

1. 需求说明

说明：

- 设置抢购活动，比如活动对应的代金券、开始时间、结束时间、秒杀券的数量
- 定时开始抢购活动，禁止超卖
- 用户抢购限制，一个用户只能购买一单

表结构设计

```
1 -- -----
2 -- Table structure for `t_seckill_vouchers`
3 --
4 DROP TABLE IF EXISTS `t_seckill_vouchers`;
5 CREATE TABLE `t_vouchers` (
6   `id` int(10) NOT NULL AUTO_INCREMENT ,
7   `title` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
8   DEFAULT NULL COMMENT '代金券标题' ,
9   `thumbnail` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
10  DEFAULT NULL COMMENT '缩略图' ,
11  `amount` int(11) NULL DEFAULT NULL COMMENT '抵扣金额' ,
12  `price` decimal(10,2) NULL DEFAULT NULL COMMENT '售价' ,
13  `status` int(10) NULL DEFAULT NULL COMMENT '-1=过期 0=下架 1=上架' ,
14  `start_use_time` datetime NULL DEFAULT NULL COMMENT '开始使用时间' ,
15  `expire_time` datetime NULL DEFAULT NULL COMMENT '过期时间' ,
16  `redeem_restaurant_id` int(10) NULL DEFAULT NULL COMMENT '验证餐厅' ,
17  `stock` int(11) NULL DEFAULT 0 COMMENT '库存' ,
18  `stock_left` int(11) NULL DEFAULT 0 COMMENT '剩余数量' ,
19  `description` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
20  DEFAULT NULL COMMENT '描述信息' ,
21  `clause` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
22  DEFAULT NULL COMMENT '使用条款' ,
23  `create_date` datetime NULL DEFAULT NULL ,
24  `update_date` datetime NULL DEFAULT NULL ,
25  `is_valid` tinyint(1) NULL DEFAULT NULL ,
26  PRIMARY KEY (`id`)
27 )
28 ENGINE=InnoDB
29 DEFAULT CHARACTER SET=utf8 COLLATE=utf8_general_ci
30 AUTO_INCREMENT=1
31 ROW_FORMAT=COMPACT
32 ;
```

抢购活动表

```
1 CREATE TABLE `t_seckill_vouchers` (
2   `id` int(11) NOT NULL AUTO_INCREMENT ,
```

```

3   `fk_voucher_id` int(11) NULL DEFAULT NULL ,
4   `amount` int(11) NULL DEFAULT NULL ,
5   `start_time` datetime NULL DEFAULT NULL ,
6   `end_time` datetime NULL DEFAULT NULL ,
7   `is_valid` int(11) NULL DEFAULT NULL ,
8   `create_date` datetime NULL DEFAULT NULL ,
9   `update_date` datetime NULL DEFAULT NULL ,
10  PRIMARY KEY (`id`)
11 )
12 ENGINE=InnoDB
13 DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_general_ci
14 AUTO_INCREMENT=1
15 ROW_FORMAT=COMPACT
16 ;

```

订单表

```

1 CREATE TABLE `t_voucher_orders` (
2   `id` int(11) NOT NULL AUTO_INCREMENT ,
3   `order_no` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci
NULL DEFAULT NULL ,
4   `fk_voucher_id` int(11) NULL DEFAULT NULL ,
5   `fk_diner_id` int(11) NULL DEFAULT NULL ,
6   `qrcode` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
DEFAULT NULL COMMENT '图片地址' ,
7   `payment` tinyint(4) NULL DEFAULT NULL COMMENT '0=微信支付 1=支付宝支付' ,
8   `status` tinyint(1) NULL DEFAULT NULL COMMENT '订单状态: -1=已取消 0=未支付 1=
已支付 2=已消费 3=已过期' ,
9   `fk_seckill_id` int(11) NULL DEFAULT NULL COMMENT '如果是抢购订单时, 抢购订单的
id' ,
10  `order_type` int(11) NULL DEFAULT NULL COMMENT '订单类型: 0=正常订单 1=抢购订
单' ,
11  `create_date` datetime NULL DEFAULT NULL ,
12  `update_date` datetime NULL DEFAULT NULL ,
13  `is_valid` int(11) NULL DEFAULT NULL ,
14  PRIMARY KEY (`id`)
15 )
16 ENGINE=InnoDB
17 DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_general_ci
18 AUTO_INCREMENT=1
19 ROW_FORMAT=COMPACT
20 ;

```

2. 解决方案

秒杀场景有以下几个特点:

- 大量用户同时进行抢购操作，系统流量激增，服务器瞬时压力很大；
- 请求数量远大于商品数量，只有少数客户可以抢购成功；
- 业务流程不复杂，核心功能是下订单。

针对以上特点，

1. **限流**: 从客户层面考虑，限制单个用户抢购频率；服务端层面，加强校验，识别请求是否来源于真实客户端，并限制请求频率，防止恶意刷单；应用层面，可以使用漏桶算法或者令牌桶算法实现应用级限流。

2. **缓存**: 热点数据从缓存中获得，尽可能减小数据库访问压力。
3. **异步**: 客户抢购成功后立即返回响应，之后通过消息队列，异步处理后续步骤，如发短信、更新数据库等，从而缓解服务器峰值压力。
4. **分流**: 单台服务器无法应对抢购期间大量请求造成的眼里，需要集群部署服务器，通过负载均衡共同处理客户端请求，分散压力。

3. 创建服务ms-seckill

添加依赖

```
1 <dependencies>
2     <!-- eureka client -->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
6     </dependency>
7     <!-- spring web -->
8     <dependency>
9         <groupId>org.springframework.boot</groupId>
10        <artifactId>spring-boot-starter-web</artifactId>
11    </dependency>
12    <!-- mybatis -->
13    <dependency>
14        <groupId>org.mybatis.spring.boot</groupId>
15        <artifactId>mybatis-spring-boot-starter</artifactId>
16    </dependency>
17    <!-- mysql -->
18    <dependency>
19        <groupId>mysql</groupId>
20        <artifactId>mysql-connector-java</artifactId>
21    </dependency>
22    <!-- spring data redis -->
23    <dependency>
24        <groupId>org.springframework.boot</groupId>
25        <artifactId>spring-boot-starter-data-redis</artifactId>
26    </dependency>
27    <!-- commons -->
28    <dependency>
29        <groupId>com.imooc</groupId>
30        <artifactId>commons</artifactId>
31        <version>1.0-SNAPSHOT</version>
32    </dependency>
33 </dependencies>
34
```

配置文件

```
1 server:
2     port: 8083 # 端口
3
4 spring:
5     application:
6         name: ms-seckill # 应用名
7         # 数据库
8         datasource:
9             driver-class-name: com.mysql.cj.jdbc.Driver
```

```

10     username: root
11     password: 123456
12     url: jdbc:mysql://127.0.0.1:3306/db_imoooc?
serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=f
13   else
14     # Redis
15     redis:
16       port: 6379
17       host: 192.168.10.101
18       timeout: 3000
19       password: 123456
20   # Swagger
21   swagger:
22     base-package: com.imooc.seckill
23     title: 慕课美食社交食客API接口文档
24
# 配置 Eureka Server 注册中心
25 eureka:
26   instance:
27     prefer-ip-address: true
28     instance-id: ${spring.cloud.client.ip-address}:${server.port}
29   client:
30     service-url:
31       defaultZone: http://localhost:8080/eureka/
32
33 mybatis:
34   configuration:
35     map-underscore-to-camel-case: true # 开启驼峰映射
36
37 service:
38   name:
39     ms-oauth-server: http://ms-oauth2-server/
40
41 logging:
42   pattern:
43     console: '%d{HH:mm:ss} [%thread] %-5level %logger{50} - %msg%n'
44   level:
45     root: debug
46

```

4. 代码实现 (MySQL版)

提醒：为了方便学习测试，我们将授权中心的令牌失效时间修改为一个月

```

1 # token 有效时间，单位秒
2 tokenValidityTime: 2592000
3 refreshTokenValidityTime: 2592000
4

```

4.1. 相关实体类

抢购代金券活动表

```

1 package com.imooc.commons.model.pojo;
2

```

```

3 import com.fasterxml.jackson.annotation.JsonFormat;
4 import com.imooc.commons.model.base.BaseModel;
5 import io.swagger.annotations.ApiModel;
6 import io.swagger.annotations.ApiModelProperty;
7 import lombok.Getter;
8 import lombok.Setter;
9 import org.springframework.format.annotation.DateTimeFormat;
10
11 import java.util.Date;
12
13 @Setter
14 @Getter
15 @ApiModel(description = "抢购代金券信息")
16 public class SeckillVouchers extends BaseModel {
17
18     @ApiModelProperty("代金券外键")
19     private Integer fkVoucherId;
20     @ApiModelProperty("数量")
21     private int amount;
22     @ApiModelProperty("抢购开始时间")
23     @JsonFormat(pattern = "yyyy-MM-dd HH:mm", timezone = "GMT+8")
24     @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm")
25     private Date startTime;
26     @ApiModelProperty("抢购结束时间")
27     @JsonFormat(pattern = "yyyy-MM-dd HH:mm", timezone = "GMT+8")
28     @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm")
29     private Date endTime;
30
31 }
32

```

代金券订单表

```

1 package com.imooc.commons.model.pojo;
2
3 import com.imooc.commons.model.base.BaseModel;
4 import io.swagger.annotations.ApiModel;
5 import io.swagger.annotations.ApiModelProperty;
6 import lombok.Getter;
7 import lombok.Setter;
8
9 @ApiModel(description = "代金券订单信息")
10 @Getter
11 @Setter
12 public class VoucherOrders extends BaseModel {
13
14     @ApiModelProperty("订单编号")
15     private String orderNo;
16     @ApiModelProperty("代金券")
17     private Integer fkVoucherId;
18     @ApiModelProperty("下单用户")
19     private Integer fkDinerId;
20     @ApiModelProperty("生成qrcode")
21     private String qrcode;
22     @ApiModelProperty("支付方式 0=微信支付 1=支付宝")
23     private int payment;
24     @ApiModelProperty("订单状态 -1=已取消 0=未支付 1=已支付 2=已消费 3=已过期")

```

```
25     private int status;
26     @ApiModelProperty("订单类型 0=正常订单 1=抢购订单")
27     private int orderType;
28     @ApiModelProperty("抢购订单的外键")
29     private int fkSeckillId;
30
31 }
32
```

4.2. 相关配置类

Rest配置类

```
1 package com.imooc.seckill.config;
2
3 import org.springframework.cloud.client.loadbalancer.LoadBalanced;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.http.MediaType;
7 import
8 org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;
9 import org.springframework.web.client.RestTemplate;
10
11 import java.util.Collections;
12
13 @Configuration
14 public class RestTemplateConfiguration {
15
16     @LoadBalanced
17     @Bean
18     public RestTemplate restTemplate() {
19         RestTemplate restTemplate = new RestTemplate();
20         MappingJackson2HttpMessageConverter converter = new
21         MappingJackson2HttpMessageConverter();
22
23         converter.setSupportedMediaTypes(Collections.singletonList(MediaType.TEXT_P
24         LAIN));
25         restTemplate.getMessageConverters().add(converter);
26         return restTemplate;
27     }
28
29 }
```

