# 1 用户模块

## 1.1 用户模型

用户信息表：用户ID、用户手机号、用户密码（可空）、用户盐值（可空）、用户昵称、注册时间、修改时间（可空）、删除时间（可空）、删除状态（可空）、用户状态

用户登录表：登录表ID、用户ID、登录类型、用户标识、密码凭证、登录时间

手机验证Vo：手机号、验证码

用户注册Vo：昵称、密码、盐值（随机生成，隐藏）、手机号（验证页面带过来，隐藏）、注册时间（隐藏）

用户验证码登录Vo：手机号、验证码

用户密码登录Vo：手机号、密码

游客与用户（未实现）

用户分为两种状态，游客与登录。在用户打开网站时，就拦截检测是否自动登录，如果用户自动登录，则直接登录；否则，就通过拦截器拦截，将用户IP保存在cookie中。

问题：如果cookie过期了怎么办？

采用续签功能，如果用户在cookie过期时间内上线，则续签相应时间。

问题：如果用户禁用了cookie怎么办？

简单法：自己看着办，最多给你个提示说你禁用了cookie，一般不会出现这种情况，出现了也没办法。。。我选择这种方法。

复杂法：如果检测到禁用了cookie，就是用localStore进行存储，会不会又有其他问题了？

问题：session过期了怎么办？

使用Spring-session框架，同时session可以保留在数据库中，这样服务器的session可以保留时间短一点。比如说用户自动登录一个月，那cookie就保留一个月的sessionId，而服务器中的session只保留3天，如果用户访问了系统，就续签session，如果用户超过3天不访问，则session过期。用户再次访问时从数据库中根据sessionId取出相应session。

第三方登录

怎么解决同一人用不同第三方登录，能识别成是同一个人？

（1）第三方登录是单独账号

不同第三方就是不同用户，除非用户选择了绑定不同方式登录的用户。则这里会涉及到用户账号合并的情况，根据不同的平台复杂度不同。由于此方案较为复杂，所以大多数平台都不使用。

（2）整个平台都是用一个账号。

两种情况，第一种情况，平台已登录的用户，直接用用户Id和第三方openId做关联，第二种情况，平台没登录，使用授权登录的方式登录，登录后需要绑定手机号（或其他，平台使用什么作为账号唯一凭证就绑定什么），新建用户然后用户Id和第三方的openId做关联。大部分都是使用这种方案。

这种方案的好处就是可以获取昵称、头像等授权信息，还有一个好处就是如果第三方（如微信）是一直保持登录状态的话，那么可以省略登录步骤（说到底如果手机登录也不是一样有自动登录）。

但是这样其实损失了第三方登录的最大好处，就是让用户最简便的登录，由于要需要绑定手机号（或其他唯一凭证），实际上与普通注册登录并无明显区别。

实现如下：

1. 该微信在本站未注册过，很好，直接给他注册关联并登录；2，该微信已经在本站存在，当前用户未登录，直接登录成功；3，该微信未在本站注册，但当前用户已经登录并关联的是另一个微信帐号，作何处理取决于是否允许绑定多个微信帐号；4，该微信未在本站注册过，当前用户已登录，尝试进行绑定操作；5，该微信已经注册，用户又已使用该帐号登录，为何他重复绑定自己- -. 6，该微信已经在本站存在，但当前用户已经登录并关联的是另一个微信帐号，作何处理？切换用户或是报错？（画一个流程图能更好描述这个问题）

注意：整个系统中不允许用户手机号为空，也就是说必然有一个手机号进行绑定操作，否则太麻烦了。

关于用户登录的两种实现方案

一、基于JWT登录

这个方案不推荐。主要原因在于用户登录需要对用户进行权限管理，而JWT是无状态的，也就是说使用JWT无法对用户进行管理，只有加辅助手段，如：Spring-Security、Spring-Session、Redis等技术后，才可以进行管理。且JWT生成的token是不可修改的，所以续签问题很麻烦。所以综合来说，JWT不适用于管理用户登录，只适用于对高风险请求再次进行用户确认。续签问题有几种解决办法

1. 每次请求都刷新token

改变了expire后JWT生成的token都会改变。这种方案暴力、浪费资源、性能低下。

1. token快要过期时刷新

这种方案问题在于如何判断快要过期呢？token30分钟之内过期，设定最后5分钟刷新，返回一个新的token给客户端。那客户前25分钟操作，然后最后5分钟没操作，就过期了，需要重新登录。用户体验不好。不太推荐。

1. 使用refreshToken

1.服务端有登录接口/login，刷新token接口/refreshToken。登录时获得一个 tokenA，该token有expireA。客户端需要自己注意时间，在tokenA失效前用 tokenA调用/refreshToken获得新tokenA。。。这里/refreshToken 接口的唯一作用就是在tokenA泄露后，我们可以祈求观音菩萨，让黑客一个不小心，没有及时调用/refreshToken 接口

2.服务端有登录接口/login，刷新token接口/refreshToken。登录时获得一个 tokenA(有expireA) 和一个tokenB(没有expireA)。客户端在随时可以用tokenB 调用/refreshToken获得新的tokenA。。。这里tokenB 和/refreshToken 接口的作用是减少tokenB 在网络间传输的次数，降低了被嗅探的概率。

综合来说，如果使用了https将token放入请求头中传输，实际上已经能很好的保证token在传输过程中的安全性，到了用户电脑本地后的安全性就由用户保证了。所以从这里来说refreshToken没什么实际作用，也就是多一重保险罢了。

1. 使用redis管理token

使用redis管理token，token最大的特点就是无状态，带上就是验证，使用redis不就从无状态改为有状态了？redis中存放refreshToken和token，大致上思路跟以上第三个方案差不多，使用redis管理token的好处是可以分辨不同token，比如用户第一次登录有一个token，第二次登录又有一个token，但是此时第一次登录的token还没过期，假设我使用第一次token登录，一样可以登录成功。如果说系统强制一个账号只能一个用户登录，则会出现用户可以使用不同token进行同一账号登录等问题。redis实现具体可以参照spring-session的实现方案，关键在于解决redis过期机制不保险的问题。最后实现发现变成和传统session一样了。。。那我干啥要用JWT啊。

1. token泄露

token泄露只能是用户问题了，token放在https头部里，基本就很难被盗了。如果还是不放心，可以再做一层加密，不过实话说也没什么用。

二、基于Session与Cookie的登录

没啥好说的，做了很多次了，值得注意的是CSRF攻击和XSS攻击，但是这个都有解决方案。还有就是SSO单点登录。

优点：框架多、实现方案成熟，Spring-Security+Spring-Session基本满足需求。

缺点：大量用户登录时会导致session消耗大量内存

## 1.2 新增用户

新增用户需要缓存吗？需要，从已注册用户缓存中根据手机号取是否已存在该用户，

需要加锁？要的，防止一个手机号注册多个账号。（对手机号进行锁操作）

## 1.3 删除用户

这个删除用户的功能我想了想，要考虑的问题有点多了。

假设我删了用户，第三方绑定的怎么处理？将第三方绑定信息关联用户Id的全部删除，那再次使用第三方登录时就是还要再绑定一个新手机号。

延迟删除用户，将删除用户消息放入RabbitMQ中，默认延迟7天进行删除，除非用户再次要求立即删除，则再放入消息到立即删除队列。修改时间和删除时间都放入RabbitMQ进行处理。同时用户每次上线都提醒距离删除还有多久。

