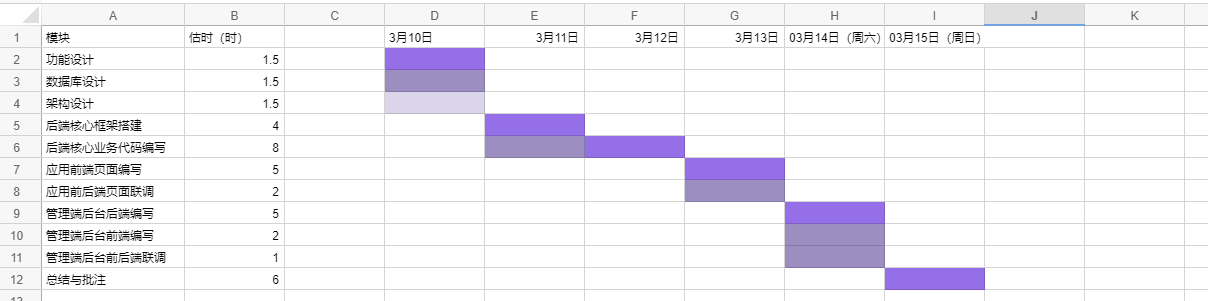
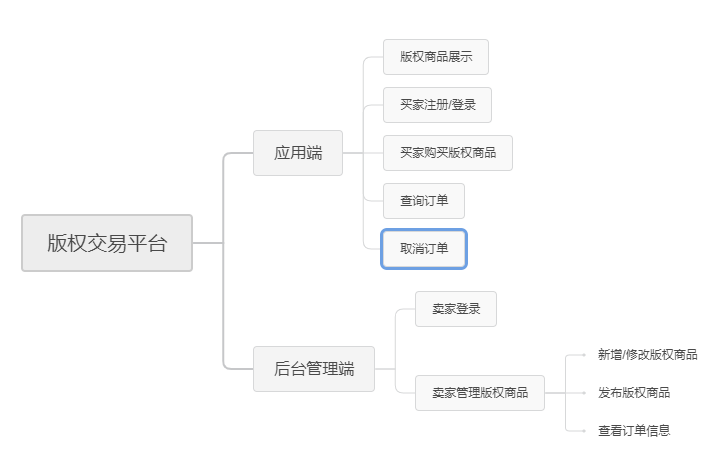
# WBS (工作任务拆解甘特图)



