# B简历：

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

### 基本信息

|  |  |  |  |
| --- | --- | --- | --- |
| 基本情况 | | | |
| **姓名** | **王帅鹏** | 性别 | 男 |
| **出生日期** | 1997-04-05 | 属相 | 牛 |
| **现住地** | 丰台区 东营里 2号院 |  |  |
| 大学情况 | | | |
| **学校** | 郑州科技学院 | 专业 | 计算机应用技术 |
| **入学时间** | 2013-9 | 毕业时间 | 2017-7 |
| **专业课** | **C语言、数据结构课程设计、数据库原理课程设 计、操作系统实训、Java程序设计** | | |
| **学历** | **本科** | 学位 | **学士学位** |
| **学校地址** | 河南省郑州市荥阳市学院路1号 | 校长 | **徐建新** |
| **去学校的方式：** | **郑州汽车站→地铁1号线→秦岭路坐302路→到郑州科技学院** | | |
| **上小学日期：** | **2001年9月 5岁六年** | | |
| **上初中日期：** | **2007年9月 11岁三年** | | |
| **上高中日期：** | **2010年9月 14岁三年** | | |
| **上大学日期：** | **2013年9月 17岁四年** | | |
| **毕业时间：** | **2017年 7月 21岁** | | |
| **在校学生：** | **5000左右** | | |
| **有教师：** | **不太清楚** | | |

|  |  |
| --- | --- |
| 第一家公司 ： | 上海泽辛信息技术有限公司 |
| **地址：** | 上海宝山区刘行(地铁站)1018弄1号楼207室 |
| **入职时间：** | **2017年8月初** |
| **离职时间：** | **2019年1月** |
| **公司性质：** | **民营** |
| **职位：** | **Java工程师** |
| **项目经理名字：** | **张惠涛** |
| **薪资方面：** | **刚刚毕业转正以后工资是6K左右，过完年大概三月份涨到7K左右** |
| **路线：** | 坐地铁7号线 到刘行站下 （富晨花苑） |

|  |  |
| --- | --- |
| **第二家公司情况：** | 上海华沁智能交通科技有限公司(只说公司简称) |
| **公司地址**： | 上海 杨浦区 紫荆广场(江浦路) 西座10楼 |
| **公司规模：** | **80人左右，我们项目组8人，一个项目经理，4名开发人员，1名前端，1个测试，1个运维** |
| **公司性质：** | **民营** |
| **公司行业：** | **互联网和虚拟现实的软件产品的开发。** |
| **去公司路线：** | 地铁八号线 到江浦路站下 （四光村住） |
| **入职时间：** | **2019年2月** |
| **离职时间：** | **2021年4月初** |
|  | **Java工程师** |
| **项目经理名字：** | **于扬** |
| **税前：** | **15K** |
| **税后：** | **14K** |
| **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：** | **16K** |
| **薪资的构成:** | **岗位工资(5K)+绩效工资(10K)** |

|  |
| --- |
| 四年涨薪： **2017年工作的时候是5.5K，税后大概5.4K左右，**  **转正6K左右**  **2018上半年涨了1K左右 ，总7K左右。**  **2018年2月跳槽涨了3K左右 总10K**  **2018年12月涨1K总 11K左右**  **2019年9月工作一年涨了2K左右**  **2020年7月工作一年涨了2K左右**  **2021年4月初离职时工资为15K** |

**面试常见问题：**

|  |
| --- |
| **你为什么从上家公司离职？** |
| **想换个工作环境，不断的提高充实自己** |
| **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。** |
| |  | | --- | | **三个词形容自己**： 交流表达能力强，团队协作能力强，抗压自学能力强 (**可以综合自己自学的知识表达**) | | **你的优缺点是啥？** |   **优点:**  **抗压能力强（有一次项目就要上线了，连着加班了一周，每天都到夜里很晚，但我最后也是成功完成任务。），**  **乐于分享，**  **团队协作能力强**  **缺点: 我这个人说话比较直，你像在和团队成员探讨问题的时候，容易得罪人，**  **不过现在已经改进的差不多了** |
| **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 16K 上家工资15K 税后14K左右 扣了1K左右的税 |
| 你的五年规划是啥？ |
| **前2,3年 继续加强自己的技术功底，**  **然后朝着项目经理(技术经理,产品经理)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： 13薪  **车补**  220左右  **项目奖金**  8k-10k，做完项目，收完尾款的时候发。  **年终奖**  就是1个月的工资，  最后一个月和工资一块儿发/年后发  **电话补助**  260  **饭/餐补**  **15、**30/天  **电脑补**  310 |

