**商品秒杀系统**

# 开发环境

Java：15

Maven：3.2.2

Mysql：5.5

SpringBoot：1.5.9

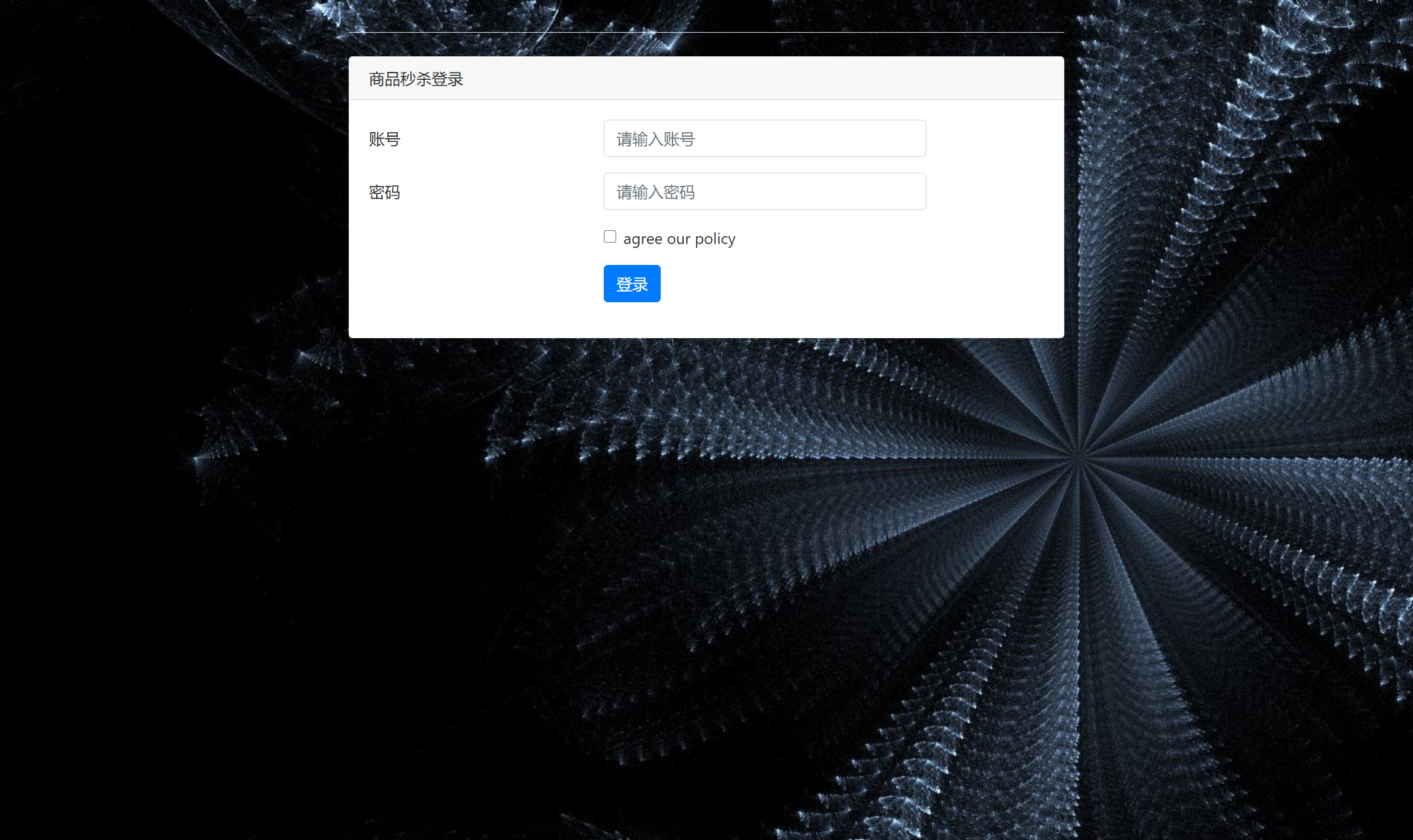
Redis：3.2

RabbitMQ：3.7.14

IDE：IntelliJ IDEA 2020.2.2 x64

# 功能特性

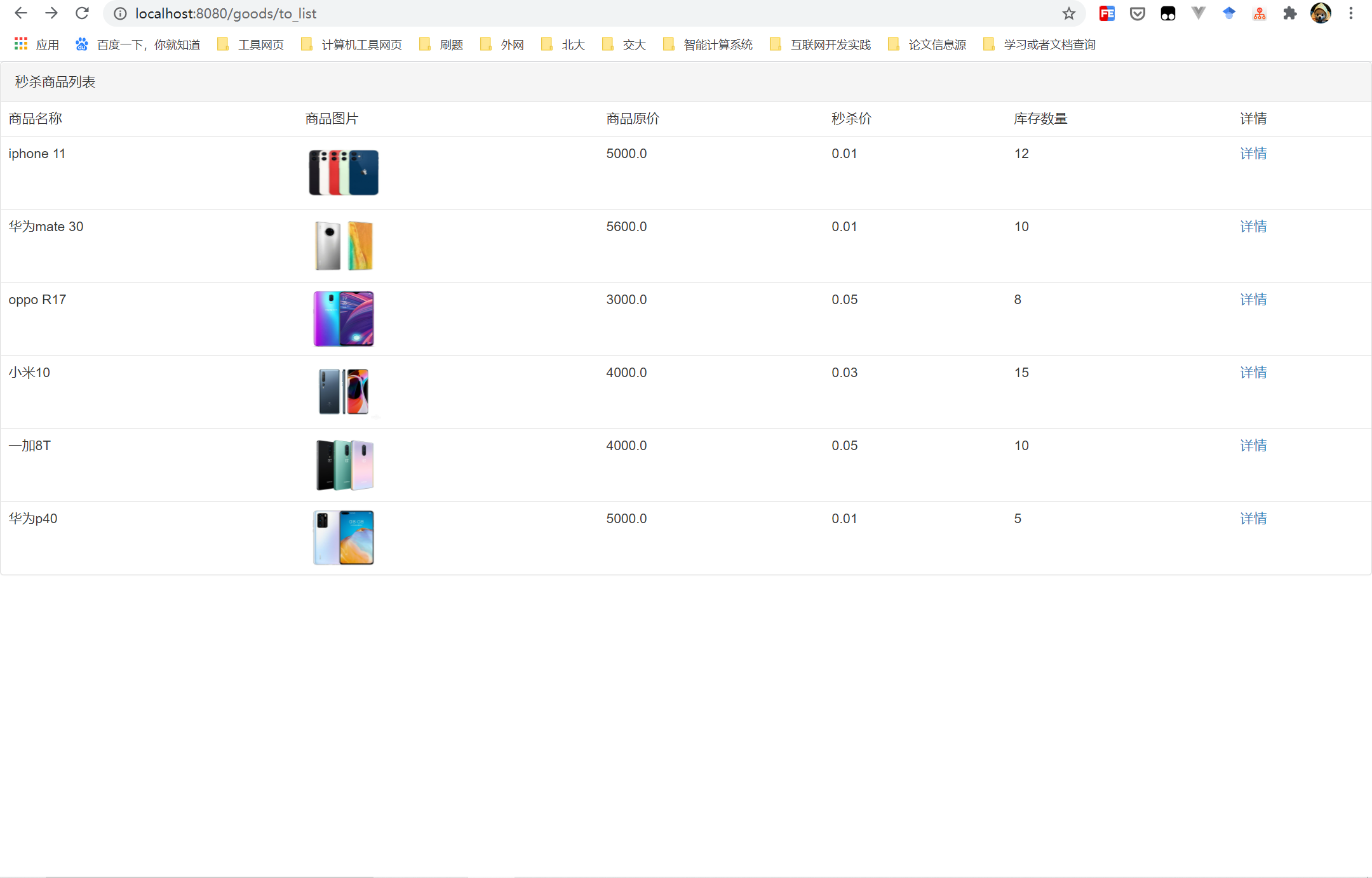
客户端登录页面：



账号密码基于两次md5码校验