4.3. 全局异常处理类

```
1 package com.imooc.seckill.handler;
2
3 import com.imooc.common.exception.ParameterException;
4 import com.imooc.common.model.domain.ResultInfo;
5 import com.imooc.common.utils.ResultInfoUtil;
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.web.bind.annotation.ExceptionHandler;
8 import org.springframework.web.bind.annotation.RestControllerAdvice;
9
10 import javax.annotation.Resource;
```

```

11 import javax.servlet.http.HttpServletRequest;
12 import java.util.Map;
13
14 @RestControllerAdvice // 将输出的内容写入ResponseBody中
15 @Slf4j
16 public class GlobalExceptionHandler {
17
18     @Resource
19     private HttpServletRequest request;
20
21     @ExceptionHandler(ParameterException.class)
22     public ResultInfo<Map<String, String>>
23     handlerParameterException(ParameterException ex) {
24         String path = request.getRequestURI();
25         ResultInfo<Map<String, String>> resultInfo = ResultInfoUtil
26             .buildError(ex.getErrorCode(), ex.getMessage(), path);
27         return resultInfo;
28     }
29
30     @ExceptionHandler(Exception.class)
31     public ResultInfo<Map<String, String>> handlerException(Exception ex) {
32         log.info("未知异常: {}", ex);
33         String path = request.getRequestURI();
34         ResultInfo<Map<String, String>> resultInfo =
35         ResultInfoUtil.buildError(path);
36         return resultInfo;
37     }
38 }
39

```

4.4. 添加秒杀活动

4.4.1. Mapper

```

1 package com.imooc.seckill.mapper;
2
3 import com.imooc.commons.model.pojo.SeckillVouchers;
4 import org.apache.ibatis.annotations.*;
5
6 /**
7 * 秒杀代金券 Mapper
8 */
9 public interface SeckillVouchersMapper {
10
11     // 新增秒杀活动
12     @Insert("insert into t_seckill_vouchers (fk_voucher_id, amount,
13     start_time, end_time, is_valid, create_date, update_date) " +
14         " values (#{fkVoucherId}, #{amount}, #{startTime}, #{endTime},
15         1, now(), now())")
16     @Options(useGeneratedKeys = true, keyProperty = "id")
17     int save(SeckillVouchers seckillVouchers);
18
19     // 根据代金券 ID 查询该代金券是否参与抢购活动
20
21 }
22

```

```

18     @Select("select id, fk_voucher_id, amount, start_time, end_time,
19      is_valid " +
20          " from t_seckill_vouchers where fk_voucher_id = #{voucherId}")
21      SeckillVouchers selectVoucher(Integer voucherId);
22
23

```

4.4.2. Service

```

1 package com.imooc.seckill.service;
2
3 import com.imooc.common.model.pojo.SeckillVouchers;
4 import com.imooc.common.utils.AssertUtil;
5 import com.imooc.seckill.mapper.SeckillVouchersMapper;
6 import org.springframework.stereotype.Service;
7 import org.springframework.transaction.annotation.Transactional;
8
9 import javax.annotation.Resource;
10 import java.util.Date;
11
12 /**
13 * 秒杀业务逻辑层
14 */
15 @Service
16 public class SeckillService {
17
18     @Resource
19     private SeckillVouchersMapper seckillVouchersMapper;
20
21     /**
22      * 添加需要抢购的代金券
23      *
24      * @param seckillVouchers
25      */
26     @Transactional(rollbackFor = Exception.class)
27     public void addSeckillVouchers(SeckillVouchers seckillVouchers) {
28         // 非空校验
29         AssertUtil.isTrue(seckillVouchers.getFkVoucherId() == null, "请选择需
要抢购的代金券");
30         AssertUtil.isTrue(seckillVouchers.getAmount() == 0, "请输入抢购总数
量");
31         Date now = new Date();
32         AssertUtil.notNull(seckillVouchers.getStartTime(), "请输入开始时
间");
33         // 生产环境下面一行代码需放行, 这里注释方便测试
34         // AssertUtil.isTrue(now.after(seckillVouchers.getStartTime()), "开
始时间不能早于当前时间");
35         AssertUtil.notNull(seckillVouchers.getEndTime(), "请输入结束时间");
36         AssertUtil.isTrue(now.after(seckillVouchers.getEndTime()), "结束时间不
能早于当前时间");
37
38         AssertUtil.isTrue(seckillVouchers.getStartTime().after(seckillVouchers.getE
ndTime()), "开始时间不能晚于结束时间");
39
        // 验证数据库中是否已经存在该券的秒杀活动

```

```

40         SeckillVouchers seckillVouchersFromDb =
41     seckillVouchersMapper.selectVoucher(seckillVouchers.getFkVoucherId());
42     AssertUtil.isTrue(seckillVouchersFromDb != null, "该券已经拥有了抢购活
43     动");
44     // 插入数据库
45     seckillVouchersMapper.save(seckillVouchers);
46 }
47

```

4.4.3. Controller

```

1 package com.imooc.seckill.controller;
2
3 import com.imooc.commons.model.domain.ResultInfo;
4 import com.imooc.commons.model.pojo.SeckillVouchers;
5 import com.imooc.commons.utils.ResultInfoUtil;
6 import com.imooc.seckill.service.SeckillService;
7 import org.springframework.web.bind.annotation.PostMapping;
8 import org.springframework.web.bind.annotation.RequestBody;
9 import org.springframework.web.bind.annotation.RestController;
10
11 import javax.annotation.Resource;
12 import javax.servlet.http.HttpServletRequest;
13
14 /**
15 * 秒杀控制层
16 */
17 @RestController
18 public class SeckillController {
19
20     @Resource
21     private SeckillService seckillService;
22     @Resource
23     private HttpServletRequest request;
24
25 /**
26 * 新增秒杀活动
27 *
28 * @param seckillVouchers
29 * @return
30 */
31 @PostMapping("add")
32 public ResultInfo<String> addSeckillVouchers(@RequestBody
SeckillVouchers seckillVouchers) {
33     seckillService.addSeckillVouchers(seckillVouchers);
34     return ResultInfoUtil.buildSuccess(request.getServletPath(),
35             "添加成功");
36 }
37
38 }
39

```

4.4.4. 启动类

```
1 package com.imooc.seckill;
2
3 import org.mybatis.spring.annotation.MapperScan;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6
7 @MapperScan("com.imooc.seckill.mapper")
8 @SpringBootApplication
9 public class SeckillApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(SeckillApplication.class);
13     }
14
15 }
16
```

4.4.5. 在微服务中配置

在ms-gateway中放行，此接口为平台后台调用，不需要食客登录

```
1 spring:
2   application:
3     name: ms-gateway
4   cloud:
5     gateway:
6       discovery:
7         locator:
8           enabled: true # 开启配置注册中心进行路由功能
9           lower-case-service-id: true # 将服务名称转小写
10      routes:
11
12        - id: ms-seckill
13          uri: lb://ms-seckill
14          predicates:
15            - Path=/seckill/**
16          filters:
17            - StripPrefix=1
18
19   secure:
20     ignore:
21       urls: #配置白名单路径
22
23       - /seckill/add
24
```

4.4.6. PostMan进行测试

访问：<http://localhost:8083/add>

POST http://localhost:8083/add

Body (raw JSON)

```
{
  "fkVoucherId": 1,
  "amount": 100,
  "startTime": "2020-11-16 11:30:30",
  "endTime": "2020-12-16 11:30:30"
}
```

Status: 200 OK Time: 390 ms Size: 234 B

id	fk_voucher_id	amount	start_time	end_time	is_valid	create_date	update_date
1	1	100	2020-11-16 11:30:30	2020-12-16 11:30:30	1	2020-11-16 11:30:30	2020-11-16 11:30:30

4.5. 客户端秒杀

4.5.1. Mapper

SeckillVoucherMapper.java

```

1 package com.imooc.seckill.mapper;
2
3 import com.imooc.commons.model.pojo.SeckillVouchers;
4 import org.apache.ibatis.annotations.*;
5
6 /**
7  * 秒杀代金券 Mapper
8 */
9 public interface SeckillVouchersMapper {
10
11     // 根据代金券 ID 查询该代金券是否参与抢购活动
12     @Select("select id, fk_voucher_id, amount, start_time, end_time,
13     is_valid " +
14             " from t_seckill_vouchers where fk_voucher_id = #{voucherId}")
15     SeckillVouchers selectVoucher(Integer voucherId);
16
17     // 减库存
18     @Update("update t_seckill_vouchers set amount = amount - 1 " +
19             " where id = #{seckillId}")
20     int stockDecrease(@Param("seckillId") int seckillId);
21 }
22

```

VoucherOrderMapper.java

```

1 package com.imooc.seckill.mapper;
2
3 import com.imooc.commons.model.pojo.Voucherorders;
4 import org.apache.ibatis.annotations.Insert;
5 import org.apache.ibatis.annotations.Param;
6 import org.apache.ibatis.annotations.Select;
7
8 /**
9 * 代金券订单 Mapper
10 */
11 public interface VoucherOrdersMapper {
12
13     // 根据食客 ID 和秒杀 ID 查询代金券订单
14     @Select("select id, order_no, fk_voucher_id, fk_diner_id, qrcode,
15 payment," +
16             " status, fk_seckill_id, order_type, create_date, update_date, "
17             +
18             " is_valid from t_voucher_orders where fk_diner_id = #{dinerId}
19             +
20             " and fk_seckill_id = #{seckillId} and is_valid = 1 and status >
21             -1 ")
22     VoucherOrders findDinerOrder(@Param("dinerId") Integer dinerId,
23                                     @Param("seckillId") Integer seckillId);
24
25     // 新增代金券订单
26     @Insert("insert into t_voucher_orders (order_no, fk_voucher_id,
27 fk_diner_id, " +
28             " status, fk_seckill_id, order_type, create_date, update_date,
29 is_valid)" +
30             " values (#{orderNo}, #{fkVoucherId}, #{fkDinerId}, #{status}, #
31 {fkSeckillId}, " +
32             " #{orderType}, now(), now(), 1)")
33     int save(VoucherOrders voucherOrders);
34
35 }

```

4.5.2. Service

- 基本参数校验
- 判断此代金券是否加入抢购
- 判断是否有效
- 判断是否开始、结束
- 判断是否买完
- 获取登录用户信息
- 判断登录用户是否已抢到（一个客户针对此次活动只能抢购一次）
- 扣库存
- 下单

```

1 @Resource
2 private VoucherOrdersMapper voucherOrdersMapper;
3 @Resource
4 private SeckillVouchersMapper seckillVouchersMapper;
5 @Value("${service.name.ms-oauth-server}")
6 private String oauthServerName;
7 @Resource