**公司内其他部门：**

**技术部，人事部，销售部，财务部**

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

**其他问题：**

**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:**

-->日志管理-->负载均衡加redis分布式session-->基于Token的用户登录【jwt】--> 基于redis的购物车-->订单

-->项目的重构【从ssmp框架升级为springboot+springcloud组成的微服务】

-->sql优化-->通过线程池提高程序的性能

**核心概念：**

**咱们也知道这几年这个微服务springboot和springcloud用的非常多，而我在上一个项目中使用的也是这个springboot+springcloud进行的开发，那我就先说一下我对这个微服务框架的理解吧：**

服务启动向Eureka注册中心注册服务，注册中心将注册信息同步给其他注册中心，当服务消费者需要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将获取的地址缓存在本地，下次调用的则直接从缓存中取，当服务注册中心检测到服务提供者宕机、网络不通等服务不可用时，则在注册中心将服务置为down状态，并向订阅者发布当前服务提供者状态，订阅过的服务消费者更新本地缓存。

服务提供者启动后周期性（30秒）向服务注册中心发送心跳，以证明当前服务可用，服务中心在一定时间（默认90秒）未收到心跳，则认为该服务宕机，注销该实例

#### Eureka自我保护机制:

默认情况下，如果Eureka在一定时间内没有接受到某个微服务实例的心跳，Eureka注册中心将会将该实例注销。但是当网络分区故障发生时，微服务与Eureka Server之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护模式”来解决这个问题——当Eureka Server节点在**短时间内丢失过多客户端时**（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入该模式，Eureka Server就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该Eureka Server节点会自动退出自我保护模式。

在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。当它收到的心跳数重新恢复到阈值以上时，该Eureka Server节点就会自动退出自我保护模式。它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话讲解：**好死不如赖活着**。

**在这个微服务的搭建过程中也有可能发生微服务雪崩的问题**

在讲Hystrix之前先说一下服务雪崩，在springcloud**可以用Feign+Ribbon来进行服务之间的调用**。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用。如客户端访问A服务，而A服务需要调用B服务，B服务需要调用C服务，由于网络原因或者自身的原因，如果B服务或者C服务不能及时响应，A服务将处于阻塞状态，直到B服务C服务响应。此时若有大量的请求涌入，容器的线程资源会被消耗完毕，导致服务瘫痪。**服务与服务之间的依赖性，故障会传播，造成连锁反应，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应**

### 服务雪崩的解决方案

**l 熔断模式**：这种模式主要是参考电路熔断，如果一条线路电压过高，保险丝会熔断，防止火灾。放到我们的系统中，如果某个目标服务调用慢或者有大量超时，此时，熔断该服务的调用，对于后续调用请求，不在继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

### 服务熔断

当下游的服务因为某种原因突然变得不可用或响应过慢，上游服务为了保证自己整体服务的可用性，不再继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

### 服务降级

l 当下游的服务因为某种原因响应过慢，下游服务主动停掉一些不太重要的业务，释放出服务器资源，增加响应速度！(**弃车保帅，壮士断腕，刮骨疗伤**)

当下游的服务因为某种原因不可用，上游主动调用本地的一些降级逻辑，避免卡顿，迅速返回给用户

### 服务熔断和服务降级的区别和联系

服务降级有很多种降级方式！如开关降级、限流降级、熔断降级!**服务熔断属于降级方式的一种**！**因为从实现上来说，熔断和降级必定是一起出现**。因为当发生下游服务不可用的情况，这个时候为了对最终用户负责，就需要进入上游的降级逻辑了。因此，**将熔断降级视为降级方式的一种**，也是可以说的通的！

### **Hystrix能干啥**

**l 服务熔断**

当Hystrix Command请求后端服务失败数量超过一定比例(默认50%), 断路器会切换到开路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态一段时间后(默认5秒), 自动切换到半开路状态(HALF-OPEN).

