尚硅谷Java技术之中频面试题

版本：V4.0

尚硅谷Java技术辅导部

目录

[第1章 面试说明 14](#_Toc61965487)

[1.1 面试过程最关键的是什么？ 14](#_Toc61965488)

[1.2 面试时该怎么说？ 14](#_Toc61965489)

[1.3 面试技巧 14](#_Toc61965490)

[1.3.1 六个常见问题 14](#_Toc61965491)

[1.3.2 两个注意事项 15](#_Toc61965492)

[1.3.3 自我介绍（控制在4分半以内，不超过5分钟） 15](#_Toc61965493)

[第2章 手写代码 15](#_Toc61965494)

[2.1 冒泡排序（Bubble Sort） 15](#_Toc61965495)

[2.2 快速排序（Quick Sort） 17](#_Toc61965496)

[2.3 归并排序（Merge Sort） 19](#_Toc61965497)

[2.4 二分查找（Binary Search） 23](#_Toc61965498)

[2.5 单例模式（Binary Search） 26](#_Toc61965499)

[2.5.1单例模式定义 26](#_Toc61965500)

[2.5.2 单例模式的特点 26](#_Toc61965501)

[2.5.3 单例的四大原则 26](#_Toc61965502)

[2.5.4 实现单例模式的方式 26](#_Toc61965503)

[第3章 Java SE 32](#_Toc61965504)

[3.1 你是怎样理解面向对象的 32](#_Toc61965505)

[3.2 int和Integer有什么区别，以及以下程序结果 32](#_Toc61965506)

[3.3 ==和Equals区别 33](#_Toc61965507)

[3.4谈谈你对反射的理解 33](#_Toc61965508)

[3.5 ArrarList和LinkedList区别 33](#_Toc61965509)

[3.6 HashMap底层源码，数据结构 34](#_Toc61965510)

[3.7 HashMap和HashTable区别 37](#_Toc61965511)

[3.8 TreeSet和HashSet区别 37](#_Toc61965512)

[3.9 String buffer和String builder区别 37](#_Toc61965513)

[3.11 什么是 java 序列化，如何实现 java 序列化？ 38](#_Toc61965514)

[3.12 Object中有哪些方法 38](#_Toc61965515)

[3.13线程有几种状态，产生的条件是什么 39](#_Toc61965516)

[3.14产生死锁的基本条件 39](#_Toc61965517)

[3.15什么是线程池，如何使用？ 40](#_Toc61965518)

[3.16 Java自带有哪几种线程池？ 40](#_Toc61965519)

[3.17 Java 中有几种类型的流 41](#_Toc61965520)

[3.18字节流如何转为字符流 42](#_Toc61965521)

[3.19请写出你最常见的5个RuntimeException 42](#_Toc61965522)

[3.20JDK和JRE有什么区别？ 43](#_Toc61965523)

[3.21两个对象的 hashCode()相同，则 equals()也一定为 true，对吗？ 43](#_Toc61965524)

[3.22final 在 java 中有什么作用？ 43](#_Toc61965525)

[3.23 java 中的 Math.round(-1.5) 等于多少？ 43](#_Toc61965526)

[3.24String 属于基础的数据类型吗？ 43](#_Toc61965527)

[3.25 java 中操作字符串都有哪些类？它们之间有什么区别？ 43](#_Toc61965528)

[3.26String str="i"与 String str=new String(“i”)一样吗？ 44](#_Toc61965529)

[3.28String 类的常用方法都有那些？ 44](#_Toc61965530)

[3.29抽象类必须要有抽象方法吗？ 44](#_Toc61965531)

[3.30普通类和抽象类有哪些区别？ 44](#_Toc61965532)

[3.31抽象类能使用 final 修饰吗？ 44](#_Toc61965533)

[3.32接口和抽象类有什么区别？ 44](#_Toc61965534)

[3.33java 中 IO 流分为几种？ 45](#_Toc61965535)

[3.34BIO、NIO、AIO 有什么区别？ 46](#_Toc61965536)

[3.35Files的常用方法都有哪些？ 47](#_Toc61965537)

[3.36java 容器都有哪些？ 47](#_Toc61965538)

[3.37Collection 和 Collections 有什么区别？ 48](#_Toc61965539)

[3.38List、Set、Map 之间的区别是什么？ 49](#_Toc61965540)

[3.39说一下 HashSet 的实现原理？ 49](#_Toc61965541)

[3.40如何决定使用 HashMap 还是 TreeMap？ 49](#_Toc61965542)

[3.41如何实现数组和 List 之间的转换？ 49](#_Toc61965543)

[3.42哪些集合类是线程安全的？ 49](#_Toc61965544)

[3.43 Iterator 怎么使用？有什么特点？ 50](#_Toc61965545)

[3.44Iterator 和 ListIterator 有什么区别？ 50](#_Toc61965546)

[3.45创建线程有哪几种方式？ 50](#_Toc61965547)

[3.46说一下 runnable 和 callable 有什么区别？ 50](#_Toc61965548)

[3.47sleep() 和 wait() 有什么区别？ 50](#_Toc61965549)

[3.48notify()和 notifyAll()有什么区别？ 51](#_Toc61965550)

[3.49线程的 run()和 start()有什么区别？ 51](#_Toc61965551)

[3.50线程池中 submit()和 execute()方法有什么区别？ 51](#_Toc61965552)

[3.51在 java 程序中怎么保证多线程的运行安全？ 51](#_Toc61965553)

[3.52什么是死锁？ 52](#_Toc61965554)

[3.53怎么防止死锁？ 52](#_Toc61965555)

[3.54 synchronized 和 volatile 的区别是什么？ 54](#_Toc61965556)

[3.55synchronized 和 Lock 有什么区别？ 54](#_Toc61965557)

[3.56 synchronized 和 ReentrantLock 区别是什么？ 55](#_Toc61965558)

[3.57什么是反射？ 55](#_Toc61965559)

[3.58动态代理是什么？有哪些应用？ 55](#_Toc61965560)

[3.59深拷贝和浅拷贝区别是什么？ 55](#_Toc61965561)

[3.60throw 和 throws 的区别？ 55](#_Toc61965562)

[3.61final、finally、finalize 有什么区别？ 56](#_Toc61965563)

[3.64 Java中CyclicBarrier 和 CountDownLatch有什么不同？ 56](#_Toc61965564)

[3.65为什么说 Synchronized 是一个悲观锁?乐观锁的实现原理 又是什么?什么是 CAS，它有什么特性? 57](#_Toc61965565)

[3.66 Java中Semaphore是什么？ 57](#_Toc61965566)

[3.67现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？ 57](#_Toc61965567)

[3.68你如何在Java中获取线程堆栈？ 58](#_Toc61965568)

[3.69提交任务时线程池队列已满会时发会生什么？ 58](#_Toc61965569)

[3.70什么是乐观锁和悲观锁？ 59](#_Toc61965570)

[第4章 JVM 60](#_Toc61965571)

[4.1 JVM内存分哪几个区，每个区的作用是什么? 60](#_Toc61965572)

[4.2 heap 和stack 有什么区别 61](#_Toc61965573)

[4.3 java类加载过程? 62](#_Toc61965574)

[4.4 什么是类加载器，类加载器有哪些? 63](#_Toc61965575)

[4.5 java中垃圾收集的方法有哪些? 63](#_Toc61965576)

[4.6 如何判断一个对象是否存活?(或者GC对象的判定方法) 63](#_Toc61965577)

[4.7 简述java内存分配与回收策略以及Minor GC和Major GC（full GC） 64](#_Toc61965578)

[4.8什么情况下会发生栈内存溢出。 64](#_Toc61965579)

[4.9 JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代 65](#_Toc61965580)

[4.10你知道哪几种垃圾收集器，各自的优缺点，重点讲下cms和G1，包括原理，流程，优缺点。 65](#_Toc61965581)

[4.11 JVM内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存。 65](#_Toc61965582)

[4.12简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。 66](#_Toc61965583)

[4.13说说你知道的几种主要的JVM参数 67](#_Toc61965584)

[4.15垃圾收集算法 68](#_Toc61965585)

[4.16调优工具用过哪些 69](#_Toc61965586)

[4.17你知道哪些JVM性能调优 69](#_Toc61965587)

[4.18 Eden和Survivor的比例分配等 70](#_Toc61965588)

[4.19说一说你对环境变量classpath的理解？如果一个类不在classpath下，为什么会抛出ClassNotFoundException异常，如果在不改变这个类路径的前期下，怎样才能正确加载这个类？ 70](#_Toc61965589)

[第5章 设计模式 70](#_Toc61965590)

[5.1你所知道的设计模式有哪些 70](#_Toc61965591)

[5.2单例设计模式 70](#_Toc61965592)

[5.3工厂设计模式（Factory） 71](#_Toc61965593)

[5.3.1什么是工厂设计模式？ 71](#_Toc61965594)

[5.3.2简单工厂（Simple Factory） 71](#_Toc61965595)

[5.3.3工厂方法（Factory Method） 72](#_Toc61965596)

[5.3.4抽象工厂（Abstract Factory） 73](#_Toc61965597)

[5.3.5三种工厂方式总结 76](#_Toc61965598)

[5.4代理模式（Proxy） 76](#_Toc61965599)

[5.4.1什么是代理模式？ 76](#_Toc61965600)

[5.4.2为什么要用代理模式？ 77](#_Toc61965601)

[5.4.3有哪几种代理模式？ 77](#_Toc61965602)

[5.4.4静态代理（Static Proxy） 77](#_Toc61965603)

[5.4.5 JDK动态代理（Dynamic Proxy） 79](#_Toc61965604)

[5.4.6 CGLib动态代理（CGLib Proxy） 80](#_Toc61965605)

[5.4.7 简述动态代理的原理， 常用的动态代理的实现方式 81](#_Toc61965606)

[第6章 MySql 82](#_Toc61965607)

[6.1 jdbc 操作数据库流程 82](#_Toc61965608)

[6.2关系数据库中连接池的机制是什么？ 82](#_Toc61965609)

[6.3 SQL 的select 语句完整的执行顺序 82](#_Toc61965610)

[6.3 MySQL的事务 83](#_Toc61965611)

[6.4行锁，表锁 83](#_Toc61965612)

[6.5索引 83](#_Toc61965613)

[6.6 b-tree和b+tree的区别 84](#_Toc61965614)

[6.7简述在MySQL数据库中MyISAM和InnoDB的区别 85](#_Toc61965615)

[6.8你们公司有哪些数据库设计规范 85](#_Toc61965616)

[6.9 MySQL性能优化 87](#_Toc61965617)

[6.10 SQL 语句优化案例 87](#_Toc61965618)

[6.11 常见面试sql 88](#_Toc61965619)

[6.12联合索引是什么?为什么需要注意联合索引中的顺序? 91](#_Toc61965620)

[6.13什么是聚簇索引和非聚簇索引 91](#_Toc61965621)

[6.14 delete、drop、truncate区别 91](#_Toc61965622)

[6.15 mysql主从复制 91](#_Toc61965623)

[6.16 MySQL中的varchar和char的区别以及varchar（50）中的50代表的涵义 92](#_Toc61965624)

[6.17一张表里面有ID自增主键，当insert了17条记录之后，删除了第15,16,17条记录，再把mysql重启，再insert一条记录，这条记录的ID是18还是15 ？ 92](#_Toc61965625)

[6.18为什么使用数据索引能提高效率 92](#_Toc61965626)

[6.19 B+树索引和哈希索引的区别 93](#_Toc61965627)

[6.20什么情况下应不建或少建索引 93](#_Toc61965628)

[6.21行级锁定的优点缺点 93](#_Toc61965629)

[6.22创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因? 94](#_Toc61965630)

[6.23 varchar(10)和int(10)代表什么含义? 94](#_Toc61965631)

[6.24关心过业务系统里面的sql耗时吗?统计过慢查询吗?对慢查询都怎么优化过? 94](#_Toc61965632)

[6.25什么是存储过程？有哪些优缺点？ 94](#_Toc61965633)

[6.26字段为什么要定义为NOT NULL? 95](#_Toc61965634)

[6.27货币字段用什么类型？ 95](#_Toc61965635)

[6.28时间字段用什么类型? 95](#_Toc61965636)

[6.29为什么不直接存储图片、音频、视频等大容量内容? 95](#_Toc61965637)

[6.30为什么一定要设一个主键？ 96](#_Toc61965638)

[6.31你们主键是用自增还是UUID? 96](#_Toc61965639)

[6.32自增主键用完了怎么办? 96](#_Toc61965640)

[6.33主键为什么不推荐有业务含义? 96](#_Toc61965641)

[第7章Java Web 96](#_Toc61965642)

[7.1 http 的长连接和短连接 96](#_Toc61965643)

[7.2 http 常见的状态码有哪些？ 96](#_Toc61965644)

[7.3 GET 和POST 的区别？ 97](#_Toc61965645)

[7.4 Cookie 和Session 的区别 97](#_Toc61965646)

[7.5在单点登录中，如果 cookie 被禁用了怎么办？ 97](#_Toc61965647)

[7.6什么是jsp，什么是Servlet？jsp 和Servlet 有什么区别？ 98](#_Toc61965648)

[7.7 servlet生命周期 98](#_Toc61965649)

[7.8 servlet特性 99](#_Toc61965650)

[7.9 servlet是单实例的吗？ 99](#_Toc61965651)

[7.10 servlet是线程安全的吗？为什么？ 99](#_Toc61965652)

[7.11如何解决Servlet线程不安全的问题？ 99](#_Toc61965653)

[7.12谈谈过滤器的作用 100](#_Toc61965654)

[7.13谈谈拦截器的作用 100](#_Toc61965655)

[7.14拦截器和过滤器有什么区别 100](#_Toc61965656)

[7.15拦截器和过滤器的执行顺序 100](#_Toc61965657)

[7.16说一下 jsp 的 4 种作用域？ 100](#_Toc61965658)

[7.17 jsp 有哪些内置对象？作用分别是什么？ 102](#_Toc61965659)

[7.18转发（Forward）和重定向（Redirect）的区别？ 102](#_Toc61965660)

[7.19Request对象的主要方法有哪些？ 102](#_Toc61965661)

[7.20request.getAttribute()和 request.getParameter()有何区别？ 103](#_Toc61965662)

[7.21JSP中动态include和静态include的区别？ 103](#_Toc61965663)

[7.22JSP乱码如何解决？ 104](#_Toc61965664)

[7.23什么是Tomcat？ 104](#_Toc61965665)

[7.24详细描述MVC 104](#_Toc61965666)

[7.25Http请求由哪三部分组成? 104](#_Toc61965667)

[7.26如何实现跨域？ 105](#_Toc61965668)

[7.27简述 tcp 和 udp的区别？ 105](#_Toc61965669)

[7.28tcp 为什么要三次握手，两次不行吗？为什么？ 105](#_Toc61965670)

[第8章 SSM框架 106](#_Toc61965671)

[8.1请写出 spring 中常用的依赖注入方式。 106](#_Toc61965672)

[8.2简述Spring中IOC容器常用的接口和具体的实现类 106](#_Toc61965673)

[8.3简述Spring中如何基于注解配置Bean和装配Bean 106](#_Toc61965674)

[8.4说出Spring 或者 Springmvc中常用的5个注解，并解释含义 106](#_Toc61965675)

[8.5请解释Spring Bean的生命周期？ 107](#_Toc61965676)

[8.6简单的谈一下SpringMVC的工作流程？ 107](#_Toc61965677)

[8.7 SpringMVC中如何解决POST请求中文乱码问题 107](#_Toc61965678)

[8.8简述SpringMvc里面拦截器是如何定义，如何配置，拦截器中三个重要的方法 108](#_Toc61965679)

[8.9 MyBatis中 #{}和${}的区别是什么？ 108](#_Toc61965680)

[8.10 Mybatis 结果集的映射方式有几种，并分别解释每种映射方式如何使用。 109](#_Toc61965681)

[8.11简述MyBatis的单个参数、多个参数如何传递及如何取值。 109](#_Toc61965682)

[8.12 MyBatis如何获取自动生成的(主)键值? 109](#_Toc61965683)

[8.13简述Mybatis的动态SQL，列出常用的6个标签及作用 109](#_Toc61965684)

[8.14 Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？ 109](#_Toc61965685)

[8.15 Mybatis 如何完成MySQL的批量操作,举例说明 110](#_Toc61965686)

[8.17简述Spring 中bean的作用域 110](#_Toc61965687)

[8.18简述Spring中自动装配常用的两种装配模式 110](#_Toc61965688)

[8.19请解释@Autowired注解的工作机制及required属性的作用 110](#_Toc61965689)

[8.20 简述Springmvc中ContextLoaderListener的作用以及实现原理 110](#_Toc61965690)

[8.21简述Mybatis提供的两级缓存，以及缓存的查找顺序 111](#_Toc61965691)

[8.22 简述Spring与Springmvc整合时，如何解决bean被创建两次的问题 111](#_Toc61965692)

[8.23简述Spring与Mybatis整合时，主要整合的两个地方 111](#_Toc61965693)

[8.24简述Spring声明式事务中@Transaction中常用的两种事务传播行为 111](#_Toc61965694)

[8.25简述@RequestMapping注解的作用，可标注的位置，常用的属性 111](#_Toc61965695)

[8.26简述Springmvc中处理模型数据的两种方式 112](#_Toc61965696)

[8.27简述REST中的四种请求方式及对应的操作 112](#_Toc61965697)

[8.28简述视图和视图解析的关系及作用 112](#_Toc61965698)

[8.29说出三个 常用的视图类 112](#_Toc61965699)

[8.30简述REST中HiddenHttpMethodFilter过滤器的作用 112](#_Toc61965700)

[8.31简述Springmvc中如何返回JSON数据 112](#_Toc61965701)

[8.32简述如何在myBatis中的增删改操作获取到对数据库的影响条数 112](#_Toc61965702)

[8.33 Springmvc中的控制器的注解用哪个，可以是否用别的注解代替 113](#_Toc61965703)

[8.34如何在Springmvc中获取客户端提交的请求参数 113](#_Toc61965704)

[8.35简述Springmvc中InternalResourceViewResolver解析器的工作机制 113](#_Toc61965705)

[8.36 Springmvc中如何完成重定向 113](#_Toc61965706)

[8.37简述Spring中切面中常用的几种通知，并简单解释 113](#_Toc61965707)

[8.38解释MyBatis中 @Param注解的作用 113](#_Toc61965708)

[8.39简述Mybatis中使用Mapper接口开发，如何完成Mapper接口与SQL映射文件、方法与SQL语句的绑定 113](#_Toc61965709)

[8.40 SpringMVC的工作原理 113](#_Toc61965710)

[8.41谈谈你对Spring 的理解 114](#_Toc61965711)

[8.42 Spring中常用的设计模式 114](#_Toc61965712)

[8.43请描述一下Spring的事务管理 115](#_Toc61965713)

[8.45说出SpringMVC常用的5个注解？如何使用 SpringMVC完成JSON操作？ 115](#_Toc61965714)

[8.46Spring 支持的事务管理类型有哪些？你在项目中使用哪种方式？怎么理解全局事务和局部事务？ 115](#_Toc61965715)

[8.47JDBC编程有哪些不足之处，MyBatis是如何解决这些问题的？ 116](#_Toc61965716)

[8.48SqlMapConfig.xml中配置有哪些内容？ 116](#_Toc61965717)

[8.49简单的说一下MyBatis的一级缓存和二级缓存？ 116](#_Toc61965718)

[8.50Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？ 117](#_Toc61965719)

[第9章 SpringBoot 117](#_Toc61965720)

[9.1什么是 Spring Boot？ 117](#_Toc61965721)

[9.2为什么要用 Spring Boot？ 117](#_Toc61965722)

[9.3 Spring Boot有哪些缺点？ 118](#_Toc61965723)

[9.4 Spring Boot 的核心配置文件有哪几个？它们的区别是什么？ 118](#_Toc61965724)

[9.5 Spring Boot 的配置文件有哪几种格式？它们有什么区别？ 118](#_Toc61965725)

[9.6 Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？ 118](#_Toc61965726)

[9.7开启 Spring Boot 特性有哪几种方式？ 118](#_Toc61965727)

[9.8 Spring Boot 需要独立的容器运行吗？ 119](#_Toc61965728)

[9.9运行 Spring Boot 有哪几种方式？ 119](#_Toc61965729)

[9.10 Spring Boot 自动配置原理是什么？ 120](#_Toc61965730)

[9.12、如何在 Spring Boot 启动的时候运行一些特定的代码？ 121](#_Toc61965731)

[9.13、Spring Boot 有哪几种读取配置的方式？ 121](#_Toc61965732)

[9.14、Spring Boot 支持哪些日志框架？推荐和默认的日志框架是哪个？ 121](#_Toc61965733)

[9.15、SpringBoot 实现热部署有哪几种方式？ 121](#_Toc61965734)

[9.16、你如何理解 Spring Boot 配置加载顺序？ 122](#_Toc61965735)

[9.17、Spring Boot 如何定义多套不同环境配置？ 123](#_Toc61965736)

[9.18 Spring Boot 可以兼容老 Spring 项目吗，如何做？ 123](#_Toc61965737)

[9.19 Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？ 123](#_Toc61965738)

[9.20Spring Boot、Spring MVC 和 Spring 有什么区别？ 123](#_Toc61965739)

[9.21什么是 Spring Boot Stater ？ 124](#_Toc61965740)

[9.22什么是 Spring Data REST? 124](#_Toc61965741)

[9.23RequestMapping 和 GetMapping 的不同之处在哪里？ 124](#_Toc61965742)

[第10章 SpringCloud 124](#_Toc61965743)

[10.1 Spring Boot和Spring 是什么关系 124](#_Toc61965744)

[10.2Spring Boot和Spring Cloud是什么关系 124](#_Toc61965745)

[10.3 Eureka和zookeeper的区别 125](#_Toc61965746)

[10.4springcloud如何实现服务的注册? 125](#_Toc61965747)

[10.5eureka自我保护机制是什么? 125](#_Toc61965748)

[10.6什么是服务熔断？什么是服务降级? 126](#_Toc61965749)

[10.7Ribbon和Feign的区别？ 126](#_Toc61965750)

[10.8Hystrix断路器能干嘛？ 126](#_Toc61965751)

[10.9分布式配置中心能干嘛？ 126](#_Toc61965752)

[10.10微服务之间是如何通讯的？ 127](#_Toc61965753)

[10.11什么是 Swagger？你用 Spring Boot 实现了它吗？ 127](#_Toc61965754)

[10.11你所知道的微服务技术栈有哪些？请列举一二 127](#_Toc61965755)

[10.12Ribbon负载均衡能干嘛？ 127](#_Toc61965756)

[10.13什么是SpringCloud Config分布式配置中心 128](#_Toc61965757)

[第11章 Redis 128](#_Toc61965758)

[11.1什么是Redis？ 128](#_Toc61965759)

[11.2 redis有哪些数据类型 128](#_Toc61965760)

[11.3一个字符串类型的值能存储最大容量是多少？ 128](#_Toc61965761)

[11.4 怎么理解Redis事务？ 128](#_Toc61965762)

[11.5 Redis事务相关的命令有哪几个？ 129](#_Toc61965763)

[11.6 Redis key的过期时间和永久有效分别怎么设置？ 129](#_Toc61965764)

[11.7 Redis 集群如何选择数据库?  129](#_Toc61965765)

[11.8 Redis持久化数据和缓存怎么做扩容？ 129](#_Toc61965766)

[11.9 为什么Redis需要把所有数据放到内存中？ 129](#_Toc61965767)

[11.10 Redis如何做内存优化？ 129](#_Toc61965768)

[11.11缓存穿透 129](#_Toc61965769)

[11.12 哨兵模式 130](#_Toc61965770)

[11.13 悲观锁 131](#_Toc61965771)

[11.14 乐观锁 131](#_Toc61965772)

[11.15 持久化 131](#_Toc61965773)

[11.16 Redis提供了哪几种持久化方式？ 132](#_Toc61965774)

[11.17如何选择合适的持久化方式？ 132](#_Toc61965775)

[11.18分布式Redis是前期做还是后期规模上来了再做好？为什么？ 133](#_Toc61965776)

[11.19 Redis的内存占用情况怎么样？ 133](#_Toc61965777)

[11.20都有哪些办法可以降低Redis的内存使用情况呢？ 133](#_Toc61965778)

[11.21查看Redis使用情况及状态信息用什么命令？ 133](#_Toc61965779)

[11.22 Redis的内存用完了会发生什么？ 133](#_Toc61965780)

[11.23 redis是单线程的，为什么那么快 133](#_Toc61965781)

[11.24 Redis是单线程的，如何提高多核CPU的利用率？ 134](#_Toc61965782)

[11.25一个Redis实例最多能存放多少的keys？List、Set、Sorted Set他们最多能存放多少元素？ 134](#_Toc61965783)

[11.26 Redis常见性能问题和解决方案？ 134](#_Toc61965784)

[11.27 Redis相比memcached有哪些优势？ 134](#_Toc61965785)

[11.28 MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？ 135](#_Toc61965786)

[11.29 Redis有哪几种数据淘汰策略？ 135](#_Toc61965787)

[11.30 Redis集群方案应该怎么做？都有哪些方案？ 135](#_Toc61965788)

[11.31说说Redis哈希槽的概念？ 135](#_Toc61965789)

[11.32 Redis集群最大节点个数是多少？ 135](#_Toc61965790)

[11.33 怎么测试Redis的连通性？ 136](#_Toc61965791)

[11.34修改配置不重启Redis会实时生效吗？ 136](#_Toc61965792)

[11.35 Redis有哪些适合的场景？ 136](#_Toc61965793)

[第12章 MQ 137](#_Toc61965794)

[12.1 ActiveMQ 如果消息发送失败怎么办？ 137](#_Toc61965795)

[12.2如何使用ActiveMQ 解决分布式事务？ 137](#_Toc61965796)

[12.3 如何防止ActiveMQ消息重复发送？ 137](#_Toc61965797)

[12.4使用消息队列有什么缺点？ 137](#_Toc61965798)

[12.5消息队列的作用与使用场景？ 138](#_Toc61965799)

[12.6如何保证消息的顺序性？ 141](#_Toc61965800)

[12.7如何保证消息队列的高可用？ 143](#_Toc61965801)

[12.8无法被路由的消息，去了哪里？ 144](#_Toc61965802)

[12.9消息队列过期失效问题 144](#_Toc61965803)

[12.10消息队列满了怎么办？ 145](#_Toc61965804)

[12.11 RabbitMQ 有哪些重要的组件？ 145](#_Toc61965805)

[12.12 RabbitMQ 有几种广播类型？ 145](#_Toc61965806)

[12.14消息如何分发？ 145](#_Toc61965807)

[12.15如何确保消息正确地发送至 RabbitMQ？ 如何确保消息接收方消费了消息？ 146](#_Toc61965808)

[12.16为什么不应该对所有的 message 都使用持久化机制？ 146](#_Toc61965809)

[12.16ActiveMQ和RabbitMQ的区别？ 146](#_Toc61965810)

[12.17 RabbitMQ 中的 broker 是指什么？cluster 又是指什么？ 147](#_Toc61965811)

[12.18什么是元数据？元数据分为哪些类型？包括哪些内容？与 cluster 相关的元数据有哪些？元数据是如何保存的？元数据在 cluster 中是如何分布的？ 147](#_Toc61965812)

[12.19 RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？ 147](#_Toc61965813)

[12.20 RabbitMQ 概念里的 channel、exchange 和 queue 这些东西是逻辑概念，还是对应着进程实体？这些东西分别起什么作用？ 147](#_Toc61965814)

[12.21消息在什么时候会变成Dead Letter（死信）？ 148](#_Toc61965815)

[12.22 RabbitMQ如何实现延时队列？ 148](#_Toc61965816)

[12.23 SpringBoot中，bean还没有初始化好，消费者就开始监听取消息，导致空指针异常，怎么让消费者在容器启动完毕后才开始监听？ 148](#_Toc61965817)

[第13章 ElasticSearch 148](#_Toc61965818)

[13.1什么是ElasticSearch？ 148](#_Toc61965819)

[13.2 Elasticsearch中的倒排索引是什么？ 148](#_Toc61965820)

[13.3 elasticsearch 了解多少，说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段。 149](#_Toc61965821)

[13.4 elasticsearch 索引数据多了怎么办，如何调优，部署 149](#_Toc61965822)

[13.5、elasticsearch 是如何实现 master 选举的 150](#_Toc61965823)

[13.6详细描述一下 Elasticsearch 索引文档的过程 150](#_Toc61965824)

[13.7详细描述一下 Elasticsearch 搜索的过程？ 151](#_Toc61965825)

[13.8 Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法 151](#_Toc61965826)

[13.9 Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办？ 151](#_Toc61965827)

[13.10对于 GC 方面，在使用 Elasticsearch 时要注意什么？ 151](#_Toc61965828)

[13.11在并发情况下，Elasticsearch 如果保证读写一致？ 151](#_Toc61965829)

[第 14章 Dubbo 152](#_Toc61965830)

[14.1 Dubbo 的连接方式有哪些？ 152](#_Toc61965831)

[14.2 Dubbo 的容错机制有哪些？ 152](#_Toc61965832)

[14.3使用dubbo 遇到过哪些问题？ 152](#_Toc61965833)

[14.4为什么要用Dubbo？ 153](#_Toc61965834)

[14.5 Dubbo 和 Spring Cloud 有什么区别？ 153](#_Toc61965835)

[14.6 dubbo都支持什么协议，推荐用哪种？ 154](#_Toc61965836)

[14.7 Dubbo需要 Web 容器吗？ 154](#_Toc61965837)

[14.8 Dubbo里面有哪几种节点角色？ 155](#_Toc61965838)

[14.9画一画服务注册与发现的流程图 155](#_Toc61965839)

[14.10 Dubbo默认使用什么注册中心，还有别的选择吗？ 156](#_Toc61965840)

[14.11 Dubbo有哪几种配置方式？ 156](#_Toc61965841)

[14.12 Dubbo 核心的配置有哪些？ 156](#_Toc61965842)

[14.13在 Provider 上可以配置的 Consumer 端的属性有哪些？ 156](#_Toc61965843)

[14.14 Dubbo启动时如果依赖的服务不可用会怎样？ 156](#_Toc61965844)

[14.15 Dubbo推荐使用什么序列化框架，你知道的还有哪些？ 157](#_Toc61965845)

[14.16 Dubbo有哪几种负载均衡策略，默认是哪种？ 157](#_Toc61965846)

[14.17注册了多个同一样的服务，如果测试指定的某一个服务呢？ 157](#_Toc61965847)

[14.18 Dubbo支持服务多协议吗？ 157](#_Toc61965848)

[14.19服务上线怎么兼容旧版本？ 157](#_Toc61965849)

[14.20 Dubbo可以对结果进行缓存吗？ 157](#_Toc61965850)

[14.21Dubbo支持服务降级吗？ 157](#_Toc61965851)

[14.22 Dubbo的架构设计 158](#_Toc61965852)

[第15章 Zookeeper 159](#_Toc61965853)

[15.1 ZooKeeper是什么？ 159](#_Toc61965854)

[15.2 Zookeeper Watcher 机制 -- 数据变更通知 159](#_Toc61965855)

[15.3四种类型的数据节点 Znode 160](#_Toc61965856)

[15.4说一下 zk的应用场景 160](#_Toc61965857)

[15.5 zookeeper负载均衡和nginx负载均衡区别 161](#_Toc61965858)

[15.6Zookeeper对节点的watch监听通知是永久的吗？ 162](#_Toc61965859)

[15.7集群支持动态添加机器吗？ 162](#_Toc61965860)

[15.8说一说选举算法及流程 162](#_Toc61965861)

[15.9有哪几种部署模式？ 163](#_Toc61965862)

[15.10集群中的机器角色都有哪些？ 163](#_Toc61965863)

[15.11集群最少要几台机器，集群规则是怎样的 163](#_Toc61965864)

[15.12集群如果有3台机器，挂掉一台集群还能工作吗？挂掉两台呢？ 163](#_Toc61965865)

[15.13说几个zookeeper常用的命令。 163](#_Toc61965866)

[15.14Zookeeper分布式锁 164](#_Toc61965867)

[15.15Zookeeper数据复制 164](#_Toc61965868)

[15.16zookeeper是如何保证事务的顺序一致性的？ 164](#_Toc61965869)

[第16章Git 164](#_Toc61965870)

[16.1 reset 与 rebase, pull 与 fetch 的区别 164](#_Toc61965871)

[16.2 git merge和git rebase的区别 165](#_Toc61965872)

[16.3 git如何解决代码冲突 165](#_Toc61965873)

[第17章Linux 165](#_Toc61965874)

[17.1 Linux常用命令 165](#_Toc61965875)

[17.2如何查看所有java进程 165](#_Toc61965876)

[17.3如何杀掉某个服务的进程 166](#_Toc61965877)

[17.4启动/停止服务 166](#_Toc61965878)

[17.5如何查看测试项目的日志 166](#_Toc61965879)

[17.6如何查看最近1000行日志 166](#_Toc61965880)

[17.7 LINUX中如何查看某个端口是否被占用 166](#_Toc61965881)

[17.7查看当前所有已经使用的端口情况 167](#_Toc61965882)

[17.8如何查找一个文件大小超过5M的文件 168](#_Toc61965883)

[17.9如果知道一个文件名称，怎么查这个文件在linux下的哪个目录 168](#_Toc61965884)

[17.10 find查找文件 168](#_Toc61965885)

[17.11 vim（vi）编辑器 168](#_Toc61965886)

[第18章Nginx 169](#_Toc61965887)

[18.1 请解释一下什么是 Nginx? 169](#_Toc61965888)

[18.2 请列举 Nginx 的一些特性。 169](#_Toc61965889)

[18.3 请解释 Nginx 如何处理 HTTP 请求。 169](#_Toc61965890)

[18.4 在 Nginx 中，如何使用未定义的服务器名称来阻止处理请求? 169](#_Toc61965891)

[18.5 使用“反向代理服务器”的优点是什么? 169](#_Toc61965892)

[18.6 请列举 Nginx 服务器的最佳用途。 169](#_Toc61965893)

[18.7 请解释是否有可能将 Nginx 的错误替换为 502 错误、503? 169](#_Toc61965894)

[18.8 请解释 ngx\_http\_upstream\_module 的作用是什么? 170](#_Toc61965895)

[18.9 请陈述 stub\_status 和 sub\_filter 指令的作用是什么? 170](#_Toc61965896)

# 第1章 面试说明

## 1.1 面试过程最关键的是什么？

1）不是你说了什么，而是你怎么说

2）大大方方的聊，放松

## 1.2 面试时该怎么说？

1）语言表达清楚

（1）思维逻辑清晰，表达流畅

（2）一二三层次表达

2）所述内容不犯错

（1）不说前东家或者自己的坏话

（2）说自己擅长的方面

（3）实质：内容听过，自我肯定；没听过，学习过程。

## 1.3 面试技巧

### 1.3.1 六个常见问题

1）你的优点是什么？

大胆的说出自己各个方面的优势和特长

2）你的缺点是什么？

不要谈为自己减分的问题；用“缺点”衬托自己的优点

3）你的离职原因是什么？

* + 不说前东家坏话，哪怕被伤过
  + 合情合理合法
  + 不要说超过1个以上的原因

4）您对薪资的期望是多少？

* + 非终面不深谈薪资
  + 只说区间，不说具体数字
  + 底线是不低于当前薪资
  + 非要具体数字，区间取中间值，或者当前薪资的+20%

5）您还有什么想问的问题？

* + 这是体现个人眼界和层次的问题
  + 问题本身不在于面试官想得到什么样的答案，而在于你跟别的应聘者的对比
  + 标准答案：

公司希望我入职后的3-6个月内，给公司解决什么样的问题

公司（或者对这个部门）未来的战略规划是什么样子的？

以你现在对我的了解，您觉得我需要多长时间融入公司？

6）您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前

### 1.3.2 两个注意事项

1）职业化的语言

2）职业化的形象

### 1.3.3 自我介绍（控制在4分半以内，不超过5分钟）

1）个人基本信息

2）工作履历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因

3）深度沟通（也叫压力面试）

刨根问底下沉式追问（注意是下沉式，而不是发散式的）

基本技巧：往自己熟悉的方向说

# 手写代码

## 2.1 冒泡排序（Bubble Sort）

**算法描述：**

* 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
* 针对所有的元素重复以上的步骤，除了最后一个；
* 重复步骤1~3，直到排序完成。



