

**SecKill讲义**

# 秒杀项目介绍

## 需求

* 有很多款商品，每款商品的库存数量有限（比如1000件）
* 秒杀活动开始后方可购买，秒杀活动未开始和秒杀活动已结束，都不能购买
* 同一款商品每个人只能参与秒杀抢购一次
* 每人下单时默认购买一件商品

## 使用技术

* + 前端：thymeleaf、js、css
  + 后端：Spring Boot、SpringCloud、MyBatis
  + 服务器/中间件：zookeeper、redis、activemq、mysql

## 开发模式

从前向后：页面->Controllor->Service->Mapper（我们使用）

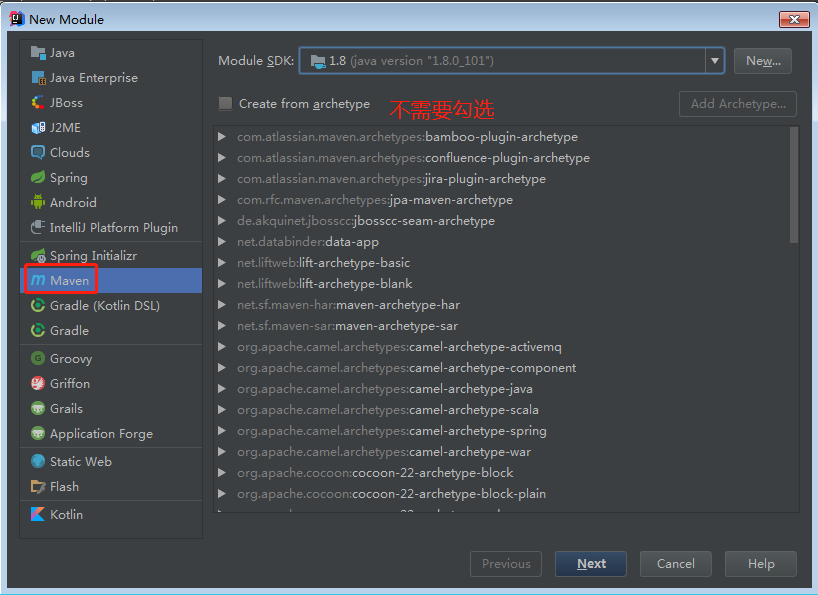
从后向前：Mapper->Service->Controller->页面

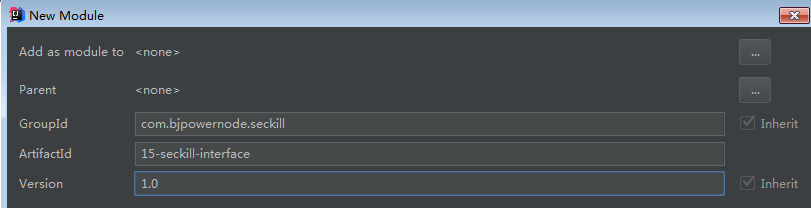
# 基本环境搭建

## 开发环境搭建

### 创建普通的Maven模块15-seckill-interface

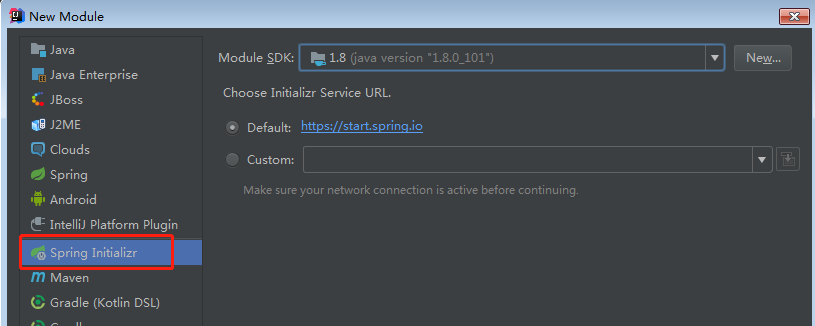
该模块下主要放model，接口及一些工具类

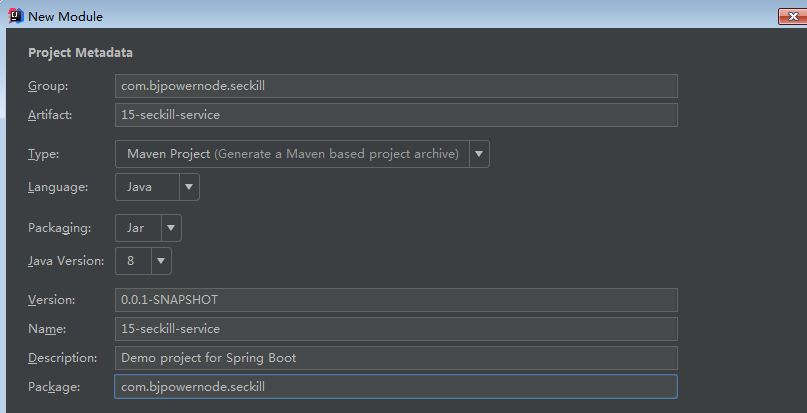


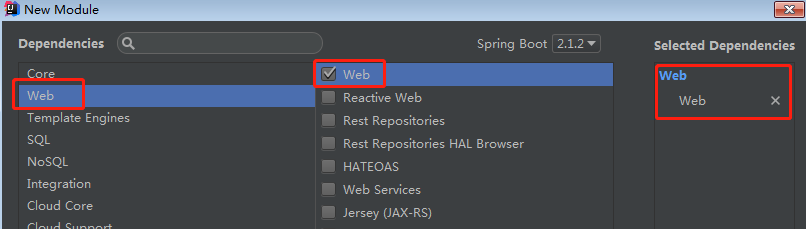


### 创建SpringBoot模块15-seckill-service，集成Web依赖

该模块主要进行业务的实现





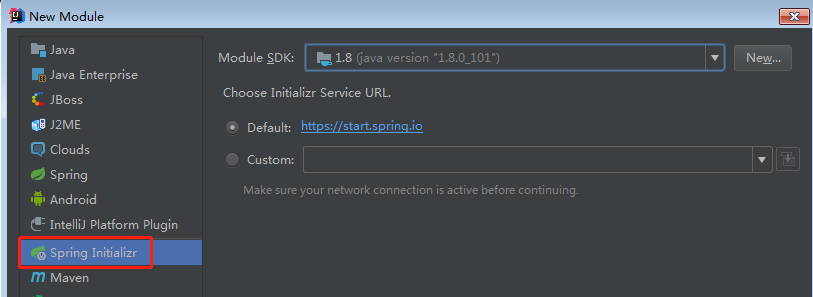


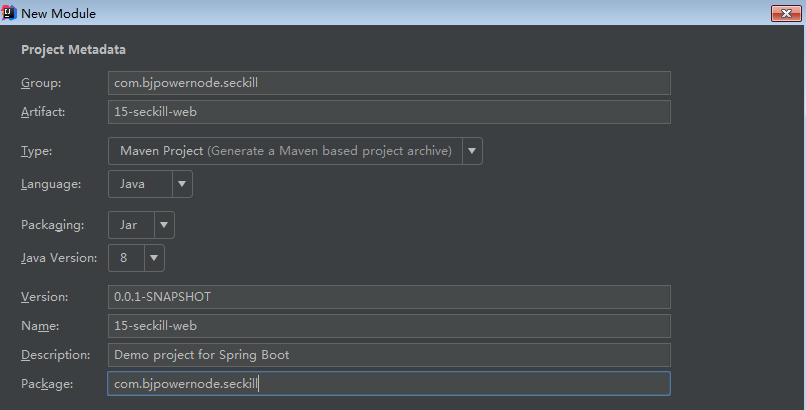
* **指定项目上下文根为/15-seckill-server，端口号为9090**

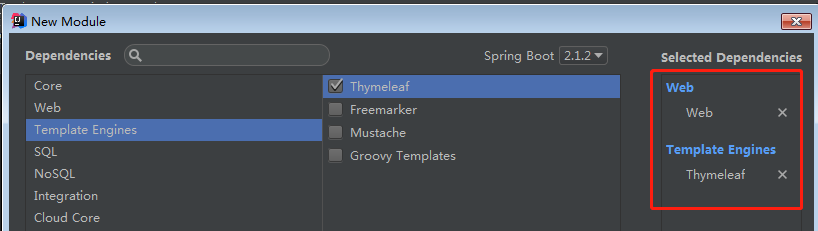
server.servlet.context-path=/15-seckill-service  
server.port=9090

### 创建SpringBoot模块15-seckill-web，集成Web和Thymeleaf依赖

该模块主要做前端的展示及处理前端请求







* **指定项目上下文根为/15-seckill-web，端口号为8080**

server.servlet.context-path=/15-seckill-web  
server.port=8080

# 开发阶段缓存建议关闭  
spring.thymeleaf.cache=false

* **在pom.xml文件中添加热部署插件**

<!--热部署插件-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
 <optional>true</optional>  
</dependency>

