# B简历：

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

### 基本信息

|  |  |  |  |
| --- | --- | --- | --- |
| 基本情况 | | | |
| 姓名 | 王鹏飞 | 性别 | 男 |
| 出生日期 | 1993-06-26 | 属相 | 鸡 |
| 现住地 | 天通苑 天通北苑三区 |  |  |
| 大学情况 | | | |
| 学校 | 中北大学信息商务学院 | 专业 | 计算机科学与技术 |
| 入学时间 | 2011-9 | 毕业时间 | 2015-7 |
| 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)**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)**、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 | **山西省晋中市工业园区8号路15号** | 校长 | **王召巴** |
| 去学校的方式： | **就在晋中火车站旁边** | | |
| 上小学日期： | **1999年9月 6岁六年** | | |
| 上初中日期： | **2005年9月 12岁三年** | | |
| 上高中日期： | **2008年9月 15岁三年** | | |
| 上大学日期： | **2011年9月 18岁四年** | | |
| 毕业时间： | **2015年 7月 22岁** | | |
| 入学时间： | **2011年9月** | | |
| 毕业时间： | **2015年7月** | | |
| 在校学生： | **5000左右** | | |
| 有教师： | **不太清楚** | | |

|  |  |
| --- | --- |
| 第一家公司 ： | 北京朗维计算机应用技术开发有限公司 |
| **地址：** | 北京市朝阳区德外马甸裕民路18号北环中心1302（只说大厦名称） |
| **入职时间：** | **2015年4月** |
| **离职时间：** | **2017年6月** |
| **公司性质：** | **民营** |
| **职位：** | **Java工程师** |
| **项目经理名字：** | **石宇文** |
| **薪资方面：** | **刚刚毕业转正以后工资是6K，过完年涨到7K** |
| **路线：** | **坐5号线到惠新西街南口，换10号线到海淀黄庄，再换4号到魏公村（1个小时）** |

|  |  |
| --- | --- |
| **第二家公司情况：** | 北京明润华创科技有限责任公司(只说公司简称) |
| **公司地址**： | 北京市石景山区政达路北方中惠国际中心D座7层 |
| **公司规模：** | **60人左右，我们项目组6人，一个项目经理，1名前端，1个测试，1个运维，其他java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：** | 5号线到东单，转乘1号到八宝山，直行1公里到万达广场，后面就是我们公司 |
| **入职时间：** | **2017年6月** |
| **离职时间：** | **2019年11月初** |
| **职位：** | **Java工程师** |
| **项目经理名字：** | **于风** |
| **税前：** | **14K** |
| **税后：** | **13K** |
| **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(9K)** |

|  |
| --- |
| 四年涨薪：  **2015年工作的时候是5K，税后还是5K，**  **转正6K左右**  **2016上半年涨了1K左右 ，总7K左右**  **2017年6月涨了2k,总的9k左右**  **2018年4月跳槽涨了1K左右 总10K**  **2018年12月涨2K总 12K左右**  **2018年9月工作一年涨了1K 13k**  **2019年7月工作一年涨了1K 14k**  **2019年11月离职时工资为14K** |

**面试常见问题：**

|  |
| --- |
| **你为什么从上家公司离职？** |
| **之前在外包公司，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:** 团队协作能力强**抗压自学能力强（有一次项目时间很紧，经常工作到12点半之后，连续攻坚一个月终于把项目完美的完成了。），**自学能力强  **自学能力强** (**1.从官网上下载需要的文件。2.导入相关的文件到项目中。3.在需要用到的文件中引入具体的js，css。具体在做例子的时候先做静态例子，再和后台结合做出来动态数据的例子。特别在做动态数据例子的时候，我们可以通过结合官网上提供的demo,通过f12观察这个几个问题:**  **<1>js插件怎么向后台发送请求；**  **<2>传递了哪些参数以及参数的含义;**  **<3>后台怎么接受参数;**  **<4>后台接受到参数后怎么处理,才是js组件所需的响应格式;**  **<5>如何将json数据响应给前台;**  **<6>前台如何接受到该json数据；**  **<7>接受到数据之后如何处理，才能将数据展现出来。按照这个思路和流程，我们在学习第三方的js组件时，就能够快速的掌握并融入到项目中。)缺点: 这几年做程序，因为要不断的对代码进行验证确认，整体达到一定效果才可以，所以感觉自己现在有点强迫症** |
| **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** |
| **能** |
| **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14K 税后13K左右 扣了1K左右的税 |
| 你的五年规划是啥？ |
| **近期以提升技术为主，然后对公司的业务也要尽快掌握，然后向项目经理方面发展。** |
| 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**  300  **饭/餐补**  30/天  **电脑补**  300 |

**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项目，负责了其中的几个模块，我先给你说下框架方面的东西吧

**框架篇**

**首先讲下对springboot的理解**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

@EnableAutoConfiguration

@ComponentScan

@Configuration是基于JavaConfig形式的Spring Ioc容器的配置类，可以把它看成xml配置文件中的beans标签。 @Configuration写到类上面，在类中的方法上如果写了 @Bean注解，那么它的返回值将作为一个bean注册到Spring的IoC容器，方法名默认作为bean的id。

@ComponentScan这个注解对应XML配置中的context:component-scan元素， 说白了它的作用就是自动扫描并加载符合条件的组件,比如 @Component和 @Service等或者bean定义，最终将这些bean定义加载到IoC容器中。

我们可以通过basePackages来指定 @ComponentScan 自动扫描的范围,如果不指定，则默认Spring框架实现会从声明 @ComponentScan所在类的package进行扫描。这也是SpringBoot的启动类最好是放在root package下的原因。

@EnableAutoConfiguration 这个注解是借助 @Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器中。

Springboot的简单实用：在biz/service业务层上通过 @Transactional(rollbackFor = Exception.class)指明了事务的控制，使其在抛出Exception异常时能够回滚。默认情况下事务只对运行时异常进行回滚所以才会通过rollbackFor = Exception.class将其改为对所有的异常都进行回滚。还需要在Mapper接口上通过注解的方式完成持久层的常规操作；如通过 @select 进行查询， @Results 将查询后的结果和实体类中的属性进行映射， @insert 进行插入， @update 进行更新， @delete 进行删除，@selectKey 可以在插入后获取新插入的记录的id值并将其赋值给实体类中的id属性。想要在service层注入 Mapper 层的接口，则还需要在启动类上加入 @MapperScan这个注解对其Mapper层的接口进行扫描。

**讲述对springcloud微服务的理解**

在具体开发一个微服务的时候，用的是SpringBoot，首先在项目的pom文件中同样需要引入Eureka的依赖，并且在application.properties文件中通过spring.application.name指明自身的服务名以及注册中心的地址，这样才能将其注册到注册中心中被其他消费者访问。

在项目中我们用Eureka建立注册中心项目，在pom文件中引入Eureka的依赖，之后在启动类中加入 @EnableEurekaServer的注解证明该项目的作用是作为注册中心使用，并且在对应的application.properties中指明Eureka注册中心服务器的地址，方便提供者和消费者对其进行访问，为了增强注册中心的高可用性，我们做了Eureka集群，防止因为单台Eureka导致的单点故障问题。

如果没有注册中心，客户端通过ip地址访问服务端，如果服务越来越多，调用也会越来越复杂，一旦ip地址发生变动，则所有用到的地方都需要发生改变，维护起来会特别麻烦；如果服务端宕机了，所有访问的客户端也都会受到影响。通过注册中心可以让服务端注册到注册中心上去，并且可以进行负载均衡；这样就可以达到高可用以及提高并发的 目的。客户端 也不需要 直接 调用服务端，而是通过注册中心获取地址列表，从地址列表中选一个服务端接口进行调用，后续的维护各方面也会比较方便。