如果两个元素相等，不会再交换位置，所以冒泡排序是一种稳定排序算法。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* 冒泡排序
6. \*/
7. **public** **class** BubbleSort {
9. /\*\*
10. \* @param data 被排序的数组
11. \*/
12. **public** **static** **void** bubbleSort(**int**[] data) {
14. **int** arrayLength = data.length;
16. **for** (**int** i = 1; i < arrayLength; i++) {//第i次排序
18. **for** (**int** j = 0; j < arrayLength - i; j++) {//从索引为j的数开始
19. **if** (data[j] > data[j + 1]) { //相邻元素两两对比
20. **int** temp = data[j + 1];  // 元素交换
21. data[j + 1] = data[j];
22. data[j] = temp;
23. }
24. }
26. System.out.println("第" + i + "次排序：\n" + java.util.Arrays.toString(data));
27. }
28. }
30. **public** **static** **void** main(String[] args) {
32. **int**[] data = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
34. System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
36. bubbleSort(data);
38. System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
39. }
40. }

## 2.2 快速排序（Quick Sort）

**算法描述：**

使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

* 从数列中挑出一个元素，称为 “基准”（pivot）；
* 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
* 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。



key值的选取可以有多种形式，例如中间数或者随机数，分别会对算法的复杂度产生不同的影响。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* 快速排序
6. \*/
7. **public** **class** QuickSort {
9. **public** **static** **void** quickSort(**int**[] data, **int** low, **int** high) {
10. **int** i, j, temp, t;
11. **if** (low > high) {
12. **return**;
13. }
14. i = low;
15. j = high;
16. //temp就是基准位
17. temp = data[low];
18. System.out.println("基准位：" + temp);
20. **while** (i < j) {
21. //先看右边，依次往左递减
22. **while** (temp <= data[j] && i < j) {
23. j--;
24. }
25. //再看左边，依次往右递增
26. **while** (temp >= data[i] && i < j) {
27. i++;
28. }
29. //如果满足条件则交换
30. **if** (i < j) {
31. System.out.println("交换：" + data[i] + "和" + data[j]);
32. t = data[j];
33. data[j] = data[i];
34. data[i] = t;
35. System.out.println(java.util.Arrays.toString(data));
37. }
38. }
39. //最后将基准位与i和j相等位置的数字交换
40. System.out.println("基准位" + temp + "和i、j相遇的位置" + data[i] + "交换");
41. data[low] = data[i];
42. data[i] = temp;
43. System.out.println(java.util.Arrays.toString(data));
45. //递归调用左半数组
46. quickSort(data, low, j - 1);
47. //递归调用右半数组
48. quickSort(data, j + 1, high);
49. }

52. **public** **static** **void** main(String[] args) {
54. **int**[] data = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
56. System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
58. quickSort(data, 0, data.length - 1);
60. System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
61. }
62. }

快速排序详细参考：

<https://blog.csdn.net/shujuelin/article/details/82423852>

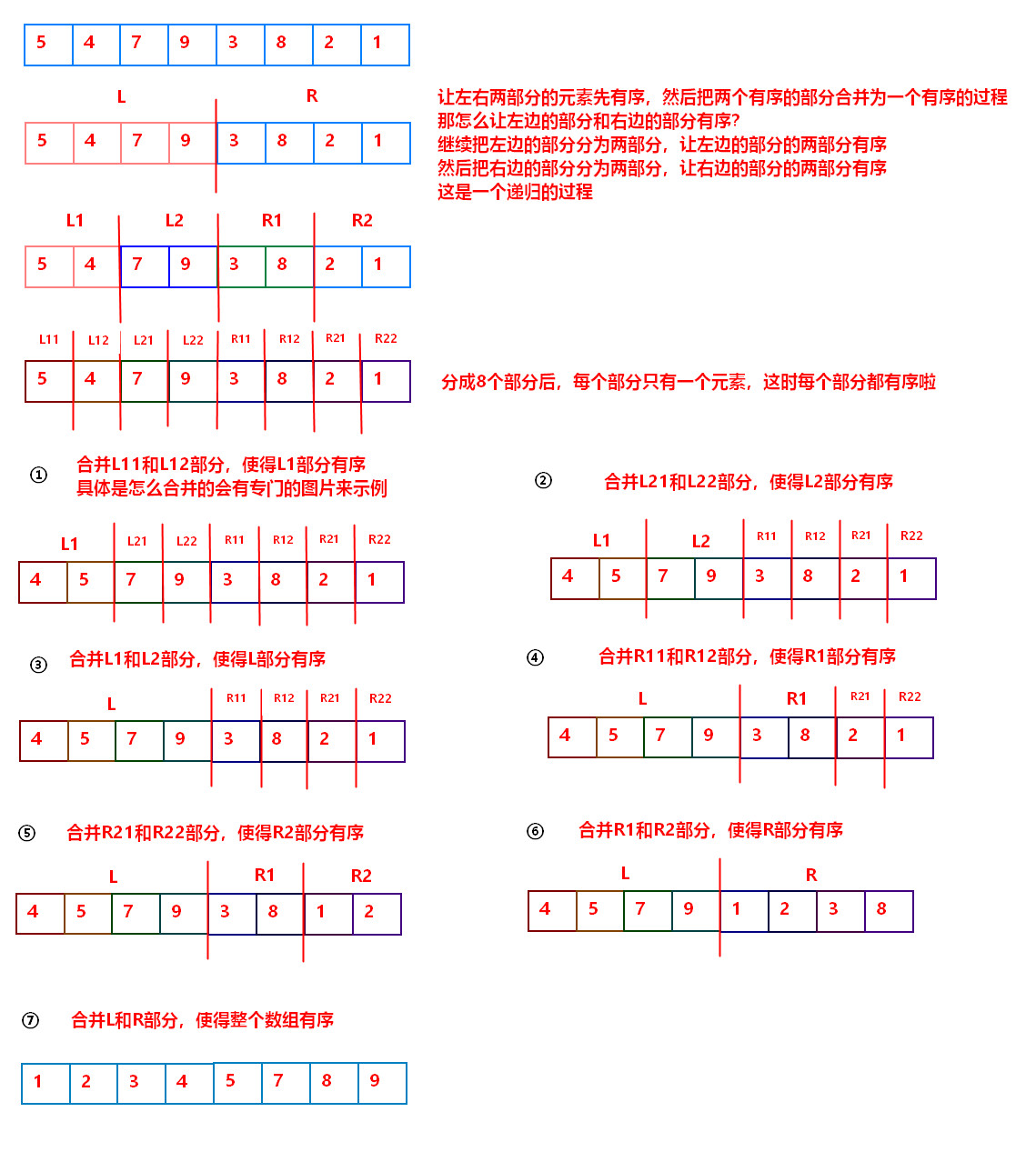
## 2.3 归并排序（Merge Sort）

**算法描述：**

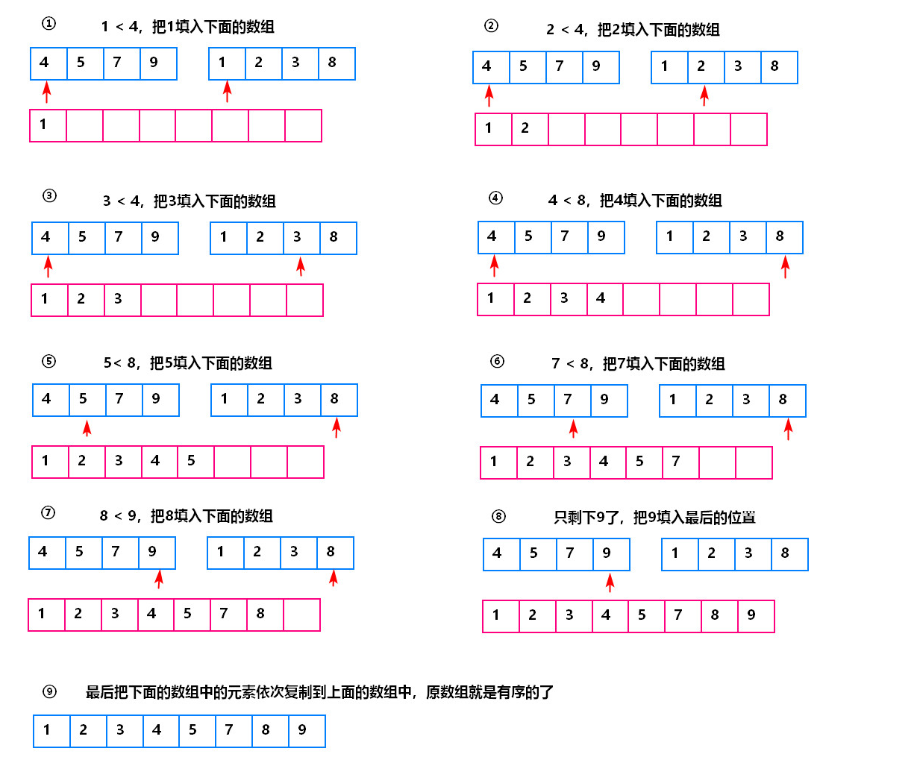
* 把长度为n的输入序列分成两个长度为n/2的子序列；
* 对这两个子序列分别采用归并排序；
* 将两个排序好的子序列合并成一个最终的排序序列。



（1）归并排序的流程



（2）合并两个有序数组的流程



**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \*/
6. **public** **class** MergeSort {
8. **public** **static** **void** mergeSort(**int**[] data) {
9. sort(data, 0, data.length - 1);
10. }
12. **public** **static** **void** sort(**int**[] arr, **int** l, **int** r) {
13. **if**(l == r) {
14. **return**;
15. }
16. **int** mid = l + ((r - l) >> 1);
17. sort(arr, l, mid);
18. sort(arr, mid + 1, r);
19. merge(arr, l, mid, r);
20. }
22. **public** **static** **void** merge(**int**[] arr, **int** l, **int** mid, **int** r) {
23. **int**[] temp = **new** **int**[r - l + 1];
24. **int** i = 0;
25. **int** p1 = l;
26. **int** p2 = mid + 1;
27. // 比较左右两部分的元素，哪个小，把那个元素填入temp中
28. **while**(p1 <= mid && p2 <= r) {
29. temp[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];
30. }
31. // 上面的循环退出后，把剩余的元素依次填入到temp中
32. // 以下两个while只有一个会执行
33. **while**(p1 <= mid) {
34. temp[i++] = arr[p1++];
35. }
36. **while**(p2 <= r) {
37. temp[i++] = arr[p2++];
38. }
39. // 把最终的排序的结果复制给原数组
40. **for**(i = 0; i < temp.length; i++) {
41. arr[l + i] = temp[i];
42. }
43. }
45. **public** **static** **void** main(String[] args) {
47. **int**[] data = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
49. System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
51. mergeSort(data);
53. System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
54. }
55. }

## 2.4 二分查找（Binary Search）

**算法描述：**

* 二分查找也称折半查找，它是一种效率较高的查找方法，要求列表中的元素首先要进行有序排列。
* 首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；
* 否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。
* 重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author Atguigu
5. \*/
6. **public** **class** BinarySearch {

9. /\*\*
10. \* 二分查找 时间复杂度O(log2n);空间复杂度O(1)
11. \*
12. \* @param arr     被查找的数组
13. \* @param left
14. \* @param right
15. \* @param findVal
16. \* @return 返回元素的索引
17. \*/
18. **public** **static** **int** binarySearch(**int**[] arr, **int** left, **int** right, **int** findVal) {
20. **if** (left > right) {//递归退出条件，找不到，返回-1
21. **return** -1;
22. }
24. **int** midIndex = (left + right) / 2;
26. **if** (findVal < arr[midIndex]) {//向左递归查找
27. **return** binarySearch(arr, left, midIndex, findVal);
28. } **else** **if** (findVal > arr[midIndex]) {//向右递归查找
29. **return** binarySearch(arr, midIndex, right, findVal);
30. } **else** {
31. **return** midIndex;
32. }
33. }
35. **public** **static** **void** main(String[] args){
37. //注意：需要对已排序的数组进行二分查找
38. **int**[] data = {-49, -30, -16, 9, 21, 21, 23, 30, 30};
39. **int** i = binarySearch(data, 0, data.length, 21);
40. System.out.println(i);
41. }
42. }

**拓展需求：**

当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
6. /\*\*
7. \* @author Atguigu
8. \*
9. \*/
10. **public** **class** BinarySearch2 {
12. /\*\*
13. \* {1, 8, 10, 89, 1000, 1000, 1234}
14. \* 一个有序数组中，有多个相同的数值，如何将所有的数值都查找到，比如这里的 1000.
15. \* 分析：
16. \* 1. 返回的结果是一个列表 list
17. \* 2. 在找到结果时，向左边扫描，向右边扫描 [条件]
18. \* 3. 找到结果后，就加入到ArrayBuffer
19. \*
20. \* @return
21. \*/
22. **public** **static** List<Integer> binarySearch2(**int**[] arr, **int** left, **int** right, **int** findVal) {
24. //找不到条件?
25. List<Integer> list = **new** ArrayList<>();
27. **if** (left > right) {//递归退出条件，找不到，返回-1
28. **return** list;
29. }
31. **int** midIndex = (left + right) / 2;
32. **int** midVal = arr[midIndex];
33. **if** (findVal < midVal) {//向左递归查找
34. **return** binarySearch2(arr, left, midIndex - 1, findVal);
35. } **else** **if** (findVal > midVal) { //向右递归查找
36. **return** binarySearch2(arr, midIndex + 1, right, findVal);
37. } **else** {
38. System.out.println("midIndex=" + midIndex);
40. //向左边扫描
41. **int** temp = midIndex - 1;
42. **while** (**true**) {
43. **if** (temp < 0 || arr[temp] != findVal) {
44. **break**;
45. }
46. **if** (arr[temp] == findVal) {
47. list.add(temp);
48. }
49. temp -= 1;
50. }
52. //将中间这个索引加入
53. list.add(midIndex);
55. //向右边扫描
56. temp = midIndex + 1;
57. **while** (**true**) {
58. **if** (temp > arr.length - 1 || arr[temp] != findVal) {
59. **break**;
60. }
61. **if** (arr[temp] == findVal) {
62. list.add(temp);
63. }
64. temp += 1;
65. }
66. **return** list;
67. }
68. }
70. **public** **static** **void** main(String[] args){
72. //注意：需要对已排序的数组进行二分查找
73. **int**[] data = {1, 8, 10, 89, 1000, 1000, 1234};
74. List<Integer> list = binarySearch2(data, 0, data.length, 1000);
75. System.out.println(list);
76. }
77. }

## 2.5 单例模式（Binary Search）

### 2.5.1单例模式定义

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态。

### 2.5.2 单例模式的特点

* 单例类只能有一个实例。
* 单例类必须自己创建自己的唯一实例。
* 单例类必须给所有其他对象提供这一实例。

单例模式保证了全局对象的唯一性，比如系统启动读取配置文件就需要单例保证配置的一致性。

### 2.5.3 单例的四大原则

* 构造私有
* 以静态方法或者枚举返回实例
* 确保实例只有一个，尤其是多线程环境
* 确保反序列换时不会重新构建对象

### 2.5.4 实现单例模式的方式

**（1）饿汉式（立即加载）：**

饿汉式单例在类加载初始化时就创建好一个静态的对象供外部使用，除非系统重启，这个对象不会改变，所以本身就是线程安全的。

Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。（事实上，通过Java反射机制是能够实例化构造方法为private的类的，会使Java单例实现失效）

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* 饿汉式（立即加载）
6. \*/
7. **public** **class** Singleton1 {
9. /\*\*
10. \* 私有构造
11. \*/
12. **private** Singleton1() {
13. System.out.println("构造函数Singleton1");
14. }
16. /\*\*
17. \* 初始值为实例对象
18. \*/
19. **private** **static** Singleton1 single = **new** Singleton1();
21. /\*\*
22. \* 静态工厂方法
23. \* @return 单例对象
24. \*/
25. **public** **static** Singleton1 getInstance() {
26. System.out.println("getInstance");
27. **return** single;
28. }
30. **public** **static** **void** main(String[] args){
31. System.out.println("初始化");
32. Singleton1 instance = Singleton1.getInstance();
33. }
34. }
35. **懒汉式（延迟加载）：**

该示例虽然用延迟加载方式实现了懒汉式单例，但在多线程环境下会产生多个Singleton对象

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \*
6. \*
7. \* 懒汉式（延迟加载）
8. \*/
9. **public** **class** Singleton2 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton2() {
15. System.out.println("构造函数Singleton2");
16. }
18. /\*\*
19. \* 初始值为null
20. \*/
21. **private** **static** Singleton2 single = null;
23. /\*\*
24. \* 静态工厂方法
25. \* @return 单例对象
26. \*/
27. **public** **static** Singleton2 getInstance() {
28. **if**(single == null){
29. System.out.println("getInstance");
30. single = **new** Singleton2();
31. }
32. **return** single;
33. }
35. **public** **static** **void** main(String[] args){
37. System.out.println("初始化");
38. Singleton2 instance = Singleton2.getInstance();
39. }
40. }
41. **同步锁（解决线程安全问题）：**

在方法上加synchronized同步锁或是用同步代码块对类加同步锁，此种方式虽然解决了多个实例对象问题，但是该方式运行效率却很低下，下一个线程想要获取对象，就必须等待上一个线程释放锁之后，才可以继续运行。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \*
6. \*
7. \* 同步锁（解决线程安全问题）
8. \*/
9. **public** **class** Singleton3 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton3() {}
16. /\*\*
17. \* 初始值为null
18. \*/
19. **private** **static** Singleton3 single = null;
21. **public** **static** Singleton3 getInstance() {
23. // 等同于 synchronized public static Singleton3 getInstance()
24. synchronized(Singleton3.**class**){
25. // 注意：里面的判断是一定要加的，否则出现线程安全问题
26. **if**(single == null){
27. single = **new** Singleton3();
28. }
29. }
30. **return** single;
31. }
32. }

**（4）双重检查锁（提高同步锁的效率）：**

使用双重检查锁进一步做了优化，可以避免整个方法被锁，只对需要锁的代码部分加锁，可以提高执行效率。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \*
6. \* 双重检查锁（提高同步锁的效率）
7. \*/
8. **public** **class** Singleton4 {
10. /\*\*
11. \* 私有构造
12. \*/
13. **private** Singleton4() {}
15. /\*\*
16. \* 初始值为null
17. \*/
18. **private** **static** Singleton4 single = null;
20. /\*\*
21. \* 双重检查锁
22. \* @return 单例对象
23. \*/
24. **public** **static** Singleton4 getInstance() {
25. **if** (single == null) {
26. synchronized (Singleton4.**class**) {
27. **if** (single == null) {
28. single = **new** Singleton4();
29. }
30. }
31. }
32. **return** single;
33. }
34. }

**（5） 静态内部类：**

这种方式引入了一个内部静态类（static class），静态内部类只有在调用时才会加载，它保证了Singleton 实例的延迟初始化，又保证了实例的唯一性。它把singleton 的实例化操作放到一个静态内部类中，在第一次调用getInstance() 方法时，JVM才会去加载InnerObject类，同时初始化singleton 实例，所以能让getInstance() 方法线程安全。

特点是：即能延迟加载，也能保证线程安全。

静态内部类虽然保证了单例在多线程并发下的线程安全性，但是在遇到序列化对象时，默认的方式运行得到的结果就是多例的。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \*
6. \*
7. \* 静态内部类（延迟加载，线程安全）
8. \*/
9. **public** **class** Singleton5 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton5() {}
16. /\*\*
17. \* 静态内部类
18. \*/
19. **private** **static** **class** InnerObject{
20. **private** **static** Singleton5 single = **new** Singleton5();
21. }
23. **public** **static** Singleton5 getInstance() {
24. **return** InnerObject.single;
25. }
26. }

**（6）内部枚举类实现（防止反射攻击）：**

事实上，通过Java反射机制是能够实例化构造方法为private的类的。这也就是我们现在需要引入的枚举单例模式。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \*
6. \*/
7. **public** **class** SingletonFactory {
9. /\*\*
10. \* 内部枚举类
11. \*/
12. **private** **enum** EnumSingleton{
13. Singleton;
14. **private** Singleton6 singleton;
16. //枚举类的构造方法在类加载是被实例化
17. **private** EnumSingleton(){
18. singleton = **new** Singleton6();
19. }
20. **public** Singleton6 getInstance(){
21. **return** singleton;
22. }
23. }
25. **public** **static** Singleton6 getInstance() {
26. **return** EnumSingleton.Singleton.getInstance();
27. }
28. }
30. **class** Singleton6 {
31. **public** Singleton6(){}
32. }

# 第3章 Java SE

## 3.1 你是怎样理解面向对象的

面向对象是利于语言对现实事物进行抽象。面向对象具有以下四大特征：

**（1）继承：**继承是从已有类得到继承信息创建新类的过程

**（2）封装：**通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。

**（3）多态性：**多态性是指允许不同子类型的对象对同一消息作出不同的响应。

**（4）抽象：**抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。

## 3.2 int和Integer有什么区别，以及以下程序结果

（1）Integer是int的包装类，int则是java的一种基本数据类型

（2）Integer变量必须实例化后才能使用，而int变量不需要

（3）Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值

（4）Integer的默认值是null，int的默认值是0

（5）java在编译Integer i = 100 ;时，会翻译成为Integer i = Integer.valueOf(100)。而java API中对Integer类型的valueOf的定义如下，对于-128到127之间的数，会进行缓存，Integer i = 127时，会将127这个Integer对象进行缓存，下次再写Integer j = 127时，就会直接从缓存中取，就不会new了

1. package com.atguigu.interview.chapter03;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/28
6. \*/
7. **public** **class** Test01 {
9. **public** **static** **void** main(String[] args){
10. Integer a = 127;
11. Integer b = 127;
12. Integer c = 128;
13. Integer d = 128;
14. System.out.println(a==b); //true
15. System.out.println(c==d); //false
16. }
17. }

## 3.3 ==和Equals区别

（1） ==

如果比较的是基本数据类型，那么比较的是变量的值

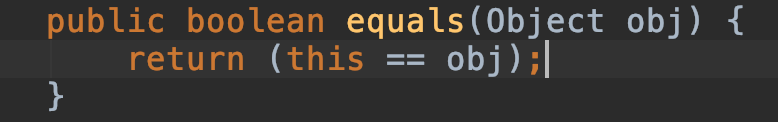
如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

1. equals

如果没重写equals方法比较的是两个对象的地址值

如果重写了equals方法后我们往往比较的是对象中的属性的内容

equals方法是从Object类中继承的，默认的实现就是使用==



## 3.4谈谈你对反射的理解

**（1）反射机制:**

所谓的反射机制就是java语言在运行时拥有一项自观的能力。通过这种能力可以彻底的了解自身的情况为下一步的动作做准备。

Java的反射机制的实现要借助于4个类：class，Constructor，Field，Method;

其中class代表的时类对 象，Constructor－类的构造器对象，Field－类的属性对象，Method－类的方法对象。通过这四个对象我们可以粗略的看到一个类的各个组成部分。

**（2）Java反射的作用：**

在Java运行时环境中，对于任意一个类，可以知道这个类有哪些属性和方法。对于任意一个对象，可以调用它的任意一个方法。这种动态获取类的信息以及动态调用对象的方法的功能来自于Java 语言的反射（Reflection）机制。

**（3）Java 反射机制提供功能**

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。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时调用任意一个对象的方法

## 3.5 ArrarList和LinkedList区别

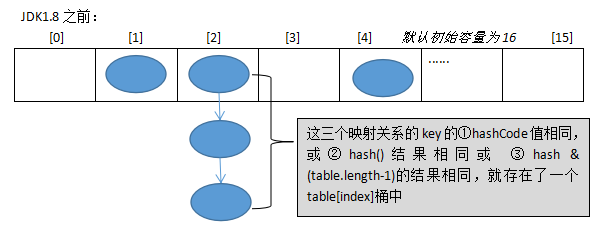
（1）ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

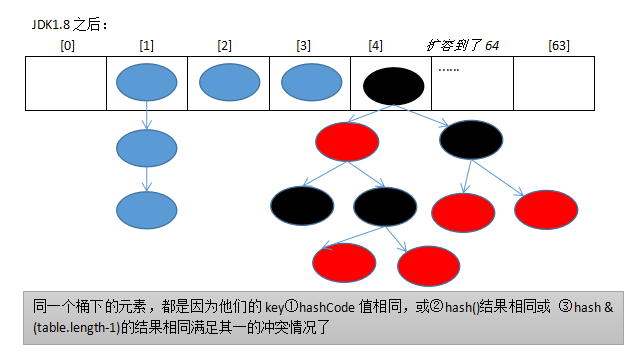
（2）对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

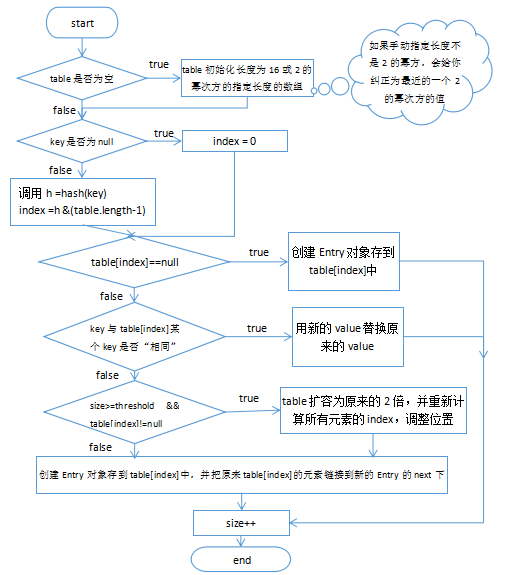
（3）对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。 这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

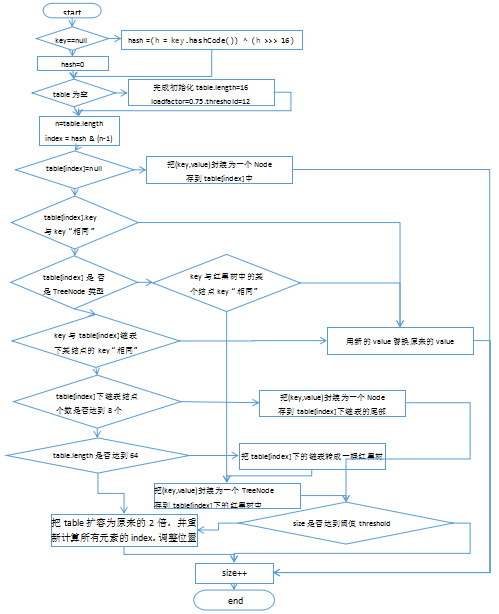
## 3.6 HashMap底层源码，数据结构

HashMap的底层结构在jdk1.7中由数组+链表实现，在jdk1.8中由数组+链表+红黑树实现，以数组+链表的结构为例。





**JDK1.8之前Put方法：**

**JDK1.8之后Put方法：**

## 3.7 HashMap和HashTable区别

**（1）线程安全性不同**

HashMap是线程不安全的，HashTable是线程安全的，其中的方法是Synchronize的，在多线程并发的情况下，可以直接使用HashTabl，但是使用HashMap时必须自己增加同步处理。

**（2）是否提供contains方法**

HashMap只有containsValue和containsKey方法；HashTable有contains、containsKey和containsValue三个方法，其中contains和containsValue方法功能相同。

**（3）key和value是否允许null值**

Hashtable中，key和value都不允许出现null值。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。

**（4）数组初始化和扩容机制**

HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

## 3.8 TreeSet和HashSet区别

HashSet是采用hash表来实现的。其中的元素没有按顺序排列，add()、remove()以及contains()等方法都是复杂度为O(1)的方法。

TreeSet是采用树结构实现(红黑树算法)。元素是按顺序进行排列，但是add()、remove()以及contains()等方法都是复杂度为O(log (n))的方法。它还提供了一些方法来处理排序的set，如first(), last(), headSet(), tailSet()等等。

## 3.9 String buffer和String builder区别

（1）StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，

（2）只是StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。

（3）在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全而StringBuffer则每次都需要判断锁，效率相对更低

## 3.11 什么是 Java 序列化，如何实现 Java 序列化？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

序 列 化 的 实 现 ： 将 需 要 被 序 列 化 的 类 实 现 Serializable 接 口 ， 该 接 口 没 有 需 要 实 现 的 方 法 ， implements Serializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的 writeObject(Object obj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。

## 3.12 Object中有哪些方法

（1）protected Object clone()--->创建并返回此对象的一个副本。   
（2）boolean equals(Object obj)--->指示某个其他对象是否与此对象“相等”。   
（3）protected void finalize()--->当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。   
（4）Class<? extendsObject> getClass()--->返回一个对象的运行时类。   
（5）int hashCode()--->返回该对象的哈希码值。   
（6）void notify()--->唤醒在此对象监视器上等待的单个线程。   
（7）void notifyAll()--->唤醒在此对象监视器上等待的所有线程。   
（8）String toString()--->返回该对象的字符串表示。   
（9）void wait()--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。   
 void wait(long timeout)--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll()方法，或者超过指定的时间量。   
 void wait(long timeout, int nanos)--->导致当前的线程等待，直到其他线程调用此对象的 notify()

## 3.13线程有几种状态，产生的条件是什么



1. 新建状态(New) ：线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。  
   （2）就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。  
   （3）运行状态(Running)：线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。  
   （4）阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

* 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。
* 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。
* 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join(）等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

（5）死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 3.14产生死锁的基本条件

**产生死锁的原因：**  
（1） 因为系统资源不足。  
（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。  
**死锁的解除与预防：**  
 理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和  
解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确  
定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态  
的情况下占用资源。因此，对资源的分配要给予合理的规划。

## 3.15什么是线程池，如何使用？

线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用 new 线程而是直接去池中拿线程即可，节省了开辟子线程的时间，提高的代码执行效率。

在 JDK 的 java.util.concurrent.Executors 中提供了生成多种线程池的静态方法。

ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();

ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(4);

ScheduledExecutorService newScheduledThreadPool = Executors.newScheduledThreadPool(4);

ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();

然后调用他们的 execute 方法即可。

**优点：**

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

## 3.16 Java自带有哪几种线程池？

**（1）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。这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

**（2）newFixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

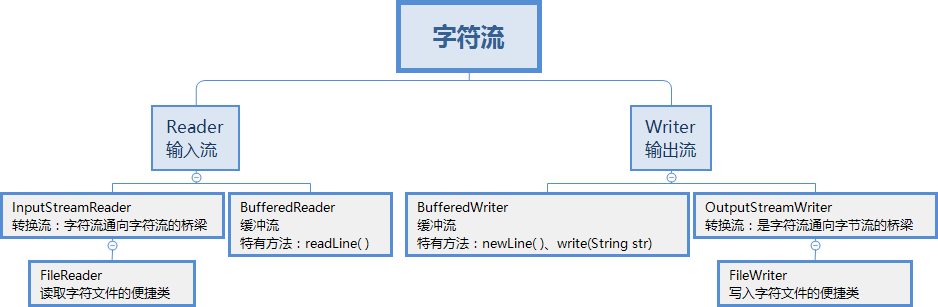
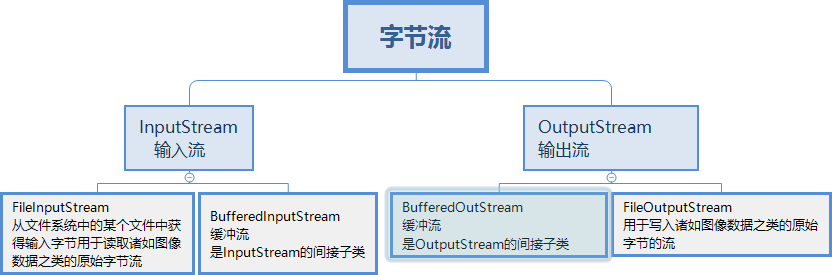
**（3）newSingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，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。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

**（4）newScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行。例如延迟3秒执行。

## 3.17 Java 中有几种类型的流



## 3.18字节流如何转为字符流

字节输入流转字符输入流通过 InputStreamReader 实现，该类的构造函数可以传入 InputStream 对象。

字节输出流转字符输出流通过OutputStreamWriter 实现，该类的构造函数可以传入 OutputStream 对象。

## 3.19请写出你最常见的5个RuntimeException

（1）java.lang.NullPointerException 空指针异常；出现原因：调用了未经初始化的对象或者是不存在的对象。

（2）java.lang.ClassNotFoundException 指定的类找不到；出现原因：类的名称和路径加载错误；通常都是程序试图通过字符串来加载某个类时可能引发异常。

（3）java.lang.NumberFormatException 字符串转换为数字异常；出现原因：字符型数据中包含非数字型字符。

（4）java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生。

（5）java.lang.IllegalArgumentException 方法传递参数错误。

（6）java.lang.ClassCastException 数据类型转换异常。

## 3.20JDK和JRE有什么区别？

（1）JRE是Java运行时环境，就是Java程序必须运行在JRE上面，它就是执行Java程序的虚拟机。

（2）JDK是包含JRE的，它比JRE多了一些工具，这些多出来的工具就是让我们程序员来开发Java程序的。

（3）JDK包含JRE，编译器和其他的工具(比如：JavaDoc，Java调试器)，可以让开发者开发、编译、执行Java应用程序

（4）JDK面向开发者，JRE面向程序使用者

## 3.21两个对象的 hashCode()相同，则 equals()也一定为 true，对吗？

（1）两个对象equals相等，则它们的hashcode必须相等，反之则不一定。

（2）两个对象==相等，则其hashcode一定相等，反之不一定成立

## 3.22final 在 java 中有什么作用？

final作为Java中的关键字可以用于三个地方。用于修饰类、类属性和类方法。

特征：凡是引用final关键字的地方皆不可修改！

(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被重写；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

## 3.23Java 中的 Math.round(-1.5) 等于多少？

Math.round(-1.5)的返回值是-1。四舍五入的原理是在参数上加0.5然后做向下取整

## 3.24String 属于基础的数据类型吗？

在Java中，数据类型分为引用类型和基本类型，基本类型分为八种

整型byte，short，int,long

浮点型：float，double

字符型：char

Boolean型：boolean

String不是基本的数据类型，是final修饰的java类，是引用类型。

## 3.25 Java 中操作字符串都有哪些类？它们之间有什么区别？

主要是一下三种：String、StringBuffer、StringBuilder

特别是在项目中。先来看一下这三种操作方式的区别：

String是不可变的对象，对每次对String类型的改变时都会生成一个新的对象，

StringBuffer和StringBuilder是可以改变对象的。

对于操作效率：StringBuilder > StringBuffer > String

对于线程安全：StringBuffer 是线程安全，可用于多线程；

StringBuilder 是非线程安全，用于单线程

不频繁的字符串操作使用 String。反之，StringBuffer 和 StringBuilder 都优于String

所以，如果在项目中需要拼接字符串最好是采用StringBuffer 而非String.