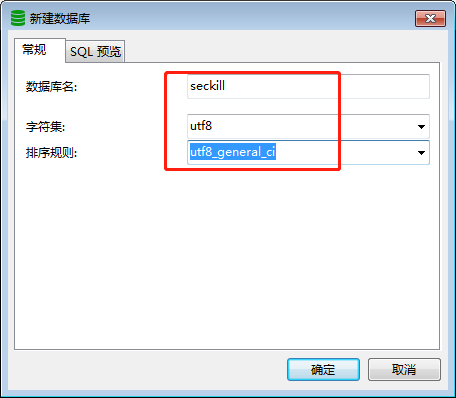
## 数据库环境搭建

连接Linux3306数据库

### 启动Linux上MySQL数据库服务

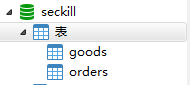


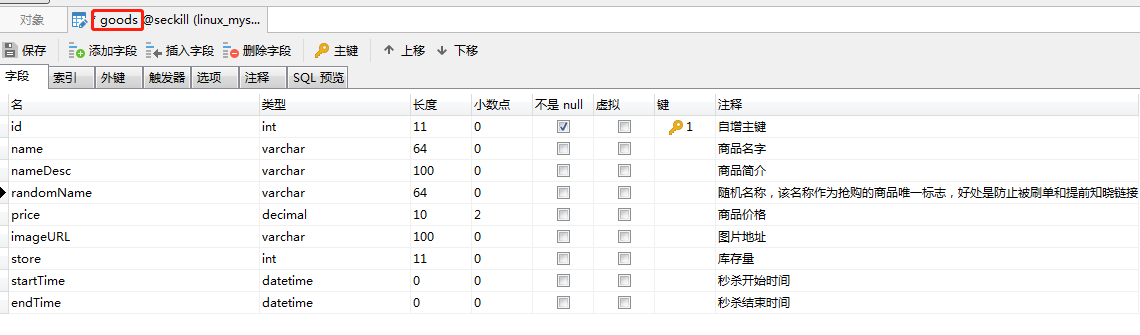
### 创建新库seckill（使用Navicat连接MySQL）



### 创建商品表和订单表

将07-SecKill\resources目录下的建表语句（goods.sql和orders.sql）拷贝到Navicat中执行





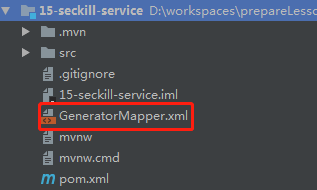


## 使用Mybatis反向工程生成数据

### 在15-seckill-service中的pom.xml文件中添加Mybatis反向工程插件

<!--mybatis代码自动生成插件-->  
<plugin>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-maven-plugin</artifactId>  
 <version>1.3.7</version>  
 <configuration>  
 <!--配置文件的位置-->  
 <configurationFile>GeneratorMapper.xml</configurationFile>  
 <verbose>true</verbose>  
 <overwrite>true</overwrite>  
 </configuration>  
</plugin>

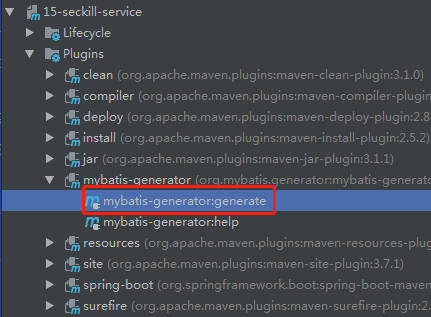
### 将反向工程配置文件拷贝到15-seckill-service模块中



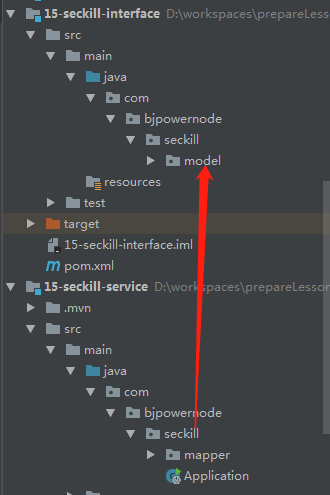
### 修改反向工程配置文件GeneratorMapper.xml

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE generatorConfiguration  
 PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  
 "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  
  
<generatorConfiguration>  
  
 <!-- 指定连接数据库的JDBC驱动包所在位置，指定到你本机的完整路径 -->  
 <classPathEntry location="D:/repository/mysql/mysql-connector-java/8.0.13/mysql-connector-java-8.0.13.jar"/>  
  
 <!-- 配置table表信息内容体，targetRuntime指定采用MyBatis3的版本 -->  
 <context id="tables" targetRuntime="MyBatis3">  
  
 <!-- 抑制生成注释，由于生成的注释都是英文的，可以不让它生成 -->  
 <commentGenerator>  
 <property name="suppressAllComments" value="true" />  
 </commentGenerator>  
  
 <!-- 配置数据库连接信息 注意：使用高版本的驱动 url后面应该加属性nullCatalogMeansCurrent=true，否则生成有问题 -->  
 <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"  
 connectionURL="jdbc:mysql://192.168.235.128:3306/seckill?nullCatalogMeansCurrent=true"  
 userId="root"  
 password="123456">  
 </jdbcConnection>  
  
 <!-- 生成model类，targetPackage指定model类的包名， targetProject指定生成的model放在eclipse的哪个工程下面-->  
 <javaModelGenerator targetPackage="com.bjpowernode.seckill.model" targetProject="src/main/java">  
 <property name="enableSubPackages" value="false" />  
 <property name="trimStrings" value="false" />  
 </javaModelGenerator>  
  
 <!-- 生成MyBatis的Mapper.xml文件，targetPackage指定mapper.xml文件的包名， targetProject指定生成的mapper.xml放在eclipse的哪个工程下面 -->  
 <sqlMapGenerator targetPackage="com.bjpowernode.seckill.mapper" targetProject="src/main/java">  
 <property name="enableSubPackages" value="false" />  
 </sqlMapGenerator>  
  
 <!-- 生成MyBatis的Mapper接口类文件,targetPackage指定Mapper接口类的包名， targetProject指定生成的Mapper接口放在eclipse的哪个工程下面 -->  
 <javaClientGenerator type="XMLMAPPER" targetPackage="com.bjpowernode.seckill.mapper" targetProject="src/main/java">  
 <property name="enableSubPackages" value="false" />  
 </javaClientGenerator>  
  
 <!-- 数据库表名及对应的Java模型类名 -->  
 <table tableName="goods"  
 domainObjectName="Goods"  
 enableCountByExample="false"  
 enableUpdateByExample="false"  
 enableDeleteByExample="false"  
 enableSelectByExample="false"  
 selectByExampleQueryId="false"/>  
 <table tableName="orders"  
 domainObjectName="Orders"  
 enableCountByExample="false"  
 enableUpdateByExample="false"  
 enableDeleteByExample="false"  
 enableSelectByExample="false"  
 selectByExampleQueryId="false"/>  
 </context>  
</generatorConfiguration>

### 生成model及mapper



### 将生成的model拷贝到15-seckill-interface模块下，然后删除掉15-seckill-service下的model



### 在15-seckill-web和15-seckill-service的pom.xml文件中添加15-seckill-interface接口依赖

<!--秒杀接口依赖-->  
<dependency>  
 <groupId>com.bjpowernode.seckill</groupId>  
 <artifactId>15-seckill-interface</artifactId>  
 <version>1.0</version>  
</dependency>

# 商品列表展示模块

## 在15-seckill-web 的pom.xml文件添加Thymeleaf依赖

我们创建SpringBoot项目的时候，已经勾选，会自动生成

## 在15-seckill-web的resources/templates下创建goods.html，用于商品列表展示

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>秒杀商品列表页</title>  
</head>  
<body th:inline="text">  
 <div th:each="goods:${goodsList}">  
 <img th:src="@{${goods.imageurl}}"><br>  
 [[${goods.name}]] [[${goods.namedesc}]]<br>  
 [[${goods.price}]] 剩余：[[${goods.store}]]件  
 </div>  
</body>  
</html>

## 后端获取所有商品列表传递给goods.html页面

### 分析1：直接从数据库中查询获取所有商品合适吗？

* 查询所有秒杀商品，商品在数据库中存储
* 因为是秒杀场景，属于高并发大流量访问
* 如果我们在Controller层中直接查询数据库，会给数据库造成很大的压力，甚至会让数据库没有响应或者崩溃
* **优化：所以，我们这里不直接查询数据库，而是先将所有商品查询出来，放到Redis中，采用Redis抗住巨大的访问流量，这种提前将数据查询出来放到Redis的操作叫做“缓存预热”**

### 分析2：从Redis取数据写在 15-seckill-web项目中还是在15-seckill- service项目中

* 直接写在15-seckill-web项目中，查询Redis获取商品列表
* 如果写在15-seckill-service项目中，需要采用RPC调用，会通过网络进行Socket连接，会有性能消耗
* **这也是一个优化**

### 分析3：什么时候向Redis中写入商品列表

在服务器启动之后，不间断的从数据库中查询商品信息，写入到Redis中

### 分析4：通过什么方式，不间断的查询数据库商品信息，写到Redis中

可以在15-seckill-service中创建一个定时任务，通过定时任务实现

### 在15-seckill-service中完成定时查询数据库向Redis放商品

#### 在15-seckill-service的pom.xml文件中添加SpringBoot对Redis的支持依赖

<!-- 加载spring boot redis包 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

#### 在15-seckill-service的SpringBoot核心配置文件application.properties中添加Redis连接信息

#配置redis连接信息  
spring.redis.host=192.168.235.128  
spring.redis.port=6379  
spring.redis.password=123456