系统会记录用户登录日志写入数据库

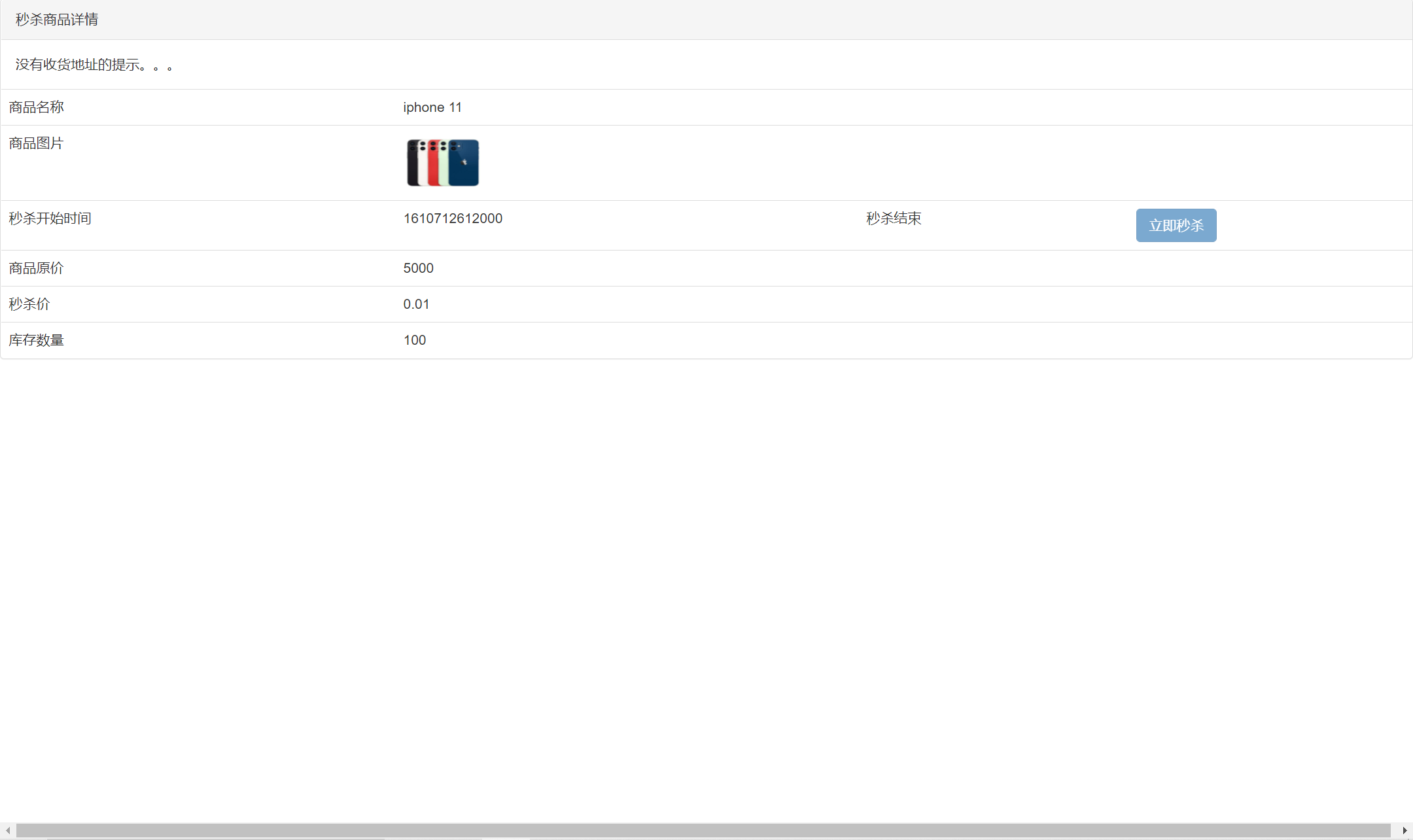
商品秒杀列表页面：



展示商品名称、商品图片、商品原价、商品秒杀价格、库存数量（这里也可以是其他类型的商品，不一定是手机）

点击详情进入详细秒杀商品页面

秒杀商品详情页面：

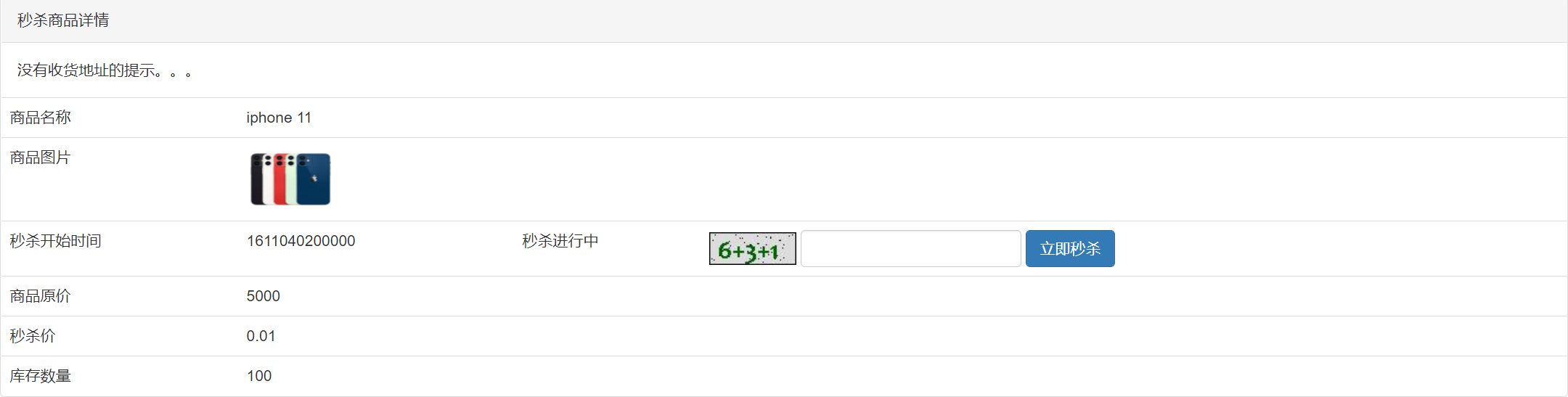


显示商品秒杀开始时间

秒杀倒计时，时间是实时刷新的



商品秒杀时间到了后，可以点击立即秒杀参与商品秒杀，需要填写验证码，后端会验证验证码填写的正确性