**l 服务降级**

Fallback相当于是降级操作, 我们可以实现一个fallback方法, 当请求后端服务出现异常或者服务熔断后的时候, 直接调用本地fallback方法，告知后面的请求服务不可用了，不要再来了**。**

**接下来我来说下这个Feign的使用吧**

我们当时在项目中有很多的模块需要互相调用，比如说这个购物车需要调用商品进行 查询，我们当时为了解决这个问题使用的是Feign来进行微服务之间的调用

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。

**--》Zuul**

Zuul是一个微服务网关，**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。

### 为什么需要外服务网关呢

不同的微服务一般会有不同的网络地址，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。

而且可能会出现很多问题。

客户端会请求多个不同的服务，需要维护不同的请求地址，增加开发难度

后端每个微服务可能是由不同语言编写的、采用了不同的协议，比如HTTP、Dubbo、GRPC等，但是你不可能要求客户端去适配这么多种协议，这是一项非常有挑战的工作，项目会变的非常复杂且很难维护。

路由转发和**过滤器**

#### **路由转发**：接收一切外界(客户端)请求，转发到后端的微服务上去；

#### **过滤器**：在服务网关中可以完成一系列的横切功能，例如权限校验、限流以及监控等，这些都可以通过过滤器完成（其实路由转发也是通过过滤器实现的）。

过滤器的四中类型

**l PRE**：这种过滤器在请求被路由之前调用。我们可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。

**l ROUTING**：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用Apache HttpClient或Netfilx Ribbon请求微服务。

**l POST**：这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的HTTPHeader、收集统计信息和指标、将响应从微服务发送给客户端等。

**ERROR**：在其他阶段发生错误时执行该过滤器。

我们可以通过将Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即**以后通过客户端(浏览器，APP)访问微服务都是通过Zuul跳转后获得**。

**至于这个springboot的运行原理的话 我也做过一些了解**

在我看来他的运行原理就是最核心的两行代码

@springbootapplication 和 springappliction.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。

**功能模块这部分的话当时**

我负责的功能模块的有这个类似jwt的基于token的登录，购物车，订单，以及支付这些模块

那我就先来介绍下这个

基于Token的用户登录吧

刚开始是用session来进行登录认证的每次登录成功之后就把信息放入session 中 后来接触了分布式框架发现session不能来回是用 我就把session放入到redis中 完成了分布式框架的登录问题 后来发现这样比较麻烦 就开始接触了tonken

刚开始我们的项目里用的session，结果发现session因为是作用在，一个服务器的会话中，在分布式框架中不能获取到数据，然后就用了redis来解决这个问题，把数据存放到redis中，再从redis中取出数据，这样就解决了跨域问题，最后接触到了JWT技术，然后就把redis替换了。

用户在前端输入账号和密码，请求路径传到后端，会遇到一个跨域问题（跨域就是协议，端口，ip，三者之一 有一个不同就构成了跨域），在后端，我们配置了一个跨域拦截器，继承的ZuulFilter来解决跨域问题，然后我们是根据用户的账号查询数据库里是否有这个数据，没有的话说明他没有注册，然后重定向到注册页面让用户注册，反之，将传过来的密码进行MD5加密，用MD5加密密码是为了防止数据库密码泄露，然后判断密码是否正确，如果不正确返回一个状态码给前端，反之，将账号和密码放到map中，用JWT将map生成token令牌，再设置一个失效时间，然后将生成的token返回到前端。

