# B简历：

# -------------------------------------------

|  |  |  |  |
| --- | --- | --- | --- |
| **基本情况** | | | |
| **姓名** | 亢佳浩 | 性别 | 男 |
| **出生日期** | 1997-07-29 | 属相 | 牛 |
| **现住地** | 秦庄家园 |  |  |
| **大学情况** | | | |
| **学校** | 郑州工商学院 | 专业 | 计算机科学与技术 |
| **入学时间** | 2014-8 | 毕业时间 | 2018-6 |
| **专业课** | **[java语言](https://www.baidu.com/s?wd=Java%E8%AF%AD%E8%A8%80&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)程序设计、Java[企业级开发](https://www.baidu.com/s?wd=%E4%BC%81%E4%B8%9A%E7%BA%A7%E5%BC%80%E5%8F%91&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)、数据库技术与应用、大型数据库处理技术、网页设计与开发、计算机网络与通讯、软件工程** | | |
| **学历** | **本科** | 学位 | **学士学位** |
| **学校地址** | 河南省郑州市郑东新区前程大道 | 校长 | 郭德欣 |
| **去学校的方式：** | **坐570路 地铁1号线** | | |
| **上小学日期：** | **2002年9月 6岁五年** | | |
| **上初中日期：** | **2007年9月 11岁四年** | | |
| **上高中日期：** | **2011年9月 15岁三年** | | |
| **上大学日期：** | **2014年9月 18岁四年** | | |
| **毕业时间：** | **2018年 6月 22岁** | | |
| **在校学生：** | **20000左右** | | |

|  |  |
| --- | --- |
| 第一家公司 ： | 浙江宏森科技有限公司（宏森科技） |
| **地址：** | **杭州滨江区聚光中心A座17层**（只说大厦名称）聚光中心A座17层 |
| **入职时间：** | **2017年底** |
| **离职时间：** | **2019年6月** |
| **公司性质：** | **民营** |
| **职位：** | **Java工程师** |
| **项目经理名字：** | **焦启阳** |
| **薪资方面：** | **毕业实习4k,三个月转正以后工资是6K，过完年涨到7K，年底涨薪到7.5k** |
| **路线：** | **骑电动车十五六（兴耀星澜之城 住得比较近）** |
| **公司规模：** | **50人左右 我们项目组8人 1个项目经理 2名前端，3名后端，2个测试，1个运维** |
| **离职前薪资** | **7.5k/税后 7k左右** |
|  |  |

|  |  |
| --- | --- |
|  |  |
| **第二家公司情况：** | 杭州联海(只说公司简称) |
| **公司地址**： | 杭州市 滨江区 华侨国际·先锋大厦 16楼 |
| **公司规模：** | **80人左右，我们项目组8人，一个项目经理，2名前端，1个测试，1个运维，其他都是后台开发** |
| **公司性质：** | **民营** |
| **公司行业：** | 计算机硬件、IT服务、计算机软件、互联网 |
| **去公司路线：** | **骑电动车二十分钟左右（兴耀星澜之城 住得比较近）一条直线** |
| **入职时间：** | **2019年6月底** |
| **离职时间：** | **2021年4月** |
| **职位：** | **Java工程师** |
| **项目经理名字：** | **刘学** |
| **税前：** | **入职10k，第一年涨1k,第二年底涨1.5k 税前共12.5k** |
| **税后：** | **11k左右** |
| **缴税：** | **1K左右(具体原因不清楚当时公司就是这样扣的税)** |
| **期望薪资：** | **15K** |
| **薪资的构成:** | **岗位工资(5K)+绩效工资(7.5K)** |

**面试常见问题：**

|  |
| --- |
| **你为什么从上家公司离职？** |
| **之前公司属于外包公司，最近公司接的项目比较少，没什么事情，太安逸了，想换一个新的平台去充实提升自己!** |
| |  | | --- | |  | | **你的优缺点是啥？** |   优点： 热爱生活，热爱工作，有团队意识，喜欢钻研。  缺点：责任心过强，总喜欢替自己和他人操心，写代码养成了略微的强迫症。 |
| **你交社保了吗？为啥没交？** |
| **没交，在上家公司的时候，人事说咱们公司的五险一金如果要上的话都是从自己工资里面扣的，当时感觉没啥必要也就没上。** |
| **你知道五险一金都是啥吗？** |
| **养老保险、医疗保险、失业保险、工伤保险、生育保险，和住房公积金。** |
| **你能接受加班吗** |
| **不是特别频繁的加班可以接受** |
| **你能接受出差吗** |
| **能** |
| 你期望薪资多少？你上家工资多少？税后拿到手的有多少？扣了多少钱的税？ |
| 期望薪资 15K 上家工资12.5K 税后11K左右 扣了1K左右的税 |
| 你的五年规划是啥？ |
| **先已熟悉公司业务为主，提升技术，尽可能向架构师和管理岗位努力。** |
| 是统招吗 是是本科吗 是能在学信网查到吗 我当时上的是民办大学，能在民教网查到！ |
| 你们公司还有什么福利？多少薪： 14薪  **车补**  200左右  **项目奖金**  7k-10k，做完项目，收完尾款的时候。  **年终奖**  这个就是2个月的工资，  一般是过年的时候和当月工资一块发  **电话补助**  200  **饭/餐补**  20/天  **电脑补**  200 |

**笔试问题：我工作这么多长的时间了，理论上的知识忘得差不多了，你和你们经理联系一下直接面试吧，不行的话那么我就去直接去下一家！**

**-----------------------------------------------------------------------------------------------------------------**

# ========（业务功能模块/核心技术点）【主动说】========最近做的是基于springboot微服务的电商项目，再之前一点做的是ssm的单体架构项目，负责了其中的几个模块，我先说一下我使用springboot+springcloud的感受吧。‘’/

**框架篇**

**首先讲下对springboot的理解，首先说一下我使用springboot的感受吧，搭建项目快，配置简单只需要一个properties文件就能完成配置，它跟我们之前使用xml的那种框架搭建方式会简化很多。另外它通过maven进行配置一些开箱即用的资源比如Mybatis,redis,mysql,web,start。同时它还内嵌了servlet,tomcat,通过一个mian方法和@SpringBootApplaction注解进行启动，部署的话只需要将项目打成jar,使用java -jar 包名就能启动了。**

开发springboot项目的时候，我发现springboot的核心就是启动类上的两行代码，启动springboot会首先加载启动类，启动类中可以看到两行代码,@SpringBootApplication和SpringApplication.run方法。

在 @SpringBootApplication 的内部包含了3个注解

@Configuration用于定义配置类，一般加在主类上

@EnableAutoConfiguration允许springboot自动配置注解加载到IoC容器中。

@ComponentScan它的作用就是让spring Boot扫描到Configuration类并把它加入到程序上下文。

简单点说，springboot就是一个微服务，主要做接口开发，

接口开发：同一个路径可以发送不同的请求方式

  知道前台传什么值，后台要返回什么样的值。

**Springboot的优点**

（1）所有 Spring 开发提供一个更快更广泛的入门体验。

①入门快

（2）零配置。没有多余的代码生成和XML 强制配置，遵循“约定大于配置” 。

　　　 ①“约定大于配置”（内部有默认的配置，不用特意去集成）

（3）集成了大量常用的第三方库的配置， Spring Boot 应用为这些第三方库提供了几乎可以零配置的开箱即用的能力。 ①开箱即用

（4）提供一系列大型项目常用的非功能性特征，如嵌入式服务器①内部嵌入tomcat、安全性、度量、运行状况检查、外部化配置等。**使用嵌入式的Servlet容器，应用最终可以打成Jar包的形式独立运行**

（5）Spring Boot 依赖于Spring，都是通过 IOC 机制来管理 Bean 的。

基于这些优点，我们拿它来做微服务框架的接口开发特别适合，首先将我们的项目按照功能模块的划分拆分成一个个的微服务，比如我在做电商项目时，将商品管理服务，订单服务，库存服务，用户服务，资源统一管理服务，购物车服务等块拆分成一个个的子模块，使用springboot框架快速搭建起来。那么就会出现一个问题，这么多的服务到底如何进行统一管理，相互之间如何调用的问题就凸显出来了？

其实springboot框架在设计之初就考虑到这些问题，那就是springcloud，它是基于springboot的微服务框架的实现，它提供了丰富的组件化开发。

**讲述对springcloud微服务的理解**

首先我们说一下**eureka注册中心**，主要是来注册保存我们的服务的地址列表，为消费者提供订阅服务的功能，它在搭建时需要引用eureka-server，然后在配置文件配上server端口号，然后配置register-with-eureka不注册自己，fetch-registry=false不去检索服务，然后在启动类上加上@EnableEurekaServer注解开启服务，这样注册中心就搭建完成了。注册中心主要功能是服务的注册与发现，生产者和消费者都需要将服务注册到注册中心上，注册中心会保存这些地址。并且服务会主动向注册中心发送续约请求，在一段时间不续约后就认为该服务已经挂掉了，就需要从注册中的地址列表中剔除。这个检查剔除服务的时间间隔可以在注册中心上配置。

如果我们要实现微服务之间调用我们就要用到**ribbon负载均衡**。它的负载均衡是软负载,也就是客户端服务器从Eureka Server拉取已注册的服务信息,然后根据负载均衡策略,直接发送请求,这整个过程都是在客户端完成的,并不需要注册中心的参与,SpringCloud在结合了Ribbon的负载均衡的实现中,封装增加了HTTP Client,是spring提供的对httpClient进行的封装，

ribbon的主要组件有ServerList,IRule,ServerListFilter等,他的整个流程是这样子的,首先,通过ServerList,获取所有的可用服务列表,然后通过ServerListFilter,过滤掉一部分地址,最后剩下的地址中,通过IRule选择一个实例,作为最终目标结果，**Ribbon就是将用户的请求平摊的分配到多个服务上，在微服务或分布式集群中经常使用**