真正删除用户，删除缓存。这里需要注意删除缓存出错问题，如果删除缓存出错，则必须在抛出异常的部分进行再次删除操作，直到成功。（这里需要考虑的方面有点多，暂且先预计实现RedisException注解和RedisAspect切面再说）还有就是当消息到期后，消费端需要判断用户状态是否为待删除状态，是才删除，否则不消费消息。同时还要注意要将第三方登录关联的第三方登录信息全部标记为删除。

取消删除用户，将用户状态设置回0，取消删除用户提醒。

## 1.4 修改用户

修改用户需要修改相应缓存，先更新数据库，再删除缓存。更新数据库更新用户修改数据，修改时间。仅当用户状态为正常时才给修改，否则返回修改错误提示。

## 1.5 查找用户

缓存中通过mobileNum查找用户，如果缓存中不存在相应customer，则将通过mobileNum从数据库中查找相应用户，并将customer写入缓存中。

这里需要加个黑名单，用于存放当天删除用户mobileNum，再加个布隆过滤器，用于判断已注册用户mobileNum，保证用户删除后读取用户依然不走数据库，通过黑名单直接返回。通过布隆过滤器将恶意请求直接过滤，防止缓存击穿问题。黑名单和布隆过滤器可以一天更新一次或更长时间（TODO）

## 1.6 业务逻辑

### 1.6.1 登录

#### 1.6.1.1 手机号短信登录

先从cookie中取出token，若无，则正常操作。若有，则查看session是否存在相应token，若存在，则token写入缓存，user写入缓存。若不存在，则返回已在别处登录提示，重新登录。

用户输入手机号，获取验证码，判断是否存在手机号，若不存在则自动注册（TODO）。然后用户输入验证码，验证码已经放在缓存中了，如果缓存中验证码与用户输入验证码相同，则判断正确。在手机短信验证码模块中，如果用户60秒只能获取一次验证码，若多次获取验证码，则验证码覆盖同一个缓存，同时缓存设置时间为60s。登录成功后返回token，token包含mobileNum信息，key为mobileNum，过期时间为token过期时间，放入缓存。同时将customer放入缓存，key为mobileNum。

多个用户覆盖登录情况，进行token过期时间验证，拦截器拦截收到token，从token中取出customerId，缓存查找customerId，取出redisToken。若token与redisToken相同，则说明同一用户。redisToken为空，则说明用户原token已过期，重新登录。若token与redisToken不相同，则说明多个用户登录，取出token和redisToken时间戳，进行比较，时间较小的一个进行logout踢出操作。

#### 1.6.1.2 账号密码登录

用户输入账号，密码。判断账号是否存在，不存在则返回。存在则通过账号查询出相应用户，将输入密码与数据库盐值进行混合，这里混合顺序为自定义的固定顺序，将混合后的密码进行MD5运算，则出来最终MD5字符串，与数据库中保存密码进行比较，若相同，则生成token，token写入缓存，返回customer，customer写入缓存。

### 1.6.2 登出

用户登出：删除缓存，弹出用户已登出提示，跳回登录页面

拦截器拦截登出：删除缓存，弹出用户已被其他设备登录提示，跳回登录页面

## 1.7 用户缓存

1：mobileNum(customerId，mobileNum)，短时间

2：Customer（mobileNum，customer），长时间

Customer只存储基本信息。

3：CustomerInfo(mobileNum, customerInfo)，短时间

customerInfo存储用户其余信息。

4：验证码（mobileNum，authCode），短时间

5：Token（mobileNum，token），长时间

6：用户登录次数（mobileNum， count），长时间

在秒杀等模块使用。

# 2 商品模块

数据冗余三份，一份放在数据库存储，一份放在Redis缓存，一份放在Elasticsearch搜索。数据库表设计具体看SPU、SKU等概念。

## 2.1 商品模型

SPU(Standard Product Unit)：标准化产品单元。是商品信息聚合的最小单位，是一组可复用、易检索的标准化信息的集合，该集合描述了一个产品的特性。通俗点讲，属性值、特性相同的商品就可以称为一个SPU。

SKU=Stock Keeping Unit（库存量单位）。即库存进出计量的基本单元，可以是以件，盒，托盘等为单位。

举个例子：iPhone6是一个SPU，iPhone6 32G 白色是一个SKU，iPhone6 128G 白色是另一个SKU。

一个SPU对应多个SKU，将较少变化的与较多变化的分开。SPU中存储较少变化的，如：品牌、分类、规格等。

一个SPU对应一个规格，规格就是较多变化的，如：内存、颜色、型号等。

一个规格对应多个规格值，规格值就是变化的具体内容，如：内存32G、红色、X型等。

一个SKU对应一个规格值。SKU就是SPU的具体化。

SPU表(goods\_spu)：spu\_id，spec\_id，category\_id，brand\_id，spu\_name，create\_time，update\_time

规格表(goods\_spec)：spec\_id，spec\_name，create\_time，update\_time

规格值表(goods\_spec\_values)：spec\_value\_id，spec\_id，spec\_value，create\_time，update\_time

SKU表(goods\_sku)：sku\_id，spec\_value\_id，shop\_id，sku\_name，price，stock，create\_time，update\_time

商店表(shop)：shop\_id，shop\_name，create\_time，update\_time

分类表(goods\_category)：category\_id，categoryname，create\_time，update\_time

品牌表(goods\_brand)：brand\_id，brand\_name，create\_time，update\_time

商品查询流程：查询SKU找出spec\_value\_id、spu\_id、商铺、价格、库存。根据spec\_value\_id查出规格值；根据spu\_id查找分类、品牌、型号、spu\_name、spec\_id；根据spec\_id查出规格；一共需要查询4张表，其中spec\_id必须等到spec\_value\_id或者spu\_id查出后才能查找到。查询到的数据直接返回进行拼装处理。

优化：规格、规格值和SKU、SPU可以分为两个线程查找，这样可以快1/2查询时间。然后大部分情况下SPU、分类、品牌放到缓存，不过期，SKU有过期，规格和规格值视情况放到缓存中。