前端我们用的是vue，在登录成功之后，我们把后端传过来的token存到sessionStorage中，然后我们在vue的main.js中设置了一个axios的全局设置，为了让每次的请求都添加上一个token参数，在axios的全局设置中判断本地的sessionStorage中是否有token，有的话就在请求上添加一个token参数，反之发请求的话会报没有token的错误。然后再配置一个axios全局设置的请求响应，判断返回回来的状态码，如果相同的话就返回到登录页面，让他登录。

session和token的区别就是session是存在一个服务器会话中，不能用在分布式框架中，相反token可以。

**接下来我给您介绍下我做的这个购物车模块吧**

考虑到用户操作购物车的性能问题，我当时做的是基于redis的购物车，将购物车的所有数据都存储到redis中。

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这块我用了用户的账号+特殊字串，在field这块因为每个用户都有一个购物车，为了保证唯一性，我用了member:商品id来表示，在value这块我是把商品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。

我们当时的购物车是用户登录后才能操作的，比如添加商品，删除商品等。关于登录这块，就是我刚才说的这个基于token的用户登录

用起来更加的灵活方便，我们在zuul过滤器中的进行token进行验证，通过**request**

获取用户的信息通过currentContext.getRequestQueryParams 把用户信息数据传递给后端微服务中使用 ，而后端微服务中就可以通过request.*getParameter*来获取值，来判断用户是否失效，如果失效了就返回一个状态码 给前台让用户重新登录。如果存在就继续下一步操作

那么具体在做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。

如果用户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，则将该商品加入用户的购物车，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，把商品的个数传过来 修改一下库存。

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。

购物车这个模块我是这样做的。

**至于订单**

**新增订单的时候我还用了redis的getset方法解决了接口的幂等性问题**

**在用户提交订单的时候把用户的名字作为key 去Redis中查询 判断key是否存在 存在的话就不走下去了并且从新设置失效时间 如果不存在就设置一个失效时间放入redis中。为了防止用户频繁发请求， 在新增的时候需要吧redis 中的数据取出来 把要结算的数量 取出来 用数据库的锁解决了超卖的问题每次修改返回一个值 判断这个值是不是1如果是1就可以去支付页面了 如果是0 就说明库存不住了。**

最后就是这个支付模块了

在进行支付的时候，我们是这样处理的，整体上来说调用微信的统一下单接口，根据返回的code\_url,生成二维码，扫描该二维码，进行支付后，通过调用查询订单状态的接口，来判断订单是否支付成功，如果成功则跳转到支付成功页面。

默认情况下code\_url的有效时间是2个小时，也就是说如果2个小时内没有进行扫码支付，过期后扫码将不能再次发起支付。我们可以通过传递接口文档中指定的参数，将这个有效时间设置为了30分钟。

在调用统一下单接口的时候需要传递一些必要的参数信息，比如

应用ID，微信平台提供的

商户号，微信平台提供的

商户订单号，我们是通过时间戳加上雪花算法生成唯一的订单标识

总金额，单位是分，这点要特别注意，我们数据库中存储的单位是元，所以得进行转换

生成签名时候的秘钥，为了避免攻击者生成假的签名，因为key这个秘钥在服务端存储

是一个随机数，为了保证签名的不可预测，通过nonce\_str来保证签名只能用一次，防止重放攻击

notify\_url：回调的url地址

trade\_type： 交易类型，我用的是Native，也就是生成二维码进行扫码支付的方式

body：商品描述

sign：签名，是为了防止数据被篡改。

生成签名的规则就是将要传递的数据按照字典排序，使用URL键值对的格式（即key1=value1&key2=value2…） 拼接成字符串，将该字符串拼接上秘钥key,并对拼接后的数据进行md5，得到的值转换为大写，这样就生成了签名。

调用统一下单接口后，获取return\_code和result\_code，只有两者都正常才证明调用接口成功，这样就可以从响应中获取code\_url信息，生成二维码。

为了将生成的二维码的有效期设置为30分钟，我通过time\_expire参数，设置这个参数的值为当前时间加上30分钟，最后还要将它转换为接口文档中指定的年月日时分秒的字符串格式。

后台写一个定时器 每隔一段时间查询一次支付状态 判断有没有支付，如果到规定的时间还没有支付的话 就开始根据用户的信息去查询其订单的表里边的信息 把商品的信息进行回滚