在服务的调用时我们还会用到**feign**,使用流程是在pom.xml文件中引入openfeign。因为fegin调用是基于接口的服务调用，同时它集成了ribbon,还能完成负载均衡，在该接口类上加上@FeignClient，并配置服务名，同时在启动类上加上@EnableFeignClients让feign注解生效，在调用接口是需要注意的是get请求传参时必须要加上@RequestParam注解，如果post请求传递的参数是一个实体对象还要加上@RequestBody，使用feign我觉他的好处就是像调用方法一样调用接口用起来比较简单。其次我们会创建对应的服务端，服务端项目中要保证 请求的方式，请求的地址以及方法名，返回值，参数都要和feign中定义的保持一致。

hystrix熔断器，在整个微服务搭建的过程中有可能发生微服务雪崩的问题，说白了就是A->B->C,如果C这个微服务出现问题，就会影响到B,同样B出现问题，就会导致A也出问题，最后导致所有的微服务都不可用。咱们都知道Hystrix可以进行熔断，降级和资源隔离。熔断其实就可以理解成咱们家里面的保险丝。保险丝是为了保证在高负载情况下电路的安全，如果某个电路出现了负载过大情况，保险丝就会熔断，从而保证其他电路不受影响。Hystrix的熔断功能比较强大，它还可以进行自动检测并修复。（作为了解：具体的原理是这样的，默认情况下在10s内，每当20个请求中，有50%失败时，就会触发熔断，导致hystrix从闭合状态切换到打开状态，这样就不需要每次都去调用远程微服务而是直接走调用者本地的降级方法。5秒后hystrix会进入到半打开状态，这时如果有新的请求再过来hystrix就会尝试再次进行远程微服务调用，如果调用成功就会进入到闭合状态，如果不成功就会再次进入到打开状态，继续进行熔断。这个就是hystrix的自动检测并修复的能力。）

zuul网关 我们的微服务架构，服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统访问到。，不同的微服务一般会有不同的网络地址，那么这里就会遇到一个问题，APP/Browser怎么去访问这些后端的服务?客户端在访问这些微服务时必须记住几十甚至几百个地址，这对于客户端方来说太复杂也难以维护，因此，**我们需要一个微服务网关**，**介于客户端与服务器之间的中间层**，所有的外部请求都会先经过微服务网关。

（作为了解：继承于ZuulFilter的类，重写里面的filterType()方法filterOrder()方法，shouldFilter()方法，还有一个最核心的用来写具体业务逻辑的run()方法。其中filterType（）：的作用是用来返回一个字符串，指明该过滤器的类型，经常用到的有pre类型，说白了就是在请求被发送到微服务之前调用；还有post类型，说白了就是微服务执行完后再执行该过滤器

filterOrder()：返回一个int类型的值，用来指明该过滤器的执行顺序，数字越小表示优先级越高，就越先执行，shouldFilter()方法返回一个boolean值，用来指明该过滤器是否执行，true表示执行,false表示不执行。

run方法中就是之前说的，用来放具体的处理逻辑。）

业务篇

在这个项目里我负责了会员登录的模块，订单模块

因为做的是电商项目，所以我们对项目的多线程，高并发问题都很重视，针对这些也做了相应的处理。比如说在订单这个模块就会因为高并发出现一些问题；

## 订单模块

订单这个模块相对比较复杂，考虑的事情也比较多。像表设计、订单接口幂等性的问题，商品超卖的问题，项目中很多接口是用户登录后才能进行操作的，所以我们采用的是用JWT做的基于token的用户登录。

要做订单管理首先要解决接口幂等性的问题，接口幂等性就是 用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。

 我们解决接口幂等性的方式是我们在前端生成了唯一标识(就是随机数)，每个用户进入到支付成功页面生成这个唯一标识，用户点击支付订单把唯一标识传到我们的后端服务中，

 后端用redis的getAndset方法把我们的唯一标识当作key,随意设置一个value，判断我们当前的这个key是否为空的，如果不为空就返回提示用户正在生成订单，请稍等

 如果不为空的话就给当前的key设置几秒的失效时间，因为我们的程序走完需要时间，防止用户在这段时间内重复点击支付订单，生成重复订单。

首先说一下我们的表设计，一共设计了两张表，一张订单表，一张订单详情表。

订单详情表有，订单详情id,订单id,商品id,订单数量，订单价格，是否是sku属性等字段。

我当时采用 订单id和商品id 作为主外键关联，剩下的字段都比较好理解，就不多说了。

我们主要来说一下订单表。

订单表有订单id,收货地址id,支付方式id,支付状态，订单备注，订单数量，总金额，总件数，购买人，购买人手机号，邮编，购买时间等字段。

订单id作为主键自增，购买人就是当前登录的会员id，是为了指明这个订单是哪个会员的订单。

订单的支付方式，是考虑到我们的系统要支持好几种支付途径，比如1代表微信支付，2代表支付宝支付。

订单的状态我当时设置的有，未支付  已支付，取消支付。

总金额总件数没什么好说的，就是用户购买的总件数的价格。

购买数量这个就比较复杂了，我们要判断库存，顾客重复下订单等问题，在判断库存的时候因为我们当初设计表的原因，我们要分开判断，结合我们设计的sku表与非sku表来判断。通过前台传过来的skuId来判断是否是sku数据，如果是sku数据就根据id查询sku表，得到sku库存，如果购买数量大于库存则提示商品库存不足，如果数量小于库存，我们就要调用减sku数据库存的接口，去减库存。

去减库存还要注意我们的商品超卖的问题，也就是多个线程的订单请求同时进入到了减库存的操作，如果数据库库存有10个，a用户查询是10个，b用户查询也是10个，

  那么它们都通过了我们的判断库存的方法，

 a执行了一次减库存操作-10，b也进行了一次减库存的操作-10，这个时候数据库存就会变成负数-10，这样就造成了我们的商品超卖扽问题。

 我们通过在减库存的sql语句上添加一个条件 and stock-count>=0库存减去购买数量大于等于0，就解决了。

非sku数据也是一样处理，通过id去查询非sku商品数据，得到商品库存，后面判断都是一样的，最后在调用我们写好的减非sku数据的接口去减非sku数据的库存。

代码进行到这里其实我们就可以进行新增订单了，去获取订单id和订单信息的list集合，遍历list集合得到订单中要新增的数据，我们将新增的订单信息进行批量保存，新增成功后我们通过1.8的新特性去清空购物车中的订单。

**JWT的登录认证**

在我们做单体架构时，登录之后的用户信息是保存在服务端session中的，每次发送请求都会自带cookie值进行验证。随着我们现在用分布式架构，session和cookie已经不适用了。而我们又想保证用户信息的安全，所以使用到了JWT进行登录验证。JWT可以理解为加密工具，在我们的用户登录成功之后，把用户对象放进去，用JWT声明token令牌，把值传到前台，并存入sessionStorage，这样我们每次发送请求时都自带token值，防止越权访问。我们现在后台用的就是分布式架构，通过zuul网关去验证token，zuul网关就相当于拦截器+过滤器，我们创建一个类去继承zuulFilter过滤器，重写方法，在run方法中我们通过request.getParameter获取token值，通过JWT去解密验证token，如果token为空，则用户没有进行登录，如果token值解密错误，则用户非法访问或者token值被人为改过。当然token也有失效时间，默认是30分钟，这时解密的token值可能就为空了，但能确定不是非法访问。如果认为还是不安全，我们也可以在登录成功之后用base64再进行加密。

补充：①**sessionStorage与localStorage的区别？**

localStorage的生命周期是永久性的，即使关闭浏览器，也不会让数据消失，除非主动的去删除数据；

sessionStorage 的生命周期是在浏览器关闭前，在整个浏览器未关闭前，其数据一直都是存在的；

sessionStorage不在不同的浏览器窗口中共享；localstorage在所有同源窗口中都是共享的；

**Tomcat调优**

1. 启用线程池，默认的tomcat没有启用线程池，在tomcat中每一个用户请求都是一个线程，所以可以使用线程池提高性能。我们在项目中将maxThreads[最大线程数]设置为1000，将排队数acceptCount和maxThreads设置相等。当tomcat的线程数达到maxThreads后，新的请求就会排队等待，超过排队数的请求会被拒绝。
2. 使用64位的tomcat和jdk，禁用AJP协议。
3. 开启APR通讯模式，支持高并发。因为默认tomcat采用的是性能最低的BIO【阻塞IO】模式。
4. tomcat中设置JVM参数 通过-server开启server模式 通过-Xms4000m和-Xmx4000m设置初始堆大小和最大堆大小，通常将两个值设置为一样，避免堆空间不断增大和缩小所带来的性能损耗。

jvm的 内存结构：

方法区【持久代】

堆【java对象】

栈【基本数据类型，对象引用（指针/对象名）】

本地方法栈

计数器

nginx负载均衡（高并发处理方式）（选说，作为了解）

我们项目在最后上线部署的时候，采用的是nginx+3个tomcat进行负载均衡，在我们不进行负载均衡之前，所有的请求都由一台tomcat进行处理，这样会 使我们的tomcat所承受的压力增大，而我们进行负载均衡之后，同样数量的请求经过nginx将其分发到多台tomcat进行处理，从而降低每台tomcat所承受的压力,而且当其中一台机器宕机时，其他机器还可以继续提供服务，保证服务不间断。

    项目在部署完成后，遇到这么个问题，用户登录输入验证码的时候，明明验证码输入的正确，但总是提醒说验证码不正确从而不能正常登录，经过分析后发现有可能第一次请求被发送到t1上，那么放在session中的验证码就被放到了t1上，当用户输入验证码点击登录时，新发送的请求有可能被发送到t2上面，而t2的session中不存在该验证码，这样在进行对比时就肯定会不一致从而提示验证码输入错误。