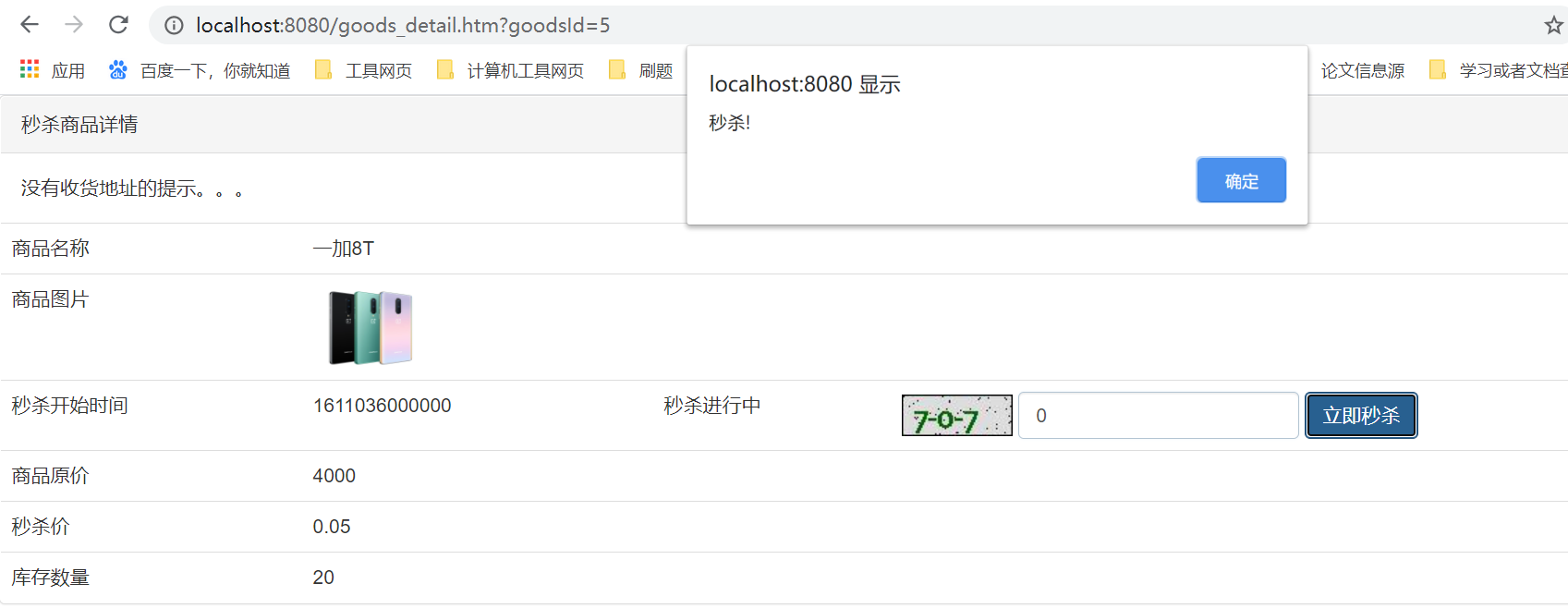
同一个用户不能秒杀两次商品，否则显示非法请求



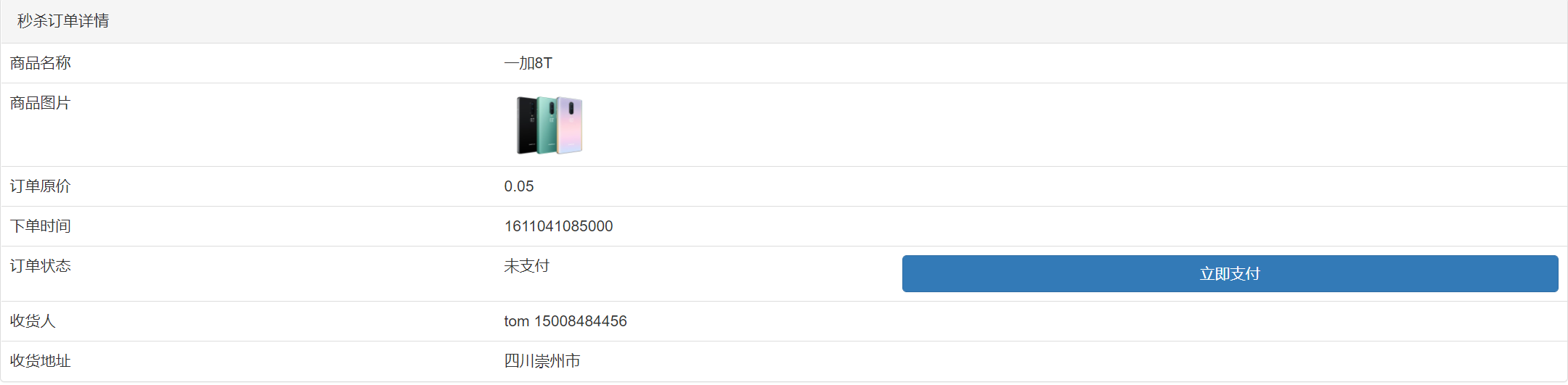
立即秒杀按钮点的太频繁，显示访问受限



秒杀成功显示



随后进入支付页面



# 系统功能实现

* **密码口令设计**：两次md5加密设计

用户端：PASS=MD5(明文+固定Salt)，

服务端：PASS=MD5（用户输入+随机Salt）