#### 向15-seckill-service的pom.xml文件中添加SpringBoot整合Mybatis依赖和数据库驱动依赖

<!-- 加载mybatis整合springboot -->  
<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <!--在springboot的父工程中没有指定版本，我们需要手动指定-->  
 <version>1.3.2</version>  
</dependency>  
<!-- MySQL的jdbc驱动包 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <!--在springboot的父工程中指定了版本，我们就不需要手动指定了-->  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>

#### 在15-seckill-service的pom.xml文件中指定将Mybatis映射文件编译到classpath下

<resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
</resource>

#### 在15-seckill-service的核心配置文件application.properties中配置MySQL数据库连接信息

#数据库的连接配置信息  
spring.datasource.username=root  
spring.datasource.password=123456  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.datasource.url=jdbc:mysql://192.168.235.128:3306/seckill?useUnicode=true&characterEncoding=utf8&useSSL=false

#### 在15-seckill-service的com.bjpowernode.seckill.task包下创建定时任务类RedisTask，缓存商品信息到Redis中

Redis Key的格式 redis:store:商品id Value的值：商品对象的json格式

@Configuration  
@EnableScheduling  
public class RedisTask {  
 @Autowired  
 private GoodsMapper goodsMapper;  
 @Autowired  
 private RedisTemplate<String,String> redisTemplate;  
 */\*\*  
 \* 每5秒执行一次定时任务，初始化一遍秒杀商品信息  
 \* 缓存预热  
 \*/* @Scheduled(cron = "0/5 \* \* \* \* \*")  
 public void initRedisGoods(){

System.*out*.println("缓存预热...........");  
 //查询所有秒杀商品数据  
 List<Goods> goodsList = goodsMapper.selectAllGoods();  
 //放入到Redis中  
 for(Goods goods:goodsList){  
 //因为后续还需要查询商品的详情，如果将整个List放进去，后续查询详情麻烦些  
 String goodsJSON = JSONObject.*toJSONString*(goods);  
 redisTemplate.opsForValue().set(Constants.*REDIS\_GOODS* +goods.getId(),goodsJSON);  
 }  
 }  
}

#### 在15-seckill-service中GoodsMapper接口中添加selectAllGoods方法

*/\*\*  
 \* 查询所有商品  
 \** ***@return*** *\*/*List<Goods> selectAllGoods();

#### 在GoodsMapper和OrderMapper接口上加@Mapper注解

@Mapper  
public interface GoodsMapper {

**OrderMapper我们也提前加上，避免后面忘了**

#### 在15-seckill-service中GoodsMappe.xml中实现selectAllGoods

目前我们表中所有商品都是秒杀商品，实际项目中，如果还有非秒杀商品，需要进行过滤

<select id="selectAllGoods" resultMap="BaseResultMap">  
 select  
 <include refid="Base\_Column\_List" />  
 from goods  
</select>

#### 在15-seckill-service的pom.xml文件中添加对fastjson的支持

为了操作商品数据方便，后续我们需要改商品库存，我们向Redis中放数据的时候，是以json字符串的形式存放的，所以导入对json的支持依赖

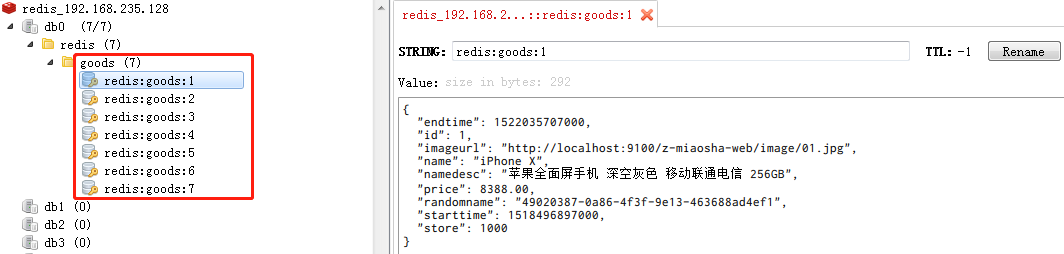
<!--fastjson对json处理的依赖-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.36</version>  
</dependency>

#### 在15-seckill-interface中的com.bjpowernode.seckill.constants包下定义常量类Constants，存放秒杀系统的常量

public class Constants {  
 */\*\*  
 \* 定义Redis中商品信息的key的前缀  
 \* Redis中存放商品的格式：redis:goods:商品id  
 \*/* public static final String *REDIS\_GOODS* = "redis:goods:";  
}

#### 运行15-seckill-service的Application，单独对定时任务进行测试

通过RedisDestopManager客户端查看效果



### 在15-seckill-web中处理页面的请求，从Redis中取商品数据响应给前端页面

#### 在15-seckill-web的pom.xml文件中添加SpringBoot对Redis的支持依赖

<!-- 加载spring boot redis包 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

#### 在15-seckill-web的核心配置文件application.properties中配置Redis连接信息

#配置redis连接信息  
spring.redis.host=192.168.235.128  
spring.redis.port=6379  
spring.redis.password=123456

#### 在15-seckill-web的pom.xml文件中添加对fastjson的支持

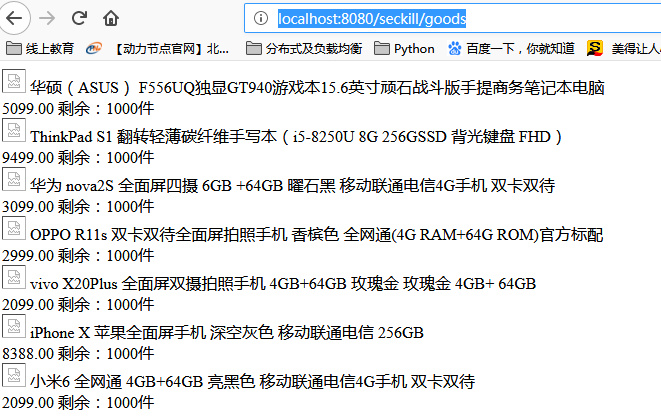
因为缓存预热向Redis中存放的商品是以json的形式存放的，所以我们这里要将json转为商品对象

<!--fastjson对json处理的依赖-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.36</version>  
</dependency>

#### 在15-seckill-web 的com.bjpowernode.seckill.controller包下创建GoodsController类

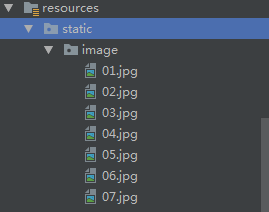
@Controller  
public class GoodsController {  
 @Autowired  
 private RedisTemplate<String,String> redisTemplate;  
 @GetMapping("/seckill/goods")  
 public String goods(Model model){  
 //从Redis获取商品集合  
 //获取所有存在Redis中的商品的key  
 Set<String> keys = redisTemplate.keys(Constants.*REDIS\_GOODS* +"\*");  
 List<String> goodsJSONList = redisTemplate.opsForValue().multiGet(keys);  
 List<Goods> goodsList = new ArrayList<Goods>();  
 for (String goodsJSON : goodsJSONList) {  
 Goods goods = JSONObject.*parseObject*(goodsJSON,Goods.class);  
 goodsList.add(goods);  
 }  
 model.addAttribute("goodsList",goodsList);  
 return "goods";  
 }  
}

### 运行15-seckill-web的Application，浏览器输入<http://localhost:8080/seckill/goods>查看效果



### 数据可以正常显示，但是图片显示不出来问题解决

* 确认数据库商品表中的图片路径是否和当前项目上下文及端口一致
* 将07-SecKill\resources下的image拷贝到15-seckill-web模块的static目录下



### 修改goods.html页面样式

* 商品div的宽度过宽：width: 285px;
* 让商品排列展开：float:left;
* 每个商品间设置内边距:margin:18px;
* 整个页面设置内边距: margin:18px;
* 每行展示4个商品：

th:style="${goodsStat.count % 5 eq 0}?'clear:both; float: left;width: 285px;':'width: 285px;float: left;'"

* 商品价钱调大，颜色为红色

<span style="color: red;font-size:22px;font-weight: bold;">[[${goods.price}]]</span>

* 给商品图片和商品名称加超链接，点击跳到详情页，并在新窗口打开

加超链接，并指定target=”\_black”

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>秒杀商品列表页</title>  
</head>  
<body th:inline="text" style="margin: 20px;">  
 <div th:each="goods:${goodsList}"   
 th:style="${goodsStat.count % 5 eq 0}?  
 'clear:both; float: left;margin:20px;width: 285px;':'width: 285px;margin:20px;float: left;'">  
   
 <a th:href="@{'/seckill/goods/' + ${goods.getId()}}" target="\_blank">  
 <img th:src="@{${goods.imageurl}}"><br>  
 </a>  
   
 <a th:href="@{'/seckill/goods/' + ${goods.getId()}}" target="\_blank">  
 [[${goods.name}]] [[${goods.namedesc}]]<br>  
 </a>  
   