## 3.26String str="i"与 String str=new String(“i”)一样吗？

不一样，因为内存的分配方式不一样。String str="i"的方式，Java 虚拟机会将其分配到常量池中；而 String str=new String(“i”) 则会被分到堆内存中。

## 3.28String 类的常用方法都有那些？

indexOf() 返回指定字符得索引

charAt() 返回指定索引处得字符

repalce() 字符串替换

trim() 去除字符串两端的空白

split() 分割字符串 返回分割后的字符串数组

getBytes() 返回字符串的byte类型数组

length() 返回字符串的长度

toLowerCase() 字符串转小写

toUpperCase() 字符串转大写

substring() 截取字符串

equals() 字符串比较

## 3.29抽象类必须要有抽象方法吗？

抽象类可以没有抽象方法，但是如果你的一个类已经声明成了抽象类，即使这个类中没有抽象方法，它也不能再实例化，即不能直接构造一个该类的对象。

如果一个类中有了一个抽象方法，那么这个类必须声明为抽象类，否则编译通不过。

## 3.30普通类和抽象类有哪些区别？

普通类不能包含抽象方法，抽象类可以包含抽象方法。

抽象类是不能被实例化的，就是不能用new调出构造方法创建对象，普通类可以直接实例化。

如果一个类继承于抽象类，则该子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为abstract类。

## 3.31抽象类能使用 final 修饰吗？

不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承，这样彼此就会产生矛盾，所以 final 不能修饰抽象类，如下图所示，编辑器也会提示错误信息。

## 3.32接口和抽象类有什么区别？

他们都不能实例化对象，都可以包含抽象方法，而且抽象方法必须被继承的类全部实现。

区别：

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。

6、抽象方法只能申明，不能实现，接口是设计的结果 ，抽象类是重构的结果

7、抽象类里可以没有抽象方法

8、如果一个类里有抽象方法，那么这个类只能是抽象类

9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

10、接口可继承接口，并可多继承接口，但类只能单根继承。

## 3.33Java 中 IO 流分为几种？

按照流的流向分，可以分为输入流和输出流；

按照操作单元划分，可以划分为字节流和字符流；

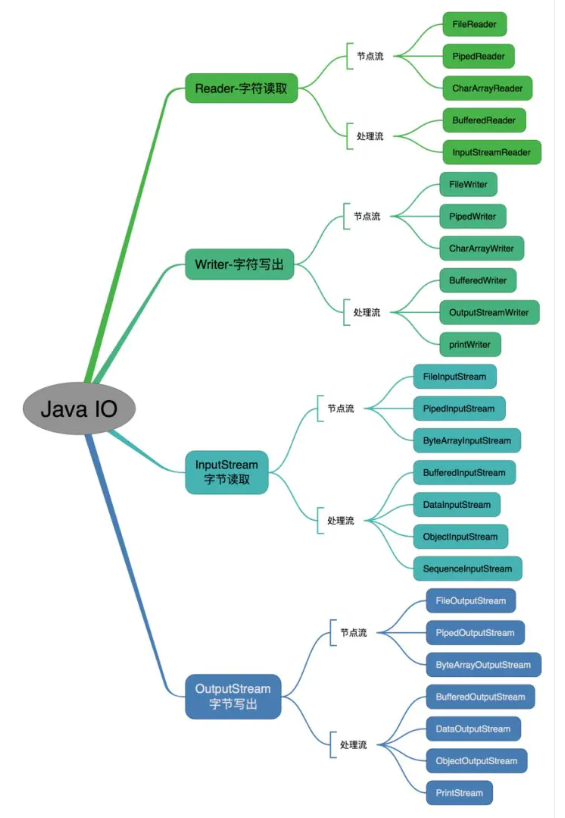
按照流的角色划分为节点流和处理流。

Java Io流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系， Java I0流的40多个类都是从如下4个抽象类基类中派生出来的。

InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。

OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作方式分类结构图：



## 3.34BIO、NIO、AIO 有什么区别？

1）BIO

在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

简单来说，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行，BIO方式适用于连接数目比较小且固定的架构。

2）NIO

NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。 当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理。

简单来说，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪（引入不必要的CPU资源浪费）

NIO方式适用于连接数目多且连接比较短的架构。

BIO的每个连接一个单独的线程，而NIO则是每个连接共用一个线程。

3）AIO

当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。read/write方法都是异步的，完成后会主动调用回调函数。 AIO是一个有效请求一个线程。

简单来说，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知。不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

AIO方式使用于连接数目多且连接比较长的架构。

## 3.35Files的常用方法都有哪些？

Files. exists()：检测文件路径是否存在。

Files. createFile()：创建文件。

Files. createDirectory()：创建文件夹。

Files. delete()：删除一个文件或目录。

Files. copy()：复制文件。

Files. move()：移动文件。

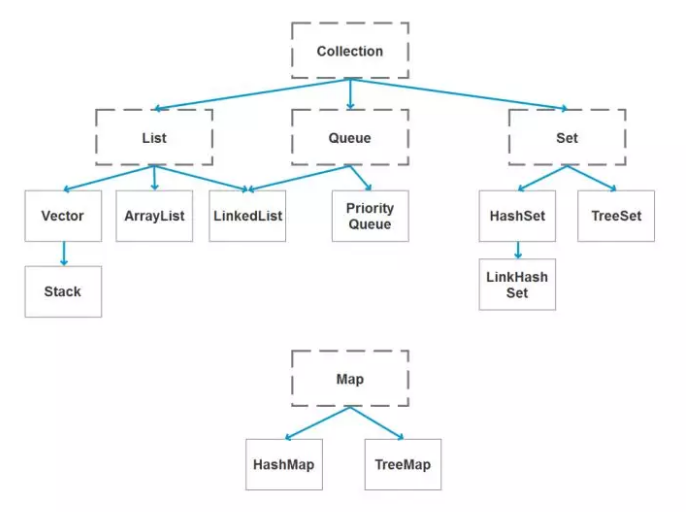
Files. size()：查看文件个数。

Files. read()：读取文件。

Files. write()：写入文件。

## 3.36Java 容器都有哪些？

如图所示



## 3.37Collection 和 Collections 有什么区别？

java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

## 3.38List、Set、Map 之间的区别是什么？



## 3.39说一下 HashSet 的实现原理？

（1）基于HashMap实现的，默认构造函数是构建一个初始容量为16，负载因子为0.75 的HashMap。封装了一个 HashMap 对象来存储所有的集合元素，所有放入 HashSet 中的集合元素实际上由 HashMap 的 key 来保存，而 HashMap 的 value 则存储了一个 PRESENT，它是一个静态的 Object 对象。

（2）当我们试图把某个类的对象当成 HashMap的 key，或试图将这个类的对象放入 HashSet 中保存时，重写该类的equals(Object obj)方法和 hashCode() 方法很重要，而且这两个方法的返回值必须保持一致：当该类的两个的 hashCode() 返回值相同时，它们通过 equals() 方法比较也应该返回 true。通常来说，所有参与计算 hashCode() 返回值的关键属性，都应该用于作为 equals() 比较的标准。

（3）HashSet的其他操作都是基于HashMap的。

## 3.40如何决定使用 HashMap 还是 TreeMap？

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

## 3.41如何实现数组和 List 之间的转换？

List转数组：toArray(arraylist.size()方法

数组转List：Arrays的asList(a)方法

## 3.42哪些集合类是线程安全的？

在集合框架中，有些类是线程安全的，这些都是jdk1.1中的出现的。在jdk1.2之后，就出现许许多多非线程安全的类。 下面是这些线程安全的同步的类：

vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。

statck：堆栈类，先进后出

hashtable：就比hashmap多了个线程安全

enumeration：枚举，相当于迭代器

StringBuffer也是线程安全的，（StringBuilder 不是线程安全的）

除了这些之外，其他的都是非线程安全的类和接口。

线程安全的类其方法是同步的，每次只能一个访问。是重量级对象，效率较低。

## 3.43 Iterator 怎么使用？有什么特点？

1.Iterator在遍历元素过程中，有线程修改集合元素会有ConcurrentModificationEception异常

2.Iterator本身不具有装载数据功能，需依附Collection对象使用

3.next（）是用游标指向的方式返回下一个元素的

## 3.44Iterator 和 ListIterator 有什么区别？

我们需要知道的第一个则是：

（1）所属关系，ListIterator是一个Iterator的子类型。

（2）局限：只能应用于各种List类的访问。

（3）优势：Iterator只能向前移动，而ListIterator可以双向移动。

还可以产生相对于迭代器在列表中指向的当前位置的前一个和后一个元素的索引

nextIndex()、previousIndex()方法。

还可以通过set（）方法替换它访问过的最后一个元素。

还可以通过调用listIterator（）方法产生一个指向List开始处的ListIterator，当然也可以有参数，即指向索引为参数处的ListIterator。

（4）ListIterator 有 add() 方法，可以向 List 中添加对象，而 Iterator 不能

## 3.45创建线程有哪几种方式？

1.继承Thread类实现多线程

2.覆写Runnable()接口实现多线程，而后同样覆写run().推荐此方式

3.覆写Callable接口实现多线程（JDK1.5）

4.通过线程池启动多线程

## 3.46说一下 runnable 和 callable 有什么区别？

Runnable应该是比较熟悉的接口，它只有一个run()函数，用于将耗时操作写在其中，该函数没有返回值，不能将结果返回给客户程序。然后使用某个线程去执行runnable即可实现多线程，Thread类在调用start()函数后就是执行的是Runnable的run()函数。

Callable与Runnable的功能大致相似，Callable中有一个call()函数，但是call()函数有返回值。

主要区别：

Runnable 接口 run 方法无返回值；Callable 接口 call 方法有返回值，支持泛型

Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；Callable 接口 call 方法允许抛出异常，可以获取异常信息

## 3.47sleep() 和 wait() 有什么区别？

sleep方法：

　　属于Thread类中的方法；会导致程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持着，当指定时间到了之后，又会自动恢复运行状态；在调用sleep方法的过程中，线程不会释放对象锁。（只会让出CPU，不会导致锁行为的改变）

wait方法：

　　属于Object类中的方法；在调用wait方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify方法后本线程才进入对象锁定池准备。获取对象锁进入运行状态。（不仅让出CPU，还释放已经占有的同步资源锁）

## 3.48notify()和 notifyAll()有什么区别？

notify():

唤醒一个处于等待状态的线程，

注意的是在调用此方法的时候，

并不能确切的唤醒某一个等待状态的线程，

而是由JVM确定唤醒哪个线程，而且不是按优先级。

notifyAll():

唤醒所有处入等待状态的线程;

并可以理解为把他们排进一个队列;

只不过只有头部的线程获得了锁，才能运行;

注意！！并不是给所有唤醒线程一个对象的锁，而是让它们竞争，

当其中一个线程运行完就开始运行下一个已经被唤醒的线程，因为锁已经转移了。

## 3.49线程的 run()和 start()有什么区别？

run()方法：

　　是在主线程中执行方法，和调用普通方法一样；（按顺序执行，同步执行）

start()方法：

　　是创建了新的线程，在新的线程中执行；（异步执行）

## 3.50线程池中 submit()和 execute()方法有什么区别？

execute() 参数 Runnable ；submit() 参数 (Runnable) 或 (Runnable 和 结果 T) 或 (Callable)

execute() 没有返回值；而 submit() 有返回值

submit() 的返回值 Future 调用get方法时，可以捕获处理异常

## 3.51在 java 程序中怎么保证多线程的运行安全？

线程的安全性问题体现在：

原子性：一个或者多个操作在 CPU 执行的过程中不被中断的特性

可见性：一个线程对共享变量的修改，另外一个线程能够立刻看到

有序性：程序执行的顺序按照代码的先后顺序执行

导致原因：

缓存导致的可见性问题

线程切换带来的原子性问题

编译优化带来的有序性问题

解决办法：

JDK Atomic开头的原子类、synchronized、LOCK，可以解决原子性问题

synchronized、volatile、LOCK，可以解决可见性问题

Happens-Before 规则可以解决有序性问题

Happens-Before 规则如下：

程序次序规则：在一个线程内，按照程序控制流顺序，书写在前面的操作先行发生于书写在后面的操作

管程锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作

volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作

线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作

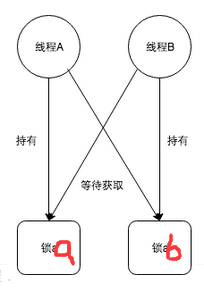
线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始

## 3.52什么是死锁？

所谓死锁，是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。 因此我们举个例子来描述，如果此时有一个线程A，按照先锁a再获得锁b的的顺序获得锁，而在此同时又有另外一个线程B，按照先锁b再锁a的顺序获得锁。如下图所示：



## 3.53怎么防止死锁？

预防死锁：

资源一次性分配：一次性分配所有资源，这样就不会再有请求了：（破坏请求条件）

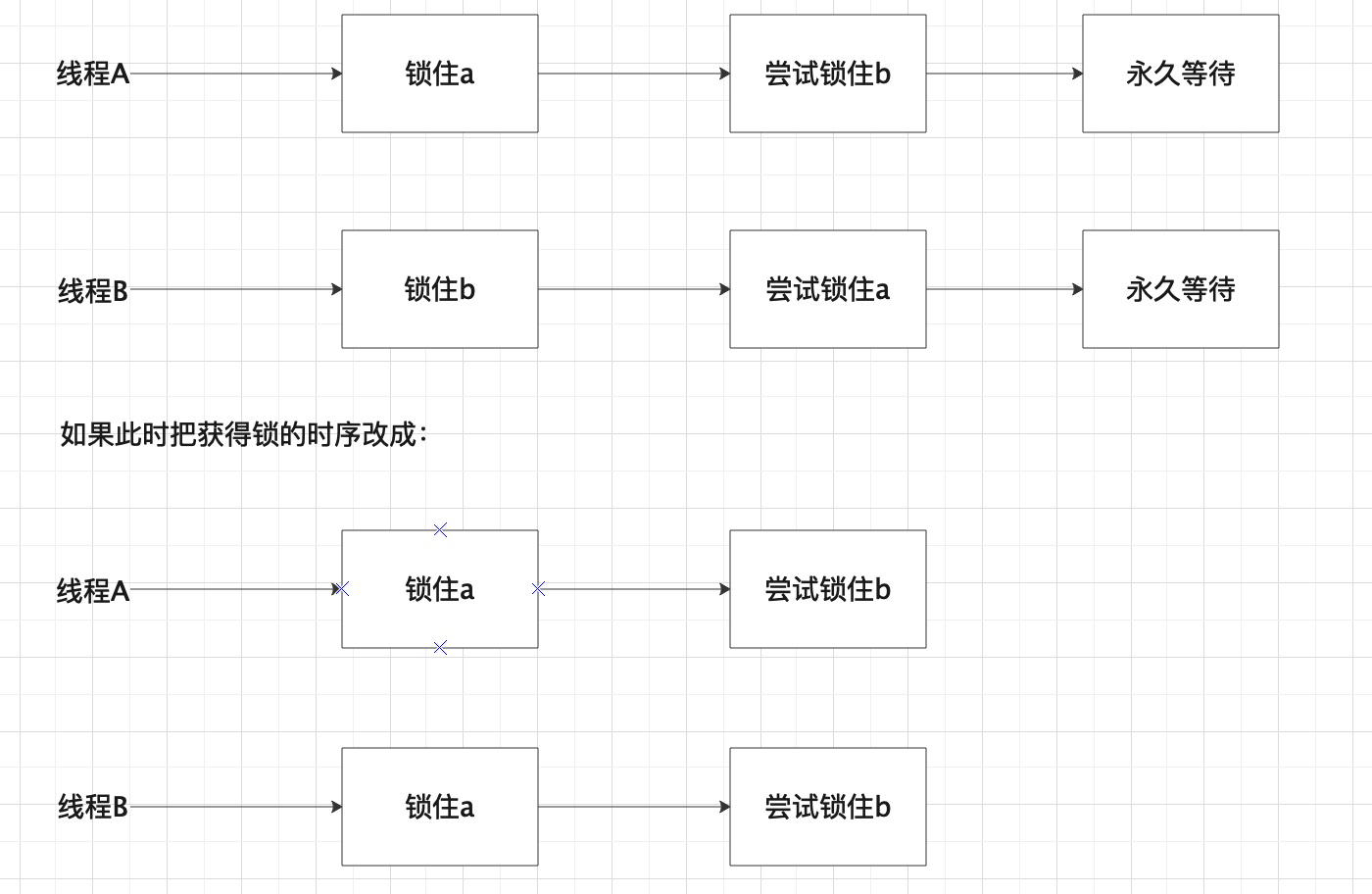
只要有一个资源得不到分配，也不给这个进程分配其他的资源：（破坏请保持条件）

可剥夺资源：即当某进程获得了部分资源，但得不到其它资源，则释放已占有的资源（破坏不可剥夺条件）

资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

1、以确定的顺序获得锁

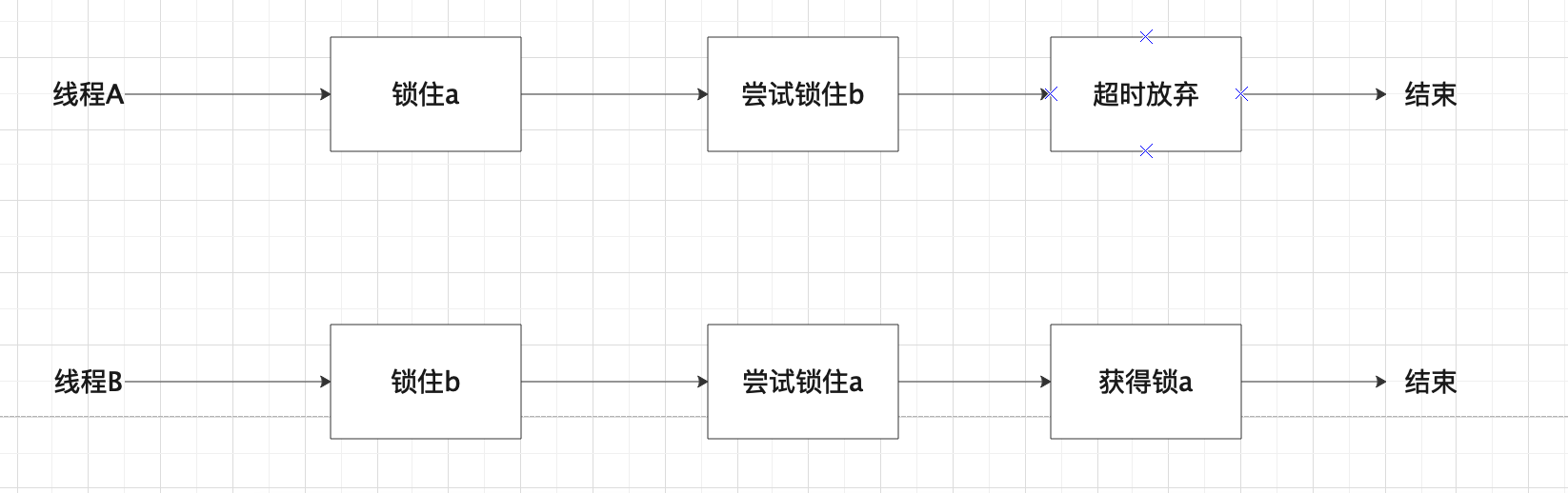
如果必须获取多个锁，那么在设计的时候需要充分考虑不同线程之前获得锁的顺序。按照上面的例子，两个线程获得锁的时序图如下：



那么死锁就永远不会发生。 针对两个特定的锁，开发者可以尝试按照锁对象的hashCode值大小的顺序，分别获得两个锁，这样锁总是会以特定的顺序获得锁，那么死锁也不会发生。问题变得更加复杂一些，如果此时有多个线程，都在竞争不同的锁，简单按照锁对象的hashCode进行排序（单纯按照hashCode顺序排序会出现“环路等待”），可能就无法满足要求了，这个时候开发者可以使用银行家算法，所有的锁都按照特定的顺序获取，同样可以防止死锁的发生，该算法在这里就不再赘述了，有兴趣的可以自行了解一下。

2、超时放弃

当使用synchronized关键词提供的内置锁时，只要线程没有获得锁，那么就会永远等待下去，然而Lock接口提供了boolean tryLock(long time, TimeUnit unit) throws InterruptedException方法，该方法可以按照固定时长等待锁，因此线程可以在获取锁超时以后，主动释放之前已经获得的所有的锁。通过这种方式，也可以很有效地避免死锁。 还是按照之前的例子，时序图如下：



避免死锁:

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得 较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全的状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。

银行家算法：首先需要定义状态和安全状态的概念。系统的状态是当前给进程分配的资源情况。因此，状态包含两个向量Resource（系统中每种资源的总量）和Available（未分配给进程的每种资源的总量）及两个矩阵Claim（表示进程对资源的需求）和Allocation（表示当前分配给进程的资源）。安全状态是指至少有一个资源分配序列不会导致死锁。当进程请求一组资源时，假设同意该请求，从而改变了系统的状态，然后确定其结果是否还处于安全状态。如果是，同意这个请求；如果不是，阻塞该进程知道同意该请求后系统状态仍然是安全的。

## 3.54 synchronized 和 volatile 的区别是什么？

1)volatile比synchronized更轻量级。

2)volatile没有synchronized使用的广泛。

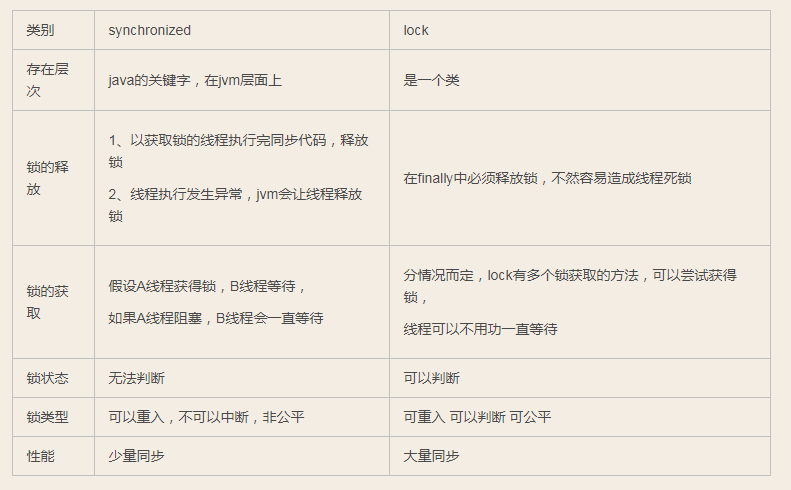
3)volatile不需要加锁，比synchronized更轻量级，不会阻塞线程。

4)从内存可见性角度看，volatile读相当于加锁，volatile写相当于解锁。

5)synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性。

6)volatile本身不保证获取和设置操作的原子性，仅仅保持修改的可见性。但是java的内存模型保证声明为volatile的long和double变量的get和set操作是原子的。

## 3.55synchronized 和 Lock 有什么区别？



## 3.56 synchronized 和 ReentrantLock 区别是什么？

**基本意义**：Synchronized是Java语言的关键字，因此Synchronized的锁是原生语法层面的互斥，需要JVM来实现。具体是通过对象内部的一个叫做监视器锁（monitor）来实现的。ReentrantLock，字面意思可重入锁，它是JDK1.5之后提供的API层面的互斥锁，锁的功能主要由2个方法完成，即lock()和unlock()。

**易用性**：Synchronized的使用比较方便简洁，由编译器去保证锁的加锁和释放，而ReentrantLock需要手动写代码来加锁和释放锁。  
注意：为避免忘记手工释放锁而造成了死锁，最好在finally中声明释放锁。

**灵活度**：ReentrantLock要优于Synchronized，可以灵活控制在哪个位置加锁和解锁。

性能区别

我们平时写代码的时候用到Java最多的锁是Synchronized，单例模式中看到的锁也是Synchronized。这是为什么呢？除了Synchronized使用方面一些，其实和ReentrantLock对比，性能也丝毫不逊色，这里面说的当然是JDK1.5以后的版本了。

JDK1.6以前的版本没有优化，这时和ReentrantLock比肯定差很多。优化以后的Synchronized引入了包括偏向锁，轻量级锁等，这样就和ReentrantLock性能差不多了。

## 3.57什么是反射？

Java 反射机制是在运行状态中，对于任意一个类，都能够获得这个类的所有属性和方法，对于任意一个对象都能够调用它的任意一个属性和方法。这种在运行时动态的获取信息以及动态调用对象的方法的功能称为Java 的反射机制。

Class 类与java.lang.reflect 类库一起对反射的概念进行了支持，该类库包含了Field,Method,Constructor类(每个类都实现了Member 接口)。这些类型的对象时由JVM 在运行时创建的，用以表示未知类里对应的成员。

这样就可以使用Constructor 创建新的对象，用get() 和set() 方法读取和修改与Field 对象关联的字段，用invoke() 方法调用与Method 对象关联的方法。另外，还可以调用getFields() getMethods() 和 getConstructors() 等很便利的方法，以返回表示字段，方法，以及构造器的对象的数组。这样匿名对象的信息就能在运行时被完全确定下来，而在编译时不需要知道任何事情。

## 3.58动态代理是什么？有哪些应用？

动态代理是运行时动态生成代理类。 动态代理的应用有 spring aop、hibernate 数据查询、测试框架的后端 mock、rpc，Java注解对象获取等。

## 3.59深拷贝和浅拷贝区别是什么？

浅拷贝：创建一个新对象，然后将当前对象的非静态字段复制到该新对象，如果字段是值类型的，那么对该字段执行复制；如果该字段是引用类型的话，则复制引用但不复制引用的对象。因此，原始对象及其副本引用同一个对象。

深拷贝：创建一个新对象，然后将当前对象的非静态字段复制到该新对象，无论该字段是值类型的还是引用类型，都复制独立的一份。当你修改其中一个对象的任何内容时，都不会影响另一个对象的内容。

## 3.60throw 和 throws 的区别？

throw:

(1)throw 语句用在方法体内，表示抛出异常，由方法体内的语句处理。

(2)throw 是具体向外抛出异常的动作，所以它抛出的是一个异常实例，执行 throw 一定是抛出了某种异常。

throws:

(1)throws 语句是用在方法声明后面，表示如果抛出异常，由该方法的调用者来进行异常的处理。

(2)throws 主要是声明这个方法会抛出某种类型的异常，让它的使用者要知道需要捕获的异常的类型。

(3)throws 表示出现异常的一种可能性，并不一定会发生这种异常。

## 3.61final、finally、finalize 有什么区别？

1、final

Final可以用于成员变量（包括方法参数），方法、类。

Final成员

 作为变量

变量一旦被初始化便不可改变（对于基本类型，指的是值不变；对于对象类型，指的是引用不变），初始化只可能在两个地方：定义处和构造函数。

作为方法参数

对于基本类型，定义成final参数没有什么意义，因为基本类型就是传值，不会影响调用语句中的变量；对于对象类型，在方法中如果参数确认不需要改变时，定义成final参数可以防止方法中无意的修改而影响到调用方法。

Final方法

不可覆写

编译器将对此方法的调用转化成行内（inline）调用，即直接把方法主体插入到调用处（方法主体内容过多的时候反而会影响效率）

Final类

不可继承

2、finally

异常处理关键字，finally中的主体总会执行，不管异常发生是否。通常放在try…catch的后面构造最终执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。

2.1、当try中有return时执行顺序

return语句并不是函数的最终出口，如果有finally语句，这在return之后还会执行finally（return的值会暂存在栈里面，等待finally执行后再返回）

2.2、return和异常获取语句的位置

3、finalize

类的Finalize方法，可以告诉垃圾回收器应该执行的操作，该方法从Object类继承而来。在从堆中永久删除对象之前，垃圾回收器调用该对象的Finalize方法。注意，无法确切地保证垃圾回收器何时调用该方法，也无法保证调用不同对象的方法的顺序。即使一个对象包含另一个对象的引用，或者在释放一个对象很久以前就释放了另一个对象，也可能会以任意的顺序调用这两个对象的Finalize方法。如果必须保证采用特定的顺序，则必须提供自己的特有清理方法。

## 3.64 Java中CyclicBarrier 和 CountDownLatch有什么不同？

1、CyclicBarrier的某个线程运行到某个点后停止运行，直到所有线程都达到同一个点，所有线程才会重新运行；

 CountDownLatch线程运行到某个点后，计数值-1，该线程继续运行，直到计数值为0，则停止运行；

2、CyclicBarrier只能唤醒一个任务；CountDownLatch可以唤醒多个任务；

3、CyccliBarrier可以重用，CountDownLatch不可重用，当计数值为0时，CountDownLatch就不可再用了。

## 3.65为什么说 Synchronized 是一个悲观锁?乐观锁的实现原理 又是什么?什么是 CAS，它有什么特性?

Synchronized显然是一个悲观锁，因为它的并发策略是悲观的：不管是否会产生竞争，任何的数据操作都必须要加锁、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。

随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略。先进行操作，如果没有其他线程征用数据，那操作就成功了；如果共享数据有征用，产生了冲突，那就再进行其他的补偿措施。这种乐观的并发策略的许多实现不需要线程挂起，所以被称为非阻塞同步。

乐观锁的核心算法是CAS（CompareandSwap，比较并交换），它涉及到三个操作数：内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此块内存。

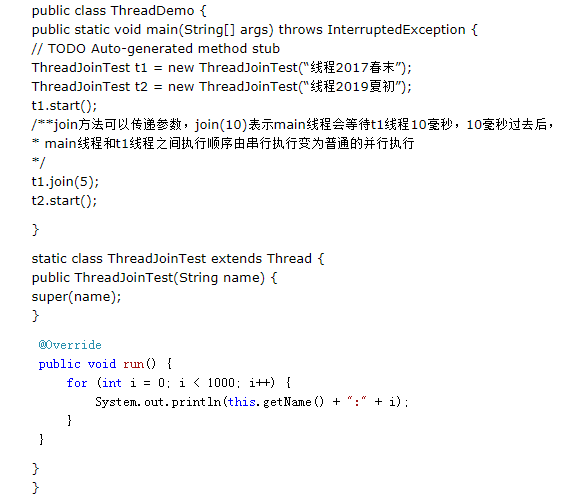
CAS具有原子性，它的原子性由CPU硬件指令实现保证，即使用JNI调用Native方法调用由C++编写的硬件级别指令，JDK中提供了Unsafe类执行这些操作。

## 3.66 Java中Semaphore是什么？

Java中的Semaphore是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用于多线程的代码中，比如数据库连接池。

## 3.67现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。



## 3.68你如何在Java中获取线程堆栈？

Java虚拟机提供了线程转储（thread dump）的后门，通过这个后门可以把线程堆栈打印出来。通常我们将堆栈信息重定向到一个文件中，便于我们分析，由于信息量太大，很可能超出控制台缓冲区的最大行数限制造成信息丢失。这里介绍一个jdk自带的打印线程堆栈的工具，jstack用于打印出给定的Java进程ID或core file或远程调试服务的Java堆栈信息。

示例：$jstack –l 23561 >> xxx.dump

命令 : $jstack [option] pid >> 文件

>>表示输出到文件尾部，实际运行中，往往一次dump的信息，还不足以确认问题，建议产生三次dump信息，如果每次dump都指向同一个问题，我们才确定问题。

## 3.69提交任务时线程池队列已满会时发会生什么？

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。但是最大线程数没有满的话，就会新建一个非核心线程去执行该任务。如果核心线程数、阻塞队列、最大线程数都满了的话，就会执行线程池的拒绝策略，如果一个任务不能被调度执行那么ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常

## 3.70什么是乐观锁和悲观锁？

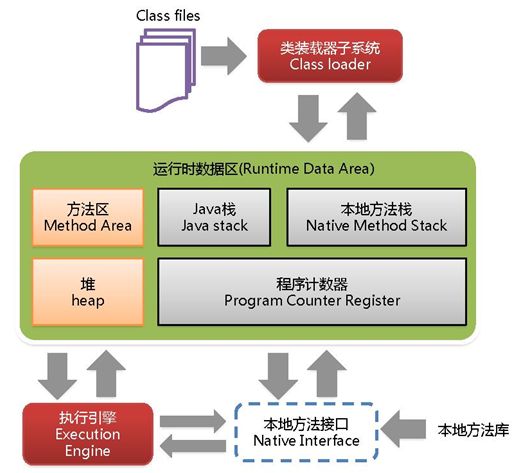
乐观锁总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

悲观锁顾名思义，就是很悲观，总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

# 第4章 JVM

## 4.1 JVM内存分哪几个区，每个区的作用是什么?



java虚拟机主要分为以下几个区:

**（1）方法区**：

1. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
3. 该区域是被线程共享的。
4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

**（2）虚拟机栈**:

1. 虚拟机栈也就是我们平常所称的栈内存,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
2. 虚拟机栈是线程私有的，它的生命周期与线程相同。
3. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
4. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
5. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

**（3）本地方法栈**：  
本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

**（4）堆**：

java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

**（5）程序计数器：**

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

## 4.2 heap 和stack 有什么区别

**（1）申请方式**

stack:由系统自动分配。例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空间

heap:需要程序员自己申请，并指明大小，在 c 中 malloc 函数，对于Java 需要手动 new Object()的形式开辟

**（2）申请后系统的响应**

**stack：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。**

**heap：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。**

**（3）申请大小的限制**

**stack：栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。**

**heap：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的， 自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见， 堆获得的空间比较灵活，也比较大。**

**（4）申请效率的比较**

**stack：由系统自动分配，速度较快。但程序员是无法控制的。**

**heap：由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。**

**（5）heap和stack中的存储内容**

stack：在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址， 然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

heap：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

## 4.3 java类加载过程?

Java类加载需要经历一下几个过程：

**（1）加载**

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。
2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
3. 在内存中生成该类的Class对象，作为该类的数据访问入口。

**（2）验证**

验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

1. 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。
5. 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

**（3）解析**

**该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。**

**（4）初始化**

**初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。**

## 4.4 什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

（1）启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。

（2）扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

（3）系统类加载器（system class loader）也叫应用类加载器：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

（4）用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

## 4.5 java中垃圾收集的方法有哪些?

**1）引用计数法**   应用于：微软的COM/ActionScrip3/Python等

a) 如果对象没有被引用，就会被回收，缺点：需要维护一个引用计算器

**2）复制算法** 年轻代中使用的是Minor GC，这种GC算法采用的是复制算法(Copying)

a) 效率高，缺点：需要内存容量大，比较耗内存

b) 使用在占空间比较小、刷新次数多的新生区

**3）标记清除** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率比较低，会差生碎片。

**4）标记压缩** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率低速度慢，需要移动对象，但不会产生碎片。

**5）标记清除**压缩标记清除-标记压缩的集合，多次GC后才Compact

a) 使用于占空间大刷新次数少的养老区，是3 4的集合体

## 4.6 如何判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:

**（1）引用计数法**

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A,B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

**（2）可达性算法(引用链法)**

该算法的基本思路就是通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。

在java中可以作为GC Roots的对象有以下几种：虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量池引用的对象、本地方法栈JNI引用的对象。

## 4.7 简述java内存分配与回收策略以及Minor GC和Major GC（full GC）

**内存分配：**

**（1）栈区**：栈分为java虚拟机栈和本地方法栈

**（2）堆区**：堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区，主要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1。

**（3）方法区**：被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）

**（4）程序计数器**：当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。线程私有的。

**回收策略以及Minor GC和Major GC：**

（1）对象优先在堆的Eden区分配。

（2）大对象直接进入老年代。

（3）长期存活的对象将直接进入老年代。

当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor GC通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生GC的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

## 4.8什么情况下会发生栈内存溢出。

栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常，方法递归调用产生这种结果。

如果Java虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么Java虚拟机将抛出一个OutOfMemory 异常。(线程启动过多)

参数 -Xss 去调整JVM栈的大小

## 4.9 JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代

对象诞生即新生代->eden，在进行minor gc过程中，如果依旧存活，移动到from，变成Survivor，进行标记。当一个对象存活默认超过15次都没有被回收掉，就会进入老年代。

## 4.10你知道哪几种垃圾收集器，各自的优缺点，重点讲下cms和G1，包括原理，流程，优缺点。

垃圾收集器包括Serial、parNew、ParallelScavenge、SerialOld、ParallelOld、CMS、G1

CMS:

一、初始标记：此时标记需要用户线程停下来；

二、并发标记：此时标记可以和用户线程一起运行；

三、重新标记：此时标记需要用户线程停下来，主要母的是为了对并发标记的垃圾进行审核；

四、并发清除：与用户线程一起与运行进行垃圾清除；

缺点：

1、CMS收集器对cpu资源非常敏感；

2、CMS收集器无法清除浮动垃圾；

3、cms基于标记清除的算法实现的，所以内存碎片会产生过多。

G1收集器：

1、初始标记：标记GC Root能直接关联的对象，并且修改TAMS的值，让下一阶段的用户进行并发运行是，能够正确运用Region创建新对象，这阶段需要停顿，但停顿时间很短

2、并发标记：从GC Root开始对堆进行可达性分析，找出存活的对象，这段耗时较长，但可以与用户线程并发执行。

3、最终标记是为了修正在并发标记阶段因用户程序继续运作导致标记产生变动的那一部分的标记记录，虚拟机将这部分标记记录在线程Remembered Set中，这阶段需要停顿线程，但是可并行执行。

4、筛选回收：首先对各个Region的回收价值和成本进行排序，根据用户所期待的GC停顿时间来制定回收计划，这个阶段也可以与用户线程并行执行，但由于只回收一部分的Region,时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

## 4.11 JVM内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存。

重排序：jvm虚拟机允许在不影响代码最终结果的情况下，可以乱序执行。

内存屏障：可以阻挡编译器的优化，也可以阻挡处理器的优化

happens-before原则：

1：一个线程的A操作总是在B之前，那多线程的A操作肯定实在B之前。

2：monitor 再加锁的情况下，持有锁的肯定先执行。

3：volatile修饰的情况下，写先于读发生

4：线程启动在一起之前 strat

5：线程死亡在一切之后 end

6：线程操作在一切线程中断之前

7：一个对象构造函数的结束都该对象的finalizer的开始之前

8：传递性，如果A肯定在B之前，B肯定在C之前，那A肯定是在C之前。

主内存：所有线程共享的内存空间

工作内存：每个线程特有的内存空间

## 4.12简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。

1) 什么是类加载器？

类加载器 就是根据指定全限定名称将class文件加载到JVM内存，转为Class对象。

启动类加载器（Bootstrap ClassLoader）：由C++语言实现（针对HotSpot）,负责将存放在<JAVA\_HOME>\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中。

其他类加载器：由Java语言实现，继承自抽象类ClassLoader。如：

扩展类加载器（Extension ClassLoader）：负责加载<JAVA\_HOME>\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库。

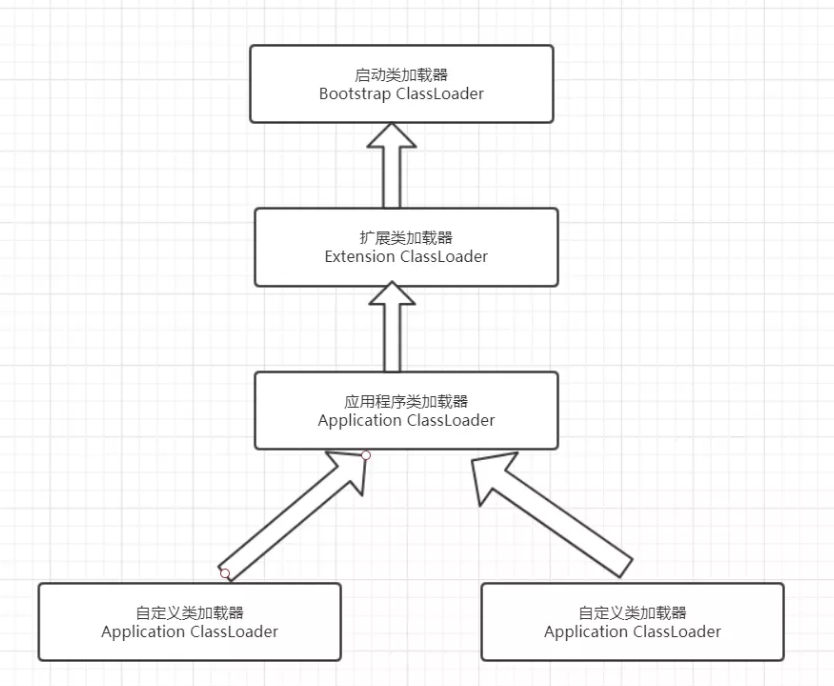
应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

2）双亲委派模型

双亲委派模型工作过程是：

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

双亲委派模型图：



3）为什么需要双亲委派模型？

在这里，先想一下，如果没有双亲委派，那么用户是不是可以自己定义一个java.lang.Object的同名类，java.lang.String的同名类，并把它放到ClassPath中,那么类之间的比较结果及类的唯一性将无法保证，因此，为什么需要双亲委派模型？防止内存中出现多份同样的字节码

4）怎么打破双亲委派模型？

打破双亲委派机制则不仅要继承ClassLoader类，还要重写loadClass和findClass方法。

## 4.13说说你知道的几种主要的JVM参数

1）堆栈配置相关

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-XX:MaxPermSize=16m -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxTenuringThreshold=0

-Xmx3550m： 最大堆大小为3550m。

-Xms3550m： 设置初始堆大小为3550m。

-Xmn2g： 设置年轻代大小为2g。

-Xss128k： 每个线程的堆栈大小为128k。

-XX:MaxPermSize： 设置持久代大小为16m

-XX:NewRatio=4: 设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。

-XX:SurvivorRatio=4： 设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxTenuringThreshold=0： 设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。

2）垃圾收集器相关

-XX:+UseParallelGC

-XX:ParallelGCThreads=20

-XX:+UseConcMarkSweepGC

-XX:CMSFullGCsBeforeCompaction=5

-XX:+UseCMSCompactAtFullCollection：

-XX:+UseParallelGC： 选择垃圾收集器为并行收集器。

-XX:ParallelGCThreads=20： 配置并行收集器的线程数

-XX:+UseConcMarkSweepGC： 设置年老代为并发收集。

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection： 打开对年老代的压缩。可能会影响性能，但是可以消除碎片

3）辅助信息相关

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGC 输出形式:

[GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]

-XX:+PrintGCDetails 输出形式:

[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs

## 4.15垃圾收集算法

jvm的垃圾回收算法有3种，列举如下：

1，标记清除算法。（即把标注的可回收对象直接清理，这样会带来内存碎片化的问题，而且效率不高）；

2，标记整理算法。（即把标注的可回收对象清理，在清理的过程中整理内存，解决了内存的碎片化问题）；

3，标记复制算法。（把标注的对象清理，没有清理的对象复制到to区，然后互换引用，解决了内存碎片化的问题，但是需要维护对象关系带来一定代价）

## 4.16调优工具用过哪些

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗

GChisto，一款专业分析gc日志的工具

## 4.17你知道哪些JVM性能调优

首先需要注意的是在对JVM内存调优的时候不能只看操作系统级别Java进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为GC过后这个值是不会变化的，因此内存调优的时候要更多地使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。

  对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

旧生代空间不足

  调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象

Pemanet Generation空间不足

  增大Perm Gen空间，避免太多静态对象

  统计得到的GC后晋升到旧生代的平均大小大于旧生代剩余空间

  控制好新生代和旧生代的比例

System.gc()被显示调用

  垃圾回收不要手动触发，尽量依靠JVM自身的机制

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，下面来看看各部分比例不良设置会导致什么后果

1). 新生代设置过小

  一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2). 新生代设置过大

  一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加

  一般说来新生代占整个堆1/3比较合适

3). Survivor设置过小

  导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4). Survivor设置过大

  导致eden过小，增加了GC频率

  另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

由内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM提供两种较为简单的GC策略的设置方式

1). 吞吐量优先

  JVM以吞吐量为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n来设置

2). 暂停时间优先

  JVM以暂停时间为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX:MaxGCPauseRatio=n来设置

## 4.18 Eden和Survivor的比例分配等

默认比例8:1。部分对象都是朝生夕死。 复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

## 4.19说一说你对环境变量classpath的理解？如果一个类不在classpath下，为什么会抛出ClassNotFoundException异常，如果在不改变这个类路径的前期下，怎样才能正确加载这个类？

classpath是javac编译器的一个环境变量。它的作用与import、package关键字有关。package的所在位置，就是设置CLASSPATH当编译器面对import packag这个语句时，它先会查找CLASSPATH所指定的目录，并检视子目录java/util是否存在，然后找出名称吻合的已编译文件（.class文件）。如果没有找到就会报错！

动态加载包

# 设计模式

## 5.1你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

## 5.2单例设计模式

见【第2章 手写代码 2.5】

## 5.3工厂设计模式（Factory）

### 5.3.1什么是工厂设计模式？

工厂设计模式，顾名思义，就是用来生产对象的，在java中，万物皆对象，这些对象都需要创建，如果创建的时候直接new该对象，就会对该对象耦合严重，假如我们要更换对象，所有new对象的地方都需要修改一遍，这显然违背了软件设计的开闭原则，如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：解耦

### 5.3.2简单工厂（Simple Factory）

**定义：**

一个工厂方法，依据传入的参数，生成对应的产品对象；  
**角色：**  
1、抽象产品  
2、具体产品  
3、具体工厂  
4、产品使用者  
**使用说明：**

先将产品类抽象出来，比如，苹果和梨都属于水果，抽象出来一个水果类Fruit，苹果和梨就是具体的产品类，然后创建一个水果工厂，分别用来创建苹果和梨。代码如下：

**水果接口：**

1. **public** interface Fruit {
2. **void** whatIm();
3. }

**苹果类：**

1. **public** **class** Apple **implements** Fruit {
2. @Override
3. **public** **void** whatIm() {
4. System.out.println("苹果");
5. }
6. }

**梨类：**

1. **public** **class** Pear **implements** Fruit {
2. @Override
3. **public** **void** whatIm() {
4. System.out.println("梨");
5. }
6. }

**水果工厂：**

1. **public** **class** FruitFactory {
3. **public** Fruit createFruit(String type) {
5. **if** (type.equals("apple")) {//生产苹果
6. **return** **new** Apple();
7. } **else** **if** (type.equals("pear")) {//生产梨
8. **return** **new** Pear();
9. }
11. **return** null;
12. }
13. }

**使用工厂生产产品：**

1. **public** **class** FruitApp {
3. **public** **static** **void** main(String[] args) {
4. FruitFactory mFactory = **new** FruitFactory();
5. Apple apple = (Apple) mFactory.createFruit("apple");//获得苹果
6. Pear pear = (Pear) mFactory.createFruit("pear");//获得梨
7. apple.whatIm();
8. pear.whatIm();
9. }
10. }

以上的这种方式，每当添加一种水果，就必然要修改工厂类，违反了开闭原则；

所以简单工厂只适合于产品对象较少，且产品固定的需求，对于产品变化无常的需求来说显然不合适。

### 5.3.3工厂方法（Factory Method）

**定义：**

将工厂提取成一个接口或抽象类，具体生产什么产品由子类决定；  
**角色：**  
1、抽象产品  
2、具体产品  
3、抽象工厂  
4、具体工厂  
**使用说明：**

和上例中一样，产品类抽象出来，这次我们把工厂类也抽象出来，生产什么样的产品由子类来决定。代码如下：  
**水果接口、苹果类和梨类：**

代码和上例一样

**抽象工厂接口：**

1. **public** interface FruitFactory {
2. Fruit createFruit();//生产水果
3. }

**苹果工厂：**

1. **public** **class** AppleFactory implements FruitFactory {
2. @Override
3. **public** Apple createFruit() {
4. **return** **new** Apple();
5. }
6. }

**梨工厂：**

1. **public** **class** PearFactory implements FruitFactory {
2. @Override
3. **public** Pear createFruit() {
4. **return** **new** Pear();
5. }
6. }

**使用工厂生产产品：**

1. **public** **class** FruitApp {
3. **public** **static** **void** main(String[] args){
4. AppleFactory appleFactory = **new** AppleFactory();
5. PearFactory pearFactory = **new** PearFactory();
6. Apple apple = appleFactory.createFruit();//获得苹果
7. Pear pear = pearFactory.createFruit();//获得梨
8. apple.whatIm();
9. pear.whatIm();
10. }
11. }

以上这种方式，虽然解耦了，也遵循了开闭原则，但是如果我需要的产品很多的话，需要创建非常多的工厂，所以这种方式的缺点也很明显。

### 5.3.4抽象工厂（Abstract Factory）

**定义：**

为创建一组相关或者是相互依赖的对象提供的一个接口，而不需要指定它们的具体类。  
**角色：**

**1、抽象产品**

**2、具体产品**

**3、抽象工厂**

**4、具体工厂**

**使用说明：**

抽象工厂和工厂方法的模式基本一样，区别在于，工厂方法是生产一个具体的产品，而抽象工厂可以用来生产一组相同，有相对关系的产品；重点在于一组，一批，一系列；举个例子，假如生产小米手机，小米手机有很多系列，小米note、红米note等；假如小米note生产需要的配件有825的处理器，6英寸屏幕，而红米只需要650的处理器和5寸的屏幕就可以了。用抽象工厂来实现：

**cpu接口和实现类：**

1. **public** interface Cpu {
2. **void** run();
4. **class** Cpu650 implements Cpu {
5. @Override
6. **public** **void** run() {
7. System.out.println("650 也厉害");
8. }
9. }
11. **class** Cpu825 implements Cpu {
12. @Override
13. **public** **void** run() {
14. System.out.println("825 更强劲");
15. }
16. }
17. }

**屏幕接口和实现类：**

1. **public** interface Screen {
3. **void** size();
5. **class** Screen5 implements Screen {
7. @Override
8. **public** **void** size() {
9. System.out.println("" +
10. "5寸");
11. }
12. }
14. **class** Screen6 implements Screen {
16. @Override
17. **public** **void** size() {
18. System.out.println("6寸");
19. }
20. }
21. }

**抽象工厂接口：**

1. **public** interface PhoneFactory {
3. Cpu getCpu();//使用的cpu
5. Screen getScreen();//使用的屏幕
6. }

**小米手机工厂：**

1. **public** **class** XiaoMiFactory implements PhoneFactory {
2. @Override
3. **public** Cpu.Cpu825 getCpu() {
4. **return** **new** Cpu.Cpu825();//高性能处理器
5. }
7. @Override
8. **public** Screen.Screen6 getScreen() {
9. **return** **new** Screen.Screen6();//6寸大屏
10. }
11. }

**红米手机工厂：**

1. **public** **class** HongMiFactory implements PhoneFactory {
3. @Override
4. **public** Cpu.Cpu650 getCpu() {
5. **return** **new** Cpu.Cpu650();//高效处理器
6. }
8. @Override
9. **public** Screen.Screen5 getScreen() {
10. **return** **new** Screen.Screen5();//小屏手机
11. }
12. }

**使用工厂生产产品：**

1. **public** **class** PhoneApp {
2. **public** **static** **void** main(String[] args){
3. HongMiFactory hongMiFactory = **new** HongMiFactory();
4. XiaoMiFactory xiaoMiFactory = **new** XiaoMiFactory();
5. Cpu.Cpu650 cpu650 = hongMiFactory.getCpu();
6. Cpu.Cpu825 cpu825 = xiaoMiFactory.getCpu();
7. cpu650.run();
8. cpu825.run();
10. Screen.Screen5 screen5 = hongMiFactory.getScreen();
11. Screen.Screen6 screen6 = xiaoMiFactory.getScreen();
12. screen5.size();
13. screen6.size();
14. }
15. }

以上例子可以看出，抽象工厂可以解决一系列的产品生产的需求，对于大批量，多系列的产品，用抽象工厂可以更好的管理和扩展。

### 5.3.5三种工厂方式总结

1、对于简单工厂和工厂方法来说，两者的使用方式实际上是一样的，如果对于产品的分类和名称是确定的，数量是相对固定的，推荐使用简单工厂模式；

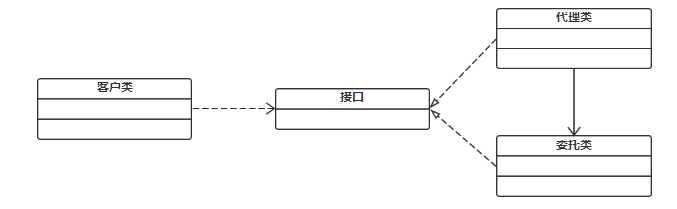
2、抽象工厂用来解决相对复杂的问题，适用于一系列、大批量的对象生产。

## 5.4代理模式（Proxy）

### 5.4.1什么是代理模式？

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

举个例子来说明：假如说我现在想买一辆二手车，虽然我可以自己去找车源，做质量检测等一系列的车辆过户流程，但是这确实太浪费我得时间和精力了。我只是想买一辆车而已为什么我还要额外做这么多事呢？于是我就通过中介公司来买车，他们来给我找车源，帮我办理车辆过户流程，我只是负责选择自己喜欢的车，然后付钱就可以了。用图表示如下：



### 5.4.2为什么要用代理模式？

**中介隔离作用：**

在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

**开闭原则，增加功能：**

代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要修改已经封装好的委托类。

### 5.4.3有哪几种代理模式？

我们有多种不同的方式来实现代理。

如果按照代理创建的时期来进行分类的话，可以分为两种：静态代理、动态代理。

* 静态代理是由程序员创建或特定工具自动生成源代码，再对其编译。在程序员运行之前，代理类.class文件就已经被创建了。
* 动态代理是在程序运行时通过反射机制动态创建的。

### 5.4.4静态代理（Static Proxy）

**第一步：创建服务类接口**

1. **public** interface BuyHouse {
2. **void** buyHouse();
3. }

**第二步：实现服务接口**

1. **public** **class** BuyHouseImpl implements BuyHouse {
3. @Override
4. **public** **void** buyHouse() {
5. System.out.println("我要买房");
6. }
7. }

**第三步：创建代理类**

1. **public** **class** BuyHouseProxy implements BuyHouse {
3. **private** BuyHouse buyHouse;
5. **public** BuyHouseProxy(final BuyHouse buyHouse) {
6. **this**.buyHouse = buyHouse;
7. }
9. @Override
10. **public** **void** buyHouse() {
11. System.out.println("买房前准备");
12. buyHouse.buyHouse();
13. System.out.println("买房后装修");
15. }
16. }

**第四步：编写测试类**

1. **public** **class** HouseApp {
3. **public** **static** **void** main(String[] args) {
4. BuyHouse buyHouse = **new** BuyHouseImpl();
5. BuyHouseProxy buyHouseProxy = **new** BuyHouseProxy(buyHouse);
6. buyHouseProxy.buyHouse();
7. }
8. }

**静态代理总结：**

优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。

缺点：我们得为每一个服务创建代理类，工作量太大，不易管理。同时接口一旦发生改变，代理类也得相应修改。

### 5.4.5 JDK动态代理（Dynamic Proxy）

在动态代理中我们不再需要再手动的创建代理类，我们只需要编写一个动态处理器就可以了。真正的代理对象由JDK在运行时为我们动态的来创建。

**第一步：创建服务类接口**

代码和上例一样

**第二步：实现服务接口**

代码和上例一样

**第三步：编写动态处理器**

1. **public** **class** DynamicProxyHandler implements InvocationHandler {
3. **private** Object object;
5. **public** DynamicProxyHandler(final Object object) {
6. **this**.object = object;
7. }
9. @Override
10. **public** Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
11. System.out.println("买房前准备");
12. Object result = method.invoke(object, args);
13. System.out.println("买房后装修");
14. **return** result;
15. }
16. }

**第四步：编写测试类**

1. **public** **class** HouseApp {
3. **public** **static** **void** main(String[] args) {
4. BuyHouse buyHouse = **new** BuyHouseImpl();
5. BuyHouse proxyBuyHouse = (BuyHouse) Proxy.newProxyInstance(
6. BuyHouse.**class**.getClassLoader(),
7. **new** Class[]{BuyHouse.**class**},
8. **new** DynamicProxyHandler(buyHouse));
9. proxyBuyHouse.buyHouse();
10. }
11. }

 Proxy是所有动态生成的代理的共同的父类，这个类有一个静态方法Proxy.newProxyInstance()，接收三个参数：

* ClassLoader loader：指定当前目标对象使用的类加载器,获取加载器的方法是固定的
* Class<?>[] interfaces：指定目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler：指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法

JDK动态代理总结：

优点：相对于静态代理，动态代理大大减少了开发任务，同时减少了对业务接口的依赖，降低了耦合度。

缺点：Proxy是所有动态生成的代理的共同的父类，因此服务类必须是接口的形式，不能是普通类的形式，因为Java无法实现多继承。

### 5.4.6 CGLib动态代理（CGLib Proxy）

JDK实现动态代理需要实现类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是继承，所以不能对final修饰的类进行代理。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

**Cglib子类代理实现方法：**

（1）引入cglib的jar文件，asm的jar文件

（2）代理的类不能为final

（3）目标业务对象的方法如果为final/static，那么就不会被拦截，即不会执行目标对象额外的业务方法

**第一步：创建服务类**

1. **public** **class** BuyHouse2 {
3. **public** **void** buyHouse() {
4. System.out.println("我要买房");
5. }
6. }

**第二步：创建CGLIB代理类**

1. **public** **class** CglibProxy implements MethodInterceptor {
3. **private** Object target;
5. **public** CglibProxy(Object target) {
6. **this**.target = target;
7. }
9. /\*\*
10. \*  给目标对象创建一个代理对象
11. \* @return 代理对象
12. \*/
13. **public** Object getProxyInstance() {
14. //1.工具类
15. Enhancer enhancer = **new** Enhancer();
16. //2.设置父类
17. enhancer.setSuperclass(target.getClass());
18. //3.设置回调函数
19. enhancer.setCallback(**this**);
20. //4.创建子类(代理对象)
21. **return** enhancer.create();
23. }
25. **public** Object intercept(Object object, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
26. System.out.println("买房前准备");
27. //执行目标对象的方法
28. Object result = method.invoke(target, args);
29. System.out.println("买房后装修");
30. **return** result;
31. }
32. }

**第三步：创建测试类**

1. **public** **class** HouseApp {
3. **public** **static** **void** main(String[] args) {
5. BuyHouse2 target = **new** BuyHouse2();
6. CglibProxy cglibProxy = **new** CglibProxy(target);
7. BuyHouse2 buyHouseCglibProxy = (BuyHouse2) cglibProxy.getProxyInstance();
8. buyHouseCglibProxy.buyHouse();
9. }
10. }

**CGLib代理总结：**

CGLib创建的动态代理对象比JDK创建的动态代理对象的性能更高，但是CGLIB创建代理对象时所花费的时间却比JDK多得多。所以对于单例的对象，因为无需频繁创建对象，用CGLIB合适，反之使用JDK方式要更为合适一些。同时由于CGLib由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。

### 5.4.7 简述动态代理的原理， 常用的动态代理的实现方式

动态代理的原理: **使用一个代理将对象包装起来**，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。

代理对象决定是否以及何时将方法调 用转到原始对象上

动态代理的方式

基于接口实现动态代理： JDK动态代理

基于继承实现动态代理： Cglib、Javassist动态代理

# 第6章 MySql

## 6.1 jdbc 操作数据库流程

第一步：Class.forName()加载数据库连接驱动；

第二步：DriverManager.getConnection()获取数据连接对象;

第三步：根据SQL 获取 sql 会话对象，有 2 种方式 Statement、PreparedStatement ;

第四步：执行SQL 处理结果集，执行 SQL 前如果有参数值就设置参数值 setXXX();

第五步：关闭结果集、关闭会话、关闭连接。

## 6.2关系数据库中连接池的机制是什么？

前提：为数据库连接建立一个缓冲池。

（1）从连接池获取或创建可用连接

（2）使用完毕之后，把连接返回给连接池

（3）在系统关闭前，断开所有连接并释放连接占用的系统资源

（4）能够处理无效连接，限制连接池中的连接总数不低于或者不超过某个限定值。

其中有几个概念需要大家理解：

最小连接数是连接池一直保持的数据连接。如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费掉。

最大连接数是连接池能申请的最大连接数。如果数据连接请求超过此数，后面的数据连接请求将被加入到等待队列中，这会影响之后的数据库操作。

如果最小连接数与最大连接数相差太大，那么，最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，它将被放到连接池中等待重复使用或是空闲超时后被释放。

上面的解释，可以这样理解：数据库池连接数量一直保持一个不少于最小连接数的数量，当数量不够时，数据库会创建一些连接，直到一个最大连接数，之后连接数据库就会等待。

## 6.3 SQL 的select 语句完整的执行顺序

SQL Select 语句完整的执行顺序：

（1）from 子句组装来自不同数据源的数据；

（2）where 子句基于指定的条件对记录行进行筛选；

（3）group by 子句将数据划分为多个分组；

（4）使用聚集函数进行计算；

（5）使用 having 子句筛选分组；

（6）计算所有的表达式；

（7）select 的字段；

（8）使用order by 对结果集进行排序。

## 6.3 MySQL的事务

**事务的基本要素（ACID）：**

* 原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位
* 一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如A向B转账，不可能A扣了钱，B却没收到。
* 隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。
* 持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

**事务的并发问题：**

* 脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
* 不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致
* 幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

**小结：**

不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

**MySQL事务隔离级别：**

事务隔离级别 脏读 不可重复读 幻读

读未提交（read-uncommitted） 是 是 是

不可重复读（read-committed） 否 是 是

可重复读（repeatable-read） 否 否 是

串行化（serializable） 否 否 否

## 6.4行锁，表锁

|  |  |  |
| --- | --- | --- |
|  | MyISAM | InnoDB |
| 行表锁 | 表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作 | 行锁,操作时只锁某一行，不对其它行有影响，适合高并发的操作 |

## 6.5索引

**数据结构：B+Tree**

一般来说能够达到range就可以算是优化了

**口诀：**

全值匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

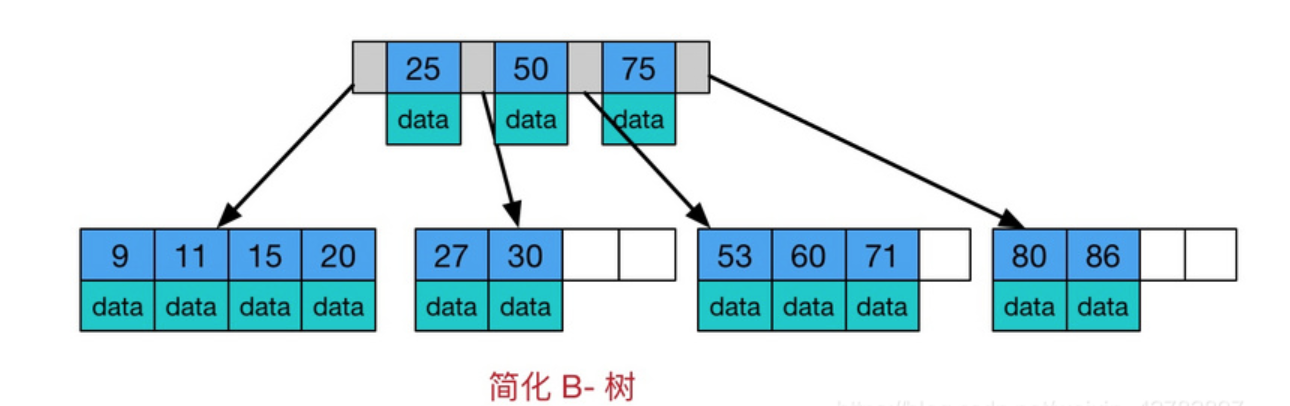
LIKE百分写最右，覆盖索引不写\*；

不等空值还有OR，索引影响要注意；

VAR引号不可丢，SQL优化有诀窍。

## 6.6 b-tree和b+tree的区别

b-tree我们称之为B树。B树是一种多路自平衡搜索树，它类似普通的二叉树，但是B书允许每个节点有更多的子节点。B树示意图如下：



B树的特点：

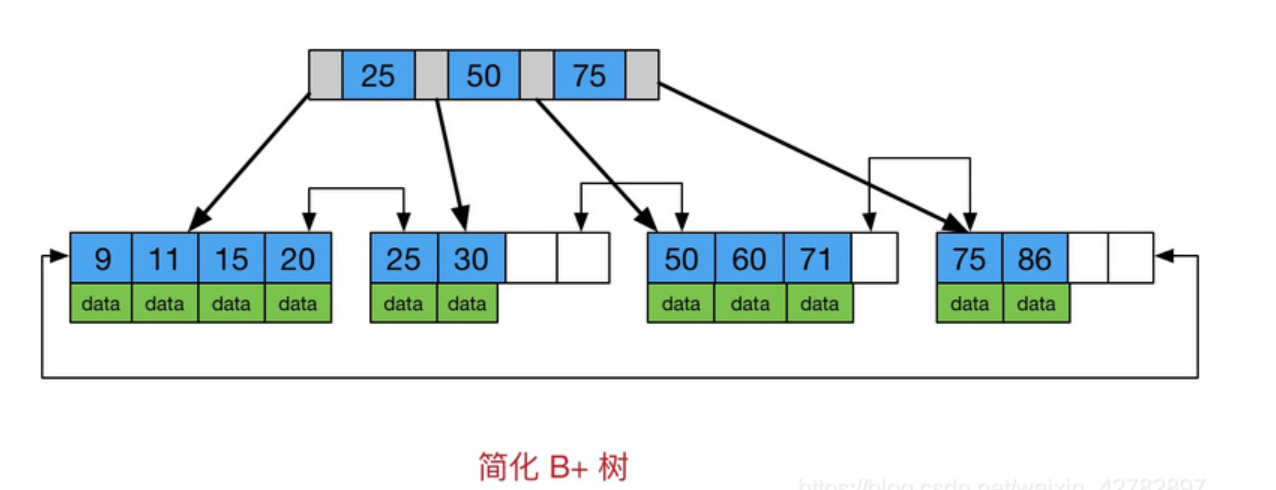
（1）所有键值分布在整个树中

（2）任何关键字出现且只出现在一个节点中

（3）搜索有可能在非叶子节点结束

（4）在关键字全集内做一次查找，性能逼近二分查找算法

B+Tree



从图中也可以看到，B+树与B树的不同在于：

（1）非叶子节点只存储键值信息

（2）所有叶子节点之间都有一个链指针

（3）数据记录都存放在叶子节点中

## 6.7简述在MySQL数据库中MyISAM和InnoDB的区别

InnoDB存储引擎

主要面向OLTP(Online Transaction Processing，在线事务处理)方面的应用

**特点：**

行锁设计、支持外键；

MyISAM存储引擎

主要面向OLAP(Online Analytical Processing,在线分析处理)方面的应用。

**特点：**

不支持事务，支持表所和全文索引。操作速度快。

## 6.8你们公司有哪些数据库设计规范

（一）基础规范

1、表存储引擎必须使用InnoD，表字符集默认使用utf8，必要时候使用utf8mb4

解读：

（1）通用，无乱码风险，汉字3字节，英文1字节

（2）utf8mb4是utf8的超集，有存储4字节例如表情符号时，使用它

2、禁止使用存储过程，视图，触发器，Event

解读：

（1）对数据库性能影响较大，互联网业务，能让站点层和服务层干的事情，不要交到数据库层

（2）调试，排错，迁移都比较困难，扩展性较差

3、禁止在数据库中存储大文件，例如照片，可以将大文件存储在对象存储系统，数据库中存储路径

4、禁止在线上环境做数据库压力测试

5、测试，开发，线上数据库环境必须隔离

（二）命名规范

1、库名，表名，列名必须用小写，采用下划线分隔

解读：abc，Abc，ABC都是给自己埋坑

2、库名，表名，列名必须见名知义，长度不要超过32字符

解读：tmp，wushan谁知道这些库是干嘛的

3、库备份必须以bak为前缀，以日期为后缀

4、从库必须以-s为后缀

5、备库必须以-ss为后缀

（三）表设计规范

1、单实例表个数必须控制在2000个以内

2、单表分表个数必须控制在1024个以内

3、表必须有主键，推荐使用UNSIGNED整数为主键

潜在坑：删除无主键的表，如果是row模式的主从架构，从库会挂住

4、禁止使用外键，如果要保证完整性，应由应用程式实现

解读：外键使得表之间相互耦合，影响update/delete等SQL性能，有可能造成死锁，高并发情况下容易成为数据库瓶颈

5、建议将大字段，访问频度低的字段拆分到单独的表中存储，分离冷热数据

（四）列设计规范

1、根据业务区分使用tinyint/int/bigint，分别会占用1/4/8字节

2、根据业务区分使用char/varchar

解读：

（1）字段长度固定，或者长度近似的业务场景，适合使用char，能够减少碎片，查询性能高

（2）字段长度相差较大，或者更新较少的业务场景，适合使用varchar，能够减少空间

3、根据业务区分使用datetime/timestamp

解读：前者占用5个字节，后者占用4个字节，存储年使用YEAR，存储日期使用DATE，存储时间使用datetime

4、必须把字段定义为NOT NULL并设默认值

解读：

（1）NULL的列使用索引，索引统计，值都更加复杂，MySQL更难优化

（2）NULL需要更多的存储空间

（3）NULL只能采用IS NULL或者IS NOT NULL，而在=/!=/in/not in时有大坑

5、使用INT UNSIGNED存储IPv4，不要用char(15)

6、使用varchar(20)存储手机号，不要使用整数

解读：

（1）牵扯到国家代号，可能出现+/-/()等字符，例如+86

（2）手机号不会用来做数学运算

（3）varchar可以模糊查询，例如like ‘138%’

7、使用TINYINT来代替ENUM

解读：ENUM增加新值要进行DDL操作

（五）索引规范

1、唯一索引使用uniq\_[字段名]来命名

2、非唯一索引使用idx\_[字段名]来命名

3、单张表索引数量建议控制在5个以内

解读：

（1）互联网高并发业务，太多索引会影响写性能

（2）生成执行计划时，如果索引太多，会降低性能，并可能导致MySQL选择不到最优索引

（3）异常复杂的查询需求，可以选择ES等更为适合的方式存储

4、组合索引字段数不建议超过5个

解读：如果5个字段还不能极大缩小row范围，八成是设计有问题

5、不建议在频繁更新的字段上建立索引

6、非必要不要进行JOIN查询，如果要进行JOIN查询，被JOIN的字段必须类型相同，并建立索引

解读：踩过因为JOIN字段类型不一致，而导致全表扫描的坑么？

7、理解组合索引最左前缀原则，避免重复建设索引，如果建立了(a,b,c)，相当于建立了(a), (a,b), (a,b,c)

（六）SQL规范

1、禁止使用select \*，只获取必要字段

解读：

（1）select \*会增加cpu/io/内存/带宽的消耗

（2）指定字段能有效利用索引覆盖

（3）指定字段查询，在表结构变更时，能保证对应用程序无影响

2、insert必须指定字段，禁止使用insert into T values()

解读：指定字段插入，在表结构变更时，能保证对应用程序无影响

3、隐式类型转换会使索引失效，导致全表扫描

4、禁止在where条件列使用函数或者表达式

解读：导致不能命中索引，全表扫描

5、禁止负向查询以及%开头的模糊查询

解读：导致不能命中索引，全表扫描

6、禁止大表JOIN和子查询

7、同一个字段上的OR必须改写问IN，IN的值必须少于50个

8、应用程序必须捕获SQL异常

解读：方便定位线上问题

说明：本规范适用于并发量大，数据量大的典型互联网业务，可直接参考。

## 6.9 MySQL性能优化

（1）尽量选择较小的列

（2）将where中用的比较频繁的字段建立索引

（3）select子句中避免使用‘\*’

（4）避免在索引列上使用计算、not in 和<>等操作

（5）当只需要一行数据的时候使用limit 1

（6）保证单表数据不超过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。

（7）避免改变索引列的类型。

（8）选择最有效的表名顺序，from字句中写在最后的表是基础表，将被最先处理，在from子句中包含多个表的情况下，你必须选择记录条数最少的表作为基础表。

（9）避免在索引列上面进行计算。

（10）尽量缩小子查询的结果

## 6.10 SQL 语句优化案例

**例1：where 子句中可以对字段进行 null 值判断吗？**

可以，比如 select id from t where num is null 这样的 sql 也是可以的。但是最好不要给数据库留NULL，尽可能的使用 NOT NULL 填充数据库。不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段，null 不占用空间。可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：select id from t where num= 0。

**例2：如何优化?下面的语句？**

select \* from admin left join log on admin.admin\_id = log.admin\_id where log.admin\_id>10

优化为：select \* from (select \* from admin where admin\_id>10) T1 lef join log on T1.admin\_id = log.admin\_id。

使用 JOIN 时候，应该用小的结果驱动大的结果（left join 左边表结果尽量小如果有条件应该放到左边先处理， right join 同理反向），同时尽量把牵涉到多表联合的查询拆分多个 query（多个连表查询效率低，容易到之后锁表和阻塞）。

**例3：limit 的基数比较大时使用 between**

例如：select \* from admin order by admin\_id limit 100000,10

优化为：select \* from admin where admin\_id between 100000 and 100010 order by admin\_id。

**例4：尽量避免在列上做运算，这样导致索引失效**

例如：select \* from admin where year(admin\_time)>2014

优化为： select \* from admin where admin\_time> '2014-01-01′

## 6.11 常见面试sql

**例1：**

用一条SQL语句查询出每门课都大于80分的学生姓名

name kecheng fenshu  
张三 语文 81  
张三 数学     75  
李四 语文     76  
李四 数学    90  
王五 语文     81  
王五 数学     100  
王五 英语     90  
**答1：**

**select** **distinct** **name** **from** **table** **where** **name** not in (**select** **distinct** **name** **from** **table** **where** fenshu<=80)    
**答2：**

**select** **name** **from** **table** **group** **by** **name** **having** **min**(fenshu)>80

**例2：**

学生表 如下:  
自动编号 学号 姓名 课程编号 课程名称 分数  
1      2005001  张三  0001    数学   69  
2      2005002  李四  0001    数学   89  
3      2005001  张三   0001    数学   69  
删除除了自动编号不同，其他都相同的学生冗余信息  
**答：**

**delete** tablename **where** 自动编号 not in(**select** **min**(自动编号) **from** tablename **group** **by**学号, 姓名, 课程编号, 课程名称, 分数)

**例3：**

一个叫team的表，里面只有一个字段name,一共有4条纪录，分别是a,b,c,d,对应四个球队，现在四个球队进行比赛，用一条sql语句显示所有可能的比赛组合.

