第2天-秒杀

学习目标

目标1：多线程下单

目标2：防止秒杀重复排队

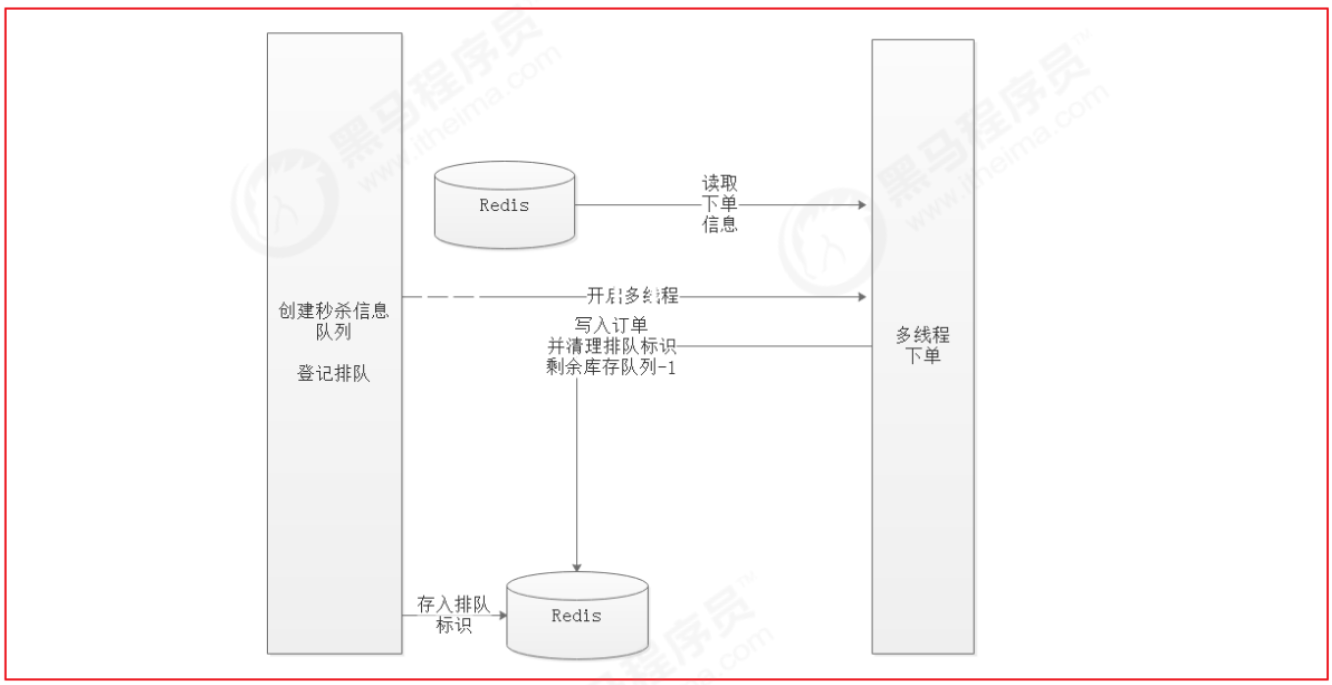
目标3：并发超卖问题解决

目标4：秒杀订单支付

目标5：超时支付订单库存回滚

第1章 多线程下单

1.1 实现思路分析



在审视秒杀中，操作一般都是比较复杂的，而且并发量特别高，比如，检查当前账号操作是否已经秒杀过该商品，检查该账号是否存在存在刷单行为，记录用户操作日志等。

下订单这里，我们一般采用多线程下单，但多线程中我们又需要保证用户抢单的公平性，也就是先抢先下单。我们可以这样实现，用户进入秒杀抢单，如果用户复合抢单资格，只需要记录用户抢单数据，存入队列，多线程从队列中进行消费即可，存入队列采用左压，多线程下单采用右取的方式。

1.2 Spring实现多线程

我们过去实现多线程的方式通常是继承Thread类或者实现Runnable 接口，这种方式实现起来比较麻烦。spring封装了Java的多线程的实现，你只需要关注于并发事物的流程以及一些并发负载量等特性。spring通过任务执行器TaskExecutor来实现多线程与并发编程。通常使用ThreadPoolTaskExecutor来实现一个基于线程池的TaskExecutor.

1.2.1 开启线程池

首先你要实现AsyncConfigurer 这个接口，目的是开启一个线程池 ,这个步骤我们可以基于spring的配置文件实现，修改qingcheng_service_seckill的applicationContext-timer.xml文件，代码如下：

```
<!--开启注解驱动-->
<task:annotation-driven executor="taskExecutor" scheduler="seckillScheduler"/>
<task:scheduler id="seckillScheduler" pool-size="10"/>

<!--
    线程池
-->
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <!--初始线程数量-->
    <property name="corePoolSize" value="10" />
    <!--最大线程数量-->
    <property name="maxPoolSize" value="100" />
    <!--最大队列数量-->
    <property name="queueCapacity" value="200" />
    <!--线程最大空闲时间-->
    <property name="keepAliveSeconds" value="3000" />
    <!--
        拒绝策略,当线程池中的线程被占用完了，没有剩余线程了，如果此时有新的任务要执行，该采取的策略
    -->
    <property name="rejectedExecutionHandler">
        <bean class="java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy" />
    </property>
</bean>
```

引用如下线程池

创建一个线程池

上图代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:task="http://www.springframework.org/schema/task"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/task
       http://www.springframework.org/schema/task/spring-task.xsd">

    <!--开启注解驱动-->
    <task:annotation-driven executor="taskExecutor" scheduler="seckillScheduler"/>
    <task:scheduler id="seckillScheduler" pool-size="10"/>

    <!--
        线程池
    -->
    <bean id="taskExecutor"
class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
        <!--初始线程数量-->
```

```
        <property name="corePoolSize" value="10" />
        <!--最大线程数量-->
        <property name="maxPoolSize" value="100" />
        <!--最大队列数量-->
        <property name="queueCapacity" value="200" />
        <!--线程最大空闲时间-->
        <property name="keepAliveSeconds" value="3000" />
        <!--
            拒绝策略,当线程池中的线程被占用完了，没有剩余线程了，如果此时有新的任务要执行，该采取的策略
        -->
        <property name="rejectedExecutionHandler">
            <bean class="java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy" />
        </property>
    </bean>
</beans>
```

1.2.2 异步执行声明

然后注入一个类，实现你的业务，并在你的Bean的方法中使用@Async注解来声明其是一个异步任务 ,例如，我们创建一个类com.qingcheng.task.MultiThreadingCreateOrder，在类里写一个方法createOrder,加上注解@Async,代码如下：

```
@Component
public class MultiThreadingCreateOrder {

    /***
     * 下单操作
     */
    @Async
    public void createOrder(){
        try {
            System.out.println("准备执行....");
            Thread.sleep( millis: 20000);
            System.out.println("开始执行....");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

上图代码如下：

```
@Component
public class MultiThreadingCreateOrder {

    /***
     * 多线程下单操作
     */
    @Async
    public void createOrder(){
        try {
            System.out.println("准备执行....");
            Thread.sleep(20000);
```

```
            System.out.println("开始执行....");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

1.2.3 异步调用

在每次创建订单的时候，我们调用上面异步方法，测试是否异步执行。

修改秒杀抢单SeckillOrderServiceImpl代码，注入MultiThreadingCreateOrder,并调用createOrder方法，代码如下：

```
@Service
public class SeckillOrderServiceImpl implements SeckillOrderService {

    @Autowired
    private MultiThreadingCreateOrder multiThreadingCreateOrder;

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private SeckillGoodsMapper seckillGoodsMapper;

    @Autowired
    private IdWorker idWorker;