咱们都知道nginx有不同的负载均衡策略，默认是轮询策略，说白了就是将请求均匀的分发到不同的tomcat上，比如我们有两台tomcat，分别是t1和t2,当4个请求到来的时候，第一个请求被nginx发送到t1上，第二个请求被nginx发送到t2上，第三个请求又被发送到t1上，第四个请求又被发送到t2上，这就是nginx默认的轮询策略，当其中一台tomcat宕机时，会被自动剔除掉。

再者在企业中不同tomcat所在机器的配置和性能都不一样，这个时候如果用默认的轮询策略就显的比较浪费资源，我们可以通过weight这个关键字配置加权轮询，（权重）除此之外nginx还有一个IP\_Hash的负载均衡策略，这个策略说白了就是根据访问的客户端的ip地址，对其进行hash化，然后如果该客户端第一次请求被发送到t1上了，那么它就会和t1进行绑定，以后的每次请求也会被发送到t1上，只有t1宕机了，请求才会被发送到其他tomcat上。

所以我们就用ip\_hash这种负载均衡策略来代替默认的轮询策略，这样就解决了验证码错误问题。但这里面还有一个问题，如果被客户端绑定的这台tomcat宕机了，那么请求就会被发送到其他tomcat上，这个时候在用户进行操作时，请求会被拦截器拦截，拦截器中的代码会验证session中是否有当前登录的用户信息，如果没有则跳转到登录页面，因为这台tomcat上的session中不存在用户登录的信息所以就会跳转到登录页面，这就会导致用户的体验度非常不好，感觉也会很怪异。

为了最终解决这个问题，我通过上网查资料，就决定通过采用基于redis的分布式session来解决session共享的问题。

=============（补充点/被动说）=============日志管理模块

在项目中我做过后台管理系统的日志管理模块，日志管理模块的作用就是记录用户的操作，这样就知道谁在什么时候干了什么事情。

**spring框架是我们java开发中最强大的框架，也是最重要的框架。**

**一、spring相当于一个粘合剂，主要有两个核心**

总结：

① ioc的工厂模式管理bean与ioc的控制反转 / di的各层之间的依赖注入关系

②aop的面向切面编程与动态代理

概念：**1、IOC的控制反转**：ioc也称控制反转，就是把原先控制代码对象的生产由代码转换到IOC容器中去实现，作用是为了解耦，降低类之间的耦合度，设计思想是设计模式的工厂模式，所谓的工厂模式就是我们不需要知道其生产的过程，只需要其产出的对象即可。

自我理解： 我理解的ioc就是@Controller @Service @Component都是把类交给ioc进行管理，ioc就是一个容器，可以自动创建对象（new对象），把两个类都交给ioc进行管理，这两个类就可以相互进行依赖注入，把A类注入B类，B类就可以调用A类里面的所有属性和方法。

@Controller ：用于对Controller实现类注解

@Service ：用于对Service实现类注解

@Repository ：用于对Dao实现类注解

@Component主要用于将一个Java类注入到Spring框架中，它相当于XML配置文件中的<bean id=”xxx” class=”xxx”/>

spring注入的两种方式：①xml文件的方式使用<bean>标签进行注入

②使用注解，把类交给ioc进行管理

**2、Di依赖注入方式**： @resource,@autowired使用注解注入

@Autowired byType根据类型注入 是spring提供的

@Resource byname根据名称注入 是jdk提供的（推荐使用）

**3、Aop的面向切面编程**：AOP简单地说就是在不改变源程序的基础上为代码段增加新的功能，对代码段进行增强处理。

aop的代理可以通过jdk动态代理，也可以通过cglib实现，默认通过jdk动态代理实现。

JDK动态代理需要接口的支持,如果没有接口只有类，则使用cglib实现。

业务：

比如我们在项目中用到的aop+注解 去实现日志记录

　　1、创建自定义注解

//自定义注解 如果有注解 就需要日志记录

创建一个类@interface

@Target(ElementType.METHOD) 声明注解使用的地方

( type 代表是类上

field 只能应用在 类的属性上

method 自定义注解只能用在方法上

parameter 只能应用在参数上）

@Retention(RetentionPolicy.RUNTIME)声明注解的声明周期

public @interface log\_znn {

String value() default ""; 定义一个空的字符串，标明此注解用来干什么

}

　我们的日志记录，首先要有一个切面类，aop的切面类与拦截器相类似，aop过滤请求，过滤后继续往下走，一般和注解连用，上面我已经介绍过了注解的使用，下面我们说一下切面类

2、在日志记录的切面类上加注解 @Component把类交给ioc管理 @Aspect 声明是切面类

3、配置前置通知 配置切点 切面 @Before切点表达式(“execution(\* 类路径.\*.\*(..))”)

4、创建方法 方法中加参数（JoinPoint joinPoint)

5、构建日志对象，通过joinPoint参数获取方法签名，根据签名获取方法对象，判断此方法是否有日志注解，有就需要做日志记录。

6、最后在业务处理层service设置日志记录的参数，内容，操作人，ip等等保存到数据库。

注意：短时之内，日志表数据量变大通过建索引的方法，但当数据量达到一定程度上，应当分库分表，按月份分表或者按天分表，(这时候不能用#{}，要用${})

**（sql优化方面）**

1、对于大数据量的分页不要用常规的 limit 开始位置，每页条数；这样越靠后查询时间越长，效率越低，可以使用id>=的方法来替代，比如select 字段名 From 表名 where id >= (select id from 表名 limit 开始位置，1) Limit 每页条数

2、SELECT语句中避免使用'\*’，只查询需要返回的字段，这样可以减少解析sql语句的时间，以及减少 带宽，cpu,内存，io等 各方面的消耗。

3、 外键必须加索引，这样可以提升多表联查的性能

1. 通过冗余字段避免多表连查，从而提高性能。比如在商品表中可以不仅有分类的id，还可以加上分类的名字，这样就不用为了显示分类名字而去进行多表联查了。又比如在会员表中不仅有地区的id，还可以有地区的名字这样也可以避免为了显示地区名而进行多表联查。

5、在 where，order by 以及GROPU BY 涉及的列上建立索引，为了避免全表扫描。

6、In和exists的区别：主要是就是驱动顺序的改变，这是性能变化的关键，如果是exists，那么以外层表为驱动表，先被访问，如果是IN，那么先执行子查询，以内层表为驱动表。所以IN适合于外表大而内表小的情况；EXISTS适合于外表小而内表大的情况。

7、通常要用小表去驱动大表，这样可以提高性能，在left join 中 左边的表为驱动表，所以应该让左边的表尽可能的为小表，在right join 中 右边的表为驱动表，所以应该让右边的表尽可能的小，在 inner join 中 mysql会自动选择较小的表为驱动表。

8、避免在 where 子句中对有索引的字段进行运算,这会导致索引失效，从而进行全表扫描。比如select 字段名 from 表名 where 库存量+1>=10,就需要改为 select 字段名 from 表名 where 库存量>=9

9、避免在where字句中对字段进行null值判断，这样会导致即便字段上有索引也会进行全表扫描，好的处理方式就是给字段设置默认值。

什么是索引

通过索引可以提高查询的速度，它就相当于书的目录；索引不是越多越好，因为索引会占用额外的空间，并且在进行增删改的时候还需要额外维护索引。创建索引的原则：给频繁查询的字段上创建索引根据当前字段的业务含义，来区分是创建唯一索引还是创建普通索引考虑到索引不是越多越好，也可以根据业务情况创建复合索引，

想要使其复合索引起作用，在查询的时候需要按照当时创建

复合索引的字段的顺序来,也就是最左匹配原则。

执行计划:通过explain查看sql语句的执行计划key:代表mysql实际使用的索引。type:这个字段比较重要，它表示MySQL在表中找到所需行的方式：

Const：主键在where条件中

Range：扫描部分索引，常见于between、<、>等的查询

Index：代表扫描全部 索引树

ALL：代表扫描全 表/全表扫描/不走索引。

创建索引-- 普通索引create index index\_status on t\_log(status);

-- 创建唯一索引create unique index index\_username on t\_user(userName);

https://www.cnblogs.com/summer0space/p/7247778.html-- 创建复合索引create index index\_status\_logincount on t\_user(status, loginCount);

索引的类型

【索引特性上来说】

唯一索引 普通索引

【字段的个数来说】

单个索引/单一索引 复合索引/组合索引

【字段的个数，如果对多个字段同时加索引则称之为复合索引】

聚集索引和非聚集索引：

聚集索引：主键就属于聚集索引。在一张表上聚集索引只能有一个。

非聚集索引: 除了主键以外的唯一索引，普通索引。在一张表上非聚集索引可以有多个。

唯一索引和普通索引的区别：

唯一索引所在字段中的值必须是唯一的。主键字段会自动创建唯一索引。

创建索引，不一定就能使用上索引；想要索引生效必须满足两个条件，第一，该字段上有索引；第二，所写的sql语句必须支持索引。

什么时候索引不起作用/什么时候索引会失效?

1.如果在有索引的字段上进行运算，那么索引失效，会进行全表扫描。

2.在使用like的时候如果%在前面则索引失效

全文检索技术

java:[开源的搜索引擎技术]

Lucene：

Solr【是对lucene的封装】：企业级的搜索引擎技术

ES【是对lucene的封装】：互联网级的搜索引擎技术

3.避免在查询条件中使用is null,可以采用默认值的方法来解决

大批量数据的插入/更新?