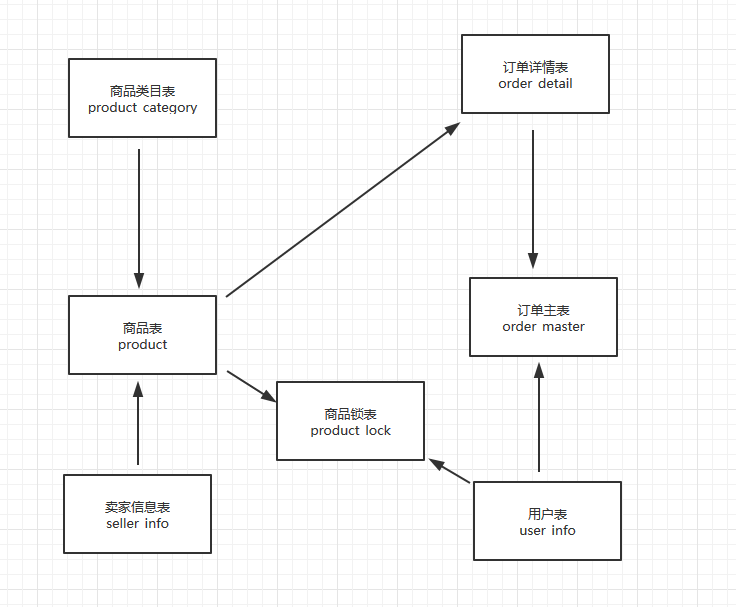
# 功能结构脑图



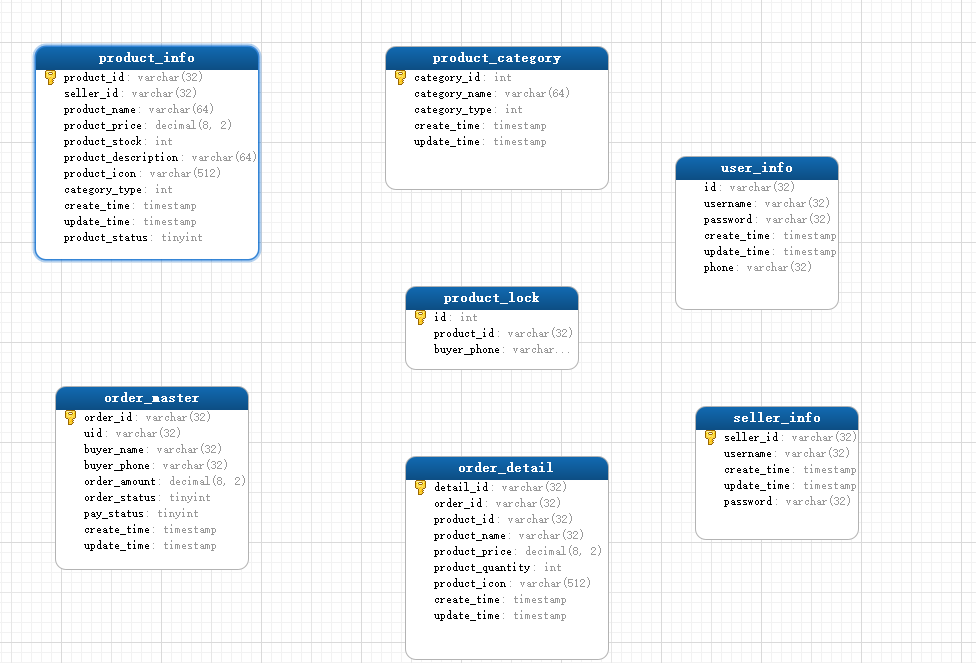
# 数据库设计

目前的数据库没有严格按照第三范式来设计，是因为在时间和空间上进行了取舍考量

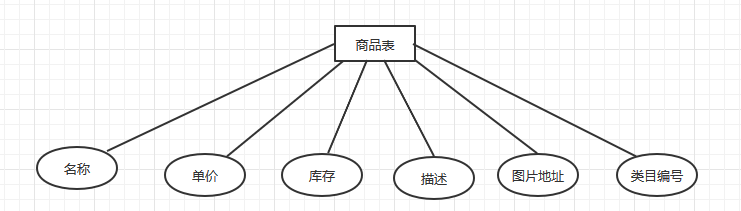
## 表之间的关系：



## 具体模型设计：

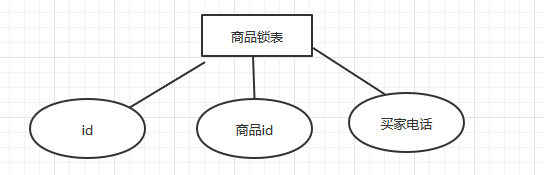


## 商品表



本项目中的版权交易个人理解用不到库存，为方便以后的业务拓展，商品表中预留了库存字段，但实际上没用到。

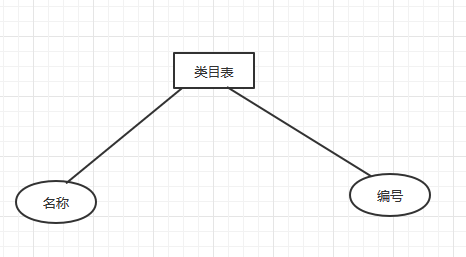