    /****
     * 添加订单
     * @param id
     * @param time
     * @param username
     */
    @Override
    public Boolean add(Long id, String time, String username){
        //多线程操作
        multiThreadingCreateOrder.createOrder();

        //...略
        return true;
    }
}
```

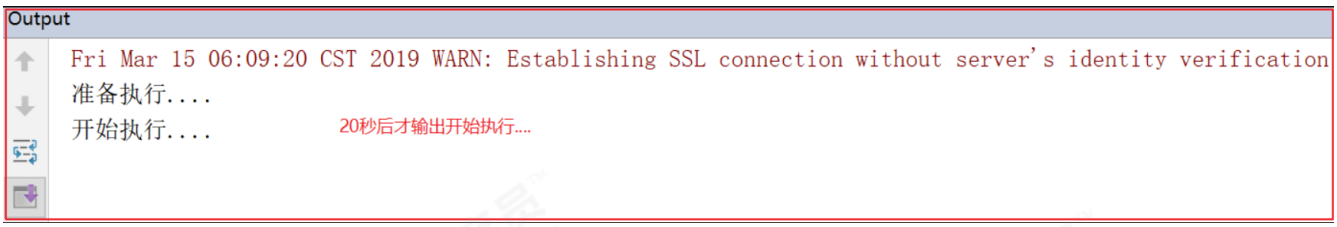
上图代码如下：

```
/****
 * 添加订单
 * @param id
 * @param time
 * @param username
 */
@Override
public Boolean add(Long id, String time, String username){
    //多线程操作
```

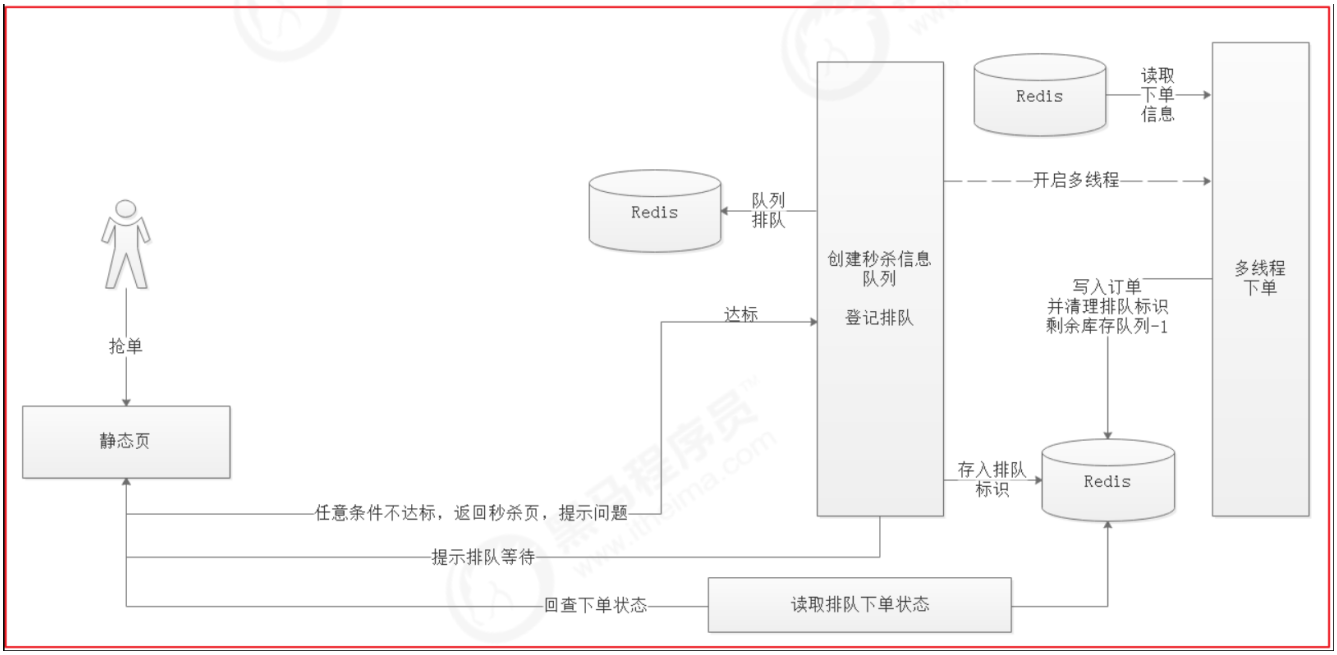
```
        multiThreadingCreateOrder.createOrder();

        //...略
        return true;
    }
```

测试效果如下：

20秒后输入剩下部分：



1.3 多线程抢单



用户每次下单的时候，我们都让他们先进行排队，然后采用多线程的方式创建订单，排队我们可以采用Redis的队列实现，多线程下单我们可以采用Spring的异步实现。

1.3.1 多线程下单

将之前下单的代码全部挪到多线程的方法中，com.qingcheng.service.impl.SeckillOrderServiceImpl类的方法值负责调用即可，代码如下：

```
@Service
public class SeckillOrderServiceImpl implements SeckillOrderService {

    @Autowired
    private MultiThreadingCreateOrder multiThreadingCreateOrder;

    /****
     * 添加订单
     * @param id
     * @param time
     * @param username
     */
    @Override
    public Boolean add(Long id, String time, String username){
        //多线程操作
        multiThreadingCreateOrder.createOrder();
        return true;
    }
}
```

多线程下单代码如下图：

```java
@Component
public class MultiThreadingCreateOrder {

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private SeckillGoodsMapper seckillGoodsMapper;

    @Autowired
    private IdWorker idWorker;

    /***
     * 多线程下单操作
     */
    @Async
    public void createOrder(){
        try {
            //时间区间
            String time = "20190509";
            //用户登录名
            String username="casuser";
            //用户抢购商品
            Long id = 1L;

            //获取商品数据
            SeckillGoods goods = (SeckillGoods) redisTemplate.boundHashOps( key: "SeckillGoods_" + time).get(id);

            //如果没有库存，则直接抛出异常
            if(goods==null || goods.getStockCount()<=0){
                throw new RuntimeException("已售罄!");
            }
            //如果有库存，则创建秒杀商品订单
            SeckillOrder seckillOrder = new SeckillOrder();
            seckillOrder.setId(idWorker.nextId());
            seckillOrder.setSeckillId(id);
            seckillOrder.setMoney(goods.getCostPrice());
            seckillOrder.setUserId(username);
            seckillOrder.setSellerId(goods.getSellerId());
            seckillOrder.setCreateTime(new Date());
            seckillOrder.setStatus("0");

            //将秒杀订单存入到Redis中
            redisTemplate.boundHashOps( key: "SeckillOrder").put(username,seckillOrder);

            //库存减少
            goods.setStockCount(goods.getStockCount()-1);

            //判断当前商品是否还有库存
            if(goods.getStockCount()<=0){
                //并且将商品数据同步到MySQL中
                seckillGoodsMapper.updateByPrimaryKeySelective(goods);
                //如果没有库存,则清空Redis缓存中该商品
                redisTemplate.boundHashOps( key: "SeckillGoods_" + time).delete(id);
            }else{
                //如果有库存，则直数据重置到Reids中
                redisTemplate.boundHashOps( key: "SeckillGoods_" + time).put(id,goods);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