```

```
8  private RestTemplate restTemplate;
9
10 /**
11 * 抢购代金券
12 *
13 * @param voucherId 代金券 ID
14 * @param accessToken 登录token
15 * @Param path 访问路径
16 */
17 public ResultInfo doSeckill(Integer voucherId, String accessToken, String
path) {
18     // 基本参数校验
19     AssertUtil.isTrue(voucherId == null || voucherId < 0, "请选择需要抢购的代金
券");
20     AssertUtil.isNotEmpty(accessToken, "请登录");
21     // 判断此代金券是否加入抢购
22     SeckillVouchers seckillVouchers =
seckillVouchersMapper.selectVoucher(voucherId);
23     AssertUtil.isTrue(seckillVouchers == null, "该代金券并未有抢购活动");
24     // 判断是否有效
25     AssertUtil.isTrue(seckillVouchers.getIsValid() == 0, "该活动已结束");
26     // 判断是否开始、结束
27     Date now = new Date();
28     AssertUtil.isTrue(now.before(seckillVouchers.getStartTime()), "该抢购还未
开始");
29     AssertUtil.isTrue(now.after(seckillVouchers.getEndTime()), "该抢购已结
束");
30     // 判断是否卖完
31     AssertUtil.isTrue(seckillVouchers.getAmount() < 1, "该券已经卖完了");
32     // 获取登录用户信息
33     String url = oauthServerName + "user/me?access_token={accessToken}";
34     ResultInfo resultInfo = restTemplate.getForObject(url, ResultInfo.class,
accessToken);
35     if (resultInfo.getCode() != ApiConstant.SUCCESS_CODE) {
36         resultInfo.setPath(path);
37         return resultInfo;
38     }
39     // 这里的data是一个LinkedHashMap, SignInDinerInfo
40     SignInDinerInfo dinerInfo = BeanUtil.fillBeanWithMap((LinkedHashMap)
resultInfo.getData(),
41                                         new
SignInDinerInfo(), false);
42     // 判断登录用户是否已抢到(一个用户针对这次活动只能买一次)
43     VoucherOrders order =
voucherOrdersMapper.findDinerOrder(dinerInfo.getId(),
44
seckillVouchers.getId());
45     AssertUtil.isTrue(order != null, "该用户已抢到该代金券, 无需再抢");
46     // 扣库存
47     int count =
seckillVouchersMapper.stockDecrease(seckillVouchers.getId());
48     AssertUtil.isTrue(count == 0, "该券已经卖完了");
49     // 下单
50     VoucherOrders voucherOrders = new VoucherOrders();
51     voucherOrders.setFkDinerId(dinerInfo.getId());
52     voucherOrders.setFkSeckillId(seckillVouchers.getId());
53     voucherOrders.setFkVoucherId(seckillVouchers.getFkVoucherId());
54     String orderNo = IdUtil.getSnowflake(1, 1).nextIdStr();
```

```
55     voucherOrders.setOrderNo(orderNo);
56     voucherOrders.setOrderType(1);
57     voucherOrders.setStatus(0);
58     count = voucherOrdersMapper.save(voucherOrders);
59     AssertUtil.isTrue(count == 0, "用户抢购失败");
60
61     return ResultInfoUtil.buildSuccess(path, "抢购成功");
62 }
63 }
```

4.5.3. Controller

```
1 /**
2  * 秒杀下单
3  *
4  * @param voucherId
5  * @param access_token
6  * @return
7 */
8 @PostMapping("{voucherId}")
9 public ResultInfo<String> doSeckill(@PathVariable Integer voucherId, String
access_token) {
10     ResultInfo resultInfo = seckillService.doSeckill(voucherId,
access_token, request.getServletPath());
11     return resultInfo;
12 }
13 }
```

5. 压力测试

Windows环境下使用JMeter5.3模拟抢购场景

下载工具JMeter5.3工具

下载地址: https://jmeter.apache.org/download_jmeter.cgi



About
• Overview
• License

Download
• Download Releases
• Release Notes

Documentation
• Get Started
• User Manual
• Best Practices
• Component Reference
• Functions Reference
• Properties Reference
• Change History
• Javadocs
• JMeter Wiki
• FAQ(Wiki)

Tutorials
• Distributed Testing
• Recording Tests
• JUnit Sampler
• Access Log Sampler
• Extending JMeter

Community
• Issue Tracking
• Security
• Mailing Lists
• Source Repositories
• Building and Contributing
• Project Info at Apache
• Contributors

Foundation

Download Apache JMeter

[Twitter](#)

[GitHub](#)

We recommend you use a mirror to download our release builds, but you must [verify the integrity](#) of the downloaded files using signatures downloaded from our main distribution directories. Recent releases (48 hours) may not yet be available from all the mirrors.

You are currently using <https://mirrors.bfsu.edu.cn/apache/>. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are backup mirrors (at the end of the mirrors list) that should be available.

Other mirrors: <https://mirror.bit.edu.cn/apache/> [Change](#)

The **KEYS** link leads to the code signing keys used to sign the product. The **PGP** link downloads the OpenPGP compatible signature from our main site. The **SHA-512** link downloads the sha512 checksum from the main site. Please [verify the integrity](#) of the downloaded file.

For more information concerning Apache JMeter, see the [Apache JMeter](#) site.

KEYS

Apache JMeter 5.3 (Requires Java 8+)

Binaries

[apache-jmeter-5.3.tgz sha512 pgp](#)
[apache-jmeter-5.3.zip sha512 pgp](#)

Source

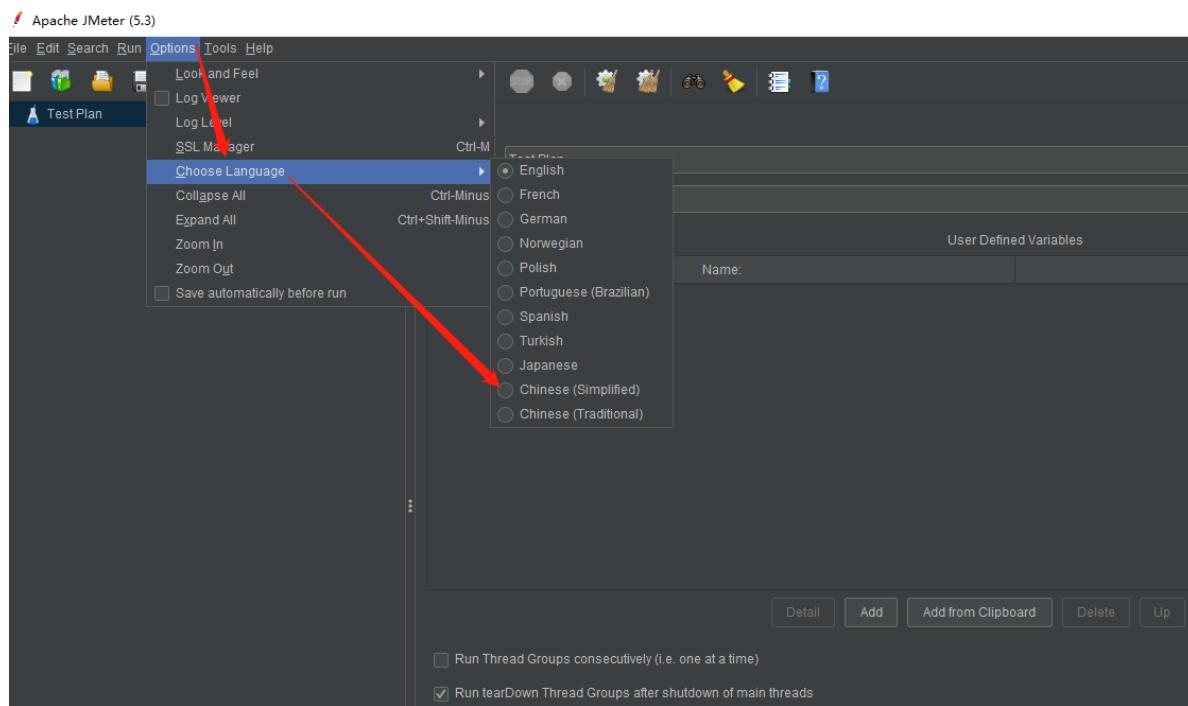
[apache-jmeter-5.3_src.tgz sha512 pgp](#)
[apache-jmeter-5.3_src.zip sha512 pgp](#)

解压启动

在解压目录下的bin目录中找到jmeter.bat，这是windows启动脚本，双击启动

名称	修改日期	类型	大小
compliates	2020/11/13 11:50	文件	1 KB
ApacheJMeter.jar	1980/2/1 0:00	Executable Jar File	14 KB
BeanShellAssertion.bshrc	1980/2/1 0:00	BSHRC 文件	2 KB
BeanShellFunction.bshrc	1980/2/1 0:00	BSHRC 文件	3 KB
BeanShellListeners.bshrc	1980/2/1 0:00	BSHRC 文件	2 KB
BeanShellSampler.bshrc	1980/2/1 0:00	BSHRC 文件	3 KB
create-rmi-keystore.bat	1980/2/1 0:00	Windows 批处理...	2 KB
create-rmi-keystore.sh	1980/2/1 0:00	Shell Script	2 KB
hc.parameters	1980/2/1 0:00	PARAMETERS 文...	2 KB
heapdump.cmd	1980/2/1 0:00	Windows 命令脚本	2 KB
heapdump.sh	1980/2/1 0:00	Shell Script	1 KB
jaas.conf	1980/2/1 0:00	CONF 文件	2 KB
jmeter	1980/2/1 0:00	文件	9 KB
jmeter.bat	1980/2/1 0:00	Windows 批处理...	9 KB
jmeter.log	2020/11/14 0:12	文本文档	4,322 KB
jmeter.properties	2020/11/13 14:56	PROPERTIES 文件	56 KB
jmeter.sh	1980/2/1 0:00	Shell Script	4 KB
jmeter-n.cmd	1980/2/1 0:00	Windows 命令脚本	2 KB
jmeter-n.r.cmd	1980/2/1 0:00	Windows 命令脚本	2 KB
jmeter-server	1980/2/1 0:00	文件	2 KB
jmeter-server.bat	1980/2/1 0:00	Windows 批处理...	3 KB
jmeter-server.log	2020/11/11 11:50	文本文档	7 KB
jmeter-t.cmd	1980/2/1 0:00	Windows 命令脚本	2 KB
jmeterw.cmd	1980/2/1 0:00	Windows 命令脚本	1 KB
krb5.conf	1980/2/1 0:00	CONF 文件	2 KB
log4j2.xml	1980/2/1 0:00	XML 文档	5 KB