## 商品锁表



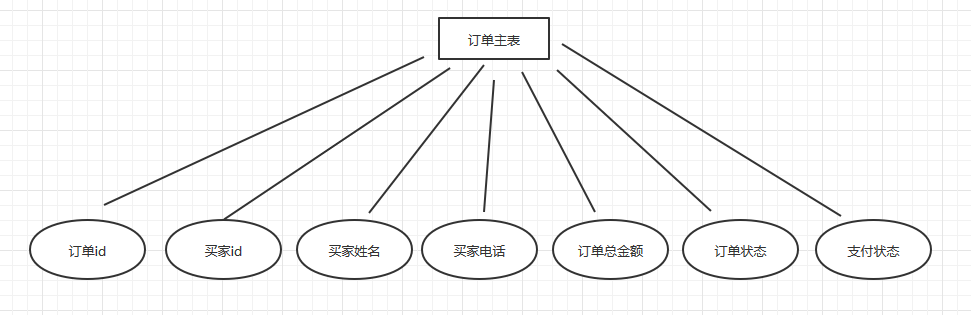
同一个版权商品同一个手机号只能购买一次，此表在商品id和phone之间建立了唯一约束，有几个好处：  
1、分布式架构下锁商品  
2、避免表单的重复提交

3、避免版权的重复消费

## 类目表

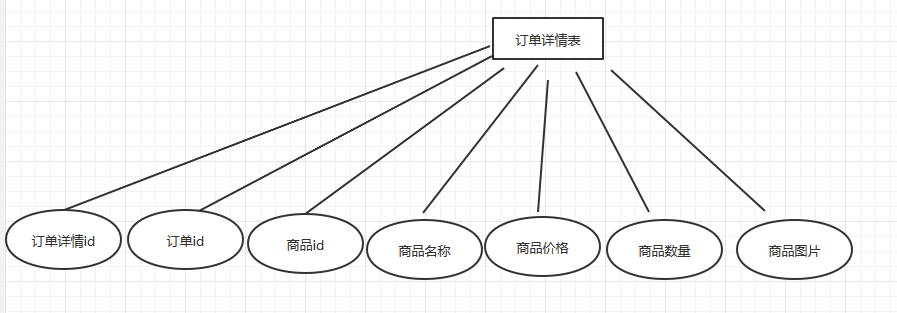


## 订单主表



此表没有按照第三范式严格设计，买家相关的应该只需要存储买家id就可以，但是出于在时间和空间上的考量，空间换时间，所以选择这样设计

## 订单详情表



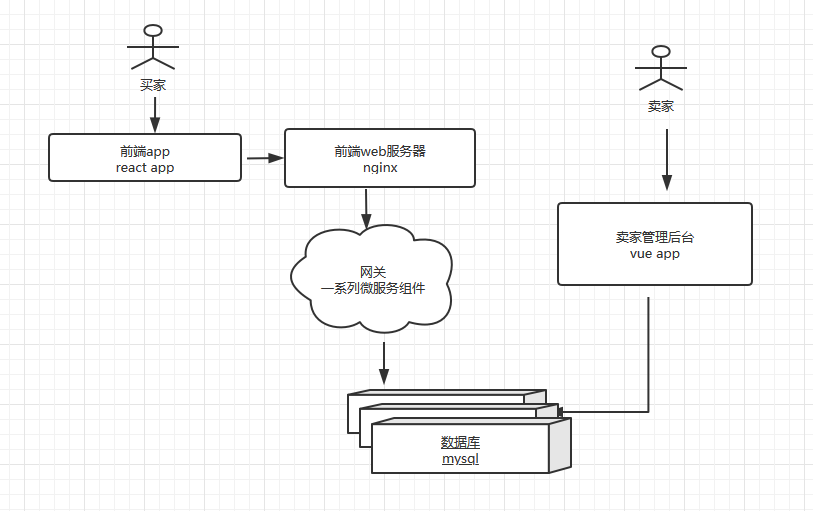
此表没有按照第三范式设计，商品相关的应该只需要存储商品id，但是出于在时间和空间上的考量，空间换时间，所以选择这样设计