先定义假数据，这里数据Reids中必须存在

```
}
```

上图代码如下：

```
@Component
public class MultiThreadingCreateOrder {

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private SeckillGoodsMapper seckillGoodsMapper;

    @Autowired
    private IdWorker idWorker;

    /***
     * 多线程下单操作
     */
    @Async
    public void createOrder(){
        try {
            //时间区间
            String time = "20190509";
            //用户登录名
            String username="casuser";
            //用户抢购商品
            Long id = 1L;

            //获取商品数据
            SeckillGoods goods = (SeckillGoods) redisTemplate.boundHashOps("SeckillGoods_"
+ time).get(id);

            //如果没有库存，则直接抛出异常
            if(goods==null || goods.getStockCount()<=0){
                throw new RuntimeException("已售罄!");
            }
            //如果有库存，则创建秒杀商品订单
            SeckillOrder seckillOrder = new SeckillOrder();
            seckillOrder.setId(idWorker.nextId());
            seckillOrder.setSeckillId(id);
            seckillOrder.setMoney(goods.getCostPrice());
            seckillOrder.setUserId(username);
            seckillOrder.setSellerId(goods.getSellerId());
            seckillOrder.setCreateTime(new Date());
            seckillOrder.setStatus("0");

            //将秒杀订单存入到Redis中
            redisTemplate.boundHashOps("SeckillOrder").put(username,seckillOrder);

            //库存减少
            goods.setStockCount(goods.getStockCount()-1);
```

```
            //判断当前商品是否还有库存
            if(goods.getStockCount()<=0){
                //并且将商品数据同步到MySQL中
                seckillGoodsMapper.updateByPrimaryKeySelective(goods);
                //如果没有库存,则清空Redis缓存中该商品
                redisTemplate.boundHashOps("SeckillGoods_" + time).delete(id);
            }else{
                //如果有库存，则直数据重置到Reids中
                redisTemplate.boundHashOps("SeckillGoods_" + time).put(id,goods);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

此时测试，是可以正常下单的，但是用户名和订单都写死了，此处需要继续优化。

1.3.2 排队下单

1.3.2.1 排队信息封装

用户每次下单的时候，我们可以创建一个队列进行排队，然后采用多线程的方式创建订单，排队我们可以采用Redis的队列实现。 排队信息中需要有用户抢单的商品信息，主要包含商品ID，商品抢购时间段，用户登录名。我们可以设计个javabean，如下：

```
public class SeckillStatus implements Serializable {

    //秒杀用户名
    private String username;
    //创建时间
    private Date createTime;
    //秒杀状态  1:排队中，2:秒杀等待支付,3:支付超时，4:秒杀失败,5:支付完成
    private Integer status;
    //秒杀的商品ID
    private Long goodsId;

    //应付金额
    private Float money;

    //订单号
    private Long orderId;
    //时间段
    private String time;

    public SeckillStatus() {
    }

    public SeckillStatus(String username, Date createTime, Integer status, Long goodsId,
String time) {
        this.username = username;
        this.createTime = createTime;
```

```
        this.status = status;
        this.goodsId = goodsId;
        this.time = time;
    }

    //get、set...略
}
```

1.3.2.2 排队实现

我们可以将秒杀抢单信息存入到Redis中,这里采用List方式存储,List本身是一个队列，用户点击抢购的时候，就将用户抢购信息存入到Redis中，代码如下：

```
@Service
public class SeckillOrderServiceImpl implements SeckillOrderService {

    @Autowired
    private MultiThreadingCreateOrder multiThreadingCreateOrder;

    @Autowired
    private RedisTemplate redisTemplate;

    /****
     * 添加订单
     * @param id
     * @param time
     * @param username
     */
    @Override
    public Boolean add(Long id, String time, String username){
        //排队信息封装
        SeckillStatus seckillStatus = new SeckillStatus(username, new Date(),1, id,time);

        //将秒杀抢单信息存入到Redis中,这里采用List方式存储,List本身是一个队列
        redisTemplate.boundListOps("SeckillOrderQueue").leftPush(seckillStatus);

        //多线程操作
        multiThreadingCreateOrder.createOrder();
        return true;
    }
}
```

多线程每次从队列中获取数据，分别获取用户名和订单商品编号以及商品秒杀时间段，进行下单操作，代码如下：

```
/***
 * 多线程下单操作
 */
@Async
public void createOrder(){
    //从队列中获取排队信息
    SeckillStatus seckillStatus = (SeckillStatus) redisTemplate.boundListOps( key: "SeckillOrderQueue").rightPop();
    try {
        if(seckillStatus!=null){
            //时间区间
            String time = seckillStatus.getTime();
            //用户登录名
            String username=seckillStatus.getUsername();
            //用户抢购商品
            Long id = seckillStatus.getGoodsId();

            //...略
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

上图代码如下：

```
/***
 * 多线程下单操作
 */
@Async
public void createOrder(){
    //从队列中获取排队信息
    SeckillStatus seckillStatus = (SeckillStatus)
redisTemplate.boundListOps("SeckillOrderQueue").rightPop();
    try {
        if(seckillStatus!=null){
            //时间区间
            String time = seckillStatus.getTime();
            //用户登录名
            String username=seckillStatus.getUsername();
            //用户抢购商品
            Long id = seckillStatus.getGoodsId();

            //...略
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

1.3.3 下单状态查询

按照上面的流程，虽然可以实现用户下单异步操作，但是并不能确定下单是否成功，所以我们需要做一个页面判断，每过1秒钟查询一次下单状态,多线程下单的时候，需要修改抢单状态，支付的时候，清理抢单状态。

1.3.3.1 下单更新抢单状态

用户每次点击抢购的时候，如果排队成功，则将用户抢购状态存储到Redis中，多线程抢单的时候，如果抢单成功，则更新抢单状态。

修改com.qingcheng.service.impl.SeckillOrderServiceImpl的add方法，记录状态，代码如下：

```
@Override
public Boolean add(Long id, String time, String username){
    //排队信息封装
    SeckillStatus seckillStatus = new SeckillStatus(username, new Date(), status: 1, id,time);

    //将秒杀抢单信息存入到Redis中,这里采用List方式存储,List本身是一个队列
    redisTemplate.boundListOps( key: "SeckillOrderQueue").leftPush(seckillStatus);

    //将抢单状态存入到Redis中
    redisTemplate.boundHashOps( key: "UserQueueStatus").put(username,seckillStatus);

    //多线程操作
    multiThreadingCreateOrder.createOrder();
    return true;
}
```

上图代码如下：

```
//将抢单状态存入到Redis中
redisTemplate.boundHashOps("UserQueueStatus").put(username,seckillStatus);
```

多线程抢单更新状态，修改com.qingcheng.task.MultiThreadingCreateOrder的createOrder方法，代码如下：

```
@Async
public void createOrder(){
    //从队列中获取排队信息
    SeckillStatus seckillStatus = (SeckillStatus) redisTemplate.boundListOps( key: "SeckillOrderQueue").rightPop();
    try {
        if(seckillStatus!=null){
            //...略

            //抢单成功，更新抢单状态,排队->等待支付
            seckillStatus.setStatus(2);
            seckillStatus.setOrderId(seckillOrder.getId());
            seckillStatus.setMoney(seckillOrder.getMoney().floatValue());
            redisTemplate.boundHashOps( key: "UserQueueStatus").put(username,seckillStatus);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

更新抢单排队信息

上图代码如下：

```
//抢单成功，更新抢单状态,排队->等待支付
seckillStatus.setStatus(2);
seckillStatus.setOrderId(seckillOrder.getId());
seckillStatus.setMoney(seckillOrder.getMoney().floatValue());
```

1.3.3.2 后台查询抢单状态

后台提供抢单状态查询方法，修改com.qingcheng.service.seckill.SeckillOrderService，添加如下查询方法：

```
/***
 * 抢单状态查询
 * @param username
 */
SeckillStatus queryStatus(String username);
```