配置中文语言



生成登录token

导入注册diners数据

数据库运行init_diners_data.sql文件。

在ms-oauth2-server项目中编写测试用例。

修改OAUTHSERVERApplicationTests代码，调用mock测试客户端

```
1 package com.imooc.oauth2.server;
2
3 import
4     org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc
5 ;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.test.web.servlet.MockMvc;
8
9 import javax.annotation.Resource;
10
11 @SpringBootTest
12 @AutoConfigureMockMvc
13 public class OAuth2ServerApplicationTests {
14
15     @Resource
16     protected MockMvc mockMvc;
17 }
```

创建OAuthControllerTests生成token，文件存在根目录下

```
1 package com.imooc.oauth2.server.controller;
2
3 import cn.hutool.json.JSONObject;
4 import cn.hutool.json.JSONUtil;
5 import com.imooc.common.model.domain.ResultInfo;
6 import com.imooc.oauth2.server.OAuth2ServerApplicationTests;
```

```

7 import org.junit.jupiter.api.Test;
8 import org.springframework.http.MediaType;
9 import org.springframework.test.web.servlet.MvcResult;
10 import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
11 import org.springframework.util.Base64Utils;
12
13 import java.nio.file.Files;
14 import java.nio.file.Paths;
15
16 import static
17     org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
18
19 public class OAuthControllerTest extends OAuth2ServerApplicationTests {
20
21     @Test
22     public void writeToken() throws Exception {
23         String authorization =
24             Base64Utils.encodeToString("appId:123456".getBytes());
25         StringBuffer tokens = new StringBuffer();
26         for (int i = 0; i < 2000; i++) {
27             MvcResult mvcResult =
28                 super.mockMvc.perform(MockMvcRequestBuilders.post("/oauth/token")
29                     .header("Authorization", "Basic " + authorization)
30                     .contentType(MediaType.APPLICATION_FORM_URLENCODED)
31                     .param("username", "test" + i)
32                     .param("password", "123456")
33                     .param("grant_type", "password")
34                     .param("scope", "api")
35             )
36             .andExpect(status().isOk())
37             // .andDo(print())
38             .andReturn();
39         String contentAsString =
40             mvcResult.getResponse().getContentAsString();
41         ResultInfo resultInfo = (ResultInfo)
42             JSONUtil.toBean(contentAsString, ResultInfo.class);
43         JSONObject result = (JSONObject) resultInfo.getData();
44         String token = result.getString("accessToken");
45         tokens.append(token).append("\r\n");
46     }
47
48     Files.write(Paths.get("tokens.txt"), tokens.toString().getBytes());
49 }

```

导入测试计划

打开文件 线程组.jmx，进行测试

(1) 模拟5000个并发，2000个账号进行抢购

结果：

- 数据库t_seckill_vouchers表中的amount会为负数，表示超买了；
- t_vouchers_orders的订单如果超过100，说明卖多了

名称: 多人测试抢购代金券

注释:

- 在取样器错误后要执行的动作

继续 启动下一进程循环 停止线程 停止测试 立即停止测试

线程属性

线程数: 5000

Ramp-Up时间 (秒): 1

循环次数 永远 1

Same user on each iteration

延迟创建线程直到需要

调度器

持续时间 (秒)

启动延迟 (秒)

对象 t_seckill_vouchers @db_imoooc (local...) t_voucher_orders @db_imoooc (local...)

开始事务 文本 筛选 排序 导入 导出

id	fk_voucher_id	amount	start_time	end_time	is_valid	create_date	update_date
1		1 -123	2020-11-16 11:00:00	2020-12-16 11:00:00	1	2020-11-16 11:00:00	(2020-11-16 11:00:00)

对象 t_seckill_vouchers @db_imoooc (local...) t_voucher_orders @db_imoooc (local...)

开始事务 文本 筛选 排序 导入 导出

id	order_no	fk_voucher_id	fk_diner_id	qrcode	payment	status	fk_seckill_id	order_type	create_time
1	13281848867	1	8 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
2	13281848867	1	7 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
3	13281848868	1	5 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
4	13281848869	1	17 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
5	13281848870	1	14 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
6	13281848870	1	10 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
7	13281848871	1	9 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
8	13281848873	1	15 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
9	13281848876	1	12 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
10	13281848879	1	18 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
11	13281848879	1	29 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
12	13281848880	1	19 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
13	13281848881	1	21 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
14	13281848882	1	13 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
15	13281848883	1	34 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
16	13281848879	1	6 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
17	13281848884	1	37 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
18	13281848886	1	22 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00
19	13281848886	1	28 (Null)	(Null)	(Null)	0	1	1	2020-11-16 11:00:00

SELECT * FROM `db_imoooc`.`t_voucher_orders` LIMIT 0,1000

第 1 条记录 (共 223 条)

(2) 模拟某个用户多次抢购 (1个账号, 10000个并发)

结果后台会报错，同时订单表中会出现针对一个voucher一个用户多个订单

先TRUNCATE t_voucher_orders清空数据，修改t_seckill_vouchers amount为100.

	id	order_no	fk_voucher_id	fk_diner_id	qrcode	payment	status	fk_seckill_id	order_type	create
▶	1	13281865619	1		5 (Null)	(Null)	0	1	1	2020-
	2	13281865619	1		5 (Null)	(Null)	0	1	1	2020-
	3	13281865619	1		5 (Null)	(Null)	0	1	1	2020-
	4	13281865620	1		5 (Null)	(Null)	0	1	1	2020-
	5	13281865618	1		5 (Null)	(Null)	0	1	1	2020-
	6	13281865621	1		5 (Null)	(Null)	0	1	1	2020-
	7	13281865625	1		5 (Null)	(Null)	0	1	1	2020-
	8	13281865628	1		5 (Null)	(Null)	0	1	1	2020-
	9	13281865629	1		5 (Null)	(Null)	0	1	1	2020-
	10	13281865630	1		5 (Null)	(Null)	0	1	1	2020-
	11	13281865631	1		5 (Null)	(Null)	0	1	1	2020-
	12	13281865632	1		5 (Null)	(Null)	0	1	1	2020-

6. Redis防止超卖

6.1. 解决思路

将活动写入Redis中，通过Redis自减命令扣除库存

RedisKeyConstant

```

1 package com.imooc.commons.constant;
2
3 import lombok.Getter;
4

```

```

5  @Getter
6  public enum RedisKeyConstant {
7
8      verify_code("verify_code:", "验证码"),
9      seckill_vouchers("seckill_vouchers:", "秒杀券的key");
10
11     private String key;
12     private String desc;
13
14     RedisKeyConstant(String key, String desc) {
15         this.key = key;
16         this.desc = desc;
17     }
18
19 }
20

```

6.2. Redis配置类

```

1 package com.imooc.seckill.config;
2
3 import com.fasterxml.jackson.annotation.JsonAutoDetect;
4 import com.fasterxml.jackson.annotation.PropertyAccessor;
5 import com.fasterxml.jackson.databind.ObjectMapper;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8 import org.springframework.data.redis.connection.RedisConnectionFactory;
9 import org.springframework.data.redis.core.RedisTemplate;
10 import
11 org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
12 import org.springframework.data.redis.serializer.StringRedisSerializer;
13
14 @Configuration
15 public class RedisTemplateConfiguration {
16
17     /**
18      * redisTemplate 序列化使用的jdkSerializable，存储二进制字节码，所以自定义序列
19      * 化类
20      *
21      * @param redisConnectionFactory
22      * @return
23      */
24     @Bean
25     public RedisTemplate<Object, Object>
26     redisTemplate(RedisConnectionFactory redisConnectionFactory) {
27         RedisTemplate<Object, Object> redisTemplate = new RedisTemplate<>();
28         redisTemplate.setConnectionFactory(redisConnectionFactory);
29
30         // 使用Jackson2JsonRedisSerialize 替换默认序列化
31         Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
32         Jackson2JsonRedisSerializer(Object.class);
33
34         ObjectMapper objectMapper = new ObjectMapper();
35         objectMapper.setVisibility(PropertyAccessor.ALL,
36         JsonAutoDetect.Visibility.ANY);
37         jackson2JsonRedisSerializer.setObjectMapper(objectMapper);
38
39     }
40
41 }
42

```

```

34     // 设置key和value的序列化规则
35     redisTemplate.setValueSerializer(jackson2JsonRedisSerializer);
36     redisTemplate.setKeySerializer(new StringRedisSerializer());
37
38     redisTemplate.setHashKeySerializer(new StringRedisSerializer());
39     redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer);
40
41     redisTemplate.afterPropertiesSet();
42     return redisTemplate;
43 }
44
45 }
46

```

6.3. 修改添加活动的业务

在SeckillVoucherService中，修改addSeckillVouchers()方法，将数据以Hash Map的方式存入Redis中

```

1 /**
2 * 添加需要抢购的代金券
3 *
4 * @param seckillVouchers
5 */
6 @Transactional(rollbackFor = Exception.class)
7 public void addSeckillVouchers(SeckillVouchers seckillVouchers) {
8     // 非空校验
9     AssertUtil.isTrue(seckillVouchers.getFkVoucherId() == null, "请选择需要抢
10    购的代金券");
11    AssertUtil.isTrue(seckillVouchers.getAmount() == 0, "请输入抢购总数量");
12
13    Date now = new Date();
14    AssertUtil.isNotNull(seckillVouchers.getStartTime(), "请输入开始时间");
15    // 生产环境下面一行代码需放行，这里注释方便测试
16    // AssertUtil.isTrue(now.after(seckillVouchers.getStartTime()), "开始时间
17    不能早于当前时间");
18
19
20    AssertUtil.isTrue(seckillVouchers.getStartTime().after(seckillVouchers.getE
21    ndTime()), "开始时间不能晚于结束时间");
22
23    // -----注释原始的 关系型数据库 的流程-----
24    // 验证数据库中是否已经存在该券的秒杀活动
25    // SeckillVouchers seckillVouchersFromDb =
26    seckillVouchersMapper.selectVoucher(seckillVouchers.getFkVoucherId());
27    // AssertUtil.isTrue(seckillVouchersFromDb != null, "该券已经拥有了抢购活
28    动");
29    // 插入数据库
30    // seckillVouchersMapper.save(seckillVouchers);
31
32    // -----采用 Redis -----
33    String redisKey = RedisKeyConstant.seckill_vouchers.getKey() +
34    seckillVouchers.getFkVoucherId();
35    // 验证 Redis 中是否已经存在该券的秒杀活动

```

```

32     Map<String, Object> seckillVoucherMaps =
33         redisTemplate.opsForHash().entries(redisKey);
34     AssertUtil.isTrue(!seckillVoucherMaps.isEmpty() && (int)
35         seckillVoucherMaps.get("amount") > 0,
36             "该券已经拥有了抢购活动");
37
38     // 将数量同步到 Redis
39     seckillVouchers.setIsValid(1);
40     seckillVouchers.setCreateDate(now);
41     seckillVouchers.setUpdateDate(now);
42     seckillVoucherMaps = BeanUtil.beanToMap(seckillVouchers);
43     redisTemplate.opsForHash().putAll(redisKey, seckillVoucherMaps);
44 }