 <span style="color: red;font-size:22px;font-weight: bold;">[[${goods.price}]]</span>   
 剩余：[[${goods.store}]]件  
 </div>  
</body>  
</html>

# 商品详情页展示模块

## 在15-seckill-web中的GoodsController中添加跳转详情页的方法，并将商品信息传递给前端页面

@GetMapping("/seckill/goods/{id}")  
public String item(@PathVariable("id") Integer id,Model model){  
 //根据商品id，从Redis中获取商品Json字符串  
 String goodsJSON = redisTemplate.opsForValue().get(Constants.*REDIS\_GOODS* + id);  
 Goods goods = JSONObject.*parseObject*(goodsJSON,Goods.class);  
 model.addAttribute("goods",goods);  
 //因为在详情页需要显示秒杀按钮，所有我们将服务器当前时间传递给详情页  
 model.addAttribute("currentTime",System.*currentTimeMillis*());  
 return "item";  
}

## 在15-seckill-web中templates下定义商品详情页item.html

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>秒杀商品详情页</title>  
</head>  
<body th:inline="text" style="margin: 50px;">  
 <!--左侧div-->  
 <div style="float: left">  
 <a th:href="@{'/seckill/goods/' + ${goods.getId()}}" target="\_blank">  
 <img th:src="@{${goods.imageurl}}"><br>  
 </a>  
 </div>  
 <!--右侧div-->  
 <div style="float: left">  
 <p>  
 <a th:href="@{'/seckill/goods/' + ${goods.getId()}}" target="\_blank">  
 [[${goods.name}]] [[${goods.namedesc}]]<br>  
 </a>  
 </p>  
 <p>  
 <span style="color: red;font-size:22px;font-weight: bold;">[[${goods.price}]]</span>  
 </p>  
 <p>  
 剩余：[[${goods.store}]]件  
 </p>  
 </div>  
</body>  
</html>

# 商品秒杀按钮的显示模块

## 功能分析

### 前台秒杀商品详情页面

* 秒杀尚未开始，显示秒杀倒计时
* 正在秒杀的商品，显示按钮
* 秒杀时间已过，显示秒杀已结束
* 以上判断逻辑需要在js中完成，为了页面逻辑看起来清晰，我们创建一个单独的js文件seckill.js来处理，在详情页面中引入该js
* 在页面中我们发送请求使用jQuery，所以需要引入jQuery的js文件
* 倒计时的显示，我们使用jQuery的countdown插件，所以需要将该插件的js也引入到详情页面中

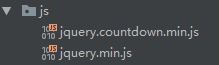
### 后台秒杀地址的暴露

* 如果前台已经开始秒杀，那么需要在后台暴露秒杀地址给前台页面
* 为了安全，在后台依旧需要对秒杀是否开始进行判断
* 为了前台处理方便，我们将后台返回的结果封装为自定义RTO（结果传输对象）

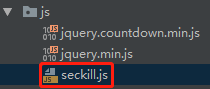
## 业务实现

### 在15-seckill-web模块的static目录下，创建js子目录，将jQuery和倒计时插件从07-SecKill\resources\js拷贝进来

倒计时插件的用法在07-SecKill\resources\countDown.txt中，直接拷贝过来用即可



### 在15-seckill-web模块的static/ js目录下，创建seckill.js文件



### 在15-seckill-web模块的item.html页面中，导入上面定义三个js文件

<!--导入jQuery的js文件-->  
<script th:src="@{/js/jquery.min.js}"></script>  
<!--导入倒计时插件的js文件-->  
<script th:src="@{/js/jquery.countdown.min.js}"></script>  
<!--导入自定义的秒杀js文件-->  
<script th:src="@{/js/seckill.js}"></script>

### 在15-seckill-web中的item.html页面调用秒杀初始化的方法，并传递参数

<script type="text/javascript" th:inline="javascript">  
 //页面加载完成之后，调用秒杀初始化方法，对时间进行判断  
 $(**function**(){  
 seckillObj.contextPath = [[${#request.getContextPath()}]];  
 **var** id = [[${goods.id}]];  
 //当前时间应该从服务器获取，在跳转到秒杀详情页的时候传递过来  
 **var** currentTime = [[${currentTime}]];  
 //时间为距1970年1月1日 00:00:00的毫秒数  
 **var** startTime = [[${goods.starttime.getTime()}]];  
 **var** endTime = [[${goods.endtime.getTime()}]];  
 seckillObj.func.initItem(id,currentTime,startTime,endTime);  
 });  
</script>

### 在15-seckill-web模块的seckill.js中定义对秒杀时间的判断函数

JS的开发方式，有两种

面向过程

面向对象，这里我们采用面向对象的方式

//定义一个json对象  
**var** seckillObj = {  
 //项目上下文根，我们可以在页面中通过seckillObj对象对其赋值  
 contextPath:"",  
 //这样操作可选，一般公司会将获取地址单独抽取出来，方便维护  
 url:{  
 randomURL:**function** () {  
 **return** seckillObj.contextPath +"/seckill/random/";  
 }  
 },  
 //定义秒杀时间判断的相关函数，本身还是json的属性，只不过是function类型  
 func:{  
 //初始化函数主要用于时间的判断,这些时间可以从item页面的秒杀商品对象上获取  
 //所以该方法提供参数，接收页面传递的信息，同时将商品的id也传递过来  
 initItem:**function** (id,currentTime,startTime,endTime) {  
 **if**(currentTime < startTime){  
 //秒杀尚未开始 使用jquery的倒计时插件实现倒计时  
 /\* + 1000 防止时间偏移 这个没有太多意义，因为我们并不知道客户端和服务器的时间偏移  
 这个插件简单了解，实际项目不会以客户端时间作为倒计时，一般在服务器端还需要验证\*/  
 **var** killTime = **new** Date(startTime + 1000);  
 $("#seckillTip").countdown(killTime, **function** (event) {  
 //时间格式  
 **var** format = event.strftime('距秒杀开始还有: %D天 %H时 %M分 %S秒');  
 $("#seckillTip").html("<span style='color:red;'>"+format+"</span>");  
 }).on('finish.countdown', **function** () {  
 //倒计时结束后回调事件，已经开始秒杀，用户可以进行秒杀了，有两种方式：  
 //1、刷新当前页面  
 location.reload();  
 //或者2、调用秒杀开始的函数  
 });  
 }**else if**(currentTime > endTime){  
 //秒杀已经结束  
 $("#seckillTip").html("<span style='color:red;'>来晚了，秒杀活动已结束</span>");  
 }**else**{  
 //秒杀已开始 调用startSeckill函数  
 seckillObj.func.startSeckill(id);  
 }  
 },  
 startSeckill:**function** (id) {  
 //秒杀术语 暴露秒杀地址 http://localhost:8080/15-seckill-web/seckill/goods/dfasjfkdjfkldajs  
 //发送ajax请求 去后台服务器判断秒杀是否已经开始，如果真正开始，暴露秒杀地址，否则不暴露  
 $.ajax({  
 url : seckillObj.url.randomURL() + id,  
 type : "post",  
 dataType:"json",  
 success:**function**(rtnMessage){  
 **if**(rtnMessage.errorCode == 0){  
 //不能显示秒杀按钮  
 $("#seckillTip").html("<span style='color:red;'>"+ rtnMessage.errorMessage +"</span>");  
 }**else**{  
 //显示秒杀按钮  
 $("#seckillTip").html("<button type='button'>立即秒杀</button>");  
 }  
 }  
 });  
 }  
 }  
}

### 在15-seckill-web中的item.html页面加显示倒计时及秒杀提示信息的段落标签

<p id="seckillTip">  
</p>

### 数据库时区问题的解决

我们在使用倒计时插件的，发现获取的数据库时间不对，因为我们15-seckill-web的数据是从Redis中获取的，Redis的数据是15-seckill-service中的定时任务获取的，所以我们需要修改15-seckill-service模块中的核心配置文件的数据库连接信息，**加serverTimezone=GMT%2B8，**指定时区，这个问题从SpringBoot2.1.0以后出现，以前的版本没有。

修改完毕后重新运行15-seckill-service获取时间。

### 在15-seckill-web中的GoodsController处理暴露地址的请求

@PostMapping("/seckill/random/{id}")  
public @ResponseBody ReturnObject random(@PathVariable("id") Integer id){  
 ReturnObject returnObject = new ReturnObject();  
 //根据商品的id获取商品对象  
 String goodsJSON = redisTemplate.opsForValue().get(Constants.*REDIS\_GOODS* + id);  
 Goods goods = JSONObject.*parseObject*(goodsJSON,Goods.class);  
 //验证秒杀时间是否已经真的开始  
 Long currentTime = System.*currentTimeMillis*();  
 Long startTime = goods.getStarttime().getTime();  
 Long endTime = goods.getEndtime().getTime();  
 if(currentTime < startTime){  
 //秒杀尚未开始  
 returnObject.setErrorCode(Constants. *ZERO*);  
 returnObject.setErrorMessage("秒杀尚未开始");  
 }else if(currentTime > endTime){  
 //秒杀已经结束  
 returnObject.setErrorCode(Constants. *ZERO*);  
 returnObject.setErrorMessage("秒杀已经结束");  
 }else{  
 //秒杀已经开始  
 returnObject.setErrorCode(Constants. *ONE*);  
 returnObject.setErrorMessage("秒杀已开始");  
 returnObject.setData(goods.getRandomname());  
 }  
 return returnObject;  
}