修改com.qingcheng.service.impl.SeckillOrderServiceImpl,添加如下实现方法：

```
/***
 * 抢单状态查询
 * @param username
 * @return
 */
@Override
public SeckillStatus queryStatus(String username) {
    return (SeckillStatus) redisTemplate.boundHashOps("UserQueueStatus").get(username);
}
```

修改com.qingcheng.controller.SeckillOrderController,添加如下查询方法：

```
@RequestMapping(value = "/query")
public Result queryStatus(){
    //获取用户名
    String username = SecurityContextHolder.getContext().getAuthentication().getName();

    //如果用户账号为anonymousUser，则表明用户未登录
    if(username.equalsIgnoreCase( anotherString: "anonymousUser")){
        //这里403错误代码表示用户没登录
        return new Result( code: 403, message: "请先登录！");
    }
    //根据用户名查询用户抢购状态
    SeckillStatus seckillStatus = seckillOrderService.queryStatus(username);

    if(seckillStatus!=null){
        return new Result(seckillStatus.getStatus(), message: "抢购状态");
    }
    return new Result( code: 404, message: "没有抢购信息");
}
```

上图代码如下：

```
@RequestMapping(value = "/query")
public Result queryStatus(){
    //获取用户名
    String username = SecurityContextHolder.getContext().getAuthentication().getName();

    //如果用户账号为anonymousUser，则表明用户未登录
    if(username.equalsIgnoreCase("anonymousUser")){
        //这里403错误代码表示用户没登录
        return new Result(403,"请先登录！");
    }
    //根据用户名查询用户抢购状态
    SeckillStatus seckillStatus = seckillOrderService.queryStatus(username);

    if(seckillStatus!=null){
        return new Result(seckillStatus.getStatus(),"抢购状态");
    }
    return new Result(404,"没有抢购信息");
}
```

1.3.3.3 前端循环查询

在js中添加一个循环查询方法，每秒钟查询一次，累计查询120秒，代码如下：

```
//每2秒钟查询一次抢单状态
queryStatus:function () {
    //查询120秒
    let count=120;
    //每1秒钟执行1次
    let queryclock = window.setInterval(() => {
        //时间递减
        count--;
        //执行查询
        axios.get('/seckill/order/query.do').then(function (resp) {
            //403,未登录
            if(resp.data.code==403){
                location.href='/redirect/back.do';
            }else if(resp.data.code==1){
                //1，排队中
                app.message="正在排队.."+count;
            }else{
                if(resp.data.code==404){
                    //404,未找到秒杀订单,提示错误信息
                    app.message="抢单失败,请稍后再试！";
                }else if(resp.data.code==2){
                    //2，等待支付
                    app.message="抢单成功，即将进入支付!";
                }else if(resp.data.code==4){
                    //4,已售罄
                    app.message="已售罄!";
                }else{
                    app.message="抢购失败，服务器繁忙！";
```

```
                }
                //结束定时
                window.clearInterval(queryclock);
            }
        })
    },1000)
}
```

抢单后，立即执行上面每秒查询1次的方法，代码如下：

```
//添加订单
add:function () {
    //提示信息
    app.message='正在下单';
    //获取地址栏time参数
    let time = getQueryString("time");
    //获取地址栏id参数
    let id = getQueryString("id");

    //下单操作
    axios.get('/seckill/order/add.do?time='+time+'&id='+id).then(function (response) {
        if(response.data.code==0){
            //正在排队
            app.queryStatus();
        }else if(response.data.code==403){
            //用户未登录,跳转到登录页
            app.message=response.data.message;

            //未登录,跳转到/back.do,携带referer，此时会被拒绝访问，跳转到CAS登录
            // CAS登录成功后，跳转到/back.do，此时跟重新跳转到referer对应的地址，也就是当前页地址
            location.href='/redirect/back.do';
        }else{
            //用户未登录
            app.message=response.data.message;
        }
    })
},
```

上图代码如下：

```
//正在排队
app.queryStatus();
```

测试效果如下：

第2章 防止秒杀重复排队

用户每次抢单的时候，一旦排队，我们设置一个自增值，让该值的初始值为1，每次进入抢单的时候，对它进行递增，如果值>1，则表明已经排队,不允许重复排队,如果重复排队，则对外抛出异常，并抛出异常信息100表示已经正在排队。

2.1 后台排队记录

修改com.qingcheng.service.impl.SeckillOrderServiceImpl的add方法，新增递增值判断是否排队中，代码如下：

```
@Override
public Boolean add(Long id, String time, String username){
    //递增，判断是否排队
    Long userQueueCount = redisTemplate.boundHashOps( key: "UserQueueCount").increment(username, l: 1);
    if(userQueueCount>1){
        //100：表示有重复抢单
        throw new RuntimeException("100");
    }
    //排队信息封装
    SeckillStatus seckillStatus = new SeckillStatus(username, new Date(), status: 1, id,time);
    //将秒杀抢单信息存入到Redis中,这里采用List方式存储,List本身是一个队列
    redisTemplate.boundListOps( key: "SeckillOrderQueue").leftPush(seckillStatus);
    //将抢单状态存入到Redis中
    redisTemplate.boundHashOps( key: "UserQueueStatus").put(username,seckillStatus);
    //多线程操作
    multiThreadingCreateOrder.createOrder();
    return true;
}
```

上图代码如下：

```
//递增，判断是否排队
Long userQueueCount = redisTemplate.boundHashOps("UserQueueCount").increment(username, 1);
if(userQueueCount>1){
    //100：表示有重复抢单
    throw new RuntimeException("100");
}
```

2.2 页面识别重复排队

修改页面下单js方法add，添加重复排队识别代码，代码如下：

```
//添加订单
add:function () {
    //提示信息
    app.message='正在下单';
    //获取地址栏time参数
    let time = getQueryString("time");
    //获取地址栏id参数
    let id = getQueryString("id");

    //下单操作
    axios.get('/seckill/order/add.do?time='+time+'&id='+id).then(function (response) {
        if(response.data.code==0 || (response.data.code==2 && response.data.message=="100")){
            //排队成功
            app.queryStatus();
        }else if(response.data.code==403){
            //用户未登录,跳转到登录页
            app.message=response.data.message;

            //未登录,跳转到/back.do,携带referer，此时会被拒绝访问，跳转到CAS登录
            // CAS登录成功后，跳转到/back.do，此时跟重新跳转到referer对应的地址，也就是当前页地址
            location.href='/redirect/back.do';
        }else{
            //用户未登录
            app.message=response.data.message;
        }
    })
},
```

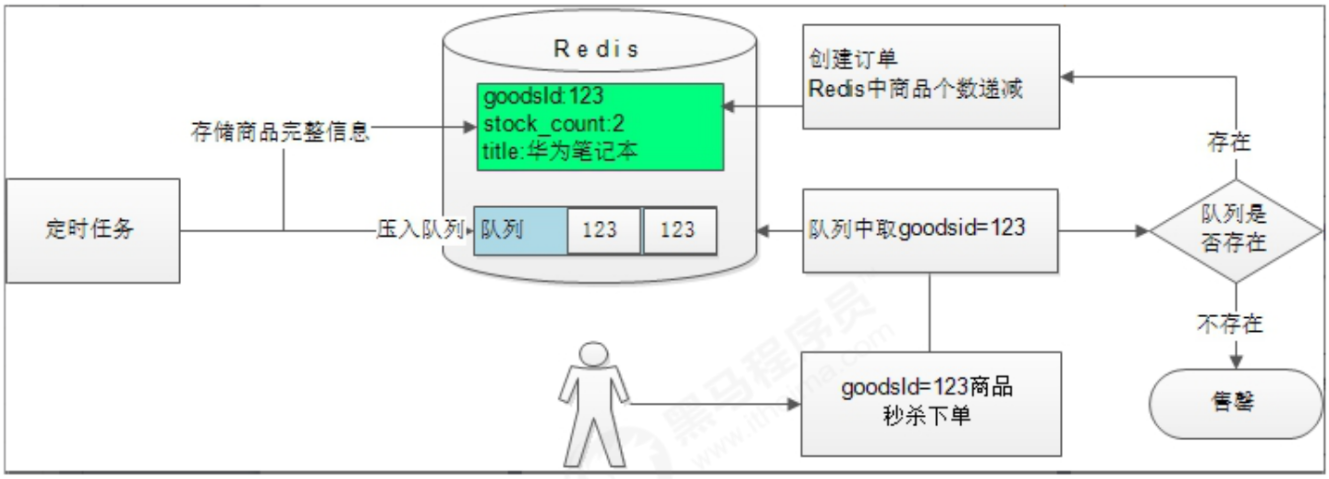
上图代码如下：

```
response.data.code==0 || (response.data.code==2 && response.data.message=="100")
```