```

6.4. 修改抢购业务

```

1 /**
2 * 抢购代金券
3 *
4 * @param voucherId 代金券 ID
5 * @param accessToken 登录token
6 * @Param path 访问路径
7 */
8 @Transactional(rollbackFor = Exception.class)
9 public ResultInfo doSeckill(Integer voucherId, String accessToken, String
path) {
10     // 基本参数校验
11     AssertUtil.isTrue(voucherId == null || voucherId < 0, "请选择需要抢购的代金
券");
12     AssertUtil.isNotEmpty(accessToken, "请登录");
13
14     // -----注释原始的走 关系型数据库 的流程-----
15     // 判断此代金券是否加入抢购
16     // SeckillVouchers seckillVouchers =
17     seckillVouchersMapper.selectVoucher(voucherId);
18     // AssertUtil.isTrue(seckillVouchers == null, "该代金券并未有抢购活动");
19
20     // -----采用 Redis 解决问题-----
21     String redisKey = RedisKeyConstant.seckill_vouchers.getKey() +
22     voucherId;
23     Map<String, Object> seckillVoucherMaps =
24     redisTemplate.opsForHash().entries(redisKey);
25     SeckillVouchers seckillVouchers = BeanUtil.mapToBean(seckillVoucherMaps,
26     SeckillVouchers.class, true, null);
27
28     // 判断是否有效
29     AssertUtil.isTrue(seckillVouchers.getIsValid() == 0, "该活动已结束");
30     // 判断是否开始、结束
31     Date now = new Date();
32     AssertUtil.isTrue(now.before(seckillVouchers.getStartTime()), "该抢购还未
开始");
33     AssertUtil.isTrue(now.after(seckillVouchers.getEndTime()), "该抢购已结
束");
34
35     // 判断是否卖完通过 Lua 脚本扣库存时判断
36     //AssertUtil.isTrue(seckillVouchers.getAmount() < 1, "该券已经卖完了");

```

```

33     // 获取登录用户信息
34     String url = oauthServerName + "user/me?access_token={accessToken}";
35     ResultInfo resultInfo = restTemplate.getForObject(url, ResultInfo.class,
36             accessToken);
37     if (resultInfo.getCode() != ApiConstant.SUCCESS_CODE) {
38         resultInfo.setPath(path);
39         return resultInfo;
40     }
41     // 这里的data是一个LinkedHashMap, SignInDinerInfo
42     SignInDinerInfo dinerInfo = BeanUtil.fillBeanWithMap((LinkedHashMap)
43             resultInfo.getData(),
44             new SignInDinerInfo(), false);
45     // 判断登录用户是否已抢到(一个用户针对这次活动只能买一次)
46     VoucherOrders order =
47             voucherOrdersMapper.findDinerOrder(dinerInfo.getId(),
48                     seckillVouchers.getId());
49     AssertUtil.isTrue(order != null, "该用户已抢到该代金券，无需再抢");
50
51     // -----注释原始的走 关系型数据库 的流程-----
52     // 扣库存
53     // int count =
54     seckillVouchersMapper.stockDecrease(seckillVouchers.getId());
55     // AssertUtil.isTrue(count == 0, "该券已经卖完了");
56
57     // -----采用 Redis 解决问题-----
58     // 扣库存
59     long count = redisTemplate.opsForHash().increment(redisKey, "amount",
60             -1);
61     AssertUtil.isTrue(count < 0, "该券已经卖完了");
62
63     // 下单
64     VoucherOrders voucherOrders = new VoucherOrders();
65     voucherOrders.setFkDinerId(dinerInfo.getId());
66     // Redis 中不需要维护外键信息
67     // voucherOrders.setFkSeckillId(seckillVouchers.getId());
68     voucherOrders.setFkVoucherId(seckillVouchers.getFkVoucherId());
69     String orderNo = IdUtil.getSnowflake(1, 1).nextIdStr();
70     voucherOrders.setOrderNo(orderNo);
71     voucherOrders.setOrderType(1);
72     voucherOrders.setStatus(0);
73     count = voucherOrdersMapper.save(voucherOrders);
74     AssertUtil.isTrue(count == 0, "用户抢购失败");
75
76     return ResultInfoUtil.buildSuccess(path, "抢购成功");
77 }

```

6.5. 测试

问题一：多扣库存问题

192.168.10.101

db0 (1)

seckill_vouchers (1)

seckill_vouchers:1

row	key	value
1	fkVoucherId	1
2	amount	-4567
3	startTime	1605497400000
4	endTime	1608089400000
5	id	
6	createDate	1605502444373
7	updateDate	1605502444373

对象 t_seckill_vouchers @db_imoo (lo... t_voucher_orders @db_imoo (lo... * 无标题 - 查询

开始事务 文本 筛选 排序 导入 导出

id	order_no	fk_voucher_id	fk_diner_id	qrcode	payment	status	fk_seckill_id	order_type	create
1	13282053129	1	13 (Null)	(Null)	(Null)	0	0	1	2020-
2	13282053129	1	79 (Null)	(Null)	(Null)	0	0	1	2020-
3	13282053129	1	61 (Null)	(Null)	(Null)	0	0	1	2020-
4	13282053129	1	165 (Null)	(Null)	(Null)	0	0	1	2020-
5	13282053129	1	472 (Null)	(Null)	(Null)	0	0	1	2020-
6	13282053129	1	467 (Null)	(Null)	(Null)	0	0	1	2020-
7	13282053129	1	75 (Null)	(Null)	(Null)	0	0	1	2020-
8	13282053129	1	98 (Null)	(Null)	(Null)	0	0	1	2020-
9	13282053129	1	99 (Null)	(Null)	(Null)	0	0	1	2020-
10	13282053129	1	12 (Null)	(Null)	(Null)	0	0	1	2020-
11	13282053129	1	85 (Null)	(Null)	(Null)	0	0	1	2020-
12	13282053129	1	111 (Null)	(Null)	(Null)	0	0	1	2020-
13	13282053129	1	131 (Null)	(Null)	(Null)	0	0	1	2020-
14	13282053129	1	57 (Null)	(Null)	(Null)	0	0	1	2020-
15	13282053129	1	44 (Null)	(Null)	(Null)	0	0	1	2020-
16	13282053129	1	60 (Null)	(Null)	(Null)	0	0	1	2020-
17	13282053129	1	166 (Null)	(Null)	(Null)	0	0	1	2020-
18	13282053129	1	50 (Null)	(Null)	(Null)	0	0	1	2020-
19	13282053129	1	490 (Null)	(Null)	(Null)	0	0	1	2020-

SELECT * FROM `db_imoo`.`t_voucher_orders` LIMIT 0,1000 第 1 条记录 (共 100 条)

分析：因为Redis在扣除库存时抛出了“该券已经卖完”的异常，导致后续代码不再执行，所以订单是不符合逻辑的。

解决：那可能大多数人都会想到，直接把Redis扣库存的代码放在下单后执行不就可以了，我们来试一下。

问题二：超卖及多扣库存问题

192.168.10.101

db0 (1)

seckill_vouchers (1)

seckill_vouchers:1

row	key	value
1	fkVoucherId	1
2	amount	-4782
3	startTime	1605497400000
4	endTime	1608089400000
5	id	
6	createDate	1605502444373
7	updateDate	1605502444373

row	key	value
1	fkVoucherId	1
2	amount	-4782
3	startTime	1605497400000
4	endTime	1608089400000
5	id	
6	createDate	1605502444373
7	updateDate	1605502444373
...		

分析：虽然Redis在扣除库存时抛出了“该券已经卖完”的异常，但是由于方法没有事务的异常回滚处理，订单也出现了超卖问题。

解决：添加事务，我们再来试一试。

问题三：多扣库存问题

id	order_no	fk_voucher_id	fk_diner_id	qrCode	payment	status	fk_seckill_id	order_type	create_time
1	13282084106	1	188	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
2	13282084106	1	99	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
3	13282084106	1	42	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
4	13282084106	1	98	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
5	13282084106	1	52	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
6	13282084106	1	50	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
7	13282084106	1	165	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
8	13282084106	1	112	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
9	13282084106	1	103	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
10	13282084106	1	131	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
11	13282084111	1	185	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
12	13282084111	1	8	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
13	13282084111	1	114	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
14	13282084111	1	53	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
15	13282084111	1	128	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
16	13282084111	1	130	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
17	13282084111	1	48	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
18	13282084112	1	172	(Null)	(Null)	0	0	1	2020-01-13 20:20:46
19	13282084112	1	109	(Null)	(Null)	0	0	1	2020-01-13 20:20:46

row	key	value
1	fkVoucherId	1
2	amount	-4560
3	startTime	1605497400000
4	endTime	1608089400000
5	id	
6	createDate	1605502444373
7	updateDate	1605502444373
...		

分析：我们发现，又回到问题一的样子了。此时因为Redis在扣除库存时抛出了“该券已经卖完了”的异常，由于方法有事务的异常回滚处理，所以订单是符合逻辑的，但是Redis却还在扣除库存。因为Redis这里实际上是一个查询库存再扣除库存的操作，并发场景下仍然会出现问题，我们只需要保证两个操作在同一个线程中执行即可，也就是保证它的原子性。

解决：采用Lua脚本

6.6. Lua脚本

在减库存时，使用Lua脚本操作了Redis，因为减库存时，我们需要判断库存够不够，然后才能减掉，这里有两个操作，如果分开执行，那么有可能会出现错误（因为客户端是多线程），因此我们采用Lua脚本将两步操作放在一起同时在Redis中执行（Redis是单线程操作，故不会出现安全问题）。

将stock.lua脚本放入resources文件夹下：

```
1 if (redis.call('hexists', KEYS[1], KEYS[2]) == 1) then
2     local stock = tonumber(redis.call('hget', KEYS[1], KEYS[2]));
3     if (stock > 0) then
4         redis.call('hincrby', KEYS[1], KEYS[2], -1);
5         return stock;
6     end;
7     return 0;
8 end;
9
```

在Redis配置类中添加以下代码：

```
1 @Bean
2 public DefaultRedisScript<Long> stockScript() {
3     DefaultRedisScript<Long> redisscript = new DefaultRedisScript<>();
4     //放在和application.yml 同层目录下
5     redisscript.setLocation(new ClassPathResource("stock.lua"));
6     redisscript.setResultType(Long.class);
7     return redisscript;
8 }
9
```

测试结果已OK

t_voucher_orders @db_imoooc (lo... * 无标题 - 查询)										
开始任务 文本 筛选 排序 导入 导出										
id	order_no	fk_voucher_id	fk_diner_id	qrcode	payment	status	fk_seckill_id	order_type	create	update
1	13282139411	1	19	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
2	13282139411	1	109	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
3	13282139411	1	92	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
4	13282139411	1	940	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
5	13282139411	1	139	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
6	13282139411	1	59	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
7	13282139411	1	31	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
8	13282139411	1	13	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
9	13282139411	1	134	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
10	13282139411	1	94	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
11	13282139418	1	86	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
12	13282139418	1	78	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
13	13282139418	1	43	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
14	13282139418	1	63	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
15	13282139419	1	81	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
16	13282139419	1	110	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
17	13282139419	1	947	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
18	13282139419	1	152	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21
19	13282139419	1	122	(Null)	(Null)	0	0	1	2020-01-13 10:28:21	2020-01-13 10:28:21

row	key	value
1	fkVoucherId	1
2	amount	0
3	startTime	1605497400000
4	endTime	1608089400000
5	id	
6	createDate	1605502444373
7	updateDate	1605502444373