1.一条条插入

2.分段 批量 提交【推荐】【减少和数据库的交互次数】

3.批量插入和数据库交互一次

删除?delete ... in

**（restful接口）**

之前工作的过程中，我们的经理给我说，我们项目需要给前台提供restful接口,当时对这个restful只是了解并没有真正的在项目中使用过,所以下去我也查阅了许多关于restful接口开发的文档,通过文档得知，restful不是一项新技术,就是新瓶装旧酒,其实就是符合restful风格的url,都被称之为restful类型的接口,因为在互联网中可以通过url去访问任意的资源，restful是基于http协议现有的Get动作进行查询，Post动作进行增加，Put动作进行修改，Delete动作进行删除。

**JVM虚拟机调优**

我们的linux服务器,安装centos7.0, 64位的操作系统和64位的软件，配置了8个cpu,128G内存。

我们在jvm优化的时候是这样做的首先JVM将内存划分为：年轻代年老代永久代（方法区）其中年轻代和年老代属于堆内存，永久代不属于堆内存，有虚拟机直接分配。年轻代：年轻代用来存放JVM刚分配的Java对象年老代：年轻代中经过垃圾回收没有回收掉的对象将被放到年老代永久代：永久代存放Class类、Method方法元信息，其大小跟项目的规模、类、方法的量有关，一般设置为256M就足够，预留30%的空间。通过-XX:PermSize=256M -XX:MaxPermSize=256M 来设置永久代（方法区）的大小。

jvm的垃圾回收算法有[GC]

串行算法（单线程）

并行算法

并发算法

吞吐量优先的并行收集器

响应时间优先的并发收集器

我们的项目设置的是响应时间优先的并发收集器，

将堆大小通过 -Xms -Xmx设置为4G,将年轻代通过 -Xmn 设置为2g,

设置年老代为并发收集，当时设置的是运行6次GC以后对 内存空间进行压缩、整理。打开对年老代的压缩，可以消除碎片。

=========（基础概念）==============

**SpringMvc的底层原理**  
整个处理过程从一个HTTP请求开始：  
springMVC是我最常用的web控制层框架，这类框架都有共同的功能，就是通过过滤器或者拦截器来处理请求，我们的用户访问资源的路径包括协议，主机IP和端口号，资源名，我们控制层框处理的就是这个资源名和资源后边跟着的参数。具体流程是：

运行原理：

1.用户发送请求到核心控制器DispatcherServlet

2.核心控制器查询一个或多个handlerMapping,找到处理请求的handler

3.核心控制器将请求转发给handler

4.核心控制器查询HandlerAdapter适配器。

5.由适配器执行handler调用业务逻辑处理后，返回ModelAndView对象给控制器。

6.核心控制器查询一个或多个视图解析器，将model渲染到views

7.将views结果显示到客户端。

**我们的springMVC用的都是注解模式**

**其中比较常用的注解有**

（1）. @RequestMapping("/query") 配置就是请求路径。注意：他可以加到方法上，也可以加到类上面（类似于namespace）

（2）. @PathVariable 加在参数上，@RequestMapping(value = "/params4/{username}/{password}", method = RequestMethod.GET) 注入访问路径中参数

（3）. @RequestParam 加载参数上使用时，可以控制该参数是否必须传递。

（4）. @ResponseBody 加载方法上，返回的是纯数据。（注意他不要和ModelAndView连用）同时它依赖jackson的jar包对返回值进行json化的处理

（5）.@DateTimeFormat（partten="yyyy-MM-dd HH:mm:ss"） 用来转换时间

SpringMvc的自我理解：

SpringMvc是一个控制层框架，就是用来接参传参页面跳转；

接参可以用属性、对象、Request接收

传参可以用model 、json

页面跳转是转发和重定向。

转发：地址栏不发生改变，数据共享，一次请求，执行效率高

重定向：地址栏发生改变，数据不共享，多次请求，执行效率低。

**HashMap的底层原理**

**1、HashMap红黑树原理分析**

相比 jdk1.7 的 HashMap 而言，JDK8之后，如果哈希表单向链表中元素超过8个，那么单向链表这种数据结构会变成红黑树数据结构。当红黑树上的节点数量小于6个，会重新把红黑树变成单向链表数据结构。

为什么要这样设计呢？好处就是避免在最极端的情况下链表变得很长很长，在查询的时候，效率会非常慢。

红黑树查询：其访问性能近似于折半查找，性能较高

链表查询：这种情况下，需要遍历全部元素才行，性能较低。

简单的说，红黑树是一种近似平衡的二叉查找树，其主要的优点就是“平衡“，即左右子树高度几乎一致，以此来防止树退化为链表，通过这种方式来查找省时省力。

**2、hashMap运行原理**

\*HashMap的底层实现是一个哈希表即数组+链表；

我们在将数据存入哈希表的时候，要得到hashcode的值，用值去模数组的长度，

得到的余数放入数组段中，此时数组段的位置上已经有值了，那么将通过equals方法比较两个hash值是否相同，如果两个hash值相等，再比较两个键值对key是否相等，如果两个hash值不等，则在此位置，以链表的方式，将新存储的数据放到表头；如果hash值和key都相等，将覆盖原先的value。

这样hashmap不仅具有了数据查询快速的特性，同时有了链表方便插入、删除的特性；因此hashMap对于数据的存储查询具有非常好的特性；

**3、HashMap中的“死锁”的原因？1.7时 头插法**

HashMap是非线程安全，死锁一般都是产生于并发情况下。我们假设有二个进程T1、T2，HashMap容量为2,T1线程放入key A、B、C、D、E。在T1线程中A、B、C Hash值相同，于是形成一个链接，假设为A->C->B，而D、E Hash值不同，于是容量不足，需要新建一个更大尺寸的hash表，然后把数据从老的Hash表中 迁移到新的Hash表中(refresh)。这时T2进程闯进来了，T1暂时挂起，T2进程也准备放入新的key，这时也 发现容量不足，也refresh一把。refresh之后原来的链表结构假设为C->A，之后T1进程继续执行，链接结构 为A->C,这时就形成A.next=B,B.next=A的环形链表。一旦取值进入这个环形链表就会陷入死循环。

**4、HashMap中能put两个相同key吗？为什么？**

能/不能

能，如果两个key相同，但hash值不等，则在此位置，以链表的方式，将新存储的数据放到表头

不能，如果两个key的hash值和key都相同，就会把相同的value值覆盖。

**5、HashMap中的键值可以为null吗？原理？**

HashMap 支持key=null 但是 Hashtable 不支持 key =null/TreeMap也不支持key=null

HashMap在调用put方法时，其实是调用了putVal方法，而该方法的第一个参数hash值的获取是通过HashMap中自定义的hash方法得到，hash方法中明确指出当key值为null时返回hash值为0，并不会报错，且putVal方法中也没有对value值进行任何约束，故HashMap中的键值都能存放null值。

HashTable在调用put方法时，首先会对put方法中的值value进行null值判断，为null抛出空指针异常；然后会对key值进行Object类的hashCode方法计算hash值，为null值也会抛出空指针异常，故不能在HashTable键、值中放入null值。

**6、HashMap扩容机制？**

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容。

当HashMap中的元素个数超过数组大小(数组总大小length,不是数组中个数size)\*loadFactor时，就会进行数组扩容，负载因子的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12这个临界值的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置。

0.75这个值成为负载因子，那么为什么负载因子为0.75呢？这是通过大量实验统计得出来的，如果过小，比如0.5，那么当存放的元素超过一半时就进行扩容，会造成资源的浪费；如果过大，比如1，那么当元素满的时候才进行扩容，会使get,put操作的碰撞几率增加。

**String,StringBuffer,StringBuilder的区别**String是一个常量，是不可变的，所以对于每一次+=赋值都会创建一个新的对象，StringBuffer和StringBuilder都是可变的，当进行字符串拼接时采用append方法，在原来的基础上进行追加，所以性能比String要高，又因为StringBuffer是线程安全的而StringBuilder是线程非安全的，所以StringBuilder的效率高于StringBuffer.对于大数据量的字符串的拼接，采用StringBuffer,StringBuilder.

**ConcurrentHashMap原理**

在java并发包里面，ConcurrentHashMap是线程安全的hashMap，通过引入分段锁的概念将一个大的Map拆分成多个小的HashTable.在存值和取值得时候通过key.hashCode()来计算key及其对应的值应该放到哪个HashTable中。默认情况下ConcurrentHashMap会创建16个分段数组集合。这样在进行操作的时候如果有3个线程A B C，这时候A B两个线程根据key.hashCode()可能被分配到同一个hashtable中，这样A在进行操作的时候B就会阻塞。但C这个线程可能被分配到了另外一个hashtable中，这样C就可以直接执行而不会阻塞。所以效率就得到了极大的提升。默认情况下可以提高16倍。

数据库相关

视图：

视图里面并没有存放真正的数据，存放的是sql语句，真正的数据还是存放在表里。可以通过 create view 视图名 as select语句 的方式来创建视图，这样可以将比较复杂的查询语句作为一个视图，后续再需要执行这个复杂的sql，就不用重新再写一遍了，而是直接 select \* from 视图名 就行了。

存储过程：

这个东西我们项目中基本不用，之前有了解过，说白了，存储过程里封装了sql语句，也支持传参和返回值，有点类似于java中的方法，这样你就可以直接在java端调用存储过程执行复杂的逻辑，而不需要发送多条 sql语句，一定程度上降低了和数据库的交互次数，提高了性能，但存储过程的可移植性比较差， mysql和oracle支持的语法结构和关键字也都不一样。

mysql的存储引擎：

有哪些，它们的区别是什么：

有MyISAM和InnoDB两种存储引擎。

它们的区别是：

MyISAM用的是表锁，InnoDB用的是行锁。

MyISAM不支持事务处理,不支持外键而InnoDB支持。

### mybatis中的#和$的区别

1. #将传入的数据根据类型进行相应的转换，如果类型不匹配则报错。如果传入的是字符串类型则会自动加上双引号。

2. $将传入的数据直接显示在sql中。

3. #方式能够很大程度上防止sql注入，$方式无法防止Sql注入，所以一般能用#的就别用$。

**mongodb**

1  mongodb简介

    mongodb是一个 非关系型数据库  （ redis  memcahe） （缓存   快 ）专业的说法

MongoDB 是一个 非关系型数据库 是一个面向文档存储的数据库，操作起来比较简单和容易。

MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写，是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。