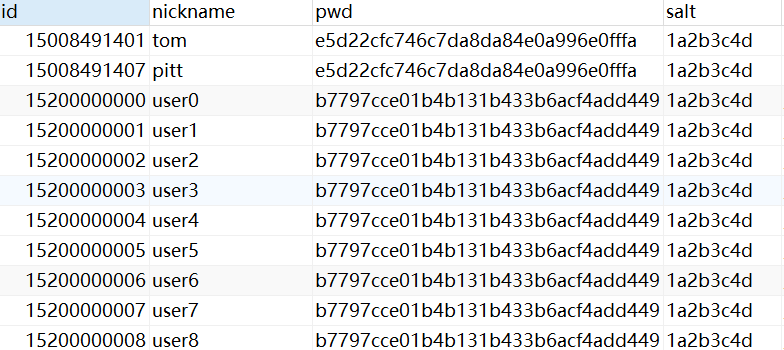
第一次 （在前端加密，客户端）：密码加密是（明文密码+固定Salt）生成md5用于传输，由于http是明文传输，当输入密码若直接发送服务端验证，此时被截取将直接获取到明文密码，获取用户信息。加盐值是为了混淆密码，原则就是明文密码不能在网络上传输。

第二次：服务端接收到已经计算过依次MD5的密码后，我们并不是直接存至数据库里面，而是生成一个随机的salt，跟用户输入的密码一起拼装，再做一次MD5，然后再把最终密码存在数据库里面。

