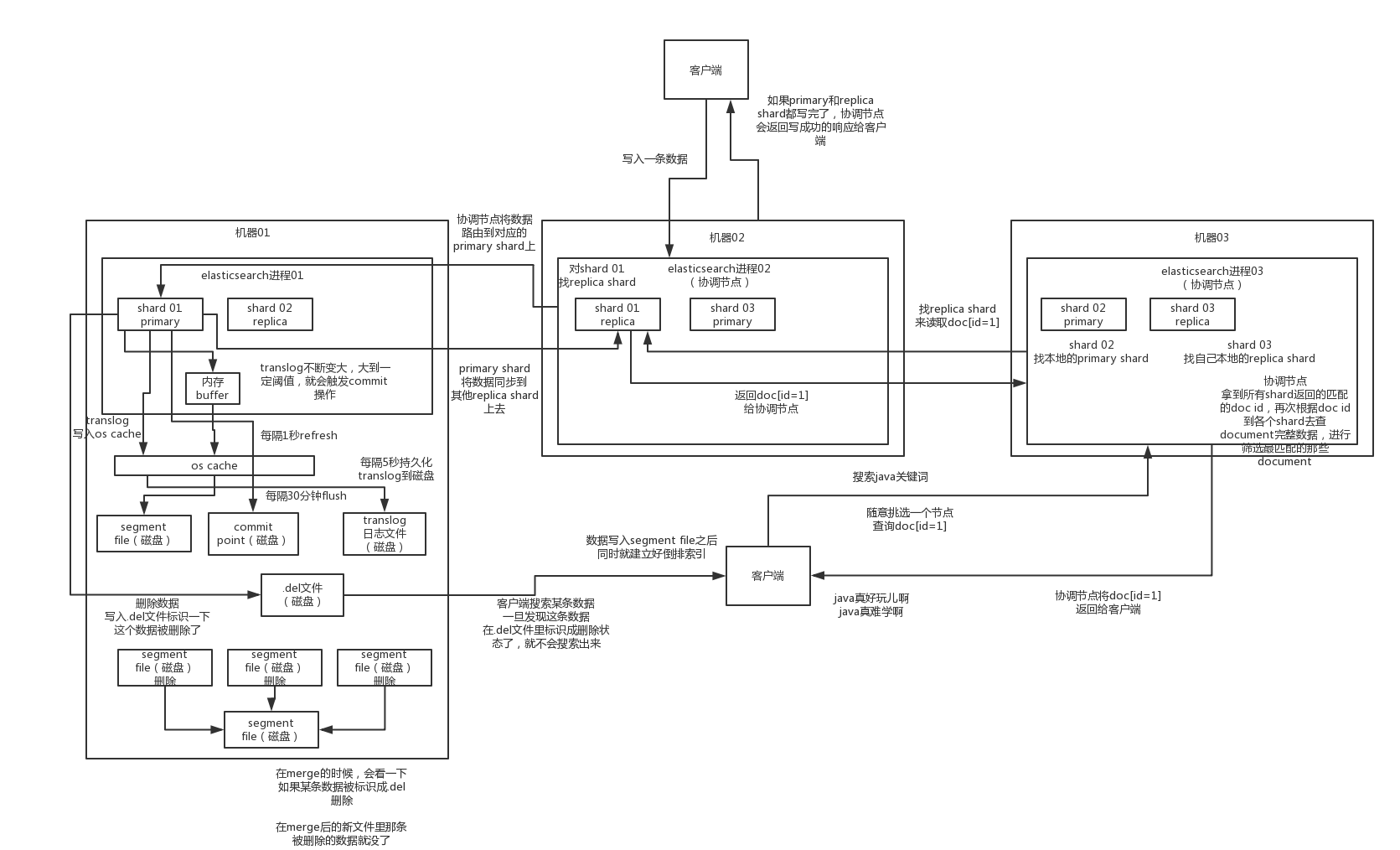
es集群多个节点，会自动选举一个节点为master节点，这个master节点其实就是干一些管理的工作的，比如维护索引元数据拉，负责切换primary shard和replica shard身份拉，之类的。

要是master节点宕机了，那么会重新选举一个节点为master节点。

如果是非master节点宕机了，那么会由master节点，让那个宕机节点上的primary shard的身份转移到其他机器上的replica shard。急着你要是修复了那个宕机机器，重启了之后，master节点会控制将缺失的replica shard分配过去，同步后续修改的数据之类的，让集群恢复正常。

其实上述就是elasticsearch作为一个分布式搜索引擎最基本的一个架构设计

### （2）es写入数据的工作原理是什么啊？es查询数据的工作原理是什么啊？底层的lucene介绍一下呗？倒排索引了解吗？



1、面试题

es的分布式架构原理能说一下么（es是如何实现分布式的啊）？

2、面试官心里分析

在搜索这块，lucene是最流行的搜索库。几年前业内一般都问，你了解lucene吗？你知道倒排索引的原理吗？现在早已经out了，因为现在很多项目都是直接用基于lucene的分布式搜索引擎——elasticsearch，简称为es。

而现在分布式搜索基本已经成为大部分互联网行业的java系统的标配，其中尤为流行的就是es，前几年es没火的时候，大家一般用solr。但是这两年基本大部分企业和项目都开始转向es了。

所以互联网面试，肯定会跟你聊聊分布式搜索引擎，也就一定会聊聊es，如果你确实不知道，那你真的就out了。

如果面试官问你第一个问题，确实一般都会问你es的分布式架构设计能介绍一下么？就看看你对分布式搜索引擎架构的一个基本理解。

3、额外的友情提示

同学啊，如果你看到这里发现自己对es一无所知，没事儿，保持淡定，暂停一下课程。然后上百度搜一下es是啥？本机启动个es？然后写个es的hello world感受一下？然后搜个帖子把es常见的几个操作都执行一遍（聚合、常见搜索语法之类的）？ok了，1~2小时熟悉足够了，回来吧，继续看我们的课程。

4、面试题剖析

elasticsearch设计的理念就是分布式搜索引擎，底层其实还是基于lucene的。

核心思想就是在多台机器上启动多个es进程实例，组成了一个es集群。

es中存储数据的基本单位是索引，比如说你现在要在es中存储一些订单数据，你就应该在es中创建一个索引，order\_idx，所有的订单数据就都写到这个索引里面去，一个索引差不多就是相当于是mysql里的一张表。index -> type -> mapping -> document -> field。

index：mysql里的一张表

type：没法跟mysql里去对比，一个index里可以有多个type，每个type的字段都是差不多的，但是有一些略微的差别。

好比说，有一个index，是订单index，里面专门是放订单数据的。就好比说你在mysql中建表，有些订单是实物商品的订单，就好比说一件衣服，一双鞋子；有些订单是虚拟商品的订单，就好比说游戏点卡，话费充值。就两种订单大部分字段是一样的，但是少部分字段可能有略微的一些差别。

所以就会在订单index里，建两个type，一个是实物商品订单type，一个是虚拟商品订单type，这两个type大部分字段是一样的，少部分字段是不一样的。

很多情况下，一个index里可能就一个type，但是确实如果说是一个index里有多个type的情况，你可以认为index是一个类别的表，具体的每个type代表了具体的一个mysql中的表

每个type有一个mapping，如果你认为一个type是一个具体的一个表，index代表了多个type的同属于的一个类型，mapping就是这个type的表结构定义，你在mysql中创建一个表，肯定是要定义表结构的，里面有哪些字段，每个字段是什么类型。。。

mapping就代表了这个type的表结构的定义，定义了这个type中每个字段名称，字段是什么类型的，然后还有这个字段的各种配置

实际上你往index里的一个type里面写的一条数据，叫做一条document，一条document就代表了mysql中某个表里的一行给，每个document有多个field，每个field就代表了这个document中的一个字段的值

接着你搞一个索引，这个索引可以拆分成多个shard，每个shard存储部分数据。

接着就是这个shard的数据实际是有多个备份，就是说每个shard都有一个primary shard，负责写入数据，但是还有几个replica shard。primary shard写入数据之后，会将数据同步到其他几个replica shard上去。

通过这个replica的方案，每个shard的数据都有多个备份，如果某个机器宕机了，没关系啊，还有别的数据副本在别的机器上呢。高可用了吧。

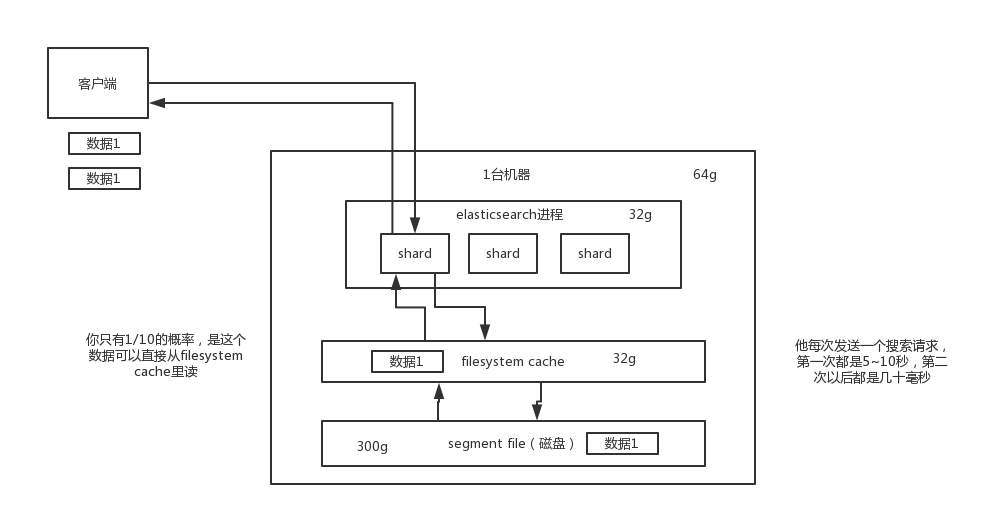
es集群多个节点，会自动选举一个节点为master节点，这个master节点其实就是干一些管理的工作的，比如维护索引元数据拉，负责切换primary shard和replica shard身份拉，之类的。

要是master节点宕机了，那么会重新选举一个节点为master节点。

如果是非master节点宕机了，那么会由master节点，让那个宕机节点上的primary shard的身份转移到其他机器上的replica shard。急着你要是修复了那个宕机机器，重启了之后，master节点会控制将缺失的replica shard分配过去，同步后续修改的数据之类的，让集群恢复正常。

其实上述就是elasticsearch作为一个分布式搜索引擎最基本的一个架构设计

### （3）es在数据量很大的情况下（数十亿级别）如何提高查询效率啊？



1、面试题

es在数据量很大的情况下（数十亿级别）如何提高查询效率啊？

2、面试官心里分析

问这个问题，是肯定的，说白了，就是看你有没有实际干过es，因为啥？es说白了其实性能并没有你想象中那么好的。很多时候数据量大了，特别是有几亿条数据的时候，可能你会懵逼的发现，跑个搜索怎么一下5秒~10秒，坑爹了。第一次搜索的时候，是5~10秒，后面反而就快了，可能就几百毫秒。

你就很懵，每个用户第一次访问都会比较慢，比较卡么？

所以你要是没玩儿过es，或者就是自己玩玩儿demo，被问到这个问题容易懵逼，显示出你对es确实玩儿的不怎么样

3、面试题剖析

说实话，es性能优化是没有什么银弹的，啥意思呢？就是不要期待着随手调一个参数，就可以万能的应对所有的性能慢的场景。也许有的场景是你换个参数，或者调整一下语法，就可以搞定，但是绝对不是所有场景都可以这样。

一块一块来分析吧

在这个海量数据的场景下，如何提升es搜索的性能，也是我们之前生产环境实践经验所得

（1）性能优化的杀手锏——filesystem cache

os cache，操作系统的缓存

你往es里写的数据，实际上都写到磁盘文件里去了，磁盘文件里的数据操作系统会自动将里面的数据缓存到os cache里面去

es的搜索引擎严重依赖于底层的filesystem cache，你如果给filesystem cache更多的内存，尽量让内存可以容纳所有的indx segment file索引数据文件，那么你搜索的时候就基本都是走内存的，性能会非常高。

性能差距可以有大，我们之前很多的测试和压测，如果走磁盘一般肯定上秒，搜索性能绝对是秒级别的，1秒，5秒，10秒。但是如果是走filesystem cache，是走纯内存的，那么一般来说性能比走磁盘要高一个数量级，基本上就是毫秒级的，从几毫秒到几百毫秒不等。

之前有个学员，一直在问我，说他的搜索性能，聚合性能，倒排索引，正排索引，磁盘文件，十几秒。。。。

学员的真实案例

比如说，你，es节点有3台机器，每台机器，看起来内存很多，64G，总内存，64 \* 3 = 192g

每台机器给es jvm heap是32G，那么剩下来留给filesystem cache的就是每台机器才32g，总共集群里给filesystem cache的就是32 \* 3 = 96g内存

我就问他，ok，那么就是你往es集群里写入的数据有多少数据量？

如果你此时，你整个，磁盘上索引数据文件，在3台机器上，一共占用了1T的磁盘容量，你的es数据量是1t，每台机器的数据量是300g

你觉得你的性能能好吗？filesystem cache的内存才100g，十分之一的数据可以放内存，其他的都在磁盘，然后你执行搜索操作，大部分操作都是走磁盘，性能肯定差

当时他们的情况就是这样子，es在测试，弄了3台机器，自己觉得还不错，64G内存的物理机。自以为可以容纳1T的数据量。

归根结底，你要让es性能要好，最佳的情况下，就是你的机器的内存，至少可以容纳你的总数据量的一半

比如说，你一共要在es中存储1T的数据，那么你的多台机器留个filesystem cache的内存加起来综合，至少要到512G，至少半数的情况下，搜索是走内存的，性能一般可以到几秒钟，2秒，3秒，5秒

如果最佳的情况下，我们自己的生产环境实践经验，所以说我们当时的策略，是仅仅在es中就存少量的数据，就是你要用来搜索的那些索引，内存留给filesystem cache的，就100G，那么你就控制在100gb以内，相当于是，你的数据几乎全部走内存来搜索，性能非常之高，一般可以在1秒以内

比如说你现在有一行数据

id name age ....30个字段

但是你现在搜索，只需要根据id name age三个字段来搜索

如果你傻乎乎的往es里写入一行数据所有的字段，就会导致说70%的数据是不用来搜索的，结果硬是占据了es机器上的filesystem cache的空间，单挑数据的数据量越大，就会导致filesystem cahce能缓存的数据就越少

仅仅只是写入es中要用来检索的少数几个字段就可以了，比如说，就写入es id name age三个字段就可以了，然后你可以把其他的字段数据存在mysql里面，我们一般是建议用es + hbase的这么一个架构。

hbase的特点是适用于海量数据的在线存储，就是对hbase可以写入海量数据，不要做复杂的搜索，就是做很简单的一些根据id或者范围进行查询的这么一个操作就可以了

从es中根据name和age去搜索，拿到的结果可能就20个doc id，然后根据doc id到hbase里去查询每个doc id对应的完整的数据，给查出来，再返回给前端。

你最好是写入es的数据小于等于，或者是略微大于es的filesystem cache的内存容量

然后你从es检索可能就花费20ms，然后再根据es返回的id去hbase里查询，查20条数据，可能也就耗费个30ms，可能你原来那么玩儿，1T数据都放es，会每次查询都是5~10秒，现在可能性能就会很高，每次查询就是50ms。

elastcisearch减少数据量仅仅放要用于搜索的几个关键字段即可，尽量写入es的数据量跟es机器的filesystem cache是差不多的就可以了；其他不用来检索的数据放hbase里，或者mysql。

所以之前有些学员也是问，我也是跟他们说，尽量在es里，就存储必须用来搜索的数据，比如说你现在有一份数据，有100个字段，其实用来搜索的只有10个字段，建议是将10个字段的数据，存入es，剩下90个字段的数据，可以放mysql，hadoop hbase，都可以

这样的话，es数据量很少，10个字段的数据，都可以放内存，就用来搜索，搜索出来一些id，通过id去mysql，hbase里面去查询明细的数据

（2）数据预热

假如说，哪怕是你就按照上述的方案去做了，es集群中每个机器写入的数据量还是超过了filesystem cache一倍，比如说你写入一台机器60g数据，结果filesystem cache就30g，还是有30g数据留在了磁盘上。

举个例子，就比如说，微博，你可以把一些大v，平时看的人很多的数据给提前你自己后台搞个系统，每隔一会儿，你自己的后台系统去搜索一下热数据，刷到filesystem cache里去，后面用户实际上来看这个热数据的时候，他们就是直接从内存里搜索了，很快。

电商，你可以将平时查看最多的一些商品，比如说iphone 8，热数据提前后台搞个程序，每隔1分钟自己主动访问一次，刷到filesystem cache里去。

对于那些你觉得比较热的，经常会有人访问的数据，最好做一个专门的缓存预热子系统，就是对热数据，每隔一段时间，你就提前访问一下，让数据进入filesystem cache里面去。这样期待下次别人访问的时候，一定性能会好一些。

（3）冷热分离

关于es性能优化，数据拆分，我之前说将大量不搜索的字段，拆分到别的存储中去，这个就是类似于后面我最后要讲的mysql分库分表的垂直拆分。

es可以做类似于mysql的水平拆分，就是说将大量的访问很少，频率很低的数据，单独写一个索引，然后将访问很频繁的热数据单独写一个索引

你最好是将冷数据写入一个索引中，然后热数据写入另外一个索引中，这样可以确保热数据在被预热之后，尽量都让他们留在filesystem os cache里，别让冷数据给冲刷掉。

你看，假设你有6台机器，2个索引，一个放冷数据，一个放热数据，每个索引3个shard

3台机器放热数据index；另外3台机器放冷数据index

然后这样的话，你大量的时候是在访问热数据index，热数据可能就占总数据量的10%，此时数据量很少，几乎全都保留在filesystem cache里面了，就可以确保热数据的访问性能是很高的。

但是对于冷数据而言，是在别的index里的，跟热数据index都不再相同的机器上，大家互相之间都没什么联系了。如果有人访问冷数据，可能大量数据是在磁盘上的，此时性能差点，就10%的人去访问冷数据；90%的人在访问热数据。

（4）document模型设计

有不少同学问我，mysql，有两张表

订单表：id order\_code total\_price

1 测试订单 5000

订单条目表：id order\_id goods\_id purchase\_count price

1 1 1 2 2000

2 1 2 5 200

我在mysql里，都是select \* from order join order\_item on order.id=order\_item.order\_id where order.id=1

1 测试订单 5000 1 1 1 2 2000

1 测试订单 5000 2 1 2 5 200

在es里该怎么玩儿，es里面的复杂的关联查询，复杂的查询语法，尽量别用，一旦用了性能一般都不太好

设计es里的数据模型

写入es的时候，搞成两个索引，order索引，orderItem索引

order索引，里面就包含id order\_code total\_price

orderItem索引，里面写入进去的时候，就完成join操作，id order\_code total\_price id order\_id goods\_id purchase\_count price

写入es的java系统里，就完成关联，将关联好的数据直接写入es中，搜索的时候，就不需要利用es的搜索语法去完成join来搜索了

document模型设计是非常重要的，很多操作，不要在搜索的时候才想去执行各种复杂的乱七八糟的操作。es能支持的操作就是那么多，不要考虑用es做一些它不好操作的事情。如果真的有那种操作，尽量在document模型设计的时候，写入的时候就完成。另外对于一些太复杂的操作，比如join，nested，parent-child搜索都要尽量避免，性能都很差的。

很多同学在问我，很多复杂的乱七八糟的一些操作，如何执行

两个思路，在搜索/查询的时候，要执行一些业务强相关的特别复杂的操作：

1）在写入数据的时候，就设计好模型，加几个字段，把处理好的数据写入加的字段里面

2）自己用java程序封装，es能做的，用es来做，搜索出来的数据，在java程序里面去做，比如说我们，基于es，用java封装一些特别复杂的操作

（5）分页性能优化

es的分页是较坑的，为啥呢？举个例子吧，假如你每页是10条数据，你现在要查询第100页，实际上是会把每个shard上存储的前1000条数据都查到一个协调节点上，如果你有个5个shard，那么就有5000条数据，接着协调节点对这5000条数据进行一些合并、处理，再获取到最终第100页的10条数据。

分布式的，你要查第100页的10条数据，你是不可能说从5个shard，每个shard就查2条数据？最后到协调节点合并成10条数据？你必须得从每个shard都查1000条数据过来，然后根据你的需求进行排序、筛选等等操作，最后再次分页，拿到里面第100页的数据。

你翻页的时候，翻的越深，每个shard返回的数据就越多，而且协调节点处理的时间越长。非常坑爹。所以用es做分页的时候，你会发现越翻到后面，就越是慢。

我们之前也是遇到过这个问题，用es作分页，前几页就几十毫秒，翻到10页之后，几十页的时候，基本上就要5~10秒才能查出来一页数据了

1）不允许深度分页/默认深度分页性能很惨

你系统不允许他翻那么深的页，pm，默认翻的越深，性能就越差

2）类似于app里的推荐商品不断下拉出来一页一页的

类似于微博中，下拉刷微博，刷出来一页一页的，你可以用scroll api，自己百度

scroll会一次性给你生成所有数据的一个快照，然后每次翻页就是通过游标移动，获取下一页下一页这样子，性能会比上面说的那种分页性能也高很多很多

针对这个问题，你可以考虑用scroll来进行处理，scroll的原理实际上是保留一个数据快照，然后在一定时间内，你如果不断的滑动往后翻页的时候，类似于你现在在浏览微博，不断往下刷新翻页。那么就用scroll不断通过游标获取下一页数据，这个性能是很高的，比es实际翻页要好的多的多。

但是唯一的一点就是，这个适合于那种类似微博下拉翻页的，不能随意跳到任何一页的场景。同时这个scroll是要保留一段时间内的数据快照的，你需要确保用户不会持续不断翻页翻几个小时。

无论翻多少页，性能基本上都是毫秒级的

因为scroll api是只能一页一页往后翻的，是不能说，先进入第10页，然后去120页，回到58页，不能随意乱跳页。所以现在很多产品，都是不允许你随意翻页的，app，也有一些网站，做的就是你只能往下拉，一页一页的翻

### es生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？

1、面试题

es生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？

2、面试官心里分析

这个问题，包括后面的redis什么的，谈到es、redis、mysql分库分表等等技术，面试必问！就是你生产环境咋部署的？说白了，这个问题没啥技术含量，就是看你有没有在真正的生产环境里干过这事儿！

有些同学可能是没在生产环境中干过的，没实际去拿线上机器部署过es集群，也没实际玩儿过，也没往es集群里面导入过几千万甚至是几亿的数据量，可能你就不太清楚这里面的一些生产项目中的细节

如果你是自己就玩儿过demo，没碰过真实的es集群，那你可能此时会懵，但是别懵。。。你一定要云淡风轻的回答出来这个问题，表示你确实干过这事儿

3、面试题剖析

其实这个问题没啥，如果你确实干过es，那你肯定了解你们生产es集群的实际情况，部署了几台机器？有多少个索引？每个索引有多大数据量？每个索引给了多少个分片？你肯定知道！

但是如果你确实没干过，也别虚，我给你说一个基本的版本，你到时候就简单说一下就好了

（1）es生产集群我们部署了5台机器，每台机器是6核64G的，集群总内存是320G

（2）我们es集群的日增量数据大概是2000万条，每天日增量数据大概是500MB，每月增量数据大概是6亿，15G。目前系统已经运行了几个月，现在es集群里数据总量大概是100G左右。

（3）目前线上有5个索引（这个结合你们自己业务来，看看自己有哪些数据可以放es的），每个索引的数据量大概是20G，所以这个数据量之内，我们每个索引分配的是8个shard，比默认的5个shard多了3个shard。

大概就这么说一下就行了

18\_总结一下分布式搜索引擎相关问题的面试技巧

消息队列、分布式搜索引擎

其实如果我是面试官的话，我如果感觉你都把刚才那些问题都答出来了，我可能会继续刨根问底，深挖，问你，直到把你给问倒

消息队列，kafka，复制的底层原理，leader选举的算法，增加partition以后的rebalance算法，扣很多很多的细节，如何优化kafka写入的吞吐量

其实这块如果挖深了可以问的极其深，如果是我来深挖，可能会挖到es底层的相关度评分算法（TF/IDF算法）、deep paging、上千万数据批处理、跨机房多集群同步、搜索效果优化，等等吧，很多的实际生产问题。

3种，一种比较水的面试官，他可能还掌握不到我们这个面试突击课程的水准，他其实连我们这个课程里的这些问题都没问到位；比较nice的面试官，问的差不多了，对你还挺欣赏的，基本上就让你过了；比较hard的面试官，干倒，虐你，故意要让你出丑，我不是这种，我只不过严格，我希望招到的是一个最好的这么一个人，我之所以会深挖深挖深挖，我想看看你的极限到底在哪里，你对这么技术掌握的最深的深度在哪儿？如果你进来成为为的下属，我就可以对你的能力各方面都非常的了解

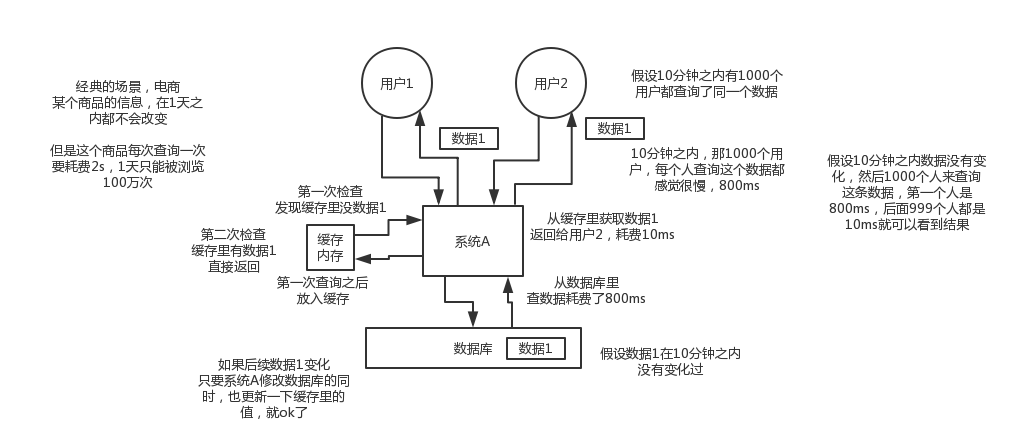
但是。。。既然我们课程定位是2周快速突击，帮助的是那些原本可能连一些基础问题都没法答出来的同学去面试，而且确实现在很多公司面试官问到这块可能也就是问一些基础的问题，那么。。。我们就不忘初心吧

我唯一能说的，就是只要有人跟你聊到es，你可以自己合盘脱出自己对分布式搜索引擎基本原理的一个理解，以及你们在项目中一般是如何优化的，包括你们生产环境是怎么部署的，数据量多大。让别人感觉你这块至少还是正经了解和干过的。

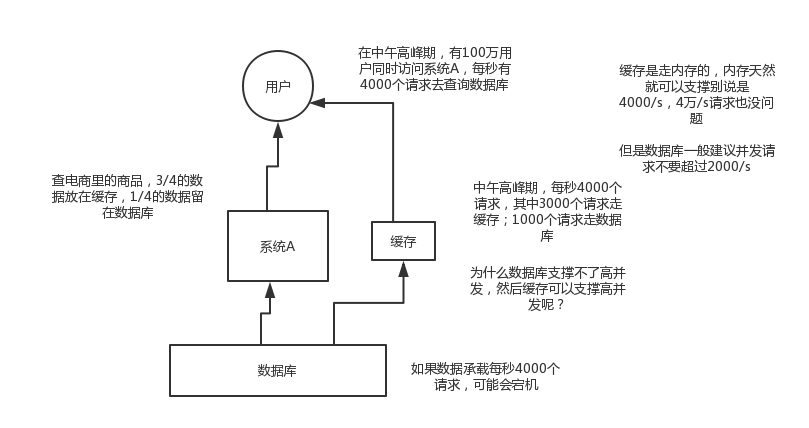
## 2.4 缓存

### （1）在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？

01\_缓存是如何实现高性能的



02\_缓存是如何实现高并发的



1、面试题

在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？

2、面试官心里分析

这个问题，互联网公司必问，要是一个人连缓存都不太清楚，那确实比较尴尬

只要问到缓存，上来第一个问题，肯定能是先问问你项目哪里用了缓存？为啥要用？不用行不行？如果用了以后可能会有什么不良的后果？

这就是看看你对你用缓存这个东西背后，有没有思考，如果你就是傻乎乎的瞎用，没法给面试官一个合理的解答。那我只能说，面试官对你印象肯定不太好，觉得你平时思考太少，就知道干活儿。

3、面试题剖析

一个一个来看

（1）在项目中缓存是如何使用的？

这个，你结合你自己项目的业务来，你如果用了那恭喜你，你如果没用那不好意思，你硬加也得加一个场景吧

（2）为啥在项目里要用缓存呢？

用缓存，主要是俩用途，高性能和高并发

1）高性能

假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作mysql，半天查出来一个结果，耗时600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办？

缓存啊，折腾600ms查出来的结果，扔缓存里，一个key对应一个value，下次再有人查，别走mysql折腾600ms了。直接从缓存里，通过一个key查出来一个value，2ms搞定。性能提升300倍。

这就是所谓的高性能。

就是把你一些复杂操作耗时查出来的结果，如果确定后面不咋变了，然后但是马上还有很多读请求，那么直接结果放缓存，后面直接读缓存就好了。

2）高并发

mysql这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql单机支撑到2000qps也开始容易报警了。

所以要是你有个系统，高峰期一秒钟过来的请求有1万，那一个mysql单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放mysql。缓存功能简单，说白了就是key-value式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发so easy。单机承载并发量是mysql单机的几十倍。

3）所以你要结合这俩场景考虑一下，你为啥要用缓存？

一般很多同学项目里没啥高并发场景，那就别折腾了，直接用高性能那个场景吧，就思考有没有可以缓存结果的复杂查询场景，后续可以大幅度提升性能，优化用户体验，有，就说这个理由，没有？？那你也得编一个出来吧，不然你不是在搞笑么

（3）用了缓存之后会有啥不良的后果？

呵呵。。。你要是没考虑过这个问题，那你就尴尬了，面试官会觉得你头脑简单，四肢也不发达。你别光是傻用一个东西，多考虑考虑背后的一些事儿。

常见的缓存问题有仨（当然其实有很多，我这里就说仨，你能说出来也可以了）

1）缓存与数据库双写不一致

2）缓存雪崩

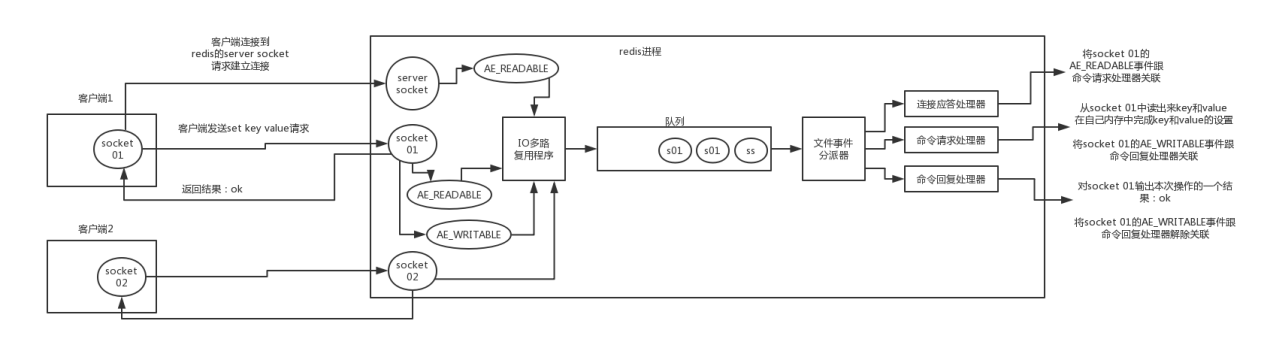
3）缓存穿透

4）缓存并发竞争

这仨问题是常见面试题，后面我要讲，大家看到后面自然就知道了，但是人要是问你，你至少自己能说出来，并且给出对应的解决方案

### （2）redis和memcached有什么区别？redis的线程模型是什么？为什么单线程的redis比多线程的memcached效率要高得多？

01\_redis单线程模型



1、面试题

redis和memcached有什么区别？redis的线程模型是什么？为什么单线程的redis比多线程的memcached效率要高得多（为什么redis是单线程的但是还可以支撑高并发）？

2、面试官心里分析

这个是问redis的时候，最基本的问题吧，redis最基本的一个内部原理和特点，就是redis实际上是个单线程工作模型，你要是这个都不知道，那后面玩儿redis的时候，出了问题岂不是什么都不知道？

还有可能面试官会问问你redis和memcached的区别，不过说实话，最近这两年，我作为面试官都不太喜欢这么问了，memched是早些年各大互联网公司常用的缓存方案，但是现在近几年基本都是redis，没什么公司用memcached了

3、额外的友情提示

同学，你要是现在还不知道redis和memcached是啥？那你赶紧百度一下redis入门和memcahced入门，简单启动一下，然后试一下几个简单操作，先感受一下。接着回来继续听课，我觉得1小时以内你就搞定了。

另外一个友情提示，要听明白redis的线程模型，你需要了解socket网络相关的基本知识，如果不懂。。。那我觉得你java没学好吧。初学者都该学习java的socket网络通信相关知识的。。。

4、面试题剖析

（1）redis和memcached有啥区别

这个事儿吧，你可以比较出N多个区别来，但是我还是采取redis作者给出的几个比较吧

1）Redis支持服务器端的数据操作：Redis相比Memcached来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。

~~2）内存使用效率对比：使用简单的key-value存储的话，Memcached的内存利用率更高，而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached。~~

~~3）性能对比：由于Redis只使用单核，而Memcached可以使用多核，所以平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis，虽然Redis最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。~~

4）集群模式：memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是redis目前是原生支持cluster模式的，redis官方就是支持redis cluster集群模式的，比memcached来说要更好

（2）redis的线程模型

1）文件事件处理器

redis基于reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器，file event handler。这个文件事件处理器，是单线程的，redis才叫做单线程的模型，采用IO多路复用机制同时监听多个socket，根据socket上的事件来选择对应的事件处理器来处理这个事件。