第3章 并发超卖问题解决

超卖问题，这里是指多人抢购同一商品的时候，多人同时判断是否有库存，如果只剩一个，则都会判断有库存，此时会导致超卖现象产生，也就是一个商品下了多个订单的现象。

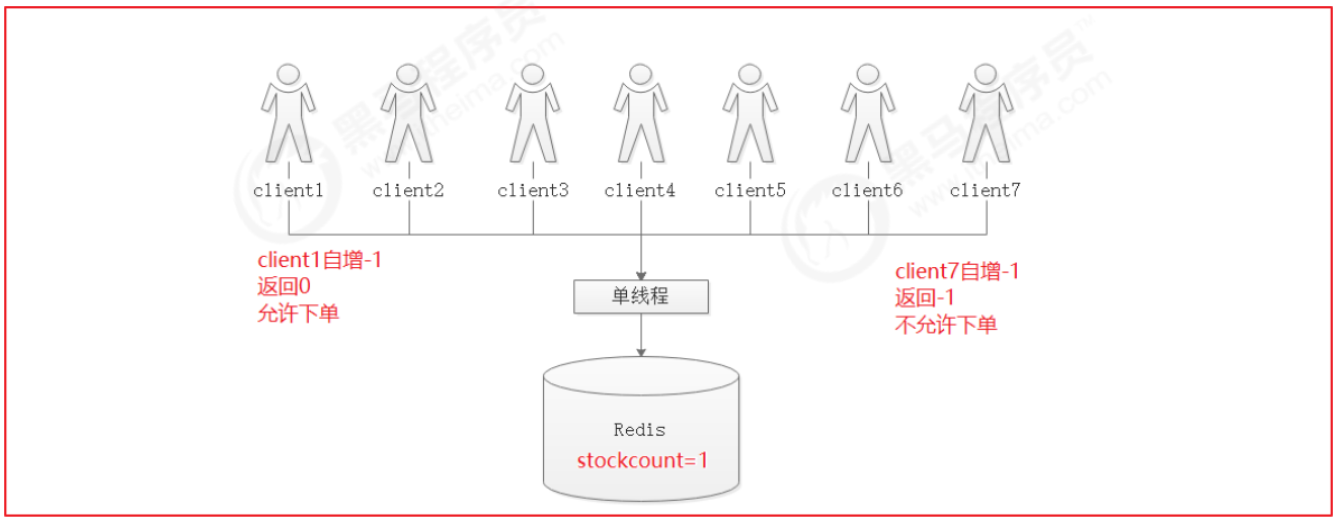
3.1 思路分析



解决超卖问题，可以利用Redis队列实现，给每件商品创建一个独立的商品个数队列，例如：A商品有2个，A商品的ID为1001，则可以创建一个队列,key=SeckillGoodsCountList_1001,往该队列中塞2次该商品ID。

每次给用户下单的时候，先从队列中取数据，如果能取到数据，则表明有库存，如果取不到，则表明没有库存，这样就可以防止超卖问题产生了。

在我们队Redis进行操作的时候，很多时候，都是先将数据查询出来，在内存中修改，然后存入到Redis，在并发场景，会出现数据错乱问题，为了控制数量准确，我们单独将商品数量整一个自增键，自增键是线程安全的，所以不担心并发场景的问题。



3.2 商品个数队列创建

每次将商品压入Redis缓存的时候，另外多创建一个商品的队列。

修改com.qingcheng.timer.SeckillGoodsPushTask,添加一个pushIds方法，用于将指定商品ID放入到指定的数字中，代码如下：

```
/***
 * 将商品ID存入到数组中
 * @param len:长度
 * @param id :值
 * @return
 */
public Long[] pushIds(int len,Long id){
    Long[] ids = new Long[len];
    for (int i = 0; i <ids.length ; i++) {
        ids[i]=id;
    }
    return ids;
}
```

修改SeckillGoodsPushTask的loadGoodsPushRedis方法，添加队列操作，代码如下：

```
@Scheduled(cron = "0/30 * * * * ?")
public void loadGoodsPushRedis(){
    //获取时间段集合
    List<Date> dateMenus = DateUtil.getDateMenus();
    //循环时间段
    for (Date startTime : dateMenus) {
        //...略
        //将秒杀商品数据存入到Redis缓存
        for (SeckillGoods seckillGood : seckillGoods) {
            redisTemplate.boundHashOps( key: "SeckillGoods_"+extName).put(seckillGood.getId(),seckillGood);
            //商品数据队列存储,防止高并发超卖
            Long[] ids = pushIds(seckillGood.getStockCount(), seckillGood.getId());
            redisTemplate.boundListOps( key: "SeckillGoodsCountList_"+seckillGood.getId()).leftPushAll(ids);
            //自增计数器
            redisTemplate.boundHashOps( key: "SeckillGoodsCount").increment(seckillGood.getId(),seckillGood.getStockCount());
        }
    }
}
```

上图代码如下：

```
//商品数据队列存储,防止高并发超卖
Long[] ids = pushIds(seckillGood.getStockCount(), seckillGood.getId());
redisTemplate.boundListOps("SeckillGoodsCountList_"+seckillGood.getId()).leftPushAll(ids);
//自增计数器
redisTemplate.boundHashOps("SeckillGoodsCount").increment(seckillGood.getId(),seckillGood.ge
tStockCount());
```

3.3 超卖控制

修改多线程下单方法，分别修改数量控制，以及售罄后用户抢单排队信息的清理，修改代码如下图：

```java
@Component
public class MultiThreadingCreateOrder {

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private SeckillGoodsMapper seckillGoodsMapper;

    @Autowired
    private IdWorker idWorker;

    /***
     * 多线程下单操作
     */
    @Async
    public void createOrder(){
        //从队列中获取排队信息
        SeckillStatus seckillStatus = (SeckillStatus) redisTemplate.boundListOps( key: "SeckillOrderQueue").rightPop();
        try {
            if(seckillStatus!=null){
                //从队列中获取一个商品
                Object sgood = redisTemplate.boundListOps( key: "SeckillGoodsCountList_" + seckillStatus.getGoodsId()).rightPop();
                if(sgood==null){
                    //清理当前用户的排队信息
                    clearQueue(seckillStatus);
                    return;
                }

                //时间区间
                String time = seckillStatus.getTime();
                //用户登录名
                String username=seckillStatus.getUsername();
                //用户抢购商品
                Long id = seckillStatus.getGoodsId();

                //获取商品数据
                SeckillGoods goods = (SeckillGoods) redisTemplate.boundHashOps( key: "SeckillGoods_" + time).get(id);

                //如果没有库存，则直接抛出异常
                if(goods==null || goods.getStockCount()<=0){
                    throw new RuntimeException("已售罄!");
                }
                //如果有库存，则创建秒杀商品订单
                SeckillOrder seckillOrder = new SeckillOrder();
                seckillOrder.setId(idWorker.nextId());
                seckillOrder.setSeckillId(id);
                seckillOrder.setMoney(goods.getCostPrice());
                seckillOrder.setUserId(username);
                seckillOrder.setSellerId(goods.getSellerId());
                seckillOrder.setCreateTime(new Date());
                seckillOrder.setStatus("0");

                //将秒杀订单存入到Redis中
                redisTemplate.boundHashOps( key: "SeckillOrder").put(username,seckillOrder);

                                使用Redis递减的方式，不会出现并发数据错乱问题
                //商品库存-1
                Long surplusCount = redisTemplate.boundHashOps( key: "SeckillGoodsCount").increment(id,  l: -1);//商品数量递减
                goods.setStockCount(surplusCount.intValue());     //根据计数器统计

                //判断当前商品是否还有库存
                if(surplusCount<=0){
                    //并且将商品数据同步到MySQL中
                    seckillGoodsMapper.updateByPrimaryKeySelective(goods);
                    //如果没有库存,则清空Redis缓存中该商品
                    redisTemplate.boundHashOps( key: "SeckillGoods_" + time).delete(id);
```