为什么要规格表，不直接把规格放入规格值表中？

答：假设一个商品10个规格，若spec\_name放入规格值表中，则规格表可去掉，可减少查询一张表的时间。商品的规格经常多于10个以上，所以为了减少查询一张表的时间，至少要额外增加7个规格字段的存储，这里就要权衡一下。我认为还是不太值得，所以在这里还是多一个规格表。毕竟查询实际上大多是放在Redis或Elasticsearch中进行的，实际上真正透到数据库的查询很少，所以这里我认为还是以减少数据冗余为主要目标。

为什么不直接把规格放入SPU表、规格值放入SKU表？

答：假设规格放到SPU表中，规格值放入SKU表中，那么实际上只要查询SPU表和SKU表，那么规格信息就出来了。但是问题在于规格、规格值是大字段，同时并不是必须字段。可以想象，用户在浏览购物时，不太会直接关注到规格参数上，一般都是进行浏览，根据外形、品牌、价格等判断后，才真正点进具体产品看规格参数。也就是说，我个人认为规格参数与SPU、SKU参数比起来，不属于热点数据，将它们单独划分为一个表，也就是相当于垂直分表，有利于横向扩展数据库。

新增SPU商品是挺麻烦的事情，必须确定新增的SPU商品类别、品牌、型号。这里有个问题就是SPU商品的新增是需要人工审核的，这点很关键。还有比较麻烦的一点是很多时候很难找出一个稳定的接口。

比如手机，手机的规格较为稳定，有颜色、内存、CPU型号等一系列业界通用的规格 所以手机的SPU容易确定，只需根据品牌、型号就可以确定SPU商品了。同时，手机技术含量较高，意味着SPU商品申请并不容易，可能全部手机SPU商品加起来最多也就几千个，这对数据库来说是绰绰有余的，可以很容易支持，同时需要人工审核也很少。

比较麻烦的，比如纸巾盒，纸巾盒规格不稳定，有颜色、形状、大小、作用位置等等。不同厂家有不同规格，比较难制定通用规格，但是实际上也没想象中困难，只要大而笼统的规格进行划分。

比如颜色分类、安装方式、商家自定义规格等也能满足需求，只是可能比起手机的规格来说稍微显得笼统一点，实际上用户体验也不算太差。

但是最大的问题是纸巾盒太容易做了。纸巾盒没有国家规范，没有行业通用规格。比如我申请一个品牌叫ABCDE，然后申请一个纸巾盒的SPU，可以吗？完全没问题，纸巾盒没有技术难度，随便做。那问题随之而来了，假设我放开了纸巾盒的SPU申请通道，一批人过来申请自己的品牌，申请自己的SPU商品，这样从数据库角度来看，单纸巾盒一类SPU商品可能多达数10W个，这样实际上来讲就已经偏离了SPU的原则了，SPU指的就是特性相同的商品。但是由于纸巾盒技术难度太低了，区分它们的最大特性就是品牌，而品牌又是相对容易申请的一个东西，所以这会对SPU概念进行很大的打击。同时还有一个很重要的因素，就是人工。我为了一个纸巾盒，可能需要新增数十人专门为了纸巾盒的SPU进行人工审核，这是一笔大支出，但是换取回来的却是小回报，甚至亏损，毕竟大部分品牌都是可以随便申请的，也意味着可以随便抛弃。

对于这类SPU商品，比较好的做法是不让申请SPU商品了，后台写死SPU商品和规格。同时由于没有通用的规格支持，所以只能用经验（想象力）进行划分规格，用大而笼统的规格进行划分。比如颜色分类，实际可以用来划分形状、颜色、大小等不同规格。那么除非该类型的SPU商品进行了大的改革，比如说国家出台了纸巾盒行业规范，指定了纸巾盒的形状、大小、材质等一系列规范，提高了纸巾盒的技术下限，那么后台可以进行相应的变化，开放纸巾盒SPU申请通道。

## 2.2 新增SPU商品

新增SPU商品需要两种屏障，一、根据分类、品牌、型号确认该SPU是否存在，若已存在，则不允许继续新增。确认不存在，还可能会存在商家通过使用伪造品牌、伪造型号进行重复添加，需要进行人工审核。在新增时，通过ZooKeeper对spu\_name上分布锁，防止重复添加SPU商品。增加后，人工审核时，依据型号图片进行确认对比。

## 2.3 新增SKU商品

商家要新增SKU商品，同样需要经过后台审核。商家在前端输入好录入商品的参数后，后台人员进行审核（是的，我还没做审核模块）。先依据分类进行SPU选定，再输入具体数据、图片（没做）、视频（没做）、sku\_name（可选，不输入则继承SPU名称根据规格拼接）、价格、库存、规格值（至少一个）。

首先根据spu\_id查询相应SPU商品，这里不进行上锁操作，因为对数据一致性要求不严格。只要修改线程一启动顺利，删除redis缓存，就保证上好分布锁了。如果这里已经判断SPU存在，则当做其竞争成功，SPU有效。然后使用ZooKeeper对sku\_name上分布锁。首先搜索是否已存在相应SKU商品，根据sku\_name进行搜索，然后再将数据插入数据库中。商品数据会放一份到redis中进行缓存，商品Id，商品名称，商品图片，价格等一些页面展示信息。剩余的其他信息，如：规格、规格值等。等到用户点进具体细节，再调用Ajax返回数据。

这里需要注意，如果新增的商品是热点key，要做好缓存预热，否则会导致缓存击穿问题。

## 2.4 删除SPU商品

管理员删除SPU商品，分为两种情况。一、事先已知，如国家新规定，所有农药不允许售卖（假设），一个星期后生效。则提前发公告，通知商家们在这个星期内尽早下架商品。在最后一天，挑选深夜时间，进行删除操作。这里由于提前通知商家，所以可以每天做个定时任务，将Elasticsearch、Redis中相应下架的SKU商品删除，同时不再开放SPU商品新增通道；二、临时删除，如有内鬼新增了不可描述SPU商品，违反国家规定，需要紧急删除。这里需要删除SPU商品及其下属SKU、规格、规格值，关闭SPU商品新增通道。

无论哪种情况，首先先使用ZooKeeper对spu\_id上分布锁。由于这里可能存在数据一致性问题，如，A删，中断；B判断SPU，存在，中断；A继续，删除缓存，删除SPU及其下属所有SKU、规格、规格值，成功；B继续，新增SKU，成功；则B新增了一个SKU。虽然通过每次在查找SKU时先判断SPU是否存在可以修补漏洞，但是新增的SKU依然未被删除，在数据库中依然存在。我不知道有没有什么好的解决方案，要不重新推翻流程设计，要不就隔5分钟后再删除一次SPU及其下属SKU、规格、规格值，确保无遗漏。

## 2.5 删除SKU商品

商家删除SKU商品，也就是将商品状态设置为删除。那么，最简单的做法就是商品状态设置为1，表示商品已删除。在Elasticsearch、Redis直接删除信息，而不是改变标记位，毕竟占内存。可以将删除消息放到RabbitMQ中进行，分别发送删除数据库、Elasticsearch、Redis消息，手工确认保证不丢失数据，删除操作是幂等操作，无所谓重复投递。记得还要删除规格值。