如果被监听的socket准备好执行accept、read、write、close等操作的时候，跟操作对应的文件事件就会产生，这个时候文件事件处理器就会调用之前关联好的事件处理器来处理这个事件。

文件事件处理器是单线程模式运行的，但是通过IO多路复用机制监听多个socket，可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了redis内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个socket，IO多路复用程序，文件事件分派器，事件处理器（命令请求处理器、命令回复处理器、连接应答处理器，等等）。

多个socket可能并发的产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用程序会监听多个socket，但是会将socket放入一个队列中排队，每次从队列中取出一个socket给事件分派器，事件分派器把socket给对应的事件处理器。

然后一个socket的事件处理完之后，IO多路复用程序才会将队列中的下一个socket给事件分派器。文件事件分派器会根据每个socket当前产生的事件，来选择对应的事件处理器来处理。

2）文件事件

当socket变得可读时（比如客户端对redis执行write操作，或者close操作），或者有新的可以应答的sccket出现时（客户端对redis执行connect操作），socket就会产生一个AE\_READABLE事件。

当socket变得可写的时候（客户端对redis执行read操作），socket会产生一个AE\_WRITABLE事件。

IO多路复用程序可以同时监听AE\_REABLE和AE\_WRITABLE两种事件，要是一个socket同时产生了AE\_READABLE和AE\_WRITABLE两种事件，那么文件事件分派器优先处理AE\_REABLE事件，然后才是AE\_WRITABLE事件。

3）文件事件处理器

如果是客户端要连接redis，那么会为socket关联连接应答处理器

如果是客户端要写数据到redis，那么会为socket关联命令请求处理器

如果是客户端要从redis读数据，那么会为socket关联命令回复处理器

4）客户端与redis通信的一次流程

在redis启动初始化的时候，redis会将连接应答处理器跟AE\_READABLE事件关联起来，接着如果一个客户端跟redis发起连接，此时会产生一个AE\_READABLE事件，然后由连接应答处理器来处理跟客户端建立连接，创建客户端对应的socket，同时将这个socket的AE\_READABLE事件跟命令请求处理器关联起来。

当客户端向redis发起请求的时候（不管是读请求还是写请求，都一样），首先就会在socket产生一个AE\_READABLE事件，然后由对应的命令请求处理器来处理。这个命令请求处理器就会从socket中读取请求相关数据，然后进行执行和处理。

接着redis这边准备好了给客户端的响应数据之后，就会将socket的AE\_WRITABLE事件跟命令回复处理器关联起来，当客户端这边准备好读取响应数据时，就会在socket上产生一个AE\_WRITABLE事件，会由对应的命令回复处理器来处理，就是将准备好的响应数据写入socket，供客户端来读取。

命令回复处理器写完之后，就会删除这个socket的AE\_WRITABLE事件和命令回复处理器的关联关系。

（3）为啥redis单线程模型也能效率这么高？

1）纯内存操作

2）核心是基于非阻塞的IO多路复用机制

3）单线程反而避免了多线程的频繁上下文切换问题（百度）

### （3）redis都有哪些数据类型？分别在哪些场景下使用比较合适？

1、面试题

redis都有哪些数据类型？分别在哪些场景下使用比较合适？

2、面试官心里分析

除非是我感觉看你简历，就是工作3年以内的比较初级的一个同学，可能对技术没有很深入的研究过，我才会问这类问题，在宝贵的面试时间里，我实在是不想多问

其实问这个问题呢。。。主要就俩原因

第一，看看你到底有没有全面的了解redis有哪些功能，一般怎么来用，啥场景用什么，就怕你别就会最简单的kv操作

第二，看看你在实际项目里都怎么玩儿过redis

要是你回答的不好，没说出几种数据类型，也没说什么场景，你完了，面试官对你印象肯定不好，觉得你平时就是做个简单的set和get。

3、面试题剖析

（1）string

这是最基本的类型了，没啥可说的，就是普通的set和get，做简单的kv缓存

（2）hash

这个是类似map的一种结构，这个一般就是可以将结构化的数据，比如一个对象（前提是这个对象没嵌套其他的对象）给缓存在redis里，然后每次读写缓存的时候，可以就操作hash里的某个字段。

key=150

value={

“id”: 150,

“name”: “zhangsan”,

“age”: 20

}

hash类的数据结构，主要是用来存放一些对象，把一些简单的对象给缓存起来，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值

value={

“id”: 150,

“name”: “zhangsan”,

“age”: 21

}

（3）list

有序列表，这个是可以玩儿出很多花样的

微博，某个大v的粉丝，就可以以list的格式放在redis里去缓存

key=某大v

value=[zhangsan, lisi, wangwu]

比如可以通过list存储一些列表型的数据结构，类似粉丝列表了、文章的评论列表了之类的东西

比如可以通过lrange命令，就是从某个元素开始读取多少个元素，可以基于list实现分页查询，这个很棒的一个功能，基于redis实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西，性能高，就一页一页走

比如可以搞个简单的消息队列，从list头怼进去，从list尾巴那里弄出来

（4）set

无序集合，自动去重

直接基于set将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于jvm内存里的HashSet进行去重，但是如果你的某个系统部署在多台机器上呢？

得基于redis进行全局的set去重

可以基于set玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁？对吧

把两个大v的粉丝都放在两个set中，对两个set做交集

（5）sorted set

排序的set，去重但是可以排序，写进去的时候给一个分数，自动根据分数排序，这个可以玩儿很多的花样，最大的特点是有个分数可以自定义排序规则

比如说你要是想根据时间对数据排序，那么可以写入进去的时候用某个时间作为分数，人家自动给你按照时间排序了

排行榜：将每个用户以及其对应的什么分数写入进去，zadd board score username，接着zrevrange board 0 99，就可以获取排名前100的用户；zrank board username，可以看到用户在排行榜里的排名

zadd board 85 zhangsan

zadd board 72 wangwu

zadd board 96 lisi

zadd board 62 zhaoliu

96 lisi

85 zhangsan

72 wangwu

62 zhaoliu

zrevrange board 0 3

获取排名前3的用户

96 lisi

85 zhangsan

72 wangwu

zrank board zhaoliu

4

### （5）redis的过期策略都有哪些？手写一下LRU代码实现？

1、面试题

redis的过期策略都有哪些？内存淘汰机制都有哪些？手写一下LRU代码实现？

2、面试官心里分析

1）老师啊，我往redis里写的数据怎么没了？

之前有同学问过我，说我们生产环境的redis怎么经常会丢掉一些数据？写进去了，过一会儿可能就没了。我的天，同学，你问这个问题就说明redis你就没用对啊。redis是缓存，你给当存储了是吧？

啥叫缓存？用内存当缓存。内存是无限的吗，内存是很宝贵而且是有限的，磁盘是廉价而且是大量的。可能一台机器就几十个G的内存，但是可以有几个T的硬盘空间。redis主要是基于内存来进行高性能、高并发的读写操作的。

那既然内存是有限的，比如redis就只能用10个G，你要是往里面写了20个G的数据，会咋办？当然会干掉10个G的数据，然后就保留10个G的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了。

所以说，这是缓存的一个最基本的概念，数据是会过期的，要么是你自己设置个过期时间，要么是redis自己给干掉。

set key value 过期时间（1小时）

set进去的key，1小时之后就没了，就失效了

2）老师，我的数据明明都过期了，怎么还占用着内存啊？

还有一种就是如果你设置好了一个过期时间，你知道redis是怎么给你弄成过期的吗？什么时候删除掉？如果你不知道，之前有个学员就问了，为啥好多数据明明应该过期了，结果发现redis内存占用还是很高？那是因为你不知道redis是怎么删除那些过期key的。

redis 内存一共是10g，你现在往里面写了5g的数据，结果这些数据明明你都设置了过期时间，要求这些数据1小时之后都会过期，结果1小时之后，你回来一看，redis机器，怎么内存占用还是50%呢？5g数据过期了，我从redis里查，是查不到了，结果过期的数据还占用着redis的内存。

如果你连这个问题都不知道，上来就懵了，回答不出来，那线上你写代码的时候，想当然的认为写进redis的数据就一定会存在，后面导致系统各种漏洞和bug，谁来负责？

3、面试题剖析

（1）设置过期时间

我们set key的时候，都可以给一个expire time，就是过期时间，指定这个key比如说只能存活1个小时？10分钟？这个很有用，我们自己可以指定缓存到期就失效。

如果假设你设置一个一批key只能存活1个小时，那么接下来1小时后，redis是怎么对这批key进行删除的？

答案是：定期删除+惰性删除

所谓定期删除，指的是redis默认是每隔100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。假设redis里放了10万个key，都设置了过期时间，你每隔几百毫秒，就检查10万个key，那redis基本上就死了，cpu负载会很高的，消耗在你的检查过期key上了。注意，这里可不是每隔100ms就遍历所有的设置过期时间的key，那样就是一场性能上的灾难。实际上redis是每隔100ms随机抽取一些key来检查和删除的。

但是问题是，定期删除可能会导致很多过期key到了时间并没有被删除掉，那咋整呢？所以就是惰性删除了。这就是说，在你获取某个key的时候，redis会检查一下 ，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除，不会给你返回任何东西。

并不是key到时间就被删除掉，而是你查询这个key的时候，redis再懒惰的检查一下

通过上述两种手段结合起来，保证过期的key一定会被干掉。

很简单，就是说，你的过期key，靠定期删除没有被删除掉，还停留在内存里，占用着你的内存呢，除非你的系统去查一下那个key，才会被redis给删除掉。

但是实际上这还是有问题的，如果定期删除漏掉了很多过期key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期key堆积在内存里，导致redis内存块耗尽了，咋整？

答案是：走内存淘汰机制。

（2）内存淘汰

如果redis的内存占用过多的时候，此时会进行内存淘汰，有如下一些策略：

redis 10个key，现在已经满了，redis需要删除掉5个key

1个key，最近1分钟被查询了100次

1个key，最近10分钟被查询了50次

1个key，最近1个小时倍查询了1次

1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了

2）allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key（这个是最常用的）

3）allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的key给干掉啊

4）volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key（这个一般不太合适）

5）volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key

6）volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除

百度，问题啊，网上鱼龙混杂

如果百度一些api操作，入门的知识，ok的，随便找一个博客都可以

一些高级别的，redis单线程模型

很简单，你写的数据太多，内存满了，或者触发了什么条件，redis lru，自动给你清理掉了一些最近很少使用的数据

（3）要不你手写一个LRU算法？

我确实有时会问这个，因为有些候选人如果确实过五关斩六将，前面的问题都答的很好，那么其实让他写一下LRU算法，可以考察一下编码功底

你可以现场手写最原始的LRU算法，那个代码量太大了，我觉得不太现实

public class LRUCache<K, V> extends LinkedHashMap<K, V> {

private final int CACHE\_SIZE;

// 这里就是传递进来最多能缓存多少数据

public LRUCache(int cacheSize) {

super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true); // 这块就是设置一个hashmap的初始大小，同时最后一个true指的是让linkedhashmap按照访问顺序来进行排序，最近访问的放在头，最老访问的就在尾

CACHE\_SIZE = cacheSize;

}

@Override

protected boolean removeEldestEntry(Map.Entry eldest) {

return size() > CACHE\_SIZE; // 这个意思就是说当map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据

}

}

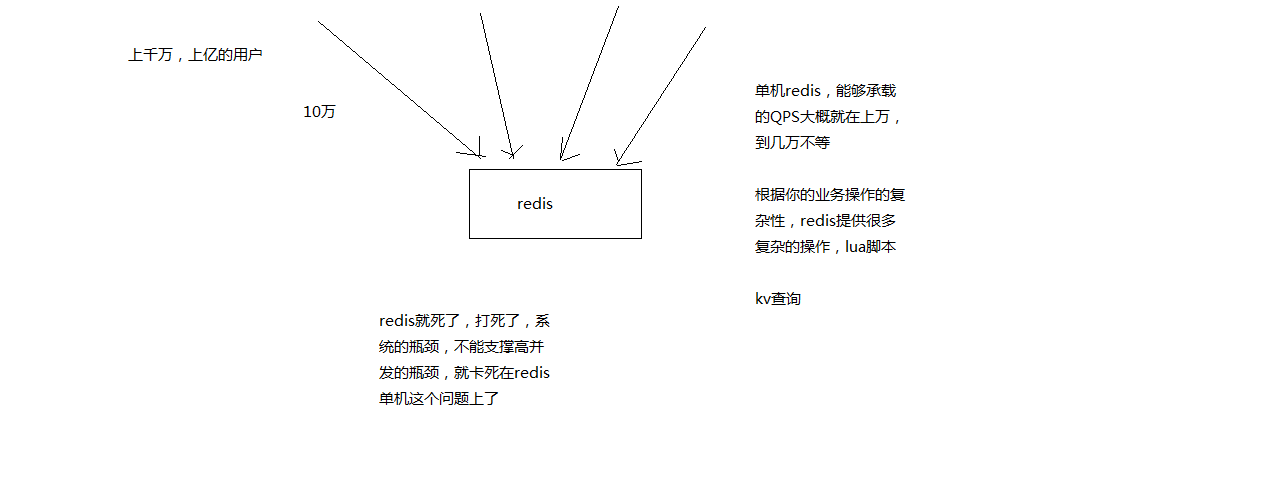
我给你看上面的代码，是告诉你最起码你也得写出来上面那种代码，不求自己纯手工从底层开始打造出自己的LRU，但是起码知道如何利用已有的jdk数据结构实现一个java版的LRU

### （6）如何保证Redis高并发、高可用、持久化？redis的主从复制原理能介绍一下么？redis的哨兵原理能介绍一下么？

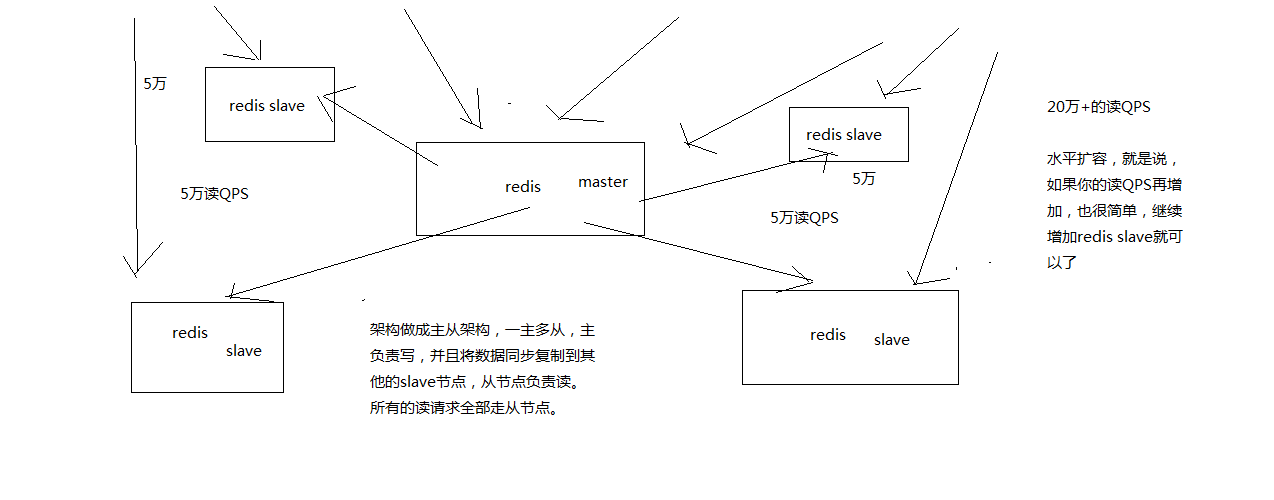
##### 01\_面试题以及解答思路介绍

##### 02\_redis如何通过读写分离来承载读请求QPS超过10万+？

redis单机的瓶颈



redis主从实现读写分离支撑10万+的高并发



1、redis高并发跟整个系统的高并发之间的关系

redis，你要搞高并发的话，不可避免，要把底层的缓存搞得很好

mysql，高并发，做到了，那么也是通过一系列复杂的分库分表，订单系统，事务要求的，QPS到几万，比较高了

要做一些电商的商品详情页，真正的超高并发，QPS上十万，甚至是百万，一秒钟百万的请求量

光是redis是不够的，但是redis是整个大型的缓存架构中，支撑高并发的架构里面，非常重要的一个环节

首先，你的底层的缓存中间件，缓存系统，必须能够支撑的起我们说的那种高并发，其次，再经过良好的整体的缓存架构的设计（多级缓存架构、热点缓存），支撑真正的上十万，甚至上百万的高并发

2、redis不能支撑高并发的瓶颈在哪里？

单机

3、如果redis要支撑超过10万+的并发，那应该怎么做？

单机的redis几乎不太可能说QPS超过10万+，除非一些特殊情况，比如你的机器性能特别好，配置特别高，物理机，维护做的特别好，而且你的整体的操作不是太复杂

单机在几万

读写分离，一般来说，对缓存，一般都是用来支撑读高并发的，写的请求是比较少的，可能写请求也就一秒钟几千，一两千

大量的请求都是读，一秒钟二十万次读

读写分离

主从架构 -> 读写分离 -> 支撑10万+读QPS的架构

4、接下来要讲解的一个topic

redis replication

redis主从架构 -> 读写分离架构 -> 可支持水平扩展的读高并发架构

##### 03\_redis replication以及master持久化对主从架构的安全意义

课程大纲

1、图解redis replication基本原理

2、redis replication的核心机制

3、master持久化对于主从架构的安全保障的意义

redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

redis replication的最最基本的原理，铺垫

------------------------------------------------------------------------

1、图解redis replication基本原理

------------------------------------------------------------------------

2、redis replication的核心机制

（1）redis采用异步方式复制数据到slave节点，不过redis 2.8开始，slave node会周期性地确认自己每次复制的数据量

（2）一个master node是可以配置多个slave node的

（3）slave node也可以连接其他的slave node

（4）slave node做复制的时候，是不会block master node的正常工作的

（5）slave node在做复制的时候，也不会block对自己的查询操作，它会用旧的数据集来提供服务; 但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了

（6）slave node主要用来进行横向扩容，做读写分离，扩容的slave node可以提高读的吞吐量

slave，高可用性，有很大的关系

------------------------------------------------------------------------

3、master持久化对于主从架构的安全保障的意义

如果采用了主从架构，那么建议必须开启master node的持久化！

不建议用slave node作为master node的数据热备，因为那样的话，如果你关掉master的持久化，可能在master宕机重启的时候数据是空的，然后可能一经过复制，salve node数据也丢了

master -> RDB和AOF都关闭了 -> 全部在内存中

master宕机，重启，是没有本地数据可以恢复的，然后就会直接认为自己IDE数据是空的

master就会将空的数据集同步到slave上去，所有slave的数据全部清空

100%的数据丢失

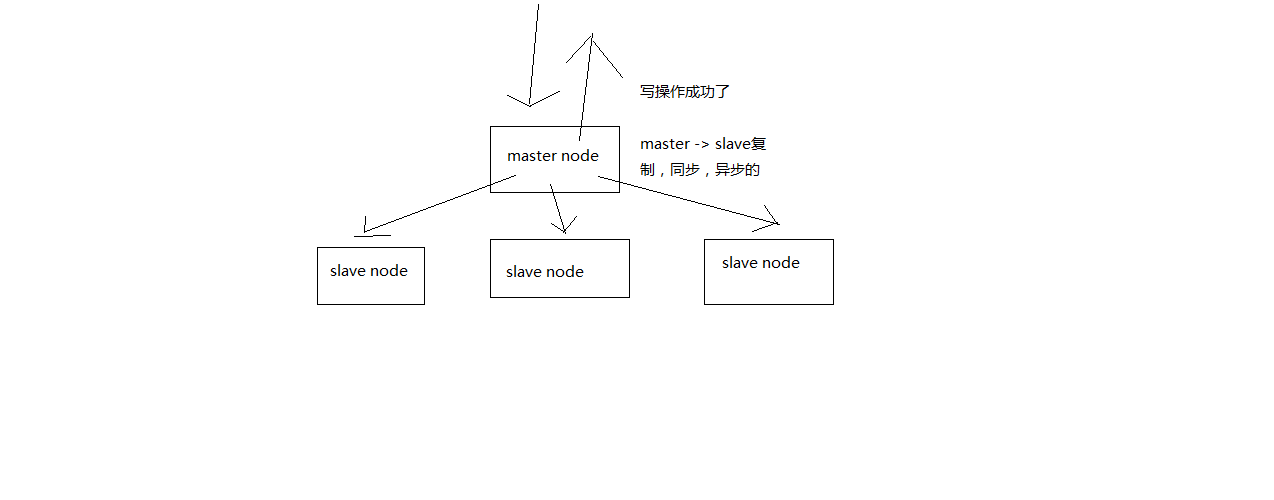
master节点，必须要使用持久化机制

第二个，master的各种备份方案，要不要做，万一说本地的所有文件丢失了; 从备份中挑选一份rdb去恢复master; 这样才能确保master启动的时候，是有数据的

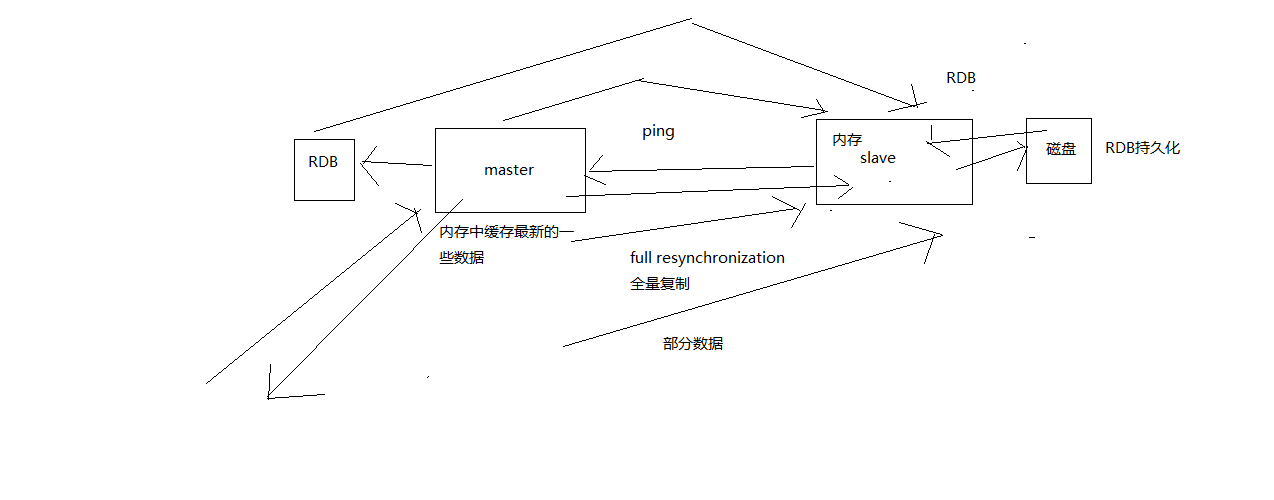
即使采用了后续讲解的高可用机制，slave node可以自动接管master node，但是也可能sentinal还没有检测到master failure，master node就自动重启了，还是可能导致上面的所有slave node数据清空故障

##### 04\_redis主从复制原理、断点续传、无磁盘化复制、过期key处理

redis replica最最基本的原理



redis主从复制的原理



课程大纲

1、主从架构的核心原理

当启动一个slave node的时候，它会发送一个PSYNC命令给master node

如果这是slave node重新连接master node，那么master node仅仅会复制给slave部分缺少的数据; 否则如果是slave node第一次连接master node，那么会触发一次full resynchronization

开始full resynchronization的时候，master会启动一个后台线程，开始生成一份RDB快照文件，同时还会将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后，master会将这个RDB发送给slave，slave会先写入本地磁盘，然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave，slave也会同步这些数据。

slave node如果跟master node有网络故障，断开了连接，会自动重连。master如果发现有多个slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

2、主从复制的断点续传

从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份

master node会在内存中常见一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset，那么就会执行一次resynchronization

3、无磁盘化复制

master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了

repl-diskless-sync

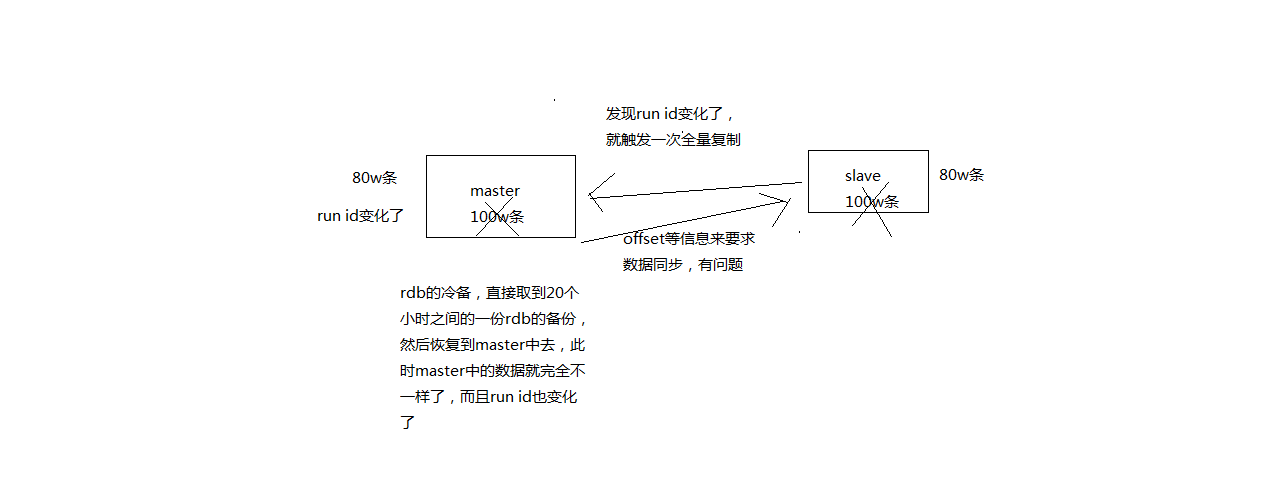
repl-diskless-sync-delay，等待一定时长再开始复制，因为要等更多slave重新连接过来

4、过期key处理

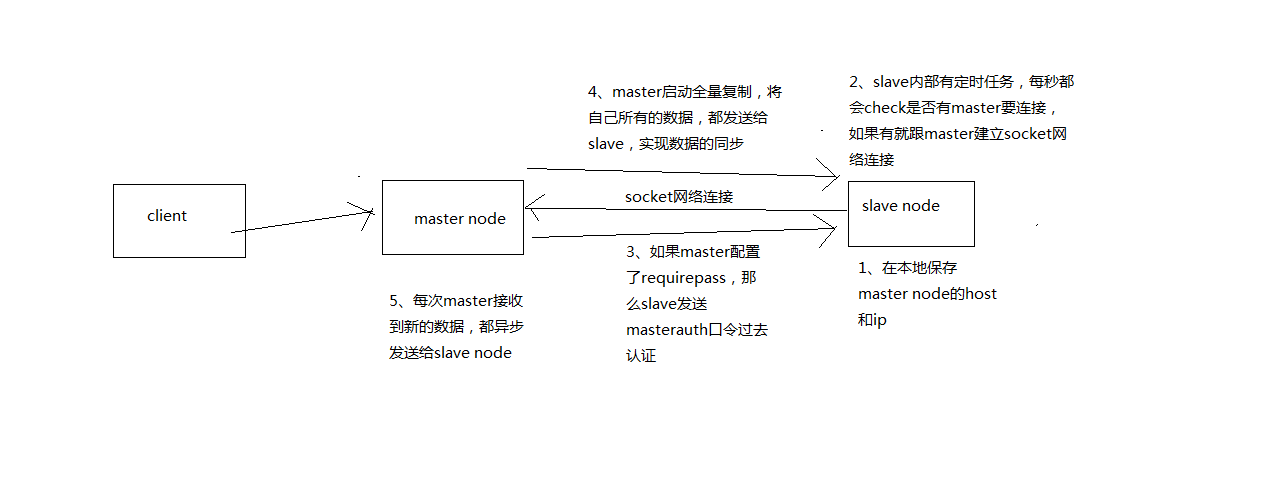
slave不会过期key，只会等待master过期key。如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。

##### 05\_redis replication的完整流运行程和原理的再次深入剖析

maste run id的作用



复制的完整的基本流程



1、复制的完整流程

（1）slave node启动，仅仅保存master node的信息，包括master node的host和ip，但是复制流程没开始

master host和ip是从哪儿来的，redis.conf里面的slaveof配置的

（2）slave node内部有个定时任务，每秒检查是否有新的master node要连接和复制，如果发现，就跟master node建立socket网络连接

（3）slave node发送ping命令给master node

（4）口令认证，如果master设置了requirepass，那么salve node必须发送masterauth的口令过去进行认证

（5）master node第一次执行全量复制，将所有数据发给slave node

（6）master node后续持续将写命令，异步复制给slave node

2、数据同步相关的核心机制

指的就是第一次slave连接msater的时候，执行的全量复制，那个过程里面你的一些细节的机制

（1）master和slave都会维护一个offset

master会在自身不断累加offset，slave也会在自身不断累加offset

slave每秒都会上报自己的offset给master，同时master也会保存每个slave的offset

这个倒不是说特定就用在全量复制的，主要是master和slave都要知道各自的数据的offset，才能知道互相之间的数据不一致的情况

（2）backlog

master node有一个backlog，默认是1MB大小

master node给slave node复制数据时，也会将数据在backlog中同步写一份

backlog主要是用来做全量复制中断候的增量复制的

（3）master run id

info server，可以看到master run id

如果根据host+ip定位master node，是不靠谱的，如果master node重启或者数据出现了变化，那么slave node应该根据不同的run id区分，run id不同就做全量复制

如果需要不更改run id重启redis，可以使用redis-cli debug reload命令

（4）psync

从节点使用psync从master node进行复制，psync runid offset

master node会根据自身的情况返回响应信息，可能是FULLRESYNC runid offset触发全量复制，可能是CONTINUE触发增量复制

3、全量复制

（1）master执行bgsave，在本地生成一份rdb快照文件

（2）master node将rdb快照文件发送给salve node，如果rdb复制时间超过60秒（repl-timeout），那么slave node就会认为复制失败，可以适当调节大这个参数

（3）对于千兆网卡的机器，一般每秒传输100MB，6G文件，很可能超过60s

（4）master node在生成rdb时，会将所有新的写命令缓存在内存中，在salve node保存了rdb之后，再将新的写命令复制给salve node

（5）client-output-buffer-limit slave 256MB 64MB 60，如果在复制期间，内存缓冲区持续消耗超过64MB，或者一次性超过256MB，那么停止复制，复制失败

（6）slave node接收到rdb之后，清空自己的旧数据，然后重新加载rdb到自己的内存中，同时基于旧的数据版本对外提供服务

（7）如果slave node开启了AOF，那么会立即执行BGREWRITEAOF，重写AOF

rdb生成、rdb通过网络拷贝、slave旧数据的清理、slave aof rewrite，很耗费时间

如果复制的数据量在4G~6G之间，那么很可能全量复制时间消耗到1分半到2分钟

4、增量复制

（1）如果全量复制过程中，master-slave网络连接断掉，那么salve重新连接master时，会触发增量复制

（2）master直接从自己的backlog中获取部分丢失的数据，发送给slave node，默认backlog就是1MB

（3）msater就是根据slave发送的psync中的offset来从backlog中获取数据的

5、heartbeat

主从节点互相都会发送heartbeat信息

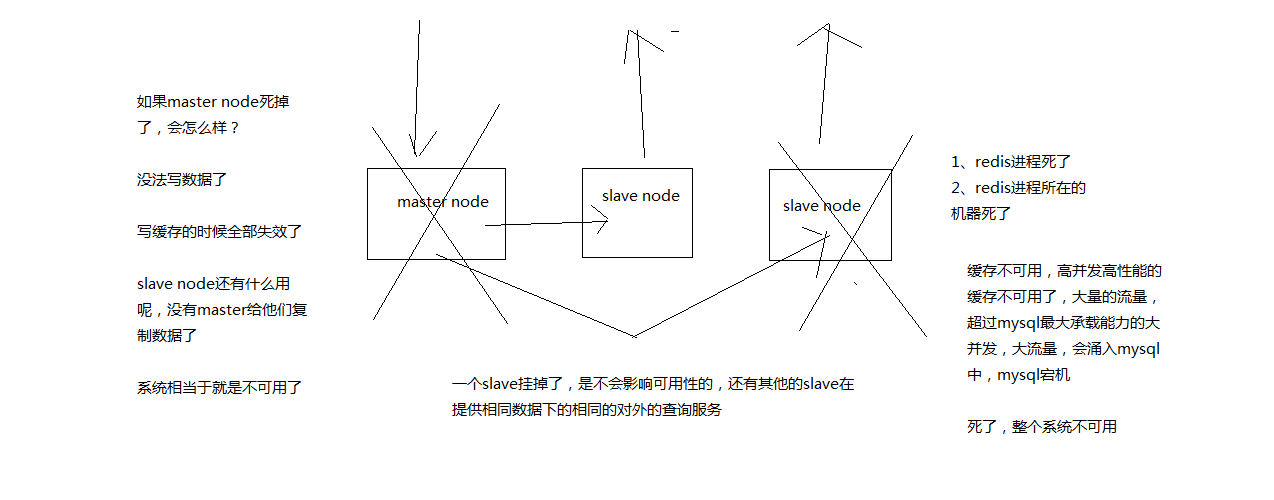
master默认每隔10秒发送一次heartbeat，salve node每隔1秒发送一个heartbeat