再重新加入购物车。

**这就是我当时在前台项目中负责的一些功能**

**我在之前还负责过后台管理项目中的日志管理模块以及项目的负载均衡，我们当时部署项目使用的是nginx加多个tomcat进行的负载均衡**

**这个技术，这是为了提高我们项目的一个高可用性，当然 负载均衡还可以承受更多并发量，不过在我们后端项目这块主要起到的就是一个备份的作用，保证服务的不间断。**

**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。**

**第二种使用的是一个加权轮询的策略，我们都知道在企业中不同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。**

**当时在项目部署完之后，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。**

**为了解决这个问题我们使用了ip\_hash这种负载均衡策略来代替默认的轮询策略。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。**

**那么这就是我当时部署项目的使用遇到的问题**

**至于我当时做的这个日志管理模块的话，**

**在之前的做其他的项目的时候**

我是使用AOP做的统一日志的处理，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。

项目中 aop做日志记录是如何实现的

首先要在spring 的配置文件中开启扫描的注解

首先在springmvc 开启aop的扫描 在创建一个自定注解 生成@Target(ElementType.METHOD)用在包上 方法上接口上等

@Retention(RetentionPolicy.RUNTIME)声明这是一个自定义注解 注解的生命周期 ，自定义注解关键字是@interface 然后把这个写好的注解类用在需要做日志记录的方法上

在生成一个类 通过Component来开启bean的注入交给aop 在写@Aspect表示这是一个切面类 写一个方法加上前置通知的注解@Before 切点是所有的controller

方法接一个参数 新建一个做日志记录的对象 通过getSignature()来获取所有的方法名 在通过getAnnotation（）来获取注解 判断是不是需要做日志的方法

如果是就 在通过getArgs()来获取方法的参数 放进一个空字符串中 通过setParams来吧字符串放到对象中 在通过annotation.value()来获取操作的内容最后通过

新增的方法进行保存

**最后我来说一下我们当时对项目的一些优化方案吧**

**我们当时使用的是这个索引来优化提高我们sql的查询速度，它就相当于是书的目录，我们通过目录可以快速锁定想要查看的内容，如果没有这个索引的话**

**我们想要查询一个东西的时候就需要一页一页的找**

但是这个索引也不是越多越好，因为它会占用额外的空间，并且在进行增删改的时候还需要额 外维护索引。

索引的类型得从两方面来说

一方面是特性，它分为

唯一索引和普通索引

但是从这个字段的个数来说他又分为

单一索引和复合索引【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】

这个唯一索引所在字段中的值必须是唯一的。

我们的主键会自动创建唯一索引

这个创建索引的语法的话 他是这样的

-- 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);

索引里面又有这个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。

至于这个创建索引的原则的话我们一般：

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。

但是创建完这个索引，不一定就能使用上索引；

要是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。

那么什么时候索引不起作用?

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

当然还有一些sql的优化方案

1. 比如说我们的SELECT语句中会避免使用'\*’，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。

2. 然后我们对于这个大数据量的分页就不在使用常规的 limit 开始位置，每页条数查询了；这样越靠后查询时间越长，效率越低，当时是使用的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

这样只需要和数据库交互两次，不需要多次交互数据库 大大的提高了查询的效率

3. 外键必须加索引，这样可以提升多表联查的性能

4. 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，

* 1. 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。
  2. 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。

那么这就是我当时对于sql优化方面的心得

**执行计划:**

如果我们想要查看当前sql使用了那些索引可以

通过explain查看sql语句的执行计划，通过执行计划可以查看

在这个执行计划里面

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。

**在提高程序性能方面我们使用的了线程池**

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】。

打个比方我的核心线程为5，队列是20，最大线程数是10 对应着 快递公司的五个正式员工，

20个快递柜，如果这时候来了五个快递，正式员工就去派送了，然后又来了20个快递就放到了快递柜等待派送，后来又来了第26个快递，我的快递柜和员工都被占用了，只能选择聘请一些临时员工，但是又因为经济有限，所以最多聘请5个，在后续多的快递我就不送了，

那么为什么叫临时员工呢，意思是说 过了一段时间后 我们的快递基本都配送完了，临时员工已经有很久没工作了，我就会选择把辞退，不然还需要给他发工资，很浪费资源。

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。

**基础概念:**

HashMap和ConcurrentHashMap底层原理

**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。

**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。**

## 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(可来可神),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(v 可特)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.**