删除需要锁，详情看2.8里的删除操作。（2.3、2.4、2.8三个删除方案结合起来看）

## 2.6 修改SPU商品

与删除方案基本一致，这里要注意的是，如果修改的是唯一属性，如SPU名称，则不仅需要对新名称上锁，也要对旧名称上锁。新名称上锁，保证不会有第二个相同名称修改；旧名称上锁，保证不能以旧名称为分布锁寻找SPU商品。

一、缓存在数据库修改后要一并写入；二、Elasticsearch无法修改文档，只能删除并替换。

## 2.7 修改SKU商品

与SPU修改方案基本一致。

## 2.8 查找商品

### 2.8.1 业务查找商品

如购物车中查找商品等，先从缓存中进行查找，如果缓存不存在，通过黑名单和布隆过滤器将恶意请求过滤掉，再根据spuId或skuId上一个分布锁，然后通过spuId或skuId进行数据库查找，并将查找到的商品写入缓存中。在缓存中查找SKU商品需要上锁吗？不上锁，redis的意义就是为了迅速返回数据，假设在查找redis前需要先上锁，那就会导致redis不起作用，瓶颈存在于分布式锁的速度。

如，A查SPU

1. 缓存有

直接返回。

1. 缓存无，上锁

2.1 上锁成功

数据库找未删除SPU，若SPU不为空，写入缓存，返回。

2.2 上锁失败

缓存找，有，返回，无，重试3次

1. 重试三次失败，抛异常

### 2.8.2 用户查找商品

这里属于搜索模块。所以使用Elasticsearch进行查找操作。通过商品名、商品价格、品牌等关键词进行搜索相关商品。（TODO）

## 2.9 业务逻辑

## 2.10 商品缓存

这里要注意的是入口！！！保证只有一个入口可以操作数据库。分布锁根据入口的属性上锁，如，spuId能找到spuName，由spuName进行数据库查找，则这一系列上分布锁的是spuName。specId找到spuId，spuId再找到spuName等。

1：categoryName（categoryId， categoryName）

2：category（categoryName， category）

3：brandName（brandId， brandName）

4：brand（brandName，brand）

5：spuName（spuId， spuName）

6：spu（spuName，spu）

7：skuName（skuId，skuName）

8：sku（skuName，sku）

## 2.11 锁操作

（1）新增SPU商品

ZooKeeper对spu\_name上分布式锁，插入数据库，再写入缓存。其余相同插入请求全部等待锁解除。

1. 删除SPU商品

删除SPU商品较为复杂，需要删除SPU商品及其下属所有SKU商品和规格和规格值，同时还有数据一致性问题。线程A正在查找SPU商品，线程B正在删除SPU商品。必然需要用到redis，A和B都从redis中找。假设A从redis中找到，判断过，中断；B从redis中找到，完成任务；则A新增了一个已删除的SPU商品的对应SKU商品或其他。为了解决这种问题，选择使用ZooKeeper上分布式读写锁。

解决方案1：修改或删除SPU商品时提前半小时先封锁SPU商品及其相应的SKU商品和规格规格值之类的入口，也就是设置为不可访问。同时限制时间内用户必须完成查找操作，否则操作无效。等到时间过后再修改或删除就可以达成强一致性。这种方案其实我个人认为最好，但是实现起来有点难度。。。

解决方案2：抛弃强一致性，删除SPU商品后联立删除相应下属商品。同时每次搜索SKU商品时都要判断该SKU商品对应SPU商品是否存在，若检测到SPU已删除则将相应SKU商品及规格和规格值删除，或在删除SPU后延时5分钟再进行一次删除操作。这种方案会比较难实现，同时有数据一致性问题。

解决方案3：每次SPU、SKU商品在redis查询前都要上分布锁，保证同一时间只有一个线程进行操作，无论是新增、删除、修改、查询，这样保证数据绝对安全，但是性能瓶颈取决于ZooKeeper分布锁速度。

日后可以方案1与方案2结合，确保安全。除非对数据一致性有严格要求，否则不使用方案3。

考虑极端情况下，客户1在支付订单，商铺2在删除商品。客户1查找SKU商品，商铺2删除SKU商品。客户端1先执行，查找redis，成功，成功支付。商铺2后执行，上分布式锁，删除redis，然后按照业务逻辑删除Elasticsearch、删除redis、删除布隆过滤器对应skuId等等。其余客户端此时由于在redis找不到相应SKU，去上锁找数据库。由于分布锁被商铺2持有，所以都拿不到锁。这时假设有100个客户1都在等待锁，它们都会在规定最大时间内（假设为平均一次查询数据库时间+冗余时间）返回获取失败。其余客户随着商铺2删除了相应Elasticsearch中SKU商品后，无法搜寻到相关SKU商品。如果通过图片、链接等手段强行打开页面，也会在等待一段时间后确定无法获取返回失败。

那么，商铺2什么时候能拿到执行权，上好分布锁且删除缓存，什么时候就能停止客户支付，止损。（这里有没有可能说优化让商铺2线程优先处理呢？这里涉及到操作系统线程优先级调度，需要处理的问题有点多啊，或者看看有没有什么好框架可用。）

这里还有一种更极端的情况，就是redis中恰好SKU商品失效了！客户1先上好了锁，那么商铺2就得等待客户1查找到数据，写入缓存，再从缓存中找到skuName，上锁，删缓存。剩下的客户只要商铺缓存删除成功，则都需要等待商铺2分布锁，知道上锁超时后再返回失败。再极端点，ZooKeeper无法实现读写锁优先级，这里是很尴尬的一点，所以无法使用读写锁，还是得使用独占锁。下一步更止损的方法可以是系统检测到删除方法的时候，将该节点往前的所有节点一一删除，或者直接将节点移到最前面（这里要设计到ZooKeeper源码修改，不太现实），尽量减少损失。

1. 修改SKU商品

与上面方案一致。分布式锁，改数据库，删缓存

1. 新增SKU商品

新增SKU商品前需要查找SPU商品是否存在，正常情况下SPU不存在则返回不存在。由于使用了ZooKeeper，不会出现线程冲突问题，确保了SPU判断安全。然后SKU新增同样需要上ZooKeeper分布锁，流程一致。

1. 删除SKU商品

删除SKU商品较为简单，需要先查找SKU商品是否存在，事务联立删除规格值。存在与否不影响SQL语句执行，同时绝大多数时候为单线程操作，即使多线程操作也是幂等服务，不会造成影响，同样上ZooKeeper分布锁。

1. 修改SKU商品

同上面SPU方案一样。

1. 查找SPU商品
2. 查找SKU商品

# 3 购物车模块

淘宝的购物车必须登录后才能使用，店大就是吊（×），好设计（√）。

京东的购物车是放在cookie中的，而且还没做禁止cookie的处理。购物车至少可以存储70多个商品，说明购物车肯定不存储SKU商品具体内容，具体的内容都放到了redis中拿取。