6、异步复制

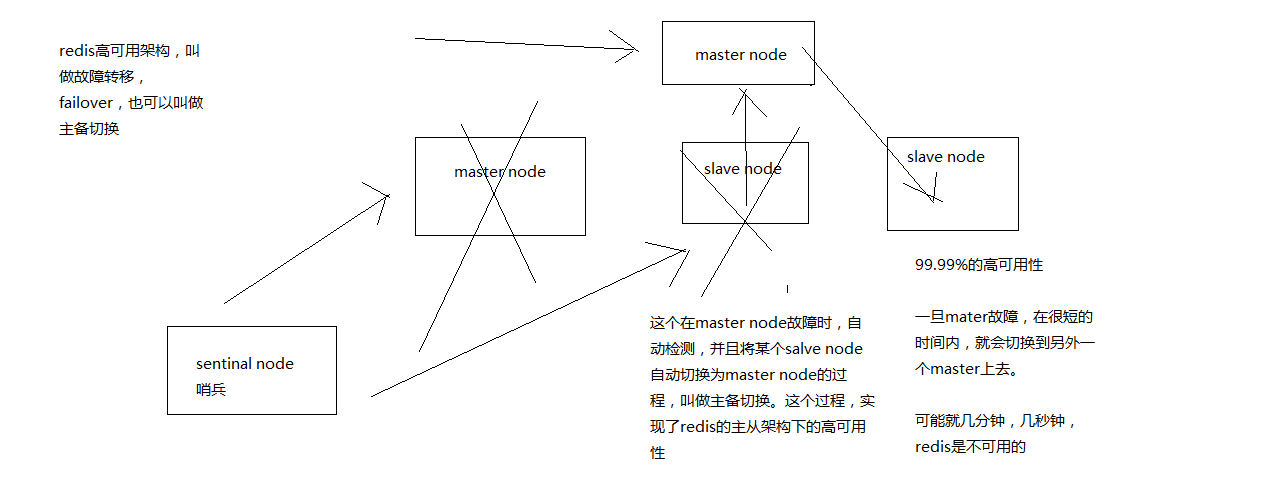
master每次接收到写命令之后，现在内部写入数据，然后异步发送给slave node

##### 06\_redis主从架构下如何才能做到99.99%的高可用性？

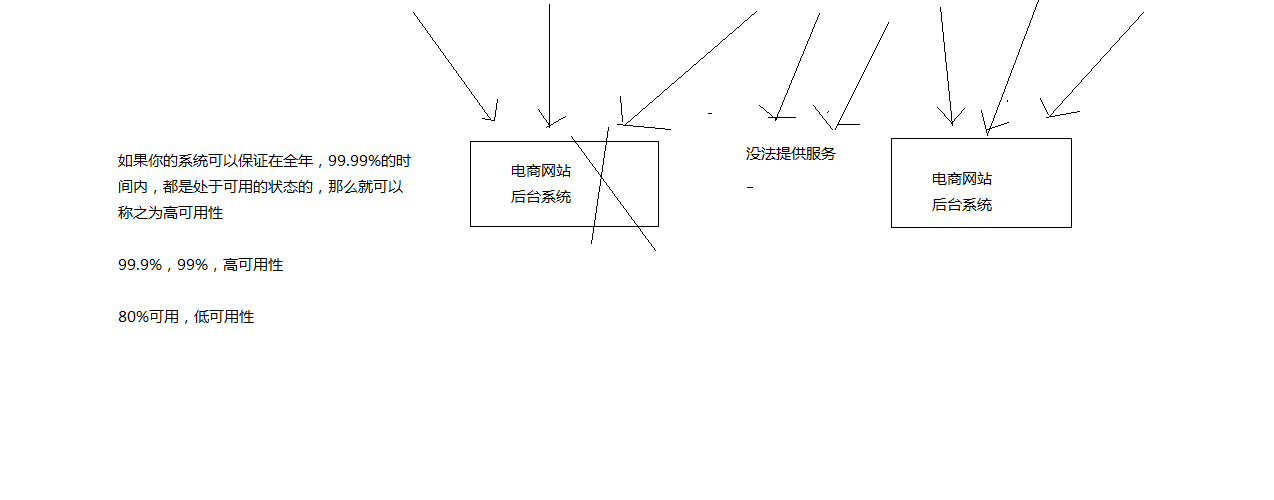
redis的不可用



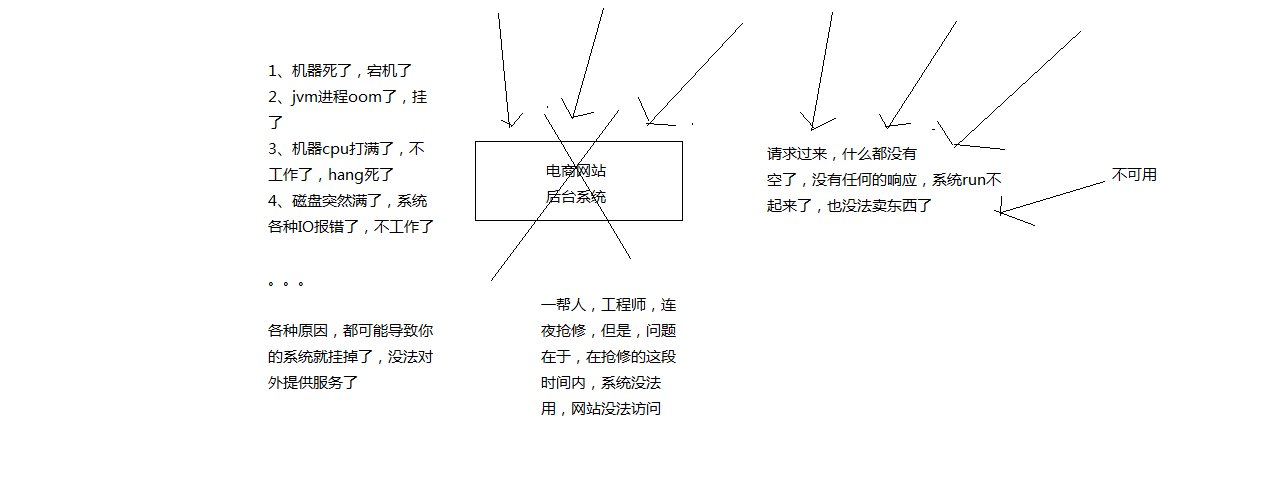
redis基于哨兵的高可用性



什么是99.99%高可用性



系统处于不可用是什么意思



1、什么是99.99%高可用？

架构上，高可用性，99.99%的高可用性

讲的学术，99.99%，公式，系统可用的时间 / 系统故障的时间，365天，在365天 \* 99.99%的时间内，你的系统都是可以哗哗对外提供服务的，那就是高可用性，99.99%

系统可用的时间 / 总的时间 = 高可用性，然后会对各种时间的概念，说一大堆解释

2、redis不可用是什么？单实例不可用？主从架构不可用？不可用的后果是什么？

3、redis怎么才能做到高可用？

##### 07\_redis哨兵架构的相关基础知识的讲解

1、哨兵的介绍

sentinal，中文名是哨兵

哨兵是redis集群架构中非常重要的一个组件，主要功能如下

（1）集群监控，负责监控redis master和slave进程是否正常工作

（2）消息通知，如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员

（3）故障转移，如果master node挂掉了，会自动转移到slave node上

（4）配置中心，如果故障转移发生了，通知client客户端新的master地址

哨兵本身也是分布式的，作为一个哨兵集群去运行，互相协同工作

（1）故障转移时，判断一个master node是宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题

（2）即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了

目前采用的是sentinal 2版本，sentinal 2相对于sentinal 1来说，重写了很多代码，主要是让故障转移的机制和算法变得更加健壮和简单

2、哨兵的核心知识

（1）哨兵至少需要3个实例，来保证自己的健壮性

（2）哨兵 + redis主从的部署架构，是不会保证数据零丢失的，只能保证redis集群的高可用性

（3）对于哨兵 + redis主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练

3、为什么redis哨兵集群只有2个节点无法正常工作？

哨兵集群必须部署2个以上节点

如果哨兵集群仅仅部署了个2个哨兵实例，quorum=1

+----+ +----+

| M1 |---------| R1 |

| S1 | | S2 |

+----+ +----+

Configuration: quorum = 1

master宕机，s1和s2中只要有1个哨兵认为master宕机就可以还行切换，同时s1和s2中会选举出一个哨兵来执行故障转移

同时这个时候，需要majority，也就是大多数哨兵都是运行的，2个哨兵的majority就是2（2的majority=2，3的majority=2，5的majority=3，4的majority=2），2个哨兵都运行着，就可以允许执行故障转移

但是如果整个M1和S1运行的机器宕机了，那么哨兵只有1个了，此时就没有majority来允许执行故障转移，虽然另外一台机器还有一个R1，但是故障转移不会执行

4、经典的3节点哨兵集群

+----+

| M1 |

| S1 |

+----+

|

+----+ | +----+

| R2 |----+----| R3 |

| S2 | | S3 |

+----+ +----+

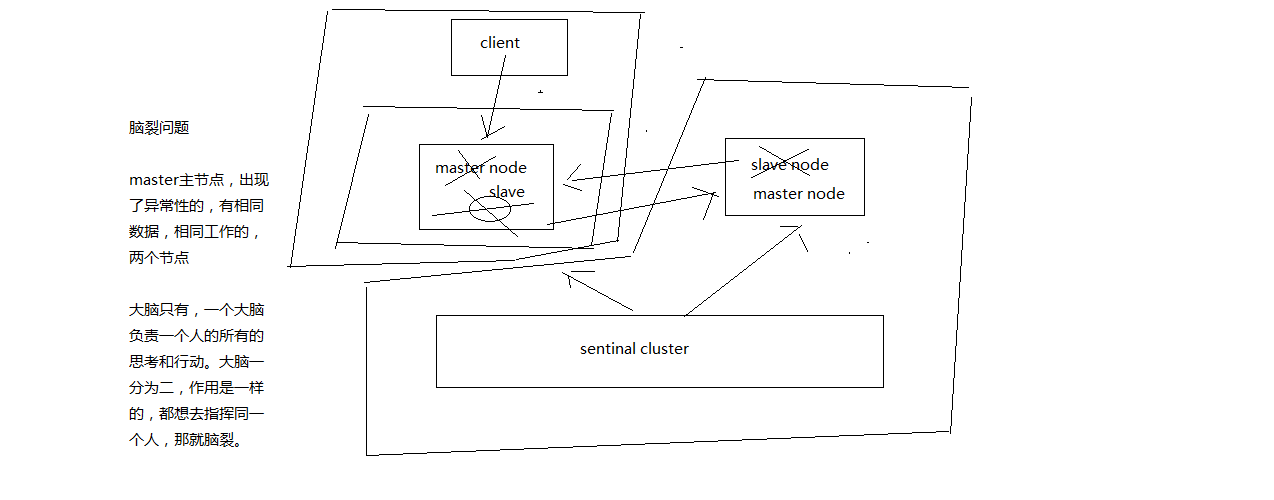
Configuration: quorum = 2，majority

如果M1所在机器宕机了，那么三个哨兵还剩下2个，S2和S3可以一致认为master宕机，然后选举出一个来执行故障转移

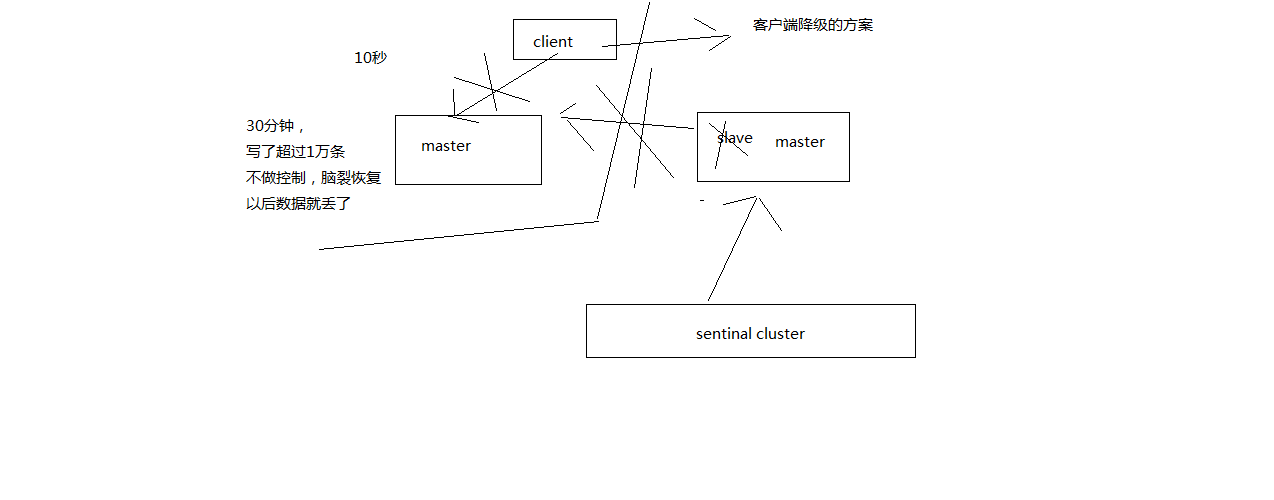
同时3个哨兵的majority是2，所以还剩下的2个哨兵运行着，就可以允许执行故障转移

##### 08\_redis哨兵主备切换的数据丢失问题：异步复制、集群脑裂

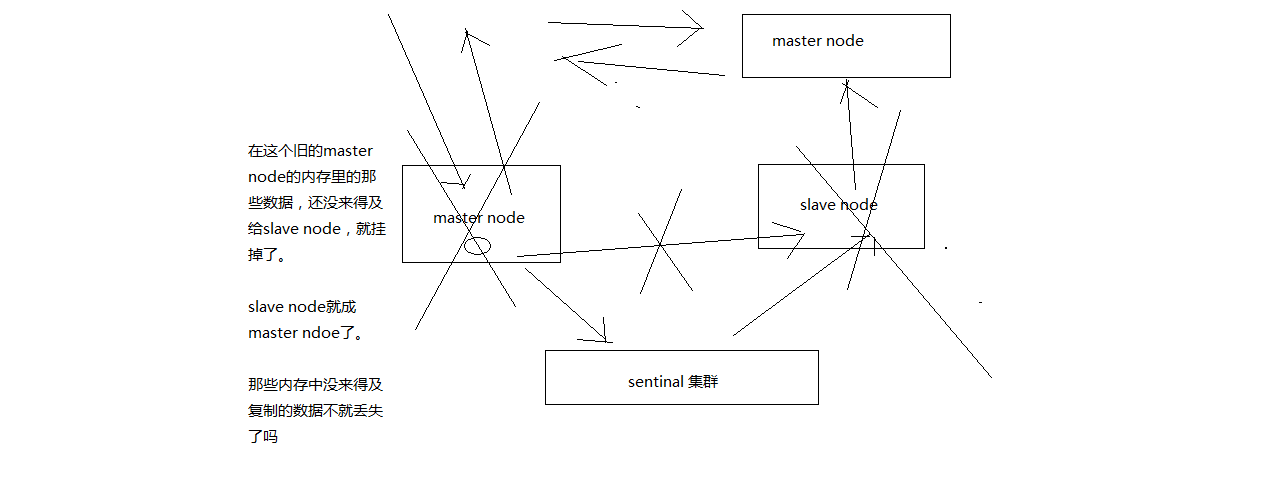
集群脑裂导致的数据丢失问题



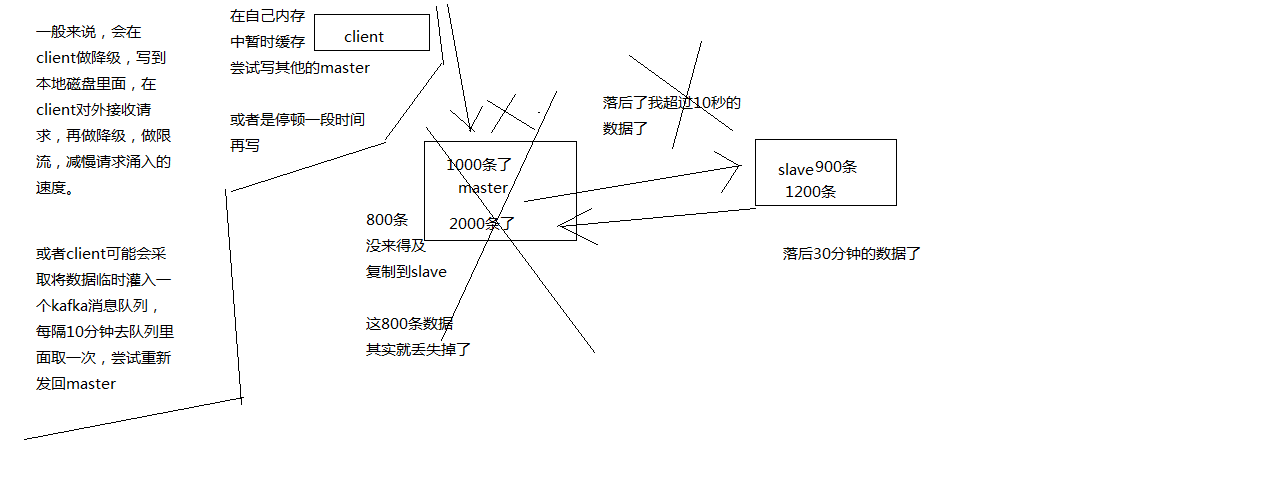
脑裂导致数据丢失的问题如何降低损失



异步复制导致的数据丢失问题



异步复制导致数据丢失如何降低损失



课程大纲

1、两种数据丢失的情况

2、解决异步复制和脑裂导致的数据丢失

------------------------------------------------------------------

1、两种数据丢失的情况

主备切换的过程，可能会导致数据丢失

（1）异步复制导致的数据丢失

因为master -> slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，此时这些部分数据就丢失了

（2）脑裂导致的数据丢失

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着

此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master

这个时候，集群里就会有两个master，也就是所谓的脑裂

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master的数据可能也丢失了

因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据

------------------------------------------------------------------

2、解决异步复制和脑裂导致的数据丢失

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了

上面两个配置可以减少异步复制和脑裂导致的数据丢失

（1）减少异步复制的数据丢失

有了min-slaves-max-lag这个配置，就可以确保说，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内

（2）减少脑裂的数据丢失

如果一个master出现了脑裂，跟其他slave丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的slave发送数据，而且slave超过10秒没有给自己ack消息，那么就直接拒绝客户端的写请求

这样脑裂后的旧master就不会接受client的新数据，也就避免了数据丢失

上面的配置就确保了，如果跟任何一个slave丢了连接，在10秒后发现没有slave给自己ack，那么就拒绝新的写请求

因此在脑裂场景下，最多就丢失10秒的数据

##### 09\_redis哨兵的多个核心底层原理的深入解析（包含slave选举算法）

1、sdown和odown转换机制

sdown和odown两种失败状态

sdown是主观宕机，就一个哨兵如果自己觉得一个master宕机了，那么就是主观宕机

odown是客观宕机，如果quorum数量的哨兵都觉得一个master宕机了，那么就是客观宕机

sdown达成的条件很简单，如果一个哨兵ping一个master，超过了is-master-down-after-milliseconds指定的毫秒数之后，就主观认为master宕机

sdown到odown转换的条件很简单，如果一个哨兵在指定时间内，收到了quorum指定数量的其他哨兵也认为那个master是sdown了，那么就认为是odown了，客观认为master宕机

2、哨兵集群的自动发现机制

哨兵互相之间的发现，是通过redis的pub/sub系统实现的，每个哨兵都会往\_\_sentinel\_\_:hello这个channel里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在

每隔两秒钟，每个哨兵都会往自己监控的某个master+slaves对应的\_\_sentinel\_\_:hello channel里发送一个消息，内容是自己的host、ip和runid还有对这个master的监控配置

每个哨兵也会去监听自己监控的每个master+slaves对应的\_\_sentinel\_\_:hello channel，然后去感知到同样在监听这个master+slaves的其他哨兵的存在

每个哨兵还会跟其他哨兵交换对master的监控配置，互相进行监控配置的同步

3、slave配置的自动纠正

哨兵会负责自动纠正slave的一些配置，比如slave如果要成为潜在的master候选人，哨兵会确保slave在复制现有master的数据; 如果slave连接到了一个错误的master上，比如故障转移之后，那么哨兵会确保它们连接到正确的master上

4、slave->master选举算法

如果一个master被认为odown了，而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave来

会考虑slave的一些信息

（1）跟master断开连接的时长

（2）slave优先级

（3）复制offset

（4）run id

如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕机的时长，那么slave就被认为不适合选举为master

(down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state

接下来会对slave进行排序

（1）按照slave优先级进行排序，slave priority越低，优先级就越高

（2）如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后，优先级就越高

（3）如果上面两个条件都相同，那么选择一个run id比较小的那个slave

5、quorum和majority

每次一个哨兵要做主备切换，首先需要quorum数量的哨兵认为odown，然后选举出一个哨兵来做切换，这个哨兵还得得到majority哨兵的授权，才能正式执行切换

如果quorum < majority，比如5个哨兵，majority就是3，quorum设置为2，那么就3个哨兵授权就可以执行切换

但是如果quorum >= majority，那么必须quorum数量的哨兵都授权，比如5个哨兵，quorum是5，那么必须5个哨兵都同意授权，才能执行切换

6、configuration epoch

哨兵会对一套redis master+slave进行监控，有相应的监控的配置

执行切换的那个哨兵，会从要切换到的新master（salve->master）那里得到一个configuration epoch，这就是一个version号，每次切换的version号都必须是唯一的

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待failover-timeout时间，然后接替继续执行切换，此时会重新获取一个新的configuration epoch，作为新的version号

7、configuraiton传播

哨兵完成切换之后，会在自己本地更新生成最新的master配置，然后同步给其他的哨兵，就是通过之前说的pub/sub消息机制

这里之前的version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的

其他的哨兵都是根据版本号的大小来更新自己的master配置的

##### 总结

1、面试题

如何保证Redis的高并发和高可用？redis的主从复制原理能介绍一下么？redis的哨兵原理能介绍一下么？

2、面试官心里分析

其实问这个问题，主要是考考你，redis单机能承载多高并发？如果单机扛不住如何扩容抗更多的并发？redis会不会挂？既然redis会挂那怎么保证redis是高可用的？

其实针对的都是项目中你肯定要考虑的一些问题，如果你没考虑过，那确实你对生产系统中的问题思考太少。

3、面试题剖析

就是如果你用redis缓存技术的话，肯定要考虑如何用redis来加多台机器，保证redis是高并发的，还有就是如何让Redis保证自己不是挂掉以后就直接死掉了，redis高可用

我这里会选用我之前讲解过这一块内容，redis高并发、高可用、缓存一致性

redis高并发：主从架构，一主多从，一般来说，很多项目其实就足够了，单主用来写入数据，单机几万QPS，多从用来查询数据，多个从实例可以提供每秒10万的QPS。

redis高并发的同时，还需要容纳大量的数据：一主多从，每个实例都容纳了完整的数据，比如redis主就10G的内存量，其实你就最对只能容纳10g的数据量。如果你的缓存要容纳的数据量很大，达到了几十g，甚至几百g，或者是几t，那你就需要redis集群，而且用redis集群之后，可以提供可能每秒几十万的读写并发。

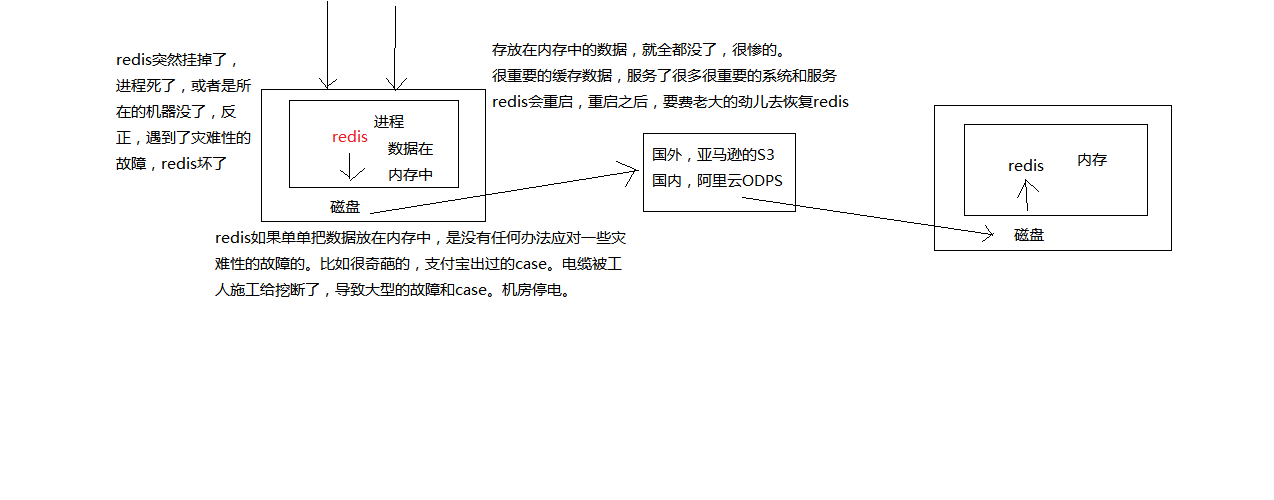
redis高可用：如果你做主从架构部署，其实就是加上哨兵就可以了，就可以实现，任何一个实例宕机，自动会进行主备切换。

### redis的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？

##### 01\_面试题以及解答思路介绍

##### 02\_redis持久化机对于生产环境中的灾难恢复的意义

redis持久化的意义



阿课程大纲

1、故障发生的时候会怎么样

2、如何应对故障的发生

很多同学，自己也看过一些redis的资料和书籍，当然可能也看过一些redis视频课程

所有的资料，其实都会讲解redis持久化，但是有个问题，我到目前为止，没有看到有人很仔细的去讲解，redis的持久化意义

redis的持久化，RDB，AOF，区别，各自的特点是什么，适合什么场景

redis的企业级的持久化方案是什么，是用来跟哪些企业级的场景结合起来使用的？？？

redis持久化的意义，在于故障恢复

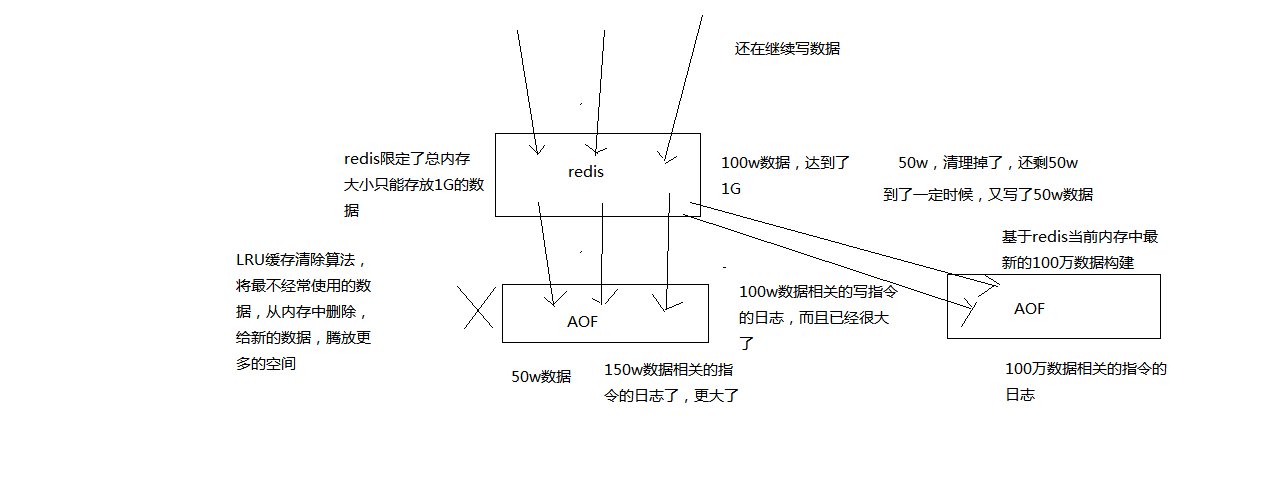
比如你部署了一个redis，作为cache缓存，当然也可以保存一些较为重要的数据

如果没有持久化的话，redis遇到灾难性故障的时候，就会丢失所有的数据

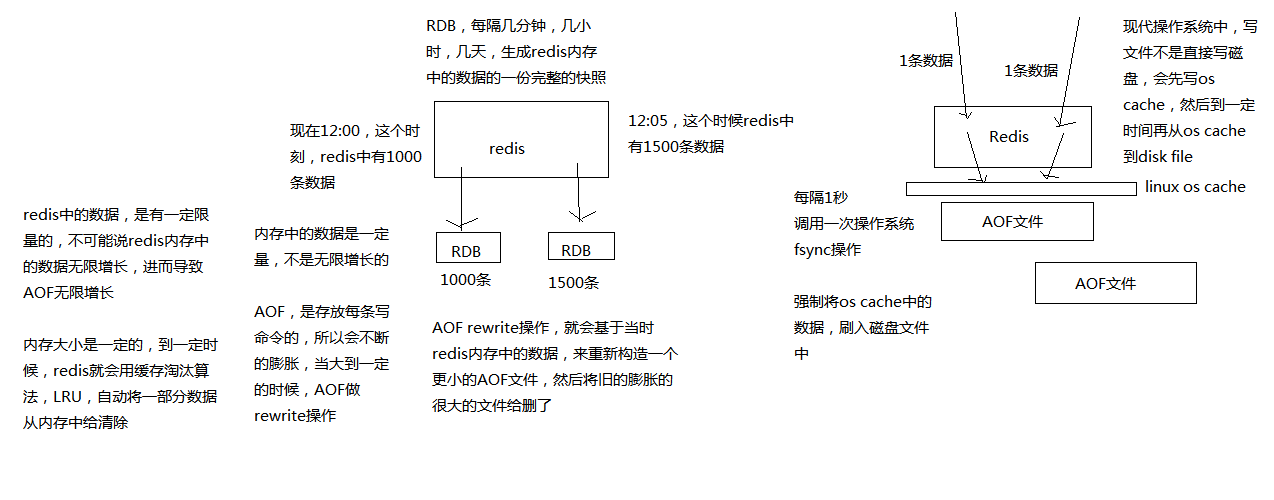
如果通过持久化将数据搞一份儿在磁盘上去，然后定期比如说同步和备份到一些云存储服务上去，那么就可以保证数据不丢失全部，还是可以恢复一部分数据回来的

##### 03\_图解分析redis的RDB和AOF两种持久化机制的工作原理

AOF rewrite原理剖析



RDB和AOF的介绍



课程大纲

1、RDB和AOF两种持久化机制的介绍

2、RDB持久化机制的优点

3、RDB持久化机制的缺点

4、AOF持久化机制的优点

5、AOF持久化机制的缺点

6、RDB和AOF到底该如何选择

我们已经知道对于一个企业级的redis架构来说，持久化是不可减少的

企业级redis集群架构：海量数据、高并发、高可用

持久化主要是做灾难恢复，数据恢复，也可以归类到高可用的一个环节里面去

比如你redis整个挂了，然后redis就不可用了，你要做的事情是让redis变得可用，尽快变得可用

重启redis，尽快让它对外提供服务，但是就像上一讲说，如果你没做数据备份，这个时候redis启动了，也不可用啊，数据都没了

很可能说，大量的请求过来，缓存全部无法命中，在redis里根本找不到数据，这个时候就死定了，缓存雪崩问题，所有请求，没有在redis命中，就会去mysql数据库这种数据源头中去找，一下子mysql承接高并发，然后就挂了

mysql挂掉，你都没法去找数据恢复到redis里面去，redis的数据从哪儿来？从mysql来。。。

具体的完整的缓存雪崩的场景，还有企业级的解决方案，到后面讲

如果你把redis的持久化做好，备份和恢复方案做到企业级的程度，那么即使你的redis故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务

redis的持久化，跟高可用，是有关系的，企业级redis架构中去讲解

redis持久化：RDB，AOF

-------------------------------------------------------------------------------------

1、RDB和AOF两种持久化机制的介绍

RDB持久化机制，对redis中的数据执行周期性的持久化

AOF机制对每条写入命令作为日志，以append-only的模式写入一个日志文件中，在redis重启的时候，可以通过回放AOF日志中的写入指令来重新构建整个数据集

如果我们想要redis仅仅作为纯内存的缓存来用，那么可以禁止RDB和AOF所有的持久化机制

通过RDB或AOF，都可以将redis内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云，云服务

如果redis挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动redis，redis就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务

如果同时使用RDB和AOF两种持久化机制，那么在redis重启的时候，会使用AOF来重新构建数据，因为AOF中的数据更加完整

-------------------------------------------------------------------------------------

2、RDB持久化机制的优点

（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说Amazon的S3云服务上去，在国内可以是阿里云的ODPS分布式存储上，以预定好的备份策略来定期备份redis中的数据

（2）RDB对redis对外提供的读写服务，影响非常小，可以让redis保持高性能，因为redis主进程只需要fork一个子进程，让子进程执行磁盘IO操作来进行RDB持久化即可

（3）相对于AOF持久化机制来说，直接基于RDB数据文件来重启和恢复redis进程，更加快速

-------------------------------------------------------------------------------------

3、RDB持久化机制的缺点

（1）如果想要在redis故障时，尽可能少的丢失数据，那么RDB没有AOF好。一般来说，RDB数据快照文件，都是每隔5分钟，或者更长时间生成一次，这个时候就得接受一旦redis进程宕机，那么会丢失最近5分钟的数据

（2）RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒

-------------------------------------------------------------------------------------

4、AOF持久化机制的优点

（1）AOF可以更好的保护数据不丢失，一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据

（2）AOF日志文件以append-only模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复

（3）AOF日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在rewrite log的时候，会对其中的指导进行压缩，创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后的日志文件ready的时候，再交换新老日志文件即可。

（4）AOF日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据

-------------------------------------------------------------------------------------

5、AOF持久化机制的缺点

（1）对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

（2）AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的

（3）以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

-------------------------------------------------------------------------------------