上图代码如下：

```
/***
 * 多线程下单操作
 */
@Async
public void createOrder(){
    //从队列中获取排队信息
    SeckillStatus seckillStatus = (SeckillStatus)
redisTemplate.boundListOps("SeckillOrderQueue").rightPop();
    try {
        if(seckillStatus!=null){
            //从队列中获取一个商品
            Object sgood = redisTemplate.boundListOps("SeckillGoodsCountList_" +
seckillStatus.getGoodsId()).rightPop();
            if(sgood==null){
                //清理当前用户的排队信息
                clearQueue(seckillStatus);
                return;
            }

            //...略

            //商品库存-1
            Long surplusCount =
redisTemplate.boundHashOps("SeckillGoodsCount").increment(id, -1);//商品数量递减
            goods.setStockCount(surplusCount.intValue());     //根据计数器统计

            //判断当前商品是否还有库存
```

```
            if(surplusCount<=0){
                //并且将商品数据同步到MySQL中
                seckillGoodsMapper.updateByPrimaryKeySelective(goods);
                //如果没有库存,则清空Redis缓存中该商品
                redisTemplate.boundHashOps("SeckillGoods_" + time).delete(id);
            }else{
                //如果有库存，则直数据重置到Reids中
                redisTemplate.boundHashOps("SeckillGoods_" + time).put(id,goods);
            }

            //...略
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/***
 * 清理用户排队信息
 * @param seckillStatus
 */
public void clearQueue(SeckillStatus seckillStatus){
    //清理排队标示
    redisTemplate.boundHashOps("UserQueueCount").delete(seckillStatus.getUsername());

    //清理抢单标示
    redisTemplate.boundHashOps("UserQueueStatus").delete(seckillStatus.getUsername());
}
```

用户抢单的时候，也做一个剩余库存数量判断，修改com.qingcheng.service.impl.SeckillOrderServiceImpl的add方法，代码如下：

```java
@Override
public Boolean add(Long id, String time, String username){
    //判断是否有库存
    Long size = redisTemplate.boundListOps( key: "SeckillGoodsCountList_" + id).size();
    if(size==null || size<=0){
        //101:没有库存
        throw new RuntimeException("101");
    }
    //递增，判断是否排队
    Long userQueueCount = redisTemplate.boundHashOps( key: "UserQueueCount").increment(username, l: 1);
    if(userQueueCount>1){
        //100：表示有重复抢单
        throw new RuntimeException("100");
    }
    //排队信息封装
    SeckillStatus seckillStatus = new SeckillStatus(username, new Date(), status: 1, id,time);
    //将秒杀抢单信息存入到Redis中,这里采用List方式存储,List本身是一个队列
    redisTemplate.boundListOps( key: "SeckillOrderQueue").leftPush(seckillStatus);
    //将抢单状态存入到Redis中
    redisTemplate.boundHashOps( key: "UserQueueStatus").put(username,seckillStatus);
    //多线程操作
    multiThreadingCreateOrder.createOrder();
    return true;
}
```

判断队列中是否还有数据

上图代码如下：

```java
//判断是否有库存
Long size = redisTemplate.boundListOps("SeckillGoodsCountList_" + id).size();
if(size==null || size<=0){
    //101:没有库存
    throw new RuntimeException("101");
}
```

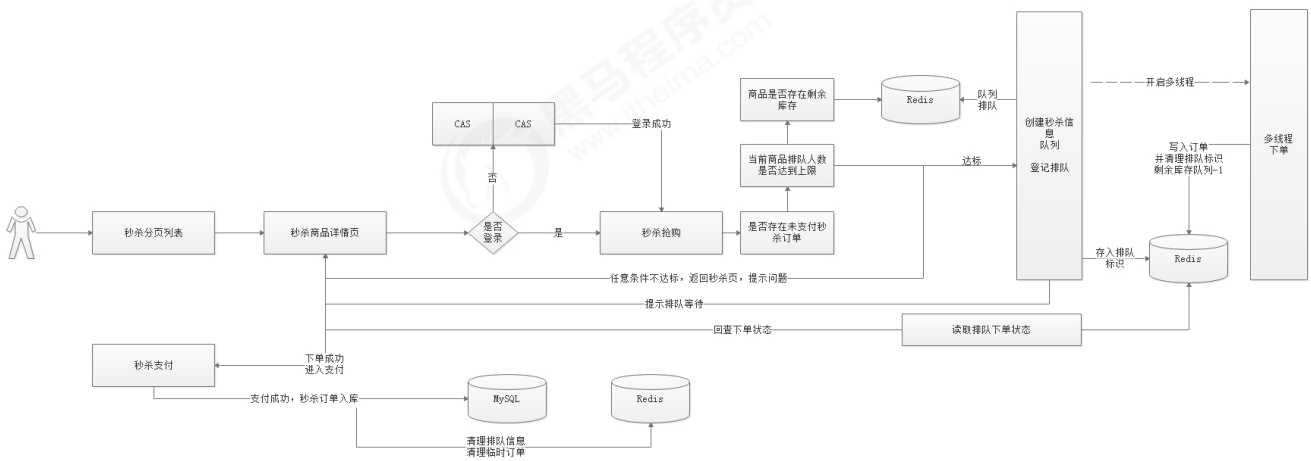
页面加上对应判断

```
//添加订单
add:function () {
    //提示信息
    app.message='正在下单';
    //获取地址栏time参数
    let time = getQueryString("time");
    //获取地址栏id参数
    let id = getQueryString("id");

    //下单操作
    axios.get('/seckill/order/add.do?time='+time+'&id='+id).then(function (response) {
        if(response.data.code==0 || (response.data.code==2 && response.data.message=="100")){
            //排队成功
            app.queryStatus();
        }else if(response.data.code==403){
            //用户未登录,跳转到登录页
            app.message=response.data.message;

            //未登录,跳转到/back.do,携带referer，此时会被拒绝访问，跳转到CAS登录
            // CAS登录成功后，跳转到/back.do，此时跟重新跳转到referer对应的地址，也就是当前页地址
            location.href='/redirect/back.do';
        }else if(response.data.code==2 && response.data.message=="101"){
            app.message="已售罄！";
        }else{
            //用户未登录
            app.message=response.data.message;
        }
    })
},
```

上图代码如下：

```
else if(response.data.code==2 && response.data.message=="101"){
    app.message="已售罄！";
}
```