京东更新了？还是有两套网页系统？不知道，现在发现京东购物车也是必须登录才能用了。

存储购物车大概分三种方式

1. 存储在cookie

存储在cookie，则cookie要保存商品Id，商品数量。

优点：实现简单，减少服务器压力。

缺点：cookie只能4K大小，同时不同设备存储的购物车数据不同，无法同步，还有就是登录时会将购物车商品同步到用户账号上，假设A用户存储购物车到设备A，B用户到设备A上登录账号时，会将购物车上的商品加到B用户的账号上，这就很麻烦，用户体验不好。不过想想，这种情况的前提是A用户不登录账号的时候浏览商品，且B用户要专门跑到A用户设备上登录，概率还是比较小的，但是无法避免这个麻烦。

1. 存储在redis

以sessionId为key，存储商品Id，数量。有意义？我觉得没意义，而且还很危险，如果不需要登录就存储在redis的话，给人爆破怎么办？

优点：有啥优点啊，垃圾

缺点：垃圾

1. 存储在数据库

登录前的购物车数据放在cookie中，登录后的购物车数据一份放在redis中，一份放在购物车，继续冗余保存cookie购物车信息。

优点：数据库中存有用户购物车数据，可以进行数据挖掘分析。且数据保存在数据库中，保证不丢失，使用redis保证速度。

缺点：数据库耗资源，redis耗内存

三种预扣库存方式：

1. 购物车预扣（一般没人用）

一般情况下很不实用的一个设计，除非特殊需求。

1. 订单预扣（大多数）

可能导致恶意下订单不支付，导致商铺缺货。

解决方法：冻结恶意下订单用户；商铺限制每个用户购买商品数量；订单额度高时要交保证金；

1. 支付扣（少数）

保证商品所有库存绝对卖的出去（如果有人买的话），同时还可能有超卖问题。

本系统暂且采用方案2，购物车商品不预扣，发起订单后预扣库存。

流程：选择cookie+数据库的方式，cookie保留1个月的购物记录，如果用户不登录账号，1个月后购物车记录消失。如果用户登录账号，将cookie中购物车与账号购物车合并（要不要给个提示呢？将购物车数据罗列在页面上，询问客户是否添加到账号，如果不添加，则继续保留在cookie上），用户账号上的购物车存到数据库上。

问题：游客的购物车是否需要记录游客IP？

如果游客添加购物车，并注册的话就将游客IP作为凭证将用户插入数据库，将购物车内容按流程添加到新用户；如果游客登录，同样按流程添加到旧账号，但是不关联游客IP，所以应该是不需要的，游客IP单独存放。

问题：cookie中存放了什么？

一个cookie存放游客IP（没登录）或sessionId（自动登录）

一个cookie存放购物车信息（购物车ID、商品ID、商品数量、是否勾选、创建时间、修改时间、删除状态）

优化：

由于每个用户只拥有自己独属的购物车，且购物车由系统设计，所以可以将购物车页面作为静态化页面，节省资源，每次购物车通过Ajax异步请求数据。

## 3.1 购物车模型

购物车选项表（customer\_cartItem）：购物车ID、用户ID、商品ID、店铺ID、商品数量、创建时间、修改时间、是否勾选、商品状态、删除状态、购买状态

购物车商品数量要进行限制，初步限制50个。同时购物车的所有选项都要保存，比如勾选了那个商品，这些信息至少也要放到cookie中，还要放到数据库中。

是否分开两个表？一个是购物车表，一个是购物车选项表？不需要，不使用购物车实体，购物车只存在于程序中，实体只存储购物车选项。

两个页面结算时，购物车买过的商品买过后就标记为已购买了，防止用户重复购买。

用户未登录状态时存储到cookie中（TODO），登录后，使用数据库和Redis进行数据查找。

## 3.2 增加购物车

没有增加购物车这种说法，不存在购物车实体，只存在购物车选项实体。

## 3.3 删除购物车

没有删除购物车这种说法

## 3.4 修改购物车

没有修改购物车这种说法

## 3.5 查找购物车

没有修改购物车这种说法

## 3.6 业务逻辑

### 3.6.1 增加购物车选项

首先使用ZooKeeper对customerId上分布锁，再对前端传入参数进行校验，，因为是个人的操作。缓存是个人的购物车缓存，也就不存在是热点key的问题，所以无需担心缓存击穿问题。同时，这里会有延迟时间过长的问题吗？目前不会，如果会就分库分表。

### 3.6.2 删除购物车选项

首先使用ZooKeeper对customerId上分布锁，再对前端传入参数进行校验，最后进行删除操作。

### 3.6.3 修改购物车选项数量

首先使用ZooKeeper对customerId上分布锁，再对前端传入参数进行校验。先校验页面与缓存specValueId是否一致，若不一致，则根据specValueId刷新数据后再修改数量。同时将刷新后的数据返回页面。

### 3.6.4 修改购物车选项规格值

首先使用ZooKeeper对customerId上分布锁，再对前端传入参数进行校验。先校验页面与缓存specValueId是否一致，若一致，则返回修改前与修改后规格一致。否则，根据specValueId刷新数据。同时将刷新后的数据返回页面。

### 3.6.5 查找购物车选项

#### 3.6.5.1 业务查找购物车选项

通过skuId查找购物车选项，若查找到缓存中有，说明存在；若缓存中没有，可能购物车选项失效或者已被用户删除，再根据skuId到数据库中查找，若存在，则返回；否则，返回失败。

通过cartItemId查找购物车选项，若查找到缓存中有，说明存在；若缓存中没有，则说明没有。

#### 3.6.5.2 用户查找购物车选项

### 3.6.6 勾选商品

每次勾选商品就刷新商品，注意，根据缓存来刷新商品的数据。如，A页面修改规格值，B页面由于是静态页面，所以无变化，勾选后返回缓存数据。假设用户开两个网页，一个在购物车中删除物品，一个在购物车中勾选物品，即使删除了我们也让用户可以勾选到。也就是说已删除的购物车选项一样可以购买。但是已购买的购物车选项无法再次购买。

### 3.6.7 确认下单

确认下单后，将前端勾选商品发送到后台进行判断，注意，这里需要判断是否已购买相同商品！如，页面1购买商品A，已下订单。页面2再购买商品A，判断已购买，返回重复下订单，返回购物车页面时刷新已购买的商品。

生成临时订单Vo，跳转页面，但是不真正写入订单表。

确认下单同样需要刷新商品数量、价格。但是这里有个问题，在锁里好还是分开好？如果分开，实现简单，不用上锁，方便。但是数据有不一致的可能。不过说到底，这里也不需要担心数据一致性啊，毕竟我不可能在这里上分布锁的，这只是生成一个订单展示页面而已，最终确认还是要到订单模块去确认。