**答：**

1. **select** a.**name**, b.**name**
2. **from** team a, team b
3. **where** a.**name** < b.**name**

**例4：**

怎么把这样一个表  
year month amount  
1991   1     1.1  
1991   2     1.2  
1991   3     1.3  
1991   4     1.4  
1992   1     2.1  
1992   2     2.2  
1992   3     2.3  
1992   4     2.4  
查成这样一个结果  
year m1 m2 m3 m4  
1991 1.1 1.2 1.3 1.4  
1992 2.1 2.2 2.3 2.4   
**答：**

1. **select** year,
2. (**select** amount **from** aaa m **where** month=1 and m.year=aaa.year) **as** m1,
3. (**select** amount **from** aaa m **where** month=2 and m.year=aaa.year) **as** m2,
4. (**select** amount **from** aaa m **where** month=3 and m.year=aaa.year) **as** m3,
5. (**select** amount **from**  aaa m **where** month=4 and m.year=aaa.year) **as** m4
6. **from** aaa **group** **by** year

**例5：**

说明：复制表(只复制结构,源表名：a新表名：b)

**答：**  
SQL:

**select** \* **into** b **from** a **where** 1<>1 (where1=1，拷贝表结构和数据内容)

ORACLE:

1. **create** **table** b
2. **As**
3. **Select** \* **from** a **where** 1=2

[<>（不等于）(SQL Server Compact)

比较两个表达式。 当使用此运算符比较非空表达式时，如果左操作数不等于右操作数，则结果为 TRUE。 否则，结果为 FALSE。]

**例6：**

原表：  
courseid coursename score  
1 java 70  
2 oracle 90  
3 xml 40  
4 jsp 30  
5 servlet 80

为了便于阅读,查询此表后的结果显式如下(及格分数为60):  
courseid coursename score mark  
1 java 70 pass  
2 oracle 90 pass  
3 xml 40 fail  
4 jsp 30 fail  
5 servlet 80 pass  
写出此查询语句

**答：**

**select** courseid, coursename ,score ,if(score>=60, "pass","fail")  **as** mark **from** course

**例7：**

表名：购物信息

购物人 商品名称 数量

A 甲 2

B 乙 4

C 丙 1

A 丁 2

B 丙 5

给出所有购入商品为两种或两种以上的购物人记录

**答：**

**select** \* **from** 购物信息 **where** 购物人 in (**select** 购物人 **from** 购物信息 **group** **by** 购物人 **having** count(\*) >= 2);

**例8：**

info 表

date result

2005-05-09 win

2005-05-09 lose

2005-05-09 lose

2005-05-09 lose

2005-05-10 win

2005-05-10 lose

2005-05-10 lose

如果要生成下列结果, 该如何写sql语句?

date win lose

2005-05-09 2 2

2005-05-10 1 2

**答1：**

**select** **date**, sum(case **when** result = "win" **then** 1 **else** 0 **end**) **as** "win", sum(case **when** result = "lose" **then** 1 **else** 0 **end**) **as** "lose" **from** info **group** **by** **date**;

**答2：**

1. **select** a.**date**, a.result **as** win, b.result **as** lose
2. **from**
3. (**select** **date**, count(result) **as** result **from** info **where** result = "win" **group** **by** **date**) **as** a
4. join
5. (**select** **date**, count(result) **as** result **from** info **where** result = "lose" **group** **by** **date**) **as** b
6. **on** a.**date** = b.**date**;

## 6.12联合索引是什么?为什么需要注意联合索引中的顺序?

mysql可以使用多个字段同时建立一个索引,叫做联合索引,同时联合索引在建立时要留意将高频的字段索引放在前面

具体原因为:

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

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

## 6.13什么是聚簇索引和非聚簇索引

聚簇索引即是主键索引，他索引的内容就是整行数据。

非聚簇索引即使非主键索引，他索引的是主键id。

如果一个表中没有主键索引，那么mysql会寻找唯一的非空索引做为主键索引，如果没有则隐式的定义一个主键来作为聚簇索引。

## 6.14 delete、drop、truncate区别

1、delete 和 truncate 仅仅删除表数据，drop 连表数据和表结构一起删除，打个比方，delete 是单杀，truncate 是团灭，drop 是把电脑摔了。

2、delete 是 DML 语句，操作完以后如果没有不想提交事务还可以回滚，truncate 和 drop 是 DDL 语句，操作完马上生效，不能回滚，打个比方，delete 是发微信说分手，后悔还可以撤回，truncate 和 drop 是直接扇耳光说滚，不能反悔。

3、执行的速度上，drop>truncate>delete，打个比方，drop 是神舟火箭，truncate 是和谐号动车，delete 是自行车。

## 6.15 mysql主从复制

slave 服务器执行 start slave，开启主从复制开关，slave 服务器的 I/O Thread 请求从 master 服务器读取 binlog（如果该线程追赶上了主库，会进入睡眠状态）

master 服务器创建 Log Dump Thread，把 binlog 发送给 slave 服务器。slave 服务器的 I/O Thread 将读取到的 binlog 日志内容写入中继日志 relay log（中继日志，mysql-relay-bin.xxxxxx，会记录位置信息，以便下次继续读取）

slave 服务器的 SQL Thread 会实时监测 relay log 新增的日志内容，把 relay log

解析成 SQL 语句，并执行。

## 6.16 MySQL中的varchar和char的区别以及varchar（50）中的50代表的涵义

1)、varchar与char的区别

char是一种固定长度的类型，varchar则是一种可变长度的类型

尽可能的使用 varchar 代替 char ，因为首先变长字段存储空间小，可以节省存储空间，

其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

varchar(50)代表的含义：

varchar(50)中50的涵义最多存放50个字符，varchar(50)和(200)存储hello所占空间一样，但后者在排序时会消耗更多内存，因为order by col采用fixed\_length计算col长度(memory引擎也一样)

int（20）中20的涵义:

是指显示字符的长度但要加参数的，最大为255，比如它是记录行数的id,插入10笔资料，

它就显示00000000001 ~~~00000000010，当字符的位数超过11,它也只显示11位，如果你没有加那个让它未满11位就前面加0的参数，

它不会在前面加020表示最大显示宽度为20，但仍占4字节存储，存储范围不变；

## 6.17一张表里面有ID自增主键，当insert了17条记录之后，删除了第15,16,17条记录，再把mysql重启，再insert一条记录，这条记录的ID是18还是15 ？

一般情况下，我们创建的表的类型是InnoDB，如果新增一条记录（不重启mysql的情况下），这条记录的id是18；但是如果重启（文中提到的）MySQL的话，这条记录的ID是15。因为InnoDB表只把自增主键的最大ID记录到内存中，所以重启数据库或者对表OPTIMIZE操作，都会使最大ID丢失。

        但是，如果我们使用表的类型是MylSAM，那么这条记录的ID就是18。因为MylSAM表会把自增主键的最大ID记录到数据文件里面，重启MYSQL后，自增主键的最大ID也不会丢失。

注：如果在这17条记录里面删除的是中间的几个记录（比如删除的是10,11,12三条记录），重启MySQL数据库后，insert一条记录后，ID都是18。因为内存或者数据库文件存储都是自增主键最大ID

## 6.18为什么使用数据索引能提高效率

索引就是通过事先排好序，从而在查找时可以应用二分查找等高效率的算法。

一般的顺序查找，复杂度为O(n)，而二分查找复杂度为O(log2n)。当n很大时，二者的效率相差及其悬殊。

举个例子：

表中有一百万条数据，需要在其中寻找一条特定id的数据。如果顺序查找，平均需要查找50万条数据。而用二分法，至多不超过20次就能找到。二者的效率差了2.5万倍！

在一个或者一些字段需要频繁用作查询条件，并且表数据较多的时候，创建索引会明显提高查询速度，因为可由全表扫描改成索引扫描。

（无索引时全表扫描也就是要逐条扫描全部记录，直到找完符合条件的，索引扫描可以直接定位）

不管数据表有无索引，首先在SGA的数据缓冲区中查找所需要的数据，如果数据缓冲区中没有需要的数据时，服务器进程才去读磁盘。

1、无索引，直接去读表数据存放的磁盘块，读到数据缓冲区中再查找需要的数据。

2、有索引，先读入索引表，通过索引表直接找到所需数据的物理地址，并把数据读入数据缓冲区中。

## 6.19 B+树索引和哈希索引的区别

首先要知道Hash索引和B+树索引的底层实现原理:

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

那么可以看出他们有以下的不同:

hash索引进行等值查询更快(一般情况下),但是却无法进行范围查询.

因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.而B+树的的所有节点皆遵循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.

hash索引不支持使用索引进行排序,原理同上.

hash索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为hash函数的不可预测.AAAA和AAAAB的索引没有相关性.

hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.

hash索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差.而B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.

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

## 6.20什么情况下应不建或少建索引

1. 表记录太少

如果一个表只有5条记录，采用索引去访问记录的话，那首先需访问索引表，再通过索引表访问数据表，一般索引表与数据表不在同一个数据块，这种情况下ORACLE至少要往返读取数据块两次。而不用索引的情况下ORACLE会将所有的数据一次读出，处理速度显然会比用索引快。

如表zl\_sybm（使用部门）一般只有几条记录，除了主关键字外对任何一个字段建索引都不会产生性能优化，实际上如果对这个表进行了统计分析后ORACLE也不会用你建的索引，而是自动执行全表访问。如：

select \* from zl\_sybm where sydw\_bh=’5401’（对sydw\_bh建立索引不会产生性能优化）

2. 经常插入、删除、修改的表

对一些经常处理的业务表应在查询允许的情况下尽量减少索引，如zl\_yhbm，gc\_dfss，gc\_dfys，gc\_fpdy等业务表。

3. 数据重复且分布平均的表字段

假如一个表有10万行记录，有一个字段A只有T和F两种值，且每个值的分布概率大约为50%，那么对这种表A字段建索引一般不会提高数据库的查询速度。

4. 经常和主字段一块查询但主字段索引值比较多的表字段

如gc\_dfss（电费实收）表经常按收费序号、户标识编号、抄表日期、电费发生年月、操作 标志来具体查询某一笔收款的情况，如果将所有的字段都建在一个索引里那将会增加数据的修改、插入、删除时间，从实际上分析一笔收款如果按收费序号索引就已 经将记录减少到只有几条，如果再按后面的几个字段索引查询将对性能不产生太大的影响。

## 6.21行级锁定的优点缺点

优点：

1、当在许多线程中访问不同的行时只存在少量锁定冲突。

2、回滚时只有少量的更改。

3、可以长时间锁定单一的行。

缺点：

1、比页级或表级锁定占用更多的内存。

2、当在表的大部分中使用时，比页级或表级锁定速度慢，因为你必须获取更多的锁。

3、如果你在大部分数据上经常进行GROUP BY操作或者必须经常扫描整个表，比其它锁定明显慢很多。

4、用高级别锁定，通过支持不同的类型锁定，你也可以很容易地调节应用程序，因为其锁成本小于行级锁定。

## 6.22创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

MySQL提供了explain命令来查看语句的执行计划,MySQL在执行某个语句之前,会将该语句过一遍查询优化器,之后会拿到对语句的分析,也就是执行计划,其中包含了许多信息. 可以通过其中和索引有关的信息来分析是否命中了索引,例如possilbe\_key,key,key\_len等字段,分别说明了此语句可能会使用的索引,实际使用的索引以及使用的索引长度.

## 6.23 varchar(10)和int(10)代表什么含义?

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.也就是说,int(1)和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示.

## 6.24关心过业务系统里面的sql耗时吗?统计过慢查询吗?对慢查询都怎么优化过?

在业务系统中,除了使用主键进行的查询,其他的我都会在测试库上测试其耗时,慢查询的统计主要由运维在做,会定期将业务中的慢查询反馈给我们.

慢查询的优化首先要搞明白慢的原因是什么? 是查询条件没有命中索引?是load了不需要的数据列?还是数据量太大?

所以优化也是针对这三个方向来的,

首先分析语句,看看是否load了额外的数据,可能是查询了多余的行并且抛弃掉了,可能是加载了许多结果中并不需要的列,对语句进行分析以及重写.

分析语句的执行计划,然后获得其使用索引的情况,之后修改语句或者修改索引,使得语句可以尽可能的命中索引.

如果对语句的优化已经无法进行,可以考虑表中的数据量是否太大,如果是的话可以进行横向或者纵向的分表.

## 6.25什么是存储过程？有哪些优缺点？

存储过程是一些预编译的SQL语句。1、更加直白的理解：存储过程可以说是一个记录集，它是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。2、存储过程是一个预编译的代码块，执行效率比较高,一个存储过程替代大量T\_SQL语句 ，可以降低网络通信量，提高通信速率,可以一定程度上确保数据安全

但是,在互联网项目中,其实是不太推荐存储过程的,比较出名的就是阿里的《Java开发手册》中禁止使用存储过程,我个人的理解是,在互联网项目中,迭代太快,项目的生命周期也比较短,人员流动相比于传统的项目也更加频繁,在这样的情况下,存储过程的管理确实是没有那么方便,同时,复用性也没有写在服务层那么好.

## 6.26字段为什么要定义为NOT NULL?

一般情况，都会设置一个默认值，不会出现字段里面有null，又有空的情况。主要有以下几个原：

1. 索引性能不好，Mysql难以优化引用可空列查询，它会使索引、索引统计和值更加复杂。可空列需要更多的存储空间，还需要mysql内部进行特殊处理。可空列被索引后，每条记录都需要一个额外的字节，还能导致MYisam 中固定大小的索引变成可变大小的索引。

2. 如果某列存在null的情况，可能导致count() 等函数执行不对的情况

## 6.27货币字段用什么类型？

如果货币单位是分，可以用LONG类型。如果坚持用元，用Decimal。

## 6.28时间字段用什么类型?

此题无固定答案，应结合自己项目背景来答！把理由讲清楚就行！

(1)varchar，如果用varchar类型来存时间，优点在于显示直观。但是坑的地方也是挺多的。比如，插入的数据没有校验，你可能某天就发现一条数据为2013111的数据，请问这是代表2013年1月11日，还是2013年11月1日？

其次，做时间比较运算，你需要用STR\_TO\_DATE等函数将其转化为时间类型，你会发现这么写是无法命中索引的。数据量一大，是个坑！

(2)timestamp，该类型是四个字节的整数，它能表示的时间范围为1970-01-01 08:00:01到2038-01-19 11:14:07。2038年以后的时间，是无法用timestamp类型存储的。

但是它有一个优势，timestamp类型是带有时区信息的。一旦你系统中的时区发生改变，例如你修改了时区

SET TIME\_ZONE = "america/new\_york";

你会发现，项目中的该字段的值自己会发生变更。这个特性用来做一些国际化大项目，跨时区的应用时，特别注意！

(3)datetime，datetime储存占用8个字节，它存储的时间范围为1000-01-01 00:00:00 ~ 9999-12-31 23:59:59。显然，存储时间范围更大。但是它坑的地方在于，他存储的是时间绝对值，不带有时区信息。如果你改变数据库的时区，该项的值不会自己发生变更！

(4)bigint，也是8个字节，自己维护一个时间戳，表示范围比timestamp大多了，就是要自己维护，不大方便。

## 6.29为什么不直接存储图片、音频、视频等大容量内容?

我们在实际应用中，都是文件形式存储的。mysql中，只存文件的存放路径。虽然mysql中blob类型可以用来存放大容量文件，但是，我们在生产中，基本不用！

主要有如下几个原因：

　　1. Mysql内存临时表不支持TEXT、BLOB这样的大数据类型，如果查询中包含这样的数据，查询效率会非常慢。

　　2. 数据库特别大，内存占用高，维护也比较麻烦。

　　3. binlog太大，如果是主从同步的架构，会导致主从同步效率问题！

因此，不推荐使用blob等类型！

## 6.30为什么一定要设一个主键？

因为不设主键，innodb也会帮你生成一个隐形列作为自增主键。反正都要生成主键，自己指定主键可以显示用上主键索引，提高查询效率。

## 6.31你们主键是用自增还是UUID?

　自增。innodb住的主键是聚簇索引，如果主键是自增的，每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，一页写满后自动开新页。不是自增主键，可能会在中间插入，引发页的分裂，产生很多表碎片。相比之下，自增插入性能更好。

## 6.32自增主键用完了怎么办?

自增主键用的是int类型的话，最大值是2147483648.如果达到最大值，数据有21亿条，这时应该做分表分库处理，不应该单表存这么多数据。

## 6.33主键为什么不推荐有业务含义?

带有业务含义的主键可能会发生变更，引发页分裂，产生空间碎片。

# 第7章Java Web

## 7.1 http 的长连接和短连接

HTTP 协议有 HTTP/1.0 版本和 HTTP/1.1 版本。HTTP1.1 默认保持长连接（HTTP persistent connection，也翻译为持久连接），数据传输完成了保持 TCP 连接不断开（不发 RST 包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

在 HTTP/1.0 中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。从 HTTP/1.1 起，默认使用的是长连接，用以保持连接特性。

## 7.2 http 常见的状态码有哪些？

200 OK //客户端请求成功

301 Moved Permanently（永久移除)，请求的 URL 已移走。Response 中应该包含一个 Location URL, 说明资源现在所处的位置

302 found 重定向

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的 URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

## 7.3 GET 和POST 的区别？

（1）GET 请求的数据会附在URL 之后（就是把数据放置在 HTTP 协议头中），以?分割URL 和传输数据，参数之间以&相连，如：login.action?name=zhagnsan&password=123456。POST 把提交的数据则放置在是 HTTP 包的包体中。

（2）GET 方式提交的数据最多只能是 1024 字节，理论上POST 没有限制，可传较大量的数据。其实这样说是错误的，不准确的：“GET 方式提交的数据最多只能是 1024 字节"，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对URL 长度的限制是2083 字节(2K+35)。对于其他浏览器，如Netscape、FireFox 等，理论上没有长度限制，其限制取决于操作系统的支持。

（3）POST 的安全性要比GET 的安全性高。注意：这里所说的安全性和上面 GET 提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的 Security 的含义，比如：通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用 GET 提交数据还可能会造成 Cross-site request forgery 攻击。

Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM（表单）中，Method

默认为"GET"，实质上，GET 和 POST 只是发送机制不同，并不是一个取一个发！

## 7.4 Cookie 和Session 的区别

Cookie 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。

Session 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Cookie 和session 的不同点：

（1）无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。

（2）在存储的数据量方面：session 能够存储任意的java 对象，cookie 只能存储 String 类型的对象。

## 7.5在单点登录中，如果 cookie 被禁用了怎么办？

单点登录的原理是后端生成一个 session ID，然后设置到 cookie，后面的所有请求浏览器都会带上 cookie， 然后服务端从 cookie 里获取 session ID，再查询到用户信息。所以，保持登录的关键不是 cookie，而是通过cookie 保存和传输的 session ID，其本质是能获取用户信息的数据。除了 cookie，还通常使用 HTTP 请求头来传输。但是这个请求头浏览器不会像 cookie 一样自动携带，需要手工处理。

## 7.6什么是jsp，什么是Servlet？jsp 和Servlet 有什么区别？

jsp 本质上就是一个Servlet，它是 Servlet 的一种特殊形式（由 SUN 公司推出），每个 jsp 页面都是一个servlet实例。

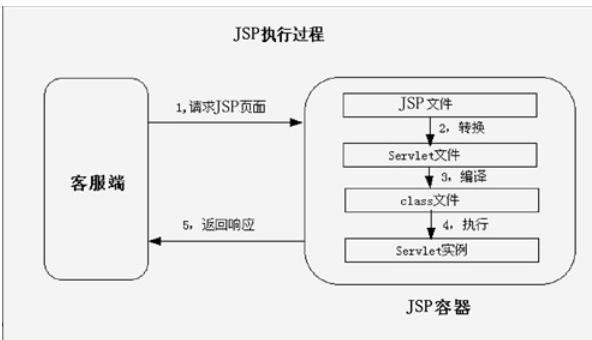
Servlet 是由 Java 提供用于开发 web 服务器应用程序的一个组件，运行在服务端，由 servlet 容器管理，用来生成动态内容。一个 servlet 实例是实现了特殊接口 Servlet 的 Java 类，所有自定义的 servlet 均必须实现 Servlet 接口。

**区别：**

jsp 是 html 页面中内嵌的Java 代码，侧重页面显示；

Servlet 是 html 代码和 Java 代码分离，侧重逻辑控制，mvc 设计思想中jsp 位于视图层，servlet 位于控制层

Jsp 运行机制：如下图



JVM 只能识别 Java 类，并不能识别 jsp 代码！web 容器收到以.jsp 为扩展名的 url 请求时，会将访问请求交给tomcat 中 jsp 引擎处理，每个 jsp 页面第一次被访问时，jsp 引擎将 jsp 代码解释为一个 servlet 源程序，接着编译servlet 源程序生成.class 文件，再有 web 容器 servlet 引擎去装载执行servlet 程序，实现页面交互。

## 7.7 servlet生命周期

Servlet 加载—>实例化—>服务—>销毁。

**生命周期详解：**

**init（）：**

在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。

**service（）：**

它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个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”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。

**destroy（）：**

仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

**如何与Tomcat 结合工作步骤：**

（1）Web Client 向Servlet容器（Tomcat）发出Http请求

（2）Servlet容器接收Web Client的请求

（3）Servlet容器创建一个HttpRequest对象，将Web Client请求的信息封装到这个对象中。

（4）Servlet容器创建一个HttpResponse对象

（5）Servlet容器调用HttpServlet对象的service方法，把HttpRequest对象与HttpResponse对象作为参数传给HttpServlet 对象。

（6）HttpServlet调用HttpRequest对象的有关方法，获取Http请求信息。

（7）HttpServlet调用HttpResponse对象的有关方法，生成响应数据。

## 7.8 servlet特性

单例多线程

## 7.9 servlet是单实例的吗？

servlet是单实例的

## 7.10 servlet是线程安全的吗？为什么？

Servlet对象并不是一个线程安全的对象。

Servlet第一次被调用的时候，init()方法会被调用，然后调用service() 方法，从第二次被请求开始，就直接调用service()方法。

因为servlet是单实例的，所以后面再次请求同一个Servlet的时候都不会创建Servlet实例，

而且web容器会针对每个请求创建一个独立的线程，这样多个并发请求会导致多个线程同时调用 service() 方法，这样就会存在线程不安全的问题。

## 7.11如何解决Servlet线程不安全的问题？

（1）不要在servlet中使用成员变量。

（2）可以给servlet中的方法添加同步锁，Synchronized，但是不提倡，数据并发访问会造成阻塞等待。

（3）可以实现 SingleThreadModel 接口，如下。这样可以避免使用成员变量的问题，但是也不提倡，原因同上。

Public class Servlet1 extends HttpServlet implements SingleThreadModel{

……..

}

## 7.12谈谈过滤器的作用

过滤器，是在java web中，你传入的request,response提前过滤掉一些信息，或者提前设置一些参数，然后再传入servlet或者struts的 action进行业务逻辑，比如过滤掉非法url（不是login.do的地址请求，如果用户没有登陆都过滤掉）,或者在传入servlet或者 struts的action前统一设置字符集，或者去除掉一些非法字符

## 7.13谈谈拦截器的作用

拦截器，是在面向切面编程的就是在你的service或者一个方法，前调用一个方法，或者在方法后调用一个方法比如动态代理就是拦截器的简单实现，在你调用方法前打印出字符串（或者做其它业务逻辑的操作），也可以在你调用方法后打印出字符串，甚至在你抛出异常的时候做业务逻辑的操作。

## 7.14拦截器和过滤器有什么区别

拦截器是基于java的反射机制的，而过滤器是基于函数回调。

拦截器不依赖servlet容器，过滤器依赖与servlet容器。

拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用。

拦截器可以访问action上下文、值栈里的对象，而过滤器不能访问。

在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次

## 7.15拦截器和过滤器的执行顺序

过滤前 – 拦截前 – Action处理 – 拦截后 – 过滤后。

过滤是一个横向的过程，首先把客户端提交的内容进行过滤(例如未登录用户不能访问内部页面的处理)；过滤通过后，拦截器将检查用户提交数据的验证，做一些前期的数据处理，接着把处理后的数据发给对应的Action；Action处理完成返回后，拦截器还可以做其他过程(还没想到要做啥)，再向上返回到过滤器的后续操作。

## 7.16说一下 jsp 的 4 种作用域？

application、session、request、page

application 作用域

如果把变量放到application里，就说明它的作用域是application，它的有效范围是整个应用。 整个应用是指从应用启动，到应用结束。我们没有说“从服务器启动，到服务器关闭”，是因为一个服务器可能部署多个应用，当然你关闭了服务器，就会把上面所有的应用都关闭了。 application作用域里的变量，它们的存活时间是最长的，如果不进行手工删除，它们就一直可以使用。

application作用域上的信息传递是通过ServletContext实现的，它提供的主要方法如下所示：

Object getAttribute（String name） //从application中获取信息；

void setAttribute（String name, Object value） //向application作用域中设置信息。

session作用域

session作用域比较容易理解，同一浏览器对服务器进行多次访问，在这多次访问之间传递信息，就是session作用域的体现。如果把变量放到session里，就说明它的作用域是session，它的有效范围是当前会话。所谓当前会话，就是指从用户打开浏览器开始，到用户关闭浏览器这中间的过程。这个过程可能包含多个请求响应。也就是说，只要用户不关浏览器，服务器就有办法知道这些请求是一个人发起的，整个过程被称为一个会话（session），而放到会话中的变量，就可以在当前会话的所有请求里使用。

session是通过HttpSession接口实现的，它提供的主要方法如下所示：

ObjectHttpSession.getAttribute（String name） //从session中获取信息。

void HttpSession.setAttribute（String name, Object value）//向session中保存信息。

HttpSession HttpServletRequest.getSessio() //获取当前请求所在的session的对象。

session的开始时刻比较容易判断，它从浏览器发出第一个HTTP请求即可认为会话开始。但结束时刻就不好判断了，因为浏览器关闭时并不会通知服务器，所以只能通过如下这种方法判断：如果一定的时间内客户端没有反应，则认为会话结束。Tomcat的默认值为120分钟，但这个值也可以通过HttpSession的setMaxInactiveInterval()方法来设置：

void setMaxInactiveInterval(int interval)

如果想主动让会话结束，例如用户单击“注销”按钮的时候，可以使用 HttpSession 的 invalidate()方法，用于强制结束当前session：void invalidate()

request作用域

一个HTTP请求的处理可能需要多个Servlet合作，而这几个Servlet之间可以通过某种方式传递信息，但这个信息在请求结束后就无效了。request里的变量可以跨越forward前后的两页。但是只要刷新页面，它们就重新计算了。如果把变量放到request里，就说明它的作用域是request，它的有效范围是当前请求周期。 所谓请求周期，就是指从http请求发起，到服务器处理结束，返回响应的整个过程。在这个过程中可能使用forward的方式跳转了多个jsp页面，在这些页面里你都可以使用这个变量。

Servlet之间的信息共享是通过HttpServletRequest接口的两个方法来实现的：

void setAttribute（String name, Object value） //将对象value以name为名称保存到request作用域中。

Object getAttribute（String name） //从request作用域中取得指定名字的信息。

JSP中的doGet()、doPost()方法的第一个参数就是HttpServletRequest对象，使用这个对象的 setAttribute()方法即可传递信息。那么在设置好信息之后，要通过何种方式将信息传给其他的Servlet呢？这就要用到RequestDispatcher接口的forward()方法，通过它将请求转发给其他Servlet。

RequestDispatcher ServletContext.getRequestDispatcher(String path) //取得Dispatcher以便转发，path为转发的目的Servlet。

void RequestDispatcher.forward(ServletRequest request, ServletResponse response)//将request和response转发

因此，只需要在当前Servlet中先通过setAttribute()方法设置相应的属性，然后使用forward()方法进行跳转，最后在跳转到的Servlet中通过使用getAttribute()方法即可实现信息传递。

需要注意两点：

1、转发不是重定向，转发是在Web应用内部进行的。

2、转发对浏览器是透明的，也就是说，无论在服务器上如何转发，浏览器地址栏中显示的仍然是最初那个Servlet的地址。

page作用域

page对象的作用范围仅限于用户请求的当前页面，对于page对象的引用将在响应返回给客户端之后被释放，或者在请求被转发到其他地方后被释放。page里的变量只要页面跳转了，它们就不见了。如果把变量放到pageContext里，就说明它的作用域是page，它的有效范围只在当前jsp页面里。从把变量放到pageContext开始，到jsp页面结束，你都可以使用这个变量。

以上介绍的作用范围越来越小，request和page的生命周期都是短暂的，它们之间的区别：一个request可以包含多个page页（include，forward及filter）。

## 7.17 jsp 有哪些内置对象？作用分别是什么？

JSP共有以下9个内置的对象：

request 用户端请求，此请求会包含来自GET/POST请求的参数

response 网页传回用户端的回应

pageContext 网页的属性是在这里管理

session 与请求有关的会话期

application servlet 正在执行的内容

out 用来传送回应的输出

config servlet的构架部件

page JSP网页本身

exception 针对错误网页，未捕捉的例外

request表示HttpServletRequest对象。它包含了有关浏览器请求的信息，并且提供了几个用于获取cookie, header,和session数据的有用的方法。

response表示HttpServletResponse对象，并提供了几个用于设置送回浏览器的响应的方法（如cookies,头信息等）

out对象是javax.jsp.JspWriter的一个实例，并提供了几个方法使你能用于向浏览器回送输出结果。

pageContext表示一个javax.servlet.jsp.PageContext对象。它是用于方便存取各种范围的名字空间、servlet相关的对象的API，并且包装了通用的

servlet相关功能的方法。

session表示一个请求的javax.servlet.http.HttpSession对象。Session可以存贮用户的状态信息

applicaton 表示一个javax.servle.ServletContext对象。这有助于查找有关servlet引擎和servlet环境的信息

config表示一个javax.servlet.ServletConfig对象。该对象用于存取servlet实例的初始化参数。

page表示从该页面产生的一个servlet实例

## 7.18转发（Forward）和重定向（Redirect）的区别？

重定向会改变URL地址，请求转发不会

重定向可以用URL绝对路径访问其他web服务器的资源，而请求转发只能在一个web应用程序内进行资源转发

重定向效率低，相当于再一次请求，请求转发跳转仅发生在服务器端。

## 7.19Request对象的主要方法有哪些？

Request对象的主要方法：

setAttribute(String name,Object)：设置名字为name的request 的参数值

getAttribute(String name)：返回由name指定的属性值

getAttributeNames()：返回request 对象所有属性的名字集合，结果是一个枚举的实例

getCookies()：返回客户端的所有 Cookie 对象，结果是一个Cookie 数组

getCharacterEncoding() ：返回请求中的字符编码方式

getContentLength() ：返回请求的 Body的长度

getHeader(String name) ：获得HTTP协议定义的文件头信息

getHeaders(String name) ：返回指定名字的request Header 的所有值，结果是一个枚举的实例

getHeaderNames() ：返回所以request Header 的名字，结果是一个枚举的实例

getInputStream() ：返回请求的输入流，用于获得请求中的数据

getMethod() ：获得客户端向服务器端传送数据的方法

getParameter(String name) ：获得客户端传送给服务器端的有 name指定的参数值

getParameterNames() ：获得客户端传送给服务器端的所有参数的名字，结果是一个枚举的实

## 7.20request.getAttribute()和 request.getParameter()有何区别？

getParameter 得到的都是 String 类型的。或者是 http://a.jsp?id=123 中的 123，或者是某个表

单提交过去的数据。

getAttribute 则可以是对象。

getParameter()是获取 POST/GET 传递的参数值；

getAttribute()是获取对象容器中的数据值；

getParameter：用于客户端重定向时，即点击了链接或提交按扭时传值用，即用于在用表单

或 url 重定向传值时接收数据用。

getAttribute：用于服务器端重定向时，即在 sevlet 中使用了 forward 函数,或 struts 中使用了

mapping.findForward。 getAttribute 只能收到程序用 setAttribute 传过来的值。

getParameter()是获取 POST/GET 传递的参数值；

getAttribute()是获取 SESSION 的值；

另外，可以用 setAttribute,getAttribute 发送接收对象.而 getParameter 显然只能传字符串。

setAttribute 是应用服务器把这个对象放在该页面所对应的一块内存中去，当你的页面服务器

重定向到另一个页面时，应用服务器会把这块内存拷贝另一个页面所对应的内存中。这样

getAttribute 就能取得你所设下的值，当然这种方法可以传对象。 session 也一样，只是对象

在内存中的生命周期不一样而已。 getParameter 只是应用服务器在分析你送上来的 request

页面的文本时，取得你设在表单或 url 重定向时的值。

getParameter 返回的是 String, 用于读取提交的表单中的值;

getAttribute 返回的是 Object，需进行转换,可用 setAttribute 设置成任意对象，使用很灵活，

可随时用；

## 7.21JSP中动态include和静态include的区别？

动态的include：

用法：<jsp:include page="1.jsp" flush="true" />

特点：行为元素，可以带参数；先编译，再页面合成；它总是会检查所含文件中的变化，适合用于包含动态页面；

因此，动态include的结构是互相独立的，所包含的jsp文件中的变量不可以同它的主文件共享，需要自行创建所使用的对象和页面设置；

静态的include:

用法：<%@ include file="1.htm" %>

特点：指令元素；不能带参数；先页面合成，再编译；不会检查所含文件的变化，适用于包含静态页面；

因此，静态include的结构是高度紧密的，所包含的jsp文件中所有的变量都可以同它的主文件共享,但不能有变量同名的冲突，连页面设置都可以借用主文件的.

扩展：

两种用法中file和page属性都被解释为一个相对的URI.

若以斜杠开头,说明它是一个环境相关的路径.将根据所指定URI的前缀进行解释,

若不是不是以斜杠开头,说明它是页面相关的路径,将根据当前页面路径进行解释.