第二次的目的：

防止数据库被入侵，被人通过彩虹表反查出密码。所以服务端接受到后，也不是直接写入到数据库，而是生成一个随机盐(salt)，再进行一次MD5后存入数据库。

数据库里面存的是做了两次MD5的用户密码与其对应的salt值



在我们登录的时候，要去取得数据库里面对应用户的密码和salt值，然后后台接收了前端做了一次MD5的密码formPass,然后将这个formPass去和数据库里面的salt一起再做一次MD5，然后检测是否与数据库里面存的那个密码一致。

* **参数校验**

JSR 303 用于对Java Bean 中的字段的值进行验证,使得验证逻辑从业务代码中脱离出来。是一个运行时的数据验证框架，在验证之后验证的错误信息会被马上返回。

使用方法：在需要验证的参数前面打上标签注解@Valid，那么此注解就会自动对该Bean 进行参数校验。具体校验规则在该Bean内部实现。

可以直接使用@NotNull、@Length(min=32)等注解进行验证，避免重复的校验代码，只需在传入的参数上打上注解就可以进行参数校验，避免代码冗余。

* **分布式session**

我们的秒杀服务，实际的应用可能不止部署在一个服务器上，而是分布式的多台服务器，这时候假如用户登录是在第一个服务器，第一个请求到了第一台服务器，但是第二个请求到了第二个服务器，那么用户的session信息就丢失了；

本系统利用一台缓存服务器集中管理session，即利用缓存统一管理session；

用户登录成功之后，给这个用户生成一个sessionId(用token来标识这个用户)，写到cookie中，传递给客户端。然后客户端在随后的访问中，都在cookie中上传这个token，然后服务端拿到这个token之后，就根据这个token来取得对应的session信息。token利用uuid生成。

token对应的是哪一个用户,将我们的私人信息存放到一个第三方的缓存中，当访问其他页面的时候，就可以从cookie中获取 token,再访问redis 拿到用户信息来判断登录情况了。

* **商品详情页和秒杀倒计时功能的实现**

后端根据商品的Id去数据库中获取秒杀开始时间和结束时间，以及系统当前时间，并定义秒杀剩余时间变量和秒杀状态，计算出相应的值，传给前端，前端拿到之后，做相对应的显示逻辑效果。

获取了商品的秒杀开始时间和结束时间，如果秒杀没有开始，那么计算一个还剩多少时间，开始，并且定义一个状态status来表示一个秒杀的状态，0代表秒杀还未开始，1代表秒杀正在进行，2代表秒杀已经结束，秒杀还未开始的情况还要计算出倒计时，(int) ((start-now)/1000)，然后将status和remailSeconds传到前端去。

前端的处理：需要写一个方法来控制秒杀按钮的可以点击与不可点击的情况，没开始的时候按钮，不可点击，开始后显示按钮，此时可以点击，但是结束也不可点击。并且从倒计时状态到正在进行秒杀状态的时候要动态切换文案。

* **秒杀业务逻辑**

**流程：**

1. 判断登录
2. 根据商品id从数据库拿到商品
3. 判断库存，库存足够，进行秒杀，不足则结束
4. 判断是否重复秒杀（我们限制一个用户只能秒杀一件商品，怎么判断？即从数据库根据商品和用户id 查询秒杀订单表，如果已经存在订单，说明重复秒杀 ，给出提示，退出）
5. 以上都通过，那么该用户可以秒杀商品

执行秒杀事务的时候，先生成详细订单，然后生成秒杀订单，为了进一步确保秒杀过程中一个用户只能秒杀一件商品，我们给秒杀订单表miaosha\_order表添加一个**唯一索引**，如果再次插入相同的id与goodsId相同的字段，那么将不会被允许，从而在事务中插入失败而回退。

为秒杀订单表添加唯一索引：

* **超卖现象：**不同用户在读请求的时候，发现商品库存足够，然后同时发起请求，进行秒杀操作，减库存，导致库存减为负数。

更新数据库减库存的时候，进行库存限制条件，sql要多加一个stock\_count > 0；那么stock\_count>0的时候才去更新，数据库本身会有锁，那么就不会在数据库中同时多个线程更新一条记录，使用数据库特性来保证超卖的问题

# 性能优化策略及实现

* **页面优化技术：页面静态化**

将页面直接缓存到用户的浏览器上面

好处：用户访问数据的时候，不用去请求服务器，直接在本地缓存中取得需要的页面缓存。

实际就是将前端分为两部分，一部分是不改变的html块，第一次是去请求后台要渲染好的html页面，之后的请求都是直接访问用户本地的浏览器缓存的html页面 ，都是静态资源，还有一块就是数据，它只要后端传数据到前端即可，用到ajax技术来请求数据。

* **秒杀优化：核心是减少数据库的访问**

1. 使用Redis预减库存减少对数据库的访问
2. 使用内存标记减少Redis的访问
3. 使用RabbitMQ队列缓冲，异步下单，增强用户体验

**实现步骤：**

1. 系统初始化，把商品库存数量加载到Redis上面来
2. 收到请求，Redis预减库存（先减少Redis里面的库存数量，库存不足，直接返回），如果库存已经到达临界值的时候，即=0，就不需要继续往下走，直接返回失败
3. 请求入队，立即返回排队中
4. 请求出队，生成订单，减少库存
5. 客户端轮询，是否秒杀成功