至此，超卖的问题已经解决了，但是运行某个用户发起多个抢购请求测试计划仍然会出错，也就是会出现一人购买多份的情况，下面我们来解决限制一人一单的问题。

7. Redis限制一人一单

7.1. 解决思路

采用Redis分布式锁限制食客

7.1.1. 锁

锁是一种保护机制，在多线程的情况下，保证数据操作的一致性

- Java中，我们通常使用的synchronized或者Lock都是线程锁，对同一个JVM进程内的多个线程有效。因为锁的本质是内存中存放的一个标记，记录获取锁的线程是谁，这个标记对每个线程都可见。比如我们启动的多个秒杀服务，就是多个JVM，内存中的锁显然是不共享的，每个JVM进程都有自己的锁，自然无法保证线程的互斥了，这个时候我们就需要分布式锁了。

7.1.2. 分布式锁

分布式锁有三种：

- 基于数据库
- 基于Zookeeper调度中心
- 基于Redis

• 分布式锁条件

实现分布式锁要满足以下三点：

- 多进程可见
- 互斥
- 可重入

7.1.3. Redis实现分布式锁

7.1.3.1. 多进程可见

Redis本身就是基于JVM之外的，因此满足多进程可见的要求。

7.1.3.2. 互斥

解决互斥

同一时间只能有一个进程获取锁标记，可以通过redis的setnx实现。

```
1 # 第一次设置lock, 成功返回1
2 127.0.0.1:0>setnx lock 123
3 "1"
4 # 如果存在, 再次设置会返回0
5 127.0.0.1:0>setnx lock 123
6 "0"
7 # 获取lock
8 127.0.0.1:0>get lock
9 "123"
10
```

✓ 解决死锁

```
1 SET KEY VALUE EX [seconds] PX [milliseconds] NX XX
2
3 # EX seconds - 设置键key的过期时间, 单位时秒
4 # PX milliseconds - 设置键key的过期时间, 单位时毫秒
5 # NX - 只有键key不存在的时候才会设置key的值
6 # XX - 只有键key存在的时候才会设置key的值
7
8 #例如: set lock 123 EX 60 NX==setnx lock 123 +expire lock 60
```

因此`set lock 123 EX 60 NX==setnx lock 123 +expire lock 60`, 而且`set`是原子操作, 因此如果使用最简单的Redis分布式锁的话, 就可以使用`set`指令。

代码如下:

```
1 package com.imooc;
2
3 import redis.clients.jedis.Jedis;
4 import redis.clients.jedis.params.SetParams;
5
6 public class RedisLock {
7
8     private static final String LOCK_SUCCESS = "OK";
9     private static final long UNLOCK_SUCCESS = 1L;
10
11     /**
12      * 尝试获取分布式锁
13      * @param jedis Redis客户端
14      * @param lockKey 锁
15      * @param value 锁的值
16      * @param expireTime 超期时间
17      * @return 是否获取成功
18      */
19     public static boolean tryLock(Jedis jedis, String lockKey,
20                                     String value, int expireTime) {
21         while(true) {
22             // set key value ex seconds nx(只有键不存在的时候才会设置key)
23             String result = jedis.set(lockKey, value,
24                                       SetParams.setParams().ex(expireTime).nx());
25             if (LOCK_SUCCESS.equals(result)) {
26                 return true;
27             }
28         }
29     }
30 }
```

```
30
31     /**
32      * 释放分布式锁
33      * @param jedis Redis客户端
34      * @param lockKey 锁
35      * @return 是否释放成功
36     */
37    public static boolean unlock(Jedis jedis, String lockKey) {
38        Long result = jedis.del(lockKey);
39        if (UNLOCK_SUCCESS == result) {
40            return true;
41        }
42        return false;
43    }
44
45 }
```

测试代码：

```
1 package com.imooc;
2
3 import redis.clients.jedis.Jedis;
4 import redis.clients.jedis.JedisPool;
5 import redis.clients.jedis.JedisPoolConfig;
6 import redis.clients.jedis.params.SetParams;
7
8 import java.util.UUID;
9
10 public class RedisLockTest {
11
12     private int count = 0;
13     private String lockKey = "lock";
14
15     private void call(Jedis jedis) {
16
17         // 加锁
18         boolean locked = RedisLock.tryLock(jedis, lockKey,
19                 UUID.randomUUID().toString(), 60);
20         try {
21             if (locked) {
22                 for (int i = 0; i < 500; i++) {
23                     count++;
24                 }
25             }
26         } catch (Exception e) {
27             e.printStackTrace();
28         } finally {
29             RedisLock.unlock(jedis, lockKey);
30         }
31     }
32
33     public static void main(String[] args) throws Exception {
34         RedisLockTest redisLockTest = new RedisLockTest();
35         JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
36         jedisPoolConfig.setMinIdle(1);
37         jedisPoolConfig.setMaxTotal(5);
38         JedisPool jedisPool = new JedisPool(jedisPoolConfig, "127.0.0.1",
```

```

39             6379, 1000);
40
41         Thread t1 = new Thread(() ->
42             redisLockTest.call(jedisPool.getResource());
43         Thread t2 = new Thread(() ->
44             redisLockTest.call(jedisPool.getResource());
45         t1.start();
46         t2.start();
47         t1.join();
48         t2.join();
49         System.out.println(redisLockTest.count);
50     }
51 }
```

□ 释放锁时BUG修复

为了避免删除别的进程产生的锁，我们可以在set锁时，存入当前线程的唯一标识，在删除所之前判断里面的线程标识与自己的表示是否一致，如果不一致就不允许删除。

调整代码：

```

1 package com.imooc;
2
3 import redis.clients.jedis.Jedis;
4 import redis.clients.jedis.params.SetParams;
5
6 public class RedisLock02 {
7
8     private static final String LOCK_SUCCESS = "OK";
9     private static final Long UNLOCK_SUCCESS = 1L;
10
11    /**
12     * 尝试获取分布式锁
13     * @param jedis Redis客户端
14     * @param lockKey 锁
15     * @param requestId 锁的值
16     * @param expireTime 超期时间
17     * @return 是否获取成功
18     */
19    public static boolean tryLock(Jedis jedis, String lockKey,
20                                  String requestId, int expireTime) {
21
22        while(true) {
23            // set key value ex seconds nx(只有键不存在的时候才会设置key)
24            String result = jedis.set(lockKey, requestId,
25                                      SetParams.setParams().ex(expireTime).nx());
26            if (LOCK_SUCCESS.equals(result)) {
27                return true;
28            }
29        }
30    }
31
32    /**
33     * 释放分布式锁
34     * @param jedis Redis客户端
35     */
```



```

41     Thread t1 = new Thread(() ->
42         redisLockTest.call(jedisPool.getResource());
43     Thread t2 = new Thread(() ->
44         redisLockTest.call(jedisPool.getResource());
45     t1.start();
46     t2.start();
47     t1.join();
48     t2.join();
49     System.out.println(redisLockTest.count);
50 }
51
52 }
```

按照以上方式实现分布式锁之后，就可以解决大部分问题了。但是仍然有些场景是不满足的，例如一个方法获取到锁之后，可能在方法内掉这个方法时就获取不到锁了，这个时候我们就需要把锁改进成**可重入锁**了。

7.1.3.3. 可重入锁

可重入锁，指的是以线程为单位，当一个线程获取对象锁之后，这个线程可以再次获取本对象上的锁，而其他的线程是不可以的。可重入锁的意义在于防止死锁。

实现原理：

代码演示：

不可重入锁

```

1 public class Lock {
2
3     private boolean isLocked = false;
4
5     public synchronized void lock() {
6         while (isLocked) {
7             try {
8                 wait();
9             } catch (InterruptedException e) {
10                 e.printStackTrace();
11             }
12         }
13         isLocked = true;
14     }
15
16     public synchronized void unlock() {
17         isLocked = false;
18         notify();
19     }
20 }
```

使用该锁：

```

1 package com.imooc.reentrant;
2
3 public class UnReentrantLockDemo {
```

```

4
5     private int count = 0;
6     private Lock lock = new Lock();
7
8     private void call() {
9         lock.lock();
10        inc();
11        lock.unlock();
12    }
13
14     private void inc() {
15         lock.lock();
16         for (int i = 0; i < 500; i++) {
17             count++;
18         }
19         lock.unlock();
20     }
21
22     public static void main(String[] args) throws Exception {
23         UnReentrantLockDemo unReentrantLockDemo = new UnReentrantLockDemo();
24         Thread t1 = new Thread(() -> unReentrantLockDemo.call());
25         Thread t2 = new Thread(() -> unReentrantLockDemo.call());
26         t1.start();
27         t2.start();
28         t1.join();
29         t2.join();
30         System.out.println(unReentrantLockDemo.count);
31     }
32 }
33

```

当前线程执行call()方法首先获取lock，接下来执行inc()方法就无法执行inc()中的逻辑，必须先释放锁。这个例子很好说明了不可重入锁。

□ 可重入锁

```

1 /**
2 * 为每个锁关联一个请求计数器和一个占有它的线程。
3 * 当计数为0时，认为锁是未被占有的；
4 * 线程请求一个未被占有的锁时，JVM将记录锁的占有者，并且将请求计数器置为1。
5 */
6 public class ReentrantLock {
7     boolean isLocked = false;
8     Thread lockBy = null; // 独占线程
9     int lockedCount = 0; // 计数器
10
11    public synchronized void lock() throws InterruptedException {
12        Thread thread = Thread.currentThread();
13        while (isLocked && lockBy != thread) { // 判断加锁，而且线程不是当前线程
14            wait();
15        }
16        isLocked = true;
17        lockedCount++; // 计数器 +1
18        lockBy = thread;
19
20    }
21

```

```
22     public synchronized void unlock() {
23         if (Thread.currentThread() == this.lockBy) { // 判断是否是当前线程
24             lockedCount--;
25             if (lockedCount == 0) { // 计数器为0时，释放锁
26                 isLocked = false;
27                 notify();
28             }
29         }
30     }
31 }
32 }
```