## 7.22JSP乱码如何解决？

1.查看jsp文件头是否设置了编码格式：

2.查看项目的编码格式：设置为UTF-8

3.提交的表单乱码等问题，需要在请求头响应头设置编码

4. 设置tomcat服务器编码格式，默认情况下，tomcat使用的的编码方式：iso8859-1，打开setting.xml文件（在tomcat文件夹conf中）

## 7.23什么是Tomcat？

Tomcat简单的说就是一个运行JAVA的网络服务器，底层是Socket的一个程序，它也是JSP和Serlvet的一个容器。

## 7.24详细描述MVC

基于java的web应用系统采用MVC设计模型，即用Model（模型）、View（视图）和Controller（控制）分离设计，这是目前web应用服务系统的主流设置方向。

Model：处理业务逻辑的模块。

View：负责页面显示，显示Model的处理结果给用户，主要实现数据到页面的转换过程。

Controller：负责每个请求的分发，把Form数据传递给Model进行处理，处理完成后，把处理结果返回给相应的View显示给用户。

## 7.25Http请求由哪三部分组成?

http协议报文

1.请求报文(请求行/请求头/请求数据/空行)

请求行

求方法字段、URL字段和HTTP协议版本

例如：GET /index.html HTTP/1.1

get方法将数据拼接在url后面，传递参数受限

请求方法：

GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT

请求头(key value形式)

User-Agent：产生请求的浏览器类型。

Accept：客户端可识别的内容类型列表。

Host：主机地址

请求数据

post方法中，会把数据以key value形式发送请求

空行

发送回车符和换行符，通知服务器以下不再有请求头

2.响应报文(状态行、消息报头、响应正文)

状态行

消息报头

响应正文

## 7.26如何实现跨域？

1、jsonp

利用了 script 不受同源策略的限制

缺点：只能 get 方式，易受到 XSS攻击

2、CORS（Cross-Origin Resource Sharing）,跨域资源共享

当使用XMLHttpRequest发送请求时，如果浏览器发现违反了同源策略就会自动加上一个请求头 origin；

后端在接受到请求后确定响应后会在后端在接受到请求后确定响应后会在 Response Headers 中加入一个属性 Access-Control-Allow-Origin；

浏览器判断响应中的 Access-Control-Allow-Origin 值是否和当前的地址相同，匹配成功后才继续响应处理，否则报错

缺点：忽略 cookie，浏览器版本有一定要求

3、代理跨域请求

前端向发送请求，经过代理，请求需要的服务器资源

缺点：需要额外的代理服务器

4、Html5 postMessage 方法

允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本、多窗口、跨域消息传递

缺点：浏览器版本要求，部分浏览器要配置放开跨域限制

5、修改 document.domain 跨子域

相同主域名下的不同子域名资源，设置 document.domain 为 相同的一级域名

缺点：同一一级域名；相同协议；相同端口

6、基于 Html5 websocket 协议

websocket 是 Html5 一种新的协议，基于该协议可以做到浏览器与服务器全双工通信，允许跨域请求

缺点：浏览器一定版本要求，服务器需要支持 websocket 协议

7、document.xxx + iframe

通过 iframe 是浏览器非同源标签，加载内容中转，传到当前页面的属性中

缺点：页面的属性值有大小限制

## 7.27简述 tcp 和 udp的区别？

1：TCP基于连接，UDP基于无连接。

2：TCP对系统资源要求高，UDP少。

3：TCP是基于字节流的，UDP是数据报文模式。

4：TCP复杂，UDP简单。

## 7.28tcp 为什么要三次握手，两次不行吗？为什么？

两次握手只能保证单向连接是畅通的。

Step1       A -> B : 你好，B。

Step2       A <- B : 收到。你好，A。

这样的两次握手过程， A 向 B 打招呼得到了回应，即 A 向 B 发送数据，B 是可以收到的。

但是 B 向 A 打招呼，A 还没有回应，B 没有收到 A 的反馈，无法确保 A 可以收到 B 发送的数据。

只有经过第三次握手，才能确保双向都可以接收到对方的发送的 数据。

Step3       A -> B : 收到，B。

这样 B 才能确定 A 也可以收到 B 发送给 A 的数据。

# SSM框架

## 8.1请写出 spring 中常用的依赖注入方式。

通过 setter 方法注入

通过构造方法注入

## 8.2简述Spring中IOC容器常用的接口和具体的实现类

* BeanFactory SpringIOC容器的基本设置，是最底层的实现， 面向框架本身的.
* ApplicationContext BeanFactory的子接口, 提供了更多高级的特定. 面向开发者的.
* ConfigurableApplicationContext, ApplicationContext的子接口，扩展出了 close 和 refresh等 关闭 刷新容器的方法
* ClassPathXmlApplicationContext：从classpath的XML配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得。
* FileSystemXmlApplicationContext ：由文件系统中的XML配置文件读取上下文。
* XmlWebApplicationContext：由Web应用的XML文件读取上下文。

## 8.3简述Spring中如何基于注解配置Bean和装配Bean

（1）首先要在Spring中配置开启注解扫描

|  |
| --- |
| <context:component-scan base-package=” ”></ context:component-scan> |

（2）在具体的类上加上具体的注解

（3）Spring 中通常使用@Autowired 或者是@Resource 等注解进行bean的装配

## 8.4说出Spring 或者 Springmvc中常用的5个注解，并解释含义

@Component 基本注解，标识一个受Spring管理的组件

@Controller 标识为一个表示层的组件

@Service 标识为一个业务层的组件

@Repository 标识为一个持久层的组件

@Autowired 自动装配

@Qualifier(“”) 具体指定要装配的组件的id值

@RequestMapping() 完成请求映射

@PathVariable 映射请求URL中占位符到请求处理方法的形参

只要说出机几个注解并解释含义即可，如上答案只做参考

## 8.5请解释Spring Bean的生命周期？

（1）默认情况下，IOC容器中bean的生命周期分为五个阶段:

* 调用构造器 或者是通过工厂的方式创建Bean对象
* 给bean对象的属性注入值
* 调用初始化方法，进行初始化， 初始化方法是通过init-method来指定的.
* 使用
* IOC容器关闭时， 销毁Bean对象.

（2）当加入了Bean的后置处理器后，IOC容器中bean的生命周期分为七个阶段:

* 调用构造器 或者是通过工厂的方式创建Bean对象
* 给bean对象的属性注入值
* 执行Bean后置处理器中的 postProcessBeforeInitialization
* 调用初始化方法，进行初始化， 初始化方法是通过init-method来指定的.
* 执行Bean的后置处理器中 postProcessAfterInitialization
* 使用
* IOC容器关闭时， 销毁Bean对象

只需要回答出第一点即可，第二点也回答可适当 加分。

## 8.6简单的谈一下SpringMVC的工作流程？

* 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调用HandlerAdapter处理器适配器
* HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
* Controller执行完成返回ModelAndView
* HandlerAdapter将controller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
* DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
* DispatcherServlet响应用户

## 8.7 SpringMVC中如何解决POST请求中文乱码问题

Springmvc中通过CharacterEncodingFilter解决中文乱码问题.

在web.xml中加入：

1. **<filter>**
2. **<filter-name>**CharacterEncodingFilter**</filter-name>**
3. **<filter-class>**org.springframework.web.filter.CharacterEncodingFilter**</filter-class>**
4. **<init-param>**
5. **<param-name>**encoding**</param-name>**
6. **<param-value>**utf-8**</param-value>**
7. **</init-param>**
8. **</filter>**
9. **<filter-mapping>**
10. **<filter-name>**CharacterEncodingFilter**</filter-name>**
11. **<url-pattern>**/\***</url-pattern>**
12. **</filter-mapping>**

## 8.8简述SpringMvc里面拦截器是如何定义，如何配置，拦截器中三个重要的方法

**定义：**有两种方式

* 实现HandlerInterceptor接口
* 继承HandlerInterceptorAdapter

**配置：**

1. **<mvc:interceptors>**
2. <!--默认是对所有请求都拦截 -->
3. **<bean** id="myFirstInterceptor" class="com.atguigu.interceptor.MyFirstInterceptor"**>**
4. **</bean>**
5. <!-- 只针对部分请求拦截或者不拦截 -->
6. **<mvc:interceptor>**
7. **<mvc:mapping** path=" " **/>**  <!—指定拦截--**>**
8. **<mvc:exclude-mapping** path=””**/>** <!—指定不拦截--**>**
9. **<bean** class=" com.atguigu.interceptor.MySecondInterceptor " **/>** **</mvc:interceptor>**
10. **</mvc:interceptors>**

**拦截器中三个重要的方法：**

* preHandle
* postHandle
* afterCompletion

## 8.9 MyBatis中 #{}和${}的区别是什么？

#{}是预编译处理，${}是字符串替换；

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

Mybatis在处理${}时，就是把${}替换成变量的值；

使用#{}可以有效的防止SQL注入，提高系统安全性。

## 8.10 Mybatis 结果集的映射方式有几种，并分别解释每种映射方式如何使用。

自动映射 ，通过resultType来指定要映射的类型即可。

自定义映射 通过resultMap来完成具体的映射规则，指定将结果集中的哪个列映射到对象的哪个属性。

## 8.11简述MyBatis的单个参数、多个参数如何传递及如何取值。

MyBatis传递单个参数，如果是普通类型(String+8个基本)的，取值时在#{}中可以任意指定，如果是对象类型的，则在#{}中使用对象的属性名来取值

MyBatis传递多个参数，默认情况下，MyBatis会对多个参数进行封装Map，取值时在#{}可以使用0 1 2 .. 或者是param1 param2..

MyBatis传递多个参数，建议使用命名参数，在Mapper接口的方法的形参前面使用

@Param() 来指定封装Map时用的key. 取值时在#{}中使用@Param指定的key

## 8.12 MyBatis如何获取自动生成的(主)键值?

在<insert>标签中使用 useGeneratedKeys 和 keyProperty 两个属性来获取自动生成的主键值。

示例:

1. **<insert** id=”insertname” usegeneratedkeys=”true” keyproperty=”id”**>**
2. insert into names (name) values (#{name})
3. **</insert>**

## 8.13简述Mybatis的动态SQL，列出常用的6个标签及作用

动态SQL是MyBatis的强大特性之一 基于功能强大的OGNL表达式。

动态SQL主要是来解决查询条件不确定的情况，在程序运行期间，根据提交的条件动态的完成查询

常用的标签:

<if> : 进行条件的判断

<where>：在<if>判断后的SQL语句前面添加WHERE关键字，并处理SQL语句开始位置的AND 或者OR的问题

<trim>：可以在SQL语句前后进行添加指定字符 或者去掉指定字符.

<set>: 主要用于修改操作时出现的逗号问题

<choose> <when> <otherwise>：类似于java中的switch语句.在所有的条件中选择其一

<foreach>：迭代操作

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

不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复。

## 8.15 Mybatis 如何完成MySQL的批量操作,举例说明

MyBatis完成MySQL的批量操作主要是通过<foreache>标签来拼装相应的SQL语句.

例如:

1. **<insert** id="insertBatch" **>**
2. insert into tbl\_employee(last\_name,email,gender,d\_id) values
3. **<foreach** collection="emps" item="curr\_emp" separator=","**>**
4. (#{curr\_emp.lastName},#{curr\_emp.email},#{curr\_emp.gender},#{curr\_emp.dept.id})
5. **</foreach>**
6. **</insert>**

## 8.17简述Spring 中bean的作用域

总共有四种作用域:

* Singleton 单例的
* Prototype 原型的
* Request
* Session

## 8.18简述Spring中自动装配常用的两种装配模式

byName: 根据bean对象的属性名进行装配

byType： 根据bean对象的属性的类型进行装配,需要注意匹配到多个兼容类型的bean对象时，会抛出异常。

## 8.19请解释@Autowired注解的工作机制及required属性的作用

（1）首先会使用byType的方式进行自动装配，如果能唯一匹配，则装配成功，

如果匹配到多个兼容类型的bean, 还会尝试使用byName的方式进行唯一确定.

如果能唯一确定，则装配成功，如果不能唯一确定，则装配失败，抛出异常.

（2）默认情况下， 使用@Autowired标注的属性必须被装配，如果装配不了，也会抛出异常.

可以使用required=false来设置不是必须要被装配.

## 8.20 简述Springmvc中ContextLoaderListener的作用以及实现原理

**作用：**

ContextLoaderListener的作用是通过监听的方式在WEB应用服务器启动时将Spring的容器对象进行初始化.

**原理：**

ContextLoaderListener 实现了ServletContextListener接口，用于监听

ServletContext的创建，当监听到ServletContext创建时，在对应contextInitialized

方法中，将Spring的容器对象进行创建，并将创建好的容器对象设置到ServletContext域对象中，

目的是让各个组件可以通过ServletContext共享到Spring的容器对象

## 8.21简述Mybatis提供的两级缓存，以及缓存的查找顺序

（1）MyBatis的缓存分为一级缓存和 二级缓存。

一级缓存是SqlSession级别的缓存，默认开启。

二级缓存是NameSpace级别(Mapper)的缓存，多个SqlSession可以共享，使用时需要进行配置开启。

（2）缓存的查找顺序：二级缓存 => 一级缓存 => 数据库

## 8.22 简述Spring与Springmvc整合时，如何解决bean被创建两次的问题

Bean被创建两次的问题是在组建扫描的配置中指定Springmvc只负责扫描WEB相关的组件，Spring扫描除了Springmvc之外的组件。

## 8.23简述Spring与Mybatis整合时，主要整合的两个地方

（1）SqlSession创建的问题，通过SqlSessionFactoryBean来配置用于创建SqlSession的信息。例如: Mybatis的核心配置文件、Mapper映射文件、数据源等

（2）Mapper接口创建的问题， 使用MapperScannerConfigurer批量为MyBatis的Mapper接口生成代理实现类并将具体的对象交给Spring容器管理

## 8.24简述Spring声明式事务中@Transaction中常用的两种事务传播行为

通过propagation来执行事务的传播行为

REQUIRED：使用调用者的事务，如果调用者没有事务，则启动新的事务运行

REQUIRES\_NEW：将调用者的事务挂起，开启新的事务运行。

## 8.25简述@RequestMapping注解的作用，可标注的位置，常用的属性

（1）该注解的作用是用来完成请求 与 请求处理方法的映射

（2）该注解可以标注在类上或者是方法上

（3）常用的属性:

value: 默认属性， 用于指定映射的请求URL

method: 指定映射的请求方式

params: 指定映射的请求参数

headers: 指定映射的请求头信息

## 8.26简述Springmvc中处理模型数据的两种方式

* 使用ModelAndView 作为方法的返回值，将模型数据和视图信息封装到ModelAndView中
* 使用Map或者是Model 作为方法的形参，将模型数据添加到Map或者是Model中

## 8.27简述REST中的四种请求方式及对应的操作

GET 查询操作

POST 添加操作

DELETE 删除操作

PUT 修改操作

## 8.28简述视图和视图解析的关系及作用

* 视图是由视图解析器解析得到的。
* 视图解析器的作用是根据ModelAndView中的信息解析得到具体的视图对象
* 视图的作用是完成模型数据的渲染工作，最终完成转发或者是重定向的操作

## 8.29说出三个 常用的视图类

InternalResourceView

JstlView

RedirectView

## 8.30简述REST中HiddenHttpMethodFilter过滤器的作用

该过滤器主要负责转换客户端请求的方式，当浏览器的请求方式为POST，并且在请求中能通过 \_method获取到请求参数值。该过滤器就会进行请求方式的转换。

一般在REST中，都是将POST请求转换为对应的DELETE 或者是PUT

## 8.31简述Springmvc中如何返回JSON数据

Step1：在项目中加入json转换的依赖，例如jackson，fastjson，gson等

Step2：在请求处理方法中将返回值改为具体返回的数据的类型， 例如数据的集合类List<Employee>等

Step3：在请求处理方法上使用@ResponseBody注解

## 8.32简述如何在myBatis中的增删改操作获取到对数据库的影响条数

直接在Mapper接口的方法中声明返回值即可

## 8.33 Springmvc中的控制器的注解用哪个，可以是否用别的注解代替

使用@Controller注解来标注控制器，不能使用别的注解代替。

## 8.34如何在Springmvc中获取客户端提交的请求参数

直接在请求处理方法中声明对应的形参，也可以是用@RequestParam注解来具体指定将那些请求参数映射到方法中对应的形参。

## 8.35简述Springmvc中InternalResourceViewResolver解析器的工作机制

使用prefix + 方法的返回值 + suffix 生成一个物理视图路径。

## 8.36 Springmvc中如何完成重定向

在请求处理方法的返回值前面加 redirect: 前缀, 最终会解析得到RedirectView，RedirectView会完成重定向的操作。

## 8.37简述Spring中切面中常用的几种通知，并简单解释

前置通知 在目标方法执行之前执行

后置通知 在目标方法执行之后执行，不管目标方法有没有抛出异常

返回通知 在目标方法成功返回之后执行， 可以获取到目标方法的返回值

异常通知 在目标方法抛出异常后执行

环绕通知 环绕着目标方法执行

## 8.38解释MyBatis中 @Param注解的作用

通过该注解来指定Mybatis底层在处理参数时封装Map使用的key，方便在SQL映射文件中取参数。

## 8.39简述Mybatis中使用Mapper接口开发，如何完成Mapper接口与SQL映射文件、方法与SQL语句的绑定

Mapper接口与SQL映射文件绑定：SQL映射文件中的namespace的值指定成Mapper接口的全类名

接口中方法与SQL语句的绑定：SQL语句的id 指定成接口中的方法名。

## 8.40 SpringMVC的工作原理

（1）用户向服务器发送请求，请求被springMVC 前端控制器 DispatchServlet 捕获；

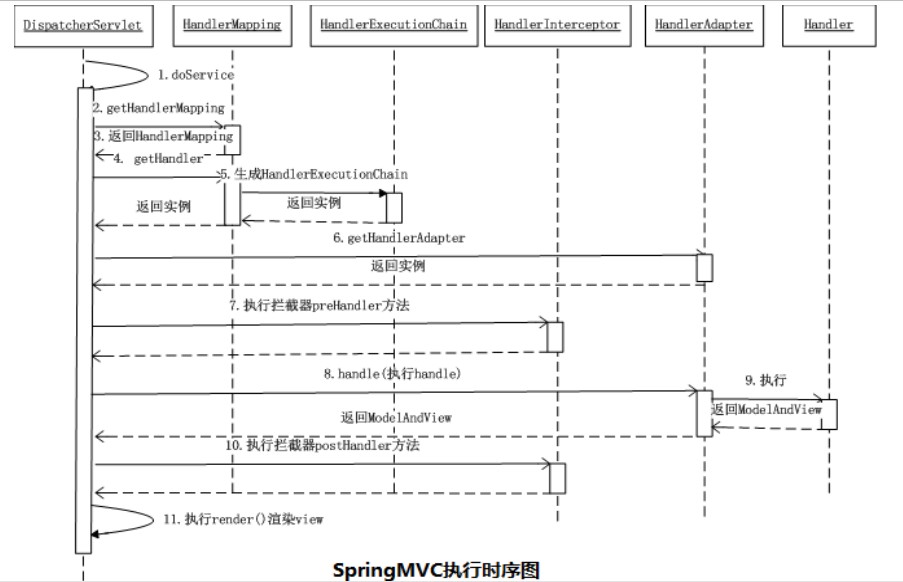
（2）DispatcherServle 对请求 URL 进行解析，得到请求资源标识符（URL），然后根据该 URL 调用 HandlerMapping将请求映射到处理器 HandlerExcutionChain；

（3）DispatchServlet 根据获得 Handler 选择一个合适的HandlerAdapter 适配器处理；

（4）Handler 对数据处理完成以后将返回一个 ModelAndView（）对象给 DisPatchServlet;

（5）Handler 返回的 ModelAndView() 只是一个逻辑视图并不是一个正式的视图， DispatcherSevlet 通过ViewResolver 试图解析器将逻辑视图转化为真正的视图View;

（6）DispatcherServle 通过 model 解析出 ModelAndView()中的参数进行解析最终展现出完整的 view 并返回给客户端;



## 8.41谈谈你对Spring 的理解

Spring 是一个开源框架，为简化企业级应用开发而生。Spring 可以是使简单的JavaBean 实现以前只有EJB 才能实现的功能。Spring 是一个 IOC 和 AOP 容器框架。

Spring 容器的主要核心是：

控制反转（IOC），传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 spring 开发模式中，spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用spring 提供的对象就可以了，这是控制反转的思想。

依赖注入（DI），spring 使用 javaBean 对象的 set 方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程，就是依赖注入的思想。

面向切面编程（AOP），在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用CGLIB 方式实现动态代理。

## 8.42 Spring中常用的设计模式

（1）代理模式——spring 中两种代理方式，若目标对象实现了若干接口，spring 使用jdk 的java.lang.reflect.Proxy类代理。若目标兑现没有实现任何接口，spring 使用 CGLIB 库生成目标类的子类。

（2）单例模式——在 spring 的配置文件中设置 bean 默认为单例模式。

（3）模板方式模式——用来解决代码重复的问题。

比如：RestTemplate、JmsTemplate、JpaTemplate

（4）工厂模式——在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用同一个接口来指向新创建的对象。Spring 中使用 beanFactory 来创建对象的实例。

## 8.43请描述一下Spring的事务管理

（1）声明式事务管理的定义：用在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。这样的好处是，事务管理不侵入开发的组件，具体来说，业务逻辑对象就不会意识到正在事务管理之中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可，这样维护起来极其方便。

基于 TransactionInterceptor 的声明式事务管理：两个次要的属性： transactionManager，用来指定一个事务治理器， 并将具体事务相关的操作请托给它； 其他一个是 Properties 类型的transactionAttributes 属性，该属性的每一个键值对中，键指定的是方法名，方法名可以行使通配符， 而值就是表现呼应方法的所运用的事务属性。

（2）基于 @Transactional 的声明式事务管理：Spring 2.x 还引入了基于 Annotation 的体式格式，具体次要触及@Transactional 标注。@Transactional 可以浸染于接口、接口方法、类和类方法上。算作用于类上时，该类的一切public 方法将都具有该类型的事务属性。

（3）编程式事物管理的定义：在代码中显式挪用 beginTransaction()、commit()、rollback()等事务治理相关的方法， 这就是编程式事务管理。Spring 对事物的编程式管理有基于底层 API 的编程式管理和基于 TransactionTemplate 的编程式事务管理两种方式。

## 8.45说出SpringMVC常用的5个注解？如何使用 SpringMVC完成JSON操作？

@Controller

@RequestMapping

@Resource和@Autowired

@PathVariable

@Repository

使用下面的注解完成json操作

@RequestBody

@ResponseBody

## 8.46Spring 支持的事务管理类型有哪些？你在项目中使用哪种方式？怎么理解全局事务和局部事务？

Spring支持编程式事务管理和声明式事务管理。Spring框架的用户选择声明式事务管理，因为这种方式和应用程序的关联较少，因此更加符合轻量级容器的概念。声明式事务管理要优于编程式事务管理，尽管在灵活性方面它弱于编程式事务管理，因为编程式事务允许你通过代码控制业务。

事务分为全局事务和局部事务。全局事务由应用服务器管理，需要底层服务器JTA支持（如WebLogic、WildFly等）。局部事务和底层采用的持久化方案有关，例如使用JDBC进行持久化时，需要使用Connetion对象来操作事务；而采用Hibernate进行持久化时，需要使用Session对象来操作事务。

这些事务的父接口都是PlatformTransactionManager。Spring的事务管理机制是一种典型的策略模式，PlatformTransactionManager代表事务管理接口，该接口定义了三个方法，该接口并不知道底层如何管理事务，但是它的实现类必须提供getTransaction()方法（开启事务）、commit()方法（提交事务）、rollback()方法（回滚事务）的多态实现，这样就可以用不同的实现类代表不同的事务管理策略。使用JTA全局事务策略时，需要底层应用服务器支持，而不同的应用服务器所提供的JTA全局事务可能存在细节上的差异，因此实际配置全局事务管理器是可能需要使用JtaTransactionManager的子类，如：WebLogicJtaTransactionManager（Oracle的WebLogic服务器提供）、UowJtaTransactionManager（IBM的WebSphere服务器提供）等。

## 8.47JDBC编程有哪些不足之处，MyBatis是如何解决这些问题的？

1） 数据库连接的创建、释放频繁造成系统资源浪费从而影响了性能，如果使用数据库连接池就可以解决这个问题。当然JDBC同样能够使用数据源。

解决：在SQLMapConfig.xml中配置数据连接池，使用数据库连接池管理数据库连接。

2） SQL语句在写代码中不容易维护，事件需求中SQL变化的可能性很大，SQL变动需要改变JAVA代码。解决：将SQL语句配置在mapper.xml文件中与java代码分离。

3） 向SQL语句传递参数麻烦，因为SQL语句的where条件不一定，可能多，也可能少，占位符需要和参数一一对应。解决：Mybatis自动将java对象映射到sql语句。

4） 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。解决：Mbatis自动将SQL执行结果映射到java对象。

## 8.48SqlMapConfig.xml中配置有哪些内容？

properties（属性）

settings（配置）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

dataSource（数据源）

mappers（映射器）

## 8.49简单的说一下MyBatis的一级缓存和二级缓存？

Mybatis首先去缓存中查询结果集，如果没有则查询数据库，如果有则从缓存取出返回结果集就不走数据库。Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象

Mybatis的二级缓存即查询缓存，它的作用域是一个mapper的namespace，即在同一个namespace中查询sql可以从缓存中获取数据。二级缓存是可以跨SqlSession的。

## 8.50Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？

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

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

# SpringBoot

## 9.1什么是 Spring Boot？

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。



## 9.2为什么要用 Spring Boot？

Spring Boot的优点

* 独立运行

Spring Boot而且内嵌了各种servlet容器，Tomcat、Jetty等，现在不再需要打成war包部署到容器中，Spring Boot只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个jar包内。

* 简化配置

spring-boot-starter-web启动器自动依赖其他组件，简少了maven的配置。除此之外，还提供了各种启动器，开发者能快速上手。

* 自动配置

Spring Boot能根据当前类路径下的类、jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。

* 无代码生成和XML配置

Spring Boot配置过程中无代码生成，也无需XML配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是Spring4.x的核心功能之一。

* 应用监控

Spring Boot提供一系列端点可以监控服务及应用，做健康检测。

## 9.3 Spring Boot有哪些缺点？

Spring Boot虽然上手很容易，但如果你不了解其核心技术及流程，所以一旦遇到问题就很棘手，而且现在的解决方案也不是很多，需要一个完善的过程。

## 9.4 Spring Boot 的核心配置文件有哪几个？它们的区别是什么？

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

**bootstrap配置文件的特性：**

* boostrap 由父 ApplicationContext 加载，比 applicaton 优先加载
* boostrap 里面的属性不能被覆盖

**bootstrap 配置文件有以下几个应用场景：**

* 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
* 一些固定的不能被覆盖的属性；
* 一些加密/解密的场景；

## 9.5 Spring Boot 的配置文件有哪几种格式？它们有什么区别？

.properties 和 .yml，它们的区别主要是书写格式不同。

1).properties

app.user.name = javastack

2).yml

app:  
  user:  
    name: javastack

另外，.yml 格式不支持 @PropertySource 注解导入配置。

## 9.6 Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

* @SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。
* @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，
  + 如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
* @ComponentScan：Spring组件扫描。

## 9.7开启 Spring Boot 特性有哪几种方式？

1. 继承spring-boot-starter-parent项目
2. **<parent>**
3. **<groupId>**
4. org.springframework.boot
5. **</groupId>**
6. **<artifactId>**
7. spring-boot-starter-parent
8. **</artifactId>**
9. **<version>**
10. 2.0.7.RELEASE
11. **</version>**
12. **</parent>**
13. 如果项目已经继承了其他父项目，则可以导入spring-boot-dependencies项目依赖
14. **<dependencyManagement>**
15. **<dependencies>**
16. **<dependency>**
17. **<groupId>**
18. org.springframework.boot
19. **</groupId>**
20. **<artifactId>**
21. spring-boot-dependencies
22. **</artifactId>**
23. **<version>**
24. 2.0.7.RELEASE
25. **</version>**
26. **<type>**
27. pom
28. **</type>**
29. **<scope>**
30. import
31. **</scope>**
32. **</dependency>**
33. **</dependencies>**
34. **</dependencyManagement>**

## 9.8 Spring Boot 需要独立的容器运行吗？

可以不需要，内置了 Tomcat/ Jetty 等容器。

## 9.9运行 Spring Boot 有哪几种方式？

1）打包命令或者放到容器中运行

2）用 Maven/ Gradle 插件运行

3）直接执行 main 方法运行

## 9.10 Spring Boot 自动配置原理是什么？

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

@EnableAutoConfiguration是实现自动配置的注解

@Configuration表示这是一个配置文件

从上面的@Import的类可以找到下面自动加载自动配置的映射。

|  |
| --- |
| 1. org.springframework.core.io.support.SpringFactoriesLoader.loadFactoryNames(Class<?>, ClassLoader) 2. public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) { 3. String factoryClassName = factoryClass.getName(); 4. try { 5. Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES\_RESOURCE\_LOCATION) : 6. lassLoader.getSystemResources(FACTORIES\_RESOURCE\_LOCATION)); 7. List<String> result = new ArrayList<String>(); 8. while (urls.hasMoreElements()) { 9. URL url = urls.nextElement(); 10. Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url)); 11. String factoryClassNames = properties.getProperty(factoryClassName); 12. result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames))); 13. } 14. return result; 15. } 16. catch (IOException ex) { 17. throw new IllegalArgumentException("Unable to load [" + factoryClass.getName() + 18. "] factories from location [" + FACTORIES\_RESOURCE\_LOCATION + "]", ex); 19. } 20. } |

这个方法会加载类路径及所有jar包下META-INF/spring.factories配置中映射的自动配置的类。

|  |
| --- |
| 1. /\*\* 2. \* The location to look for factories. 3. \* <p>Can be present in multiple JAR files. 4. \*/ 5. public static final String FACTORIES\_RESOURCE\_LOCATION = "META-INF/spring.factories"; |

查看Spring Boot自带的自动配置的包：

spring-boot-autoconfigure-1.5.6.RELEASE.jar，打开其中的META-INF/spring.factories文件会找到自动配置的映射。

|  |
| --- |
| 1. # Auto Configure 2. org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ 3. org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\ 4. org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ 5. org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\ 6. org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\ 7. org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\ 8. org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\ 9. org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\ 10. ... |

再来看看数据源自动配置的实现注解

|  |
| --- |
| 1. @Configuration 2. @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class }) 3. @EnableConfigurationProperties(DataSourceProperties.class) 4. @Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class }) 5. public class DataSourceAutoConfiguration { 6. ... |

@Configuration,@ConditionalOnClass就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

**9.11、你如何理解 Spring Boot 中的 Starters？**

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。

Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。具体请看这篇文章《[Spring Boot Starters启动器](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484257&idx=1&sn=f06b553cb8695448f7a7e6d07aacae70&scene=21#wechat_redirect)》。

## 9.12、如何在 Spring Boot 启动的时候运行一些特定的代码？

可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个 run 方法，具体请看这篇文章《[Spring Boot Runner启动器](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484366&idx=1&sn=7dc94038861fe9e10cdf132ffc83092f&scene=21#wechat_redirect)》。

## 9.13、Spring Boot 有哪几种读取配置的方式？

Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量，具体请看这篇文章《[Spring Boot读取配置的几种方式](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484575&idx=1&sn=56c88cd7283374345d891e85a800539b&scene=21#wechat_redirect)》。

## 9.14、Spring Boot 支持哪些日志框架？推荐和默认的日志框架是哪个？

Spring Boot 支持 Java Util Logging, Log4j2, Lockback 作为日志框架，如果你使用 Starters 启动器，Spring Boot 将使用 Logback 作为默认日志框架。

## 9.15、SpringBoot 实现热部署有哪几种方式？

主要有两种方式：

Spring Loaded

Spring-boot-devtools

Spring-boot-devtools 使用步骤如下：

1、引用devtools依赖

|  |
| --- |
| 1. <dependency> 2. <groupId>org.springframework.boot</groupId> 3. <artifactId>spring-boot-devtools</artifactId> 4. <optional>true</optional> 5. </dependency> |

2、自定义配置热部署，以下配置用于自定义配置热部署，可以不设置。

|  |
| --- |
| 1. # 热部署开关，false即不启用热部署 2. spring.devtools.restart.enabled: true 3. # 指定热部署的目录 4. #spring.devtools.restart.additional-paths: src/main/java 5. # 指定目录不更新 6. spring.devtools.restart.exclude: test/\*\* |

3、Intellij Idea修改，如果是idea，需要改以下两个地方：

1）、勾上自动编译或者手动重新编译

|  |
| --- |
| File > Settings > Compiler-Build Project automatically |

2）、注册

|  |
| --- |
| ctrl + shift + alt + / > Registry > 勾选Compiler autoMake allow when app running |

**注意事项**

**1、生产环境devtools将被禁用，如java -jar方式或者自定义的类加载器等都会识别为生产环境。**

**2、打包应用默认不会包含devtools，除非你禁用SpringBoot Maven插件的 excludeDevtools属性。**

**3、Thymeleaf无需配置 spring.thymeleaf.cache:false，devtools默认会自动设置，参考完整属性。**

## 9.16、你如何理解 Spring Boot 配置加载顺序？

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

1）properties文件；

2）YAML文件；

3）系统环境变量；

4）命令行参数；

等等……

|  |
| --- |
| 1、开发者工具 `Devtools` 全局配置参数；  2、单元测试上的 `@TestPropertySource` 注解指定的参数；  3、单元测试上的 `@SpringBootTest` 注解指定的参数；  4、命令行指定的参数，如 `java -jar springboot.jar --name="Java技术栈"`；  5、命令行中的 `SPRING\_APPLICATION\_JSONJSON` 指定参数, 如 `java -Dspring.application.json='{"name":"Java技术栈"}' -jar springboot.jar`  6、`ServletConfig` 初始化参数；  7、`ServletContext` 初始化参数；  8、JNDI参数（如 `java:comp/env/spring.application.json`）；  9、Java系统参数（来源：`System.getProperties()`）；  10、操作系统环境变量参数；  11、`RandomValuePropertySource` 随机数，仅匹配：`ramdom.\*`；  12、JAR包外面的配置文件参数（`application-{profile}.properties（YAML）`）  13、JAR包里面的配置文件参数（`application-{profile}.properties（YAML）`）  14、JAR包外面的配置文件参数（`application.properties（YAML）`）  15、JAR包里面的配置文件参数（`application.properties（YAML）`）  16、`@Configuration`配置文件上 `@PropertySource` 注解加载的参数；  17、默认参数（通过 `SpringApplication.setDefaultProperties` 指定）； |

数字小的优先级越高，即数字小的会覆盖数字大的参数值，我们来实践下，验证以上配置参数的加载顺序。

## 9.17、Spring Boot 如何定义多套不同环境配置？

提供多套配置文件，如：

applcation.properties  
application-dev.properties  
application-test.properties  
application-prod.properties

运行时指定具体的配置文件，具体请看这篇文章《[Spring Boot Profile 不同环境配置](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484369&idx=1&sn=1155fccb4fef1db88cb76fd17b1756d7&scene=21#wechat_redirect)》。