第4章 订单支付



完成秒杀下订单后，进入支付页面，此时前端会每3秒中向后台发送一次请求用于判断当前用户订单是否完成支付，如果完成了支付，则需要清理掉排队信息，并且需要修改订单状态信息。

4.1 修改订单状态

创建一个SeckillOrderMapper接口，作为Dao层，代码如下：

```
public interface SeckillOrderMapper extends Mapper<SeckillOrder> {
}
```

修改com.qingcheng.service.seckill.SeckillOrderService接口，添加如下方法：

```
/***
 * 更新订单状态
 * @param out_trade_no
 * @param transaction_id
 * @param username
 */
void updatePayStatus(String out_trade_no, String transaction_id,String username);
```

修改com.qingcheng.service.impl.SeckillOrderServiceImpl添加updatePayStatus方法，代码如下：

```
/***
 * 更新订单状态
 * @param out_trade_no
 * @param transaction_id
 * @param username
 */
@Override
public void updatePayStatus(String out_trade_no, String transaction_id,String username) {
    //订单数据从Redis数据库查询出来
    SeckillOrder seckillOrder = (SeckillOrder)
redisTemplate.boundHashOps("SeckillOrder").get(username);
    //修改状态
    seckillOrder.setStatus("1");

    //支付时间
    seckillOrder.setPayTime(new Date());
    //同步到MySQL中
    seckillOrderMapper.insertSelective(seckillOrder);

    //清空Redis缓存
    redisTemplate.boundHashOps("SeckillOrder").delete(username);

    //清空用户排队数据
    redisTemplate.boundHashOps("UserQueueCount").delete(username);
```

```
    //删除抢购状态信息
    redisTemplate.boundHashOps("UserQueueStatus").delete(username);
}
```

4.2 创建支付二维码

下单成功后，会跳转到支付选择页面，在支付选择页面要显示订单编号和订单金额，所以我们需要在下单的时候，将订单金额以及订单编号信息存储到用户查询对象中。

选择微信支付后，会跳转到微信支付页面，微信支付页面会根据用户名查看用户秒杀订单，并根据用户秒杀订单的ID创建预支付信息并获取二维码信息，展示给用户看,此时页面每3秒查询一次支付状态，如果支付成功，需要修改订单状态信息。

4.2.2 回显订单号、金额

下单后，进入支付选择页面，需要显示订单号和订单金额，所以需要在用户下单后将该数据传入到pay.html页面，所以查询订单状态的时候，需要将订单号和金额封装到查询的信息中，修改查询订单装的方法加入他们即可。

修改com.qingcheng.controller.SeckillOrderController的queryStatus方法，代码如下：

```
@RequestMapping(value = "/query")
public Result queryStatus(){
    //获取用户名
    String username = SecurityContextHolder.getContext().getAuthentication().getName();
    //如果用户账号为anonymousUser，则表明用户未登录
    if(username.equalsIgnoreCase( anotherString: "anonymousUser")){
        //这里403错误代码表示用户没登录
        return new Result( code: 403, message: "请先登录！");
    }
    //根据用户名查询用户抢购状态
    SeckillStatus seckillStatus = seckillOrderService.queryStatus(username);

    if(seckillStatus!=null){
        //获取订单号
        Result result = new Result(seckillStatus.getStatus(), message: "抢单状态！");
        result.setOther(seckillStatus);
        return result;
    }
    return new Result( code: 404, message: "没有抢购信息");
}
```

抢单成功后，封装订单信息

上图代码如下：

```
if(seckillStatus!=null){
    //获取订单号
    Result result = new Result(seckillStatus.getStatus(),"抢单状态! ");
    result.setOther(seckillStatus);
    return result;
}
```

4.2.3 用户订单查询

编写一个方法用于根据用户名查询用户订单信息。

修改com.qingcheng.service.seckill.SeckillOrderService接口，添加如下方法

```
/***
 * 根据用户名查询用户秒杀订单信息
 * @param username
 * @return
 */
SeckillOrder findById(String username);
```

修改com.qingcheng.service.impl.SeckillOrderServiceImpl,添加根据用户名查询订单信息的方法，代码如下：

```
/***
 * 根据用户名查询用户秒杀订单信息
 * @param username
 * @return
 */
@Override
public SeckillOrder findById(String username) {
    return (SeckillOrder) redisTemplate.boundHashOps("SeckillOrder").get(username);
}
```

4.2.4 创建二维码

在qingcheng_web_seckill工程中添加com.qingcheng.controller.PayController,并创建获取二维码信息的方法，代码如下：

```
@RestController
@RequestMapping("/pay")
public class PayController {

    @Reference
    private WeixinPayService weixinPayService;

    @Reference
    private SeckillOrderService seckillOrderService;

    /**
     * 生成二维码
     * @return
     */
    @GetMapping("/createNative")
    public Map createNative(){
        String username = SecurityContextHolder.getContext().getAuthentication().getName();
        SeckillOrder seckillOrder = seckillOrderService.findById(username);
        if(seckillOrder!=null){
            //校验该订单是否时当前用户的订单
            if(username.equals(seckillOrder.getUserId())){
```

```
                int money = (int) ((seckillOrder.getMoney().doubleValue())*100);
                return weixinPayService.createNative(
                            seckillOrder.getId().toString(),
                            money,
                        "http://qingcheng.cross.echosite.cn/pay/notify.do");
            }else{
                return null;
            }
        }else{
            return null;
        }
    }
}
```

4.2.5 创建二维码页面对接

将支付工程中的pay.html，weixinpay.html，paysuccess.html，payfail.html拷贝到qingcheng_web_seckill工程根目录。

修改seckill-item.html页面的queryStatus方法，一旦支付成功，跳转到支付选择页面，代码如下：

```
//每秒钟查询一次抢单状态
queryStatus:function () {
    //查询120秒
    let count=120;
    //每1秒钟执行1次
    let queryclock = window.setInterval(() => {
        //时间递减
        count--;
        //执行查询
        axios.get('/seckill/order/query.do').then(function (resp) {
            //403,未登录
            if(resp.data.code==403){
                location.href='/redirect/back.do';
            }else if(resp.data.code==1){
                //1，排队中
                app.message="正在排队.."+count;
            }else{
                if(resp.data.code==404){
                    //404,未找到秒杀订单,提示错误信息
                    app.message="抢单失败,请稍后再试！";
                }else if(resp.data.code==2){
                    //2，等待支付
                    app.message="抢单成功，即将进入支付!";
                    location.href='/pay.html?orderId='+resp.data.other.orderId+'&money='+resp.data.other.money;
                }else if(resp.data.code==4){
                    //4,已售罄
                    app.message="已售罄!";
                }else{
                    app.message="抢购失败，服务器繁忙！";
                }
                //结束定时
                window.clearInterval(queryclock);
            }
        })
    },1000)
},
```