### 3.6.8 验证购物车选项数据

首先页面传入cartItemId，根据cartItemId在缓存中进行查找，若找不到，则到数据库中进行查找（除非缓存意外失效，否则不会找不到），还找不到，则说明传入恶意cartItemId，刷新重试。将找到的cartItem缓存和页面传入数据对比，先比较specValueId是否相同，若不同且用户修改规格值，则根据用户新输入规格值进行修改，若不同但用户不是修改规格值，则根据cartItem缓存刷新数量，再进行修改。

一、全验证

1.页面传入cartItem数据与缓存(数据库)cartItem数据比较

验证specValueId是否相等。

若不相等，则以缓存(数据库)cartItem数据为准。

若相等，继续验证shopId、skuId、price。若不相等，返回服务器内部错误提示。

调用SKU接口验证缓存(数据库)cartItem数据与SKU数据是否相等，若不相等，设置cartItem为只读，返回failed（SKU已更新，重新购买）。

使用缓存(数据库)cartItem购买数量验证SKU商品，看是否可购买，若可，返回success（cartItem）；若不可，则设置cartItem只读，返回failed（SKU库存信息）。

1. 规格验证
2. 页面传入cartItem数据与缓存(数据库)cartItem数据比较

验证specValueId是否相等。

若相等，则返回新修改规格与原规格不可相等，failed（）。

若不相等，则根据specValueId查找SKU商品，将SKU数据写入缓存(数据库)cartItem。

使用缓存(数据库)cartItem购买数量验证SKU商品，看是否可购买，若可，返回success（cartItem）；若不可，则设置cartItem只读，返回failed（SKU库存信息）。

## 3.7 购物车缓存

# 4 订单模块

在线支付模块不做，要不自己做（天啊，我怎么会有这种想法），要不调api接口。

订单业务实体相关的业务流程如下：下单（create）--> 买家付款（pay）--> 卖家发货（deliver）-->买家收货（receive）-->退货（rereturn）；此外，还有退款（refund）和评论（comment）。

问题中的‘已评论’由‘评论’行为产生，而‘评论’这个action并不是订单业务实体的核心业务流程，且可能存在多个前向依赖action（支付、发货、收货等），所以应当独立到一个字段标识。

问题中的‘已退货’由‘退货’行为产生，而‘退货’这个action是订单业务实体的核心业务流程，用户非常关心，且只单向依赖于‘收货’action，所以应当记录到订单业务实体表的‘订单状态’字段中。

问题中的‘已退款’由‘退款’行为产生，而‘退款’这个action是订单业务实体的核心业务流程，用户非常关心，但是这个action存在多个前向依赖action（支付、发货、收货等），所以应当独立到一个字段标识。

## 4.1 订单模型

数据库设计原则是：

1. 为提高读的性能，尽可能把写的操作拆分到另一张表，因为对表的更新操作会导致锁表，会降低数据表的读取的性能。  
 2. 交易时一些关联信息可能在后来会被修改或删除，如商品、收货地址等，因此要在订单中记录交易时的商品信息和收货地址，以便后来商品或收货地址被删除的时候，依然能在历史订单中看到快照信息。  
 3. 不要怕拆分成很多表，读的时候多张表关联读取，会比读取一张字段非常多的数据量庞大的表效率高很多。

分为五个数据库表，主订单表、子订单表、订单详情表、用户订单表（冗余），卖家订单表（冗余）。

主订单表以用户Id、商铺Id区别。

子订单表是因为用户可以在一个订单选择卖家的不同商品，每个商品有自己的信息，自己的优惠，甚至不同商品可能使用不同的物流等，需要子订单的概念。

一个主订单可有一到多个子订单表，每个子订单表具体内容由订单详情表决定。

主订单表：主订单ID、用户ID、总商品金额、总商品数量、创建时间、修改时间

子订单表：子订单ID、主订单ID、店铺ID、子订单商品总金额、子订单商品数量、创建时间、修改时间、子订单状态、退款状态（没退，部分退，全退）、退款理由

订单明细表：订单明细ID、子订单ID、商品ID、商品名称、商品数量、商品价格、商品规格、创建时间、修改时间、评论

用户订单表：用户ID、主订单ID、订单状态

卖家订单表：店铺ID、子订单ID、订单状态

订单生成了，但是商铺不卖不行，法律规定给了钱订单就成立了！

## 4.2 增加订单

首先使用ZooKeeper对customerId上分布锁，防止多次插入订单，再对前端传入数据进行验证。增加订单一共有七个步骤，检查购物车商品、插入主订单、插入子订单、插入订单详情、订单放入RabbitMQ、修改购物车表、删除购物车缓存。

检查购物车商品单独分开，后面四个步骤在同一个事务中进行，一旦出错全部回滚。同时还要由于这里商品需要递减库存，所以严格点，skuGoodsService专门开一个接口，用来递减库存，每次递减前先拿到对应skuId分布锁才能递减。

最后两个步骤可以使用单独事务或者放到队列中，因为它们逻辑上为一体，使用事务出错一起回滚，放到队列可以有重试操作，都行。如果后两个步骤出错了，那么用户不知情，但是在购物车中依然存在相应的商品，只不过点击后依然可以购买，用户体验可能会不好。

用户提交完订单后放到缓存中。缓存过期时间，额，看情况，最好就是随着用户下线就直接删除，不过不太可能，不行就搞个30分钟，过期就再拿。

秒杀的时候，数据库量瞬间达到高峰，用户提交订单后马上写入RabbitMQ、Redis中。然后实际插入数据库慢慢来，要注意的是RabbitMQ的持久化，还有RabbitMQ消息挤压，还有消息幂等性，还有设置死信队列等各种问题。

## 4.3 删除订单

订单删除状态设置为1，先写数据库，再删除redis。

## 4.4 修改订单

修改状态，如退货、退款、取消订单等，需要几个表同时修改。

还有RabbitMQ中取消超时订单

## 4.5 查找订单

### 4.5.1 业务查找订单

查找订单的时候，根据主订单Id查找，但是会联立查询出子订单给用户。订单详情等到用户点进具体订单细节再显示。

### 4.5.2 用户查找订单

### 4.5.3 卖家查找订单

## 4.6 业务逻辑

## 4.7 订单缓存

（1）(orderId, order)

# 5 活动模块

## 5.1 活动模型

## 5.2 增加活动

## 5.3 删除活动

## 5.4 修改活动

## 5.5 查找活动

## 5.6 业务逻辑

## 5.7 活动缓存

## 5.8 锁操作

# 6 优惠券模块

由于促销模块需求变化太快了，面向对象跟不上需求变化，所以使用数据库化，不用面向对象。其实会不会使用面向过程也能设计良好呢？这里有时间可以在想想。

优惠规则：渠道限制（领取优惠券的渠道）、对象限制（限定某类用户、某个用户）、金额限制（累计总优惠金额、单张优惠券金额）、数量限制（优惠券总数量、用户单人获取总数量）、扣减规则（直接减、折扣、传递）、封顶规则（最多减多少、最多减几折）。