在phone和商品id间建立唯一约束，主要用于防止同一个用户对同一个版权进行重复交易

此表中的商品数量针对版权交易中的场景均为1

# 架构设计

## 项目架构图



本项目分为后台管理和前台应用：

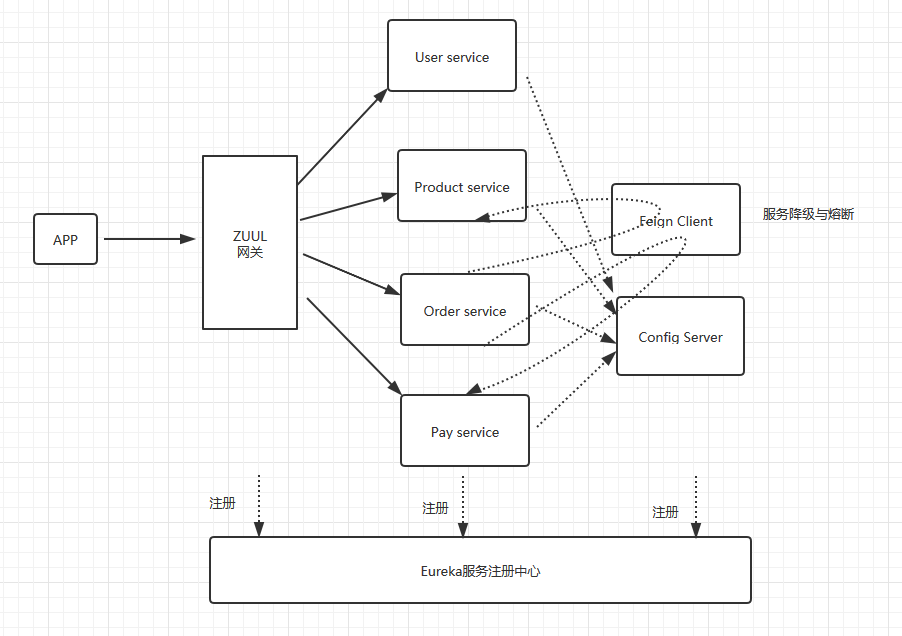
**后台管理**：主要针对卖家端对版权商品进行管理，后端使用springboot开发单体应用，前端使用vue技术栈做操作页面

**前台应用**：主要针对买家端对版权商品进行浏览跟购买交易，后端使用springCloud开发微服务分布式应用，前端使用react技术栈做操作页面

**数据库**：通常来说，微服务里的其中一个拆分就是数据拆分，本项目中暂且不考虑数据库拆分和主从集群，所有数据都存储在一个数据库实例里

**Nginx**: 通常前端应用也需要使用类似keepalive的东西做的前端应用高可用，本项目里也暂且不考虑

## 后端微服务架构图



**ZUUL网关**：提供统一访问入口控制、使用限流算法对访问浏览进行控制、跨域设置、访问权限控制（也可集成zipkin进行链路追踪，本项目暂且不集成）

**Eureka**：服务注册与发现，是的业务服务只需关心业务本身逻辑，无需关心目标服务的获取，使得每个服务都可多示例部署

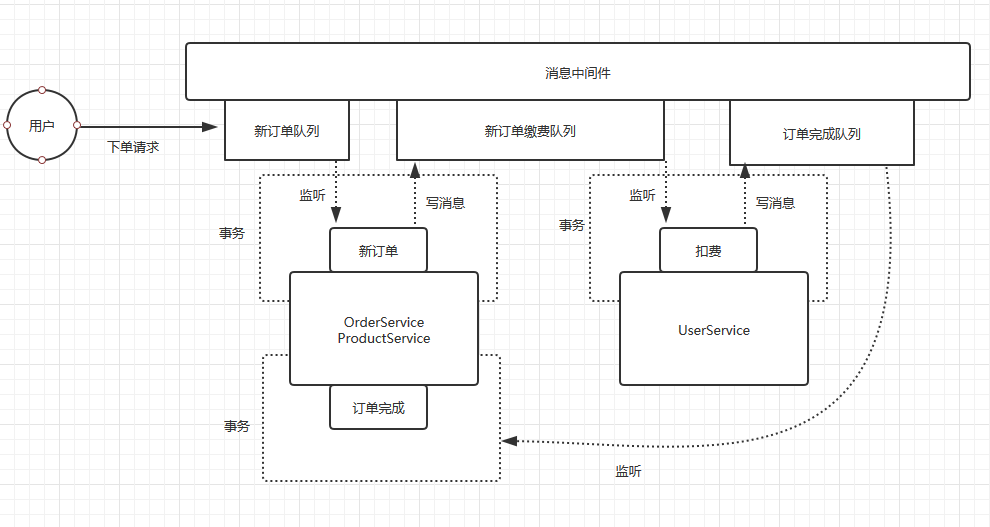
**Config Server**: 统一配置中心，使得修改配置文件无需重启实例，无缝修改