### 在15-seckill-interface中的com.bjpowernode.seckill.rto包下ReturnObject类，用于封装返回结果对象

public class ReturnObject {  
 private int errorCode;  
 private String errorMessage;  
 private Object data;  
 //get|set方法略  
}

### 在15-seckill-interface中的常量类Constants中定义返回的错误码常量 1成功 0失败

//返回结果码的常量，0失败， 1 成功  
public static final int *ZERO* = 0;  
public static final int *ONE* = 1;

### 启动15-seckill-service和15-seckill-web，修改数据库商品表数据查看效果

# 请求执行秒杀模块

## 前台页面操作

### 在seckill.js中给立即秒杀按钮加一个id，绑定单击事件，单击秒杀按钮后，页面需要将按钮置为不可用，防止重复提交

//显示秒杀按钮  
$("#seckillTip").html("<button type='button' id='seckillBtn'>立即秒杀</button>");  
$("#seckillBtn").click(**function** () {  
 //页面上防止重复点击的控制 减少对后台服务器的请求  
 $("#seckillBtn").attr("disabled",**true**);  
 //调用执行秒杀请求的函数 传递商品的随机名和商品id  
 **var** random = rtnMessage.data;  
 seckillObj.func.execSeckill(random,id)  
});

### seckill.js中，立即秒杀按钮单击后发送ajax请求，执行秒杀操作，单独封装一个函数execSeckill处理，秒杀的结果暂不处理

* 在seckill.js的seckillObj的url属性上添加seckillURL

url:{  
 randomURL:**function** () {  
 **return** seckillObj.contextPath +"/seckill/random/";  
 },  
 seckillURL:**function**(){  
 **return** seckillObj.contextPath +"/seckill/gooods/";  
 }  
},

* execSeckill函数

//执行秒杀请求  
execSeckill:**function** (random,id) {  
 $.ajax({  
 //url格式： /15-seckill-web/seckill/gooods/Ffdaskfjkadlsjklfa/1  
 url: seckillObj.url.seckillURL + random +"/" +id,  
 type:"post",  
 dataType:"json",  
 success:**function** (rtnMessage) {  
 //处理响应结果  
 }  
 });  
}

## 后台Controller处理

### 在GoodsController中定义seckill方法对秒杀请求进行处理，在处理的时候需要进行一些判断

//执行秒杀  
@PostMapping("/seckill/goods/{random}/{id}")  
public @ResponseBody ReturnObject seckill(@PathVariable("random") String random,@PathVariable("id") Integer id){  
 ReturnObject returnObject = new ReturnObject();  
 return returnObject;  
}

### 请求参数random合法性验证，我们这里采用的是长度判断，有些公司将random的某个位置值固定，判断是否为那个值

//1.random参数合法性验证，我们这里采用的是长度判断，有些公司将random的某个位置值固定，判断是否为那个值  
if(random.length() != 36){  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("请求参数有误");  
 return returnObject;  
}

### 根据商品id从Redis中查询出缓存的商品，判断请求参数random和商品的randomName是否匹配

//2.根据商品id从Redis中查询出缓存的商品，判断请求参数random和商品的randomName是否匹配  
String goodsJSON = redisTemplate.opsForValue().get(Constants.*REDIS\_GOODS*+id);  
Goods goods = JSONObject.*parseObject*(goodsJSON,Goods.class);  
if (!random.equals(goods.getRandomname())){  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("请求参数有误");  
 return returnObject;  
}

### 为了保险起见，我们再次验证一下是否在秒杀时间内（这步可以省略）

这里既没有操作磁盘，也没有操作数据库，也没有走网络，所以不会对性能产生影响

//3.为了保险起见，我们再次验证一下是否在秒杀时间内  
Long currentTime = System.*currentTimeMillis*();  
Long startTime = goods.getStarttime().getTime();  
Long endTime = goods.getEndtime().getTime();  
if(currentTime < startTime){  
 //秒杀尚未开始  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("秒杀尚未开始");  
 return returnObject;  
}else if(currentTime > endTime){  
 //秒杀已经结束  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("秒杀已经结束");  
 return returnObject;  
}else{  
 //如果秒杀已经开始，处理业务继续写在这里  
  
 return returnObject;  
}

### 如果已经开始秒杀验证商品是否已经卖光

* 需求

如果商品已经卖光，那么提示用户，不能参与秒杀了

* 常规思路
* 直接查询数据库中商品的库存，如果直接操作数据库，秒杀场景，高并发大流量会给数据库带来很大的压力。
* 从Redis中缓存的商品信息中获取，但是后续秒杀结束后，涉及对库存做修改，操作Redis的商品信息比较麻烦。另外，如果我们5秒缓存预热一次，数据库中商品的库存还没有修改，会被再次把数据库中的库存更新到Redis中。
* 解决方案

所以我们在缓存预热的时候，直接将商品的库存单独存放到Redis中。并且这个信息需要在缓存预热的时候生成，而且只能生成一次，因为我们减库存我的时候，也是操作Redis，数据库暂时不会变，如果每5秒初始化一次，那么会将数据库的原始库存又初始化到Redis中。

**设置值的时候使用setIfAbsent方法**

如果key不存在，那么设置值，如果已经存在，不对其进行设置值了

#### 在15-seckill-service缓存预热的定时任务中缓存商品库存

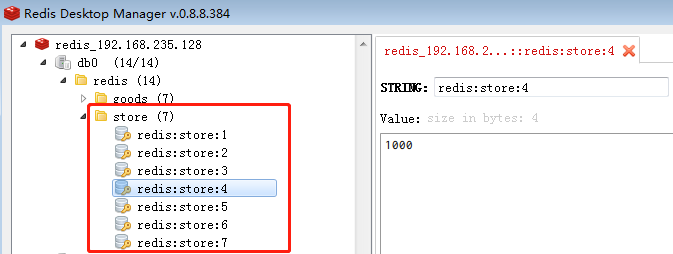
Key的格式 redis:store:商品id Value的值：就是商品的库存

*/\*\*  
 \* 把数据库中商品的库存也预热到Redis  
 \* 注意：这里只能放一次，因为我们减库存我的时候，也是操作Redis，数据库暂时不会变  
 \* 如果每5秒初始化一次，那么会将数据库的原始库存又初始化到Redis中  
 \* setIfAbsent:如果key不存在，那么设置值，如果已经存在，不对其进行设置值了  
 \*/*redisTemplate.opsForValue()  
 .setIfAbsent(Constants.*REDIS\_STORE* + goods.getId(),String.*valueOf*(goods.getStore()));

#### 在15-seckill-interface的Constants常量类下定义商品库存key的前缀

*/\*\*  
 \* 定义Redis中商品库存的key的前缀  
 \* Redis中存放商品库存的格式：redis:goods:商品id  
 \*/*public static final String *REDIS\_STORE* = "redis:store:";

#### 重新运行15-seckill-service，通过Redis DeskTop Manager查看Redis数据



#### 在GoodsControll编写验证商品是否卖光代码

//4.验证商品是否已经卖光了  
//根据商品id，从Redis中获取商品库存  
String redisStore = redisTemplate.opsForValue().get(Constants.*REDIS\_STORE* + id);  
//判断是否为空 如果不为空将redis存放的库存转换为整形  
Integer store = StringUtils.*isEmpty*(redisStore)? 0 :Integer.*valueOf*(redisStore);  
//其实不会出现小于0的情况  
if(store <= 0 ){  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("来晚了，商品已经抢光了");  
 return returnObject;  
}

**为了对String操作更加方便，在15-seckill-web中引入commons-lang的依赖**

<!--对常用类操作的增强包-->  
<dependency>  
 <groupId>commons-lang</groupId>  
 <artifactId>commons-lang</artifactId>  
 <version>2.5</version>  
</dependency>

### 验证该用户是否已经秒杀过该商品

* 需求

同一件商品，同一个用户只能秒杀一次

* 常规思路

去数据库订单表中查询，是否有用户对该商品的下单信息，但是秒杀场景，高并发大流量下，会给数据库带来很大的压力

* 解决方案

我们这里还是查询采用Redis，**如果用户秒杀了该商品，那就将用户信息及商品信息组合放到Redis中，生成一条秒杀记录**，然后再秒杀的时候，从Redis中取数据进行判断

格式：redis:buy:id:uid

#### 在15-seckill-interface的Constants常量中添加用户是否购买过商品的key的前缀

*/\*\*  
 \* 定义Redis中用户是否买过该商品的key的前缀  
 \* Redis中存放用户是否买过该商品的格式：redis:buy:商品id:用户id  
 \*/*public static final String *REDIS\_BUY* = "redis:buy:";