**单例：**

饿汉式

public class Sing {  
  
 private volatile static Sing *sing* = new Sing();  
  
 private Sing (){}  
  
 public static Sing getSing(){  
 return *sing*;  
 }  
}

懒汉式

public class Sing2 {  
  
 private volatile static Sing2 *sing2* = null;  
  
 private Sing2 (){}  
  
 public static Sing2 getSing2(){  
 if(*sing2*==null){  
 *sing2* = new Sing2();  
 }  
 return *sing2*;  
 }  
}

双重判断锁

public class Sing3 {  
  
 private volatile static Sing3 *sing3* = null;  
  
 private Sing3 (){}  
  
 public static Sing3 getSing3(){  
 if(*sing3*==null){  
 synchronized (Sing3.class){  
 if(*sing3*==null){  
 *sing3* = new Sing3();  
 }  
 }  
 }  
 return *sing3*;  
 }  
}

**大批量数据的插入/更新?**

1.一条条插入

1. 分段 批量 提交【推荐】【减少和数据库的交互次数】
2. 批量插入和数据库交互一次

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

**springmvc的运行原理**  
 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中的ioc我是这样理解的**

·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]

jdk1.8新特性:  
 在项目中/在工作中 我用lambda表达式还有stream

**数据库持久层相关知识点**

1.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。

2.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。

1. 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 俩种取参数方式的区别：

.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，$方式无法防止Sql注入，所以一般能用#的就别用$。

**spring cloud的版本:**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？  
1.SpringCloud Config  
2.用的是携程的 Apollo 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。

冒泡排序：

for (int i = 0; i < arr.length; i++) {  
 for (int j = 0; j < arr.length-1-i; j++) {  
 if(arr[j]>arr[j+1]){  
 int temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 }  
 }  
}

**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

**linux的常用命令**

1 ifconfig；

查看ip地址

2 mkdir

创建文件夹

3 rpm -ivh

安装软件

4 rpm -qa | grep 软件名

查看是否安装

5 rpm -e 软件名

卸载软件

6 echo $PATH :查看环境变量

7 pwd

查看完整路径

8 vi/vim

编辑

ESC 命令模式

/ 输入关键字 查找

n 下一个

shift + n 上一个

9

source /etc/profile

让配置文件立即生效

10

ps -ef | grep 查看服务是否启动

11

在bin目录下通过 tail -f ../logs/catalina.out 查看启动日志

. 代表目录

根据当前目录判断是否需要加 .

tar -zxvf 解压文件

tar -zcvf 压缩文件

mv 原文件名 新文件名 ; 重命名

删除文件的时候 文件夹名称 / \* 是删除这个文件夹下的所有东西

不加 是删除这个文件夹

make 编译

mysql数据库 默认情况下 不允许远程连接【只要是通过ip地址访问 就不行 只能通

localhost访问】

如果想要远程连接 需要 授权

命令： all是权限 root是用户名

grand all on \*.\* to ‘root’@‘%’ identified by ‘密码’（123456）

flush privileges; 立即生效

默认 linux系统的防火墙是开启的 需要放过 tomcat 服务器的端口

vi /etc/sysconfig/ip

关闭tomcat

kill -9 进程号：强制杀死进程

ps - ef grep java

查看查看进程号

linux安装软件的方式

1. rpm

2. 解压缩/绿色版

3. 在线安装 yum 需要联网 推荐

在linux中安装一些软件的时候，有时候需要先配置很多相关的环境，下载一些相关的插

件，但是使用yum的话，这些事情都不用我来负责了，yum会自动安装这些软件所需要

的软件，而且这些软件版本号都是匹配的，不会出现版本冲突的或者版本过高、过低的

问题

wget 在线下载

locate 全局搜索

nginx 启动命令

/usr/local/webserver/nginx/sbin/nginx

/usr/local/webserver/nginx/sbin/nginx -s reload # 重新载入配置文件 /usr/local/webserver/nginx/sbin/nginx -s reopen # 重启 Nginx /usr/local/webserver/nginx/sbin/nginx -s stop # 停止 Nginx

nginx -t 验证配置文件是否正确

main 调用 网易云短信 发短信

tailf /opt/tomcat/logs/catalina.out

unzip apache-tomcat-7.0.67.zip 解压 zip文件

查看日志

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。

**锁：**

在这个多线程的情况下如果有多个线程操作同一个资源会引发一个线程安全的问题说白了就是执行的最终结果和预期的结果不一致，遇到这个多线程并发访问的线程安全问题，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。