6、RDB和AOF到底该如何选择

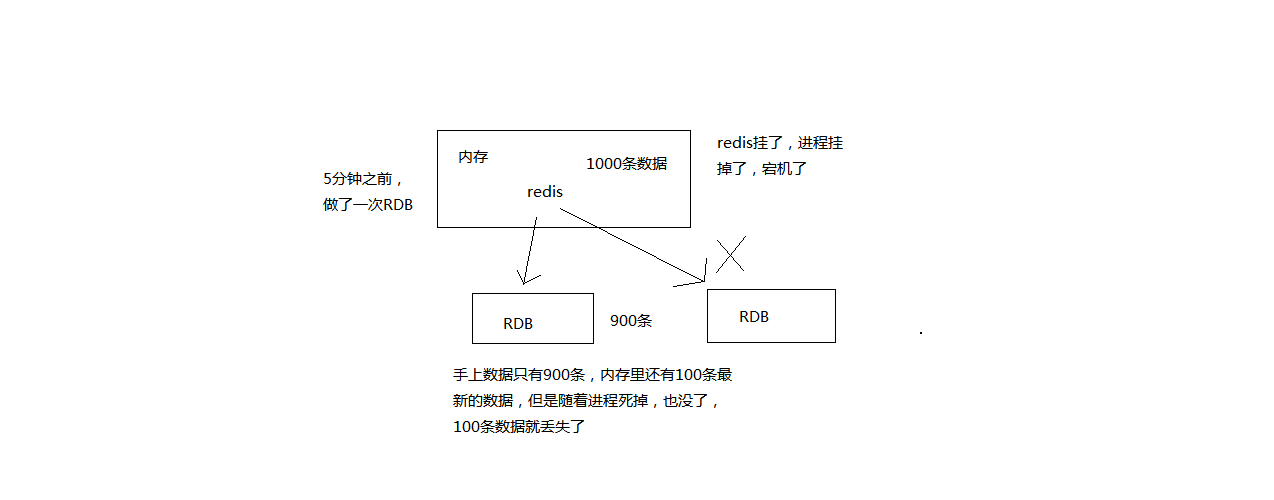
（1）不要仅仅使用RDB，因为那样会导致你丢失很多数据

（2）也不要仅仅使用AOF，因为那样有两个问题，第一，你通过AOF做冷备，没有RDB做冷备，来的恢复速度更快; 第二，RDB每次简单粗暴生成数据快照，更加健壮，可以避免AOF这种复杂的备份和恢复机制的bug

（3）综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择; 用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复

##### 04\_redis的RDB和AOF两种持久化机制的优劣势对比

RDB丢失数据的问题



课程大纲

1、RDB和AOF两种持久化机制的介绍

2、RDB持久化机制的优点

3、RDB持久化机制的缺点

4、AOF持久化机制的优点

5、AOF持久化机制的缺点

6、RDB和AOF到底该如何选择

我们已经知道对于一个企业级的redis架构来说，持久化是不可减少的

企业级redis集群架构：海量数据、高并发、高可用

持久化主要是做灾难恢复，数据恢复，也可以归类到高可用的一个环节里面去

比如你redis整个挂了，然后redis就不可用了，你要做的事情是让redis变得可用，尽快变得可用

重启redis，尽快让它对外提供服务，但是就像上一讲说，如果你没做数据备份，这个时候redis启动了，也不可用啊，数据都没了

很可能说，大量的请求过来，缓存全部无法命中，在redis里根本找不到数据，这个时候就死定了，缓存雪崩问题，所有请求，没有在redis命中，就会去mysql数据库这种数据源头中去找，一下子mysql承接高并发，然后就挂了

mysql挂掉，你都没法去找数据恢复到redis里面去，redis的数据从哪儿来？从mysql来。。。

具体的完整的缓存雪崩的场景，还有企业级的解决方案，到后面讲

如果你把redis的持久化做好，备份和恢复方案做到企业级的程度，那么即使你的redis故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务

redis的持久化，跟高可用，是有关系的，企业级redis架构中去讲解

redis持久化：RDB，AOF

-------------------------------------------------------------------------------------

1、RDB和AOF两种持久化机制的介绍

RDB持久化机制，对redis中的数据执行周期性的持久化

AOF机制对每条写入命令作为日志，以append-only的模式写入一个日志文件中，在redis重启的时候，可以通过回放AOF日志中的写入指令来重新构建整个数据集

如果我们想要redis仅仅作为纯内存的缓存来用，那么可以禁止RDB和AOF所有的持久化机制

通过RDB或AOF，都可以将redis内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云，云服务

如果redis挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动redis，redis就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务

如果同时使用RDB和AOF两种持久化机制，那么在redis重启的时候，会使用AOF来重新构建数据，因为AOF中的数据更加完整

-------------------------------------------------------------------------------------

2、RDB持久化机制的优点

（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说Amazon的S3云服务上去，在国内可以是阿里云的ODPS分布式存储上，以预定好的备份策略来定期备份redis中的数据

RDB也可以做冷备，生成多个文件，每个文件都代表了某一个时刻的完整的数据快照

AOF也可以做冷备，只有一个文件，但是你可以，每隔一定时间，去copy一份这个文件出来

RDB做冷备，优势在哪儿呢？由redis去控制固定时长生成快照文件的事情，比较方便; AOF，还需要自己写一些脚本去做这个事情，各种定时

RDB数据做冷备，在最坏的情况下，提供数据恢复的时候，速度比AOF快

（2）RDB对redis对外提供的读写服务，影响非常小，可以让redis保持高性能，因为redis主进程只需要fork一个子进程，让子进程执行磁盘IO操作来进行RDB持久化即可

RDB，每次写，都是直接写redis内存，只是在一定的时候，才会将数据写入磁盘中

AOF，每次都是要写文件的，虽然可以快速写入os cache中，但是还是有一定的时间开销的,速度肯定比RDB略慢一些

（3）相对于AOF持久化机制来说，直接基于RDB数据文件来重启和恢复redis进程，更加快速

AOF，存放的指令日志，做数据恢复的时候，其实是要回放和执行所有的指令日志，来恢复出来内存中的所有数据的

RDB，就是一份数据文件，恢复的时候，直接加载到内存中即可

结合上述优点，RDB特别适合做冷备份，冷备

-------------------------------------------------------------------------------------

3、RDB持久化机制的缺点

（1）如果想要在redis故障时，尽可能少的丢失数据，那么RDB没有AOF好。一般来说，RDB数据快照文件，都是每隔5分钟，或者更长时间生成一次，这个时候就得接受一旦redis进程宕机，那么会丢失最近5分钟的数据

这个问题，也是rdb最大的缺点，就是不适合做第一优先的恢复方案，如果你依赖RDB做第一优先恢复方案，会导致数据丢失的比较多

（2）RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒

一般不要让RDB的间隔太长，否则每次生成的RDB文件太大了，对redis本身的性能可能会有影响的

-------------------------------------------------------------------------------------

4、AOF持久化机制的优点

（1）AOF可以更好的保护数据不丢失，一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据

每隔1秒，就执行一次fsync操作，保证os cache中的数据写入磁盘中

redis进程挂了，最多丢掉1秒钟的数据

（2）AOF日志文件以append-only模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复

（3）AOF日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在rewrite log的时候，会对其中的指导进行压缩，创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后的日志文件ready的时候，再交换新老日志文件即可。

（4）AOF日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据

-------------------------------------------------------------------------------------

5、AOF持久化机制的缺点

（1）对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

（2）AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的

如果你要保证一条数据都不丢，也是可以的，AOF的fsync设置成没写入一条数据，fsync一次，那就完蛋了，redis的QPS大降

（3）以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

（4）唯一的比较大的缺点，其实就是做数据恢复的时候，会比较慢，还有做冷备，定期的备份，不太方便，可能要自己手写复杂的脚本去做，做冷备不太合适

-------------------------------------------------------------------------------------

6、RDB和AOF到底该如何选择

（1）不要仅仅使用RDB，因为那样会导致你丢失很多数据

（2）也不要仅仅使用AOF，因为那样有两个问题，第一，你通过AOF做冷备，没有RDB做冷备，来的恢复速度更快; 第二，RDB每次简单粗暴生成数据快照，更加健壮，可以避免AOF这种复杂的备份和恢复机制的bug

（3）综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择; 用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复

##### 总结

1、面试题

redis的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？

2、面试官心里分析

redis如果仅仅只是将数据缓存在内存里面，如果redis宕机了，再重启，内存里的数据就全部都弄丢了啊。。。。。。你必须得用redis的持久化机制，将数据写入内存的同时，异步的慢慢的将数据写入磁盘文件里，进行持久化

如果redis宕机了，重启启动，自动从磁盘上加载之前持久化的一些数据，就可以了，也许会丢失少许数据，但是至少不会将所有数据都弄丢

这个其实一样，针对的都是redis的生产环境可能遇到的一些问题，就是redis要是挂了再重启，内存里的数据不就全丢了？能不能重启的时候把数据给恢复了？

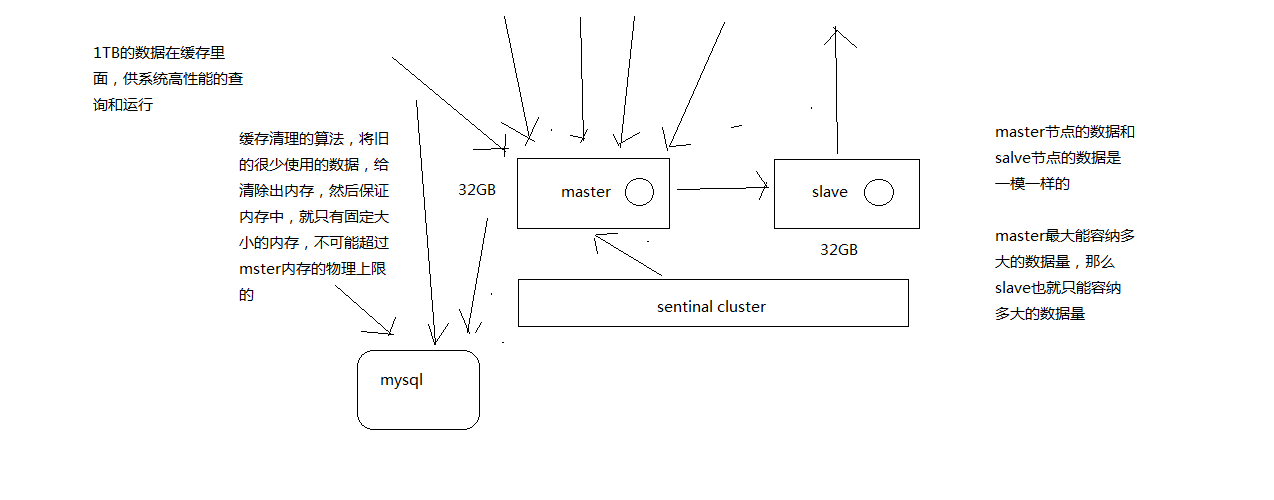
3、面试题剖析

### redis集群模式的工作原理能说一下么？在集群模式下，redis的key是如何寻址的？分布式寻址都有哪些算法？了解一致性hash算法吗？如何动态增加和删除一个节点？

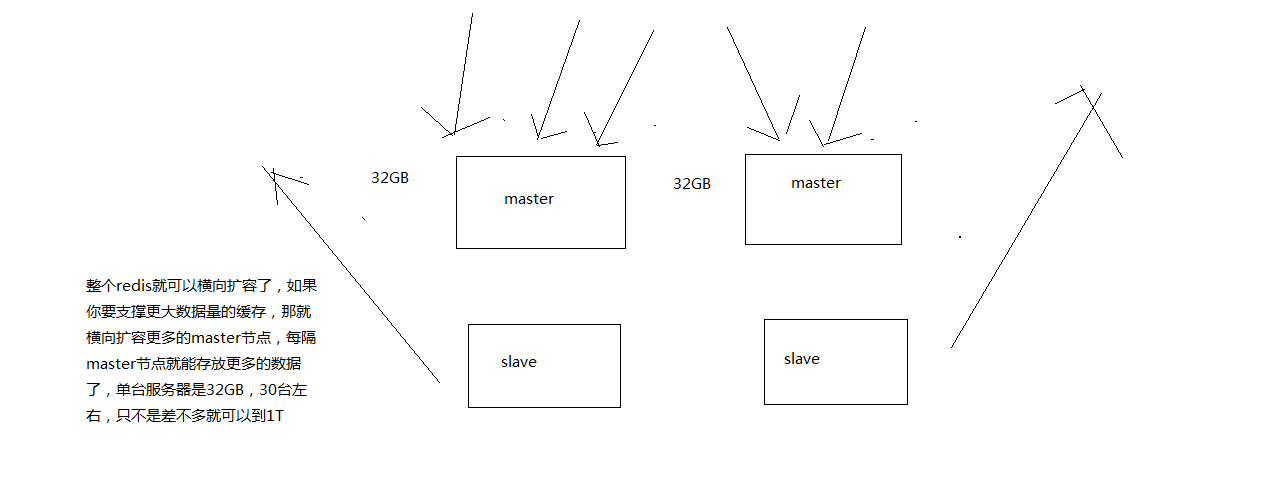
#### 01\_课程介绍

#### 02\_redis如何在保持读写分离+高可用的架构下，还能横向扩容支撑1T+海量数据

redis单master架构的容量的瓶颈问题



redis如何通过master横向扩容支撑1T+数据量



1、单机redis在海量数据面前的瓶颈

2、怎么才能够突破单机瓶颈，让redis支撑海量数据？

3、redis的集群架构

redis cluster

支撑N个redis master node，每个master node都可以挂载多个slave node

读写分离的架构，对于每个master来说，写就写到master，然后读就从mater对应的slave去读

高可用，因为每个master都有salve节点，那么如果mater挂掉，redis cluster这套机制，就会自动将某个slave切换成master

redis cluster（多master + 读写分离 + 高可用）

我们只要基于redis cluster去搭建redis集群即可，不需要手工去搭建replication复制+主从架构+读写分离+哨兵集群+高可用

4、redis cluster vs. replication + sentinal

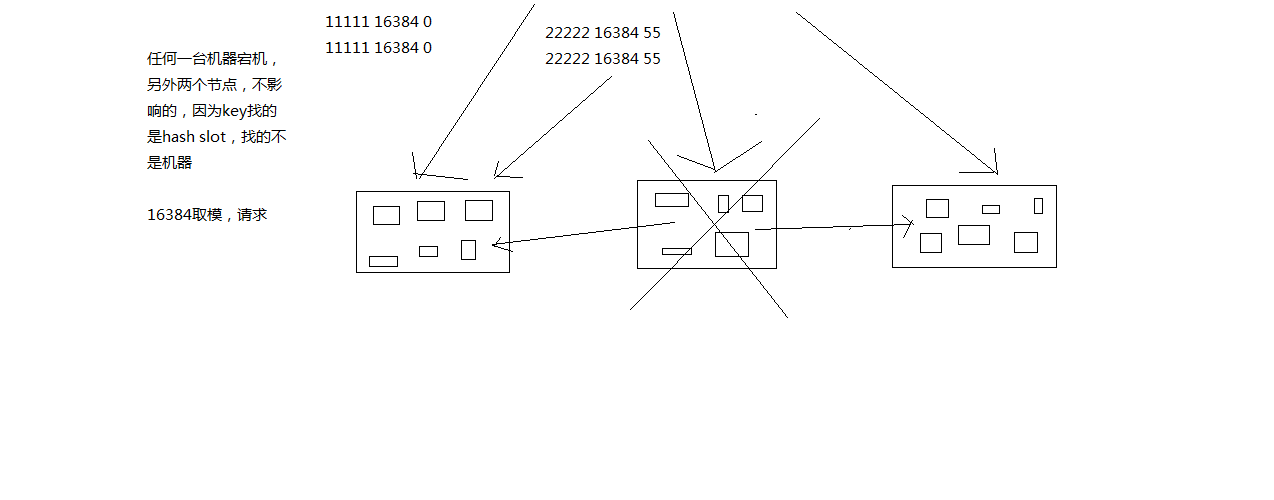
如果你的数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个G，单机足够了

replication，一个mater，多个slave，要几个slave跟你的要求的读吞吐量有关系，然后自己搭建一个sentinal集群，去保证redis主从架构的高可用性，就可以了

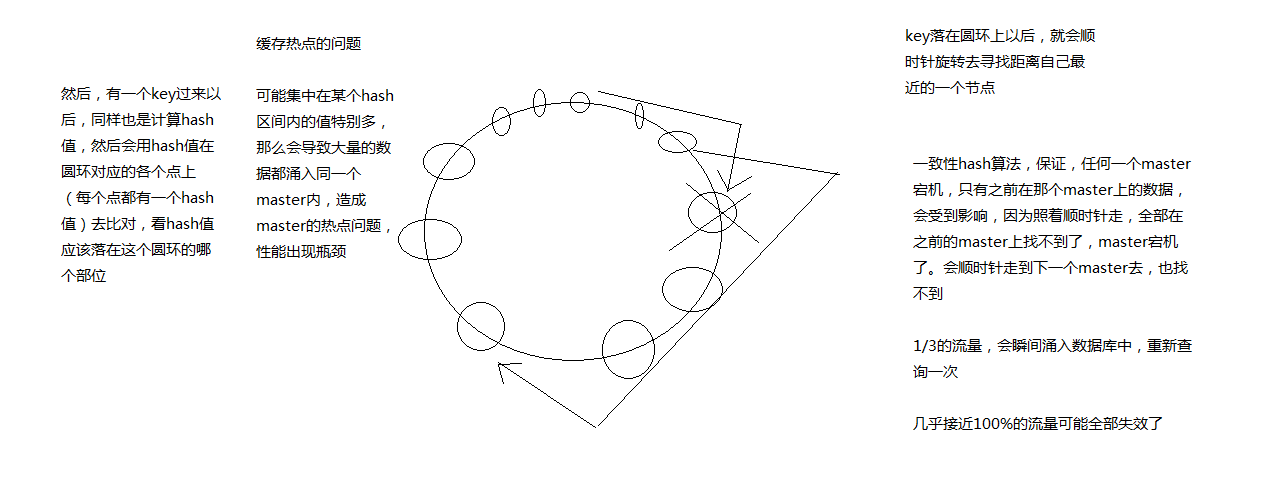
redis cluster，主要是针对海量数据+高并发+高可用的场景，海量数据，如果你的数据量很大，那么建议就用redis cluster

#### 03\_数据分布算法：hash+一致性hash+redis cluster的hash slot

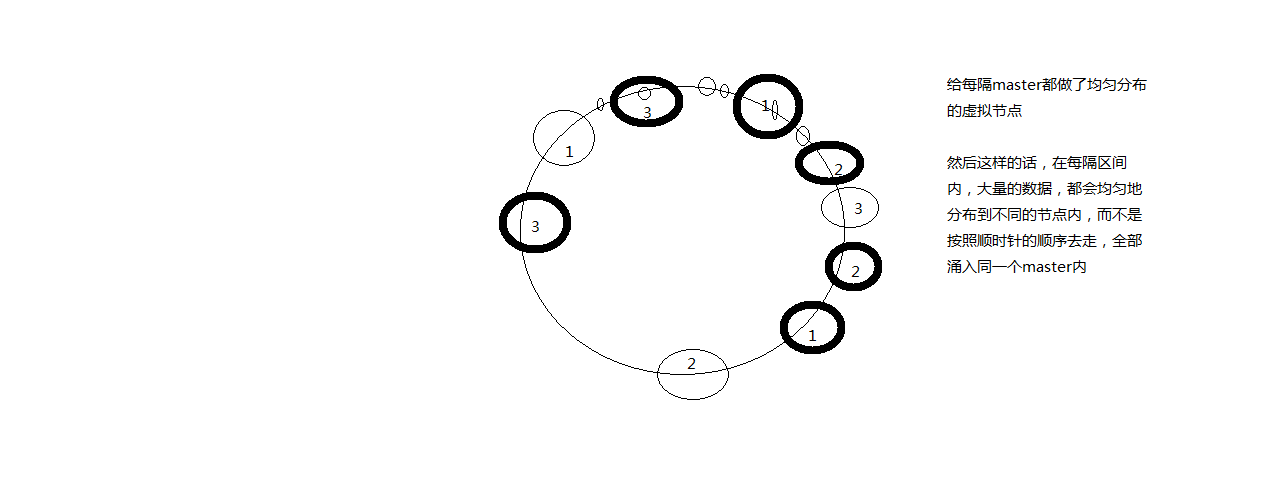
redis cluster hash slot算法



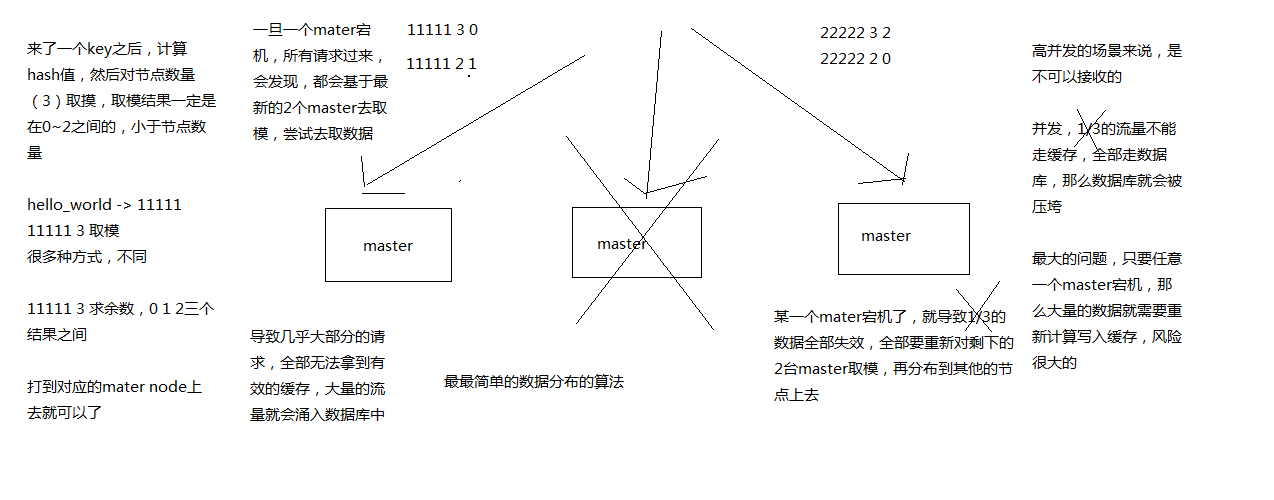
一致性hash算法的讲解和优点



一致性hash算法的虚拟节点实现负载均衡



最老土的hash算法以及弊端



讲解分布式数据存储的核心算法，数据分布的算法

hash算法 -> 一致性hash算法（memcached） -> redis cluster，hash slot算法

用不同的算法，就决定了在多个master节点的时候，数据如何分布到这些节点上去，解决这个问题

1、redis cluster介绍

redis cluster

（1）自动将数据进行分片，每个master上放一部分数据

（2）提供内置的高可用支持，部分master不可用时，还是可以继续工作的

在redis cluster架构下，每个redis要放开两个端口号，比如一个是6379，另外一个就是加10000的端口号，比如16379

16379端口号是用来进行节点间通信的，也就是cluster bus的东西，集群总线。cluster bus的通信，用来进行故障检测，配置更新，故障转移授权

cluster bus用了另外一种二进制的协议，主要用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间

2、最老土的hash算法和弊端（大量缓存重建）

3、一致性hash算法（自动缓存迁移）+虚拟节点（自动负载均衡）

4、redis cluster的hash slot算法

redis cluster有固定的16384个hash slot，对每个key计算CRC16值，然后对16384取模，可以获取key对应的hash slot

redis cluster中每个master都会持有部分slot，比如有3个master，那么可能每个master持有5000多个hash slot

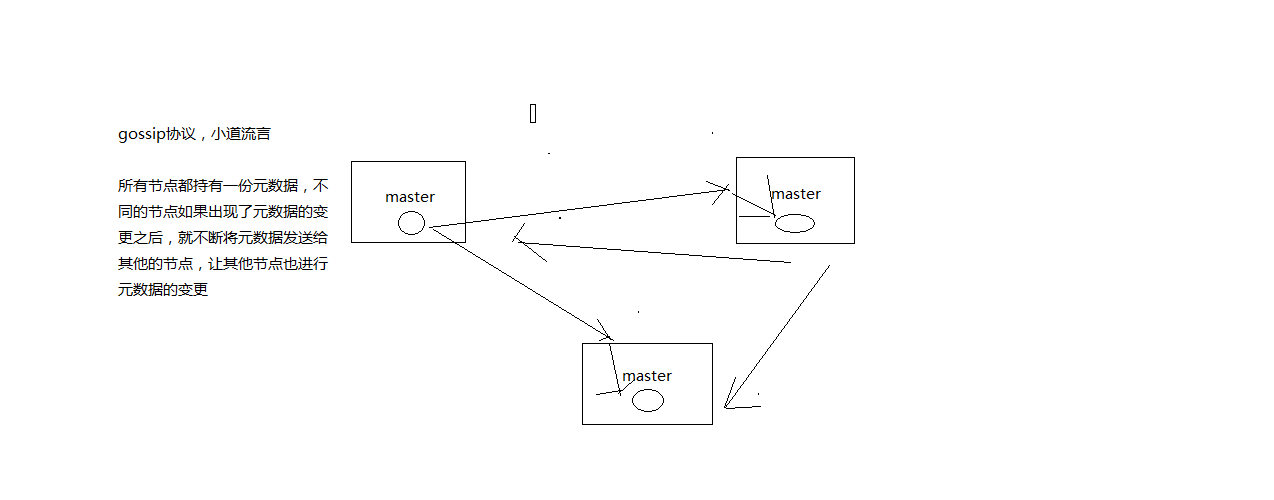
hash slot让node的增加和移除很简单，增加一个master，就将其他master的hash slot移动部分过去，减少一个master，就将它的hash slot移动到其他master上去

移动hash slot的成本是非常低的

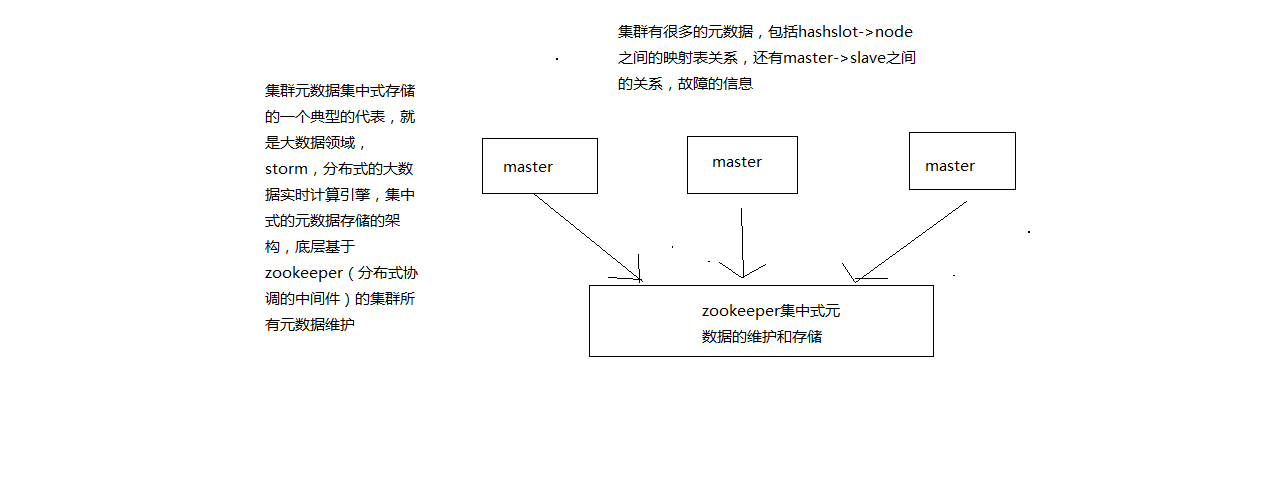
客户端的api，可以对指定的数据，让他们走同一个hash slot，通过hash tag来实现

#### 04\_redis cluster的核心原理分析：gossip通信、jedis smart定位、主备切换

gossip协议维护集群元数据



集中式的集群元数据存储和维护



一、节点间的内部通信机制

1、基础通信原理

（1）redis cluster节点间采取gossip协议进行通信

跟集中式不同，不是将集群元数据（节点信息，故障，等等）集中存储在某个节点上，而是互相之间不断通信，保持整个集群所有节点的数据是完整的

维护集群的元数据用得，集中式，一种叫做gossip

集中式：好处在于，元数据的更新和读取，时效性非常好，一旦元数据出现了变更，立即就更新到集中式的存储中，其他节点读取的时候立即就可以感知到; 不好在于，所有的元数据的跟新压力全部集中在一个地方，可能会导致元数据的存储有压力

gossip：好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续，打到所有节点上去更新，有一定的延时，降低了压力; 缺点，元数据更新有延时，可能导致集群的一些操作会有一些滞后

我们刚才做reshard，去做另外一个操作，会发现说，configuration error，达成一致

（2）10000端口

每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如7001，那么用于节点间通信的就是17001端口

每隔节点每隔一段时间都会往另外几个节点发送ping消息，同时其他几点接收到ping之后返回pong

（3）交换的信息

故障信息，节点的增加和移除，hash slot信息，等等

2、gossip协议

gossip协议包含多种消息，包括ping，pong，meet，fail，等等

meet: 某个节点发送meet给新加入的节点，让新节点加入集群中，然后新节点就会开始与其他节点进行通信

redis-trib.rb add-node

其实内部就是发送了一个gossip meet消息，给新加入的节点，通知那个节点去加入我们的集群

ping: 每个节点都会频繁给其他节点发送ping，其中包含自己的状态还有自己维护的集群元数据，互相通过ping交换元数据

每个节点每秒都会频繁发送ping给其他的集群，ping，频繁的互相之间交换数据，互相进行元数据的更新

pong: 返回ping和meet，包含自己的状态和其他信息，也可以用于信息广播和更新

fail: 某个节点判断另一个节点fail之后，就发送fail给其他节点，通知其他节点，指定的节点宕机了

3、ping消息深入

ping很频繁，而且要携带一些元数据，所以可能会加重网络负担

每个节点每秒会执行10次ping，每次会选择5个最久没有通信的其他节点

当然如果发现某个节点通信延时达到了cluster\_node\_timeout / 2，那么立即发送ping，避免数据交换延时过长，落后的时间太长了

比如说，两个节点之间都10分钟没有交换数据了，那么整个集群处于严重的元数据不一致的情况，就会有问题

所以cluster\_node\_timeout可以调节，如果调节比较大，那么会降低发送的频率

每次ping，一个是带上自己节点的信息，还有就是带上1/10其他节点的信息，发送出去，进行数据交换

至少包含3个其他节点的信息，最多包含总节点-2个其他节点的信息

-------------------------------------------------------------------------------------------------------

二、面向集群的jedis内部实现原理

开发，jedis，redis的java client客户端，redis cluster，jedis cluster api

jedis cluster api与redis cluster集群交互的一些基本原理

1、基于重定向的客户端

redis-cli -c，自动重定向

（1）请求重定向

客户端可能会挑选任意一个redis实例去发送命令，每个redis实例接收到命令，都会计算key对应的hash slot

如果在本地就在本地处理，否则返回moved给客户端，让客户端进行重定向

cluster keyslot mykey，可以查看一个key对应的hash slot是什么

用redis-cli的时候，可以加入-c参数，支持自动的请求重定向，redis-cli接收到moved之后，会自动重定向到对应的节点执行命令

（2）计算hash slot

计算hash slot的算法，就是根据key计算CRC16值，然后对16384取模，拿到对应的hash slot

用hash tag可以手动指定key对应的slot，同一个hash tag下的key，都会在一个hash slot中，比如set mykey1:{100}和set mykey2:{100}

（3）hash slot查找

节点间通过gossip协议进行数据交换，就知道每个hash slot在哪个节点上

2、smart jedis

（1）什么是smart jedis

基于重定向的客户端，很消耗网络IO，因为大部分情况下，可能都会出现一次请求重定向，才能找到正确的节点

所以大部分的客户端，比如java redis客户端，就是jedis，都是smart的

本地维护一份hashslot -> node的映射表，缓存，大部分情况下，直接走本地缓存就可以找到hashslot -> node，不需要通过节点进行moved重定向

（2）JedisCluster的工作原理

在JedisCluster初始化的时候，就会随机选择一个node，初始化hashslot -> node映射表，同时为每个节点创建一个JedisPool连接池

每次基于JedisCluster执行操作，首先JedisCluster都会在本地计算key的hashslot，然后在本地映射表找到对应的节点

如果那个node正好还是持有那个hashslot，那么就ok; 如果说进行了reshard这样的操作，可能hashslot已经不在那个node上了，就会返回moved

如果JedisCluter API发现对应的节点返回moved，那么利用该节点的元数据，更新本地的hashslot -> node映射表缓存

重复上面几个步骤，直到找到对应的节点，如果重试超过5次，那么就报错，JedisClusterMaxRedirectionException

jedis老版本，可能会出现在集群某个节点故障还没完成自动切换恢复时，频繁更新hash slot，频繁ping节点检查活跃，导致大量网络IO开销

jedis最新版本，对于这些过度的hash slot更新和ping，都进行了优化，避免了类似问题

（3）hashslot迁移和ask重定向

如果hash slot正在迁移，那么会返回ask重定向给jedis

jedis接收到ask重定向之后，会重新定位到目标节点去执行，但是因为ask发生在hash slot迁移过程中，所以JedisCluster API收到ask是不会更新hashslot本地缓存

已经可以确定说，hashslot已经迁移完了，moved是会更新本地hashslot->node映射表缓存的

-------------------------------------------------------------------------------------------------------

三、高可用性与主备切换原理

redis cluster的高可用的原理，几乎跟哨兵是类似的

1、判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是pfail，主观宕机

如果多个节点都认为另外一个节点宕机了，那么就是fail，客观宕机，跟哨兵的原理几乎一样，sdown，odown

在cluster-node-timeout内，某个节点一直没有返回pong，那么就被认为pfail

如果一个节点认为某个节点pfail了，那么会在gossip ping消息中，ping给其他节点，如果超过半数的节点都认为pfail了，那么就会变成fail

2、从节点过滤

对宕机的master node，从其所有的slave node中，选择一个切换成master node

检查每个slave node与master node断开连接的时间，如果超过了cluster-node-timeout \* cluster-slave-validity-factor，那么就没有资格切换成master

这个也是跟哨兵是一样的，从节点超时过滤的步骤

3、从节点选举

哨兵：对所有从节点进行排序，slave priority，offset，run id

每个从节点，都根据自己对master复制数据的offset，来设置一个选举时间，offset越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举

所有的master node开始slave选举投票，给要进行选举的slave进行投票，如果大部分master node（N/2 + 1）都投票给了某个从节点，那么选举通过，那个从节点可以切换成master

从节点执行主备切换，从节点切换为主节点

4、与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，redis cluster功能强大，直接集成了replication和sentinal的功能