测试可重入锁

```
1  public class ReentrantLockDemo {
2
3      private int count = 0;
4      private ReentrantLock lock = new ReentrantLock();
5
6      private void call() {
7          try {
8              lock.lock();
9          } catch (InterruptedException e) {
10              e.printStackTrace();
11          }
12          inc();
13          lock.unlock();
14      }
15
16      private void inc() {
17          try {
18              lock.lock();
19          } catch (InterruptedException e) {
20              e.printStackTrace();
21          }
22          for (int i = 0; i < 500; i++) {
23              count++;
24          }
25          lock.unlock();
26      }
27
28      public static void main(String[] args) throws Exception {
29          ReentrantLockDemo demo = new ReentrantLockDemo();
30          Thread t1 = new Thread(() -> demo.call());
31          Thread t2 = new Thread(() -> demo.call());
32          t1.start();
33          t2.start();
34          t1.join();
35          t2.join();
36          System.out.println(demo.count);
37      }
38  }
```

7.2. Redis可重入锁

7.2.1. 设计思路

```
1 假设锁的key为“ lock ”, hashKey是当前线程的id: “ threadId ”, 锁自动释放时间假设为20
2 获取锁的步骤:
3   1、判断lock是否存在 EXISTS lock
4     2、不存在, 则自己获取锁, 记录重入层数为1.
5       2、存在, 说明有人获取锁了, 下面判断是不是自己的锁, 即判断当前线程id作为hashKey是否存在: HEXISTS lock threadId
6         3、不存在, 说明锁已经有了, 且不是自己获取的, 锁获取失败.
7           3、存在, 说明是自己获取的锁, 重入次数+1: HINCRBY lock threadId 1 , 最后更新锁自动释放时间, EXPIRE lock 20
8
9 释放锁的步骤:
10    1、判断当前线程id作为hashKey是否存在: HEXISTS lock threadId
11      2、不存在, 说明锁已经失效, 不用管了
12        2、存在, 说明锁还在, 重入次数减1: HINCRBY lock threadId -1 ,
13          3、获取新的重入次数, 判断重入次数是否为0, 为0说明锁全部释放, 删除key: DEL lock
14
```

因此, 存储在锁中的信息就必须包含: key、线程标识、重入次数。

- 不能再使用简单的key-value结构, 推荐使用**Hash**结构
- 而且要让所有指令在同一个线程中操作, 那么使用**Lua脚本**

7.2.2. 设计lua脚本

lock.lua脚本

```
1 local key = KEYS[1]; -- 第1个参数, 锁的key
2 local threadId = ARGV[1]; -- 第2个参数, 线程唯一标识
3 local releaseTime = ARGV[2]; -- 第3个参数, 锁的自动释放时间
4
5 if(redis.call('exists', key) == 0) then -- 判断锁是否已存在
6   redis.call('hset', key, threadId, '1'); -- 不存在, 则获取锁
7   redis.call('expire', key, releaseTime); -- 设置有效期
8   return 1; -- 返回结果
9 end;
10
11 if(redis.call('hexists', key, threadId) == 1) then -- 锁已经存在, 判断threadId
12   是不是自己
13   redis.call('hincrby', key, threadId, '1'); -- 如果是自己, 则重入次数+1
14   redis.call('expire', key, releaseTime); -- 设置有效期
15   return 1; -- 返回结果
16 end;
17 return 0; -- 代码走到这里, 说明获取锁的不是自己, 获取锁失败
```

unlock.lua脚本

```

1 local key = KEYS[1]; -- 第1个参数,锁的key
2 local threadId = ARGV[1]; -- 第2个参数,线程唯一标识
3
4 if (redis.call('HEXISTS', key, threadId) == 0) then -- 判断当前锁是否还是被自己
持
5     return nil; -- 如果已经不是自己,则直接返回
6 end;
7 local count = redis.call('HINCRBY', key, threadId, -1); -- 是自己的锁,则重入次
数-1
8
9 if (count == 0) then -- 判断是否重入次数是否已经为0
10    redis.call('DEL', key); -- 等于0说明可以释放锁,直接删除
11    return nil;
12 end;
13

```

7.3. 在项目中集成

7.3.1. 编写RedisLock类

```

1 @Getter
2 @Setter
3 public class RedisLock {
4
5     private RedisTemplate redisTemplate;
6     private DefaultRedisScript<Long> lockScript;
7     private DefaultRedisScript<Object> unlockScript;
8
9     public RedisLock(RedisTemplate redisTemplate) {
10         this.redisTemplate = redisTemplate;
11         // 加载释放锁的脚本
12         this.lockScript = new DefaultRedisScript<>();
13         this.lockScript.setScriptSource(new ResourceScriptSource(new
ClassPathResource("lock.lua")));
14         this.lockScript.setResultType(Long.class);
15         // 加载释放锁的脚本
16         this.unlockScript = new DefaultRedisScript<>();
17         this.unlockScript.setScriptSource(new ResourceScriptSource(new
ClassPathResource("unlock.lua")));
18     }
19
20     /**
21      * 获取锁
22      * @param lockName 锁名称
23      * @param releaseTime 超时时间(单位:秒)
24      * @return key 解锁标识
25      */
26     public String tryLock(String lockName, long releaseTime) {
27         // 存入的线程信息的前缀,防止与其它JVM中线程信息冲突
28         String key = UUID.randomUUID().toString();
29
30         // 执行脚本
31         Long result = (Long)redisTemplate.execute(
32             lockScript,
33             Collections.singletonList(lockName),
34             key + Thread.currentThread().getId(), releaseTime);

```

```

35     // 判断结果
36     if(result != null && result.intValue() == 1) {
37         return key;
38     }else {
39         return null;
40     }
41 }
42 /**
43 * 释放锁
44 * @param lockName 锁名称
45 * @param key 解锁标识
46 */
47 public void unlock(String lockName, String key) {
48     // 执行脚本
49     redisTemplate.execute(
50         unlockScript,
51         Collections.singletonList(lockName),
52         key + Thread.currentThread().getId(), null);
53 }
54 }
55 }
56 }
```

7.3.2. 初始化Bean

```

1 @Configuration
2 public class RedisLockConfiguration {
3
4     @Resource
5     private RedisTemplate redisTemplate;
6
7     @Bean
8     public RedisLock redisLock() {
9         RedisLock redisLock = new RedisLock(redisTemplate);
10        return redisLock;
11    }
12 }
13 }
```

7.3.3. 分析业务修改Mapper和Service

Mapper

```

1 // 根据食客 ID 和代金券 ID 及订单状态查询代金券订单
2 @Select("select id, order_no, fk_voucher_id, fk_diner_id, qrcode, payment," +
3         " status, fk_seckill_id, order_type, create_date, update_date, " +
4         " is_valid from t_voucher_orders where fk_diner_id = #{dinerId} " +
5         " and fk_voucher_id = #{voucherId} and is_valid = 1 and status
6         between 0 and 1 ")
7 VoucherOrders findDinerOrder(@Param("dinerId") Integer dinerId,
8                               @Param("voucherId") Integer voucherId);
```

Service

```
1 // 判断登录用户是否已抢到(一个用户针对这次活动只能买一次)
2 VoucherOrders order = voucherOrdersMapper.findDinerOrder(dinerInfo.getId(),
3
4 seckillVouchers.getFkVoucherId());
5 AssertUtil.isTrue(order != null, "该用户已抢到该代金券, 无需再抢");
```

7.3.4. 修改SeckillService中的秒杀业务

```
1 @Resource
2 private RedisLock redisLock;
3
4 /**
5 * 抢购代金券
6 *
7 * @param voucherId 代金券 ID
8 * @param accessToken 登录token
9 * @Param path 访问路径
10 */
11 @Transactional(rollbackFor = Exception.class)
12 public ResultInfo doSeckill(Integer voucherId, String accessToken, String path) {
13     // 基本参数校验
14     AssertUtil.isTrue(voucherId == null || voucherId < 0, "请选择需要抢购的代
15     金券");
16     AssertUtil.isNotEmpty(accessToken, "请登录");
17
18     // -----注释原始的走 关系型数据库 的流程-----
19     // 判断此代金券是否加入抢购
20     // SeckillVouchers seckillVouchers =
21     // seckillVouchersMapper.selectVoucher(voucherId);
22     // AssertUtil.isTrue(seckillVouchers == null, "该代金券并未有抢购活动");
23
24     // -----采用 Redis 解决问题-----
25     String redisKey = RedisKeyConstant.seckill_vouchers.getKey() +
voucherId;
26     Map<String, Object> seckillVoucherMaps =
redisTemplate.opsForHash().entries(redisKey);
27     SeckillVouchers seckillVouchers =
28     BeanUtil.mapToBean(seckillVoucherMaps, SeckillVouchers.class, true, null);
29
30     // 判断是否有效
31     AssertUtil.isTrue(seckillVouchers.getIsValid() == 0, "该活动已结束");
32     // 判断是否开始、结束
33     Date now = new Date();
34     AssertUtil.isTrue(now.before(seckillVouchers.getStartTime()), "该抢购还
未开始");
35     AssertUtil.isTrue(now.after(seckillVouchers.getEndTime()), "该抢购已结
束");
36
37     // 判断是否卖完通过 Lua 脚本扣库存时判断
38     //AssertUtil.isTrue(seckillVouchers.getAmount() < 1, "该券已经卖完了");
39
40     // 获取登录用户信息
```