* **RabbitMQ异步下单**

1. 系统初始化，把商品库存数量stock加载到Redis上面来。
2. 后端收到秒杀请求，Redis预减库存，如果库存已经到达临界值的时候，就不需要继续请求下去，直接返回失败，即后面的大量请求无需给系统带来压力。
3. 判断这个秒杀订单形成没有，判断是否已经秒杀到了，避免一个账户秒杀多个商品，判断是否重复秒杀。
4. 库存充足，且无重复秒杀，将秒杀请求封装后消息入队，同时给前端返回一个code (0)，即代表返回排队中。（返回的并不是失败或者成功，此时还不能判断）
5. 前端接收到数据后，显示排队中，并根据商品id轮询请求服务器（考虑200ms轮询一次）。
6. 后端RabbitMQ监听秒杀MIAOSHA\_QUEUE的这名字的通道，如果有消息过来，获取到传入的信息，执行真正的秒杀之前，要判断数据库的库存，判断是否重复秒杀，然后执行秒杀事务（秒杀事务是一个原子操作：库存减1，下订单，写入秒杀订单）。
7. 此时，前端根据商品id轮询请求接口MiaoshaResult，查看是否生成了商品订单，如果请求返回-1代表秒杀失败，返回0代表排队中，返回>0代表商品id说明秒杀成功。

返回结果说明：

前端会根据后端返回的值来判断是秒杀结果。

  -1 ：库存不足秒杀失败

   0 ：排队中，继续轮询

   >0 ：返回的是商品id ，说明秒杀成功

# 应用安全策略及实现

* **秒杀接口地址隐藏**

每次点击秒杀按钮，才会生成秒杀地址，秒杀地址不是写死的，是从服务端获取，动态拼接而成的地址。（HTTP协议是明文传输，前端是防不住恶意用户的攻击，所以安全校验要放在服务端，从而禁止掉这些恶意攻击。）

思路：在进行秒杀之前，去后端获取一个动态的秒杀地址path（服务端生成随机数作为path），在然后将这个随机数返回给前端，前端用这个path拼接在新的请求url(url : “/miaosha/” + path + “/do\_miaosha”)上作为参数，再去发请求到后台开始我们的秒杀。

加上了秒杀接口地址隐藏之后可以防止恶意用户登陆之后，通过不断调用秒杀地址接口，骚扰服务器，所以使用动态获取秒杀地址，只有真正点击秒杀按钮，才会根据用户id和商品goodsId生成对应的秒杀接口地址。

* **数学图形验证码：**防止机器人，刷票软件恶意频繁点击按钮来刷请求秒杀地址接口的操作

思路：点击秒杀之前，先输入验证码，分散用户的请求。具体实现是服务端生成类似1+2-3的验证码，把结果计算出来存至服务端（缓存），把验证码图片发至客户端，此后客户端在请求秒杀地址前输入验证码值发请求验证，（去缓存里面取得值验证是否与用户输入相同），验证通过才会动态生成秒杀地址给前端。

* **接口地址限流防刷：**限制同一个用户一秒钟或者一分钟之内只能访问固定次数，在服务端对系统做一层保护

**思路：**利用缓存实现，用户每次点击之后访问接口的时候，在缓存中生成一个计数器，第一次将这个计数器置1后存入缓存，并给其设定有效期，比如一分钟，一分钟之内再访问，那么数值加一。一分钟之内访问次数超过限定数值，直接返回失败。下一个一分钟，数据重新从0开始计算。因为缓存具有一个有效期，一分钟之后自动失效。

**步骤：**

1. 获取访问路径
2. 拼接用户的Id作为一个记录该用户访问次数的key
3. 缓存里面取得该key，做判断；如果缓存里面没有取到，代表是第一次访问，所以给缓存设置该key，并设置初始值value为1；如果缓存里面取得值并且小于5，那么直接将该key对应的值value+1
4. 如果缓存里面的次数大于超过4（>=5），那么代表在限制时间内（在缓存还没有失效的时间内），访问次数达到限制

# 部分功能代码

* **秒杀倒计时**

@RequestMapping(value="/detail/{goodsId}") //produces="text/html"  
@ResponseBody  
public Result<GoodsDetailVo> toDetail\_staticPage(Model model**,** MiaoshaUser user**,** HttpServletRequest request**,** HttpServletResponse response**,** @PathVariable("goodsId")long goodsId) {//id一般用snowflake算法  
 System.*out*.println("页面静态化/detail/{goodsId}")**;** model.addAttribute("user"**,** user)**;** GoodsVo goodsVo=goodsService.getGoodsVoByGoodsId(goodsId)**;** model.addAttribute("goods"**,** goodsVo)**;** //既然是秒杀，还要传入秒杀开始时间，结束时间等信息  
 long start=goodsVo.getStartDate().getTime()**;** long end=goodsVo.getEndDate().getTime()**;** long now=System.*currentTimeMillis*()**;** //秒杀状态量  
 int status=**0;** //开始时间倒计时  
 int remailSeconds=**0;** //查看当前秒杀状态  
 if(now<start) {//秒杀还未开始，--->倒计时  
 status=**0;** remailSeconds=(int) ((start-now)/**1000**)**;** //毫秒转为秒  
 }else if(now>end){ //秒杀已经结束  
 status=**2;** remailSeconds=-**1;** //毫秒转为秒  
 }else {//秒杀正在进行  
 status=**1;** remailSeconds=**0;** //毫秒转为秒  
 }  
 model.addAttribute("status"**,** status)**;** model.addAttribute("remailSeconds"**,** remailSeconds)**;** GoodsDetailVo gdVo=new GoodsDetailVo()**;** gdVo.setGoodsVo(goodsVo)**;** gdVo.setStatus(status)**;** gdVo.setRemailSeconds(remailSeconds)**;** gdVo.setUser(user)**;** //将数据填进去，传至页面  
 return Result.*success*(gdVo)**;**}