没有办法去给大家深入讲解redis底层的设计的细节，核心原理和设计的细节，那个除非单独开一门课，redis底层原理深度剖析，redis源码

对于咱们这个架构课来说，主要关注的是架构，不是底层的细节，对于架构来说，核心的原理的基本思路，是要梳理清晰的

#### 05\_课程总结

#### 总结

1、面试题

redis集群模式的工作原理能说一下么？在集群模式下，redis的key是如何寻址的？分布式寻址都有哪些算法？了解一致性hash算法吗？

2、面试官心里分析

在以前，如果前几年的时候，一般来说，redis如果要搞几个节点，每个节点存储一部分的数据，得借助一些中间件来实现，比如说有codis，或者twemproxy，都有。有一些redis中间件，你读写redis中间件，redis中间件负责将你的数据分布式存储在多台机器上的redis实例中。

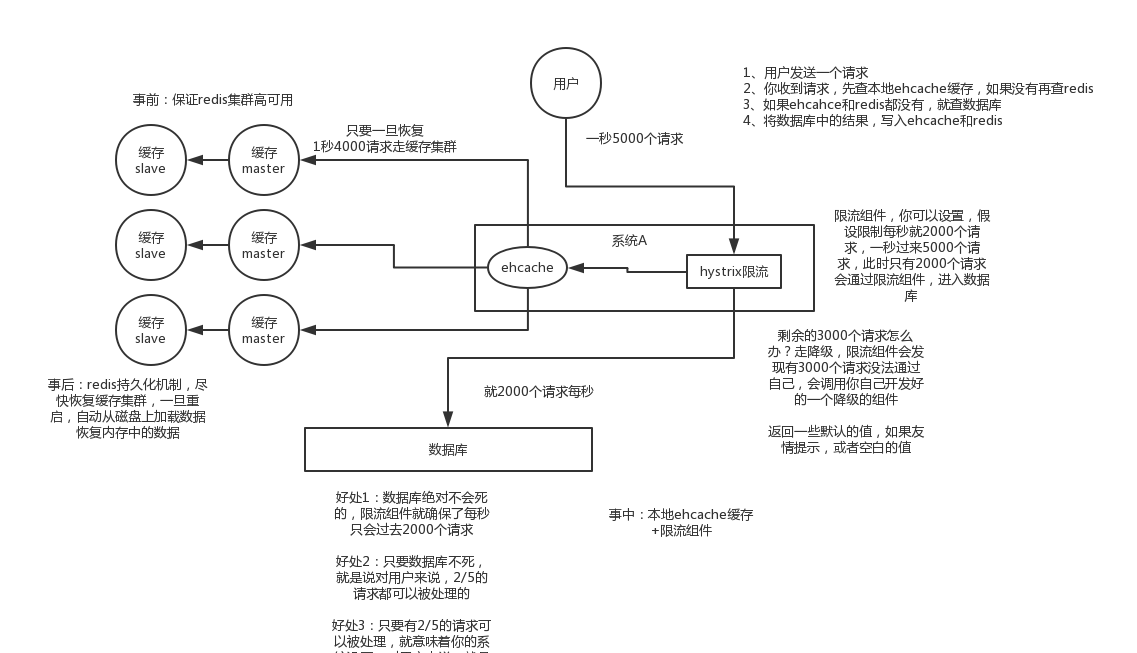
这两年，redis不断在发展，redis也不断的有新的版本，redis cluster，redis集群模式，你可以做到在多台机器上，部署多个redis实例，每个实例存储一部分的数据，同时每个redis实例可以挂redis从实例，自动确保说，如果redis主实例挂了，会自动切换到redis从实例顶上来。

现在redis的新版本，大家都是用redis cluster的，也就是redis原生支持的redis集群模式，那么面试官肯定会就redis cluster对你来个几连炮。要是你没用过redis cluster，正常，以前很多人用codis之类的客户端来支持集群，但是起码你得研究一下redis cluster吧。

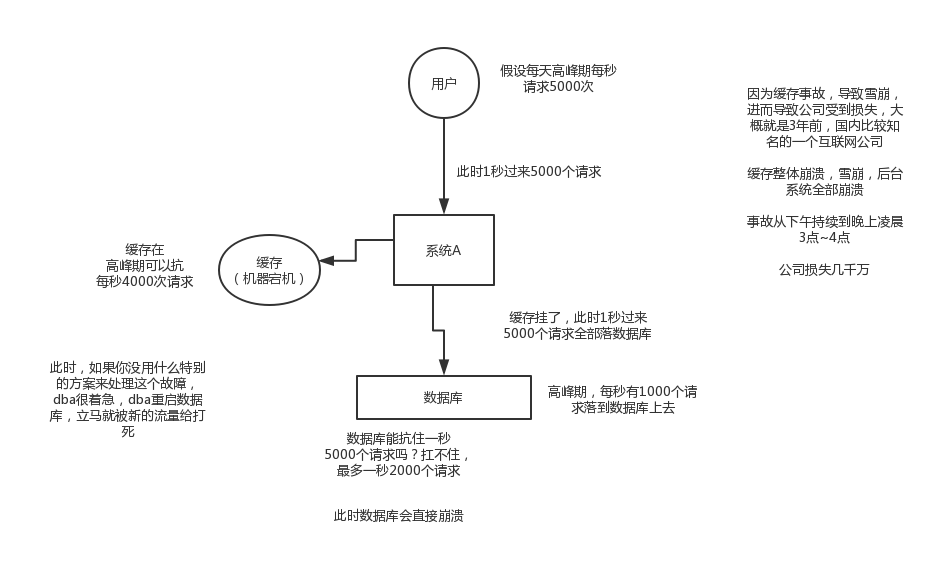
3、面试题剖析

### 了解什么是redis的雪崩和穿透？redis崩溃之后会怎么样？系统该如何应对这种情况？如何处理redis的穿透？

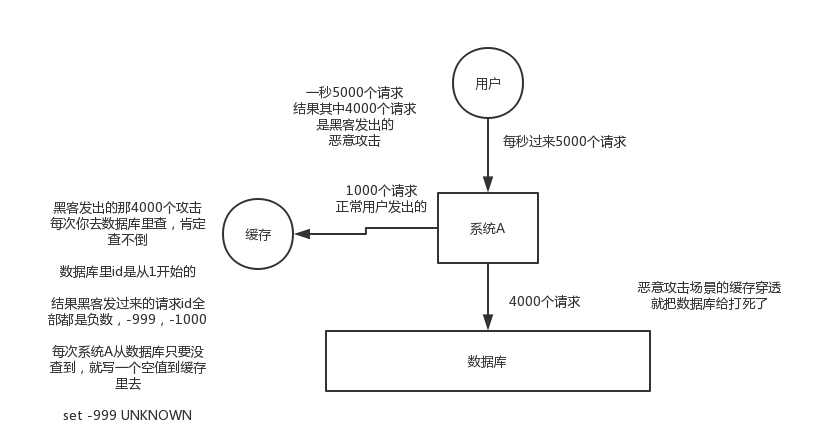
01\_缓存雪崩现象



02\_如何解决缓存雪崩



03\_缓存穿透现象以及解决方案



1、面试题

了解什么是redis的雪崩和穿透？redis崩溃之后会怎么样？系统该如何应对这种情况？如何处理redis的穿透？

2、面试官心里分析

其实这是问到缓存必问的，因为缓存雪崩和穿透，那是缓存最大的两个问题，要么不出现，一旦出现就是致命性的问题。所以面试官一定会问你。

3、面试题剖析

缓存雪崩发生的现象

缓存雪崩的事前事中事后的解决方案

事前：redis高可用，主从+哨兵，redis cluster，避免全盘崩溃

事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL被打死

事后：redis持久化，快速恢复缓存数据

缓存穿透的现象

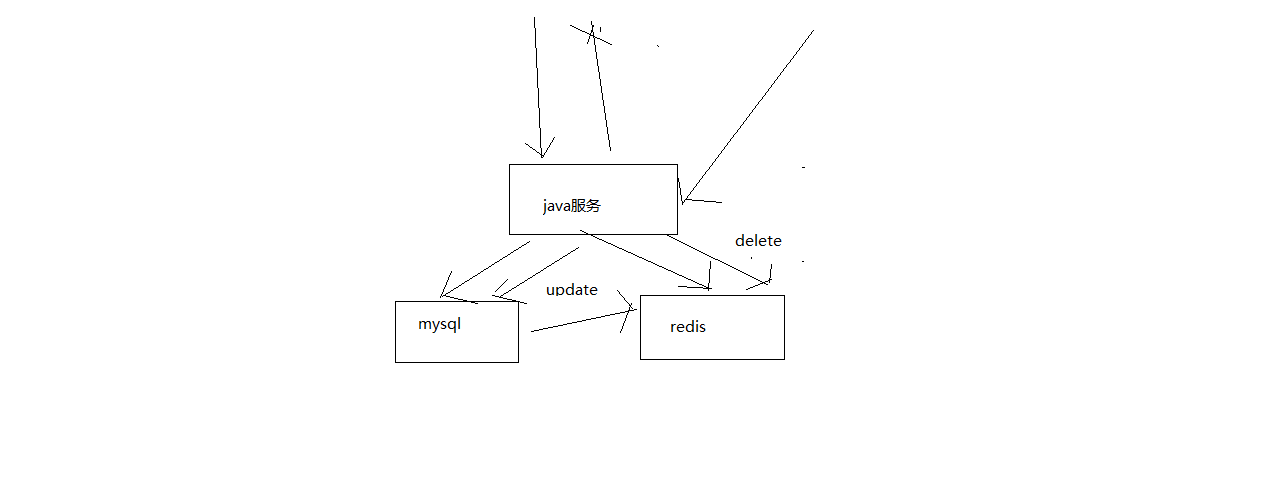
缓存穿透的解决方法

### 如何保证缓存与数据库的双写一致性？

#### 01\_课程介绍

#### 02\_Cache Aside Pattern缓存+数据库读写模式的分析

cache aside pattern



最经典的缓存+数据库读写的模式，cache aside pattern

1、Cache Aside Pattern

（1）读的时候，先读缓存，缓存没有的话，那么就读数据库，然后取出数据后放入缓存，同时返回响应

（2）更新的时候，先删除缓存，然后再更新数据库

2、为什么是删除缓存，而不是更新缓存呢？

原因很简单，很多时候，复杂点的缓存的场景，因为缓存有的时候，不简单是数据库中直接取出来的值

商品详情页的系统，修改库存，只是修改了某个表的某些字段，但是要真正把这个影响的最终的库存计算出来，可能还需要从其他表查询一些数据，然后进行一些复杂的运算，才能最终计算出

现在最新的库存是多少，然后才能将库存更新到缓存中去

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据，并进行运算，才能计算出缓存最新的值的

更新缓存的代价是很高的

是不是说，每次修改数据库的时候，都一定要将其对应的缓存去跟新一份？也许有的场景是这样的，但是对于比较复杂的缓存数据计算的场景，就不是这样了

如果你频繁修改一个缓存涉及的多个表，那么这个缓存会被频繁的更新，频繁的更新缓存

但是问题在于，这个缓存到底会不会被频繁访问到？？？

举个例子，一个缓存涉及的表的字段，在1分钟内就修改了20次，或者是100次，那么缓存跟新20次，100次; 但是这个缓存在1分钟内就被读取了1次，有大量的冷数据

28法则，黄金法则，20%的数据，占用了80%的访问量

实际上，如果你只是删除缓存的话，那么1分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低

每次数据过来，就只是删除缓存，然后修改数据库，如果这个缓存，在1分钟内只是被访问了1次，那么只有那1次，缓存是要被重新计算的，用缓存才去算缓存

其实删除缓存，而不是更新缓存，就是一个lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算

mybatis，hibernate，懒加载，思想

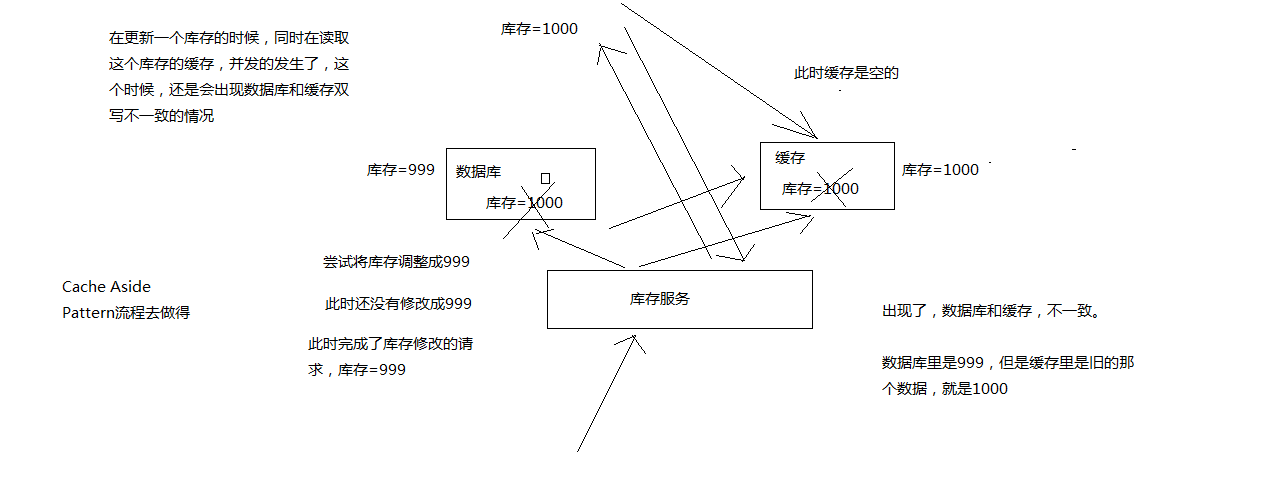
查询一个部门，部门带了一个员工的list，没有必要说每次查询部门，都里面的1000个员工的数据也同时查出来啊

80%的情况，查这个部门，就只是要访问这个部门的信息就可以了

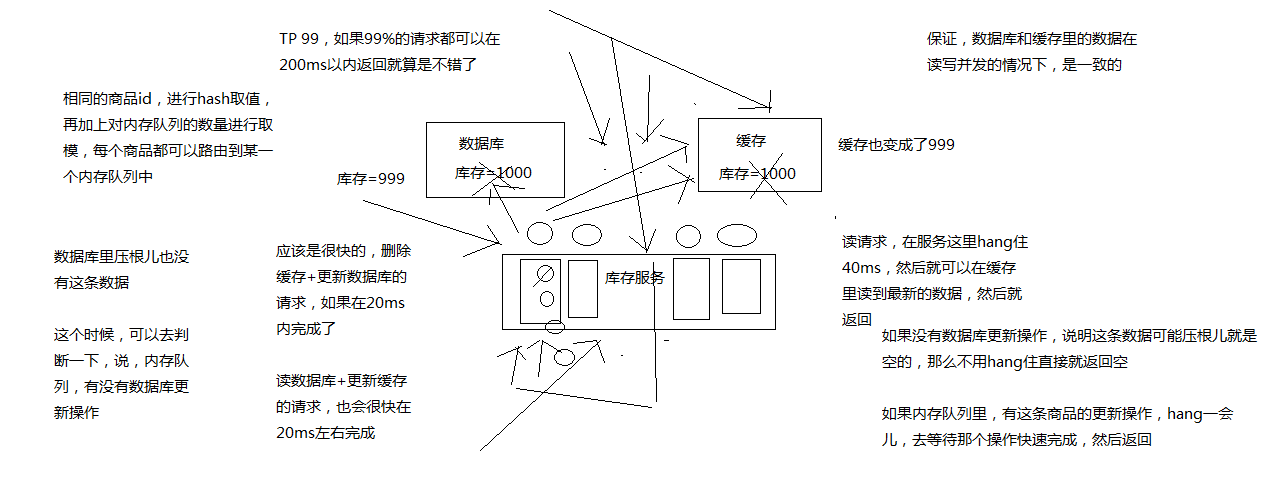
先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工

#### 03\_高并发场景下的缓存+数据库双写不一致问题分析与解决方案设计

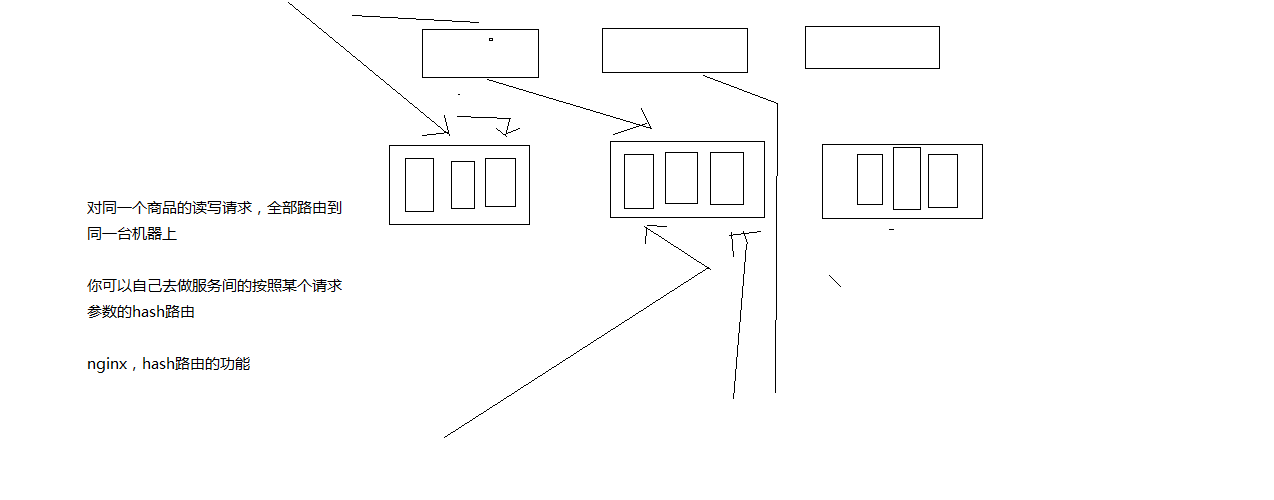
读写并发的时候复杂的数据库+缓存双写不一致的场景



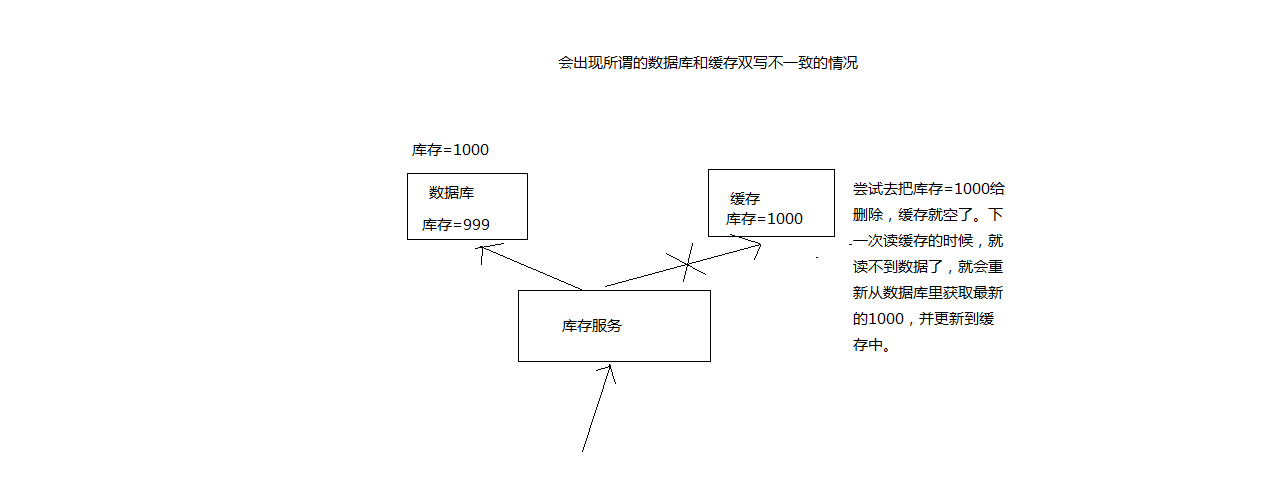
复杂的数据库+缓存双写一致保障方案



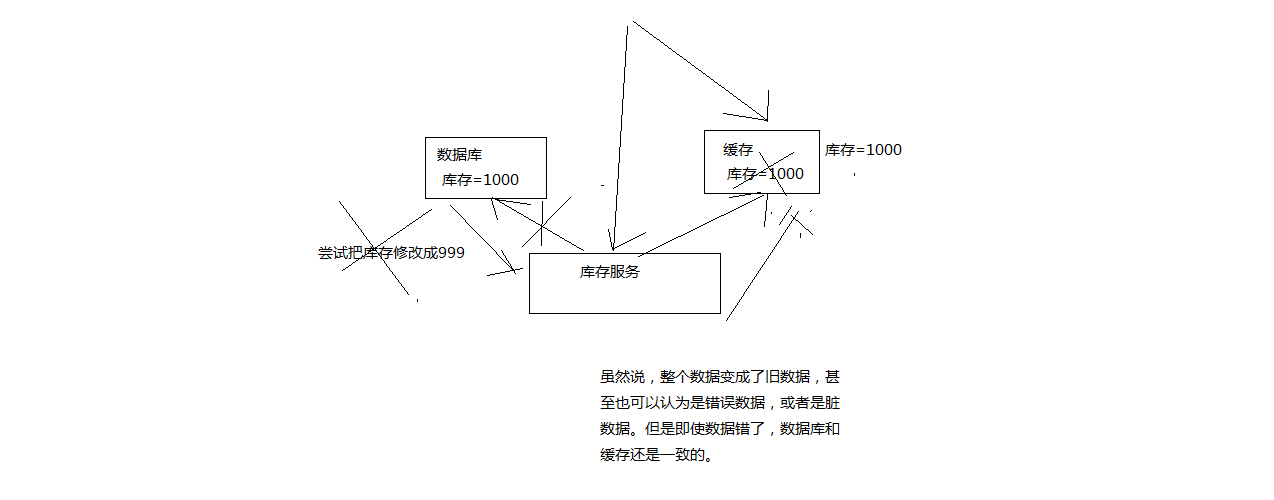
机器级别的请求路由问题



最初级的数据库+缓存双写不一致问题



最初级的数据库+缓存双写不一致问题的解决方案



马上开始去开发业务系统

从哪一步开始做，从比较简单的那一块开始做，实时性要求比较高的那块数据的缓存去做

实时性比较高的数据缓存，选择的就是库存的服务

库存可能会修改，每次修改都要去更新这个缓存数据; 每次库存的数据，在缓存中一旦过期，或者是被清理掉了，前端的nginx服务都会发送请求给库存服务，去获取相应的数据

库存这一块，写数据库的时候，直接更新redis缓存

实际上没有这么的简单，这里，其实就涉及到了一个问题，数据库与缓存双写，数据不一致的问题

围绕和结合实时性较高的库存服务，把数据库与缓存双写不一致问题以及其解决方案，给大家讲解一下

数据库与缓存双写不一致，很常见的问题，大型的缓存架构中，第一个解决方案

大型的缓存架构全部讲解完了以后，整套架构是非常复杂，架构可以应对各种各样奇葩和极端的情况

也有一种可能，不是说，来讲课的就是超人，万能的

讲课，就跟写书一样，很可能会写错，也可能有些方案里的一些地方，我没考虑到

也可能说，有些方案只是适合某些场景，在某些场景下，可能需要你进行方案的优化和调整才能适用于你自己的项目

大家觉得对这些方案有什么疑问或者见解，都可以找我，沟通一下

如果的确我觉得是我讲解的不对，或者有些地方考虑不周，那么我可以在视频里补录，更新到网站上面去

多多包涵

1、最初级的缓存不一致问题以及解决方案

问题：先修改数据库，再删除缓存，如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据出现不一致

解决思路

先删除缓存，再修改数据库，如果删除缓存成功了，如果修改数据库失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致

因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中

2、比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改

一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中

数据变更的程序完成了数据库的修改

完了，数据库和缓存中的数据不一样了。。。。

3、为什么上亿流量高并发场景下，缓存会出现这个问题？

只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题

其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就1万次，那么很少的情况下，会出现刚才描述的那种不一致的场景

但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就可能会出现上述的数据库+缓存不一致的情况

高并发了以后，问题是很多的

4、数据库与缓存更新与读取操作进行异步串行化

更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个jvm内部的队列中

读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm内部的队列中

一个队列对应一个工作线程

每个工作线程串行拿到对应的操作，然后一条一条的执行

这样的话，一个数据变更的操作，先执行，删除缓存，然后再去更新数据库，但是还没完成更新

此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成

这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值

5、高并发的场景下，该解决方案要注意的问题

（1）读请求长时阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回

该解决方案，最大的风险点在于说，可能数据更新很频繁，导致队列中积压了大量更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库

务必通过一些模拟真实的测试，看看更新数据的频繁是怎样的

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分摊一些数据的更新操作

如果一个内存队列里居然会挤压100个商品的库存修改操作，每隔库存修改操作要耗费10ms区完成，那么最后一个商品的读请求，可能等待10 \* 100 = 1000ms = 1s后，才能得到数据

这个时候就导致读请求的长时阻塞

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会hang多少时间，如果读请求在200ms返回，如果你计算过后，哪怕是最繁忙的时候，积压10个更新操作，最多等待200ms，那还可以的

如果一个内存队列可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少

其实根据之前的项目经验，一般来说数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的

针对读高并发，读缓存架构的项目，一般写请求相对读来说，是非常非常少的，每秒的QPS能到几百就不错了

一秒，500的写操作，5份，每200ms，就100个写操作

单机器，20个内存队列，每个内存队列，可能就积压5个写操作，每个写操作性能测试后，一般在20ms左右就完成

那么针对每个内存队列中的数据的读请求，也就最多hang一会儿，200ms以内肯定能返回了

写QPS扩大10倍，但是经过刚才的测算，就知道，单机支撑写QPS几百没问题，那么就扩容机器，扩容10倍的机器，10台机器，每个机器20个队列，200个队列

大部分的情况下，应该是这样的，大量的读请求过来，都是直接走缓存取到数据的

少量情况下，可能遇到读跟数据更新冲突的情况，如上所述，那么此时更新操作如果先入队列，之后可能会瞬间来了对这个数据大量的读请求，但是因为做了去重的优化，所以也就一个更新缓存的操作跟在它后面

等数据更新完了，读请求触发的缓存更新操作也完成，然后临时等待的读请求全部可以读到缓存中的数据

（2）读请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时hang在服务上，看服务能不能抗的住，需要多少机器才能抗住最大的极限情况的峰值

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大

按1:99的比例计算读和写的请求，每秒5万的读QPS，可能只有500次更新操作

如果一秒有500的写QPS，那么要测算好，可能写操作影响的数据有500条，这500条数据在缓存中失效后，可能导致多少读请求，发送读请求到库存服务来，要求更新缓存

一般来说，1:1，1:2，1:3，每秒钟有1000个读请求，会hang在库存服务上，每个读请求最多hang多少时间，200ms就会返回

在同一时间最多hang住的可能也就是单机200个读请求，同时hang住

单机hang200个读请求，还是ok的

1:20，每秒更新500条数据，这500秒数据对应的读请求，会有20 \* 500 = 1万

1万个读请求全部hang在库存服务上，就死定了

（3）多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过nginx服务器路由到相同的服务实例上

（4）热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能造成某台机器的压力过大

就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以更新频率不是太高的话，这个问题的影响并不是特别大

但是的确可能某些机器的负载会高一些

#### 04\_课程总结

总结

1、面试题

如何保证缓存与数据库的双写一致性？

1. 面试官心里分析

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题？

3、面试题剖析

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以保证一定不会出现不一致的情况

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

|

### redis的并发竞争问题是什么？如何解决这个问题？了解Redis事务的CAS方案吗？

01\_redis并发竞争问题以及解决方案



1、面试题

redis的并发竞争问题是什么？如何解决这个问题？了解Redis事务的CAS方案吗？

2、面试官心里分析

这个也是线上非常常见的一个问题，就是多客户端同时并发写一个key，可能本来应该先到的数据后到了，导致数据版本错了。或者是多客户端同时获取一个key，修改值之后再写回去，只要顺序错了，数据就错了。

而且redis自己就有天然解决这个问题的CAS类的乐观锁方案

3、面试题剖析

### 生产环境中的redis是怎么部署的？

1、面试题

生产环境中的redis是怎么部署的？

2、面试官心里分析

看看你了解不了解你们公司的redis生产集群的部署架构，如果你不了解，那么确实你就很失职了，你的redis是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上redis给几个G的内存？设置了哪些参数？压测后你们redis集群承载多少QPS？

兄弟，这些你必须是门儿清的，否则你确实是没好好思考过

3、面试题剖析

redis cluster，10台机器，5台机器部署了redis主实例，另外5台机器部署了redis的从实例，每个主实例挂了一个从实例，5个节点对外提供读写服务，每个节点的读写高峰qps可能可以达到每秒5万，5台机器最多是25万读写请求/s。

机器是什么配置？32G内存+8核CPU+1T磁盘，但是分配给redis进程的是10g内存，一般线上生产环境，redis的内存尽量不要超过10g，超过10g可能会有问题。

5台机器对外提供读写，一共有50g内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis从实例会自动变成主实例继续提供读写服务

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是10kb。100条数据是1mb，10万条数据是1g。常驻内存的是200万条商品数据，占用内存是20g，仅仅不到总内存的50%。

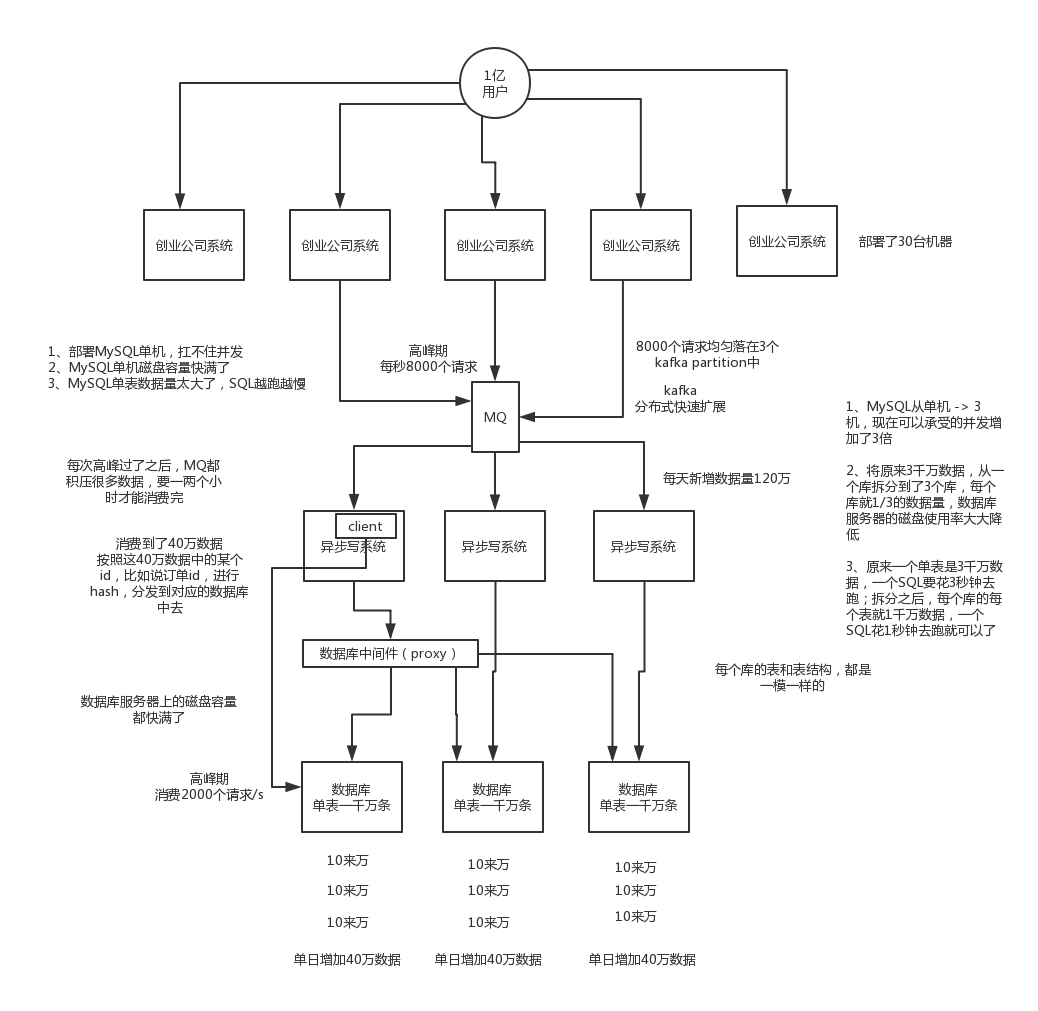
目前高峰期每秒就是3500左右的请求量

比如我们吧，大型的公司，其实基础架构的team，会负责缓存集群的运维

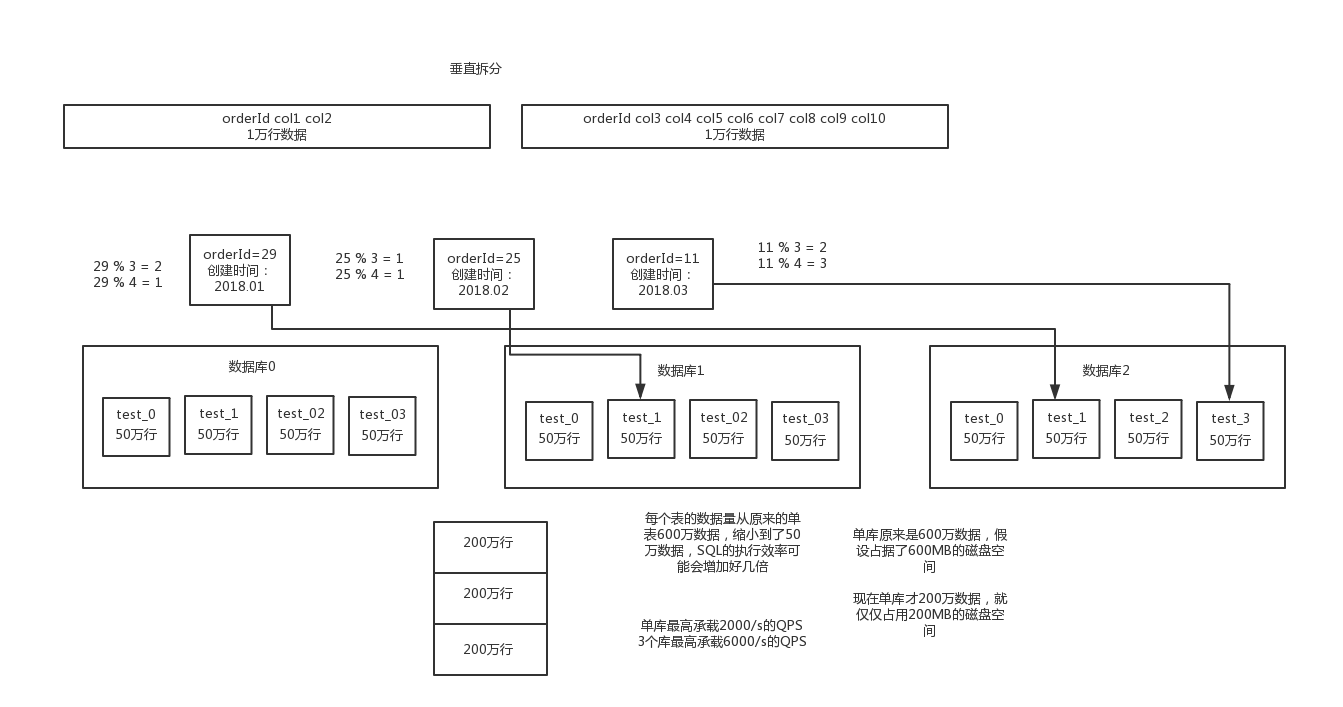
## 2.5 分库分表

## 1. 为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？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？

01\_分库分表的由来



02\_数据库如何拆分



1、面试题

为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？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？