#### 在15-seckill-web的GoodsController中编写验证是否买过该商品的代码

//5.验证用户是否买过该商品  
//假设用户的id为888888，实际开发的使用用户的id可以从session中获取  
Integer uid = 888888;   
String redisBuy = redisTemplate.opsForValue().get(Constants.*REDIS\_BUY* + id +":"+ uid);  
//这里我们不需要关心redisBuy中放了什么，只要不为空，就说明用户买个该商品  
if(StringUtils.*isNotEmpty*(redisBuy)){  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("您已经秒杀过该商品了，请换个商品秒杀");  
 return returnObject;  
}

### 限流

* 需求

在秒杀场景中，为每一个商品限制最大的参与抢购的人数为10w

不能为所有商品整体一个限流，否则会不平衡的问题，很多人都去秒杀一件商品，但是另一件商品在秒杀的时候，被限制了，误杀！

* 实现方式

一般有专门的限流算法

* 我们使用Redis的List类型或者Redis计数器实现

如果用户参与秒杀，向Redis的List中放一条记录，然后判断List的长度

Redis格式： redis:limit:商品id

#### 在15-seckill-interface的Constants类中，添加限流最大值以及商品秒杀限流key的前缀常量

//商品限流最大值  
public static final int *MAX\_LIMIT* = 100000;   
*/\*\*  
 \* 定义Redis中商品秒杀限流key的前缀  
 \* Redis中存放当前商品的流量访问值的格式：redis:limit:商品id  
 \*/*public static final String *REDIS\_LIMIT* = "redis:limit:";

#### 在15-seckill-web的GoodsController中编写限流代码

//6.限流  
//从Redis中查询出当前商品的访问量  
Long currentSize = redisTemplate.opsForList().size(Constants.*REDIS\_LIMIT* + id);  
if(currentSize > Constants.*MAX\_LIMIT*){  
 //超过最大限流值，拒绝访问  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("服务器繁忙，请稍后再试！~");  
 return returnObject;  
}else{  
 //可以继续执行秒杀  
 // 先向Redis的限流List中放一条数据 返回放完数据之后List的长度  
 Long afterPushSize = redisTemplate.opsForList().leftPush(Constants.*REDIS\_LIMIT* + id,String.*valueOf*(uid));  
 /\*放完元素之后再次判断List的长度是否大于限流值  
 主要处理多线程情况下，很多线程都满足限流条件，都向Redis的List添加元素，避免List元素超出限流值  
 \*/  
 if(afterPushSize >Constants.*MAX\_LIMIT*){  
 redisTemplate.opsForList().rightPop(Constants.*REDIS\_LIMIT* + id);  
 //超过最大限流值，拒绝访问  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("服务器繁忙，请稍后再试！~");  
 return returnObject;  
 }  
}

### 秒杀

#### 减库存

* 需求

秒杀结束后，将商品的库存信息减1

* 常规的方式

减库存是直接操作数据库，高并发大流量的秒杀场景下，会给数据库瞬间带来极大的压力，可能数据库无法支撑（单台MySQL并发能力700左右，单台Redis并发能力5w左右）

* 解决方案

所以减库存在Redis中减

##### 15-seckill-web减库存代码

//减库存  
Long leftStore = redisTemplate.opsForValue().decrement(Constants.*REDIS\_STORE* +"id",1);

#### 下订单（仅仅是将订单发送给MQ）

* 需求

秒杀之后，仅仅将订单发送给MQ，暂时不想数据库订单表中插入数据

* 常规的做法

直接是向数据库中插入订单信息

秒杀场景，可能有很多订单可以插入到数据库，而且主要是瞬间的操作，例如：1s,5s内向数据库插入10w条数据。

所以下单的时候不能直接操作数据库

* 解决方案

我们采用MQ，进行异步下单

同步是阻塞的，是需要等结果的，是可以拿到结果的

异步是非阻塞的，不需要等结果，但是有可能马上拿不到结果

让MQ接收瞬间的巨大的下单请求，但并不是马上瞬间处理完毕，而是一个个处理，插入数据库的频率的降低。**这个频率的降低，我们叫做流量削峰,将单位时刻内，对数据库的操作降缓**

MQ处理完毕之后，仅仅是将消息发送到了ActiveMQ的消息队列中，并没有真正的同步数据库，所以不能马上给前台结果，那么这个时候我们可以告诉前台页面一个中间结果，秒杀请求提交成功，正在处理……或一个图片转动

##### 在15-seckill-web的pom.xml文件添加ActiveMQ起步依赖

<!--SpringBoot集成ActiveMQ的起步依赖-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-activemq</artifactId>  
</dependency>

##### 在15-seckill-web的核心配置文件中配置ActiveMQ信息

#配置activemq的连接信息  
spring.activemq.broker-url=tcp://192.168.235.128:61616  
# 用户名  
spring.activemq.user=system  
# 密码  
spring.activemq.password=123456  
#目的地  
spring.jms.template.default-destination=seckillQueue

##### 在15-seckill-web的GoodsController类中注入JmsTemplate

@Autowired  
private JmsTemplate jmsTemplate;

##### 在15-seckill-web的GoodsController类中编写下订单代码

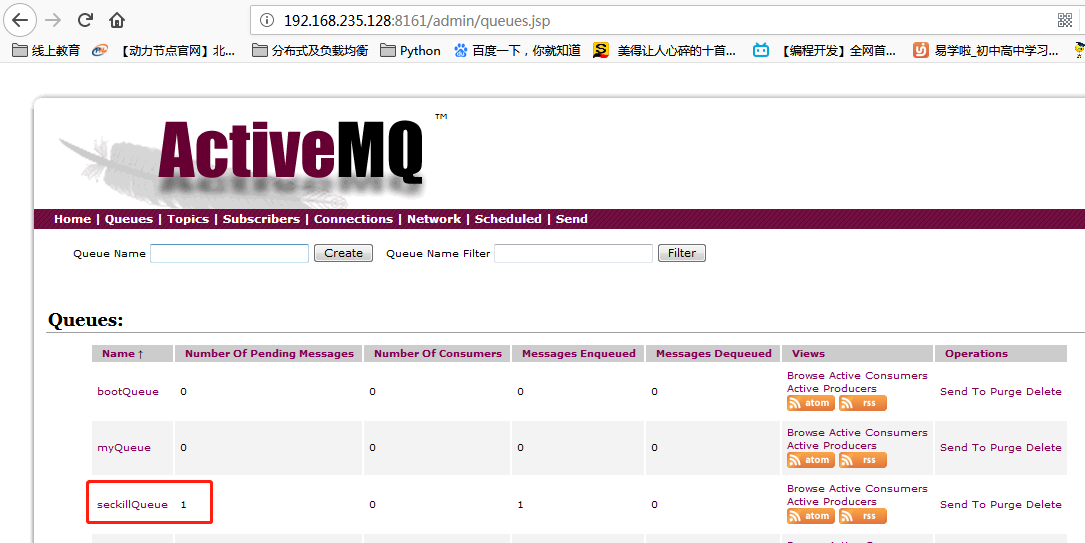
//7.减库存  
Long leftStore = redisTemplate.opsForValue().decrement(Constants.*REDIS\_STORE* +id,1);

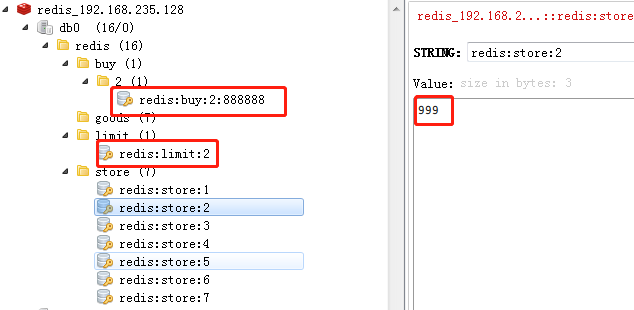
//8.下单到MQ  
if(leftStore >= 0){  
 //可以秒杀，执行下单操作  
 //标记用户已经买过该商品  
 redisTemplate.opsForValue().set(Constants.*REDIS\_BUY* + id +":" +uid,String.*valueOf*(uid));  
 //创建订单对象  
 Orders orders = new Orders();  
 orders.setBuynum(1);  
 orders.setBuyprice(goods.getPrice());  
 orders.setCreatetime(new Date());  
 orders.setGoodsid(id);  
 orders.setOrdermoney(goods.getPrice().multiply(new BigDecimal(1)));  
 orders.setStatus(1);//待支付  
 orders.setUid(uid);  
 //将订单对象转换为json字符串  
 String ordersJSON = JSONObject.*toJSONString*(orders);  
 //通过JmsTemplate向ActiveMQ发送消息  
 jmsTemplate.send(new MessageCreator() {  
 @Override  
 public Message createMessage(Session session) throws JMSException {  
 return session.createTextMessage(ordersJSON);  
 }  
 });  
 returnObject.setErrorCode(Constants.*ONE*);  
 returnObject.setErrorMessage("秒杀请求提交成功，正在处理....");  
 return returnObject;  
}else{  
 //不可以卖了，不能执行下单操作  
 /\*  
 此时Redis中的商品库存可能已经减成负数了，但是对我们业务的处理没有任何影响  
 但为了保持数据的一致性，我们将值再恢复一下  
 \*/  
 redisTemplate.opsForValue().increment(Constants.*REDIS\_STORE* + id,1);  
 returnObject.setErrorCode(Constants.*ZERO*);  
 returnObject.setErrorMessage("来晚了，商品已经抢光了");  
 return returnObject;  
}