<tr>  
 <td>秒杀开始时间</td>  
 <td th:text="${#dates.format(goods.startDate,'yyyy-MM-dd HH:mm:ss')}"></td>  
 <td id="miaoshaTip">  
 <!-- 先取得这个时间 -->  
 <input type="hidden" id="remailSeconds" th:value="${remailSeconds}"></input>  
   
 <span th:if="${status eq 0}">秒杀还未开始，倒计时:<span id="countDown" th:text="${remailSeconds}"></span>秒</span>   
 <span th:if="${status eq 1}">秒杀正在进行</span>   
 <span th:if="${status eq 2}">秒杀已经结束</span>   
 </td>  
 <td>  
 <form id="miaoshaForm" method="post" action="/miaosha/do\_miaosha">  
 <button class="btn btn-primary btn-block" type="submit" id="buyButton">立即秒杀</button>  
 <input type="hidden" name="goodsId" th:value="${goods.id}"></input>  
 </form>  
 </td>  
</tr>

* **MQSender**
* //发送者  
  @Service  
  public class MQSender {  
     
   private static Logger *log*=LoggerFactory.*getLogger*(MQSender.class)**;** @Autowired  
   RedisService redisService**;** @Autowired  
   AmqpTemplate amqpTemplate**;** */\*\*  
   \* 发送秒杀信息，使用derict模式的交换机。（包含秒杀用户信息，秒杀商品id）  
   \*/* public void sendMiaoshaMessage(MiaoshaMessage mmessage) {  
   // 将对象转换为字符串  
   String msg = RedisService.*beanToString*(mmessage)**;** *log*.info("send message:" + msg)**;** // 第一个参数队列的名字，第二个参数发出的信息  
   amqpTemplate.convertAndSend(MQConfig.*MIAOSHA\_QUEUE***,** msg)**;** }
* **MQReceiver**
* //接收者  
  @Service  
  public class MQReceiver {  
   @Autowired  
   GoodsService goodsService**;** @Autowired  
   RedisService redisService**;** @Autowired  
   MiaoshaUserService miaoshaUserService**;** //作为秒杀功能事务的Service  
   @Autowired  
   MiaoshaService miaoshaService**;** @Autowired  
   OrderService orderService**;** private static Logger *log*=LoggerFactory.*getLogger*(MQReceiver.class)**;** @RabbitListener(queues=MQConfig.*MIAOSHA\_QUEUE*)//指明监听的是哪一个queue  
   public void receiveMiaosha(String message) {  
   *log*.info("receiveMiaosha message:"+message)**;** //通过string类型的message还原成bean  
   //拿到了秒杀信息之后。开始业务逻辑秒杀，  
   MiaoshaMessage mm=RedisService.*stringToBean*(message**,** MiaoshaMessage.class)**;** MiaoshaUser user=mm.getUser()**;** long goodsId=mm.getGoodsId()**;** GoodsVo goodsvo=goodsService.getGoodsVoByGoodsId(goodsId)**;** int stockcount=goodsvo.getStockCount()**;** //1.判断库存不足  
   if(stockcount<=**0**) {//失败 库存至临界值1的时候，此时刚好来了加入10个线程，那么库存就会-10  
   //model.addAttribute("errorMessage", CodeMsg.MIAOSHA\_OVER\_ERROR);  
   return**;** }  
   //2.判断这个秒杀订单形成没有，判断是否已经秒杀到了，避免一个账户秒杀多个商品  
   MiaoshaOrder order = orderService.getMiaoshaOrderByUserIdAndCoodsId(user.getId()**,** goodsId)**;** if (order != null) {// 重复下单  
   // model.addAttribute("errorMessage", CodeMsg.REPEATE\_MIAOSHA);  
   return**;** }  
   //原子操作：1.库存减1，2.下订单，3.写入秒杀订单--->是一个事务  
   //miaoshaService.miaosha(user,goodsvo);  
   miaoshaService.miaosha1(user**,**goodsvo)**;** }
* **秒杀业务逻辑**
* @RequestMapping(value="/{path}/do\_miaosha\_ajaxcache"**,**method=RequestMethod.*POST*)  
   @ResponseBody  
   public Result<Integer> doMiaoshaCache(Model model**,**MiaoshaUser user**,** @RequestParam(value="goodsId"**,**defaultValue="0") long goodsId**,** @PathVariable("path")String path) {  
   model.addAttribute("user"**,** user)**;** //1.如果用户为空，则返回至登录页面  
   if(user==null){  
   return Result.*error*(CodeMsg.*SESSION\_ERROR*)**;** }  
   //验证path,去redis里面取出来然后验证。  
   boolean check=miaoshaService.checkPath(user**,**goodsId**,**path)**;** if(!check) {  
   return Result.*error*(CodeMsg.*REQUEST\_ILLEAGAL*)**;** }  
   //内存标记，减少对redis的访问 localMap.put(goodsId,false);  
  // boolean over=localMap.get(goodsId);  
  // //在容量满的时候，那么就打标记为true  
  // if(over) {  
  // return Result.error(CodeMsg.MIAOSHA\_OVER\_ERROR);  
  // }  
   //2.预减少库存，减少redis里面的库存  
   long stock=redisService.decr(GoodsKey.*getMiaoshaGoodsStock***,**""+goodsId)**;** //3.判断减少数量1之后的stock，区别于查数据库时候的stock<=0  
   if(stock<**0**) {  
   return Result.*error*(CodeMsg.*MIAOSHA\_OVER\_ERROR*)**;** }  
   //4.判断这个秒杀订单形成没有，判断是否已经秒杀到了，避免一个账户秒杀多个商品  
   MiaoshaOrder order = orderService.getMiaoshaOrderByUserIdAndCoodsId(user.getId()**,** goodsId)**;** if (order != null) {// 重复下单  
   // model.addAttribute("errorMessage", CodeMsg.REPEATE\_MIAOSHA);  
   return Result.*error*(CodeMsg.*REPEATE\_MIAOSHA*)**;** }  
   //5.正常请求，入队，发送一个秒杀message到队列里面去，入队之后客户端应该进行轮询。  
   MiaoshaMessage mms=new MiaoshaMessage()**;** mms.setUser(user)**;** mms.setGoodsId(goodsId)**;** mQSender.sendMiaoshaMessage(mms)**;** //返回0代表排队中  
   return Result.*success*(**0**)**;** }
* **商品库存数量预加载到redis上**