## 9.18 Spring Boot 可以兼容老 Spring 项目吗，如何做？

可以兼容，使用 @ImportResource 注解导入老 Spring 项目配置文件。

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

* 依赖 JDK 版本升级：2.x 里面的许多方法应用了 JDK 8 的许多高级新特性，至少需要 JDK 8 的支持；
* 第三方类库升：2.x 对第三方类库升级了所有能升级的稳定版本，例如：Spring Framework 5+、Tomcat 8.5+、Hibernate 5.2+、Thymeleaf 3+；
* 响应式 Spring 编程：2.x 通过启动器和自动配置全面支持 Spring 的响应式编程，响应式编程是完全异步和非阻塞的，它是基于事件驱动模型，而不是传统的线程模型；
* 连接池：2.x 默认使用 HikariCP 连接池；
* json：提供了一个 spring-boot-starter-json 启动器对 JSON 读写的支持；
* Quartz：2.x 提供了一个 spring-boot-starter-quartz 启动器对定时任务框架 Quartz 的支持；
* HTTP/2 支持：提供对HTTP/2 的支持，如：Tomcat, Undertow, Jetty；
* Actuator加强：在 2.x 中，对执行器端点进行了许多改进，所有的 HTTP 执行端点现在都暴露在 /actuator路径下，并对 JSON 结果集也做了改善。

## 9.20Spring Boot、Spring MVC 和 Spring 有什么区别？

SpringFrame

      SpringFramework 最重要的特征是依赖注入。所有 SpringModules（多模块） 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

SpringMVC

      Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServelet，MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。

SpringBoot

Spring 和 SpringMVC 的问题在于需要配置大量的参数。

|  |
| --- |
| <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix">   <value>/WEB-INF/views/</value>   </property>  <property name="suffix">  <value>.jsp</value>  </property>  </bean>   <mvc:resources mapping="/webjars/\*\*" location="/webjars/"/> |

Spring Boot 通过一个自动配置和启动的项来目解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

## 9.21什么是 Spring Boot Stater ？

启动器是一套方便的依赖没描述符，它可以放在自己的程序中。你可以一站式的获取你所需要的 Spring 和相关技术，而不需要依赖描述符的通过示例代码搜索和复制黏贴的负载。

## 9.22什么是 Spring Data REST?

Spring Data TEST 可以用来发布关于 Spring 数据库的 HATEOAS RESTful 资源。

## 9.23RequestMapping 和 GetMapping 的不同之处在哪里？

RequestMapping 具有类属性的，可以进行 GET,POST,PUT 或者其它的注释中具有的请求方法。

GetMapping 是 GET 请求方法中的一个特例。它只是 ResquestMapping 的一个延伸，目的是为了提高清晰度。

# SpringCloud

## 10.1 Spring Boot和Spring 是什么关系

SpringBoot底层就是Spring，简化使用Spring的方式而已，多加了好多的自动配置

## 10.2Spring Boot和Spring Cloud是什么关系

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的开发工具；Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架； Spring Boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring Boot来实现，必须基于Spring Boot开发。

可以单独使用Spring Boot开发项目，但是Spring Cloud离不开 Spring Boot。

## 10.3 Eureka和zookeeper的区别

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP, 而Eureka则是AP。

**Zookeeper保证CP**

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

**Eureka保证AP**

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)

3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

**总结**

Eureka作为单纯的服务注册中心来说要比zookeeper更加“专业”，因为注册服务更重要的是可用性，我们可以接受短期内达不到一致性的状况。不过Eureka目前1.X版本的实现是基于servlet的java web应用，它的极限性能肯定会受到影响。期待正在开发之中的2.X版本能够从servlet中独立出来成为单独可部署执行的服务。

## 10.4springcloud如何实现服务的注册?

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

## 10.5eureka自我保护机制是什么?

自我保护机制的工作机制是如果在15分钟内超过85%的客户端节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，Eureka Server自动进入自我保护机制，此时会出现以下几种情况：

1、Eureka Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。  
2、Eureka Server仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上，保证当前节点依然可用。  
3、当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中。

因此Eureka Server可以很好的应对因网络故障导致部分节点失联的情况，而不会像ZK那样如果有一半不可用的情况会导致整个集群不可用而变成瘫痪。

## 10.6什么是服务熔断？什么是服务降级?

服务熔断的作用类似于我们家用的保险丝，当某服务出现不可用或响应超时的情况时，为了防止整个系统出现雪崩，

暂时停止对该服务的调用。

服务降级是从整个系统的负荷情况出发和考虑的，对某些负荷会比较高的情况，为了预防某些功能（业务场景）出现负荷过载或者响应慢的情况，在其内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的fallback（退路）错误处理信息。这样，虽然提供的是一个有损的服务，但却保证了整个系统的稳定性和可用性。

## 10.7Ribbon和Feign的区别？

1.Ribbon都是调用其他服务的，但方式不同。

2.启动类注解不同，Ribbon是@RibbonClient feign的是@EnableFeignClients

3.服务指定的位置不同，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。

4.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需要将调用的方法定义成抽象方法即可。

## 10.8Hystrix断路器能干嘛？

服务降级整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来

服务熔断熔断机制是应对雪崩效应的一种微服务链路保护机制。当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回”错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand。

服务限流

接近实时的监控除了隔离依赖服务的调用以外，Hystrix还提供了准实时的调用监控（Hystrix Dashboard），Hystrix会持续地记录所有通过Hystrix发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求多少成功，多少失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。Spring Cloud也提供了Hystrix Dashboard的整合，对监控内容转化成可视化界面。

## 10.9分布式配置中心能干嘛？

（1）集中管理配置文件不同环境不同配置，动态化的配置更新，分环境部署比如

dev/test/prod/beta/release

（2）运行期间动态调整 置，不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取配置自己的信息

（3）当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置将配置信息以REST接口的形式暴露

## 10.10微服务之间是如何通讯的？

微服务能解决了单体应用以及SOA带来的的问题，但是微服务使整个应用服务增多，服务间通讯更复杂，也会带来大量 的问题。比如单体如何拆分成多个微服务，团队间沟通更多，运维成本增高，分布式事务问题，依赖管理变得复杂，测试 更加困难，故障更难于定位等等。

微服务架构能够解决单体架构带来的问题。但是在微服务架构中，一个整体的程序被拆分为许多子微服务。每个微服务实现某一个单一的功能，但是这使得应用程序的应用服务变多，这就使得这些微服务程序之间的通信变得十分复杂。而在程序运行过程中存在着大量的业务逻辑，这往往需要各个微服务之间交换大量的数据。这就带来了许多微服务通信的问题。但是微服务之间是如何通信的呢？

1、客户端到服务端通信，API Gateway方法。

API Gateway是解决微服务通信的一个不错的方法。以客户端为例。一个客户端可以向多个微服务中的任意一个微服务发出请求。API Gateway负责请求转发、合成和协议转换。所有请求都要先经过API Gateway，然后再将请求转发到对应的微服务中。

2、进程通信IPC

这种通信不但可以实现一对一、一对多。还可以实现同步和异步请求。

## 10.11什么是 Swagger？你用 Spring Boot 实现了它吗？

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

## 10.11你所知道的微服务技术栈有哪些？请列举一二

服务注册与发现Eureka、Consul、Zookeeper等

服务调用Rest、RPC、gRPC

服务熔断器Hystrix、Envoy等

负载均衡Ribbon、Nginx等

服务接口调用(客户端调用服务的简化工具)Feign等

消息队列Kafka、RabbitMQ、ActiveMQ等

服务配置中心管理SpringCloudConfig、Chef等

服务路由（API网关）Zuul等

服务监控Zabbix、Nagios、Metrics、Spectator等

## 10.12Ribbon负载均衡能干嘛？

LB（负载均衡LB，即负载均衡(Load Balance)，在微服务或分布式集群中经常用的一种应用。负载均衡简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA。常见的负载均衡有软件Nginx，LVS，硬件 F5等。相应的在中间件，例如：dubbo和SpringCloud中均给我们提供了负载均衡，SpringCloud的负载均衡算法可以自定义。

集中式LB即在服务的消费方和提供方之间使用独立的LB设施(可以是硬件，如F5, 也可以是软件，如nginx), 由该设施负责把访问请求通过某种策略转发至服务的提供方；

进程内LB将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

注意： Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

## 10.13什么是SpringCloud Config分布式配置中心

SpringCloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。

# 第11章 Redis

## 11.1什么是Redis？

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。 Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。 Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

## 11.2 redis有哪些数据类型

|  |  |
| --- | --- |
| string | 字符串 |
| list | 可以重复的集合 |
| set | 不可以重复的集合 |
| hash | 类似于Map<String,String> |
| zset(sorted set） | 带分数的set |

## 11.3一个字符串类型的值能存储最大容量是多少？

512M

## 11.4 怎么理解Redis事务？

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

## 11.5 Redis事务相关的命令有哪几个？

MULTI、EXEC、DISCARD、WATCH

## 11.6 Redis key的过期时间和永久有效分别怎么设置？

EXPIRE和PERSIST命令。

## 11.7 Redis 集群如何选择数据库?

Redis 集群目前无法做数据库选择，默认在 0 数据库。

## 11.8 Redis持久化数据和缓存怎么做扩容？

如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。

如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。4、Redis主要消耗什么物理资源？

内存。

## 11.9 为什么Redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。 如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

## 11.10 Redis如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key,而是应该把这个用户的所有信息存储到一张散列表里面.

## 11.11缓存穿透

**缓存系统定义：**

按照KEY去查询VALUE，当KEY对应的VALUE一定不存在的时候并对KEY并发请求量很大的时候，就会对后端造成很大的压力。（查询一个必然不存在的数据。比如文章表，查询一个不存在的id，每次都会访问DB，如果有人恶意破坏，很可能直接对DB造成影响。）

由于缓存不命中，每次都要查询持久层。从而失去缓存的意义。

**解决方法：**

（1）缓存层缓 存空值。

缓存太多空值，占用更多空间。（优化：给个空值过 期时间）

存储层更新代码了，缓存层还是空值。（优化：后台设置时主动删除空值，并缓存把值进去）

（2）将数据库中所有的查询条件，放布隆过滤器中。当一个查询请求来临的时候，先经过布隆过滤器进行查，如果请求存在这个条件中，那么继续执行，如果不在，直接丢弃。

**备注 ：**

比如数据库中有10000个条件，那么布隆过滤器的容量size设置的要稍微比10000大一些，比如12000.

对于误判率的设置，根据实际项目，以及硬件设施来具体定。但一定不能设置为0，并且误判率设置的越小，哈希函数跟数组长度都会更多跟更长，那么对硬件，内存中间的要求就会相应 的高

private st atic BloomFilter<Inte ger> bloomFi lt er = BloomFilter.create(Funnels.integerFue l(), s ize, 000 01) ;

有了siz跟误判率，那么布隆过滤器会产相应的哈希函数跟数组。

综上：我们可以利用布隆过滤器，将redis缓存击穿制在一个可容的范围内。

## 11.12 哨兵模式

如果Master异常，则会进行Master-Slave切换，将其中一Slae作为Master，将之前的Master作为Slave

**下线：**

①主观下线：Subjectively Down，简称 SDOWN，指的是当前 Sentinel 实例对某个redis服务器做出的下线判断。

②客观下线：Objectively Down， 简称 ODOWN，指的是多个 Sentinel 实例在对Master Server做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，然后开启failover.

**工作原理：**

（1）每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令 ；

（2）如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线；

（3）如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态；

（4）当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线 ；

（5）在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令

（6）当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次 ；

（7）若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除；

若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除；

## 11.13 悲观锁

执行操作前假设当前的操作肯定（或有很大几率）会被打断（悲观）。基于这个假设，我们在做操作前就会把相关资源锁定，不允许自己执行期间有其他操作干扰。

Redis不支持悲观锁。Redis作为缓存服务器使用时，以操作为主，很少写操作，相应的操作被打断的几率较少。不采用悲观锁是为了防止降低性能。

## 11.14 乐观锁

执行操作前假设当前操作不会被打断（乐观）。基于这个假设，我们在做操作前不会锁定资源，万一发生了其他操作的干扰，那么本次操作将被放弃。

## 11.15 持久化

**（1）RDB持久化：**

每隔一段时间，将内存中的数据集写到磁盘

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。

**保存策略：**

save 9 00 1 900 秒内如果至少有 1 个 key 的值变化，则保存

save 300 10 300 秒内如果至少有 10 个 key 的值变化，则保存

save 60 1 0000 60 秒内如果至10000 个 key 的值变化，则保存

**（2）AOF 持久化: 以日志形式记录每个更新（(总结、改）操作**

Redis重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。

**保存策略：**

appendfsync always：每次产生一条新的修改数据的命令都执行保存操作；效率低，但是安全！

appendfsync everysec：每秒执行一次保存操作。如果在未保存当前秒内操作时发生了断电，仍然会导致一部分数据丢失（即1秒钟的数据）。

appendfsync no：从不保存，将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

缺点：

1 比起RDB占用更多的磁盘空间

2 恢复备份速度要慢

3 每次读写都同步的话，有一定的性能压力

4 存在个别Bug，造成恢复不能

**（3）选择策略：**

可读的日志文本，通过操作AOF

官方推荐：

如果对数据不敏感，可以选单独用RDB；不建议单独用AOF，因为可能出现Bug;如果只是做纯内存缓存，可以都不用

## 11.16 Redis提供了哪几种持久化方式？

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大。

如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

你也可以同时开启两种持久化方式，在这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

## 11.17如何选择合适的持久化方式？

一般来说， 如果想达到足以媲美PostgreSQL的数据安全性， 你应该同时使用两种持久化功能。如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。

有很多用户都只使用AOF持久化，但并不推荐这种方式：因为定时生成RDB快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比AOF恢复的速度要快，除此之外， 使用RDB还可以避免之前提到的AOF程序的bug。

## 11.18分布式Redis是前期做还是后期规模上来了再做好？为什么？

既然Redis是如此的轻量（单实例只使用1M内存）,为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让Redis以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的Redis实例从第一台机器迁移到第二台机器。

## 11.19 Redis的内存占用情况怎么样？

给你举个例子： 100万个键值对（键是0到999999值是字符串“hello world”）在我的32位的Mac笔记本上 用了100MB。同样的数据放到一个key里只需要16MB， 这是因为键值有一个很大的开销。 在Memcached上执行也是类似的结果，但是相对Redis的开销要小一点点，因为Redis会记录类型信息引用计数等等。

当然，大键值对时两者的比例要好很多。

64位的系统比32位的需要更多的内存开销，尤其是键值对都较小时，这是因为64位的系统里指针占用了8个字节。 但是，当然，64位系统支持更大的内存，所以为了运行大型的Redis服务器或多或少的需要使用64位的系统。

## 11.20都有哪些办法可以降低Redis的内存使用情况呢？

如果你使用的是32位的Redis实例，可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。

## 11.21查看Redis使用情况及状态信息用什么命令？

info

## 11.22 Redis的内存用完了会发生什么？

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以将Redis当缓存来使用配置淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

## 11.23 redis是单线程的，为什么那么快

1)完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

2)数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的

3)采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

4)使用多路I/O复用模型，非阻塞IO

5)使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

## 11.24 Redis是单线程的，如何提高多核CPU的利用率？

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的， 所以，如果你想使用多个CPU，你可以考虑一下分片（shard）。

## 11.25一个Redis实例最多能存放多少的keys？List、Set、Sorted Set他们最多能存放多少元素？

理论上Redis可以处理多达232的keys，并且在实际中进行了测试，每个实例至少存放了2亿5千万的keys。我们正在测试一些较大的值。

任何list、set、和sorted set都可以放232个元素。

换句话说，Redis的存储极限是系统中的可用内存值。

## 11.26 Redis常见性能问题和解决方案？

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

## 11.27 Redis相比memcached有哪些优势？

(1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型

(2) redis的速度比memcached快很多

1. redis可以持久化其数据

## 11.28 MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

## 11.29 Redis有哪几种数据淘汰策略？

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

## 11.30 Redis集群方案应该怎么做？都有哪些方案？

（1）twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通redis无任何区别，设置好它下属的多个redis实例后，使用时在本需要连接redis的地方改为连接twemproxy，它会以一个代理的身份接收请求并使用一致性hash算法，将请求转接到具体redis，将结果再返回twemproxy。使用方式简便(相对redis只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy自身单端口实例的压力，使用一致性hash后，对redis节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

（2）codis，目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新hash节点。

（3）redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

（4）在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key 进行hash计算，然后去对应的redis实例操作数据。 这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

## 11.31说说Redis哈希槽的概念？

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

## 11.32 Redis集群最大节点个数是多少？

16384个。

## 11.33 怎么测试Redis的连通性？

ping

## 11.34修改配置不重启Redis会实时生效吗？

针对运行实例，有许多配置选项可以通过 CONFIG SET 命令进行修改，而无需执行任何形式的重启。 从 Redis 2.2 开始，可以从 AOF 切换到 RDB 的快照持久性或其他方式而不需要重启 Redis。检索 ‘CONFIG GET \*’ 命令获取更多信息。

但偶尔重新启动是必须的，如为升级 Redis 程序到新的版本，或者当你需要修改某些目前 CONFIG 命令还不支持的配置参数的时候。

## 11.35 Redis有哪些适合的场景？

（1）会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

（2）全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

（4）排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

（5）发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

# MQ

## 12.1 ActiveMQ 如果消息发送失败怎么办？

Activemq 有两种通信方式，点到点形式和发布订阅模式。

如果是点到点模式的话，如果消息发送不成功此 消息默认会保存到 activemq 服务端知道有消费者将其消费，所以此时消息是不会丢失的。

如果是发布订阅模式的通信方式，默认情况下只通知一次，如果接收不到此消息就没有了。这种场景只适 用于对消息送达率要求不高的情况。如果要求消息必须送达不可以丢失的话，需要配置持久订阅。每个订阅端定义一个 id， 在订阅是向 activemq 注册。发布消息和接收消息时需要配置发送模式为持久化。此时 如果客户端接收不到消息，消息会持久化到服务端，直到客户端正常接收后为止。

## 12.2如何使用ActiveMQ 解决分布式事务？

在互联网应用中，基本都会有用户注册的功能。在注册的同时，我们会做出如下操作：

收集用户录入信息，保存到数据库向用户的手机或邮箱发送验证码等等…

如果是传统的集中式架构，实现这个功能非常简单：开启一个本地事务，往本地数据库中插入一条用户数据，发送验证码，提交事物。

但是在分布式架构中，用户和发送验证码是两个独立的服务，它们都有各自的数据库，那么就不能通过本地事物保证操作的原子性。这时我们就需要用到 ActiveMQ（消息队列）来为我们实现这个需求。

在用户进行注册操作的时候，我们为该操作创建一条消息，当用户信息保存成功时，把这条消息发送到消息队列。验证码系统会监听消息，一旦接受到消息，就会给该用户发送验证码。

## 12.3 如何防止ActiveMQ消息重复发送？

解决方法很简单：增加消息状态表。通俗来说就是一个账本，用来记录消息的处理状态，每次处理消息之前，都去状态表中查询一次，如果已经有相同的消息存在，那么不处理，可以防止重复发送。

## 12.4使用消息队列有什么缺点？

优点就是在特殊场景下有其对应的好处，解耦、异步、削峰。

缺点有以下几个：

系统可用性降低

系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？如何保证消息队列的高可用，可以点击这里查看。

系统复杂度提高

硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，做好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的。

## 12.5消息队列的作用与使用场景？

消息队列在实际应用中常用的使用场景。异步处理，应用解耦，流量削锋和消息通讯四个场景

2.1异步处理

场景说明：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

（1）串行方式：将注册信息写入[数据库](http://lib.csdn.net/base/mysql)成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端



（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间



假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

因为CPU在单位时间内处理的请求数是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次（1000/150）。并行方式处理的请求量是10次（1000/100）

小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。因此架构改变后，系统的吞吐量提高到每秒20 QPS。比串行提高了3倍，比并行提高了两倍

2.2应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图



传统模式的缺点：

假如库存系统无法访问，则订单减库存将失败，从而导致订单失败

订单系统与库存系统耦合

如何解决以上问题呢？引入应用消息队列后的方案，如下图：



订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦

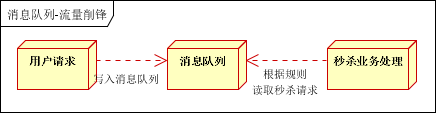
2.3流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

可以控制活动的人数

可以缓解短时间内高流量压垮应用



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面

秒杀业务根据消息队列中的请求信息，再做后续处理

2.4日志处理

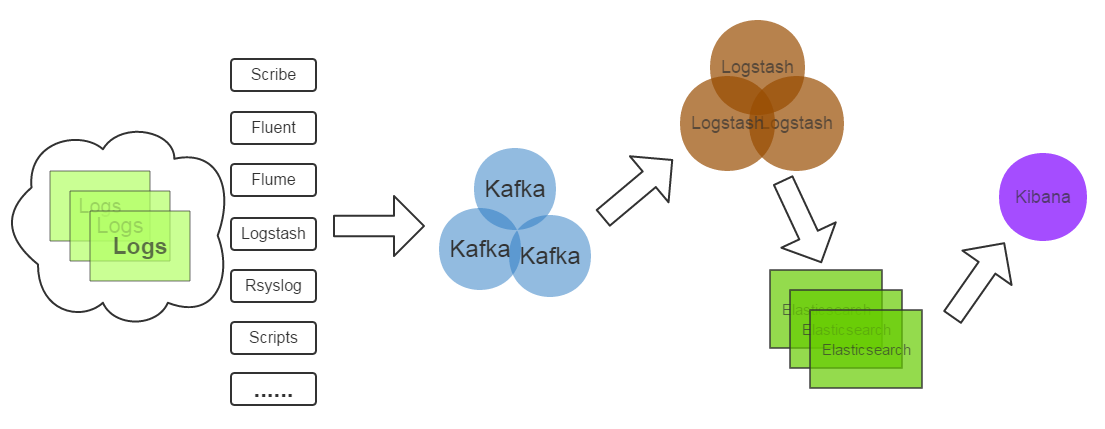
日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。

日志采集客户端，负责日志数据采集，定时写受写入Kafka队列

Kafka消息队列，负责日志数据的接收，存储和转发

日志处理应用：订阅并消费kafka队列中的日志数据

以下是新浪kafka日志处理应用案例：转自（http://cloud.51cto.com/art/201507/484338.htm）



(1)Kafka：接收用户日志的消息队列

(2)Logstash：做日志解析，统一成JSON输出给Elasticsearch

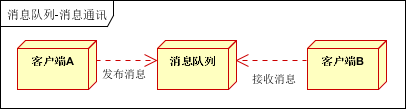
(3)Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能

(4)Kibana：基于Elasticsearch的数据可视化组件，超强的数据可视化能力是众多公司选择ELK stack的重要原因

2.5消息通讯

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等

点对点通讯：



客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：

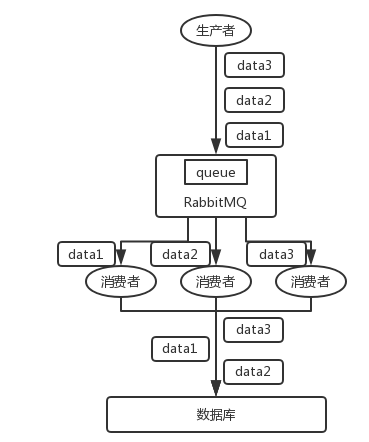
客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。模型为示意图，供参考。

## 12.6如何保证消息的顺序性？

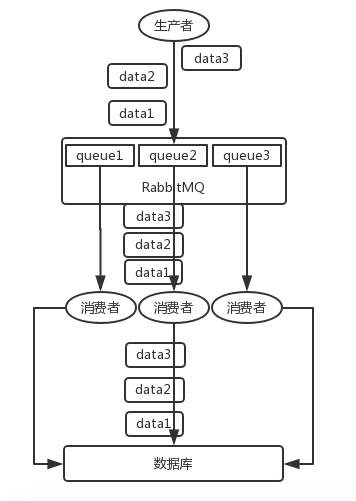
顺序会错乱的俩场景：

RabbitMQ：一个 queue，多个 consumer。比如，生产者向 RabbitMQ 里发送了三条数据，顺序依次是 data1/data2/data3，压入的是 RabbitMQ 的一个内存队列。有三个消费者分别从 MQ 中消费这三条数据中的一条，结果消费者2先执行完操作，把 data2 存入数据库，然后是 data1/data3。这不明显乱了。



解决方案：

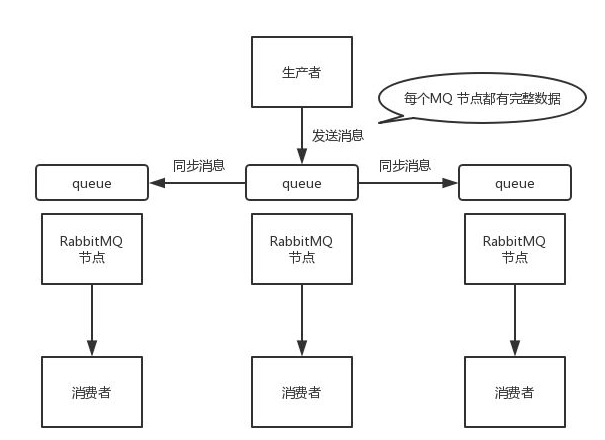
拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。



## 12.7如何保证消息队列的高可用？

镜像集群模式（高可用性）

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，在镜像集群模式下，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，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。



那么如何开启这个镜像集群模式呢？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，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，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。

这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。

## 12.8无法被路由的消息，去了哪里？

如果没有任何设置，无法路由的消息会被直接丢弃。

无法路由的情况：Routing key不正确

解决方案：

1.使用mandatory = true配合ReturnListener，实现消息回发

2.声明交换机时，指定备份交换机。

## 12.9消息队列过期失效问题

假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间的，也就是 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。

## 12.10消息队列满了怎么办？

 修复现有consumer的问题，并将其停掉。

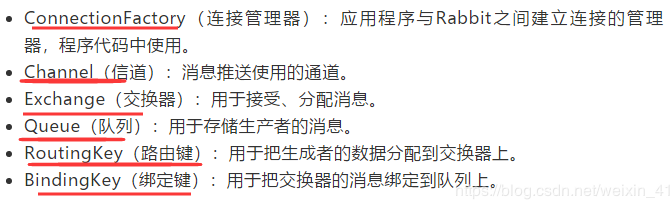
 重新创建一个容量更大的topic，比如patition是原来的10倍。

 编写一个临时consumer程序，消费原来积压的队列。该consumer不做任何耗时的操作，将消息均匀写入新创建的队列里。

 将修复好的consumer部署到原来10倍的机器上消费新队列。

 消息积压解决后，恢复原有架构。

## 12.11 RabbitMQ 有哪些重要的组件？



## 12.12 RabbitMQ 有几种广播类型？

三种广播模式：

fanout: 所有bind到此exchange的queue都可以接收消息（纯广播，绑定到RabbitMQ的接受者都能收到消息）；

direct: 通过routingKey和exchange决定的那个唯一的queue可以接收消息；

topic:  所有符合routingKey(此时可以是一个表达式)的routingKey所bind的queue可以接收消息；

## 12.14消息如何分发？

1.Round-Robin（轮询）

默认的策略，消费者轮流，平均地收到消息。

2.Fair dispatch（公平分发）

如果要实现根据消费者的处理能力来分发消息，给空闲的消费者发送更多消息，可以用basicQos(int prefetch\_count)来设置。prefetch\_count的含义：当消费者有多少条消息没有响应ACK时，不再给这个消费者发送消息。

## 12.15如何确保消息正确地发送至 RabbitMQ？ 如何确保消息接收方消费了消息？

发送方确认模式

将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。

一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。

如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（notacknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

## 12.16为什么不应该对所有的 message 都使用持久化机制？

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现“坑爹”问题。矛盾点在于，若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该 queue 的 message 将被 blackholed ；若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务器），则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。另外一种处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

## 12.16ActiveMQ和RabbitMQ的区别？

1.ActiveMq，传统的消息队列，使用Java语言编写。基于JMS（Java Message Service），采用多线程并发，资源消耗比较大。支持P2P和发布订阅两种模式。

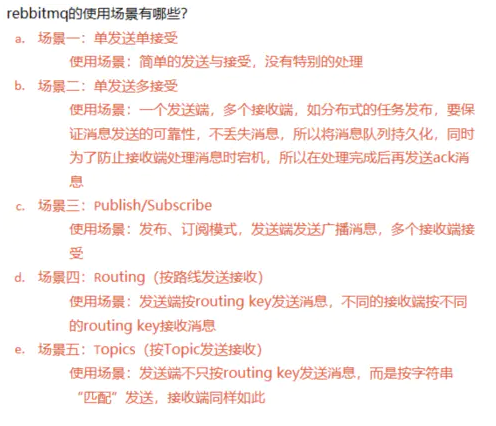
2.RabbitMQ，基于AMQP协议实现，支持多种场景，社区活跃量大。高性能，高可用，支持海量数据。

两者区别在于JMS和AMQP,

 JMS提供了两种消息模型，peer-2-peer(点对点)以及publish-subscribe（发布订阅）模型。当采用点对点模型时，消息将发送到一个队列，该队列的消息只能被一个消费者消费。而采用发布订阅模型时，消息可以被多个消费者消费。在发布订阅模型中，生产者和消费者完全独立，不需要感知对方的存在。

在AMQP中，消息路由（messagerouting）和JMS存在一些差别，在AMQP中增加了Exchange和binding的角色。producer将消息发送给Exchange，binding决定Exchange的消息应该发送到那个queue，而consumer直接从queue中消费消息。queue和exchange的bind有consumer来决定。

## 12.17 RabbitMQ 中的 broker 是指什么？cluster 又是指什么？

broker 是指一个或多个 erlang node 的逻辑分组，且 node 上运行着 RabbitMQ 应用程序。cluster 是在 broker 的基础之上，增加了 node 之间共享元数据的约束。  


## 12.18什么是元数据？元数据分为哪些类型？包括哪些内容？与 cluster 相关的元数据有哪些？元数据是如何保存的？元数据在 cluster 中是如何分布的？

在非 cluster 模式下，元数据主要分为 Queue 元数据（queue 名字和属性等）、Exchange 元数据（exchange 名字、类型和属性等）、Binding 元数据（存放路由关系的查找表）、Vhost 元数据（vhost 范围内针对前三者的名字空间约束和安全属性设置）。

在 cluster 模式下，还包括 cluster 中 node 位置信息和 node 关系信息。元数据按照 erlang node 的类型确定是仅保存于 RAM 中，还是同时保存在 RAM 和 disk 上。元数据在 cluster 中是全 node 分布的。

## 12.19 RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？

可以认为是无限制，因为限制取决于机器的内存，但是消息过多会导致处理效率的下降。

## 12.20 RabbitMQ 概念里的 channel、exchange 和 queue 这些东西是逻辑概念，还是对应着进程实体？这些东西分别起什么作用？

queue 具有自己的 erlang 进程；exchange 内部实现为保存 binding 关系的查找表；

channel 是实际进行路由工作的实体，即负责按照 routing\_key 将 message 投递给queue 。由 AMQP 协议描述可知，channel 是真实 TCP 连接之上的虚拟连接，所有AMQP 命令都是通过 channel 发送的，且每一个 channel 有唯一的 ID。一个 channel 只能被单独一个操作系统线程使用，故投递到特定 channel 上的 message 是有顺序的。但一个操作系统线程上允许使用多个 channel 。channel 号为 0 的 channel 用于处理所有对于当前 connection 全局有效的帧，而 1-65535 号 channel 用于处理和特定 channel 相关的帧。AMQP 协议给出的 channel 复用模型如下其中每一个 channel 运行在一个独立的线程上，多线程共享同一个 socket。

## 12.21消息在什么时候会变成Dead Letter（死信）？

1.消息被拒绝并且没有设置重新入队：（NACK || Reject）&&requeue == false

2.消息过期（消息或者队列的TTL设置）

3.消息堆积，并且队列达到最大长度，先入队的消息会变成DL。

可以在声明队列的时候，指定一个Dead Letter Exchange，来实现Dead Letter的转发。

## 12.22 RabbitMQ如何实现延时队列？

利用TTL（队列的消息存活时间或消息的存活时间），加上死信交换机。

当然还有一种方式就是先保存到数据库，用调度器扫描发送（时间不够精准）

## 12.23 SpringBoot中，bean还没有初始化好，消费者就开始监听取消息，导致空指针异常，怎么让消费者在容器启动完毕后才开始监听？

RabbitMQ中有一个auto\_startup参数，可以控制是否在容器启动时就监听启动

全局参数：

spring.rabbitmq.listener.auto-startup=true ##默认是true

自定义容器，容器可以应用到消费者：

// 默认是true

factory.setAutoStartup(true);

消费者单独设置

@RabbitListener( queues = "${com.gupaoedu.thirdqueue}" ,autoStartup = "false")

# ElasticSearch

## 13.1什么是ElasticSearch？

Elasticsearch是一个基于Lucene的搜索引擎。它提供了具有HTTP Web界面和无架构JSON文档的分布式，多租户能力的全文搜索引擎。Elasticsearch是用Java开发的，根据Apache许可条款作为开源发布。

## 13.2 Elasticsearch中的倒排索引是什么？

倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。

## 13.3 elasticsearch 了解多少，说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段。

ES 集群架构 13 个节点，索引根据通道不同共 20+索引，根据日期，每日递增 20+，索引：10 分片，每日递增 1 亿+数据，每个通道每天索引大小控制：150GB 之内。

仅索引层面调优手段：

1.1、设计阶段调优

（1）根据业务增量需求，采取基于日期模板创建索引，通过 roll over API 滚动索引；

（2）使用别名进行索引管理；

（3）每天凌晨定时对索引做 force\_merge 操作，以释放空间；

（4）采取冷热分离机制，热数据存储到 SSD，提高检索效率；冷数据定期进行 shrink操作，以缩减存储；

（5）采取 curator 进行索引的生命周期管理；

（6）仅针对需要分词的字段，合理的设置分词器；

（7）Mapping 阶段充分结合各个字段的属性，是否需要检索、是否需要存储等。……..