总体来说在项目中我们使用SpringBoot开发微服务并结合SpringCloud的注册中心Eureka和支持负载均衡的客户端Feign来完成整个项目的开发,使用Zuul充当网关；其中Feign是为了在微服务之间进行调用的时候使用，而Zuul发挥了反向代理和负载均衡的作用，前端发送的所有请求经过Zuul，进行安全验证，通过后再转发到具体的微服务上。

**讲项目的统一管理**

之前我们在做项目的时候，是把所有的接口都写到一个API接口项目中，那时候处理接口安全问题，我们是写了一个AOP切面或者拦截器，统一对控制层的各个方法进行安全验证，但现在用到SpringBoot+SpringCloud微服务，每个功能模块都是一个单独的微服务项目，如果还用之前AOP切面或者拦截器的方式，就会出现很多重复代码，维护起来也特别麻烦。

因为外部客户端的请求都是通过Zuul网关路由到具体的微服务，所以为了保证微服务的安全，我们就在Zuul中自定义了过滤器，对所有微服务的安全进行统一的处理，还有，因为涉及到前后端分离，前端项目访问后端微服务涉及到跨域问题，所以我们在Zuul中也自定义了关于跨域的过滤器，进行统一处理。

具体是这么做的，首先你得先定义一个继承于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方法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；我们的微服务安全认证以及跨域这块都是用的pre类型的过滤器，这样对非法请求，就可以在发送到具体的微服务之前拒绝它；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()方法返回一个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。

在run方法中，首先获取一个RequestContext对象，之后就可以通过调用它的getRequest()方法来获取request对象；这样就可以获取头信息，按照基于token的方式进行接口的安全验证。

过程中特别需要注意的就是当验证不通过的时候，需要通过fastjson将要响应的数据转换为json格式的字符串，之后设置响应的内容类型为application/json并且指定utf-8的编码方式，用来处理中文乱码问题；通过setResponseBody将Json格式的字符串设置为响应的内容，最后通过setSendZuulResponse为false，禁止路由转发。如果想要将zuul过滤器中的数据传递给后端微服务中使用，则需要通过addZuulRequestHeader方法来进行，而后端微服务中就可以通过request.getHeader来获取值。这里面需要特别注意的就是，如果传递的数据中含有中文则需要通过URLEncoder进行utf-8的编码，同样在获取数据后也需要通过URLDecoder进行解码。

最后想要使自定义的过滤器生效，得进行相关的配置。我通过创建一个配置类，并且在类上通过 @Configuration 和 方法上的 @Bean 结合起来，完成自定义filter的配置。

**讲微服务之间的相互调用及雪崩问题处理**

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。就像咱们现实生活中，一个路口堵车，就会导致整条街慢慢拥堵，再后来影响的范围就会越来越大。可以用SpringCloud中的Hystrix来解决这个问题。

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。

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。

降级说白了就是进行熔断后需要执行本地的方法。也是为了保证微服务的高可用。[说白了就像咱们看的打仗时候的急行军，为了保证按时到达作战地点，我就把身上其他的干粮，背包全部丢掉，只带上最基本作战的武器，快速前进保证最终的结果。可以省略]

在Hystrix中, 有两种资源隔离方式，信号量隔离和线程池隔离，我们主要用的是线程池隔离. 通常在使用的时候我们会根据调用的远程服务划分出多个线程池. 比如调用产品服务的Command放入A线程池, 调用会员服务的Command放入B线程池. 这样就可以将运行环境隔离开。就算调用服务的代码存在bug或者由于其他原因导致自己所在线程池被耗尽, 也不会对系统的其他服务造成影响。

我们在项目中是用Feigin进行微服务之间的调用的，又因为Feign集成了Hystrix,所以我们当时通过在配置文件中开启hystrix,并且在具体使用Feign的接口上通过 @FeignClient 中的 Fallback指定降级时候需要执行的类即可。这样在熔断后就会自动调用Fallback类中对应的降级方法，降级方法中，可以记录下日志并且返回null。这样使用Feign的客户端在调用 远程微服务 的方法后，判断返回值是否为null,如果是null则证明走了降级方法，进行特殊处理就行。

Feign还拥有负载均衡的特性，说白了它是靠Ribbon进行的负载均衡。在 @FeignClient的注解上指明要调用的微服务的名字，这样就可以通过该服务名从注册中心Eureka中获取对应的地址列表，方便进行负载均衡的调用。

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。

性能调优篇

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

## 当时做这个项目的时候，考虑了很多东西，这个模块相对比较复杂，像表设计，订单接口幂等性问题，商品的超卖问题，包括订单也都是用户登录后才能进行的操作等等。

首先说下高并发情况下产生的超卖问题吧：超卖是这样产生的，同时下订单时先对商品库存进行查询、满足条件才能下单，但在高并发的情况下，可能好几个查到了同样的结果，同时下单，将数据库中的库存买到了负数。为了杜绝这种情况呢，我做了以下的处理：【超卖】

首先从redis中取出用户的购物车数据，如果为空，则响应给客户端一个用户购物车为空的提示信息，如果不为空则取出用户对应的购物车的json数据，将其转换为java对象。

首先通过年月日时分的时间戳加上雪花算法生成一个订单的唯一标识，然后从购物车对应的java对象中取出购物车中的商品列表，这个商品列表其实对应的就是订单明细，循环遍历商品列表，减库存，如果减库存成功，则将该商品插入订单明细表，如果不成功就不需要插入订单明细表了，而且还需要从购物车中剔除该商品，更新购物车的总价格和总件数，并将该库存不足的商品加入一个列表，用于响应给前端进行提示。

将更新后的购物车信息中的总价格和商品总件数插入到订单表中，并设置创建时间为当前时间，当前订单状态为未支付。

最后插入支付日志表，这个表记录的是支付时候的相关信息，比如outTradeNo,payMoney等，将支付日志表中的订单id设置为刚才插入的订单id,将支付状态设置为 未支付，并将该支付日志插入到redis缓存中，方便后续在支付的时候直接从缓存中取出来进行使用。紧接着就清空该用户对应的redis中的购物车数据。因为当时购物车数据是以hash的方式存储的，所以通过hdel进行删除就行了。

在这个过程中要注意，如果整个购物车中的商品都库存不足，则直接提示前端，下单失败，库存不足；也就不需要再插入订单表和支付日志表了。

我当时采用的是基于数据库乐观锁的方式进行订单超卖的处理。核心思路就是在update语句中加入库存量是否大于或等于购买量的where条件。比如原来更新数据库中的商品库存我这样写，先查询下该商品对应的库存量，如果库存量大于等于购买量则执行update 商品表 set 库存=库存-购买量 where 商品id=购物中的商品id;否则提示库存不足。

表面看上去没有任何问题，但在遇到多线程高并发的时候，就会出现多个线程同时查询商品对应的库存量，都符合条件，所以多个线程都会依次进行update语句，最后就造成商品可能为负数，出现商品超卖的情况。

我是通过将update语句改造为 update 商品表 set 库存=库存-购买量 where 商品id=购物中的商品id and 库存量>=购买量，然后根据update的返回值，如果返回值为0，则证明没有执行该sql语句，证明库存已经不足了。这样即便在多线程的时候也不会出现超卖问题。

项目中很多接口是用户登录后才能进行操作的，所以我们采用的是基于token的用户登录，并且自定义了一个注解，这样在需要登录的控制层的方法上直接加上这个自定义注解配合自定义的拦截器，就可以达到先登录才能访问接口的目的，使用起来非常方便！！！

**【幂等性/重复下单】**

订单接口要考虑重复下单的问题，超卖的问题，以及把redis中存储的购物车数据取出来，插入订单明细表，订单表以及支付日志表。

关于重复下单这块，说白了就是要考虑接口的幂等性。幂等性我是这样理解的，就是对接口调用一次和调用多次，产生的结果都是一样的。像查询方法，就天生具有幂等性的特点。

但下订单属于添加操作，对于添加操作本身不具备幂等性的特点，所以需要我们特别处理下，如果不处理就会出现因为网络抖动或者服务器重发请求，造成虽然用户只操作了一次，但是可能最终发送了多次添加订单的请求，这样就会让用户感觉到很奇怪，明明我就下了一个订单，怎么会出现两个一模一样的订单呢。