public void afterPropertiesSet() throws Exception {  
 List<GoodsVo> goodslist=goodsService.getGoodsVoList()**;** if(goodslist==null) {  
 return**;** }  
 for(GoodsVo goods:goodslist) {  
 //如果不是null的时候，将库存加载到redis里面去 prefix---GoodsKey:gs , key---商品id, value  
 redisService.set(GoodsKey.*getMiaoshaGoodsStock***,** ""+goods.getId()**,** goods.getStockCount())**;** }  
}

@Autowired  
GoodsDao goodsDao**;**@Autowired  
RedisService redisService**;**public List<GoodsVo> getGoodsVoList() {  
 return goodsDao.getGoodsVoList()**;**}

//两个查询  
@Select("select g.\*,mg.stock\_count,mg.start\_date,mg.end\_date,mg.miaosha\_price from miaosha\_goods mg left join goods g on mg.goods\_id=g.id")   
public List<GoodsVo> getGoodsVoList()**;**

* **数字图形验证码**
* */\*\*  
   \* 生成图片验证码  
   \*/*@RequestMapping(value ="/vertifyCode")  
  @ResponseBody  
  public Result<String> getVertifyCode(Model model**,** MiaoshaUser user**,** @RequestParam("goodsId") Long goodsId**,** HttpServletResponse response) {  
   model.addAttribute("user"**,** user)**;** //如果用户为空，则返回至登录页面  
   if(user==null){  
   return Result.*error*(CodeMsg.*SESSION\_ERROR*)**;** }  
   BufferedImage img=miaoshaService.createMiaoshaVertifyCode(user**,** goodsId)**;** try {  
   OutputStream out=response.getOutputStream()**;** ImageIO.*write*(img**,**"JPEG"**,** out)**;** out.flush()**;** out.close()**;** return null**;** } catch (IOException e) {  
   e.printStackTrace()**;** return Result.*error*(CodeMsg.*MIAOSHA\_FAIL*)**;** }  
  }  
    
  */\*\*  
   \* 获取秒杀的path,并且验证验证码的值是否正确  
   \*/*//@AccessLimit(seconds=5,maxCount=5,needLogin=true)  
  //加入注解，实现拦截功能，进而实现限流功能  
  //@AccessLimit(seconds=5,maxCount=5,needLogin=true)  
  @RequestMapping(value ="/getPath")  
  @ResponseBody  
  public Result<String> getMiaoshaPath(HttpServletRequest request**,**Model model**,**MiaoshaUser user**,** @RequestParam("goodsId") Long goodsId**,** @RequestParam(value="vertifyCode"**,**defaultValue="0") int vertifyCode) {  
   model.addAttribute("user"**,** user)**;** //如果用户为空，则返回至登录页面  
   if(user==null){  
   return Result.*error*(CodeMsg.*SESSION\_ERROR*)**;** }  
   //限制访问次数  
   String uri=request.getRequestURI()**;** String key=uri+"\_"+user.getId()**;** //限定key5s之内只能访问5次  
   Integer count=redisService.get(AccessKey.*access***,** key**,** Integer.class)**;** if(count==null) {  
   redisService.set(AccessKey.*access***,** key**, 1**)**;** }else if(count<**5**) {  
   redisService.incr(AccessKey.*access***,** key)**;** }else {//超过5次  
   return Result.*error*(CodeMsg.*ACCESS\_LIMIT*)**;** }  
     
   //验证验证码  
   boolean check=miaoshaService.checkVCode(user**,** goodsId**,**vertifyCode )**;** if(!check) {  
   return Result.*error*(CodeMsg.*REQUEST\_ILLEAGAL*)**;** }  
   System.*out*.println("通过!")**;** //生成一个随机串  
   String path=miaoshaService.createMiaoshaPath(user**,**goodsId)**;** System.*out*.println("@MiaoshaController-tomiaoshaPath-path:"+path)**;** return Result.*success*(path)**;**}

# 仓库地址

https://github.com/bjtuql/miaosha