通过这个双重判定锁可以在保证线程安全的情况下，提高程序的性能，通过这个同步代码块

可以减小这个锁定的范围，调高性能，而锁内部的判断是为了保证这个多线程的情况下的单例

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，然后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。如果后续又有新的线程过来，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！！!

但是这种双重判定锁本身也有一个缺陷就是可能会出现一个重排序的问题，就是说实例化一个对象其实是分为三步 1 开辟一个空间 2 初始化对象 3 将引用指向这个对象，但是由于指令重排的问题，可能他的顺序就不再是这三步了 可能会先执行 1 3 步 那么这个时候 这个对象已经不为空了就跳过了if判断 那么在后续调用这个不完整的对象的时候就会报错，为了解决这个问题，我们使用了volatile这个关键字来避免这个重排序的问题

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【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。

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。

**我们当时使用的是restful风格的接口**

**这个restful风格的接口**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,,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，当然这是标准的restful接口的定义，具体在使用的时候可以灵活变通

分布式事务：

分布式事务:跨了多个数据库事务管理,在微 服 wu 架构每个服 wu 都有自己数据库,在微服 wu 架构中必然要用到分布式事务.  
分布式事务这块，在项目中我没有具体做过，但我之前看过这方面文章，要么可以通过LCN实现，要么可以通过 最终消息一致性 实现。

这个lcn框架

主要是为微服务框架提供分布式事务的支持，在微服务框架上做了进一步的事务机制优化，在一些负载场景上LCN事务机制要比本地事务机制的性能更好，4.0以后框架开方了插件机制可以让更多的第三方框架支持进来。  
特点：  
   ①支持各种基于spring的db框架  
   ②兼容SpringCloud、Dubbo、motan  
   ③使用简单，低依赖，代码完全开源  
   ④基于切面的强一致性事务框架  
   ⑤高可用，模块可以依赖RPC模块做集群化，TxManager也可以做集群化  
   ⑥支持本地事务和分布式事务共存  
   ⑦支持事务补偿机制，增加事务补偿决策提醒  
   ⑧添加插件拓展机制  
如果问到说项目中是否用过，就说暂且还没用到或者说自己没有负责这块。  
【咱们现在的事务只能解决单个数据库的问题，如果出现了跨数据库的操作，则需要分布式事务才能解决】  
  
分库分表：  
当一张表数据越来越大，这时候就需要分表。  
按地区分  
按时间分  
地区+时间  
   当分表已经不能解决问题的时候，就需要分库了。  
业务功能分

关于这方面的技术的话我知道有这个**MyCat**

**这个Mycat可以简单概括为may**  
-  一个彻底开源的，面向企业应用开发的大数据库集群  
-  支持事务、ACID、可以替代MySQL的加强版数据库  
-  一个可以视为MySQL集群的企业级数据库，用来替代昂贵的Oracle集群  
-  一个融合内存缓存技术、NoSQL技术、HDFS大数据的新型SQL Server  
-  结合传统数据库和新型分布式数据仓库的新一代企业级数据库产品  
-  一个新颖的数据库中间件产品

没用过  
  
SSO单点登录:  
我之前看到过用CAS[开源的框架]可以做单点登录，就是有A,B两个项目，在其中任意一个项目上登录后，再访问另外一个项目就可以直接访问而不需要再次登录，这里面可以用CAS做为中央认证服务器， A项目和B项目的登录全部都跳转到CAS这个中央认证服务器上面统一进行。

在项目中自己没有负责这块  
  
工作流都用过哪些:  
我知道有Jbpm,Activiti工作流引擎。

项目中暂且还没用到

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，我们使用了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。