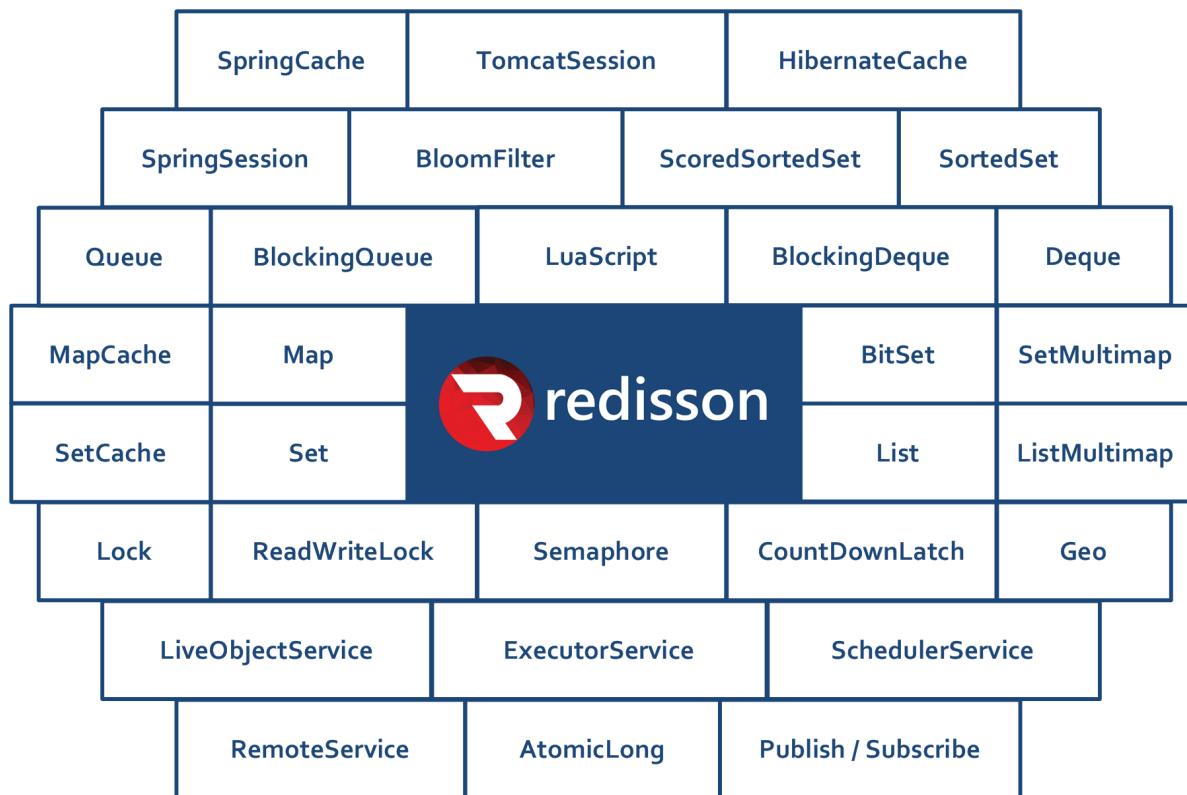
```
38     String url = oauthServerName + "user/me?access_token={accessToken}";
39     ResultInfo resultInfo = restTemplate.getForObject(url,
40             ResultInfo.class, accessToken);
41     if (resultInfo.getCode() != ApiConstant.SUCCESS_CODE) {
42         resultInfo.setPath(path);
43         return resultInfo;
44     }
45     // 这里的data是一个LinkedHashMap, SignInDinerInfo
46     SignInDinerInfo dinerInfo = BeanUtil.fillBeanwithMap((LinkedHashMap)
47             resultInfo.getData(),
48             new SignInDinerInfo(), false);
49     // 判断登录用户是否已抢到(一个用户针对这次活动只能买一次)
50     VoucherOrders order =
51             voucherOrdersMapper.findDinerOrder(dinerInfo.getId(),
52                     seckillVouchers.getFkVoucherId());
53     AssertUtil.isTrue(order != null, "该用户已抢到该代金券, 无需再抢");
54
55     // -----注释原始的走 关系型数据库 的流程-----
56     // 扣库存
57     // int count =
58     seckillVouchersMapper.stockDecrease(seckillVouchers.getId());
59     // AssertUtil.isTrue(count == 0, "该券已经卖完了");
60
61     // 使用 Redis 锁一个账号只能购买一次
62     String lockName = RedisKeyConstant.lock_key.getKey() +
63             dinerInfo.getId() + ":" + voucherId;
64     // 加锁
65     long expireTime = seckillVouchers.getEndTime().getTime() -
66             now.getTime();
67     String lockKey = redisLock.tryLock(lockName, expireTime);
68
69     try {
70         // 不为空意味着拿到锁了, 执行下单
71         if (StrUtil.isNotBlank(lockKey)) {
72             // 下单
73             VoucherOrders voucherOrders = new VoucherOrders();
74             voucherOrders.setFkDinerId(dinerInfo.getId());
75             // Redis 中不需要维护外键信息
76             // voucherOrders.setFkSeckillId(seckillVouchers.getId());
77             voucherOrders.setFkVoucherId(seckillVouchers.getFkVoucherId());
78             String orderNo = IdUtil.getSnowflake(1, 1).nextIdStr();
79             voucherOrders.setOrderNo(orderNo);
80             voucherOrders.setOrderType(1);
81             voucherOrders.setStatus(0);
82             long count = voucherOrdersMapper.save(voucherOrders);
83             AssertUtil.isTrue(count == 0, "用户抢购失败");
84
85             // -----采用 Redis 解问题-----
86             // 扣库存
87             // long count = redisTemplate.opsForHash().increment(redisKey,
88             "amount", -1);
89             // AssertUtil.isTrue(count < 0, "该券已经卖完了");
90
91             // -----采用 Redis + Lua 解问题-----
92             // 扣库存
93             List<String> keys = new ArrayList<>();
94             keys.add(redisKey);
95             keys.add("amount");
```

```

89             Long amount = (Long) redisTemplate.execute(redisScript, keys);
90             AssertUtil.isTrue(amount == null || amount < 1, "该券已经卖完了");
91         }
92     } catch (Exception e) {
93         // 手动回滚事物
94
95         TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
96         // 解锁
97         redisLock.unlock(clockName, lockKey);
98         if (e instanceof ParameterException) {
99             return ResultInfoUtil.buildError(0, "该券已经卖完了", path);
100        }
101    }
102
103    return ResultInfoUtil.buildSuccess(path, "抢购成功");
104

```

7.4. 引入Redission分布式锁



在这里我们使用它分布式锁的功能。

添加依赖

```

1 <dependency>
2   <groupId>org.redisson</groupId>
3   <artifactId>redisson-spring-boot-starter</artifactId>
4   <version>3.13.6</version>
5 </dependency>
6

```

引入对象

```
1 | @Resource  
2 | private RedissonClient redissonClient;
```

完成代码改造

```
1 | @Resource  
2 | private RedissonClient redissonClient;  
3 |  
4 | /**  
5 | * 抢购代金券  
6 | *  
7 | * @param voucherId 代金券 ID  
8 | * @param accessToken 登录token  
9 | * @Param path 访问路径  
10| */  
11| @Transactional(rollbackFor = Exception.class)  
12| public ResultInfo doSeckill(Integer voucherId, String accessToken, String  
path) {  
13|     // 基本参数校验  
14|     AssertUtil.isTrue(voucherId == null || voucherId < 0, "请选择需要抢购的代  
金券");  
15|     AssertUtil.isNotEmpty(accessToken, "请登录");  
16|  
17|     // -----注释原始的走 关系型数据库 的流程-----  
18|     // 判断此代金券是否加入抢购  
19|     // SeckillVouchers seckillVouchers =  
20|     seckillVouchersMapper.selectVoucher(voucherId);  
21|     // AssertUtil.isTrue(seckillVouchers == null, "该代金券并未有抢购活动");  
22|  
23|     // -----采用 Redis 解问题-----  
24|     String redisKey = RedisKeyConstant.seckill_vouchers.getKey() +  
voucherId;  
25|     Map<String, Object> seckillVoucherMaps =  
redisTemplate.opsForHash().entries(redisKey);  
26|     SeckillVouchers seckillVouchers =  
BeanUtil.mapToBean(seckillVoucherMaps, SeckillVouchers.class, true, null);  
27|  
28|     // 判断是否有效  
29|     AssertUtil.isTrue(seckillVouchers.getIsValid() == 0, "该活动已结束");  
30|     // 判断是否开始、结束  
31|     Date now = new Date();  
32|     AssertUtil.isTrue(now.before(seckillVouchers.getStartTime()), "该抢购还  
未开始");  
33|     AssertUtil.isTrue(now.after(seckillVouchers.getEndTime()), "该抢购已结  
束");  
34|  
35|     // 判断是否卖完通过 Lua 脚本扣库存时判断  
36|     //AssertUtil.isTrue(seckillVouchers.getAmount() < 1, "该券已经卖完了");  
37|  
38|     // 获取登录用户信息  
39|     String url = oauthServerName + "user/me?access_token={accessToken}";  
40|     ResultInfo resultInfo = restTemplate.getForObject(url,  
ResultInfo.class, accessToken);  
41|     if (resultInfo.getCode() != ApiConstant.SUCCESS_CODE) {  
         resultInfo.setPath(path);
```

```
42     return resultInfo;
43 }
44 // 这里的data是一个LinkedHashMap, SignInDinerInfo
45 SignInDinerInfo dinerInfo = BeanUtil.fillBeanWithMap((LinkedHashMap)
46 resultInfo.getData(),
47     new SignInDinerInfo(), false);
48 // 判断登录用户是否已抢到(一个用户针对这次活动只能买一次)
49 VoucherOrders order =
50 voucherOrdersMapper.findDinerOrder(dinerInfo.getId(),
51         seckillVouchers.getFkVoucherId());
52 AssertUtil.isTrue(order != null, "该用户已抢到该代金券, 无需再抢");
53
54 // -----注释原始的走 关系型数据库 的流程-----
55 // 扣库存
56 // int count =
57 seckillVouchersMapper.stockDecrease(seckillVouchers.getId());
58 // AssertUtil.isTrue(count == 0, "该券已经卖完了");
59
60 // 使用 Redis 锁一个账号只能购买一次
61 String lockName = RedisKeyConstant.lock_key.getKey() +
62     dinerInfo.getId() + ":" + voucherId;
63 // 加锁
64 long expireTime = seckillVouchers.getEndTime().getTime() -
65     now.getTime();
66 // 自定义 Redis 分布式锁
67 // String lockKey = redisLock.tryLock(lockName, expireTime);
68
69 // Redisson 分布式锁
70 RLock lock = redissonClient.getLock(lockName);
71
72 try {
73     // 不为空意味着拿到锁了, 执行下单
74     // 自定义 Redis 分布式锁处理
75     //if (StrUtil.isNotBlank(lockKey)) {
76
77     // Redisson 分布式锁处理
78     boolean isLocked = lock.tryLock(expireTime, TimeUnit.MILLISECONDS);
79     if (isLocked) {
80         // 下单
81         VoucherOrders voucherOrders = new VoucherOrders();
82         voucherOrders.setFkDinerId(dinerInfo.getId());
83         // Redis 中不需要维护外键信息
84         //voucherOrders.setFkSeckillId(seckillVouchers.getId());
85         voucherOrders.setFkVoucherId(seckillVouchers.getFkVoucherId());
86         String orderNo = IdUtil.getSnowflake(1, 1).nextIdStr();
87         voucherOrders.setOrderNo(orderNo);
88         voucherOrders.setOrderType(1);
89         voucherOrders.setStatus(0);
90         long count = voucherOrdersMapper.save(voucherOrders);
91         AssertUtil.isTrue(count == 0, "用户抢购失败");
92
93         // -----采用 Redis 解问题-----
94         // 扣库存
95         // long count = redisTemplate.opsForHash().increment(redisKey,
96         "amount", -1);
97         // AssertUtil.isTrue(count < 0, "该券已经卖完了");
98
99         // -----采用 Redis + Lua 解问题-----
```

```
94         // 扣库存
95         List<String> keys = new ArrayList<>();
96         keys.add(redisKey);
97         keys.add("amount");
98         Long amount = (Long) redisTemplate.execute(redisScript, keys);
99         AssertUtil.isTrue(amount == null || amount < 1, "该券已经卖完了");
100    }
101 } catch (Exception e) {
102     // 手动回滚事物
103
104     TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
105     // 自定义 Redis 解锁
106     // redisLock.unlock(lockName, lockKey);
107
108     // Redisson 解锁
109     lock.unlock();
110     if (e instanceof ParameterException) {
111         return ResultInfoUtil.buildError(0, "该券已经卖完了", path);
112     }
113
114     return ResultInfoUtil.buildSuccess(path, "抢购成功");
115 }
116 }
```