对于接口的幂等性我是基于redis+token的方式解决的。具体的思路是这样的。

首先我会写一个创建token的接口，这个接口的代码也特别简单，就是生成一个uuid作为token，然后响应给客户端，并且将该token存储到redis中，而且设置它的过期时间为10分钟，这样在客户端中就可以调用这个接口生成一个唯一的token标识,并在后续的请求中将该token作为header中的信息传入到接口的服务端进行相关验证。

接着我写了一个自定义的注解，这个注解没有什么实质性的内容，说白了就是一个标识，只要是加上了这个注解的方法，都会在我自定义的拦截器中进行幂等性的处理。

幂等性拦截器中的判断是最核心的。首先我会判断请求中是否包含token头信息，如果没有就响应给客户端一个提示，头信息丢失。如果存在则获取该token的内容，调用redis中的exist方法来判断在redis中是否存在该token，如果不存在则提示token不存在。最后就是调用redis的delete方法，将该token作为key进行删除操作，如果返回的值等于0,则证明已经有其他请求发送过来了，该请求就不在是第一个发送过来的请求，就提示请求重复，否则证明是第一个请求，放行即可。

之后我会在下订单的方法上加入自定义的幂等性注解，就解决了重复下单的问题。

【表设计/冗余字段】分为两表：订单表和订单明细表 再说下订单表的设计吧，订单这个模块中包括 订单表和订单明细表，像订单表我当时设计的有这些字段：

订单id

用户id

订单支付方式

订单的总金额

订单中商品的总件数

订单的创建时间

订单的支付时间

订单交易关闭的时间

订单交易成功的时间

订单完成评价的时间

订单的状态 【未支付10 已支付20 已发货30 交易成功40 交易关闭50 已完成评价60】

订单状态描述

收件人手机号

收件人地址

邮编

收件人名字

邮费

是否需要发票

在这里面 订单id 作为主键，它的生成方法是年月日时分加上雪花算法生成的标识，这样既能从一个订单id中看到这个订单是啥时候生成的也能通过雪花算法保证整个订单编号的唯一性。

用户id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。

订单的总金额，总件数这个没啥要特别说的，这个就是当时购物车中添加的商品总价格及其总件数，但如果添加到购物车的商品在提交订单进行购买时发现库存不足，我当时的做法就是找到库存不足的商品，弹出提示框告诉用户，并在订单的总金额和总件数中要减去这些库存不足的商品，当然这些库存不足的商品也就不再插入订单明细表了。

订单的创建时间这个字段比较重要，因为在用户下单后可能不及时支付订单，那我们的系统就是在1个小时后会自动取消订单。所以就可以用当前时间减去订单创建时间，如果大于1小时就将订单状态设置为交易关闭，并在订单状态描述中记录“订单支付超时取消订单”的描述。

除此之外还有订单的支付时间，交易关闭的时间，交易成功的时间，发货时间，完成评价的时间。

订单的状态我当时设置的有

未支付 已支付 已发货 交易成功 交易关闭 已完成评价

取消订单的时候会进入交易关闭状态，可以是人工取消订单也可以是系统在1个小时后自动取消订单。

当用户支付后，如果要退款也会进入到交易关闭状态。

当用户收到货物后点击确认收货，会进入到交易成功状态，如果用户没有点击确认收货，那么是在发货时间15天后自动将订单变为交易成功状态，只有交易成功后才能对商品发表评论。

还有一些收件人的信息 包括手机号，地址，邮编，名字等等，这样做是通过这些个冗余字段，避免为了获取这些信息还要进行多表联查，因为多表联查会降低性能而通过冗余字段就可以避免多表联查，无形中也就提升了性能。

订单明细表:

订单id:

商品id:

用户id:

商品名:

商品价格:

商品个数：

商品小计:

商品图片:

我当时采用 订单id和商品id 作为联合主键，说白了也就是用多个字段来保证一条数据唯一性。

除此之外还融入了用户id这个字段，目的也是为了通过该冗余字段就能知道某个用户购买的商品明细，而不需要再连接订单表，避免多表联查。

## 购物车模块购物车这块也是我做的。同样为了提高性能，这个模块我当时做的就是基于redis的购物车，将购物车的所有数据都存储到redis中。这是个项目呢，当时从表设计到最后用postman进行接口的测试，都是我负责的。

当时用了redis中的hash结构来存储所有用户购物车的数据，咱们都知道在使用hash的时候，涉及到key,field,value这三个方面的参数信息，key这块我用了 CartInfo 这个固定的字符串来表示，在field这块为了能够表示不同用户的购物车，我用来 member:会员id来表示，在value这块我是把会员对应的整个购物车的信息都转换为Json格式进行了存储。整个购物车对应的数据结构是这个样子的。

购物车中包含 商品总件数，总价格，以及 购物车的商品项列表，在商品项中包含 商品id,商品名，商品单价，商品件数，商品图片，商品小计 等，这里面特别要注意的就是关于价格这块在java中要用bigDecimal这个数据类型，这样可以防止精度丢失。

只有会员登录后才能操作购物车，比如添加商品，删除商品等。关于登录这块，我采用的是基于token的用户登录，类似于自己写了一个jwt【json web token】。

这样用起来更加的灵活方便，在自定义的拦截器中，当通过了层层关卡的验证后，将会员信息存入到request中，这样在其他的控制层方法中就可以通过request来获取当前登录的用户信息。为了方便在每个控制层方法中获取会员信息，我采用了 spring中自定义的参数解析器，这样就避免了在每个控制层方法中都使用requst.getAttribue来获取会员信息，而是在自定义的参数解析器中写一遍即可。

具体我是这样做的，自定义了一个实现HandlerMethodArgumentResolver接口的参数解析器 类，这里面最主要有两个方法，supportsParameter和resolveArgument，在supportsParameter中我通过methodParameter获取参数的类型和我自定义的会员类进行对比如果相等则返回true,证明找到了会员类型的参数，当返回true时就会走resolveArgument方法，在该方法中，通过request.getAttribute()获取拦截器中存入的会员信息，并将其强转为指定的会员类，进行返回。最后要想使自定义的参数解析器生效，则需要在spring-mvc的配置文件中配置mvc:annotation-driven，并在其中配置mvc:argument-resolvers，通过bean class指定自定义的参数解析器类，这样就可以在控制层的各个方法中，直接写上会员类参数，就可以通过会员类获取其中的会员id等信息。

这里面涉及到 添加商品到购物车的业务逻辑 是这样的。这个接口方法接受3个参数，其中商品id和商品个数是直接从前台传过来的，而会员类则是通过自定义的参数解析器来完成赋值的。首先我会根据商品id判断该商品是否存在，如果不存在则提示商品不存在的信息，如果存在则需要判断该商品的状态是否正常，比如是否是上架状态，如果状态是下架则提示该商品已经下架。

经过这些验证后，我会判断该会员在redis中是否有对应的购物车信息，是通过封装的工具类redisUtil中的hget来获取的，如果该会员对应的购物车信息为空，则首先要创建购物车，并且将当前商品添加到购物车中，最后还要计算购物车中的总件数和总价格，通过阿里巴巴的fastjson将其转换为json格式，并通过我封装的redisUtil的工具类hset方法，将其存入到redis中的hash结构里。

如果会员对应的购物车信息不为空，则分情况进行处理。首先看看该商品是否已经在购物车中了，如果是，则找到该商品并更改商品的数量为商品原有的数量加上前台传过来的商品的数量，还要重新计算该商品的小计，以及重新计算整个购物车中商品的件数以及总价格，最后更新redis中的数据。

如果该商品不存在购物车中，则将该商品加入会员的购物车，并重新计算整个购物车中商品的件数以及总价格，最后更新redis中的数据。

这就是整个往购物车中添加商品的逻辑，至于减少商品的个数，完全可以使用和添加商品一样的方法，只不过传过来的商品的个数为负数就行了。

而且还有一些小细节，就是如果不断的减少商品个数，当商品个数减少到零的时候，就将该商品一并从购物车中删除掉。当购物车中所有的商品都被删除的时候，也同时删除整个购物车。最后都得更新redis。

## 线程

线程的状态/生命周期:

新建状态：new了一个线程

就绪状态:?调用线程对象的start方法后

运行状态：当被操作系统选中，获取cpu的使用权，执行的是run方法

阻塞状态：因为某种原因，放弃了cpu的使用权

死亡状态：线程运行结束了或者运行过程中出现异常退出了

为啥会产生线程安全问题?