2、面试官心里分析

其实这块肯定是扯到高并发了，因为分库分表一定是为了支撑高并发、数据量大两个问题的。而且现在说实话，尤其是互联网类的公司面试，基本上都会来这么一下，分库分表如此普遍的技术问题，不问实在是不行，而如果你不知道那也实在是说不过去！

3、面试题剖析

（1）为什么要分库分表？（设计高并发系统的时候，数据库层面该如何设计？）

说白了，分库分表是两回事儿，大家可别搞混了，可能是光分库不分表，也可能是光分表不分库，都有可能。我先给大家抛出来一个场景。

假如我们现在是一个小创业公司（或者是一个BAT公司刚兴起的一个新部门），现在注册用户就20万，每天活跃用户就1万，每天单表数据量就1000，然后高峰期每秒钟并发请求最多就10。。。天，就这种系统，随便找一个有几年工作经验的，然后带几个刚培训出来的，随便干干都可以。

结果没想到我们运气居然这么好，碰上个CEO带着我们走上了康庄大道，业务发展迅猛，过了几个月，注册用户数达到了2000万！每天活跃用户数100万！每天单表数据量10万条！高峰期每秒最大请求达到1000！同时公司还顺带着融资了两轮，紧张了几个亿人民币啊！公司估值达到了惊人的几亿美金！这是小独角兽的节奏！

好吧，没事，现在大家感觉压力已经有点大了，为啥呢？因为每天多10万条数据，一个月就多300万条数据，现在咱们单表已经几百万数据了，马上就破千万了。但是勉强还能撑着。高峰期请求现在是1000，咱们线上部署了几台机器，负载均衡搞了一下，数据库撑1000 QPS也还凑合。但是大家现在开始感觉有点担心了，接下来咋整呢。。。。。。

再接下来几个月，我的天，CEO太牛逼了，公司用户数已经达到1亿，公司继续融资几十亿人民币啊！公司估值达到了惊人的几十亿美金，成为了国内今年最牛逼的明星创业公司！天，我们太幸运了。

但是我们同时也是不幸的，因为此时每天活跃用户数上千万，每天单表新增数据多达50万，目前一个表总数据量都已经达到了两三千万了！扛不住啊！数据库磁盘容量不断消耗掉！高峰期并发达到惊人的5000~8000！别开玩笑了，哥。我跟你保证，你的系统支撑不到现在，已经挂掉了！

好吧，所以看到你这里你差不多就理解分库分表是怎么回事儿了，实际上这是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。

这就是所谓的分库分表，为啥要分库分表？你明白了吧

（2）用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？

这个其实就是看看你了解哪些分库分表的中间件，各个中间件的优缺点是啥？然后你用过哪些分库分表的中间件。

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

cobar：阿里b2b团队开发和开源的，属于proxy层方案。早些年还可以用，但是最近几年都没更新了，基本没啥人用，差不多算是被抛弃的状态吧。而且不支持读写分离、存储过程、跨库join和分页等操作。

TDDL：淘宝团队开发的，属于client层方案。不支持join、多表查询等语法，就是基本的crud语法是ok，但是支持读写分离。目前使用的也不多，因为还依赖淘宝的diamond配置管理系统。

atlas：360开源的，属于proxy层方案，以前是有一些公司在用的，但是确实有一个很大的问题就是社区最新的维护都在5年前了。所以，现在用的公司基本也很少了。

sharding-jdbc：当当开源的，属于client层方案。确实之前用的还比较多一些，因为SQL语法支持也比较多，没有太多限制，而且目前推出到了2.0版本，支持分库分表、读写分离、分布式id生成、柔性事务（最大努力送达型事务、TCC事务）。而且确实之前使用的公司会比较多一些（这个在官网有登记使用的公司，可以看到从2017年一直到现在，是不少公司在用的），目前社区也还一直在开发和维护，还算是比较活跃，个人认为算是一个现在也可以选择的方案。

mycat：基于cobar改造的，属于proxy层方案，支持的功能非常完善，而且目前应该是非常火的而且不断流行的数据库中间件，社区很活跃，也有一些公司开始在用了。但是确实相比于sharding jdbc来说，年轻一些，经历的锤炼少一些。

所以综上所述，现在其实建议考量的，就是sharding-jdbc和mycat，这两个都可以去考虑使用。

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。

通常来说，这两个方案其实都可以选用，但是我个人建议中小型公司选用sharding-jdbc，client层方案轻便，而且维护成本低，不需要额外增派人手，而且中小型公司系统复杂度会低一些，项目也没那么多；

但是中大型公司最好还是选用mycat这类proxy层方案，因为可能大公司系统和项目非常多，团队很大，人员充足，那么最好是专门弄个人来研究和维护mycat，然后大量项目直接透明使用即可。

我们，数据库中间件都是自研的，也用过proxy层，后来也用过client层

（3）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。

这个其实挺常见的，不一定我说，大家很多同学可能自己都做过，把一个大表拆开，订单表、订单支付表、订单商品表。

还有表层面的拆分，就是分表，将一个表变成N个表，就是让每个表的数据量控制在一定范围内，保证SQL的性能。否则单表数据量越大，SQL性能就越差。一般是200万行左右，不要太多，但是也得看具体你怎么操作，也可能是500万，或者是100万。你的SQL越复杂，就最好让单表行数越少。

好了，无论是分库了还是分表了，上面说的那些数据库中间件都是可以支持的。就是基本上那些中间件可以做到你分库分表之后，中间件可以根据你指定的某个字段值，比如说userid，自动路由到对应的库上去，然后再自动路由到对应的表里去。

你就得考虑一下，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。

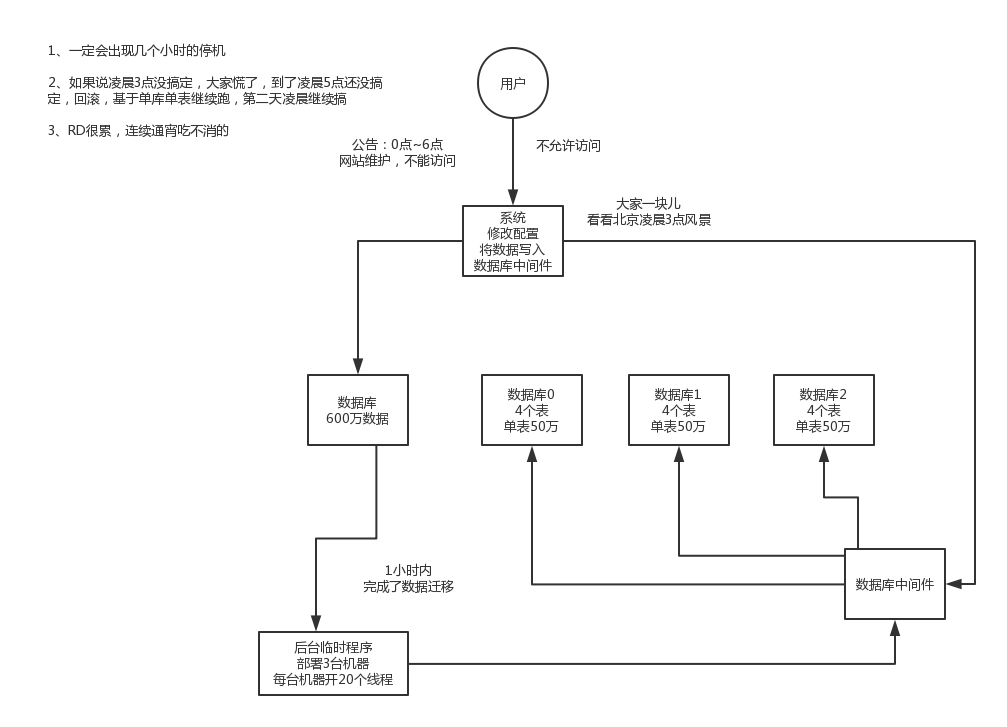
而且这儿还有两种分库分表的方式，一种是按照range来分，就是每个库一段连续的数据，这个一般是按比如时间范围来的，但是这种一般较少用，因为很容易产生热点问题，大量的流量都打在最新的数据上了；或者是按照某个字段hash一下均匀分散，这个较为常用。

range来分，好处在于说，后面扩容的时候，就很容易，因为你只要预备好，给每个月都准备一个库就可以了，到了一个新的月份的时候，自然而然，就会写新的库了；缺点，但是大部分的请求，都是访问最新的数据。实际生产用range，要看场景，你的用户不是仅仅访问最新的数据，而是均匀的访问现在的数据以及历史的数据

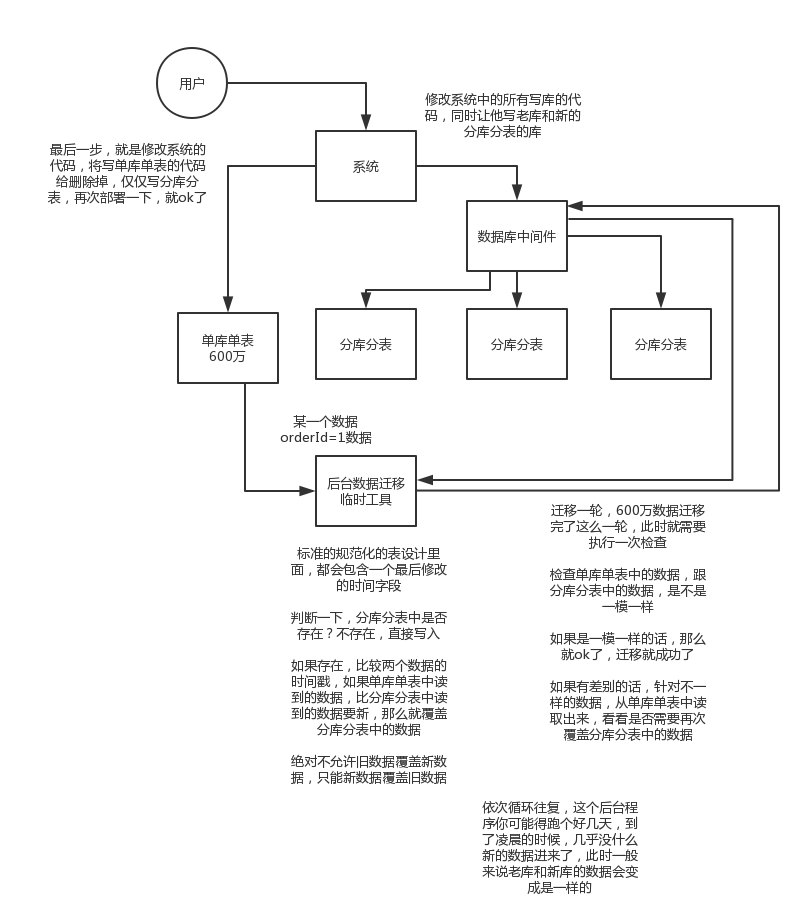
hash分法，好处在于说，可以平均分配没给库的数据量和请求压力；坏处在于说扩容起来比较麻烦，会有一个数据迁移的这么一个过程

### 2.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？

01\_长时间停机分库分表



02\_不停机双写方案



1、面试题

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？

2、面试官心里分析

你看看，你现在已经明白为啥要分库分表了，你也知道常用的分库分表中间件了，你也设计好你们如何分库分表的方案了（水平拆分、垂直拆分、分表），那问题来了，你接下来该怎么把你那个单库单表的系统给迁移到分库分表上去？

所以这都是一环扣一环的，就是看你有没有全流程经历过这个过程

友情提示

假设，你现有有一个单库单表的系统，在线上在跑，假设单表有600万数据

3个库，每个库里分了4个表，每个表要放50万的数据量

假设你已经选择了一个分库分表的数据库中间件，sharding-jdbc，mycat，都可以

你怎么把线上系统平滑地迁移到分库分表上面去

sharding-jdbc：自己上官网，找一个官网最基本的例子，自己写一下，试一下，跑跑看，是非常简单的

mycat：自己上官网，找一个官网最基本的例子，自己写一下，试一下看看

1个小时以内就可以搞定了

3、面试题剖析

这个其实从low到高大上有好几种方案，我们都玩儿过，我都给你说一下

（1）停机迁移方案

我先给你说一个最low的方案，就是很简单，大家伙儿凌晨12点开始运维，网站或者app挂个公告，说0点到早上6点进行运维，无法访问。。。。。。

接着到0点，停机，系统挺掉，没有流量写入了，此时老的单库单表数据库静止了。然后你之前得写好一个导数的一次性工具，此时直接跑起来，然后将单库单表的数据哗哗哗读出来，写到分库分表里面去。

导数完了之后，就ok了，修改系统的数据库连接配置啥的，包括可能代码和SQL也许有修改，那你就用最新的代码，然后直接启动连到新的分库分表上去。

验证一下，ok了，完美，大家伸个懒腰，看看看凌晨4点钟的北京夜景，打个滴滴回家吧

但是这个方案比较low，谁都能干，我们来看看高大上一点的方案

（2）双写迁移方案

这个是我们常用的一种迁移方案，比较靠谱一些，不用停机，不用看北京凌晨4点的风景

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。

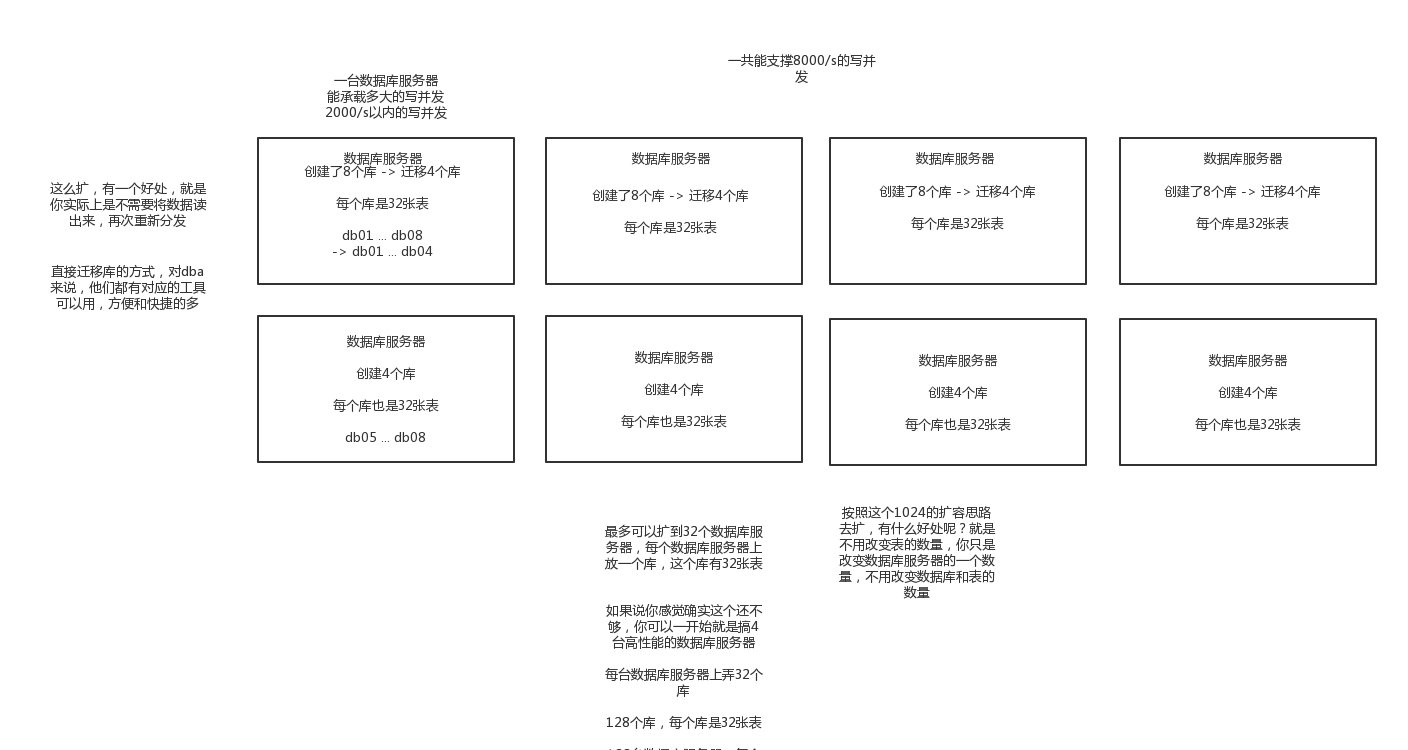
然后系统部署之后，新库数据差太远，用之前说的导数工具，跑起来读老库数据写新库，写的时候要根据gmt\_modified这类字段判断这条数据最后修改的时间，除非是读出来的数据在新库里没有，或者是比新库的数据新才会写。

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。

### 3.如何设计可以动态扩容缩容的分库分表方案？

01\_分库分表扩容方案



1、面试题

如何设计可以动态扩容缩容的分库分表方案？

2、面试官心里分析

（1）选择一个数据库中间件，调研、学习、测试

（2）设计你的分库分表的一个方案，你要分成多少个库，每个库分成多少个表，3个库每个库4个表

（3）基于选择好的数据库中间件，以及在测试环境建立好的分库分表的环境，然后测试一下能否正常进行分库分表的读写

（4）完成单库单表到分库分表的迁移，双写方案

（5）线上系统开始基于分库分表对外提供服务

（6）扩容了，扩容成6个库，每个库需要12个表，你怎么来增加更多库和表呢？

这个是你必须面对的一个事儿，就是你已经弄好分库分表方案了，然后一堆库和表都建好了，基于分库分表中间件的代码开发啥的都好了，测试都ok了，数据能均匀分布到各个库和各个表里去，而且接着你还通过双写的方案咔嚓一下上了系统，已经直接基于分库分表方案在搞了。

那么现在问题来了，你现在这些库和表又支撑不住了，要继续扩容咋办？这个可能就是说你的每个库的容量又快满了，或者是你的表数据量又太大了，也可能是你每个库的写并发太高了，你得继续扩容。

这都是玩儿分库分表线上必须经历的事儿

3、面试题剖析

（1）停机扩容

这个方案就跟停机迁移一样，步骤几乎一致，唯一的一点就是那个导数的工具，是把现有库表的数据抽出来慢慢倒入到新的库和表里去。但是最好别这么玩儿，有点不太靠谱，因为既然分库分表就说明数据量实在是太大了，可能多达几亿条，甚至几十亿，你这么玩儿，可能会出问题。

从单库单表迁移到分库分表的时候，数据量并不是很大，单表最大也就两三千万

写个工具，多弄几台机器并行跑，1小时数据就导完了

3个库+12个表，跑了一段时间了，数据量都1亿~2亿了。光是导2亿数据，都要导个几个小时，6点，刚刚导完数据，还要搞后续的修改配置，重启系统，测试验证，10点才可以搞完

（2）优化后的方案

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。

比如说最多可以扩展到32个数据库服务器，每个数据库服务器是一个库。如果还是不够？最多可以扩展到1024个数据库服务器，每个数据库服务器上面一个库一个表。因为最多是1024个表么。

这么搞，是不用自己写代码做数据迁移的，都交给dba来搞好了，但是dba确实是需要做一些库表迁移的工作，但是总比你自己写代码，抽数据导数据来的效率高得多了。

哪怕是要减少库的数量，也很简单，其实说白了就是按倍数缩容就可以了，然后修改一下路由规则。

对2 ^ n取模

orderId 模 32 = 库

orderId / 32 模 32 = 表

259 3 8

1189 5 5

352 0 11

4593 17 15

1、设定好几台数据库服务器，每台服务器上几个库，每个库多少个表，推荐是32库 \* 32表，对于大部分公司来说，可能几年都够了

2、路由的规则，orderId 模 32 = 库，orderId / 32 模 32 = 表

3、扩容的时候，申请增加更多的数据库服务器，装好mysql，倍数扩容，4台服务器，扩到8台服务器，16台服务器

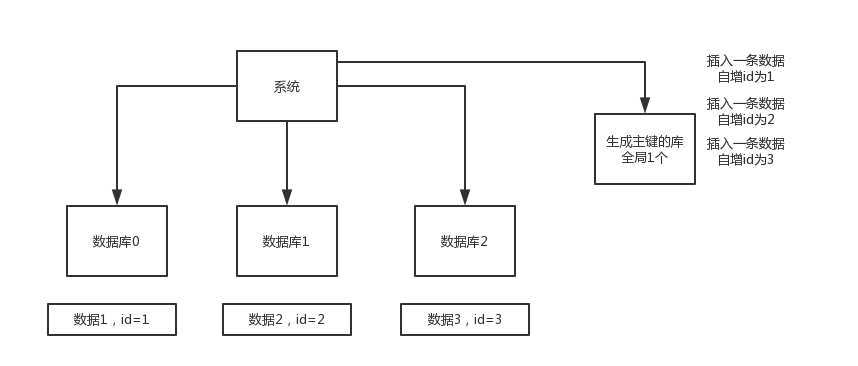
4、由dba负责将原先数据库服务器的库，迁移到新的数据库服务器上去，很多工具，库迁移，比较便捷

5、我们这边就是修改一下配置，调整迁移的库所在数据库服务器的地址

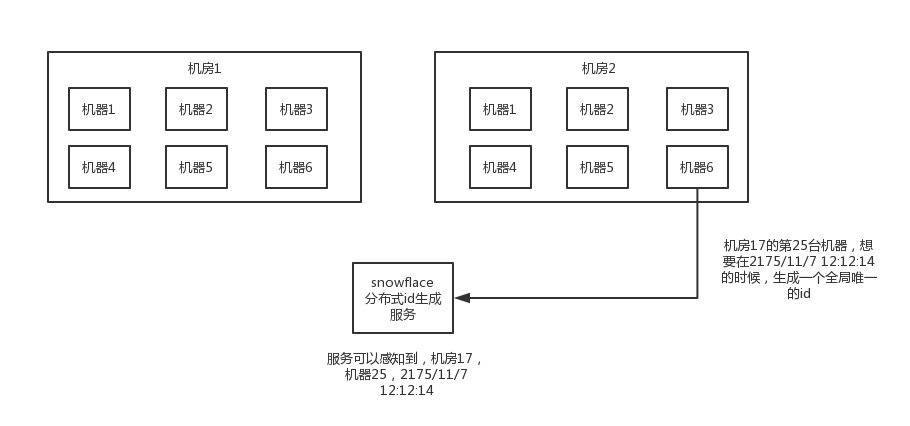
6、重新发布系统，上线，原先的路由规则变都不用变，直接可以基于2倍的数据库服务器的资源，继续进行线上系统的提供服务

### 4.分库分表之后，id主键如何处理？

01\_分库分表的id主键问题



02\_snowflake算法



1、面试题

分库分表之后，id主键如何处理？

2、面试官心里分析

其实这是分库分表之后你必然要面对的一个问题，就是id咋生成？因为要是分成多个表之后，每个表都是从1开始累加，那肯定不对啊，需要一个全局唯一的id来支持。所以这都是你实际生产环境中必须考虑的问题。

3、面试题剖析

（1）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；但是无论怎么说都是基于单个数据库。

适合的场景：你分库分表就俩原因，要不就是单库并发太高，要不就是单库数据量太大；除非是你并发不高，但是数据量太大导致的分库分表扩容，你可以用这个方案，因为可能每秒最高并发最多就几百，那么就走单独的一个库和表生成自增主键即可。

并发很低，几百/s，但是数据量大，几十亿的数据，所以需要靠分库分表来存放海量的数据

（2）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，不适合用于主键。

适合的场景：如果你是要随机生成个什么文件名了，编号之类的，你可以用uuid，但是作为主键是不能用uuid的。

UUID.randomUUID().toString().replace(“-”, “”) -> sfsdf23423rr234sfdaf

（3）获取系统当前时间

这个就是获取当前时间即可，但是问题是，并发很高的时候，比如一秒并发几千，会有重复的情况，这个是肯定不合适的。基本就不用考虑了。

适合的场景：一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个id，如果业务上你觉得可以接受，那么也是可以的。你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号，订单编号，时间戳 + 用户id + 业务含义编码

（4）snowflake算法

twitter开源的分布式id生成算法，就是把一个64位的long型的id，1个bit是不用的，用其中的41 bit作为毫秒数，用10 bit作为工作机器id，12 bit作为序列号

1 bit：不用，为啥呢？因为二进制里第一个bit为如果是1，那么都是负数，但是我们生成的id都是正数，所以第一个bit统一都是0

41 bit：表示的是时间戳，单位是毫秒。41 bit可以表示的数字多达2^41 - 1，也就是可以标识2 ^ 41 - 1个毫秒值，换算成年就是表示69年的时间。

10 bit：记录工作机器id，代表的是这个服务最多可以部署在2^10台机器上哪，也就是1024台机器。但是10 bit里5个bit代表机房id，5个bit代表机器id。意思就是最多代表2 ^ 5个机房（32个机房），每个机房里可以代表2 ^ 5个机器（32台机器）。

12 bit：这个是用来记录同一个毫秒内产生的不同id，12 bit可以代表的最大正整数是2 ^ 12 - 1 = 4096，也就是说可以用这个12bit代表的数字来区分同一个毫秒内的4096个不同的id

64位的long型的id，64位的long -> 二进制

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

2018-01-01 10:00:00 -> 做了一些计算，再换算成一个二进制，41bit来放 -> 0001100 10100010 10111110 10001001 01011100 00

机房id，17 -> 换算成一个二进制 -> 10001

机器id，25 -> 换算成一个二进制 -> 11001

snowflake算法服务，会判断一下，当前这个请求是否是，机房17的机器25，在2175/11/7 12:12:14时间点发送过来的第一个请求，如果是第一个请求

假设，在2175/11/7 12:12:14时间里，机房17的机器25，发送了第二条消息，snowflake算法服务，会发现说机房17的机器25，在2175/11/7 12:12:14时间里，在这一毫秒，之前已经生成过一个id了，此时如果你同一个机房，同一个机器，在同一个毫秒内，再次要求生成一个id，此时我只能把加1

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000001

比如我们来观察上面的那个，就是一个典型的二进制的64位的id，换算成10进制就是910499571847892992。

public class IdWorker{

private long workerId;

private long datacenterId;

private long sequence;

public IdWorker(long workerId, long datacenterId, long sequence){

// sanity check for workerId

// 这儿不就检查了一下，要求就是你传递进来的机房id和机器id不能超过32，不能小于0

if (workerId > maxWorkerId || workerId < 0) {

throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0",maxWorkerId));

}

if (datacenterId > maxDatacenterId || datacenterId < 0) {

throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0",maxDatacenterId));

}

System.out.printf("worker starting. timestamp left shift %d, datacenter id bits %d, worker id bits %d, sequence bits %d, workerid %d",

timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId);

this.workerId = workerId;

this.datacenterId = datacenterId;

this.sequence = sequence;

}

private long twepoch = 1288834974657L;

private long workerIdBits = 5L;

private long datacenterIdBits = 5L;

private long maxWorkerId = -1L ^ (-1L << workerIdBits); // 这个是二进制运算，就是5 bit最多只能有31个数字，也就是说机器id最多只能是32以内

private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits); // 这个是一个意思，就是5 bit最多只能有31个数字，机房id最多只能是32以内

private long sequenceBits = 12L;

private long workerIdShift = sequenceBits;

private long datacenterIdShift = sequenceBits + workerIdBits;

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

private long sequenceMask = -1L ^ (-1L << sequenceBits);

private long lastTimestamp = -1L;

public long getWorkerId(){

return workerId;

}

public long getDatacenterId(){

return datacenterId;

}

public long getTimestamp(){

return System.currentTimeMillis();

}

public synchronized long nextId() {

// 这儿就是获取当前时间戳，单位是毫秒

long timestamp = timeGen();

if (timestamp < lastTimestamp) {

System.err.printf("clock is moving backwards. Rejecting requests until %d.", lastTimestamp);

throw new RuntimeException(String.format("Clock moved backwards. Refusing to generate id for %d milliseconds",

lastTimestamp - timestamp));

}

// 0

// 在同一个毫秒内，又发送了一个请求生成一个id，0 -> 1

if (lastTimestamp == timestamp) {

sequence = (sequence + 1) & sequenceMask; // 这个意思是说一个毫秒内最多只能有4096个数字，无论你传递多少进来，这个位运算保证始终就是在4096这个范围内，避免你自己传递个sequence超过了4096这个范围

if (sequence == 0) {

timestamp = tilNextMillis(lastTimestamp);

}

} else {

sequence = 0;

}

// 这儿记录一下最近一次生成id的时间戳，单位是毫秒

lastTimestamp = timestamp;

// 这儿就是将时间戳左移，放到41 bit那儿；将机房id左移放到5 bit那儿；将机器id左移放到5 bit那儿；将序号放最后10 bit；最后拼接起来成一个64 bit的二进制数字，转换成10进制就是个long型

return ((timestamp - twepoch) << timestampLeftShift) |

(datacenterId << datacenterIdShift) |

(workerId << workerIdShift) |

sequence;

}

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

private long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {

timestamp = timeGen();

}

return timestamp;

}

private long timeGen(){

return System.currentTimeMillis();

}

//---------------测试---------------

public static void main(String[] args) {

IdWorker worker = new IdWorker(1,1,1);

for (int i = 0; i < 30; i++) {

System.out.println(worker.nextId());

}

}

}

怎么说呢，大概这个意思吧，就是说41 bit，就是当前毫秒单位的一个时间戳，就这意思；然后5 bit是你传递进来的一个机房id（但是最大只能是32以内），5 bit是你传递进来的机器id（但是最大只能是32以内），剩下的那个10 bit序列号，就是如果跟你上次生成id的时间还在一个毫秒内，那么会把顺序给你累加，最多在4096个序号以内。

所以你自己利用这个工具类，自己搞一个服务，然后对每个机房的每个机器都初始化这么一个东西，刚开始这个机房的这个机器的序号就是0。然后每次接收到一个请求，说这个机房的这个机器要生成一个id，你就找到对应的Worker，生成。

他这个算法生成的时候，会把当前毫秒放到41 bit中，然后5 bit是机房id，5 bit是机器id，接着就是判断上一次生成id的时间如果跟这次不一样，序号就自动从0开始；要是上次的时间跟现在还是在一个毫秒内，他就把seq累加1，就是自动生成一个毫秒的不同的序号。

这个算法那，可以确保说每个机房每个机器每一毫秒，最多生成4096个不重复的id。

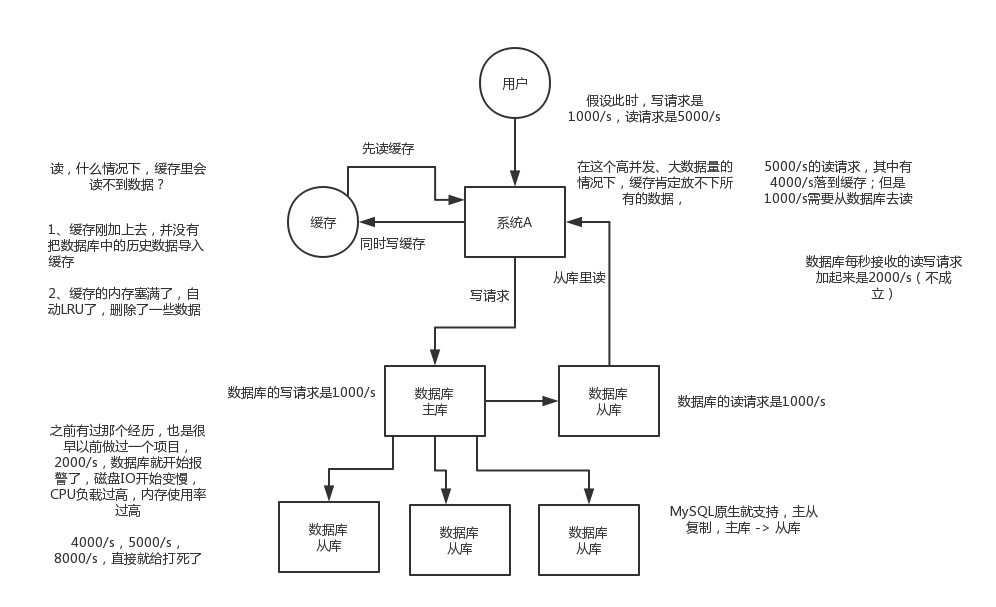
利用这个snowflake算法，你可以开发自己公司的服务，甚至对于机房id和机器id，反正给你预留了5 bit + 5 bit，你换成别的有业务含义的东西也可以的。

这个snowflake算法相对来说还是比较靠谱的，所以你要真是搞分布式id生成，如果是高并发啥的，那么用这个应该性能比较好，一般每秒几万并发的场景，也足够你用了。