你可以在MongoDB记录中设置任何属性的索引 (如：FirstName="Sameer",Address="8 Gandhi Road")来实现更快的排序。

2 环境搭建

  以window平台为准

  启动服务  mongod.exe

  连接    mongo.exe

3 语法

mongodb可以说分为三个方面 ①database （数据库） ②collection （集合 或者表）

③document（文档 json）

4、数据库相关操作

show dbs（查看所有数据库（有数据的数据库））

use（创建或者切换数据库）

db（查看当前数据库）

db.dropDatabase() 删除数据库

5、集合的相关操作

db.createCollection()  创建集合

db.集合名称.drop() 删除集合

show collections()/tables 查看所有集合

6、文档相关操作

db.集合名.remove 移除集合中的数据

db.集合名.update 更新数据

db.集合名.find() 查找

db.集合名.insert(文档)  集合添加数据    save   insertone   insertmany文档的操作

Redis缓存

**Redis（了解）**

**关系型数据库 mysql  orale  db2  sqlserver**

**非关系型数据库（缓存）  redis   mogodb**

**Redis是一个开源的，高性能的 key-value 数据库。作用：减轻数据库压力，提高性能。**

**Redis优势**

**高性能 ： Redis读和写的速度特别快，因为它是基于内存的，可以将内存中的数据保存在磁**

**盘中，重启的时候可以再次加载进行使用。**

**丰富的数据类型 ： Redis支5种数据类型 String、hash，list、set、zset。**

**redis 五大数据类型**

**1.String 字符串类型**

**是redis中最基本的数据类型，一个key对应一个value。**

**2.Hash （哈希）**

**是一个Mapmap，指值本身又是一种键值对结构，如 value={{field1,value1},......fieldN,valueN}}**

**3.list**

**就像链表，可以双向存储数据，list集合是有序的。**

**List 说白了就是链表（redis 使用双端链表实现的 List），是有序的，value可以重复，可以通过下标取出对应的value值，左右两边都能进行插入和删除数据。**

**4.Set 集合**

**集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中 1. 不允许有重复的元素，2.集合中的元素是无序的，不能通过索引下标获取元素，3.支持集合间的操作，可以取多个集合取交集、并集、差集**

**5.zset 有序集合**

**有序集合和集合有着必然的联系，保留了集合不能有重复成员的特性，区别是，有序集合中的元素是可以排序的，它给每个元素设置一个分数，作为排序的依据。**

**原子性：Redis的所有操作都是原子性的，要么成功执行要么失败完全不执行。单个操作是原子性的、多个操作也支持事务，也就是单线程。**

**持久化：Redis 的持久化方案  （rdb （快照）以快照的方式保存  容易丢失数据）  aof（基于日志）以记录日志的方式保存的  不容易丢数据）**

**Redis的持久化策略**

有AOF和RDB两种方式,我们一般使用默认的RDB方式。因为它的效率会更高。

RDB持久化是在指定的时间间隔内将内存中的 数据 写入到硬盘的rdb文件中，指定每隔多长时间有多少个key发生变化，就将其持久化。

AOF：是将每个 写操作对应的指令 保存到后缀名为aof的 文本 文件中。

rdb的性能要高于aof：

因为rdb中直接存储的是数据，所以在redis服务器启动的时候，直接加载rdb文件中的数据到内存中就行了。

而aof中存储的是写指令，所以在redis服务器启动的时候，需要先加载aof

文件中的指令，之后还需要再次执行这些指令。

因为每个写操作指令都会及时写入aof文件中，所以aof这种持久化策略不容易丢失数据，

而rdb是在指定的时间间隔后才会写入数据到文件中，所以有可能会丢失一部分数据。

**Redis缺点**

由于是内存数据库，所以，单台机器，存储的数据量，跟机器本身的内存大小相同。虽然redis本身有key过

期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。

**redis有两种过期策略，定期删除和惰性删除**

**定期删除：redis每个100ms随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。**

**惰性删除：在获取某个key的时候，redis检查一下，如果该key设置了过期时间则判断该过期时间是否已经过期，如果过期了就直接删掉并不返回任何东西。**

**redis的内存淘汰机制**

　　当redis内存快耗尽时，redis会启动内存淘汰机制，将部分key清掉以腾出内存。

　　redis提供6中数据淘汰策略

**redis的主从复制**

**主从复制是指将一台Redis服务器的数据，复制到其它的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。**

**默认情况下，每台Redis服务器都是主节点，且一个主节点可以有多个从节点（或没有从节点），但一个从节点只能有一个主节点。所以说主从复制是Redis高可用的基础**。

**redis的哨兵模式**

配置：vi sentinel.conf 把daemonize yes , 把sentinel monitor ....1234 几台哨兵就写几

**l 监控**（Monitoring）：哨兵会不断地检查主节点和从节点是否运作正常。

**自动故障转移**（ Automatic failover）：**当主节点不能正常工作时，哨兵会开始自动故障转移操作，它会将失效主节点的其中一个从节点升级为新的主节点，并让其他从节点改为复制新的主节点。（会自动更改配置文件）**

**redis集群模式**

**组成**：Redis集群采用**无中心结构**，集群由**多个节点(Node)组成**，Redis的数据分布在这些节点中

**数据分区**：数据分区(或称数据分片)是Redis集群最核心的功能。

Redis集群将数据分散到多个节点，一方面突破了Redis单机内存大小的限制，存储容量大大增加；另一方面每个主节点都可以对外提供读服务和写服务，大大提高了集群的响应能力。

**高可用**：集群支持主从复制和主节点的自动故障转移（与哨兵类似）；当任一节点发生故障时，集群仍然可以对外提供服务。

Redis集群需要至少要三个主节点，三个从节点，总共6个Redis节点，这里用一台机器（可以多台机器部署，修改一下IP地址就可以了）部署6个Redis实例。为什么要三主三从，并且最好是奇数，如果我们只有两台集群，当我们有一台挂掉，redis就挂掉了，集群中超过半数的节点失效时我们的redis就失效了，而我们设置三台，当有一台挂掉了，我们剩下的两台仍然可以工作，只不过会丢掉一部分槽里的数据。

**例子：（**集群模式与hashMap底层类似**）**

一个 Redis 集群包含 16384 个哈希槽（hash slot）， 数据库中的每个键都属于这 16384 个哈希槽的其中一个，集群使用公式CRC16(key) % 16384 来计算键key属于哪个槽，其中 CRC16(key)语句用于计算键key的CRC16校验和 。集群中的每个节点负责处理一部分哈希槽。

举个例子，一个集群可以有三个节点，其中：

节点 A 负责处理 0 号至 5500 号哈希槽。

节点 B 负责处理 5501 号至 11000 号哈希槽。

节点 C 负责处理 11001 号至 16383 号哈希槽。

这种将哈希槽分布到不同节点的做法使得用户可以很容易地向集群中添加或者删除节点。 比如说：

如果用户将新节点D添加到集群中，那么集群只需要将节点A、B、C 中的某些槽移动到节点D就可以了。

如果用户要从集群中移除节点A， 那么集群只需要将节点A中的所有哈希槽移动到节点B 和节点C，然后再移除空白（不包含任何哈希槽）的节点A就可以了。

**redis的缓存**

**缓存穿透**

产生这个问题的原因可能是外部的恶意攻击，例如，对用户信息进行了缓存，但恶意攻击者使用不存在的用户id频繁请求接口，导致查询缓存不命中，然后穿透 DB 查询依然不命中。这时会有大量请求穿透缓存访问到 DB。

解决的办法如下。

缓存穿透 1、我会在接口层增加校验，比如参数做校验，不合法的参数直接代码 Return，

2、id 做基础校验，id <=0的直接拦截，或者id大于一定数量拦截等等。

**缓存击穿**

而缓存击穿不同的是缓存击穿是指一个Key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个Key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个完好无损的桶上凿开了一个洞。

简单地说：缓存击穿，就是某个热点数据失效时，大量针对这个数据的请求会击穿到数据库。

解决这个问题有如下办法。

1、可以使用互斥锁更新，保证同一个进程中针对同一个数据不会并发请求到 DB，减小 DB 压力。

2、针对多个热点 key 同时失效的问题，可以在缓存时使用固定时间加上一个小的随机数，避免大量热点 key 同一时刻失效。

**缓存雪崩**

**例子**：如果所有首页的Key失效时间都是12小时，中午12点刷新的，我零点有个秒杀活动大量用户涌入，假设当时每秒 6000 个请求，本来缓存在可以扛住每秒 5000 个请求，但是缓存当时所有的Key都失效了。此时 1 秒 6000 个请求全部落数据库，数据库必然扛不住，它会报一下警，真实情况可能DB都没反应过来就直接挂了。此时，如果没用什么特别的方案来处理这个故障，重启数据库，但是数据库立马又被新的流量给打死了。这就是我理解的缓存雪崩。

缓存雪崩，产生的原因是缓存挂掉，这时所有的请求都会穿透到 DB。

解决方法：

1、处理缓存雪崩简单，在批量往Redis存数据的时候，把每个Key的失效时间都加个随机值就好了，这样可以保证数据不会在同一时间大面积失效，我相信，Redis这点流量还是顶得住的。