当?多个线程?操作?同一个资源?才会出现线程安全问题。

什么叫线程安全，什么叫线程不安全？

当?多个线程?操作?同一个资源，最终的结果和预期值

不一样，则为线程不安全，最终结果和预期值一样，

则为线程安全！！！

## 线程池概念

## 对于高并发、多线程的问题，我们还通过线程池的方式提高程序的性能。

首先线程池可以**避免频繁的创建和销毁线程所造成的性能损耗**，原理就和数据库连接池差不多，说白了就是项目启动的时候在线程池中就已经创建好了指定数量的线程，需要的时候直接去用，用完后再放回线程池供其他程序进行使用。再者用线程池中的多线程可以处理大批量的数据，比如我要将数据库中的图片加水印或者在硬盘上批量生成文件，这时候都可以用线程池。就像要洗100个碗，你可以让一个人去干，这一个人就像是一个线程，你也可以让10个人一块去干，这就是线程池中的多线程。相比而言多线程执行的时间更短，效率更高。在项目中我们是用**ThreadPoolExecutor**来创建线程池的，它里面有几个核心的参数信息，线程池的**核心大小**，**队列**以及**线程池的最大值**。线程池的工作原理是这样的，默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。如果当前线程池中的线程数目小于核心线程数，则每来一个任务，就会创建一个线程去执行这个任务；如果当前线程池中的线程数目>=核心线程数，则每来一个任务，会尝试将其添加到队列中，若添加成功，则该任务会等待空闲线程将其取出去执行；当队列已满，添加失败，就会尝试创建新的线程去执行这个任务；这时候创建新的线程是根据线程池时的最大线程数为依据的，如果当前线程池中的线程数目达到最大线程数，则会采取任务**拒绝策略**进行处理；说的简单点就是先把核心线程数给占满了，不够用就开始往队列里面放，如果队列也占满了，就往创建的最大线程池里面放，如果也占满了就可以根据策略进行拒绝处理。

如果线程池中的线程数量大于 **corePoolSize**【核心线程】时，某线程的空闲时间超过**keepAliveTime**【存活时间】，线程将被终止，直至线程池中的线程数目不大于corePoolSize【核心线程】。

线程池中的队列，一般常用的有**ArrayBlockingQueue**【有界阻塞队列】：基于数组的队列，创建时必须指定大小；还有**LinkedBlockingQueue**【无界阻塞队列】：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

被动说：一般需要根据我们会根据任务的类型来配置**线程池大小**，如果是CPU密集型任务，就需要尽量压榨CPU，可以设为 CPU个数\*CPU核数+1，如果是IO密集型任务，参考值可以设置为2\*CPU个数\*CPU核数。

业务篇

在这个项目里我负责了会员登录的模块，购物车模块，订单模块

## 会员登录模块

我们之前做的登陆都是基于sesssion的登录，登录后把会员信息放在session中，我知道，session是存放在服务端的，当登陆的用户过多时，采用这种登陆方式就很浪费性能，所以这次我做的是基于jwt的登录，在网上有不少jwt（json web token）的登录，都有些缺陷，我就自己写了一套。

基于Token的登录呢,说白了就是用户在前台输入用户名密码后通过ajax发送请求到后台接口，后台接口在进行层层验证后，如果验证成功则直接响应相关的信息给前台，前台在ajax的成功回调函数里面接受到数据，将其暂时存入cookie中，之后在后续发送请求时，再从cookie中取出数据将其写到自定义的header中，我当时自定义的header叫(x-token),以后数据就通过header传递给后台接口的拦截器进行各种验证，通过验证后才能访问登录后的资源。

之前基于session的登录方式，是在用户登录成功后将用户信息存入到session中，这样不利于程序的横向扩展， 并且随着登录用户的增加，会不断的占用服务端的内存资源；而现在这种基于token的登录方式，是在登录成功 后将用户信息存入到客户端的中，不会额外占用服务端的内存资源，并且通过签名和验签可以保证数据不被篡改。

具体在做的时候需要解决几个问题。

首先是ajax跨域请求的问题，因为采用了前后端分离的开发方式，所以前台页面和后台接口在两个不同的项目中，因为端口号和ip地址不同所以就涉及到了跨域问题，咱们都知道ajax是不允许跨域的，解决这个跨域问题可以通过jsonp,也可以通过cors,因为jsonp发送的请求只能是get请求，所以我们项目中当时采用了cors这种解决方案，这样就可以发送各种请求方式的请求，像get,post,delete,put这些。

具体在项目中做的时候，要么可以通过spring给咱们提供的一个叫 @CrossOrigin的注解来实现，要么可以自己写一个CorsFilter来实现，当时的项目中因为涉及到所有的请求都会先经过springMVC的拦截器进行验证，所以我就将跨域处理的相关代码写到了拦截器里面，这样也就不需要在每个Controller中添加 @CrossOrigin这个注解了。

具体在做的时候是通过调用response的setHeader方法添加几个头信息来允许ajax的跨域请求，具体记得不是太清楚了， 好像叫allow\_methods,allow\_headers还有allow\_origin;其中allow\_methods指明能够接受的跨域的请求类型，比如 put,get,post,delete,options这些，allow\_headers指的是允许接受的自定义的头信息，allow\_origin表示允许 跨域的远程ip地址和端口号。

再者ajax跨域的时候要传递头信息，我们为了让每个ajax发送请求时候都能自动加上这个自定义头信息进行传递，我就写一个公共的js文件，并且在这个js文件中通过了$.ajaxSetup来定义全局的ajax设置，并且通过beforeSend 函数来在所有ajax发送请求前，通过xhr.setRequestHeader传递自定义的头。这样在每个前台页面中引入这个公共的js文件就可以了。

再者就是后台登录接口的处理逻辑，先进行各种非空判断以及用户名和密码的验证，如果都成功则将会员id, 会员名，以及一个uuid组成json格式的字符串，并通过base64进行编码；之后将base64编码后的用户信息结合服务端定义的秘钥用md5生成签名，目的是为了保证数据不被篡改，因为秘钥在服务端；之后再将会员名和uuid组合起来作为key,设置指定的过期时间，将其存入到redis中，这样做是为了后期在拦截器中进行过期判断。最后将前面用户信息的base64编码的内容和签名信息的base64编码内容通过.作为分隔符连接起来，响应给 前台，前台就可以在登录的回调函数中将这个信息暂时存入cookie中，方便后期取出cookie中的数据放到ajax的请求头中。

最后最重要的就是后端接口中拦截器的处理逻辑。

1.跨域处理

2.options特殊请求的处理

3.对于不需要拦截的请求直接放行

4.判断头信息是否完整

5.验签

6.判断是否过期

7.将会员信息存入request作用域，方便在其他控制层获取会员信息

跨域处理之前已经说过了。

关于options特殊请求是这个样子的。之所以会发送options请求，是因为我们前台客户端在发送ajax请求的时候传递了自定义的头信息，这就会导致发送真实的请求前先发送一个options请求，但这个options的请求不会传递 任何实质的数据，所以在后台拦截器处理的时候我们可以通过request.getMethod来判断下请求方式是否是options,如果是就不放行，也就是return false,处理完后前台就会自动发送真实的请求过来。