##### 在15-seckill-web的seckill.js的execseckill函数中处理返回信息

//执行秒杀请求  
execSeckill:**function** (random,id) {  
 $.ajax({  
 //url格式： /15-seckill-web/seckill/goods/Ffdaskfjkadlsjklfa/1  
 url: seckillObj.url.seckillURL() + random +"/" +id,  
 type:"post",  
 dataType:"json",  
 success:**function** (rtnMessage) {  
 //处理响应结果  
 **if**(rtnMessage.errorCode == 1){  
 //秒杀成功，已经下单到MQ，返回中间结果 可以做动画处理  
 $("#seckillTip").html("<span style='color:red;'>"+ rtnMessage.errorMessage +"</span>");  
 //接下来再发送一个请求获取最终秒杀的结果  
 }**else**{  
 //秒杀失败 展示失败信息  
 $("#seckillTip").html("<span style='color:red;'>"+ rtnMessage.errorMessage +"</span>");  
 }  
 }  
 });  
}

##### 启动ActiveMQ，Redis，MySQL，15-seckill-service，15-seckill-web测试





#### 在15-seckill-service中的RedisTask中同步MySQL数据库库存

*/\*\*  
 \* 每3秒同步一次Redis中的库存到数据库  
 \*/*@Scheduled(cron = "0/3 \* \* \* \* \*")  
public void syncRedisStoreToDB(){  
 System.*out*.println("同步Redis中的库存到数据库...........");  
 //1.查询出所有秒杀商品在Redis中的库存值  
 Set<String> keys = redisTemplate.keys(Constants.*REDIS\_STORE* + "\*");  
 for (String key : keys) {  
 //根据Redis的商品库存key，获取商品的库存  
 int store = Integer.*valueOf*(redisTemplate.opsForValue().get(key));  
 //获取商品的id 在Redis中存放商品库存的格式 redis:store:id  
 int goodsId = Integer.*valueOf*(key.split(":")[2]);  
 //同步到数据库  
 Goods goods = new Goods();  
 goods.setId(goodsId);  
 goods.setStore(store);  
 goodsMapper.updateByPrimaryKeySelective(goods);  
 }  
}

### 异步下单的处理

* 需求

将MQ中的订单同步到数据库

* 实现思路
* 在15-seckill-service中使用异步接收消息的方式对秒杀的订单消息进行消费
* 为了方便对事务的处理，我们在消息消费者MyMessageListener中不直接调用Mapper，而是调用订单的Service
* 如果下单成功

在Service中将秒杀的最终结果返回给前台页面，这里存在一个问题，就是如何将秒杀的结果响应给前台页面？

传统的做法，前台页面可以直接查询数据库的订单表，获取最终的秒杀结果，但是会对数据库造成压力，我们这里借助第三方Redis，将返回的结果保存到Redis中，然后让前台页面到Redis中进行查询。

* 如果下单失败

在Service层中抛出异常，在MyMessageListener中捕获异常，对之前做的处理进行恢复，主要包括库存恢复、购买标记、限流列表中删除一个元素

恢复的操作我们也专门在Service中封装方法

#### 在15-seckill-service中添加ActiveMQ相关依赖

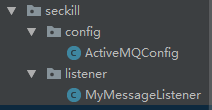
<!--SpringBoot集成ActiveMQ的起步依赖-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-activemq</artifactId>  
</dependency>

#### 在15-seckill-service中的pom.xml文件中添加ActiveMQ配置信息

#配置activemq的连接信息  
spring.activemq.broker-url=tcp://192.168.235.128:61616  
spring.activemq.user=system  
spring.activemq.password=123456  
#目的地  
spring.jms.template.default-destination=seckillQueue  
# 消息发送模式 true发布订阅 false点对点 默认false点对点  
spring.jms.pub-sub-domain=**false**

# SpringBoot 2.1.3之后需要配置  
spring.jms.cache.enabled=**false**

#### 从13-activemq-boot-receiver-async-02中拷贝ActiveMQ异步接收的代码config和listener目录下的内容



#### ActiveMQConfig代码（不需要修改）

@Configuration//相当于applicationContext-jms.xml文件  
public class ActiveMQConfig {  
  
 @Autowired  
 private ActiveMQConnectionFactory connectionFactory;  
  
 @Autowired  
 private MyMessageListener myMessageListener;  
   
 @Value("${spring.jms.template.default-destination}")  
 private String destination;  
  
 @Value("${spring.jms.pub-sub-domain}")  
 private boolean pubSubDomain;  
   
 @Bean //@Bean注解就相当于配置文件的bean标签  
 public DefaultMessageListenerContainer defaultMessageListenerContainer(){  
 DefaultMessageListenerContainer listenerContainer = new DefaultMessageListenerContainer();  
 listenerContainer.setConnectionFactory(connectionFactory);  
 listenerContainer.setDestinationName(destination);  
 listenerContainer.setMessageListener(myMessageListener);  
 //设置消息发送模式方式为发布订阅  
 listenerContainer.setPubSubDomain(pubSubDomain);  
 return listenerContainer;  
 }  
}

#### 修改15-seckill-service中的MyMessageListener消费消息

@Component  
public class MyMessageListener implements MessageListener{  
 @Autowired  
 private OrdersService ordersService;  
  
 public void onMessage(Message message) {  
 if(message instanceof TextMessage){  
 try {  
 String ordersJSON = ((TextMessage) message).getText();  
 System.*out*.println("SpringBoot监听器异步接收到的消息为：" + ordersJSON);  
 Orders orders = JSONObject.*parseObject*(ordersJSON,Orders.class);  
 try {  
 //接收到消息，下订单  
 ordersService.addOrders(orders);  
 } catch (Exception e) {  
 e.printStackTrace();  
 //下单失败了，要将之前的一些处理恢复一下  
 ordersService.processException(orders);  
 }  
 } catch (JMSException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

#### 在15-seckill-interface的com.bjpowernode.seckill.service包下创建订单接口OrdersService

public interface OrdersService {  
 */\*\*  
 \* 下订单  
 \*/* int addOrders(Orders orders);  
 */\*\*  
 \* 下单失败对异常的处理  
 \*/* void processException(Orders orders);  
}

#### 在15-seckill-service的com.bjpowernode.seckill.service.impl包下中创建订单接口实现类OrdersServiceImpl

@Service  
public class OrdersServiceImpl implements OrdersService{  
 @Autowired  
 private OrdersMapper ordersMapper;  
  
 @Autowired  
 private RedisTemplate<String,String> redisTemplate;  
  
 @Transactional  
 @Override  
 public int addOrders(Orders orders) {  
 int addRow = ordersMapper.insertSelective(orders);  
 if(addRow >0){  
 /\*下单成功，告知前台秒杀最终结果，我们这是service项目，由消息消费者调用，不能  
 直接和前台打交道，所以需要前台重新发送请求，去数据库订单表中查询结果，但是这样  
 对数据库带来压力，所以我们将秒杀最终结果放到Redis中，然后前台页面去Redis中查询  
 \*/  
 //用我们自定义我的RTO对象封装秒杀结果  
 ReturnObject returnObject = new ReturnObject();  
 returnObject.setErrorCode(Constants.*ONE*);  
 returnObject.setErrorMessage("秒杀成功");  
 returnObject.setData(orders);  
 String returnJSON = JSONObject.*toJSONString*(returnObject);  
 redisTemplate.opsForValue().set(Constants.*REDIS\_RESULT* +  
 orders.getGoodsid() +":" + orders.getUid(),returnJSON);  
 //当前这个人秒杀全部结束，应该把当前这个人从限流列表中删除,让后面的人再进来秒杀  
 redisTemplate.opsForList().rightPop(Constants.*REDIS\_LIMIT* + orders.getGoodsid());  
 }else{  
 //下单失败，抛出运行时异常  
 throw new RuntimeException("秒杀下单失败");  
 }  
 return addRow;  
 }  
 */\*\*  
 下单失败之后，进行之前处理数据的恢复  
 \*/* @Override  
 public void processException(Orders orders) {  
 // 1.库存恢复  
 redisTemplate.opsForValue().increment(Constants.*REDIS\_STORE* + orders.getGoodsid(),1);  
 //2.购买标记清除  
 redisTemplate.delete(Constants.*REDIS\_BUY* + orders.getGoodsid() +":" + orders.getUid());  
 // 3.限流列表中删除一个元素  
 redisTemplate.opsForList().rightPop(Constants.*REDIS\_LIMIT* + orders.getGoodsid());

//4.将失败信息放到Redis中，便于前台页面再次获取

ReturnObject returnObject = new ReturnObject();

returnObject.setErrorCode(Constants.*ZERO*);

returnObject.setErrorMessage("秒杀失败");

returnObject.setData(orders);

String returnJSON = JSONObject.*toJSONString*(returnObject);

redisTemplate.opsForValue().set(Constants.*REDIS\_RESULT* +

orders.getGoodsid() + ":" + orders.getUid(), returnJSON);  
 }  
}

#### 在15-seckill-interface的Constants类中添加存放最终秒杀结果Key的前缀

*/\*\*  
 \* 定义Redis中商品秒杀秒杀结果key的前缀  
 \* Redis中存放当前商品的流量访问值的格式：redis:result:商品id:用户id  
 \*/*public static final String *REDIS\_RESULT* = "redis:result:";