2、使用快速失败的熔断策略，减少 DB 瞬间压力；

3、使用主从模式和集群模式来尽量保证缓存服务的高可用。

**事务+分布式事务**

**什么是事务？**

事务就是一组有逻辑操作单元的sql集合，要么都成功 要么都不成功

**1、事务的四大特性**

1. 原子性：事务包含的所有操作要么都发生，要么都不发生

2. 持久性：事务一旦提交成功，对数据库的改变应该是永久性的，即便是遇到了故障也不会丢失丢失提交事务的操作

3. 隔离性：事务的操作不应该被并发的其他操作所影响，多个并发事务之间应该是相互隔离的

4. 一致性：事务的所有操作必须是数据库从一个一致性状态到另一个一致性状态

**事务的几种传播特性**

**传播特性**

**就是事务的 传播范围 A事务调用A事务 然后在调用B事务 这就用到了传播特性 是公用一个事务 还是在创建一个新的事务**

1. PROPAGATION\_REQUIRED (瑞块儿): 如果存在一个事务，则支持当前事务。如果没有事务则开启

 2. PROPAGATION\_SUPPORTS: 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行

 3. PROPAGATION\_MANDATORY: 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

 4. PROPAGATION\_REQUIRES\_NEW: 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

 5. PROPAGATION\_NOT\_SUPPORTED: 总是非事务地执行，并挂起任何存在的事务。

 6. PROPAGATION\_NEVER: 总是非事务地执行，如果存在一个活动事务，则抛出异常

 7. PROPAGATION\_NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务,则按TransactionDefinition.PROPAGATION\_REQUIRED 属性执行

**事务的隔离级别**

**隔离级别**

**A事务对B事务是否有影响 A事务提交数据还没成功 B事务来访问 让不让他访问 就是根据事务来确定的 这就是隔离级别**

 1. ISOLATION\_DEFAULT： 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.另外四个与JDBC的隔离级别相对应

 2. ISOLATION\_READ\_UNCOMMITTED： 这是事务最低的隔离级别，它充许令外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。

 3. ISOLATION\_READ\_COMMITTED： 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据

 4. ISOLATION\_REPEATABLE\_READ： 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。

5. ISOLATION\_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读。

**什么是分布式事务？**

前提：1支持事务 2 可以直接操控数据库

就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

**CAP 理论在分布式系统中**

一致性：分布式环境下多个节点的数据是否强一致

可用性：分布式服务能一直保证可用状态。当用户发出一个请求后，服务能在有限时间内返回结果

分区容忍性：特指对网络分区的容忍性

其中，分区容忍性又是不可或缺的。

**使用场景：**

最经典的场景就是支付了，一笔支付，是对买家账户进行扣款，同时对卖家账户进行加钱，这些操作必须在一个事务里执行，要么全部成功，要么全部失败。而对于买家账户属于买家中心，对应的是买家数据库，而卖家账户属于卖家中心，对应的是卖家数据库，对不同数据库的操作必然需要引入分布式事务

**为什么要用分布式事务？**

一个微服务对应多个数据库，可用沙丁jdbc解决，而多个服务对应多个（不同的）数据库就要用分布式事务了，而对不同的数据库进行操作又必然要引入分布式事务seata.

而seata是怎么去控制事务的呢？

一张图说明一切。

**Docker**

Docker（解决项目部署的）

　　　没用docker之前，想在一台机器上跑很多服务，每个服务都要隔离开，分别在liunx上进行部署，每一台机器上都要安装很多服务。。。

1、什么是docker

Docker是一款可以解决运行环境和配置问题的软件容器引擎。

2、docker的组成

　　　Docker由镜像，容器，仓库组成

3、什么是镜像

　　　一个镜像可以用来创建多个docker，透过镜像，使得应用程序所需要的环境系统从上而下的进行打包。

　　　　docker images 列出本机的所有镜像 镜像名 版本号 id 创建时间 大小

　　　 docker search 镜像名 搜索镜像 收藏数

　　　　docker pull centos 下载镜像不指定版本 （下载最新的）拉取镜像仓库

　　　 docker rmi 镜像名（id） 注意正在被容器使用的镜像是无法删除

4、什么是容器

　　　Docker 利用容器（Container）独立运行的一个或一组应用

　　　docker run -it --name 容器名称（唯一的） 镜像名（创建容器 并进入终端 一旦退出 容器结束） 创建或运行

　　　docker run -itd --name 容器名称（唯一的） 镜像（创建容器 进入后台运行不会进行终端 （exec 进入终端） 终端退出 容器继续运行）

　　　docker exec -it 容器名称（容器id） /bin/bash 进入后台运行容器的终端

　　　docker ps 查看运行中的容器 、

　　　docker ps -a 查看所有的容器

　　　docker rm 容器id（名称）（运行中的容器不能删 如果想把运行中的容器也删掉 加参数 -f） 删除容器

　　　docker stop 容器id（名称） 停止容器

　　　docker start 容器id（名称）启动容器

5、什么是仓库

　　　仓库（Repository）是集中存放镜像文件的场所。

6、怎么制作镜像

　　　制作镜像 两种方式：

　　　 commit （）

　　　 build（Dockerfile）

　　　 编写Dockerfile文件

　　　 语法 ：

　　　 FROM 指定基础镜像

　　　 MAINTAINER 指定镜像维护者信息

　　　 RUN 用于执行指定脚本命令

　　　 CMD 指定启动容器时执行的命令

　　　 EXPOSE 指定容器暴露的端口

　　　 ENV 指定环境变量

　　　 ADD 将文件从宿主机复制到容器指定位置，同时对压缩文件有自动解压功能

　　　 COPY 将文件从宿主机复制到容器指定位置

　　　 ENTRPOINT 设置容器启动时需要运行的命令

　　　 WORKDIR 为后续的如RUN、CMD、ENTRYPOINT、COPY、ADD指定工作目录

　　　 加深对语法的理解和制作镜像 （制作tomcat的镜像 依赖那些环境 jdk） 参考地址 https://blog.csdn.net/qq\_36538012/article/details/80839361

　　　 制作镜像命令 进入Dockfile目录

　　　 docker build -f Dockerfile的路径 -t 镜像名称:版本号 .

2、环境准备

Docker 运行在 CentOS 7 上， 要求系统为64位、系统内核版本为 3.10 以上。

3、 确认环境

怎么查看centos版本 cat /etc/redhat-release

怎么查看内核 uname -r

**反射 reflect**

什么是反射： 对类来说我能知道里面所有的属性和方法，

对对象来说，我能知道里面所有的属性值和调用他的方法。

动态获取信息 以及 动态调用对象的方法 就是反射机制！！！

获取类对象的三种方式

①类名点 . class 接收返回值 获取到类对象

Class areaclass= Area.class;

②填写类的全路径

Class areaclass2=Class.forName(“com.jgxi.entity.po.Area”);

③New对象  通过对象名 get获取class对象

Area a = new Area;

Class areaClass3=a.getClass();

反射的属性对象：class类对象，field属性，method方法，annotation注解，Constructor构造器

里面的方法：

类对象名.getDeclaredFields() 得到所有属性

类对象名.getFields() 得到所有公共的属性

getName() 获取属性(方法)名

getType() 获取属性(方法)类型

**写过接口文档吗？接口文档是你自己定的还是他们给你的模板？主要有什么内容？这个接口文档你会发给谁？**

写过，给的模板

简要描述、请求URL、请求方式、前后端负责人、请求参数、返回参数、返回示例、返回参数说明、接口完成状态、备注。

主要发给前端。

**微服务,多服务之间频繁调用导致系统接口超时,怎么调优**

**如果是恶意频繁请求：接口限流**

**1、消息队列：多服务之间A调用B，B调用C，如果其中B服务响应过慢不能及时返回结果给C或者网络延迟导致超时，个服务之间依赖性特别强，我们应该用消息队列，去解耦。**

**2、日志记录：通过日志记录去记录每个服务一次请求需要多长时间，是否因为某个请求时间过长导致接口超时。**

**3、负载均衡：多次访问 请求量大，我们需要进行负载均衡。**

**4、sql优化：看我们调用接口时是否连接数据库，多次调用数据库压力比较大，可将请求频繁的服务放入redis进行缓存。**

**消息队列**

**什么是RabbitMQ？为什么使用RabbitMQ？**

RabbitMQ是一款开源的，Erlang编写的，消息中间件；

可以用它来：解耦、异步、削峰。

**RabbitMQ 使用场景**

解耦（为面向服务的架构（SOA）提供基本的最终一致性实现）

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。

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

引入消息队列后

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

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

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

（如果问下订单成功之后，等到访问库存时候，这时发现库存不足怎么办，可以在下订单时候加判断，如果订单数量大于内存，就不让下订单成功，这样就解决了。）

异步提升效率

场景说明：用户注册成功·s

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(多线程)

（1）串行方式：将注册信息写入数据库成功后，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。这种做法让客户端等待没有必要等待的东西。影响用户体验度

（2）并行方式(使用多线程)：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

引入消息队列后

引入消息队列，将发送邮件,短信不是必须的业务逻辑异步处理

引入消息队列后，用户的响应时间就等于写入数据库的时间+写入消息队列的时间(可以忽略不计),引入消息队列后处理后,响应时间是串行的3倍,是并行的2倍。

流量削峰

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：

系统其他时间A系统每秒请求量就100个，系统可以稳定运行。系统每天晚间八点有秒杀活动，每秒并发请求量增至1万条，但是系统最大的处理能力只能每秒处理1000个请求，于是系统崩溃，服务器宕机。

之前架构：大量用户（100万用户）通过浏览器在晚上八点高峰期同时参与秒杀活动。大量的请求涌入我们的系统中，高峰期达到每秒钟5000个请求，大量的请求打到MySQL上，每秒钟预计执行3000条SQL。但是一般的MySQL每秒钟扛住2000个请求就不错了，如果达到3000个请求的话可能MySQL直接就瘫痪了，从而系统无法被使用

引入消息队列后 ：

100万用户在高峰期的时候，每秒请求有5000个请求左右，将这5000请求写入MQ里面，系统A每秒最多只能处理2000请求，因为MySQL每秒只能处理2000个请求。系统A从MQ中慢慢拉取请求，每秒就拉取2000个请求，不要超过自己每秒能处理的请求数量即可。MQ，每秒5000个请求进来，结果只有2000个请求出去，所以在秒杀期间（将近一小时）可能会有几十万或者几百万的请求积压在MQ中。这个短暂的高峰期积压是没问题的，因为高峰期过了之后，每秒就只有50个请求进入MQ了，但是系统还是按照每秒2000个请求的速度在处理，所以说，只要高峰期一过，系统就会快速将积压的消息消费掉。我们在此计算一下，每秒在MQ积压3000条消息，1分钟会积压18万，1小时积压1000万条消息，高峰期过后，1个多小时就可以将积压的1000万消息消费掉。

作用:

1.用户的请求,服务器收到之后,首先写入消息队列,加入消息队列长度超过最大值,则直接抛弃用户请求或跳转到错误页面.