1.2、写入调优

（1）写入前副本数设置为 0；

（2）写入前关闭 refresh\_interval 设置为-1，禁用刷新机制；

（3）写入过程中：采取 bulk 批量写入；

（4）写入后恢复副本数和刷新间隔；

（5）尽量使用自动生成的 id。

1.3、查询调优

（1）禁用 wildcard；

（2）禁用批量 terms（成百上千的场景）；

（3）充分利用倒排索引机制，能 keyword 类型尽量 keyword；

（4）数据量大时候，可以先基于时间敲定索引再检索；

（5）设置合理的路由机制。

1.4、其他调优

部署调优，业务调优等。

上面的提及一部分，面试者就基本对你之前的实践或者运维经验有所评估了。

## 13.4 elasticsearch 索引数据多了怎么办，如何调优，部署

1 动态索引层面

基于模板+时间+rollover api 滚动创建索引，举例：设计阶段定义：blog 索引的模板格式为：blog\_index\_时间戳的形式，每天递增数据。这样做的好处：不至于数据量激增导致单个索引数据量非常大，接近于上线 2 的32 次幂-1，索引存储达到了 TB+甚至更大。

一旦单个索引很大，存储等各种风险也随之而来，所以要提前考虑+及早避免。

2 存储层面

冷热数据分离存储，热数据（比如最近 3 天或者一周的数据），其余为冷数据。

对于冷数据不会再写入新数据，可以考虑定期 force\_merge 加 shrink 压缩操作，节省存储空间和检索效率。

3 部署层面

一旦之前没有规划，这里就属于应急策略。

结合 ES 自身的支持动态扩展的特点，动态新增机器的方式可以缓解集群压力，注意：如果之前主节点等规划合理，不需要重启集群也能完成动态新增的。

## 13.5、elasticsearch 是如何实现 master 选举的

前置前提：

（1）只有候选主节点（master：true）的节点才能成为主节点。

（2）最小主节点数（min\_master\_nodes）的目的是防止脑裂。

核对了一下代码，核心入口为 findMaster，选择主节点成功返回对应 Master，否则返回 null。选举流程大致描述如下：

第一步：确认候选主节点数达标，elasticsearch.yml 设置的值

discovery.zen.minimum\_master\_nodes；

第二步：比较：先判定是否具备 master 资格，具备候选主节点资格的优先返回；

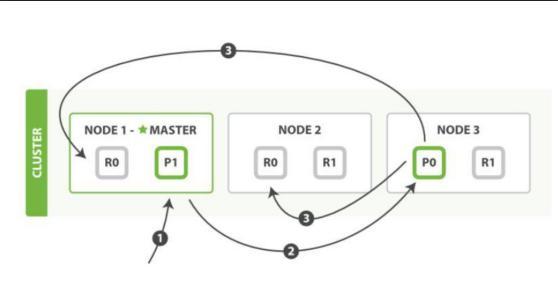
若两节点都为候选主节点，则 id 小的值会主节点。注意这里的 id 为 string 类型。

## 13.6详细描述一下 Elasticsearch 索引文档的过程

这里的索引文档应该理解为文档写入 ES，创建索引的过程。

文档写入包含：单文档写入和批量 bulk 写入，这里只解释一下：单文档写入流程。

记住官方文档中的这个图。



第一步：客户写集群某节点写入数据，发送请求。（如果没有指定路由/协调节点，请求的节点扮演路由节点的角色。）

第二步：节点 1 接受到请求后，使用文档\_id 来确定文档属于分片 0。请求会被转到另外的节点，假定节点 3。因此分片 0 的主分片分配到节点 3 上。

第三步：节点 3 在主分片上执行写操作，如果成功，则将请求并行转发到节点 1和节点 2 的副本分片上，等待结果返回。所有的副本分片都报告成功，节点 3 将向协调节点（节点 1）报告成功，节点 1 向请求客户端报告写入成功。

## 13.7详细描述一下 Elasticsearch 搜索的过程？

搜索拆解为“query then fetch” 两个阶段。

query 阶段的目的：定位到位置，但不取。

步骤拆解如下：

（1）假设一个索引数据有 5 主+1 副本 共 10 分片，一次请求会命中（主或者副本分片中）的一个。

（2）每个分片在本地进行查询，结果返回到本地有序的优先队列中。

（3）第 2）步骤的结果发送到协调节点，协调节点产生一个全局的排序列表。

fetch 阶段的目的：取数据。

路由节点获取所有文档，返回给客户端。

## 13.8 Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法

（1）关闭缓存 swap;

（2）堆内存设置为：Min（节点内存/2, 32GB）;

（3）设置最大文件句柄数；

（4）线程池+队列大小根据业务需要做调整；

（5）磁盘存储 raid 方式——存储有条件使用 RAID10，增加单节点性能以及避免单节点存储故障。

## 13.9 Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办？

（1）当集群 master 候选数量不小于 3 个时，可以通过设置最少投票通过数量（discovery.zen.minimum\_master\_nodes）超过所有候选节点一半以上来解决脑裂问题；

（2）当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data节点，避免脑裂问题。

## 13.10对于 GC 方面，在使用 Elasticsearch 时要注意什么？

（1）倒排词典的索引需要常驻内存，无法 GC，需要监控 data node 上 segmentmemory 增长趋势。

（2）各类缓存，field cache, filter cache, indexing cache, bulk queue 等等，要设置合理的大小，并且要应该根据最坏的情况来看 heap 是否够用，也就是各类缓存全部占满的时候，还有 heap 空间可以分配给其他任务吗？避免采用 clear cache等“自欺欺人”的方式来释放内存。

（3）避免返回大量结果集的搜索与聚合。确实需要大量拉取数据的场景，可以采用scan & scroll api 来实现。

（4）cluster stats 驻留内存并无法水平扩展，超大规模集群可以考虑分拆成多个集群通过 tribe node 连接。

（5）想知道 heap 够不够，必须结合实际应用场景，并对集群的 heap 使用情况做持续的监控。

（6）根据监控数据理解内存需求，合理配置各类circuit breaker，将内存溢出风险降低到最低

## 13.11在并发情况下，Elasticsearch 如果保证读写一致？

（1）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；

（2）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。

（3）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。

# 第 14章 Dubbo

## 14.1 Dubbo 的连接方式有哪些？

Dubbo 的客户端和服务端有三种连接方式，分别是：广播，直连和使用 zookeeper 注册中心。

## 14.2 Dubbo 的容错机制有哪些？

（1）Failover Cluster 模 式

失败自动切换，当出现失败，重试其它服务器。(默认) 2）Failfast Cluster

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

（2）Failsafe Cluster

失败安全，出现异常时，直接忽略。 通常用于写入审计日志等操作。

（3）Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。 通常用于消息通知操作。

（4）Forking Cluster

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

（5）Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0 开始支持) 通常用于通知所有提供者更新缓存或日志等本地资源信息。

总结： 在实际应用中查询语句容错策略建议使用默认Failover Cluster ，而增删改建议使用 Failfast Cluster 或者 使用 Failover Cluster（retries=”0”） 策略 防止出现数据 重复添加等等其它问题！建议在设计接口时候把查询接口方法单独做一个接口提供查询。

## 14.3使用dubbo 遇到过哪些问题？

（1）增加提供服务版本号和消费服务版本号

这个具体来说不算是一个问题,而是一种问题的解决方案,在我们的实际工作中会面临各种环境资源短缺的问题,也是很实际的问题,刚开始我们还可以提供一个服务进行相关的开发和测试,但是当有多个环境多个版本,多个任务的时候就不满足我们的需求,这时候我们可以通过给提供方增加版本的方式来区分.这样能够剩下很多的物理资源,同时为今后更换接口定义发布在线时，可不停机发布，使用版本号.

引用只会找相应版本的服务,例如：

<dubbo:serviceinterface=“com.xxx.XxxService”ref=“xxxService” version=“1.0” />

<dubbo:referenceid=“xxxService”interface=“com.xxx.XxxService” version=“1.0”/>

（2）dubbo reference 注解问题

@Reference 只能在 springbean 实例对应的当前类中使用，暂时无法在父类使用；如果确实要在父类声明一个引用，可通过配置文件配置 dubbo:reference，然后在需要引用的地方跟引用 springbean 一样就可以了.

（3）出现 RpcException: No provider available for remote service 异常，表示没有可用的服务提供者

* 检查连接的注册中心是否正确
* 到注册中心查看相应的服务提供者是否存在
* 检查服务提供者是否正常运行

（4）服务提供者没挂，但在注册中心里看不到

首先，确认服务提供者是否连接了正确的注册中心，不只是检查配置中的注册中心地址，而且要检查实际的网络连接。

其次，看服务提供者是否非常繁忙，比如压力测试，以至于没有CPU 片段向注册中心发送心跳，这种情况，减小压力，将自动恢复。

## 14.4为什么要用Dubbo？

因为是阿里开源项目，国内很多互联网公司都在用，已经经过很多线上考验。内部使用了 Netty、Zookeeper，保证了高性能高可用性。

使用 Dubbo 可以将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，可用于提高业务复用灵活扩展，使前端应用能更快速的响应多变的市场需求。

最重要的一点是，分布式架构可以承受更大规模的并发流量。

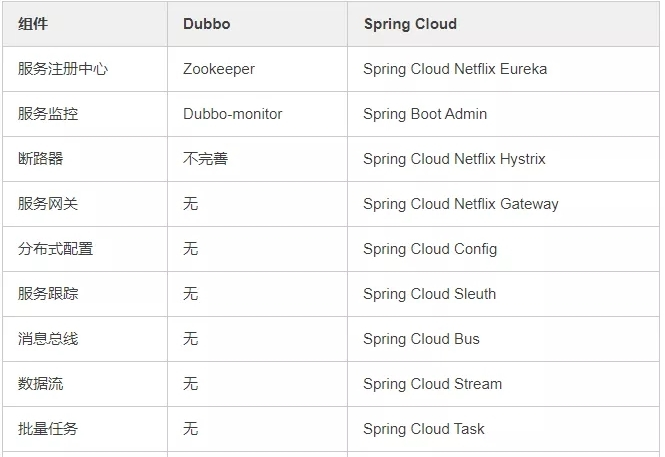
## 14.5 Dubbo 和 Spring Cloud 有什么区别？

两个没关联，如果硬要说区别，有以下几点。

1）通信方式不同

Dubbo 使用的是 RPC 通信，而 Spring Cloud 使用的是 HTTP RESTFul 方式。

2）组成部分不同



## 14.6 dubbo都支持什么协议，推荐用哪种？

dubbo://（推荐）

rmi://

hessian://

http://

webservice://

thrift://

memcached://

redis://

rest://

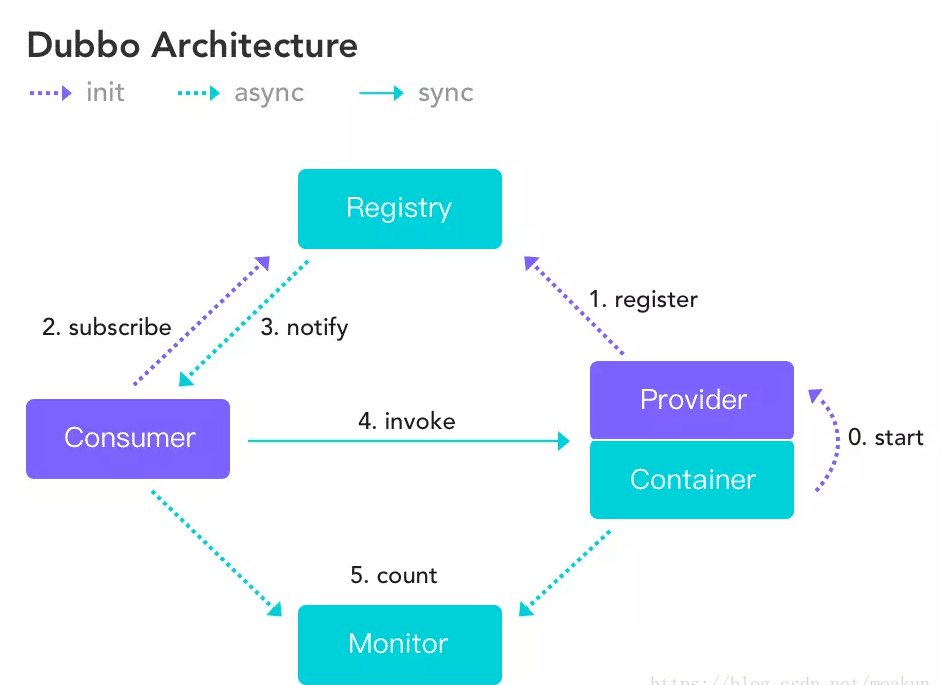
## 14.7 Dubbo需要 Web 容器吗？

不需要，如果硬要用 Web 容器，只会增加复杂性，也浪费资源。

## 14.8 Dubbo里面有哪几种节点角色？



## 14.9画一画服务注册与发现的流程图



## 14.10 Dubbo默认使用什么注册中心，还有别的选择吗？

推荐使用 Zookeeper 作为注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。

## 14.11 Dubbo有哪几种配置方式？

1）Spring 配置方式

2）Java API 配置方式

## 14.12 Dubbo 核心的配置有哪些？



## 14.13在 Provider 上可以配置的 Consumer 端的属性有哪些？

1）timeout：方法调用超时

2）retries：失败重试次数，默认重试 2 次

3）loadbalance：负载均衡算法，默认随机

4）actives 消费者端，最大并发调用限制

## 14.14 Dubbo启动时如果依赖的服务不可用会怎样？

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，默认 check="true"，可以通过 check="false" 关闭检查。

## 14.15 Dubbo推荐使用什么序列化框架，你知道的还有哪些？

Dubbo 默认使用 Netty 框架，也是推荐的选择，另外内容还集成有Mina、Grizzly。

## 14.16 Dubbo有哪几种负载均衡策略，默认是哪种？



## 14.17注册了多个同一样的服务，如果测试指定的某一个服务呢？

可以配置环境点对点直连，绕过注册中心，将以服务接口为单位，忽略注册中心的提供者列表。

## 14.18 Dubbo支持服务多协议吗？

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

## 14.19服务上线怎么兼容旧版本？

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

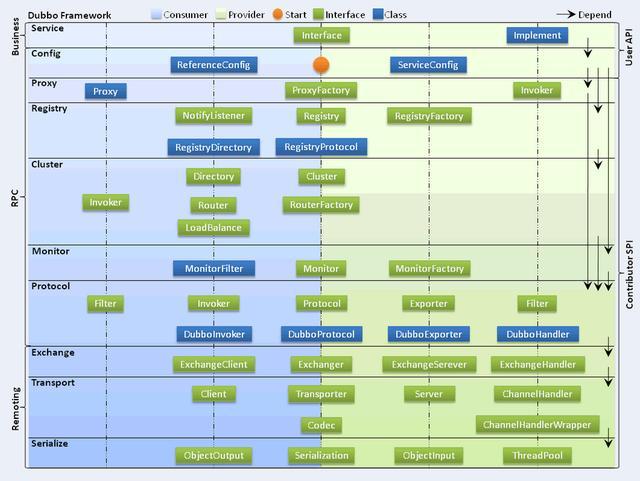
## 14.20 Dubbo可以对结果进行缓存吗？

可以，Dubbo 提供了声明式缓存，用于加速热门数据的访问速度，以减少用户加缓存的工作量。

## 14.21Dubbo支持服务降级吗？

Dubbo 2.2.0 以上版本支持。

## 14.22 Dubbo的架构设计



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

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

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

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

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

 监控层(Monitor):RPC调用次数和调用时间监控。

 远程调用层(Protocol):封将RPC调用，以Invocation和Result

为中心，扩展接口为 Protocol、Invoker 和 Exporter。

 信息交换层(Exchange):封装请求响应模式，同步转异步，以

Request 和 Response 为中心。

 网络传输层(Transport):抽象mina和netty为统一接口，以

Message 为中心。

# Zookeeper

## 15.1 ZooKeeper是什么？

ZooKeeper是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于Zookeeper实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master选举、分布式锁和分布式队列等功能。

Zookeeper保证了如下分布式一致性特性：

* 顺序一致性
* 原子性
* 单一视图
* 可靠性
* 实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理。对于写请求，这些请求会同时发给其他zookeeper机器并且达成一致后，请求才会返回成功。因此，随着zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是zookeeper中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper最新的zxid。

## 15.2 Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper允许客户端向服务端的某个Znode注册一个Watcher监听，当服务端的一些指定事件触发了这个Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

**工作机制：**

* 客户端注册watcher
* 服务端处理watcher
* 客户端回调watcher

Watcher特性总结：

（1）一次性  
无论是服务端还是客户端，一旦一个Watcher被触发，Zookeeper都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

（2）客户端串行执行  
客户端Watcher回调的过程是一个串行同步的过程。

（3）轻量

Watcher通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

客户端向服务端注册Watcher的时候，并不会把客户端真实的Watcher对象实体传递到服务端，仅仅是在客户端请求中使用boolean类型属性进行了标记。

watcher event异步发送watcher的通知事件从server发送到client是异步的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于Zookeeper本身提供了ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper只能保证最终的一致性，而无法保证强一致性。

注册watcher getData、exists、getChildren

触发watcher create、delete、setData

当一个客户端连接到一个新的服务器上时，watch将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。

## 15.3四种类型的数据节点 Znode

1、PERSISTENT-持久节点

除非手动删除，否则节点一直存在于 Zookeeper 上

2、EPHEMERAL-临时节点

临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与

zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

3、PERSISTENT\_SEQUENTIAL-持久顺序节点

基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

4、EPHEMERAL\_SEQUENTIAL-临时顺序节点

基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

## 15.4说一下 zk的应用场景

数据发布/订阅

（1）数据发布/订阅系统即所谓的配置中心，也就是发布者将数据发布到ZooKeeper的一个节点或者一系列节点上，提供订阅者进行数据订阅，从而实现动态更新数据的目的，实现配置信息的集中式管理和数据的动态更新。ZooKeeper采用的是推拉相结合的方式：客户端向服务器注册自己需要关注的节点，一旦该节点的数据发生改变，那么服务端就会向相应的客户端发送Wacher事件通知，客户端接收到消息通知后，需要主动到服务端获取最新的数据。

（2）实际系统开发过程中：我们可以将初始化配置信息放到节点上集中管理，应用在启动时都会主动到ZooKeeper服务端进行一次配置读取，同时在指定节点注册Watcher监听，主要配置信息一旦变更，订阅者就可以获取读取最新的配置信息。通常系统中需要使用一些通用的配置信息，比如机器列表信息、运行时的开关配置、数据库配置信息等全局配置信息，这些都会有以下3点特性：

　　1） 数据量通常比较小（通常是一些配置文件）

　　2） 数据内容在运行时会经常发生动态变化（比如数据库的临时切换等）

　　3） 集群中各机器共享，配置一致（比如数据库配置共享）。

（3）利用的ZooKeeper特性是：ZooKeeper对任何节点（包括子节点）的变更，只要注册Wacther事件（使用Curator等客户端工具已经被封装好）都可以被其它客户端监听

Master选举

（1）在一些读写分离的应用场景中，客户端写请求往往是由Master处理的，而另一些场景中，Master则常常负责处理一些复杂的逻辑，并将处理结果同步给集群中其它系统单元。比如一个广告投放系统后台与ZooKeeper交互，广告ID通常都是经过一系列海量数据处理中计算得到（非常消耗I/O和CPU资源的过程），那就可以只让集群中一台机器处理数据得到计算结果，之后就可以共享给整个集群中的其它所有客户端机器。

　　（2）利用ZooKeeper的特性：利用ZooKeeper的强一致性，即能够很好地保证分布式高并发情况下节点的创建一定能够保证全局唯一性，ZooKeeper将会保证客户端无法重复创建一个已经存在的数据节点，也就是说如果多个客户端请求创建同一个节点，那么最终一定只有一个客户端请求能够创建成功，这个客户端就是Master，而其它客户端注在该节点上注册子节点Wacther，用于监控当前Master是否存活，如果当前Master挂了，那么其余客户端立马重新进行Master选举。

（3）竞争成为Master角色之后，创建的子节点都是临时顺序节点，比如：\_c\_862cf0ce-6712-4aef-a91d-fc4c1044d104-lock-0000000001，并且序号是递增的。需要注意的是这里有"lock"单词，这说明ZooKeeper这一特性，也可以运用于分布式锁。

分布式锁

（1）对于排他锁：ZooKeeper通过数据节点表示一个锁，例如/exclusive\_lock/lock节点就可以定义一个锁，所有客户端都会调用create()接口，试图在/exclusive\_lock下创建lock子节点，但是ZooKeeper的强一致性会保证所有客户端最终只有一个客户创建成功。也就可以认为获得了锁，其它线程Watcher监听子节点变化（等待释放锁，竞争获取资源）。

　　　　 对于共享锁：ZooKeeper同样可以通过数据节点表示一个锁，类似于/shared\_lock/[Hostname]-请求类型（读/写）-序号的临时节点，比如/shared\_lock/192.168.0.1-R-0000000000

## 15.5 zookeeper负载均衡和nginx负载均衡区别

Nginx是著名的反向代理服务器，也被广泛的作为负载均衡服务器

ZooKeeper是分布式协调服务框架，有时也被用来做负载均衡

那么他们的区别是什么？如何选择呢？

下面从实际场景看下他们的关系

Nginx的负载均衡配置非常简单，把多个web server配置到nginx中，用户访问Nginx时，就会自动被分配到某个web server

upstream backend {

server 192.168.1.10;

server 192.168.1.11;

当网站规模变大，通常会进行服务拆分，各个服务独立部署，通过远程调用方式协同工作

为了保证稳定性，每个服务不会只使用一台服务器，也会作为一个集群存在，那么这个子集群同样需要一个负载均衡器，可以使用Nginx

到这里还是没有感觉有使用ZooKeeper的必要，因为使用Nginx完全没问题

但随着整个系统的演进， 服务 的数量会 增加 、每个服务集群中的 服务器 数量会增加

这时就会有一些小麻烦，例如

（1）配置维护的成本变高，因为节点太多

（2）单点故障的风险增加了，因为热点服务的访问量很高，如果这个服务集群内的负载均衡服务出现问题，这个服务将失效

第一个问题，可以通过自己开发程序解决，但只是降低复杂度，并没有实际解决

第二个问题，可以通过双机高可用部署方案，使用另一台nginx负载均衡服务器随时待命，只是成本较高

为了解决这些问题，就有人提出了使用ZooKeeper负载均衡的方案，之前就看到淘宝介绍过此类方案

ZooKeeper负载均衡的实现思路

把ZooKeeper作为一个服务的注册中心，在其中登记每个服务，每台服务器知道自己是属于哪个服务，在服务器启动时，自己向所属服务进行登记，这样，一个树形的服务结构就呈现出来了

服务的调用者到注册中心里面查找：能提供所需服务的服务器列表，然后自己根据负载均衡算法，从中选取一台服务器进行连接

调用者取到服务器列表后，就可以缓存到自己内部，省得下次再取，当服务器列表发生变化，例如某台服务器宕机下线，或者新加了服务器，ZooKeeper会自动通知调用者重新获取服务器列表

由于ZooKeeper并没有内置负载均衡策略，需要调用者自己实现，这个方案只是利用了ZooKeeper的树形数据结构、watcher机制等特性，把ZooKeeper作为服务的注册和变更通知中心，解决了Nginx负载均衡方案带来的问题

总结：如果Nginx服务挂掉，那么这台Nginx服务器下部署的反向代理服务均会失效, ZooKeeper作为服务的注册和变更通知中心，解决了Nginx负载均衡方案带来的问题

## 15.6Zookeeper对节点的watch监听通知是永久的吗？

不是。官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch

的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，

以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况

下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。

一般是客户端执行 getData(“/节点 A”,true)，如果节点 A 发生了变更或删除，

客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没

有设置 watch 事件，就不再给客户端发送。

在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我

只要最新的数据即可。

## 15.7集群支持动态添加机器吗？

其实就是水平扩容了，Zookeeper在这方面不太好。两种方式：

全部重启：关闭所有Zookeeper服务，修改配置之后启动。不影响之前客户端的会话。

逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5版本开始支持动态扩容。

## 15.8说一说选举算法及流程

目前有5台服务器，每台服务器均没有数据，它们的编号分别是1,2,3,4,5,按编号依次启动，它们的选择举过程如下：

服务器1启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器1的状态一直属于Looking(选举状态)。

服务器2启动，给自己投票，同时与之前启动的服务器1交换结果，由于服务器2的编号大所以服务器2胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是LOOKING。

服务器3启动，给自己投票，同时与之前启动的服务器1,2交换信息，由于服务器3的编号最大所以服务器3胜出，此时投票数正好大于半数，所以服务器3成为领导者，服务器1,2成为小弟。

服务器4启动，给自己投票，同时与之前启动的服务器1,2,3交换信息，尽管服务器4的编号大，但之前服务器3已经胜出，所以服务器4只能成为小弟。

服务器5启动，后面的逻辑同服务器4成为小弟。

## 15.9有哪几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

## 15.10集群中的机器角色都有哪些？

Leader

事务请求的唯一调度和处理者，保证集群事务处理的顺序性

集群内部各服务的调度者

Follower

处理客户端的非事务请求，转发事务请求给Leader服务器

参与事务请求Proposal的投票

参与Leader选举投票

Observer

3.3.0版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力

处理客户端的非事务请求，转发事务请求给Leader服务器

不参与任何形式的投票

## 15.11集群最少要几台机器，集群规则是怎样的

集群规则为2N+1台，N>0，即3台。

## 15.12集群如果有3台机器，挂掉一台集群还能工作吗？挂掉两台呢？

Zookeeper本身也是集群，推荐配置不少于3个服务器。Zookeeper自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个Follower宕机，还有2台服务器提供访问，因为Zookeeper上的数据是有多个副本的，数据并不会丢失；

如果是一个Leader宕机，Zookeeper会选举出新的Leader。

ZK集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3个节点的cluster可以挂掉1个节点(leader可以得到2票>1.5)

2个节点的cluster就不能挂掉任何1个节点了(leader可以得到1票<=1)

## 15.13说几个zookeeper常用的命令。

常用命令：ls get set create delete等。

## 15.14Zookeeper分布式锁

有了zookeeper的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的distribute\_lock 节点就释放出锁。

对于第二类， /distribute\_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，编号最小的获得锁，用完删除，依次方便。

## 15.15Zookeeper数据复制

Zookeeper作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处：

1、容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；

2、提高系统的扩展能力 ：把负载分布到多个节点上，或者增加节点来提高系统的负载能力；

3、提高性能：让客户端本地访问就近的节点，提高用户访问速度。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：

1、写主(WriteMaster) ：对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离；

2、写任意(Write Any)：对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对zookeeper来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立observer的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

## 15.16zookeeper是如何保证事务的顺序一致性的？

zookeeper采用了全局递增的事务Id来标识，所有的proposal（提议）都在被提出的时候加上了zxid，zxid实际上是一个64位的数字，高32位是epoch（时期; 纪元; 世; 新时代）用来标识leader周期，如果有新的leader产生出来，epoch会自增，低32位用来递增计数。当新产生proposal的时候，会依据数据库的两阶段过程，首先会向其他的server发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

# 第16章Git

## 16.1 reset 与 rebase, pull 与 fetch 的区别

git reset 不修改commit相关的东西，只会去修改.git目录下的东西。

git rebase 会试图修改你已经commit的东西，比如覆盖commit的历史等，但是不能使用rebase来修改已经push过的内容，容易出现兼容性问题。rebase还可以来解决内容的冲突，解决两个人修改了同一份内容，然后失败的问题。

git pull pull=fetch+merge,

使用git fetch是取回远端更新，不会对本地执行merge操作，不会去动你的本地的内容。 pull会更新你本地代码到服务器上对应分支的最新版本

## 16.2 git merge和git rebase的区别

git merge把本地代码和已经取得的远程仓库代码合并。

git rebase是复位基底的意思，gitmerge会生成一个新的节点，之前的提交会分开显示，而rebase操作不会生成新的操作，将两个分支融合成一个线性的提交。

## 16.3 git如何解决代码冲突

git stash

git pull

git stash pop

这个操作就是把自己修改的代码隐藏，然后把远程仓库的代码拉下来，然后把自己隐藏的修改的代码释放出来，让gie自动合并。

如果要代码库的文件完全覆盖本地版本。

git reset –hard

git pull

# 第17章Linux

## 17.1 Linux常用命令

|  |  |  |
| --- | --- | --- |
| 序号 | 命令 | 命令解释 |
| 1 | top | 查看内存 |
| 2 | df -h | 查看磁盘存储情况 |
| 3 | iotop | 查看磁盘IO读写(yum install iotop安装） |
| 4 | iotop -o | 直接查看比较高的磁盘读写程序 |
| 5 | netstat -tunlp | grep 端口号 | 查看端口占用情况 |
| 6 | uptime | 查看报告系统运行时长及平均负载 |
| 7 | ps aux | 查看进程 |

## 17.2如何查看所有java进程

grep是搜索关键字

>ps -ef | grep java

-aux 显示所有状态

>ps -aux | grep java

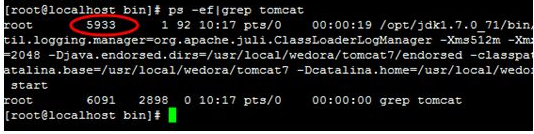
## 17.3如何杀掉某个服务的进程

kill 命令用于终止进程

-9 强迫进程立即停止

>kill -9 [PID]

这里pid需要用 ps -ef | grep 查询pid



## 17.4启动/停止服务

以启动Tomcat为例,先cd到启动的.sh文件目录

> cd /java/tomcat/bin

> ./startup.sh

停止Tomcat服务命令

>./shutdown.sh

## 17.5如何查看测试项目的日志

一般测试的项目里面，有个logs的目录文件，会存放日志文件，有个xxx.out的文件，可以用tail -f 动态实时查看后端日志

先cd 到logs目录(里面有xx.out文件)

>tail -f xx.out

这时屏幕上会动态实时显示当前的日志，ctr+c停止

## 17.6如何查看最近1000行日志

>tail -1000 xx.out

## 17.7 LINUX中如何查看某个端口是否被占用

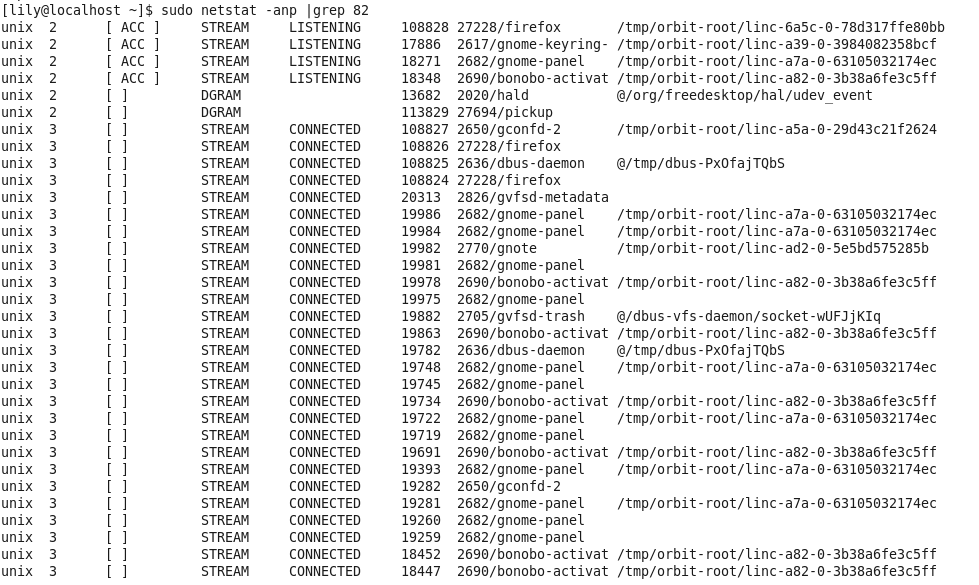
>netstat -anp | grep 端口号

IMG_256

图中主要看监控状态为LISTEN表示已经被占用，最后一列显示被服务mysqld占用，查看具体端口号，只要有如图这一行就表示被占用了

查看82端口的使用情况，如图

>netstat -anp |grep 82

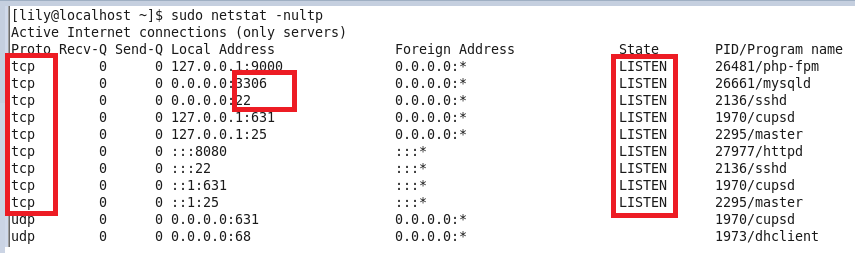


可以看出并没有LISTEN那一行，所以就表示没有被占用。此处注意，图中显示的LISTENING并不表示端口被占用，不要和LISTEN混淆哦，查看具体端口时候，必须要看到tcp，端口号，LISTEN那一行，才表示端口被占用了

## 17.7查看当前所有已经使用的端口情况

如图：

netstat -nultp（此处不用加端口号）



## 17.8如何查找一个文件大小超过5M的文件

>find . -type f -size +100M

## 17.9如果知道一个文件名称，怎么查这个文件在linux下的哪个目录

如：要查找tnsnames.ora文件

>find / -name tnsnames.ora

查到：

/opt/app/oracle/product/10.2/network/admin/tnsnames.ora

/opt/app/oracle/product/10.2/network/admin/samples/tnsnames.ora

还可以用locate 来查找

>locate tnsnames.ora

结果是：

/opt/app/oracle/product/10.2/hs/admin/tnsnames.ora.sample

/opt/app/oracle/product/10.2/network/admin/tnsnames.ora

/opt/app/oracle/product/10.2/network/admin/samples/tnsnames.ora

## 17.10 find查找文件

find / -name httpd.conf　　#在根目录下查找文件httpd.conf，表示在整个硬盘查找

find /etc -name httpd.conf　　#在/etc目录下文件httpd.conf

find /etc -name 'srm'　　#使用通配符(0或者任意多个)。表示在/etc目录下查找文件名中含有字符串‘srm’的文件

find . -name 'srm' 　　#表示当前目录下查找文件名开头是字符串‘srm’的文件

按照文件特征查找

find / -amin -10 　　# 查找在系统中最后10分钟访问的文件(access time)

find / -atime -2　　 # 查找在系统中最后48小时访问的文件

find / -empty 　　# 查找在系统中为空的文件或者文件夹

find / -group cat 　　# 查找在系统中属于 group为cat的文件

find / -mmin -5 　　# 查找在系统中最后5分钟里修改过的文件(modify time)

find / -mtime -1 　　#查找在系统中最后24小时里修改过的文件

find / -user fred 　　#查找在系统中属于fred这个用户的文件

find / -size +10000c　　#查找出大于10000000字节的文件(c:字节，w:双字，k:KB，M:MB，G:GB)

find / -size -1000k 　　#查找出小于1000KB的文件

## 17.11 vim（vi）编辑器

有命令模式、输入模式、末行模式三种模式。

* 命令模式：查找内容(/abc、跳转到指定行(20gg)、跳转到尾行(G)、跳转到首行(gg)、删除行(dd)、插入行(o)、复制粘贴(yy,p)
* 输入模式：编辑文件内容
* 末行模式：保存退出(wq)、强制退出(q!)、显示文件行号(set number)

在命令模式下，输入a或i即可切换到输入模式，输入冒号(:)即可切换到末行模式；在输入模式和末行模式下，按esc键切换到命令模式

# 第18章Nginx

## 18.1 请解释一下什么是 Nginx?

Nginx 是一个 web 服务器和反向代理服务器，用于 HTTP、HTTPS、SMTP、POP3 和 IMAP 协议。

## 18.2 请列举 Nginx 的一些特性。

Nginx 服务器的特性包括:

反向代理/L7 负载均衡器

嵌入式 Perl 解释器

动态二进制升级

可用于重新编写 URL，具有非常好的 PCRE 支持

## 18.3 请解释 Nginx 如何处理 HTTP 请求。

Nginx 使用反应器模式。主事件循环等待操作系统发出准备事件的信号，这样 数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。单个线程可 以提供数万个并发连接。

## 18.4 在 Nginx 中，如何使用未定义的服务器名称来阻止处理请求?

只需将请求删除的服务器就可以定义为:

Server {listen 80;server\_name “ “ ;return 444;

}

这里，服务器名被保留为一个空字符串，它将在没有“主机”头字段的情况下 匹配请求，而一个特殊的 Nginx 的非标准代码 444 被返回，从而终止连接。

## 18.5 使用“反向代理服务器”的优点是什么?

反向代理服务器可以隐藏源服务器的存在和特征。它充当互联网云和 web 服务 器之间的中间层。这对于安全方面来说是很好的，特别是当您使用 web 托管服务时。

## 18.6 请列举 Nginx 服务器的最佳用途。

Nginx 服务器的最佳用法是在网络上部署动态 HTTP 内容，使用 SCGI、WSGI 应 用程序服务器、用于脚本的 FastCGI 处理程序。它还可以作为负载均衡器。

## 18.7 请解释是否有可能将 Nginx 的错误替换为 502 错误、503?

502 =错误网关 503 =服务器超载

有可能，但是您可以确保 fastcgi\_intercept\_errors 被设置为 ON，并使用错 误页面指令。

Location / {fastcgi\_pass 127.0.01:9001;fastcgi\_intercept\_errors on;error\_page 502 =503/error\_page.html;#...}

## 18.8 请解释 ngx\_http\_upstream\_module 的作用是什么?

ngx\_http\_upstream\_module 用于定义可通过 fastcgi 传递、proxy 传递、uwsgi 传递、memcached 传递和 scgi 传递指令来引用的服务器组。

## 18.9 请陈述 stub\_status 和 sub\_filter 指令的作用是什么?

Stub\_status 指令:该指令用于了解 Nginx 当前状态的当前状态，如当前的活 动连接，接受和处理当前读/写/等待连接的总数

Sub\_filter 指令:它用于搜索和替换响应中的内容，并快速修复陈旧的数据。