在项目中有些请求是不需要登录就能访问的，有些请求是登录后才能访问的，为了对处理这类请求，我写了一个自定义注解，然后把注解添加到登录后才能访问的方法上，这样在拦截器中，我就可以通过反射获取方法上是否有这个自定义注解，如果没有证明不需要进行拦截验证，就可以直接return true;放行。

接下来我会判断前端传过来的请求中是否包含自定的头信息，如果没有则抛出异常，告诉前端头信息不完整，因为在项目中我还通过springMVC的 @ControllerAdvice 注解定义了一个全局的异常统一处理器，所以当抛出异常后会被这个全局异常处理器拦截到，将其转换为json格式的数据包含状态码和错误消息传递给前端，这样前端就可以根据这个进行判断，进行相关的处理和信息提示。

如果有头信息就到了最关键的验签过程，目的 就是防止内容被篡改，具体的逻辑是这样处理的，首先将传过来的头信息根据.进行切割，切割后就取到了经过base64编码的用户信息和经过base64编码的签名，之后用经过base64编码的用户信息和服务端的秘钥经过md5得到签名，再对base64编码的签名进行解码，然后两个签名进行对比如果一致，则证明数据没被篡改，否则抛出异常提示数据被篡改。

验签后就要判断存储在redis中的key是否过期，这个比较简单，就是将base64编码的用户信息进行解码得到 json格式的字符串，并通过阿里巴巴提供的fastjson这个第三方jar包，将json格式的字符串转换为对应的 会员对象，这样就能从中获取到会员的相关信息包含会员名，会员id，以及uuid这个标识，紧接着根据 会员名和uuid组合key，调用redis工具类判断该key是否还存在，如果不存在则抛出异常提示数据过期了。

如果存在就会给对应的key进行续命，其实也特别简单就是调用redis工具类中的expire方法就行了。这个续命说白了有点类似于session的过期时间处理，比如session的过期时间是30分钟，如果到最后一分钟了 又有客户端请求发送过来则会重置为30分钟。

经过这些层层判断后就可以将会员信息存入到request作用域中，这样做是为了方便在其他控制层的方法中获取会员的相关数据，最后通过return true进行 放行 就行了

**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）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，比如将性能高的那台tomcat的权重设置为4，将普通配置的那台tomcat设置为1，这样就能达到让性能高的那台tomcat处理更多的请求。

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。

要想完成分布式session的编码工作，就必须需要搞清楚单机版的session是如何工作的，这样我们就可以模拟它工作的流程和机制，并且把相关的信息存储到redis这个缓存服务器中。

单机版session的工作原理是这样的，当访问页面时如果遇到getSession()这个方法，那么就会在服务端的内存中开辟一块空间用来存储以后要放到session中的数据，并且会生成一个唯一的 sessionId 作为和这块空间关联的依据，写入到客户端浏览器的会话Cookie中。说的通俗一点可以这样理解，就相当于我们去超市购物的时候一个个的存储柜，其中sessionId就是这个存储柜上的钥匙，而存储在session中的数据就是存储到这一个个存储柜中的物品。一把钥匙只能开一个柜子们，这样就保证了不同人拿着他们唯一的那把钥匙去开各自的柜子们，取到属于自己的物品。

当知道了这些以后，我们就清楚，这里面最主要涉及到会话cookie的操作以及将数据存储到redis中并且还得让会话cookie和redis中的key产生关联，这样每次发送请求时，会将本地存储cookie的域名和当前访问的域名进行对比，如果匹配则会将cookie自动提交到服务器端，这样就可以根据cookie中存储那个自定义的sessionId在redis中找到对应的数据，

首先我封装了一个cookie的工具类，然后在里面写了常见的 写cookie,读cookie,删cookie的常见操作。这里面要注意的是，cookie被分为会话cookie和持久化cookie，他们的区别就是会话cookie的生命周期和浏览器保持一致，一旦浏览器关闭了，就会消失；而持久化cookie是存储在硬盘上，他的生命周期取决于在编程的时候设置的maxAge的值；在具体编码的时候会话cookie不需要设置maxAge属性，而持久化cookie要根据项目的要求设置maxAge；再者在设置cookie时候还需要设置 域名以及Path,通常将域名设置为当前域名，将path设置为 / ；代表网站的根目录。写cookie通过的是response的addCookie方法，而读cookie用的是request的getCookies方法，返回一个cookie数组，然后根据cookie的name进行对比找到需要的cookie信息，删除cookie比较简单，说白了就是将指定name的cookie的maxAge设置为0,然后重新通过response的addCookie方法写入到客户端浏览器中，这样就完成了对指定cookie的删除工作。

在项目中我是通过jedis来作为中间的桥梁操作redis服务器的，并且考虑到性能问题配置了jedis连接池，这样就可以在大并发访问时候提高性能；并且把对redis的常规操作也都封装成了工具类，方便后续的使用。我将项目中单机版session的地方都改为了通过redis的进行信息的存储。首先在生成验证码的地方，我首先通过CookieUtil读取指定name的cookie，如果没有读取到则通过uuid生成一个唯一的标识作为cookie的值，用指定的LoginId作为cookie的name将其通过CookieUtil的writeCookie方法写入到客户端浏览器中，这样就保证了最开始没有LoginId的时候会生成新的,如果LoginId存在则无论怎么刷新会使用现存的LoginId而不会刷新一次就创建一个新的。并且会将Cookie的值作为redis中的key,将生成的随机数验证码作为值，调用redisUtil工具类存储到redis中，并设置过期时间为5分钟，因为验证码没必要一直存在占用内存空间。

在登录方法中首先通过CookieUtil的readCookie方法根据LoginId作为key，读取指定的cookie，获取cookie中的值，将该值作为redis中key,读取对应的验证码信息，和用户输入的验证码进行对比，如果一致则验证码正确进行后面的验证，否则验证码输入错误。在各项验证都成功后将获取的用户信息通过Gson转换为json格式的字符串作为值，用“user:”+LoginId作为key，将用户信息存入到redis中，并设置过期时间为30分钟，同时删除redis中的验证码，释放空间。

最后就是在拦截器中通过CookieUtil的readCookie方法读取到会话cookie中存储的LoginId的值，用“user:”+LoginId作为redis中的key读取用户信息,如果为null则证明用户没有成功登录跳转到登录页面，否则证明用户成功登录，调用redisUtil中的expire方法给redis中的用户信息续命，重新设置为30分钟，这也是参考本地session的工作机制。这样就完成将整个项目中的本地session切换为基于自定义cookie+redis的分布式session。

就解决了在进行负载均衡配置后，如果一台tomcat宕机，用户也无需重新登录的问题。

=============（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说白了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。

之前做其他项目的时候，日志处理模块通常都是在控制层结合log4j进行日志的 控制台打印以及日志文件的存储，除此之外还会把日志信息插入到mysql数据库中存储起来，方便查看。

但这样做会有个问题，就是需要在每个Controller类的每个方法中都写上相关的日志记录代码，这样就会出现大量的代码重复，以后维护起来也特别麻烦。

所以说后面在做日志管理模块的时候，我就考虑到可以使用AOP做统一日志的处理，这样就可以让我们程序员在工作的时候把精力花在核心业务代码的处理上。

具体在做的时候我是这样写的。首先写一个日志切面类，这个切面类说白了就是一个普通的java类，后面会通过配置文件的配置让他具备切面类的功能。在这个普通的java类中我会自定义一个横切逻辑，说白了就是一个普通的方法，但这个方法中需要特别注意几点，因为当时在项目中我使用的是环绕通知，所以就会让这个方法返回一个Object类型值，再者就是在方法中有一个ProceedingJoinPoint类型的参数。