2.秒杀业务根据消息队列中的请求信息，再做后续处理.

**RabbitMQ基础概念**

消息队列服务通常有三个概念： 提供者、消息队列、消费者者

RabbitMQ 在这个基本概念之上在提供者和队列之间,加入了交换器 (Exchange)。这样提供者和消息队列就没有直接联系转而变成提供者把消息发给交换器，交换器根据调度策略再把消息转发给消息队列

常用的交换机有以下三种

Direct Exchange

直连型交换机，根据消息携带的路由键将消息投递给对应队列

Fanout Exchange

扇型交换机，这个交换机没有路由键概念，就算你绑了路由键也是无视的。 这个交换机在接收到消息后，会直接转发到绑定到它上面的所有队列

Topic Exchange

主题交换机，这个交换机其实跟直连交换机流程差不多，但是它的特点就是在它的路由键和绑定键之间是有规则的：

\* (星号) 用来表示一个单词 (必须出现的)

# (井号) 用来表示任意数量（零个或多个）单词

通配的绑定键是跟队列进行绑定的，举个小例子

队列Q1 绑定键为 \*.TT.\* 队列Q2绑定键为 TT.#

如果一条消息携带的路由键为 A.TT.B，那么队列Q1将会收到；

如果一条消息携带的路由键为TT.AA.BB，那么队列Q2将会收到；

**消息确认机制**

什么是消息确认ACK

如果在处理消息的过程中，消费者的服务器在处理消息的时候出现异常，那么可能这条正在处理的消息就没有完成消息消费，数据就会丢失。为了确保数据不会丢失，RabbitMQ支持消息确定-ACK。

ACK的消息确认机制

ACK机制是消费者从RabbitMQ收到消息并处理完成后，反馈给RabbitMQ，RabbitMQ收到反馈后才将此消息从队列中删除。

1.如果一个消费者在处理消息出现了网络不稳定、服务器异常等现象，那么就不会有ACK反馈，RabbitMQ会认为这个消息没有正常消费，会将消息重新放入队列中。

2.如果在集群的情况下，RabbitMQ会立即将这个消息推送给这个在线的其他消费者。这种机制保证了在消费者服务端故障的时候，不丢失任何消息和任务。

3.消息永远不会从RabbitMQ中删除，只有当消费者正确发送ACK反馈，RabbitMQ确认收到后，消息才会从RabbitMQ服务器的数据中删除。

4.消息的ACK确认机制默认是打开的。

ACK机制的开发注意事项

如果忘记了ACK，那么后果很严重。当Consumer退出时候，Message会一直重新分发。然后RabbitMQ会占用越来越多的内容，由于RabbitMQ会长时间运行，因此这个"内存泄漏"是致命的

**session和cookie的区别**

1、cookie数据存放在客户的浏览器上，session数据放在服务器上

2、cookie存储在浏览器里不安全，session相对安全

3、cookie保存的数据大小不能超过4K，session理论上来说没有限制

4、cookie是预先设置的生存周期或永久的保存于本地的文件、session是浏览器页面一关 ,session就会消失

我们以前在做**单体架构**oa项目时，用的就是session和cookie,因为http协议是无状态的，我们要把服务端和客户端建立某种联系，知道谁是服务端，谁是客户端。并且一直把用户信息存储在session服务端，对于服务端来说压力也比较大。

随着我们开始接触**分布式架构**，这时session和cookie已经不管用了，因为涉及到了跨域的问题，所谓的跨域简单来说：一个域请求另一个域的数据

跨域问题的出现是因为浏览器的同源策略问题（浏览器要保护用户信息），浏览器请求必须遵循同源策略：同协议、同域名、同端口。

而此时我们做的登录时基于JWT做的，JWT可以直接理解为加密工具，就像单体架构时每次发送请求都会自动带上cookie一样，分布式架构每次发送请求都会带上token值，如果再想要保护安全，我们可以在登录成功后再次使用base64进行加密。

**JWT的登录认证**

在我们做单体架构时，登录之后的用户信息是保存在服务端session中的，每次发送请求都会自带cookie值进行验证。随着我们现在用分布式架构，session和cookie已经不适用了。而我们又想保证用户信息的安全，所以使用到了JWT进行登录验证。JWT可以理解为加密工具，在我们的用户登录成功之后，把用户对象放进去，用JWT声明token令牌，把值传到前台，并存入sessionStorage，这样我们每次发送请求时都自带token值，防止越权访问。我们现在后台用的就是分布式架构，通过zuul网关去验证token，zuul网关就相当于拦截器+过滤器，我们创建一个类去继承zuulFilter过滤器，重写方法，在run方法中我们通过request.getParameter获取token值，通过JWT去解密验证token，如果token为空，则用户没有进行登录，如果token值解密错误，则用户非法访问或者token值被人为改过。当然token也有失效时间，默认是30分钟，这时解密的token值可能就为空了，但能确定不是非法访问。如果认为还是不安全，我们也可以在登录成功之后用base64再进行加密。

补充：①**sessionStorage与localStorage的区别？**

localStorage的生命周期是永久性的，即使关闭浏览器，也不会让数据消失，除非主动的去删除数据；

sessionStorage 的生命周期是在浏览器关闭前，在整个浏览器未关闭前，其数据一直都是存在的；

sessionStorage不在不同的浏览器窗口中共享；localstorage在所有同源窗口中都是共享的；

**线程和线程池**

**线程**

**1、什么是单线程和多线程？**

单线程，只有一条线程在执行任务，这种情况在我们日常的工作学习中很少遇到，所以我们只是简单做一下了解

多线程，创建多条线程同时执行任务，这种方式在我们的日常生活中比较常见。

2、并行和并发的区别

并行：从宏观方面来说，并行就是同时进行多种时间，实际上，这几种时间，并不是同时进行的，而是交替进行的，而由于CPU的运算速度非常的快，会造成我们的一种错觉，就是在同一时间内进行了多种事情

并发：则是真正意义上的同时进行多种事情。这种只可以在多核CPU的基础下完成。

3、多线程的安全问题

还有就是多线程的安全问题？为什么会造成多线程的安全问题呢？我们可以想象一下，如果多个线程同时执行一个任务，那么意味着他们共享同一种资源，由于线程CPU的资源不一定可以被谁抢占到，这时，第一条线程先抢占到CPU资源，他刚刚进行了第一次操作，而此时第二条线程抢占到了CPU的资源，那么，共享资源还来不及发生变化，就同时有两条数据使用了同一条资源，具体请参考多线程买票问题。这个问题我们应该如何解决那？

  有造成问题的原因我们可以看出，这个问题主要的矛盾在于，CPU的使用权抢占和资源的共享发生了冲突，解决时，我们只需要让一条线程占用CPU的资源时，阻止第二条线程同时抢占CPU的执行权，在代码中，我们只需要在方法中使用同步代码块即可。

**线程池**

1、什么是线程池？

在一个应用程序中，我们需要多次使用线程，也就意味着，我们需要多次创建并销毁线程。而创建并销毁线程的过程势必会消耗内存。而在Java中，内存资源是及其宝贵的，所以，我们就提出了线程池的概念。

2、如何创建线程池？

Java中已经提供了创建线程池的一个类：Executor

而我们创建时，一般使用它的子类：ThreadPoolExecutor.

public ThreadPoolExecutor(int corePoolSize,  **核心线程数量**

                   int maximumPoolSize, **线程池中可以容纳最大线程数量**

                   long keepAliveTime,  **除了核心线程之外的其他的最长可以保留的时间**（因为在线程池中，除了核心线程即使在无任务的情况下也不能被清除，其余的都是有存活时间的，意思就是非核心线程可以保留的最长的空闲时间）

                              TimeUnit unit,**计算这个时间的一个单位**

BlockingQueue<Runnable> workQueue,  **等待队列**

ThreadFactory threadFactory,  **创建线程的线程工厂**

                              RejectedExecutionHandler handler **是一种拒绝策略**)

3、handler的拒绝策略

第一种AbortPolicy:不执行新任务，直接抛出异常，提示线程池已满

       第二种DisCardPolicy:不执行新任务，也不抛出异常

       第三种DisCardOldSetPolicy:将消息队列中的第一个任务替换为当前新进来的任务执行

       第四种CallerRunsPolicy:直接调用execute来执行当前任务

4、四种常见的线程池

CachedThreadPool:可缓存的线程池，该线程池中没有核心线程，非核心线程的数量为Integer.max\_value，就是无限大，当有需要时创建线程来执行任务，没有需要时回收线程，适用于耗时少，任务量大的情况。

SecudleThreadPool:周期性执行任务的线程池，按照某种特定的计划执行线程中的任务，有核心线程，但也有非核心线程，非核心线程的大小也为无限大。适用于执行周期性的任务。