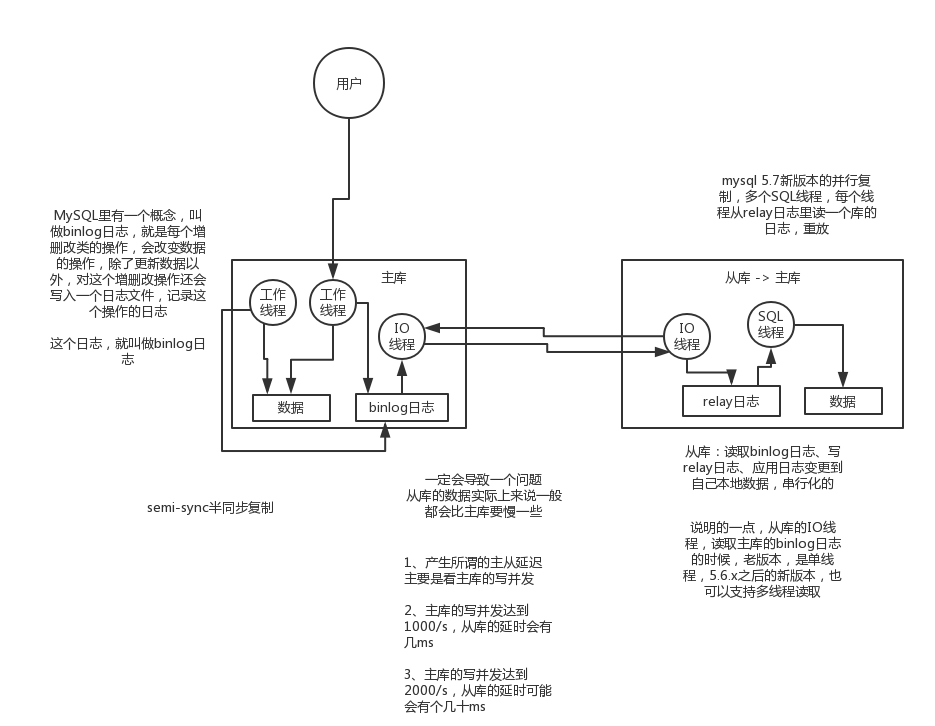
## 2.6 读写分离

### （1）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？

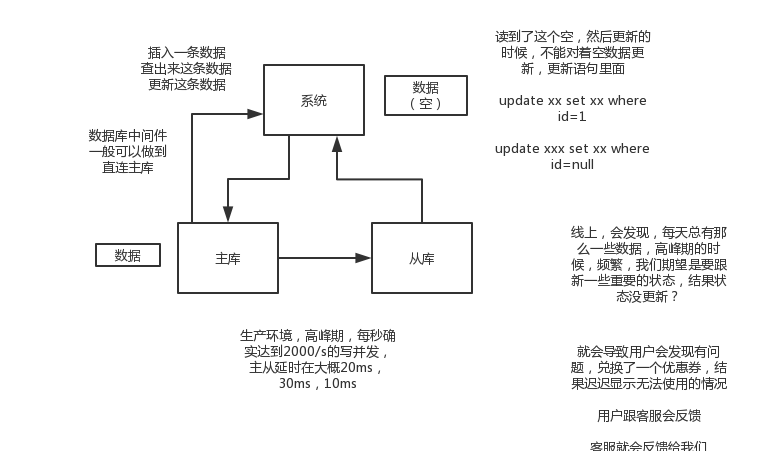
01\_为什么MySQL要读写分离？



02\_MySQL主从复制原理



03\_MySQL主从延迟导致的生产环境的问题



1、面试题

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？

2、面试官心里分析

这个，高并发这个阶段，那肯定是需要做读写分离的，啥意思？因为实际上大部分的互联网公司，一些网站，或者是app，其实都是读多写少。所以针对这个情况，就是写一个主库，但是主库挂多个从库，然后从多个从库来读，那不就可以支撑更高的读并发压力了吗？

3、面试题剖析

（1）如何实现mysql的读写分离？

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。

（2）MySQL主从复制原理的是啥？

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。

而且这里还有另外一个问题，就是如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

所以mysql实际上在这一块有两个机制，一个是半同步复制，用来解决主库数据丢失问题；一个是并行复制，用来解决主从同步延时问题。

这个所谓半同步复制，semi-sync复制，指的就是主库写入binlog日志之后，就会将强制此时立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库的ack之后才会认为写操作完成了。

所谓并行复制，指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。

1. 主从复制的原理
2. 主从延迟问题产生的原因
3. 主从复制的数据丢失问题，以及半同步复制的原理
4. 并行复制的原理，多库并发重放relay日志，缓解主从延迟问题

（3）mysql主从同步延时问题（精华）

线上确实处理过因为主从同步延时问题，导致的线上的bug，小型的生产事故

show status，Seconds\_Behind\_Master，你可以看到从库复制主库的数据落后了几ms

其实这块东西我们经常会碰到，就比如说用了mysql主从架构之后，可能会发现，刚写入库的数据结果没查到，结果就完蛋了。。。。

所以实际上你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

所以这个时候，我们可以考虑的一个事情就是，你可以用mysql的并行复制，但是问题是那是库级别的并行，所以有时候作用不是很大

所以这个时候。。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，这样就可以保证你肯定的可以读到数据了吧。其实用一些数据库中间件是没问题的。

一般来说，如果主从延迟较为严重

1. 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
2. 打开mysql支持的并行复制，多个库并行复制，如果说某个库的写入并发就是特别高，单库写并发达到了2000/s，并行复制还是没意义。28法则，很多时候比如说，就是少数的几个订单表，写入了2000/s，其他几十个表10/s。
3. 重写代码，写代码的同学，要慎重，当时我们其实短期是让那个同学重写了一下代码，插入数据之后，直接就更新，不要查询
4. 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库。不推荐这种方法，你这么搞导致读写分离的意义就丧失了

# 3、高可用架构

## 3.1 如何设计一个高可用系统？

## 3.2 限流

### 如何限流？在工作中是怎么做的？说一下具体的实现？

## 3.3 熔断

### （1）如何进行熔断？熔断框架都有哪些？具体实现原理知道吗？

## 3.4 降级

### （1）如何进行降级？

**4\_如何设计高可用系统架构？限流？熔断？降级？什么鬼！**

#### 01\_课程介绍

#### 02\_hystrix与高可用系统架构：资源隔离+限流+熔断+降级+运维监控

前半部分，专注在高并发这一块，缓存架构，承载高并发，在各种高并发导致的令人崩溃/异常的场景下，运行着

缓存架构，高可用性，在各种系统的各个地方有乱七八糟的异常和故障的情况下，整套缓存系统还能继续健康的run着

HA，HAProxy，主备服务间的切换，这就做到了高可用性，主备实例，多冗余实例，高可用最最基础的东西

什么样的情况下，可能会导致系统的崩溃，以及系统不可用，针对各种各样的一些情况，然后我们用什么技术，去保护整个系统处于高可用的一个情况下

1、hystrix是什么？

netflix（国外最大的类似于，爱奇艺，优酷）视频网站，五六年前，也是，感觉自己的系统，整个网站，经常出故障，可用性不太高

有时候一些vip会员不能支付，有时候看视频就卡顿，看不了视频。。。

影响公司的收入。。。

五六年前，netflix，api team，提升高可用性，开发了一个框架，类似于spring，mybatis，hibernate，等等这种框架

高可用性的框架，hystrix

hystrix，框架，提供了高可用相关的各种各样的功能，然后确保说在hystrix的保护下，整个系统可以长期处于高可用的状态，100%，99.99999%

最理想的状况下，软件的故障，就不应该说导致整个系统的崩溃，服务器硬件的一些故障，服务的冗余

唯一有可能导致系统彻底崩溃，就是类似于之前，支付宝的那个事故，工人施工，挖断了电缆，导致几个机房都停电

不可用，和产生一些故障或者bug的区别

2、高可用系统架构

资源隔离、限流、熔断、降级、运维监控

资源隔离：让你的系统里，某一块东西，在故障的情况下，不会耗尽系统所有的资源，比如线程资源

我实际的项目中的一个case，有一块东西，是要用多线程做一些事情，小伙伴做项目的时候，没有太留神，资源隔离，那块代码，在遇到一些故障的情况下，每个线程在跑的时候，因为那个bug，直接就死循环了，导致那块东西启动了大量的线程，每个线程都死循环

最终导致我的系统资源耗尽，崩溃，不工作，不可用，废掉了

资源隔离，那一块代码，最多最多就是用掉10个线程，不能再多了，就废掉了，限定好的一些资源

限流：高并发的流量涌入进来，比如说突然间一秒钟100万QPS，废掉了，10万QPS进入系统，其他90万QPS被拒绝了

熔断：系统后端的一些依赖，出了一些故障，比如说mysql挂掉了，每次请求都是报错的，熔断了，后续的请求过来直接不接收了，拒绝访问，10分钟之后再尝试去看看mysql恢复没有

降级：mysql挂了，系统发现了，自动降级，从内存里存的少量数据中，去提取一些数据出来

运维监控：监控+报警+优化，各种异常的情况，有问题就及时报警，优化一些系统的配置和参数，或者代码

3、如何讲解这块内容？

（1）如何将eshop-cache，核心的缓存服务改造成高可用的架构

（2）hystrix中的一部分内容，单拉出来，做成一个免费的小课程，作为福利发放出去

（3）eshop-cache，写代码，eshop-cache-ha，业务场景，跟之前衔接起来，重新去写代码

（4）hystrix做服务高可用这一块的内容，讲解成只有一个业务背景，重新写代码，独立

eshop-cache，在各级缓存数据都失效的情况下，会重新从源系统中调用接口，依赖源系统去查询mysql数据库去重新获取数据

如果你的各种依赖的服务有了故障，那么很可能会导致你的系统不可用

hystrix对系统进行各种高可用性的系统加固，来应对各种不可用的情况

缓存雪崩那一块去讲解，redis肯定挂，mysql有较大概率挂掉，在风雨飘摇中

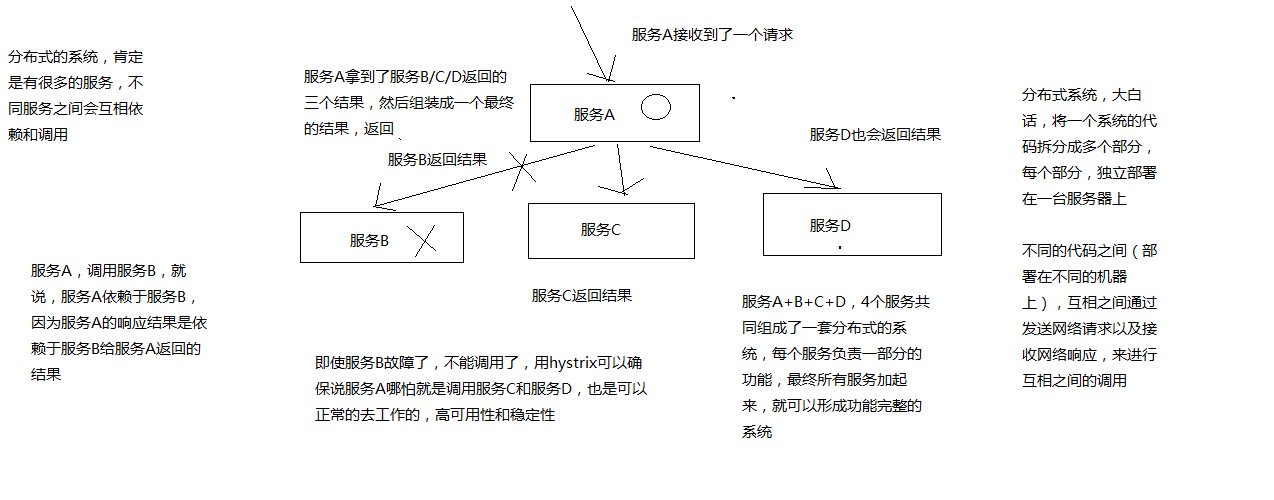
我之前做的一个项目，我们多个项目都用了公司里公用的缓存的存储，缓存彻底挂了，雪崩了，导致各种业务系统全部崩溃，崩溃了好几个小时

导致公司损失了大量的资金的损失

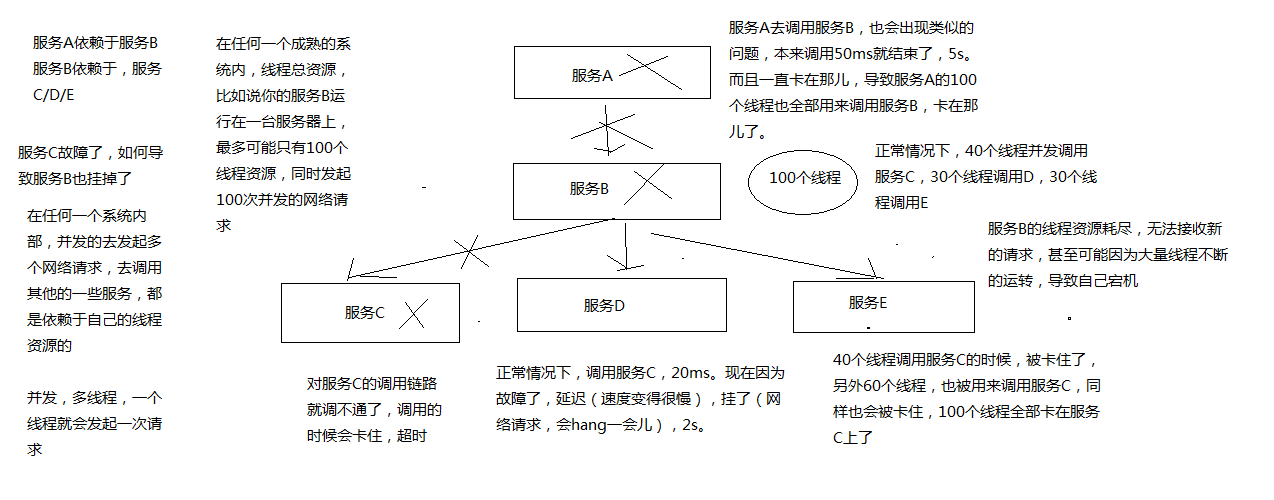
其中导致公司损失最大的负责人，受到了很大的处分

#### 03\_hystrix要解决的分布式系统可用性问题以及其设计原则

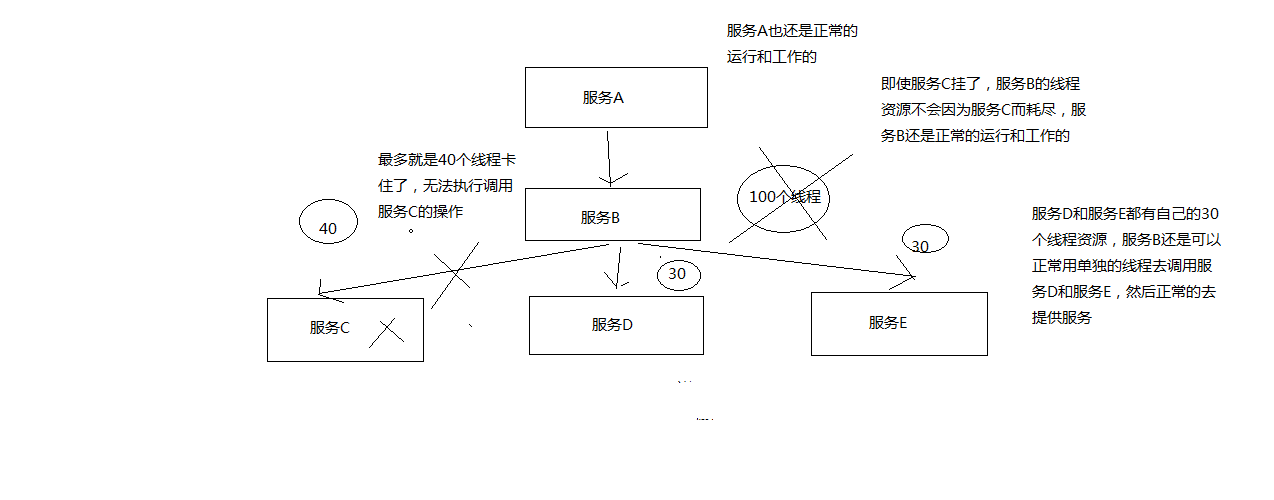
什么是分布式系统以及其中的故障和hystrix



依赖服务的故障导致服务被拖垮以及故障的蔓延



资源隔离如何保护依赖服务的故障不要拖垮整个系统



高可用性这个topic，然后咱们会用几讲的时间来讲解一下如何用hystrix，来构建高可用的服务的架构

咱们会用一个真实的项目背景，作为业务场景，来带出来在这个特定的业务场景下，可能会产生哪些各种各样的可用性的一些问题

针对这些问题，我们用hystrix的解决方案和原理是什么

带着大家，纯手工将所有的服务的高可用架构的代码，全部纯手工自己敲出来

形成高可用服务架构的项目实战的一个课程

1、Hystrix是什么？

在分布式系统中，每个服务都可能会调用很多其他服务，被调用的那些服务就是依赖服务，有的时候某些依赖服务出现故障也是很正常的。

Hystrix可以让我们在分布式系统中对服务间的调用进行控制，加入一些调用延迟或者依赖故障的容错机制。

Hystrix通过将依赖服务进行资源隔离，进而组织某个依赖服务出现故障的时候，这种故障在整个系统所有的依赖服务调用中进行蔓延，同时Hystrix还提供故障时的fallback降级机制

总而言之，Hystrix通过这些方法帮助我们提升分布式系统的可用性和稳定性

2、Hystrix的历史

hystrix，就是一种高可用保障的一个框架，类似于spring（ioc，mvc），mybatis，activiti，lucene，框架，预先封装好的为了解决某个特定领域的特定问题的一套代码库

框架，用了框架之后，来解决这个领域的特定的问题，就可以大大减少我们的工作量，提升我们的工作质量和工作效率，框架

hystrix，高可用性保障的一个框架

Netflix（可以认为是国外的优酷或者爱奇艺之类的视频网站），API团队从2011年开始做一些提升系统可用性和稳定性的工作，Hystrix就是从那时候开始发展出来的。

在2012年的时候，Hystrix就变得比较成熟和稳定了，Netflix中，除了API团队以外，很多其他的团队都开始使用Hystrix。

时至今日，Netflix中每天都有数十亿次的服务间调用，通过Hystrix框架在进行，而Hystrix也帮助Netflix网站提升了整体的可用性和稳定性

3、初步看一看Hystrix的设计原则是什么？

hystrix为了实现高可用性的架构，设计hystrix的时候，一些设计原则是什么？？？

（1）对依赖服务调用时出现的调用延迟和调用失败进行控制和容错保护

（2）在复杂的分布式系统中，阻止某一个依赖服务的故障在整个系统中蔓延，服务A->服务B->服务C，服务C故障了，服务B也故障了，服务A故障了，整套分布式系统全部故障，整体宕机

（3）提供fail-fast（快速失败）和快速恢复的支持

（4）提供fallback优雅降级的支持

（5）支持近实时的监控、报警以及运维操作

调用延迟+失败，提供容错

阻止故障蔓延

快速失败+快速恢复

降级

监控+报警+运维

完全描述了hystrix的功能，提供整个分布式系统的高可用的架构

4、Hystrix要解决的问题是什么？

在复杂的分布式系统架构中，每个服务都有很多的依赖服务，而每个依赖服务都可能会故障

如果服务没有和自己的依赖服务进行隔离，那么可能某一个依赖服务的故障就会拖垮当前这个服务

举例来说，某个服务有30个依赖服务，每个依赖服务的可用性非常高，已经达到了99.99%的高可用性

那么该服务的可用性就是99.99%的30次方，也就是99.7%的可用性

99.7%的可用性就意味着3%的请求可能会失败，因为3%的时间内系统可能出现了故障不可用了

对于1亿次访问来说，3%的请求失败，也就意味着300万次请求会失败，也意味着每个月有2个小时的时间系统是不可用的

在真实生产环境中，可能更加糟糕

上面也就是说，即使你每个依赖服务都是99.99%高可用性，但是一旦你有几十个依赖服务，还是会导致你每个月都有几个小时是不可用的

画图分析说，当某一个依赖服务出现了调用延迟或者调用失败时，为什么会拖垮当前这个服务？以及在分布式系统中，故障是如何快速蔓延的？

5、再看Hystrix的更加细节的设计原则是什么？

（1）阻止任何一个依赖服务耗尽所有的资源，比如tomcat中的所有线程资源

（2）避免请求排队和积压，采用限流和fail fast来控制故障

（3）提供fallback降级机制来应对故障

（4）使用资源隔离技术，比如bulkhead（舱壁隔离技术），swimlane（泳道技术），circuit breaker（短路技术），来限制任何一个依赖服务的故障的影响

（5）通过近实时的统计/监控/报警功能，来提高故障发现的速度

（6）通过近实时的属性和配置热修改功能，来提高故障处理和恢复的速度

（7）保护依赖服务调用的所有故障情况，而不仅仅只是网络故障情况

调用这个依赖服务的时候，client调用包有bug，阻塞，等等，依赖服务的各种各样的调用的故障，都可以处理

6、Hystrix是如何实现它的目标的？

（1）通过HystrixCommand或者HystrixObservableCommand来封装对外部依赖的访问请求，这个访问请求一般会运行在独立的线程中，资源隔离

（2）对于超出我们设定阈值的服务调用，直接进行超时，不允许其耗费过长时间阻塞住。这个超时时间默认是99.5%的访问时间，但是一般我们可以自己设置一下

（3）为每一个依赖服务维护一个独立的线程池，或者是semaphore，当线程池已满时，直接拒绝对这个服务的调用

（4）对依赖服务的调用的成功次数，失败次数，拒绝次数，超时次数，进行统计

（5）如果对一个依赖服务的调用失败次数超过了一定的阈值，自动进行熔断，在一定时间内对该服务的调用直接降级，一段时间后再自动尝试恢复

（6）当一个服务调用出现失败，被拒绝，超时，短路等异常情况时，自动调用fallback降级机制

（7）对属性和配置的修改提供近实时的支持

画图分析，对依赖进行资源隔离后，如何避免依赖服务调用延迟或失败导致当前服务的故障

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

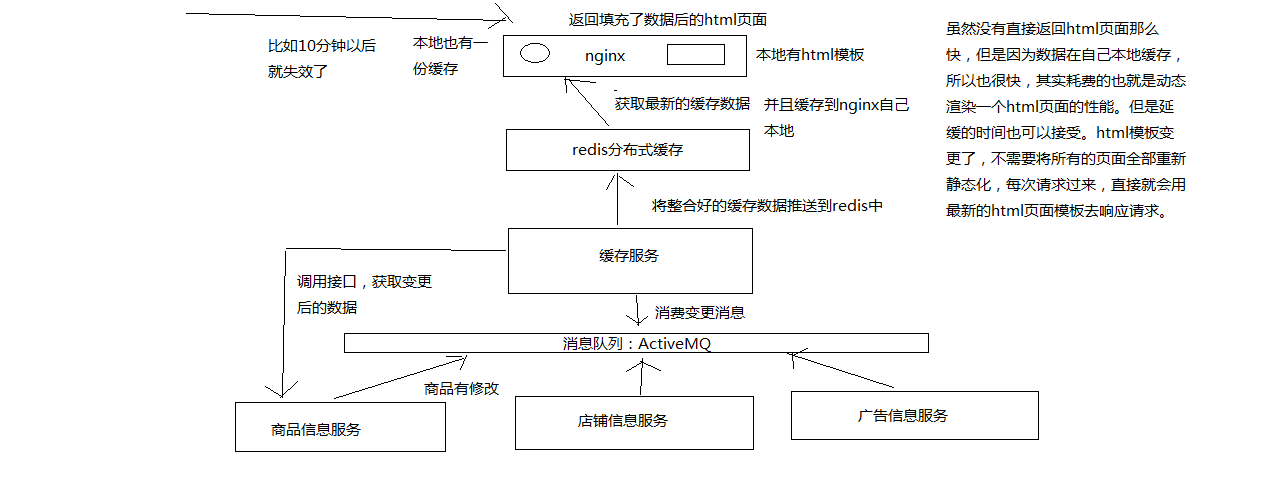
13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

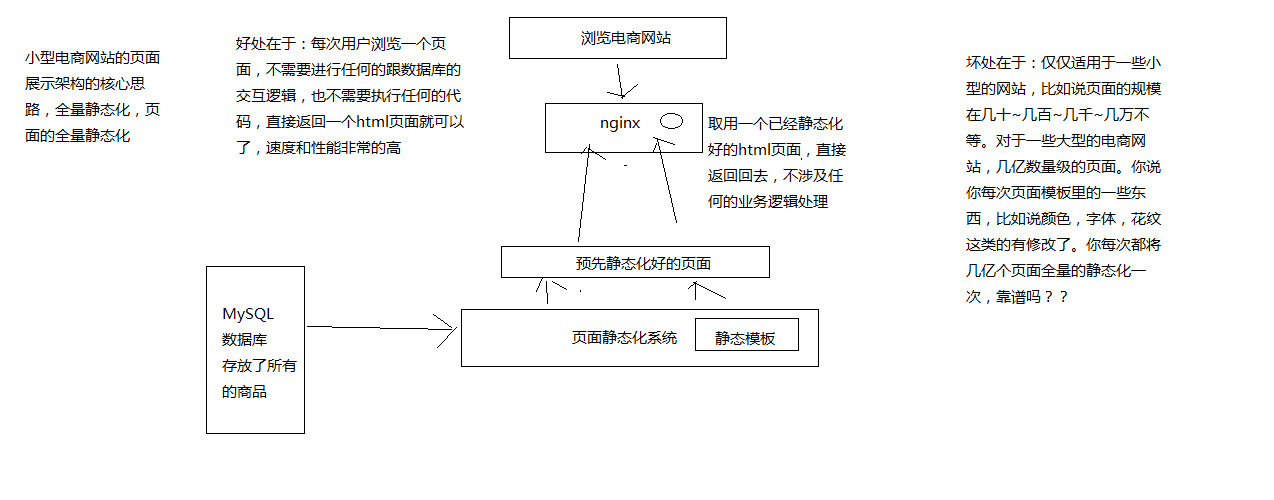
15、高并发场景下的缓存失效解决方案

#### 04\_电商网站的商品详情页缓存服务业务背景以及框架结构说明

大型电商网站的详情页系统的架构



小型电商网站的静态化方案



我们这个课程，基于hystrix，如何来构建高可用的分布式系统的架构，项目实战

模拟真实业务的这么一个小型的项目，来全程贯穿，用这个项目中的业务场景去一个一个的讲解hystrix高可用的每个技术

纯讲hystrix，脱离实际的业务背景，听起来有点枯燥，大家学完了hystrix以后，可能没法完全感受到技术是如何融入我们的项目中的

大背景：电商网站，首页，商品详情页，搜索结果页，广告页，促销活动，购物车，订单系统，库存系统，物流系统

小背景：商品详情页，如何用最快的结果将商品数据填充到一个页面中，然后将页面显示出来

分布式系统：商品详情页，缓存服务，+底层源数据服务，商品信息服务，店铺信息服务，广告信息服务，推荐信息服务，综合起来组成一个分布式的系统

1、电商网站的商品详情页系统架构

（1）小型电商网站的商品详情页系统架构（不是我们要讲解的）

（2）大型电商网站的商品详情页系统架构

（3）页面模板

举个例子

将数据动态填充/渲染到一个html模板中，是什么意思呢？

<html>

<title>#{name}的页面</title>

<body>

商品的价格是：#{price}

商品的介绍：#{description}

</body>

</html>

上面这个就可以认为是一个页面模板，里面的很多内容是不确定的，#{name}，#{price}，#{description}，这都是一些模板脚本，不确定里面的值是什么？

将数据填充/渲染到html模板中，是什么意思呢？

{

"name": "iphone7 plus（玫瑰金+32G）",

"price": 5599.50

"description": "这个手机特别好用。。。。。。"

}

<html>

<title>iphone7 plus（玫瑰金+32G）的页面</title>

<body>

商品的价格是：5599.50

商品的介绍：这个手机特别好用。。。。。。

</body>

</html>

上面这个就是一份填充好数据的一个html页面

2、缓存服务

缓存服务，订阅一个MQ的消息变更，如果有消息变更的话，那么就会发送一个网络请求，调用一个底层的对应的源数据服务的接口，去获取变更后的数据

将获取到的变更后的数据填充到分布式的redis缓存中去

高可用这一块儿，最可能出现说可用性不高的情况，是什么呢？就是说，在接收到消息之后，可能在调用各种底层依赖服务的接口时，会遇到各种不稳定的情况

比如底层服务的接口调用超时，200ms，2s都没有返回; 底层服务的接口调用失败，比如说卡了500ms之后，返回一个报错

在分布式系统中，对于这种大量的底层依赖服务的调用，就可能会出现各种可用性的问题，一旦没有处理好的话

可能就会导致缓存服务自己本身会挂掉，或者故障掉，就会导致什么呢？不可以对外提供服务，严重情况下，甚至会导致说整个商品详情页显示不出来

缓存服务接收到变更消息后，去调用各个底层依赖服务时的高可用架构的实现

我们刚才讲解的整套大型电商网站的商品详情页的缓存架构，完整的那个流程，《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

3、框架结构

围绕着缓存服务去拉取各种底层的源数据服务的数据，调用其接口时，可能出现的系统不可用的问题

从简

spring boot，微服务的非常快速，非常好用的技术框架，脱胎于spring，具体的东西就不讲解，直接带着大家上手搭建一个spring boot的框架

2个服务，缓存服务，商品服务，缓存服务依赖于商品服务

模拟各种商品服务可能接口调用时出现的各种问题，导致系统不可用的场景，然后用hystrix完整的各种技术点全部贯穿在里面

解决了一大堆设计业务背景下的系统不可用问题，hystrix整个技术体系，知识体系，也就讲解完了

消息队列，redis，咱们都不搞了

分布式系统，微服务，dubbo，不用dubbo，目前比较明显的一个趋势是，行业里，未来主要还是spring boot，spring cloud，主流的开源技术，去构建微服务的分布式系统

基于dubbo，官方很久之前就停止更新了，支持也不是太好

spring boot + http client + hystrix

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

#### 05\_基于spring boot快速构建缓存服务以及商品服务

#### 06\_快速完成缓存服务接收数据变更消息以及调用商品服务接口的代码编写

#### 07\_商品服务接口故障导致的高并发访问耗尽缓存服务资源的场景分析

#### 08\_基于hystrix的线程池隔离技术进行商品服务接口的资源隔离

资源隔离，两种策略，线程池隔离，信号量隔离

对资源隔离这一块东西，做稍微更加深入一些的讲解，告诉你，除了可以选择隔离策略以外，对你选择的隔离策略，可以做一定的细粒度的一些控制

1、execution.isolation.strategy

指定了HystrixCommand.run()的资源隔离策略，THREAD或者SEMAPHORE，一种是基于线程池，一种是信号量

线程池机制，每个command运行在一个线程中，限流是通过线程池的大小来控制的

信号量机制，command是运行在调用线程中，但是通过信号量的容量来进行限流

如何在线程池和信号量之间做选择？

默认的策略就是线程池

线程池其实最大的好处就是对于网络访问请求，如果有超时的话，可以避免调用线程阻塞住

而使用信号量的场景，通常是针对超大并发量的场景下，每个服务实例每秒都几百的QPS，那么此时你用线程池的话，线程一般不会太多，可能撑不住那么高的并发，如果要撑住，可能要耗费大量的线程资源，那么就是用信号量，来进行限流保护

一般用信号量常见于那种基于纯内存的一些业务逻辑服务，而不涉及到任何网络访问请求

netflix有100+的command运行在40+的线程池中，只有少数command是不运行在线程池中的，就是从纯内存中获取一些元数据，或者是对多个command包装起来的facacde command，是用信号量限流的

// to use thread isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.THREAD)

// to use semaphore isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)

2、command名称和command组

线程池隔离，依赖服务->接口->线程池，如何来划分

你的每个command，都可以设置一个自己的名称，同时可以设置一个自己的组

private static final Setter cachedSetter =

Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"));

public CommandHelloWorld(String name) {

super(cachedSetter);

this.name = name;

}

command group，是一个非常重要的概念，默认情况下，因为就是通过command group来定义一个线程池的，而且还会通过command group来聚合一些监控和报警信息

同一个command group中的请求，都会进入同一个线程池中

3、command线程池

threadpool key代表了一个HystrixThreadPool，用来进行统一监控，统计，缓存

默认的threadpool key就是command group名称

每个command都会跟它的threadpool key对应的thread pool绑定在一起

如果不想直接用command group，也可以手动设置thread pool name

public CommandHelloWorld(String name) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")));

this.name = name;

}

command threadpool -> command group -> command key

command key，代表了一类command，一般来说，代表了底层的依赖服务的一个接口

command group，代表了某一个底层的依赖服务，合理，一个依赖服务可能会暴露出来多个接口，每个接口就是一个command key

command group，在逻辑上去组织起来一堆command key的调用，统计信息，成功次数，timeout超时次数，失败次数，可以看到某一个服务整体的一些访问情况

command group，一般来说，推荐是根据一个服务去划分出一个线程池，command key默认都是属于同一个线程池的

比如说你以一个服务为粒度，估算出来这个服务每秒的所有接口加起来的整体QPS在100左右

你调用那个服务的当前服务，部署了10个服务实例，每个服务实例上，其实用这个command group对应这个服务，给一个线程池，量大概在10个左右，就可以了，你对整个服务的整体的访问QPS大概在每秒100左右

一般来说，command group是用来在逻辑上组合一堆command的

举个例子，对于一个服务中的某个功能模块来说，希望将这个功能模块内的所有command放在一个group中，那么在监控和报警的时候可以放一起看

command group，对应了一个服务，但是这个服务暴露出来的几个接口，访问量很不一样，差异非常之大

你可能就希望在这个服务command group内部，包含的对应多个接口的command key，做一些细粒度的资源隔离

对同一个服务的不同接口，都使用不同的线程池

command key -> command group

command key -> 自己的threadpool key

逻辑上来说，多个command key属于一个command group，在做统计的时候，会放在一起统计

每个command key有自己的线程池，每个接口有自己的线程池，去做资源隔离和限流

但是对于thread pool资源隔离来说，可能是希望能够拆分的更加一致一些，比如在一个功能模块内，对不同的请求可以使用不同的thread pool

command group一般来说，可以是对应一个服务，多个command key对应这个服务的多个接口，多个接口的调用共享同一个线程池

如果说你的command key，要用自己的线程池，可以定义自己的threadpool key，就ok了

4、coreSize

设置线程池的大小，默认是10

HystrixThreadPoolProperties.Setter()