**Feign Client**：自动集成负载均衡策略，改变传统的http请求方式，其中再集成降级与熔断机制，给与用户友好提示

其余各服务都根据业务独立划分成单独的服务，为的就是当业务量扩大时，方便横向扩展和独立维护

**PayService**：正常来说支付服务应该如上图架构所示单独拆分出来，因为这个体系可以做到很大，比如里面又分微信支付，支付宝支付，银联支付等，而且还应该有支付日志，支付记录等，这些在本项目中将不做考虑，本项目的支付是直接在用户表里扣除余额，所以直接放在UserService服务里

## 下单架构设计图



因为下单的这个链路可能会涉及到很多服务的调用，比如商品服务、订单服务、支付服务、短信服务（本项目暂且不考虑），如果使用传统的同步调用方式可能会出现以下几个问题：

**1、用户等待时间长，体验度极差**

**2、如果某个服务不可用，就导致整个服务坍塌，出现雪崩**

**3、服务与服务之间强耦合，牵一发则动全身**

考虑到以上几点问题，所以本项目采用消息驱动模式来实现下单流程，结合分布式的CAP定论，需要注意以下几点

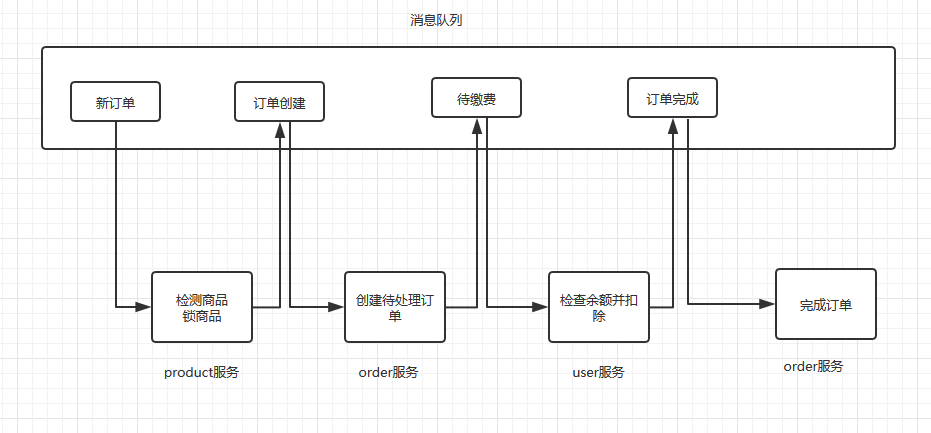
**1、各服务监听自己的消息实现业务解耦**

**2、如果同一服务里存在多个数据源，为了保证事务的一致性，应根据实际情况进行分析，在JTA、事务同步、链式事务里权衡考量**

**3、为了保证不重复消费消息，所有服务在设计接口的时候应考虑幂等性（接口层、sql层）**

**4、为了防止消息丢失或业务处理失败，应该使用定时任务去检测弥补（本项目中暂时不采用分布式任务调度）**

## 下单具体编码设计图-顺利下单



## 下单具体编码设计图-锁商品失败