SingleThreadPool:只有一条线程来执行任务，适用于有顺序的任务的应用场景。

FixedThreadPool:定长的线程池，有核心线程，核心线程的即为最大的线程数量，没有非核心线程

**多线程导出（业务）**

**1.为什么需要多线程导出？**

因为订单需要数据迁移，而且订单量比较大。如果单线程导出的话会速度非常慢。

**2、多线程导出都用到了哪些技术？**

用到了poi，自定义注解，java反射，多线程和线程池。

**3、业务**

线程的实现方式的话就是实现一个Runnable接口，还有继承Thread类重写它的run方法，在里面写具体实现，在主线程中创建线程调用一个start方法就创建好了，线程的五个状态的话就是，你创建Thread的时候就是新建状态，然后调用start方法的时候就就绪状态，当cpu把时间片分配给你的时候，这个线程就是运行状态，其他线程的话就在就绪状态中排队，当你执行完就到了死亡状态，如果你没有执行完，cpu把时间片分给了别人，这时候你就会回到就绪状态，等待着cpu选到你，还有个阻塞状态的话（sleep,wait），就是当你加锁的情况下，a线程拿到了锁，但是cou分配的时间到了，你没执行完，就会把资源锁住，这时候b线程被cpu选到了，它去执行一看被锁了，这时候b线程就会进入到阻塞状态，直到等到a线程执行完，把锁给释放了b线程才会从阻塞状态出来

我当时的话是用它做了一个poi的导出，当时一开始是正常的通过反射加自定义注解通过这个HSSFWorkbook做一个导出，但是发现效率太慢，数据过多，之后我就把每创建一个Excel就启动一个线程，然后每次查询只查一个sheet页的数据，这样确实快了很多，但是还是不完美，之后我又加了线程池，把线程数设置最大线程数，这样线程也不需要频繁的创建和销毁了，就快完成的时候了，一下载发现导出来的是个空的，之后一想，多线程是异步的，线程还没完成，主线程是不等待的，直接执行了，然后我就百度搜索解决方案，最后找到了CountDownLatch这个方法，把线程的总数给它 ，在主线程最后下载的地方调用一个await方法，这个方法就是让你等待，当CountDownLatch数量到0的时候才会执行， 然后再线程内调用countDown方法，这个就是每次执行了这个方法就会线程减一当，减到0就证明所有线程走完了，然后主线程就会执行，加了这个之后就把这个问题解决了

**多线程定时任务(分布式锁)**

定时任务 （发邮件 发短信 会员续费）

做多线程定时任务我们就要涉及到了分布式锁的问题，我们是通过redis的getAndSet去解决的。

上流程

我当时是做的一个定时发邮件的任务，就是用户生日的时候发一封祝福邮件

到了这个时间段就一下发了好多封邮件，每个集群的服务都调用了这个方法，这时候我们发现，用户会重复收到多条一样的短信祝福，为了解决这个问题，我们使用了redis中的getAndSet分布式锁去解决的，当第一条请求过来的时候，此时getAndSet前 key和value都为空，我们返回一个空值，然后set 一个key1 value1,

我们的逻辑判断就是当返回的key==null的时候程序继续往下走，去查询所有的用户信息，判断当前日期是否是某个会员的生日，如果是就给会员发送祝福邮件。

然后我们还要给key设定一个失效时间。时间过了，下一个请求就进来了，而这个请求发现这个线程已经有值了(返回的旧值已经有值了)，就不再继续给此用户发送邮件了。这样就解决了给一个用户重复发邮件的问题了。

**权限管理**

前端动态权限栏

简单来说就是每个人根据每个人的权限看到的内容是不同的

具体实现是 创建五张表 表结构 表的关系 表如何查询的 然后怎么展示 来设置的

五张表有用户表 角色表 权限表 角色权限表 用户角色表

表结构就是

用户表 账号 密码

角色表 id 和角色名称

权限表 id 和权限名称 和地址

角色权限表 角色id 和权限id

用户角色表 用户id 和角色id

查询是 用户登录以后 发送请求时把 用户的id传到后台 然后根据id查找用户的角色 在根据角色查找用户的权限

然后返回值以后把 权限信息以拼接的方式拼接到导航栏上面

**mysql优化**

**1、表设计上 存储引擎和字段类型**

(1) 存储引擎常用的有两种MylSam和InnoDB

① Mylsam不支持事务，不支持外键，采用表锁，侧重于性能；

② InnoDB支持事务，支持外键，采用行锁，侧重于事务；根据需求选择合适的存储引擎；

(2) 字段类型  给字段设置合适的数据类型，在合适的情况下使用整数型来代替字符串

**2、功能上  创建索引**

(1) 索引简单说在列上添加标识，帮助Mysql高效获取数据。索引就是一种数据结构。

**l 或者~~通俗点理解成是字典的目录~~。**

create index idx\_test03\_c1234 on test03(c1,c2,c3,c4);

show index from test03;

(2) 索引的优点：

提高数据检索的效率，降低数据库的IO成本。

通过索引列对数据进行排序，降低数据排序的成本，降低了CPU的消耗

(3) 索引的缺点：

虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为在更新表时，MySQL不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

3. 为什么要SQL优化

性能低、执行时间太长、等待时间太长、SQL语句写的烂、关联查询太多(设计缺陷或者一些不得已的需求)、索引失效、服务器参数设置不合理

4、什么情况下应该建索引？

**l 主键自动建立唯一索引；**

**l 频繁作为查询条件的字段应该创建索引**

**l 查询中与其它表关联的字段，外键关系建立索引**

**l 查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度**

5、什么情况下不应该建索引？

**l 表记录太少**

**l 经常增删改的表或者字段**

**l Where 条件里用不到的字段不创建索引**

**6、MySQL索引分类**

**l 单值索引：**即一个索引只包含单个列，一个表可以有多个单列索引。

**l 唯一索引：**索引列的值必须唯一，但允许有空值

**l 主键索引：**设定为主键后数据库会自动建立索引

**l 复合/组合索引：**即一个索引包含多个列

7、分库分表

当我们一张表的数据达到上百万甚者上千万，那么即便是建立索引查询效率依然会很慢，我们就可以进行分表，分表可以分为水平分表和垂直分表

水平分表就是创建多个结构相同的表，然后表名是动态的，这样子就可以实现分表,把数据分开。查询效率就可以提高

垂直分表就是一个表中有常用数据和非常用数据，我们可以把这两个数据分开储存建立两个表，两个表是一对一的关系，这样也可以提高查询效率

============面试问题==============

你们用的spring cloud的版本:  
1.5  
  
你咋没用2.0呢，1.5都淘汰了?  
我们当时项目用的就是1.5的  
  
你知道gateway吗?【网关】  
springCloud 2.0里面的gateway就是1.5里面的zuul  
  
你知道consul?【注册中心】  
springCloud 2.0里面的consul就是1.5里面的eureka  
  
你们的springCloud的配置中心用的是啥？  
用的是携程的 Apollo配置中心。  
如果一个配置发生变化，则可能需要修改很多的服务配置。为了解决这个问题，spring cloud提供配置中心。

具体咋用的？

============以下了解============

各种锁

在多线程的程序中如果涉及到共享资源的访问，就会引发线程安全问题，说白了就是执行的结果和预期的结果不一致。遇到多线程并发访问的线程安全问题，我们通常可以采用锁的方式进行解决。

最经常用的就是synchronized同步锁，我在写单例设计模式的时候就会用到。

【通过双重判定锁可以保证在线程安全的情况下，极大的提升程序的性能。通过synchronized代码块，减小锁定的范围从而比将其加到方法上提高了性能，通过最外层的if判断可以使其后面的线程直接返回所需要的单例，而无需再次进行锁定。比如A,B两个线程过来后同时进入方法，都满足最外层的判断，这时候两个线程抢占锁资源，比如A获取了锁，则B只能在外面等待，A在最里层的if判断中，因为最开始对象为null,所以会给对象赋值。当A线程离开了同步代码块后，会自动释放锁；这时候B线程获得锁资源，同样进行最里层判断，因为对象已经不为null,则可以直接返回该对象不需要再次创建，这就保证了单例。如果后续又有C线程过来，则在最外层的判断中已经不为null,就可以直接返回对象，无需再次锁定。这样也就提高了性能！！!

但是这种双重判断锁方式自身会有一个缺陷，会出现指令重排序的问题。比如 Product p1 = new Product(),这个过程被分为3步。第一步在堆里面开辟空间，第二部初始化对象，第三步将p1指向初始化后的对象。但是由于指令重排的问题，可能它的执行顺序就不再是1,2,3而变为了1,3,2。在这个时候会出现p1指向堆空间中的对象还没有被完全创建。如果这个时候另一个线程过来了，那么最外层的if判断中，等于null的条件就不成立，那么就会直接返回一个还 不完全 的对象，程序在后续调用这个不完全对象的时候肯定会报错。为了解决这个指令重排序的问题，我们可以通过volatile这个关键字，来避免指令重排！！！】 这段可以根据个人情况，进行省略！！！

像java里面的hashtable和vector都是用的synchronized来保证线程安全。除此之外，还有读写锁。读写锁表示有两个锁，一个是读操作相关的锁，也称为共享锁；另一个是写操作相关的锁，也叫排它锁。

多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥。

通过分离读锁和写锁，使得并发性相比一般的排它锁有了很大的提升。也就是说多个Thread可以同时进行读取操作，但是同一时刻只允许一个Thread进行写入操作。通过lock()方法加锁，通过unlock()方法解锁。

除此之外还有分布式锁，普通的锁[synchronized]是为了解决多线程访问产生的安全问题，分布式锁是为了解决多进程【多个tomcat】多线程访问产生的安全问题。我们在项目中使用Redisson来完成分布式锁。

除此之外从数据库层面来说有悲观锁，乐观锁。包括有表锁和行锁。

悲观锁,

每次去查询数据的时候都认为别人会修改，

所以每次在查询数据的时候都会上锁，

这样别人想拿这个数据就会阻塞直到它拿到锁。

传统的关系型数据库里边就用到了这种锁机制，

比如通过select ....for update进行数据锁定。

乐观锁,

每次去查询数据的时候都认为别人不会修改，

所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有

去更新这个数据，可以使用版本号，时间戳等机制。

乐观锁适用于多读的应用类型，这样可以提高吞吐量。