具体在写方法的时候有几个特别关键的地方，首先因为要获取当前登录的用户，用户的信息是存储在session中的， 我当时采用的解决方案是使用ThreadLocal+Filter来完成在一个普通的java类中获取当前请求的request对象，进而获取存储在session中的用户信息。ThreadLocal可以把它理解成一个Map，但它特殊的地方就是它用当前线程充当key，所以在使用的时候，存储信息用set(value)就行了，之所以没有写key，就是因为当前访问的线程就是默认的key,同理取数据用get();当时我封装了一个工具类，工具类中有个setRequest方法，就是将request对象存储到ThreadLocal中，同样还有个getRequest方法，就是获取当前线程对应的reqeust对象。

之后我会在自定义的Filter中的doFilter方法中，调用工具类的setRequest方法，将当前请求存储到ThreadLocal中，当然还得在web.xml中配置Filter使其生效。接下来我会在日志切面类中通过调用工具类的getRequest方法来获取request,进而通过getSession来获取session，这样就可以取到存在session中的相关用户信息了。

获取了用户信息之后就要记录用户做了什么事情，在这块我们当时的项目是这么规定格式的，就是要记录用户执行了哪个类的哪个方法，并且要把执行这个方法时候对应的参数信息也给获取到，比如用户添加了一个商品，那记录的信息就应该是调用了ProductController类的addProduct方法，并且也要获取到添加的商品信息参数，这样才能看的更明白。

获取类名和方法名这块是通过反射机制，调用ProceedingJoinPoint的相关方法获取的，获取参数信息是通过 request.getParameterMap()之后对其进行循环遍历，这样就获取到了提交时候的参数详情.

在这整个切面类中还有一个特别重要的方法就是ProceedingJoinPoint.proceed();它代表的就是实际要执行的 核心业务逻辑，它的返回值就是实际执行方法的返回值，比如刚才说的ProductController类中的addProduct方法， 这个proceed()代表的就是addProduct方法，而他的返回值就是控制层中addProduct方法的返回值。之所以 它的返回值类型为Object就是因为不同方法的返回值不一样，但它们都属于Object对象。

接着我会把proceed方法进行try...catch...捕获，然后在这个方法执行完后记录操作成功的日志并且把日志信息插入数据库，在catch捕获异常的时候记录错误日志，同时会把异常信息记录数据库。

最后需要在spring的配置文件中配置aop:config以及配置切点表达式来对控制层中的增，删，改方法进行拦截，这里就用到了切点表达式中特殊符号的双竖杠||。

在上交完任务后，我们经理给我说这个东西，做的整体上还不错，但是有个问题，就是日志虽然记录了操作哪个 类的哪个方法，程序员可以读懂，但业务员根本就看不明白，不够人性化，让我的日志记录再改进下。

我通过和我们团队的人讨论，最终决定通过自定义注解来完成这个改进。

在写自定义日志注解时候，通过 @Target 设置为Method指明该注解只能用在方法上面，通过将 @Retention设置为RunTime指明将注解保留至运行时。这样就可以通过反射去获取注解信息。

在注解中声明了一个String类型的value来让程序员手工设置日志的信息，之所以采用value，是因为value这个字段有特殊的含义，它可以在使用自定义注解给日志信息赋值的时候省略不写，用起来更加方便。之后就可以在Controller中的方法上加入自定义注解并且对value进行日志信息的赋值，如 @Log("增加商品")。

最后在AOP的日志切面类中通过获取方法签名得到Method，通过Method的isAnnotationPresent判断该方法上面是否加入了自定义日志注解，如果 是 则再通过Method的getAnnotation来获取自定义的日志注解，最后再通过.value()方法获取自定义注解中日志信息的值，这样在记录日志的时候就可以显示更加人性化的信息。

**（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是ROA面向资源编程的一种设计风格，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,比如：文章、图片、视频等,像这个restful在使用的时候就要注意，路径中不能出现动词，最好是名词并且要以复数的形式出现，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。

经过了解之后，我就开始了基于SpringMVC进行restful接口项目的开发。开发时用到的注解有 @RestController @PathVariable @RequestBody @ReponseBody @Value @RestControllerAdvice这些，其中 @RestController就相当于 @Controller 和 @ResponseBody的组合体，这样就可以避免在各个方法上加入 @ResponseBody注解了，@ResponseBody的作用是将java对象转换为对应的json格式，默认使用的是Jackson这个第三方的工具包，这个注解的本质说白了就是通过Jackson将java对象转换为对应的json格式的字符串并且通过response获取输出流将json格式的字符串响应给客户端，想要使@ResponseBody起作用除了要在maven的pom文件中导入jackson的依赖外还需要在spring的配置文件中加入<mvc:annotation-driven/>，@PathVariable是为了从路径中获取参数信息比如在删除操作的时候就需要从路径中获取id信息， @RequestBody起到的作用和@ResponseBody看好相反，就是将前端提交过来的json格式的数据，自动转换为对应的java对象；@Value可以从属性文件中根据key获取对应的信息，比如我们可以把调用第三方接口时需要的信息以键值对的形式写到自定义的属性文件中，然后通过context:property-placeholder加载属性文件并且通过@Value根据key找到指定的值，这样可以避免程序中出现硬编码，提高程序代码的可维护性。对于@RestControllerAdvice来说它的作用就是异常的统一处理，就相当于@ResponseBody和@ControllerAdvice的结合体,我们项目中异常的统一处理就是用它来完成的。

我们采用JSON格式的数据作为restful的响应结果，其中用code表示状态码，用message表示提示信息，用data表示返回的数据。为了方便返回指定格式的JSON数据，我们自己封装了一个ServerResponse对象，该实体对象中就包含code,message,data三项属性，这样我们在Controller层的各个方法中，统一返回的都是我们自定义的Response对象，并且我们也通过自定义的枚举对象ResponseEnum来方便进行code和message的统一赋值。

对于客户端前台调用restful接口，如果涉及到跨域则可以通过多种方案解决。首先我们可以通过ajax的jsonp来解决ajax跨域问题，如果采用这种方案则客户端在发送ajax请求时候需要将datatype设置为jsonp,在服务端需要接受callback参数，结合spring提供的MappingJacksonValue来将要响应的信息作为构造参数进行传递，并调用对应的setJsonpFunction设置为参数callback对应的值；但是这种方案只能发送get请求，如果服务端接口接受的是其他类型的请求如post,put,delete则jsonp就没办法。这个时候我们可以采用spring 给我们提供的@CrossOrigin注解来解决，使用这个注解的时候要特别注意一点它需要和jdk1.8及其以上配套使用，否则会报错。通过@CrossOrign的value属性指定客户端所在的域名这样客户端前台就可以使用普通的ajax进行跨域访问了而且可以发送任意类型的请求，这样就解决了上面所说的ajax jsonp只能发送get请求的问题。

写完接口之后我们通常会通过PostMan去测试接口的正确性，并且会自己编写接口文档，在编写接口文档时通常需要写清楚接口的业务描述，接口的请求地址，请求方式，请求参数的类型，是否含有头信息，请求参数的含义以及接口的响应格式【json/xml】，响应内容，响应内容的各个字段含义。

当时我是将商品信息，文章信息，分类信息，品牌信息,评论信息发布出去。

**（接口安全与接口限流）**

关于接口的安全方面，我使用的是基于签名验证的方式来保证数据传输过程中的安全性。

为了防止重放攻击，也就是攻击者探测到该url后，再次发送一个一模一样的请求过来获取数据，我采用了nonce+time的解决方案。

nonce是为了解决指定时间内重放攻击的问题，而time是为了解决指定时间外重放攻击的问题，也就是该url只能被访问一次，之后带有同样header的url则不能被再次访问。

nonce是一个唯一的随机数，通过uuid加上时间戳，再加上10位的随机数构成，time是客户端当前发送请求的时间。通过time我可以判断客户端发送请求的时间加上指定的过期时间是否小于接口的服务端接收到该请求的时间，如果是则证明请求超时；我设置的超时时间是60s,这样攻击者在探测到该url后，如果在60s后再次发送则返回请求超时的提示；但是如果攻击者在60s内再次发送该请求就会获取数据，这个时候nonce就发挥了作用，它是一个唯一的随机数，每次发送请求时会通过header传递到服务端接口，这样第一次客户端访问时，redis缓存中还没有该nonce,则会将该nonce存入到redis缓存中并设置过期时间为60s,如果攻击者在60s内再次访问该url，因为redis中已经存在这个唯一标识则提示不能访问，这样就解决了60s内进行重放攻击的问题。60s后该nonce就会在redis中自动消失，这样攻击者再次访问就会因为超过60s的指定时间而提示超时。