肉眼可见的繁杂，多，乱。各种规则之间的优先度、互斥非常复杂。同时还不是最终的目标，可预想未来会发展出更多更乱的优惠规则。

优惠券与buf设计分开，优惠券存储buf效果，在订单中使用优惠券的具体作用就是上一个buf和一个debuf。

数据库化，让一个订单拿起一个优惠券，就得到一个效果；换另一个优惠券，就得到另一个效果。这里的效果就是一个或多个buff和一个或多个debuff。

整个优惠券体系被分为一组组的秒杀buf/debuf、满减buf/debuf，折扣buf/debuf等等。这些buf/debuf会影响价格公式中的某个因子或者造成直接减价效果；而减价效果又分为无门槛减价和满减和折扣减等等；最后则是一套影响范围判定机制。

Buff系统如何设计呢？

假设如下：1.折扣Buff；2.满减Buff

举例，用户拿到了一张满200减20的优惠券，他得到了一个buf和一个debuf。

buf是造成满减效果，效果相当于设置价格公式为money - a\*x（a为满足条件，x为次数），当不足200时，a为0，则无优惠。

debuf是用了满200减20不能与其他优惠券叠加。debuf可以叠加，比如满减最多叠加5层，也就是最多减100。

用户下订单选择了满减券，于是后台判断其他优惠券，查询商品优惠状态，发现处于满减debuf，返回无效。

buf设置在订单模块里的购买公式中，debuf设置在订单模块的状态中。

之所以如此设计，主要是因为优惠券需求变化太剧烈了，面向对象设计满足不了。有可能这个月说满减与其他优惠券不能叠加，到了什么狗屎节日后，老板一拍脑子，说为了冲营业额，都可以叠加。那么只需要在数据库中将满减优惠券的debuf改成无debuf，那么就完成改动了，与代码解耦。过了节日后，老板说又改回来，那还是同样的，在数据库中改回来就可以。

优惠券先生成后给用户领取，因为用户领取是一个高并发的操作，不宜做过多的业务逻辑进去，尽量简化用户领券的逻辑，简化成一个步骤最好，那就是从提前生成的"券池"中拿一个券SN号即可。

## 6.1 优惠券模型

优惠券表：

优惠券Id、优惠券描述（json存储）、渠道Id、渠道名称、范围Id、范围名称、作用Id、总数量、单人总数量、过期时间、特殊设定

特殊设定，用来开特例。如：用户A优惠券花了10块钱，用户B是A的推荐人，可以获得1块钱。这可以由特殊设定使用LUA脚本来搜索传递目标。

渠道表：

渠道Id、渠道名称

范围表：

范围Id（Id须有意义，由相应范围类型的Id组成）、范围名称（可为Category、Brand、Shop、邮费及其四种任意组合）

作用表：

作用Id、作用类型Id、作用效果、作用优先级（这里这么设计好吗？我认为很难在设计之初就明确优先级吧，而且优先级的设计也很难设计，优先级分多少级好呢？同一优先级又如何处理呢？但是如果不这样做又如何明确作用的优先级呢？直接在程序中判断吗？那不就跟数据库一样吗？或者优先级再分一个表？）、作用互斥（json存储互斥作用Id）、叠加次数（1-n次）

作用类型表：

作用类型Id、作用类型名称（目前只有满减、折扣、传递减）

## 6.2 增加优惠券

（模块删除了，写的时候太乱了，全部推倒重写）

## 6.3 删除优惠券

## 6.4 修改优惠券

## 6.5 查找优惠券

## 6.6 业务逻辑

## 6.7 优惠券缓存

## 6.8 锁操作

# 7 秒杀模块

秒杀商品单独生成，单独存库，与原商品不共享。

秒杀可能造成问题：超卖、少卖、MySQL性能差、Redis持久性差、数据不可靠

解决方案：前端静态化、验证码、IP限流、有损服务

后台单机：MySQL+Redis+RabbitMQ

逻辑层

首先应该进入校验逻辑，例如参数的合法性，是否有资格，如果失败的用户，快速返回，避免请求洞穿到 db。

异步补单，对于已经扣除秒杀资格的用户，如果发货失败后，通常的两种做法是：

事务回滚，回滚本次行为，提示用户重试。这个代价特别大，而且用户重试和前面的重试策略结合的话，用户体验也不大流畅。

异步重做，记录本次用户的 log，提示用户【稍后查看，正在发货中】，后台在峰值过后，启动异步补单。需要服务支持幂等

对于发货的库存，需要处理热 key。通常的做法是，维护多个 key，每个用户固定去某个查询库存。对于大量人抢红包的场景，可以提前分配。

放到RabbitMQ中进行，每个商品一个队列，同一用户的请求保证放到同一队列中，RabbitMQ可以做定长队列，比如说110%容量的队列，只有前100%可以付款，如果用户取消订单，则继续从队列往后拉去用户。