.withCoreSize(int value)

一般来说，用这个默认的10个线程大小就够了

5、queueSizeRejectionThreshold

控制queue满后reject的threshold，因为maxQueueSize不允许热修改，因此提供这个参数可以热修改，控制队列的最大大小

HystrixCommand在提交到线程池之前，其实会先进入一个队列中，这个队列满了之后，才会reject

默认值是5

HystrixThreadPoolProperties.Setter()

.withQueueSizeRejectionThreshold(int value)

6、execution.isolation.semaphore.maxConcurrentRequests

设置使用SEMAPHORE隔离策略的时候，允许访问的最大并发量，超过这个最大并发量，请求直接被reject

这个并发量的设置，跟线程池大小的设置，应该是类似的，但是基于信号量的话，性能会好很多，而且hystrix框架本身的开销会小很多

默认值是10，设置的小一些，否则因为信号量是基于调用线程去执行command的，而且不能从timeout中抽离，因此一旦设置的太大，而且有延时发生，可能瞬间导致tomcat本身的线程资源本占满

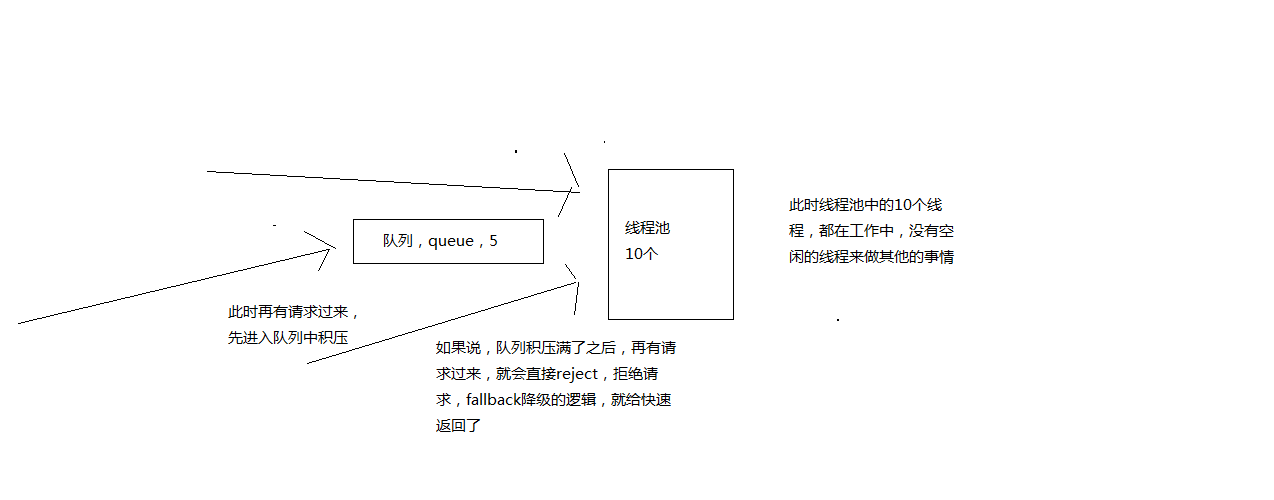
HystrixCommandProperties.Setter()

.withExecutionIsolationSemaphoreMaxConcurrentRequests(int value)

#### 09\_基于hystrix的信号量技术对地理位置获取逻辑进行资源隔离与限流

#### 10\_hystrix的线程池+服务+接口划分以及资源池的容量大小控制

线程池+queue的工作原理



资源隔离，两种策略，线程池隔离，信号量隔离

对资源隔离这一块东西，做稍微更加深入一些的讲解，告诉你，除了可以选择隔离策略以外，对你选择的隔离策略，可以做一定的细粒度的一些控制

1、execution.isolation.strategy

指定了HystrixCommand.run()的资源隔离策略，THREAD或者SEMAPHORE，一种是基于线程池，一种是信号量

线程池机制，每个command运行在一个线程中，限流是通过线程池的大小来控制的

信号量机制，command是运行在调用线程中，但是通过信号量的容量来进行限流

如何在线程池和信号量之间做选择？

默认的策略就是线程池

线程池其实最大的好处就是对于网络访问请求，如果有超时的话，可以避免调用线程阻塞住

而使用信号量的场景，通常是针对超大并发量的场景下，每个服务实例每秒都几百的QPS，那么此时你用线程池的话，线程一般不会太多，可能撑不住那么高的并发，如果要撑住，可能要耗费大量的线程资源，那么就是用信号量，来进行限流保护

一般用信号量常见于那种基于纯内存的一些业务逻辑服务，而不涉及到任何网络访问请求

netflix有100+的command运行在40+的线程池中，只有少数command是不运行在线程池中的，就是从纯内存中获取一些元数据，或者是对多个command包装起来的facacde command，是用信号量限流的

// to use thread isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.THREAD)

// to use semaphore isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)

2、command名称和command组

线程池隔离，依赖服务->接口->线程池，如何来划分

你的每个command，都可以设置一个自己的名称，同时可以设置一个自己的组

private static final Setter cachedSetter =

Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"));

public CommandHelloWorld(String name) {

super(cachedSetter);

this.name = name;

}

command group，是一个非常重要的概念，默认情况下，因为就是通过command group来定义一个线程池的，而且还会通过command group来聚合一些监控和报警信息

同一个command group中的请求，都会进入同一个线程池中

3、command线程池

threadpool key代表了一个HystrixThreadPool，用来进行统一监控，统计，缓存

默认的threadpool key就是command group名称

每个command都会跟它的threadpool key对应的thread pool绑定在一起

如果不想直接用command group，也可以手动设置thread pool name

public CommandHelloWorld(String name) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")));

this.name = name;

}

command threadpool -> command group -> command key

command key，代表了一类command，一般来说，代表了底层的依赖服务的一个接口

command group，代表了某一个底层的依赖服务，合理，一个依赖服务可能会暴露出来多个接口，每个接口就是一个command key

command group，在逻辑上去组织起来一堆command key的调用，统计信息，成功次数，timeout超时次数，失败次数，可以看到某一个服务整体的一些访问情况

command group，一般来说，推荐是根据一个服务去划分出一个线程池，command key默认都是属于同一个线程池的

比如说你以一个服务为粒度，估算出来这个服务每秒的所有接口加起来的整体QPS在100左右

你调用那个服务的当前服务，部署了10个服务实例，每个服务实例上，其实用这个command group对应这个服务，给一个线程池，量大概在10个左右，就可以了，你对整个服务的整体的访问QPS大概在每秒100左右

一般来说，command group是用来在逻辑上组合一堆command的

举个例子，对于一个服务中的某个功能模块来说，希望将这个功能模块内的所有command放在一个group中，那么在监控和报警的时候可以放一起看

command group，对应了一个服务，但是这个服务暴露出来的几个接口，访问量很不一样，差异非常之大

你可能就希望在这个服务command group内部，包含的对应多个接口的command key，做一些细粒度的资源隔离

对同一个服务的不同接口，都使用不同的线程池

command key -> command group

command key -> 自己的threadpool key

逻辑上来说，多个command key属于一个command group，在做统计的时候，会放在一起统计

每个command key有自己的线程池，每个接口有自己的线程池，去做资源隔离和限流

但是对于thread pool资源隔离来说，可能是希望能够拆分的更加一致一些，比如在一个功能模块内，对不同的请求可以使用不同的thread pool

command group一般来说，可以是对应一个服务，多个command key对应这个服务的多个接口，多个接口的调用共享同一个线程池

如果说你的command key，要用自己的线程池，可以定义自己的threadpool key，就ok了

4、coreSize

设置线程池的大小，默认是10

HystrixThreadPoolProperties.Setter()

.withCoreSize(int value)

一般来说，用这个默认的10个线程大小就够了

5、queueSizeRejectionThreshold

控制queue满后reject的threshold，因为maxQueueSize不允许热修改，因此提供这个参数可以热修改，控制队列的最大大小

HystrixCommand在提交到线程池之前，其实会先进入一个队列中，这个队列满了之后，才会reject

默认值是5

HystrixThreadPoolProperties.Setter()

.withQueueSizeRejectionThreshold(int value)

6、execution.isolation.semaphore.maxConcurrentRequests

设置使用SEMAPHORE隔离策略的时候，允许访问的最大并发量，超过这个最大并发量，请求直接被reject

这个并发量的设置，跟线程池大小的设置，应该是类似的，但是基于信号量的话，性能会好很多，而且hystrix框架本身的开销会小很多

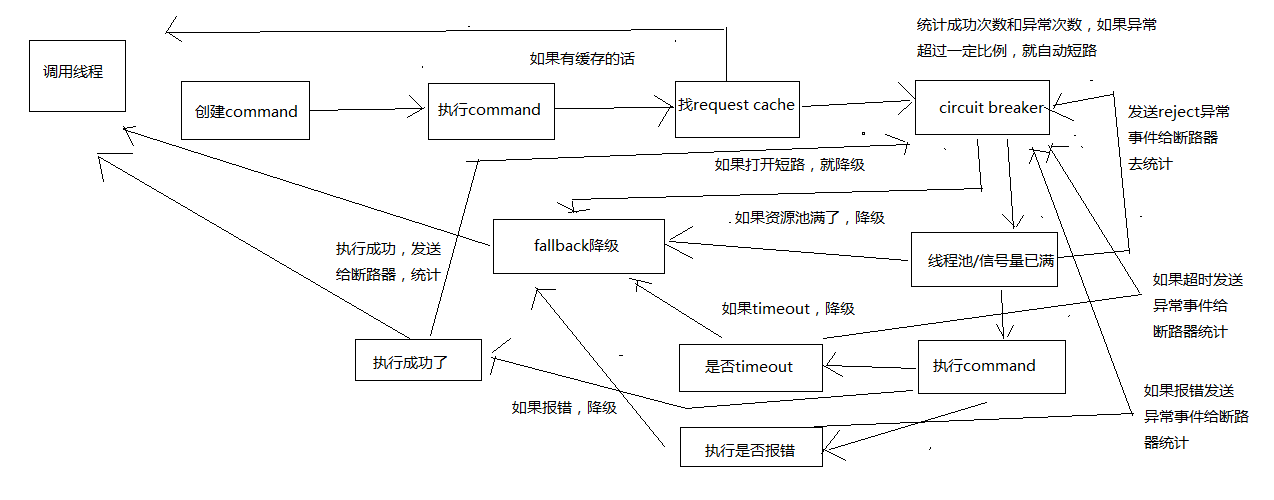
默认值是10，设置的小一些，否则因为信号量是基于调用线程去执行command的，而且不能从timeout中抽离，因此一旦设置的太大，而且有延时发生，可能瞬间导致tomcat本身的线程资源本占满

HystrixCommandProperties.Setter()

.withExecutionIsolationSemaphoreMaxConcurrentRequests(int value)

#### 11\_深入分析hystrix执行时的8大流程步骤以及内部原理

hystrix执行时的8大流程以及内部原理



之前几讲，我们用实际的业务背景给了一些可用性的问题

然后借着那些最最基础的可用性的问题，然后讲解了hystrix最基本的支持高可用的技术，资源隔离+限流

创建command，执行这个command，配置这个command对应的group和线程池，以及线程池/信号量的容量和大小

我们要去讲解一下，你开始执行这个command，调用了这个command的execute()方法以后，hystrix内部的底层的执行流程和步骤以及原理是什么呢？

在讲解这个流程的过程中，我们会带出来hystrix其他的一些核心以及重要的功能

画图分析整个8大步骤的流程，然后再对每个步骤进行细致的讲解

1、构建一个HystrixCommand或者HystrixObservableCommand

一个HystrixCommand或一个HystrixObservableCommand对象，代表了对某个依赖服务发起的一次请求或者调用

构造的时候，可以在构造函数中传入任何需要的参数

HystrixCommand主要用于仅仅会返回一个结果的调用

HystrixObservableCommand主要用于可能会返回多条结果的调用

HystrixCommand command = new HystrixCommand(arg1, arg2);

HystrixObservableCommand command = new HystrixObservableCommand(arg1, arg2);

2、调用command的执行方法

执行Command就可以发起一次对依赖服务的调用

要执行Command，需要在4个方法中选择其中的一个：execute()，queue()，observe()，toObservable()

其中execute()和queue()仅仅对HystrixCommand适用

execute()：调用后直接block住，属于同步调用，直到依赖服务返回单条结果，或者抛出异常

queue()：返回一个Future，属于异步调用，后面可以通过Future获取单条结果

observe()：订阅一个Observable对象，Observable代表的是依赖服务返回的结果，获取到一个那个代表结果的Observable对象的拷贝对象

toObservable()：返回一个Observable对象，如果我们订阅这个对象，就会执行command并且获取返回结果

K value = command.execute();

Future<K> fValue = command.queue();

Observable<K> ohValue = command.observe();

Observable<K> ocValue = command.toObservable();

execute()实际上会调用queue().get().queue()，接着会调用toObservable().toBlocking().toFuture()

也就是说，无论是哪种执行command的方式，最终都是依赖toObservable()去执行的

3、检查是否开启缓存

从这一步开始，进入我们的底层的运行原理啦，了解hysrix的一些更加高级的功能和特性

如果这个command开启了请求缓存，request cache，而且这个调用的结果在缓存中存在，那么直接从缓存中返回结果

4、检查是否开启了短路器

检查这个command对应的依赖服务是否开启了短路器

如果断路器被打开了，那么hystrix就不会执行这个command，而是直接去执行fallback降级机制

5、检查线程池/队列/semaphore是否已经满了

如果command对应的线程池/队列/semaphore已经满了，那么也不会执行command，而是直接去调用fallback降级机制

6、执行command

调用HystrixObservableCommand.construct()或HystrixCommand.run()来实际执行这个command

HystrixCommand.run()是返回一个单条结果，或者抛出一个异常

HystrixObservableCommand.construct()是返回一个Observable对象，可以获取多条结果

如果HystrixCommand.run()或HystrixObservableCommand.construct()的执行，超过了timeout时长的话，那么command所在的线程就会抛出一个TimeoutException

如果timeout了，也会去执行fallback降级机制，而且就不会管run()或construct()返回的值了

这里要注意的一点是，我们是不可能终止掉一个调用严重延迟的依赖服务的线程的，只能说给你抛出来一个TimeoutException，但是还是可能会因为严重延迟的调用线程占满整个线程池的

即使这个时候新来的流量都被限流了。。。

如果没有timeout的话，那么就会拿到一些调用依赖服务获取到的结果，然后hystrix会做一些logging记录和metric统计

7、短路健康检查

Hystrix会将每一个依赖服务的调用成功，失败，拒绝，超时，等事件，都会发送给circuit breaker断路器

短路器就会对调用成功/失败/拒绝/超时等事件的次数进行统计

短路器会根据这些统计次数来决定，是否要进行短路，如果打开了短路器，那么在一段时间内就会直接短路，然后如果在之后第一次检查发现调用成功了，就关闭断路器

8、调用fallback降级机制

在以下几种情况中，hystrix会调用fallback降级机制：run()或construct()抛出一个异常，短路器打开，线程池/队列/semaphore满了，command执行超时了

一般在降级机制中，都建议给出一些默认的返回值，比如静态的一些代码逻辑，或者从内存中的缓存中提取一些数据，尽量在这里不要再进行网络请求了

即使在降级中，一定要进行网络调用，也应该将那个调用放在一个HystrixCommand中，进行隔离

在HystrixCommand中，上线getFallback()方法，可以提供降级机制

在HystirxObservableCommand中，实现一个resumeWithFallback()方法，返回一个Observable对象，可以提供降级结果

如果fallback返回了结果，那么hystrix就会返回这个结果

对于HystrixCommand，会返回一个Observable对象，其中会发返回对应的结果

对于HystrixObservableCommand，会返回一个原始的Observable对象

如果没有实现fallback，或者是fallback抛出了异常，Hystrix会返回一个Observable，但是不会返回任何数据

不同的command执行方式，其fallback为空或者异常时的返回结果不同

对于execute()，直接抛出异常

对于queue()，返回一个Future，调用get()时抛出异常

对于observe()，返回一个Observable对象，但是调用subscribe()方法订阅它时，理解抛出调用者的onError方法

对于toObservable()，返回一个Observable对象，但是调用subscribe()方法订阅它时，理解抛出调用者的onError方法

9、不同的执行方式

execute()，获取一个Future.get()，然后拿到单个结果

queue()，返回一个Future

observer()，立即订阅Observable，然后启动8大执行步骤，返回一个拷贝的Observable，订阅时理解回调给你结果

toObservable()，返回一个原始的Observable，必须手动订阅才会去执行8大步骤

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

#### 12\_基于request cache请求缓存技术优化批量商品数据查询接口

#### 13\_开发品牌名称获取接口的基于本地缓存的fallback降级机制

#### 14\_深入理解hystrix的短路器执行原理以及模拟接口异常时的短路实验

#### 15\_深入理解线程池隔离技术的设计原则以及动手实战接口限流实验

#### 16\_基于timeout机制来为商品服务接口的调用超时提供安全保护

#### 17\_基于hystrix的高可用分布式系统架构项目实战课程的总结

已经学到的东西

hystrix的核心知识

1、hystrix内部工作原理：8大执行步骤和流程

2、资源隔离：你如果有很多个依赖服务，高可用性，先做资源隔离，任何一个依赖服务的故障不会导致你的服务的资源耗尽，不会崩溃

3、请求缓存：对于一个request context内的多个相同command，使用request cache，提升性能

4、熔断：基于短路器，采集各种异常事件，报错，超时，reject，短路，熔断，一定时间范围内就不允许访问了，直接降级，自动恢复的机制

5、降级：报错，超时，reject，熔断，降级，服务提供容错的机制

6、限流：在你的服务里面，通过线程池，或者信号量，限制对某个后端的服务或资源的访问量，避免从你的服务这里过去太多的流量，打死某个资源

7、超时：避免某个依赖服务性能过差，导致大量的线程hang住去调用那个服务，会导致你的服务本身性能也比较差

学会了这些东西以后，我们特意设置了大电商背景，商品详情页系统，缓存服务的业务场景，尽量的去结合一些仿真的业务，去学习hystrix的各项技术

这个东西做起来没那么容易，尽量做了，学习效果更好一些，兴趣也会更好一些

已经可以快速利用hystrix给自己开发的服务增加各种高可用的保障措施了，避免你的系统因为各种各样的异常情况导致崩溃，不可用

hystrix的高阶知识

1、request collapser，请求合并技术

2、fail-fast和fail-slient，高阶容错模式

3、static fallback和stubbed fallback，高阶降级模式

4、嵌套command实现的发送网络请求的降级模式

5、基于facade command的多级降级模式

6、request cache的手动清理

7、生产环境中的线程池大小以及timeout配置优化经验

8、线程池的自动化动态扩容与缩容技术

9、hystrix的metric高阶配置

10、基于hystrix dashboard的可视化分布式系统监控

11、生产环境中的hystrix工程运维经验

Jdk乐观锁cas

登录

## Hadoop

基础运营管理平台:告警规则、组件(hdfs、hbase、hive、yarn)监控、主机监控.目录增长分析、用户管理、日志管理的编写。

预警:磁盘寿命预警

NodeManager管理一个YARN集群中的每一个节点。比如监视资源使用情况（ CPU，内存，硬盘，网络）

主机监控: 以图形化方式展示内存使用、内存分配、文件系统使用率、磁盘IO耗费时间、磁盘IO读速度、磁盘IO写速度、CPU使用率、网络流量、24小时网络流量、进程数量、负载使用等指标监控。

Yarn：大数据组件运行的job的管理器

Spark：分布式的利用内存进行分布式运算的大数据组件

Hbase：基于Hadoop的大数据常用数据库 NoSQL 的Key/vale数据库 进行实时查询

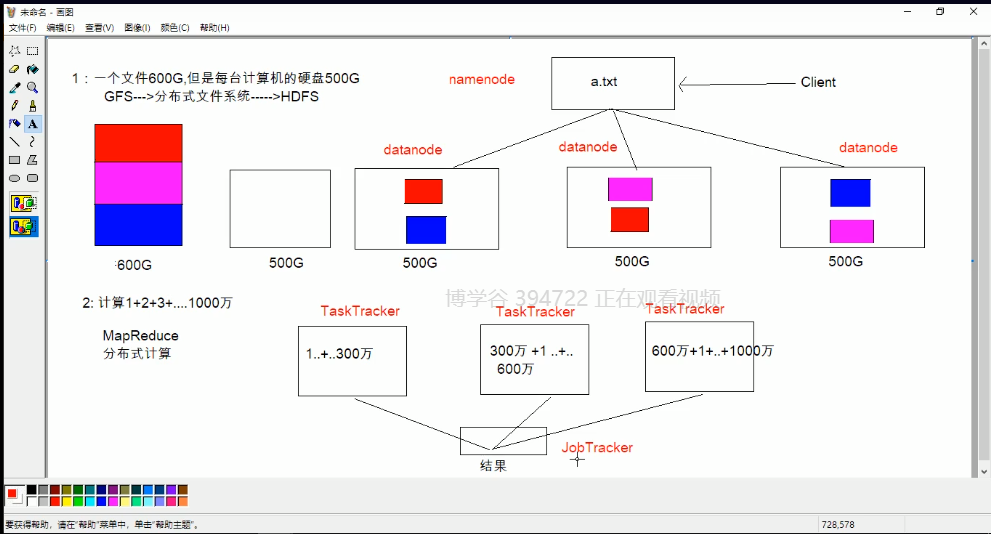
Hive：基于Hadoop的大数据数据仓库，操作和关系型数据库（MySQL）类似 进行统计查询

三大组件

hdfs MapReduce yarn资源调度

### 数据采集与展示

* 溧阳新集群每台DataNode节点批量安装ganglia工具，更新配置文件，完成主机数据自动采集、汇聚和上报，数据入MySql数据库；
* 对接Ambrari jmx接口，完成集群数据自动采集、汇聚和上报，数据入MySql数据库；
* 在大数据基础运营管理平台中完成溧阳新集群元数据配置，完成首页运营视图、监控中心、告警中心、智能预警中心、自动化运维中心、运营分析中心、资源管理中心、故障信息中心等功能数据展示。



从事java开发工作,之前做的大数据运营管理平台,springbootspringcloud微服务架构,主要做了

告警中心.组件中心.日志管理.用户管理

### 告警中心:

展示集群告警数、紧急告警数、主机告警数、组件告警数、业务告警数的变化趋势；

### 组件监控:

#### 1.hbase:RegionServer节点实时请求监控

#### 2.Hdfs监控

1. 提供历史趋势监控功能

* 以图形化方式提供Heap使用率变化趋势、RPC请求队列长度变化趋势、HDFS使用率变化趋势等指标监控；

1. 提供DataNode监控功能

* 展示DataNode节点HDFS使用情况；
* 展示DataNode节点HDFS使用历史趋势。

3.Yarn:三大组件

* ResourceManager(RM):主角色 统一入口 调度资源
* Nodemanager(NM)：负责每台机器具体的资源管理
* ApplicationMaster(AM):程序内部的老大 负责程序执行的监督管理 资源申请

内存使用率变化趋势、Heap使用率变化趋势、Applications运行情况变化趋势等指标监控。

1. 提供mr任务时长/Map数监控功能

### 主机监控:

以图形化方式展示内存使用、内存分配、文件系统使用率、磁盘IO耗费时间、磁盘IO读速度、磁盘IO写速度、CPU使用率、网络流量、24小时网络流量、进程数量、负载使用等指标监控。

1.广告模块

广告模块就是我们商城项目的首页,他是作为系统的入口,考虑到它的访问量比较大,而且数据都是由后台内容管理系统进行动态维护,如果每次访问都直接去查询数据库,数据库的压力就会特别的大,如果遇到访问高峰期,巨大的流量很可能直接将数据库打死,然后整个系统就瘫痪了.而且广告内容的时效性要求也不是很高,所以我们在广告模块里用了redis缓存技术.第一次访问请求,先去redis中查询数据,如果数据不存在,就去数据库中去查询,查询到结果存入到缓存中,存储到缓存中的数据类型是hash类型,大key是固定的,而小key则是不同广告分类的id(比如说首页轮播,猜你喜欢等等),存进去则是广告对象集合并返回给用户,下一次的请求直接去数据库中查询即可,减轻数据库的访问压力,而且从缓存中查询的速度要比从数据库中查询快,提高了用户的体验.当运营商后台修改首页的内容之后,它会先执行修改操作,然后将修改的请求发送到activeMQ,以jms queue的形式发送,广告模块收到消息后,就先去删除redis中的广告缓存,然后再去执行数据库的修改操作,这样就能初步防止缓存与数据库中数据不一致的情况

2.商家后台模块

商家后台模块我主要的负责的是品牌管理模块和商品规格管理部分,商品分类管理

首先,品牌管理主要是品牌种类的增删改查,主要为后面商品录入时选择对应的品牌做服务,商家可以在后台录入自家的经营的品牌,然后再数据库中生成品牌相应的数据,然后商家在后台就可以查看到自家的品牌,在录入商品时可以选择商品的品牌;

在商品规格管理模块的逻辑和商品品牌管理类似,唯一不同的是商品规格管理这一块数据库里有两张表,分别为规格表和规格选项表,规格选项表是对规格表的补充,规格表和规格选项表是一对多的关系

我们先设置一个规格传输类,属性分别由规格的属性,以及还有一个规格选项的集合,商家在新增规格的时候,可以点击按钮增加该规格的规格选项,所以前端传来的数据是一个规格对象和规格选项的集合,然后数据传到后台经过解析,得到一个规格传输类对象,进行数据库操作时,先创建一个规格对象对其进行赋值,然后存入数据库,然后遍历规格选项的集合,然后往数据库里面进行插入操作;查询就是根据该规格id去规格表查询对应的规格对象,然后再去规格选项表中查询规格id所对应的规格选项对象,返回一个规格选项的集合,然后将规格对象和规格选项集合设置进规格传输类对象去,返回到页面前台;删除值得注意的是,就是删除规格对象的同时,要将对应的规格选项也要一同删除

3.商品详情页

商品详情页我们是储存在Nginx服务器上的,所有的商品详情页就公用一套模板.运营商通过了商品的审核或者发生改变,就会发送消息到activeMQ,发送的消息内容,就是商品的id,详情页生成模块会有消息监听器,当监听到消息,获取到消息的内容,也就是商品id,然后通过商品id,去数据库里面查询该商品所对应的属性,比如商品信息,商品详情信息,所属的分类信息,最主要的该商品的sku信息,把查询到信息都存放到modelMap里,然后发送到模板里,模板根据对应的信息进行数据的填充,生成商品详情页,商品详情页就是商品的id.html,这样就能保证我们修改商品后新的详情页会覆盖之前的详情页;值得注意的是该商品的每个sku信息,先存在模板的java script定义的sku集合中.这样在客户选择规格选项后就不用再去后台数据库查询数据,直接可以在页面上获得数据.

4.购物车的实现

我觉得国内购物网站的购物车最人性化的就是京东了,它在用户未登录的情况下就可以使用,往里面添加商品,然后等到你点击结算的时候,才提醒你去登录,而且登录后购物车里的商品依然存在.我们购物车的实现也是类似于京东的逻辑,在用户为登录的时候,就将用户的购物车存在cookie中,设置cookie的最大存活时间为7天,用户可以往cookie里面添加商品,等到用户点击结算的后,就会跳转登录页面,用户登录后.就会把cookie中的购物车信息取出来与redis中的购物车的购物车信息进行合并,如果是同一个商家的同一个商品就进行商品数量的增加;

首先,我们项目里的购物车是每个商家对应一个购物车,购物车对象里面由商家id,商家名称,已经商家的商品的list集合

用户在商品页面点击添加到购物车时,后台先去cookie中查找购物车集合的json字符串,,经过解析成购物车list集合.然后判断用户有没有登录,如果用户已经登录,就从redis中提取用户的购物车,购物车存储在redis中是hash类型,大key是固定的,小key则是当前用户的id,存进去的是个购物车list集合.根据用户id得到购物车后,再去根据添加商品的skuID去查询商品明细SKU对象,查询到sku对象就能商家id,然后根据商家id在购物车列表中查询购物车对象,如果不存在该商家的购物车对象,就会新创建一个购物车对象,把商品添加到这个购物车对象中,然后再购物车list集合中添加该购物车对象;如果购物车list集合中已经存在该商家的购物车对象,就直接将商品添加到对应的购物车对象中,如果该商品已经存在于该购物车对象,就进行数量的增加,同时更新金额.相应的如果是商品移除,如果该商品的数量等于0,就要在购物车中移除该商品明细对象,如果购物车的商品列表等于0,就要移除该商家的购物车

5.秒杀模块

运营商后台,先设置秒杀商品信息,包括商品的id,秒杀的库存等,将这些放到redis中,因为redis性能高,读写快,能够应对高并发的访问量

首先用户先去查询当前可以秒杀的商品信息,如果商品库存大于0,可以点击秒杀按钮,参与秒杀,去redis中查询商品商品的库存,如果库存小于1,提示用户秒杀已结束,如果库存大于1,则可以继续进行秒杀,就会进行下单,减库存,生成订单,此时订单的状态显示未付款;如果用户超过100秒未付款,则秒杀失败返还库存,并且订单的状态修改为取消;如果用户在规定时间内付款成功,则修改订单状态为已付款

由于这里考虑到高并发访问请求,可能导致超卖的现象,所以这里采用了redis分布式锁,来保障线程安全问题,redis分布式锁主要应用了redis的setnx和getset两个命令,先通过setnx不成立的情况下加锁,如果客户端获得锁,则setnx返回1,如果setnx返回0,则表示该键已经被其它客户端给锁定,但是这种情况下容易发生死锁,如果客户端获得锁后,执行发生了异常,导致该锁没有被释放,外面的客户端一致无法获得锁;所以通过getset命令来解决死锁情况;所有的请求都带着相同的key也就是商品的id来获取锁,他们的value则是当前时间+超时时间的字符串,如果线程获得锁后执行成功没有发生异常,就会解锁,删除这个key,以便接下来的线程能够获得锁;如果线程执行的过程中发生了异常,则就先取出当前redis中key所对应的value和当前时间比较,判断有没有过期,如果以过期,就执行getset方法,再获得上一个锁的value,判断两次value是否相等,如果相等则获得锁对象, 如果不等则继续阻塞中.经过aparch ab压力测试,在每秒500并发请求下,没有出现线程安全

Nginx

服务器端负载均衡Nginx

 nginx是客户端所有请求统一交给nginx，由nginx进行实现负载均衡请求转发，属于服务器端负载均衡。

 既请求有nginx服务器端进行转发。

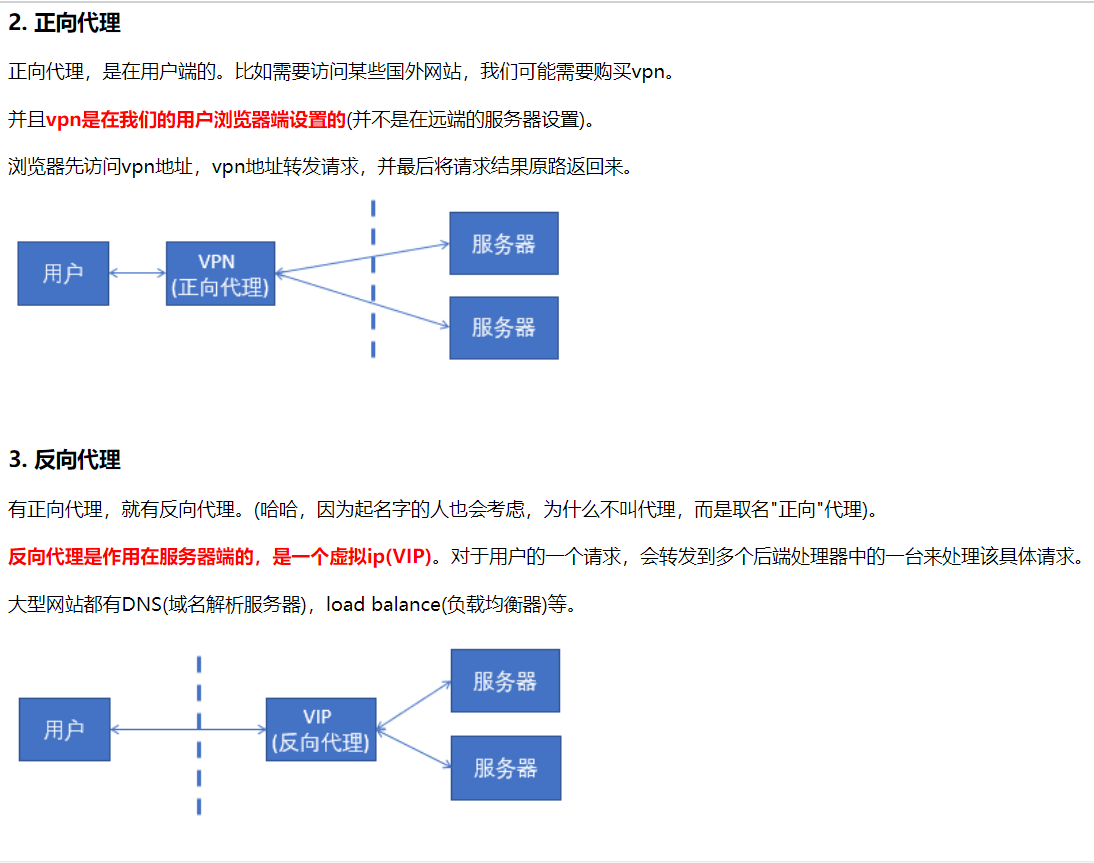
客户端负载均衡Ribbon

 Ribbon是从eureka注册中心服务器端上获取服务注册信息列表，缓存到本地，让后在本地实现轮训负载均衡策略。

 既在客户端实现负载均衡。

应用场景的区别：

Nginx适合于服务器端实现负载均衡 比如Tomcat ，Ribbon适合与在微服务中RPC远程调用实现本地服务负载均衡，比如Dubbo、SpringCloud中都是采用本地负载均衡。



Linux