具体做的流程是这样，我们后台有个APP管理模块，当有客户需要访问我们接口的时候，就会通过app管理模块中的功能，生成appkey和appSecret,这个appkey和appsecret说白了就是一个UUID的字符串。然后将其分配给客户端,并且将这个信息存入到数据库表中。还会存入到redis的缓存中，这样可以提高查找的性能。

客户端在调用接口时需要传入appkey,nonce,time,sign这四个header;其中appkey是服务端接口分配给他的，nonce是一个uuid加上6位随机数构成的一个唯一标识，time是客户端访问时的时间戳，sign签名是将appSecret结合nonce以及time后通过SHA1算法得出来的签名信息。

在服务端接口中我会写一个切面类来进行安全验证，切面类中的具体逻辑是通过request获取appkey,nonce,time,sign这四个请求头，这几个头中有任何一个获取不到就认为缺少头信息，就会以json的方式返回请求头丢失的错误提示；之后我会通过上面说的nonce+time的方式来防止重放攻击;紧接着我会根据传递过来的appKey先从redis缓存中查询对应的appSecret如果没有再从数据库中查询；如果还是查找不到则证明appKey无效，返回appKey无效的错误提示；如果能找到则根据查询出来的appSecret和nonce以及time结合起来，同样通过SHA1算法得出签名和客户端传递过来的签名进行对比，如果成功则有权限访问服务端接口否则返回签名错误的提示拒绝访问。

在这整个过程中可以通过ThreadLocal结合Filter在切面类中获取request，并且通过在spring的配置文件中配置相关的切面，切点表达式以及环绕通知来使其切面类生效。

我们为了防止合法的客户端在调用接口时频繁的刷新接口，造成服务端接口不能正常工作甚至宕机，所以根据经理的要求进行了接口限流。又因为要对不同的远程接口指定不同的限流策略所以就通过自定义注解的方式来为不同的接口设置限流。

具体的流程是这样做的，首先我定义了一个自定义注解叫@AccessLimit；这个注解中有两个方法一个叫seconds,一个叫maxCount分别代表多长时间内接口允许被访问的最大次数。之后在需要进行限流的接口上就通过@AccessLimit设置seconds和maxCount进行限流，比如设置seconds为60，maxCount为5，则证明60秒内最多只能访问5次，如果超过指定的最大次数则提示接口限流。接着在上面接口安全的切面类中，加入了新的判断逻辑。具体来说就是当通过层层关卡的验证，证明客户合法的情况下，会获取方法签名，之后获取方法签名对应的方法，这样就可以判断该方法上是否有AccessLimit这个自定义的注解，如果有则获取该注解，并获取该注解的seconds和maxCount的值，然后将appKey和请求的方式以及当前访问的远程接口的url组合起来作为redis中的key,通过调用incr这个递增方法进行访问次数的累加，当incr的返回值为1时证明原来该key不存在，第一次设置该key，这时就将该key的过期时间设置为seconds,当incr的返回值大于maxCount中指定的值,则证明在指定时间内访问次数超过了指定的最大值，那么就提示接口限流

**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。

**JAVA 反射**

java反射机制 说白了 就是在运行过程中，对于类来说，咱们可以获取它的属性和方法，对于对象来说，可以动态调用它的任意方法和属性。 动态获取信息 以及 动态调用对象的方法 就是反射机制！！！

我在项目中做导出excel功能的时候就是用的反射机制，这样就能达到导出excel的通用性，只用这么一个工具类，就可以导出不同的业务数据，比如商品，会员，品牌，分类，而且用起来也特别方便。

想要用反射首先得获取类，获取类的方式我知道的有这几种，可以通过对象的getClass()方法获取类，可以通过Class.forName("类的全称")来获取类，可以通过 类名.class 获取类。

之后可以通过 getDeclaredFields()获取类的所有属性，getDeclaredField(属性名)获取类中指定 名字的 属性,getDeclaredMethods()获取类的所有方法，获取属性后还可以通过getType()获取属性的类型，通过getName()获取属性名。

动态调用方法这块如果是get方法可以使用 属性.get(对象) 来调用get方法，如果是set方法可以通过 属性.set(对象，值)来调用，但因为get,set方法一般都是私有的，所以得先通过 属性.setAccessible(true) 来开启访问权限。如果是普通的方法可以通过 方法.invoke(对象，参数) 来调用。

=========（基础概念）==============

**SpringMvc的底层原理**  
整个处理过程从一个HTTP请求开始：  
1.Tomcat在启动时加载解析web.xml,找到spring mvc的前端总控制器DispatcherServlet,并且通过DispatcherServlet来加载相关的配置文件信息。  
2.DispatcherServlet接收到客户端请求，找到对应HandlerMapping，根据映射规则，找到对应的处理器（Handler）。  
3.调用相应处理器中的处理方法，处理该请求后，会返回一个ModelAndView。  
4.DispatcherServlet根据得到的ModelAndView中的视图对象，找到一个合适的ViewResolver（视图解析器），根据视图解析器的配置，DispatcherServlet将要显示的数据传给对应的视图，最后显示给用户

**Spring IOC**

我的理解就是说原来咱们想要创建一个类的对象，得自己通过new的方式进行，  
而现在可以通过在spring的配置文件中写上一段<bean id="" class=""/>的配置，  
从而让创建对象实例的过程交给spring容器来完成，而我们在程序中可以直接获取  
创建好的对象实例进行使用。这就是IOC的控制反转。  
  
AOP的切面在项目中用的还是比较多的，咱们都知道spring中的事务管理就是通过AOP切面的方式来实现的。切面的好处就是把分散在代码中的重复代码提取出来进行统一的维护和控制。我们在项目中就使用aop完成日志的统一处理。原来是把日志记录的相关代码分散到各个控制层的相关方法中，但这就会导致程序员在开发时候不能将精力集中到业务逻辑的处理上，还得考虑记录日志，工作效率就大打折扣，我在项目中通过....[将aop中日志的统一处理融进去]

**HashMap的底层原理**

HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。

Entry就是数组中的元素，每个 Entry 其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，再根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

默认是构建一个初始容量为 16，负载因子为 0.75 的 HashMap。

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，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。Jdk1.8中引入了红黑树，目的是避免单条链表过长而影响查询效率。

**Hashtable与HashMap的区别**

1.Map是一个以键值对存储的接口。Map下有两个具体的实现，分别是HashMap和HashTable.2.HashMap是线程非安全的，HashTable是线程安全的，所以HashMap的效率高于HashTable.3.HashMap允许键或值为空，而HashTable不允许键或值为空4.HashTable之所以是线程安全的是因为在它的方法上加上了synchronized锁。5.LinkedHashMap是有序的

**Set,List,Collection,Collections的区别？**

List和Set都是接口，他们都继承于接口Collection,List是一个有序的可重复的集合，而Set的无序的不可重复的集合。 Collection是集合的顶层接口，Collections是一个封装了众多关于集合操作的静态方法的工具类,因为构造方法是私有的，所以不能实例化。