存储层

对于业务模型而言，对于 db 的要求需要保证几个原则：

* 可靠性

主备：主备能互相切换，一般要求在同城跨机房

异地容灾：当一地异常，数据能恢复，异地能选主

数据需要持久化到磁盘，或者更冷的设备

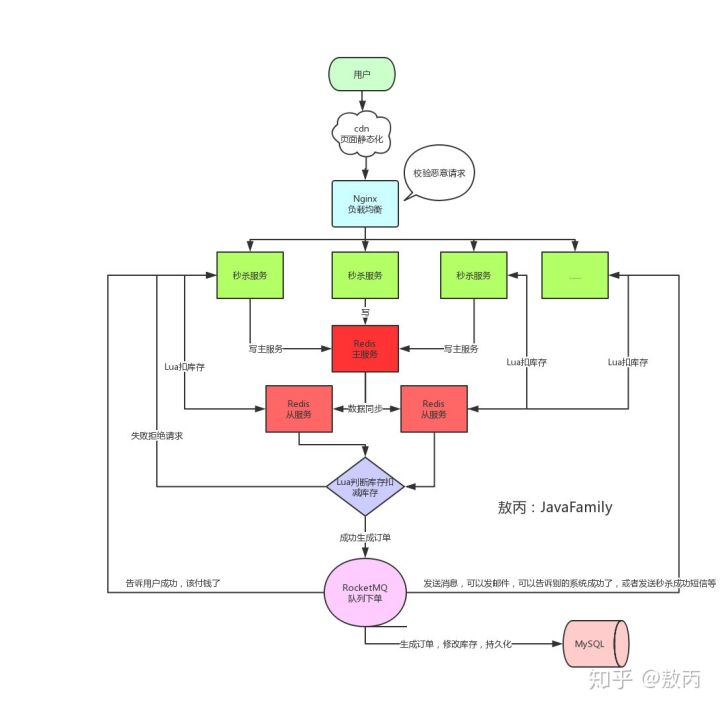
* 一致性

对于秒杀而言，需要严格的一致性，一般要求主备严格的一致

降级预案：当前端顶不住时，随机抛弃请求，后端也可以随机抛弃。

秒杀链接要用MD5加密随机生成，防止链接泄露。

限流&降级&熔断&隔离



## 7.1 秒杀商品模型

秒杀商品表和商品表分开，原因：每次秒杀活动都会操作数据库，修改秒杀商品的字段，每次秒杀时候对应的秒杀时间段，秒杀实现，秒杀功能，价格等都不同，商品表不易于维护。

秒杀商品Id、秒杀商品名称、秒杀商品规格名称、秒杀商品规格值、秒杀商品价格、秒杀商品库存

秒杀活动开始时间与结束时间由秒杀活动控制，规定秒杀商品在活动开始半小时前不能再调整，因为要做好页面静态化、缓存预热等问题，改动会有很多麻烦。

秒杀URL需要动态化，使用用户输入验证码进行动态化。用户输入验证码，根据验证码作为盐值，加到商品Id中，使用MD5算法进行加密。这样即使源码、数据库全泄露了，也不会导致秒杀商品地址提前泄露。

如：黑客知道了商品Id、MD5算法，但是他不知道验证码！验证码随机生成，由时间推算出来。同时，验证码具有相当多种实现方法，可以数字、算式、图像等等。系统可以由人工提前半小时决定选用那一套验证码进行验证。除非黑客提前把所有生成验证码的算法答案都计算出来，并且提前知道系统决定选用那一套验证码进行验证。我觉得吧，都泄露到这种地步了，算了算了，爱咋咋地吧。

## 7.2 增加秒杀商品

正常新增SKU商品流程。

## 7.3 删除秒杀商品

正常删除SKU商品流程，注意在活动开始前半小时后不能删除。

## 7.4 修改秒杀商品

正常修改SKU商品流程，

## 7.5 查找秒杀商品

## 7.6 业务逻辑

### 7.6.1 秒杀商品

商品库存放到redis中。

用户进入秒杀页面，点击秒杀，发送用户ID、验证码、商品ID到后台，后台校验验证码，正确后再通过用户ID在redis的set集合中判断用户是否已秒杀该商品，若已秒杀成功，返回。若无，则递减库存，若递减失败，返回秒杀失败；若递减成功，则生成订单，放到RabbitMQ队列，将用户ID放入redis的set队列，告诉用户已秒杀成功。要注意的是若用户Id写入redis失败或RabbitMQ队列放入失败抛异常，则记得要递增回秒杀商品库存。

优化手段：商品库存key可以分多个key，如，商品A库存1000，分为10个key，每个100，这样下来并发量就上去了10倍了。

## 7.7 秒杀商品缓存

# 8 评论模块

用户对商品进行评论，问题：如果用户已删除怎么办？已删除不影响，将头像、名字改为已注销，用户点进看用户都是注销模板。

## 8.1 评论模型

# 9 通用模块

## 9.1 全局响应对象

## 9.2 全局异常

# 10 所用技术

## 10.1 前端

## 10.2 后台

### 10.2.1 SpringBoot

### 10.2.2 SpringMVC

### 10.2.3 MyBatis

### 10.2.4 MySQL

### 10.2.5 Redis

### 10.2.6 RabbitMQ

### 10.2.7 Elasticsearch

### 10.2.8 ZooKeeper

树形结构，通过持久化父节点下创建临时子节点+递增数字，确保可用性。通过线性唤醒规避羊群效应。

如：

SPU/lock/0001、SPU/lock/0002、SPU/lock/0003、SPU/lock/0004……

创建临时节点后，监视前一个节点若前一个节点消失，则判断自己是否为序号最小，如果是，则获取锁。若0003服务器宕机，则临时节点检测不到，自动删除。0004判断，发现不是序号最小，继续监视前一个节点。其中，临时节点都是可重入的，比如0001，获取锁后，一共加锁3次，计数器负责计数，直到计数器为0时才真正释放锁。释放锁时删除临时节点。

优点：锁健壮安全，性能消耗小

缺点：性能差，每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后Leader服务器还需要将数据同步到所有的Follower机器上。

# 11 杂谈

## 11.1 Vo

千万不要自作聪明优化Vo，Vo就是要一个页面对应一个Vo，顶天两个页面对应一个Vo，千万不要弄一个主Vo，其余Vo继承之类的，Vo的作用就是因为PO字段太多，页面不一定需要这么多字段，所以单独一个Vo减少网络负担。如果妄图弄一个主Vo，其余Vo继承的话，主Vo只能为空，那实际上这样做相当于以前的Serializable，仅仅是一个标记，还不如用注解进行替换。

## 11.2 分布锁

ZooKeeper分布锁无法序列化，这导致锁传递有点小问题了。暂时先到三种实现方法：

一、在Controller中使用锁，在Service中使用锁，比较不好理解，且需要使用两个锁，但是规范性高（个人认为），Service层不绑定返回值，可重复调用。性能差，理解难度大，但是可以直接复制就行，不需要考虑别的。。。

二、在Controller中不使用锁，锁操作统一放到Service中使用。由于存在Service内部方法调用，所有依赖于注解的内部方法调用都要取得代理对象调用。且Service方法大部分返回值为CommonResult，与Controller绑定。

锁只需使用一个，性能比上面方法好，但是Service基本上不可重用了，因为全返回的CommonResult。

三、在Controller中使用锁，且锁传递到Service中，除个别方法（如新增）可不传递锁以外，其余方法都要传递锁，也就是说Service方法基本上与分布锁绑定死，必须在调用方法前先构造出一个分布锁。同样只需要一个锁，且所有锁定义都在Controller中，Service只涉及上锁解锁。

四、上一个模块的改进版，在Controller中使用锁，且额外新增第三方自制工具类，专门存储锁，使用LRULinkedHashMap进行存储，Controller与Service中的锁都在第三方工具类中存取，这样Service与分布锁解耦了，可复用。分开两个方法，一个setLock、一个getLock，setLock无论有无锁都直接新生成锁，getLock先去第三方取锁，无锁时再生成。setLock适用于对同一个锁上多个不同节点的情况，但是由于增加了第三方工具类，所以复杂度提高了，理解难度大点。同时如果在分布式环境还涉及一个问题，分布锁统一存储还是单独存储呢？统一存储好处就是省去创建时间、内存，但是多一次服务器调用时间。单独存储可能会有用户请求在A服务器创建锁，过几秒后另一个请求到了B服务器，又创建了相同的锁。统一还是单独存储我觉得还是得看创建锁的代价大还是调用服务器代价大。

决定以后的模块都沿用方案四设计。