#### 在15-seckill-service的Application类上开启事务

@EnableTransactionManagement  
@SpringBootApplication  
public class Application {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
}

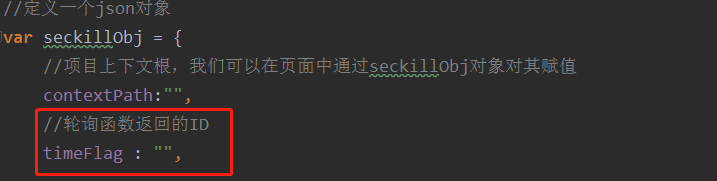
### 库存超卖的解读

参照面试题11-Summary\互联网金融项目-面试.docx

## 前台页面再次获取秒杀结果（MQ处理之后的）

分析：再次获取秒杀结果，获取的是消息队列处理后的结果，有可能秒杀成功，有可能秒杀失败，还有可能是发送请求了，但是消息队列的消息还没有被消费，这个时候，就**需要轮询的调用获取秒杀结果的函数，**我们可以通过window.setInterval方法实现，该方法对应的clearInterval可以终止轮询，但是需要将setInterval的返回值ID作为参数，所以我们需要定义一个全局变量接收setInterval的返回值

### 在15-seckill-web的模块的seckill.js中的execSeckill函数中轮询调用再次获取秒杀结果的queryResult函数



//处理响应结果  
**if**(rtnMessage.errorCode == 1){  
 //秒杀成功，已经下单到MQ，返回中间结果 可以做动画处理  
 $("#seckillTip").html("<span style='color:red;'>"+ rtnMessage.errorMessage +"</span>");  
 //接下来再发送一个请求获取最终秒杀的结果（轮询，每3秒查一次）  
 seckillObj.timeFlag = window.setInterval(**function**(){  
 seckillObj.func.queryResult(id)  
 },3\*1000);  
}**else**{  
 //秒杀失败 展示失败信息  
 $("#seckillTip").html("<span style='color:red;'>"+ rtnMessage.errorMessage +"</span>");  
}

### 在15-seckill-web的模块的seckill.js中编写queryResult函数

//查询最终秒杀结果  
queryResult:**function** (id) {  
 $.ajax({  
 url: seckillObj.url.resultURL() +id,  
 type:"post",  
 dataType:"json",  
 success:**function** (rtnMessage) {  
 **if**(rtnMessage.errorCode == 1){  
 //秒杀成功  
 $("#seckillTip").html("<span style='color:blue;'>"+ rtnMessage.errorMessage +"</span>");  
 //终止轮询  
 window.clearInterval(seckillObj.timeFlag);  
 }**else if**(rtnMessage.errorCode == 0){  
 //秒杀失败  
 $("#seckillTip").html("<span style='color:blue;'>"+ rtnMessage.errorMessage +"</span>");  
 //终止轮询  
 window.clearInterval(seckillObj.timeFlag);  
 }**else**{  
 //3秒后，依然没有查询到结果，那么需要3秒后，继续发送请求获取秒杀结果，我们这里不需要做什么  
 }  
 }  
 });  
}

### 在15-seckill-web的seckill.js的url属性中定义resultURL

resultURL:**function** () {  
 **return** seckillObj.contextPath +"/seckill/result/";  
}

### 在15-seckill-web的GoodsController中处理获取最终秒杀结果的请求

@PostMapping("/seckill/result/{id}")  
public @ResponseBody ReturnObject result(@PathVariable("id") Integer id){  
 ReturnObject returnObject = new ReturnObject();  
 //在Redis中暂时没有查询到结果  
 returnObject.setErrorCode(2);  
 //用户再次查询，肯定处于登录状态，可以从session获取用户信息（我们这里省略了用户登录）  
 String resultJSON = redisTemplate.opsForValue().get(Constants.*REDIS\_RESULT* + id + ":" + "888888");  
 return StringUtils.*isEmpty*(resultJSON)?returnObject : JSONObject.*parseObject*(resultJSON,ReturnObject.class);  
}

## 秒杀业务测试

测试前清空Redis，恢复数据库数据，查看ActiveMQ控制台情况

### 单用户测试

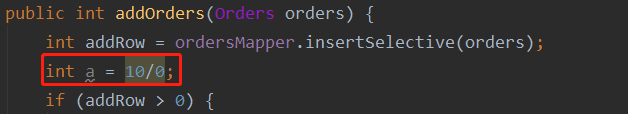
### 多线程模拟多用户测试

修改15-seckill-web中GoodsController中的seckill方法

//接收用户秒杀请求，通过多线程模拟多用户访问  
@PostMapping("/seckill/goods/{random}/{id}")  
public @ResponseBody ReturnObject seckill(@PathVariable("random") String random,@PathVariable("id") Integer id){  
 //最佳实践，线程的个数等于cpu的个数或者2倍，如果过多，会导致上下文切换过于频繁  
 int cpu = 8;  
 ExecutorService executorService = Executors.*newFixedThreadPool*(cpu \* 2);  
 for (int i = 0; i < 1000000; i++) {  
 Integer uid = i;  
 executorService.submit(new Runnable() {  
 @Override  
 public void run() {  
 seckillTest(uid,random,id);  
 }  
 });  
 }  
 return seckillTest(888888,random,id);  
}  
  
//执行秒杀  
/\*@PostMapping("/seckill/goods/{random}/{id}")  
public @ResponseBody ReturnObject seckill(@PathVariable("random") String random,@PathVariable("id") Integer id){\*/  
public ReturnObject seckillTest(Integer uid,String random,Integer id){

### 测试下单事务是否生效

在15-seckill-service中将OrderServiceImpl中的addOrders方法中构建一个异常



将线程并发量降低一些，1w

### 设置ActiveMQ消费者的个数

设置消费者的个数，可以解决消息堆积的问题，发消息太多太快，导致消费太慢，获取最终秒杀结果延时比较明显，一般也是cpu个数的2倍

在15-seckill-service中的ActiveMQConfig配置类中加一个配置即可



# 秒杀流程总结

## 商品列表页，用户浏览商品

## 点击进入某个商品的详情页

## 进入商品详情页后

秒杀未开始显示秒杀倒计时

秒杀已开始显示秒杀按钮

秒杀已结束显示秒杀结束

## 用户在商品详情页点击秒杀按钮后

### 验证商品id和秒杀唯一标志是否合法 （秒杀唯一标识：暴露秒杀地址）

### 判断秒杀时间是否开始

判断商品的秒杀开始时间和结束时间，用当前系统时间和他们做比较

### 判断秒杀是否抢光了

判断Redis中该商品的库存>0可以秒杀，<=0秒杀结束

### 判断用户是否已经秒杀过该商品

判断Redis中的key是否存在，用户秒杀后会在Redis中设置一个占位的key来标识用户已经秒杀过该商品

## 判断当前系统流量是否已经超过阈值

* 通过Redis的List数据结构实现
* 每有一个用户请求，就将用户的请求放入List
* 当List的长度达到我们设置的最大值后（通常设置为商品库存数的100倍等）
* 拒绝后续用户的访问，减轻系统的压力
* 用户秒杀流程执行结束后，不管是秒杀成功还是失败，都需要将限流的List弹出一个元素，以便于让后面的人可以再进来一个

## 进行秒杀

### 减库存

**在redis中减库存，采用redis减库存 decrBy 方法（单线程的操作，不会产生数据冲突）**

我们数据库中并没有直接减库存，因为数据库性能瓶颈问题

最终我们采用定时任务每隔几秒同步一次Redis库存到数据库，让数据库的库存和redis的库存同步

### 下订单

* + 异步下订单，也是为了避免直接操作数据库
  + 采用队列ActiveMQ下订单
  + 减库存成功后，给MQ发一个消息
  + 消息监听器接收到消息后在数据库创建订单
  + 如果消息消费不过来，可以设置concurrency="8" 8个消费者，那么消费消息的速度就会加快，不会产生消息的堆积

### 告知前台页面秒杀结果

创建订单成功或者失败后，都把秒杀结果放入到redis中

前台页面采用ajax轮询方式查询redis获取最终秒杀结果，给用户提示

# 简历

## P2P项目

* + 在P2P项目中添加秒杀功能模块
  + P2P项目增加秒杀功能模块，该模式是我开发的；
  + 投资抢标 （功能模块）
  + 秒杀标
    - 每周三和周五 上午10点 会有秒杀标
    - 我们会发布1个20W左右的秒杀标
    - 利率较高 17% 、 15%，周期较短，1个月，45天等等
    - 抢购投资的人就比较多，几秒内投资满标

## 和商品秒杀比较：

商品的库存（1000件） == 理财产品的剩余可投金额（20万）

商品减库存每次减1（-1） == 理财产品用户每次投资的金额，假设用户投1000元（-1000元）

seckill-interface ： p2p-commons + p2p-exterface

seckill-web : p2p-web

seckill-service : p2p-dataservice