List接口实现类有ArrayList,LinkedList,Vector。ArrayList和Vector是基于数组实现的,所以查询的时候速度快，而在进行增加和删除的时候速度较慢LinkedList是基于链式存储结构，所以在进行查询的时候速度较慢但在进行增加和删除的时候速度较快。又因为Vector是线程安全的，所以他和ArrayList相比而言，查询效率要低。

**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。

**Session和cookie的区别**

session是存储在服务器端，cookie是存储在客户端的，所以安全来讲session的安全性要比cookie高，然后我们获取session里的信息是通过存放在会话cookie里的sessionid获取的。又由于session是存放在服务器的内存中，所以session里的东西不断增加会造成服务器的负担，所以会把很重要的信息存储在session中，而把一些次要东西存储在客户端的cookie里，然后cookie确切的说分为两大类分为会话cookie和持久化cookie，会话cookie确切的说是存放在客户端浏览器的内存中，所以说他的生命周期和浏览器是一致的，浏览器关了会话cookie也就消失了，然而持久化cookie是存放在客户端硬盘中，而持久化cookie的生命周期就是我们在设置cookie时候设置的那个保存时间，然后我们考虑一问题当浏览器关闭时session会不会丢失，从上面叙述分析session的信息是通过sessionid获取的，而sessionid是存放在会话cookie当中的，当浏览器关闭的时候会话cookie消失所以我们的sessionid也就消失了，但是session的信息还存在服务器端，这时我们只是查不到所谓的session但它并不是不存在。那么，session在什么情况下丢失，就是在服务器关闭的时候，或者是sessio过期，再或者调用了invalidate()的或者是我们想要session中的某一条数据消失调用session.removeAttribute()方法，然后session在什么时候被创建呢，确切的说是通过调用getsession()来创建，这就是session与cookie的区别

JDK1.8新特性

在项目中/在工作中 我用过jdk1.8的lambda表达式还有stream

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，$方式无法防止Sql注入，所以一般能用#的就别用$。

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。

缓存

**缓存【redis】穿透**：

缓存穿透是指查询一个一定不存在的数据，比如商品最多只有1000个，我查询1001,1002这些一定不存在的id,这个时候因为查询的信息在缓存中不存在，大量的并发请求就会直接到数据库上，而数据库中也不存在这样的数据，所以缓存就形同虚设，这个id一定不存在的请求以后都会到数据库上，增加数据库的压力。

为了解决这个问题，可以缓存空值,也就是说即便在数据库中 查询的数据不存在，也会在缓存中缓存个空值，这样下一次就可以走缓存而不是数据库了，但太多的空值也会占用内存空间，导致内存的浪费，可以通过设置过期时间比如3-5分钟，这样当到期后，空值对应的key就会消失；释放空间；但导致的另外一个问题就是如果在这3-5分钟内后台增加数据，这个时候就有id为1001的数据了，而缓存中还是空，所以就会导致数据库和缓存中数据的不一致，为了解决这个问题可以在后台增加数据的同时主动更新缓存中的数据。

**缓存【redis】雪崩：**

如果缓存在一段时间内同时失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

解决这个问题可以

1.为不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

2.做双缓存策略。

A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，

A1缓存失效时间设置为短期，A2设置为长期。

**redis持久化：**

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，实际操作过程是fork[分]一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用 二进制 压缩存储。具体是在redis.conf中配置的，找到save关键字， 指定每隔多长时间有多少个key发生变化，就将其持久化。

save 900 1

#在900秒(15分钟)之后，如果至少有1个key发生变化，就会持久化。

save 300 10

#在300秒(5分钟)之后，如果至少有10个key发生变化，就会持久化。

save 60 10000rdbcompression yes

rdbchecksum yes

dbfilename dump.rdb

save 900 1save 300 10save 60 10000

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。

aof的 实时性、可靠性 要高于rdb：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。

============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？  
  
这个当时是其他人配置的，我在项目中看到过，自己具体还没配过，不过，应该也不难啥，  
要是咱们公司需要，我也可以搭建。  
  
你们的链路追踪用的是啥？  
我知道的有Spring Cloud Sleuth和Zipkin，但项目中我们还没用到。

它主要用于 聚集 来自各个 异构系统 的 实时监控数据，用来追踪 微服务架构 下的 系统延时问题。

============以下了解============

## Docker[容器化部署]:

基于Docker的容器化部署，整体来说就是让配置整套环境更加方便， 可以达到一键部署的效果。

说白了不用Docker我们也能搭建环境，部署项目，但是比较费劲，步骤比较繁琐，而且如果要把项目迁移到另外一台机器上，还将相关的软件重头到尾安装一遍，而且有时候机器的操作系统版本不一样，即便按照同样的配置步骤也有可能出现软件的不兼容，而且在改配置文件的时候可能会因为手误导致配置出错，多个逗号，少个分号啥的，这些都比较耗费时间，也降低了工作效率。

通过docker就可以解决刚刚说的这些个问题。

==============================================

我觉的docker中最重要的就是镜像和容器这两个东西。镜像可以理解成

以前的光盘，这个光盘上包含了我们所有需要的软件及其相关环境；通过docker run 启动镜像后就有了容器。在执行docker run时候可以通过 -d 设置为以后台程序的方式运行，通过-p指定容器端口号到宿主 主机端口号的映射，在最后跟上要启动的镜像的名字。

之后可以通过docer ps 查看启动的容器对应的进程信息，通过docker stop 容器id 来停止容器，通过docker start 容器id来启动已经停止的容器。

在项目中还可以通过 docker-Compose 进行服务编排，说的简单点一个项目想要跑起来需要数据库，需要redis,需要nginx等等，我们可以通过这种服务编排将整个项目所需要的所有软件对应的镜像及其相关信息都写到一个YAML格式的文件中，这样在部署项目的时候直接通过docker-compose up启动服务，通过docker-compose ps查看启动的进程信息，通过docker-compose down来停止服务。就可以达到一键部署的效果。

jenkins:

它是做持续化集成用的，说白了以前 部署项目的流程是 获取源代码，编译打成war包，上传到linux服务器中的tomcat下，再重新启动tomcat。整个过程都需要人为的一步步操作，特别麻烦，特别是当项目多了以后，部署起来更是麻烦。现在使用jenkins后，整个打包，编译，部署，启动 的过程就完全自动化，只需要登录jenkins的后台管理，配置git,目的是为了从指定的git服务器上下载源代码；配置maven,目的是为了对源代码进行编译，打包；配置tomcat，目的为了指明项目部署的位置，还要编写一个简单的shell脚本，目的是为了杀死对应的tomcat进程，再重新启动。这样就配置了一个完全自动化的任务，后续 再部署项目 只需要运行该任务，就可以自动完成项目部署的工作。====================================================================

## Redis-cluster

Redis Cluster:  
为了避免redis服务器产生单点故障，也为了提高redis服务器的存储数据的容量；我们将redis配置为集群模式，为了提高集群的高可用性，我们配置了主从，使用3台redis作为master【主】;  
并且为每台master都配置了对应的slave【从】；这样就可以达到数据扩容同时提高集群的可用性。  
  
整个集群是由不同的node【节点】组成的，而每个node【节点】上面被分配了slot也就是槽，数据是被放到了对应的slot【槽】中。  
  
redis集群中共有16384个槽，集群会把这16384个槽平均分配到不同的节点上；当向集群中存数据的时候，会根据key采用CRC16算法，算出来一个值，让这个值和16384求余数，这样就把数据分片的存储到不同的槽中，也就落到了不同的节点上，实现了数据的分片，同时也起到了扩容的目的。  
  
如果没有配置对应的slave服务器，则一台master宕机就会导致整个集群不可用，这就大大削弱了集群的功能，所以我们可以为每台master配置一个slave，这样当master宕机时，对应的slave就会充当master，使集群继续提供服务。如果宕掉的那台master又被重启了，则该master会变为slave。  
  
在我们构建的redis集群中，有3个主，3个从，当存活的 主 节点的数量小于总节点的一半时，集群不可用。即当存活的主节点数量为2时，集群不可用。