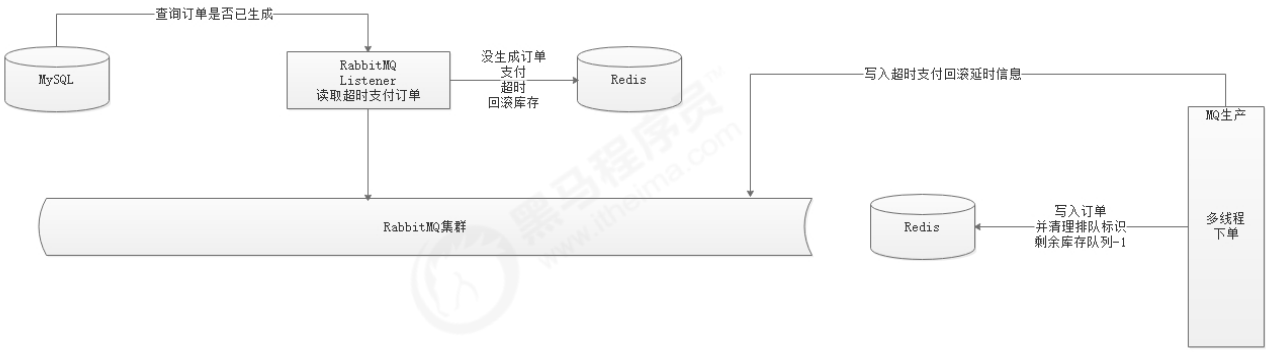
4.3 支付状态查询

用户支付后，从前端循环查询状态，如果支付成功了，则修改订单状态，并清理用户排队信息。

在PayController.java中，添加后台查询状态的方法，代码如下：

```
    /***
     * 查询订单状态
     * @param orderId
     * @return
     */
    @GetMapping("/queryPayStatus")
    public Map<String, String> queryPayStatus(String orderId){
        Map<String,String> resultMap = weixinPayService.queryPayStatus(orderId);
        if(resultMap.get("return_code").equalsIgnoreCase("success") &&
           resultMap.get("result_code").equalsIgnoreCase("success")){
            //获取支付状态
            String result = resultMap.get("trade_state");
            if(result.equalsIgnoreCase("success")){
                String username =
SecurityContextHolder.getContext().getAuthentication().getName();
                //支付成功,修改订单状态

seckillOrderService.updatePayStatus(resultMap.get("out_trade_no"),resultMap.get("transaction
_id"),username);
            }
        }
        return resultMap;
    }
```

第5章 超时订单库存回滚



用户每次下单后，不一定会立即支付，甚至有可能不支付，那么此时我们需要删除用户下的订单，并回滚库存。这里我们可以采用MQ的延时消息实现，每次用户下单的时候，如果订单创建成功，则立即发送一个延时消息到MQ中，等待消息被消费的时候，先检查对应订单是否下单支付成功，如果支付成功，会在MySQL中生成一个订单，如果MySQL中没有支付，则Redis中还有该订单信息的存在，需要删除该订单信息以及用户排队信息，并恢复库存。

5.1 RabbitMQ延时队列讲解

RabbitMQ并没有直接实现延时队列，但是可以利用RabbitMQ两个属性实现延时队列特性：

1、x-message-ttl：消息过期时间（Time To Live，TTL），超过过期时间之后即变为死信（Dead-letter），不会再被消费者消费。

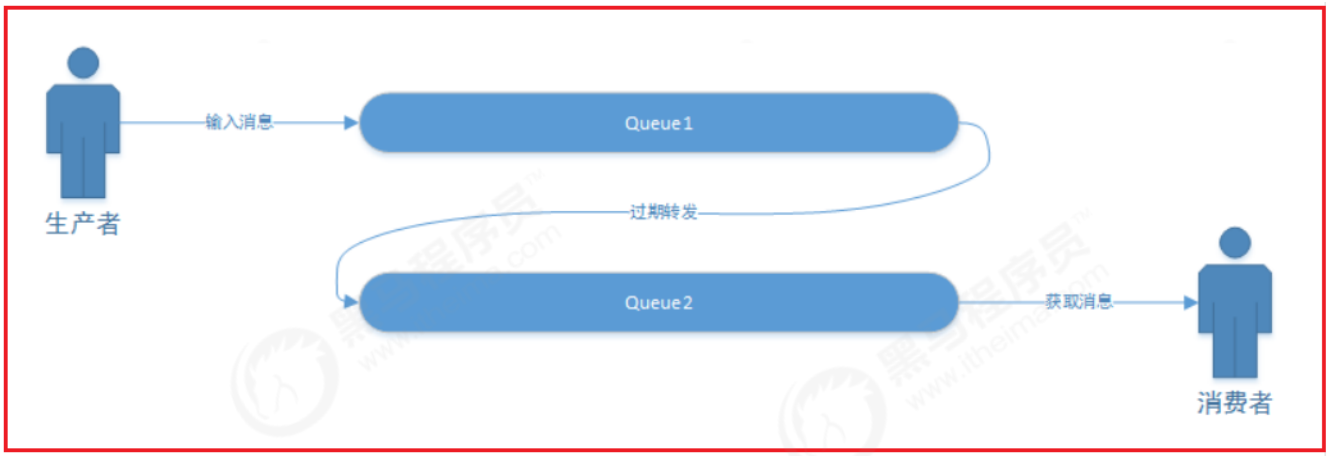
设置TTL有两种方式：

（1）创建队列时指定x-message-ttl，此时整个队列具有统一过期时间；

（2）发送消息为每个消息设置expiration，此时消息之间过期时间不同。

注意：如果两者都设置，过期时间取两者最小。

2、x-dead-letter-exchange：过期消息路由转发，当消息达到过期时间由该exchange按照配置的x-dead-letter-routing-key转发到指定队列，最后被消费者消费。



5.2 关闭微信支付订单

取消订单库存回滚的时候，需要注意这么个场景，用户有可能正在扫码支付，所以我们需要先关闭微信支付，然后再取消本地订单回滚库存。

5.2.1 关闭微信订单API

微信支付API参考地址：https://pay.weixin.qq.com/wiki/doc/api/native.php?chapter=9_3

接口链接：https://api.mch.weixin.qq.com/pay/closeorder

请求参数：

字段名	变量名	必填	类型	示例值
公众账号ID	appid	是	String(32)	wx8888888888888888
商户号	mch_id	是	String(32)	1900000109
商户订单号	out_trade_no	是	String(32)	1217752501201407033233368018
随机字符串	nonce_str	是	String(32)	5K8264ILTKCH16CQ2502SI8ZNMTM67VS
签名	sign	是	String(32)	C380BEC2BFD727A4B6845133519F3AD6
签名类型	sign_type	否	String(32)	HMAC-SHA256

返回结果：

字段名	变量名	必填	类型	示例值	描述
返回状态码	return_code	是	String(16)	SUCCESS	SUCCESS/FAIL此字段是通信标识，非交易标识，交易是否成功需要查看trade_state来判断
业务结果	result_code	是	String(16)	SUCCESS	SUCCESS/FAIL

这里我们只关心成功结果，失败结果人工处理。

5.2.2 关闭微信支付实现

修改com.qingcheng.service.pay.WeixinPayService,添加关闭支付方法，代码如下：

```
/***
 * 关闭支付订单
 * @param orderId
 * @return
 */
Map<String, String> closePay(Long orderId) throws Exception;
```

修改com.qingcheng.service.impl.WeixinPayServiceImpl,添加关闭支付实现，代码如下：

```
/***
 * 关闭微信支付
 * @param orderId
 * @return
 * @throws Exception
 */
@Override
public Map<String, String> closePay(Long orderId) throws Exception{
```

```
    //参数设置
    Map<String,String> paramMap = new HashMap<String,String>();
    paramMap.put("appid",appid); //应用ID
    paramMap.put("mch_id",partner);    //商户编号
    paramMap.put("nonce_str",WXPayUtil.generateNonceStr());//随机字符
    paramMap.put("out_trade_no",String.valueOf(orderId));   //商家的唯一编号

    //将Map数据转成XML字符
    String xmlParam = WXPayUtil.generateSignedXml(paramMap,partnerkey);

    //确定url
    String url = "https://api.mch.weixin.qq.com/pay/closeorder";

    //发送请求
    HttpClient httpClient = new HttpClient(url);
    //https
    httpClient.setHttps(true);
    //提交参数
    httpClient.setXmlParam(xmlParam);

    //提交
    httpClient.post();

    //获取返回数据
    String content = httpClient.getContent();

    //将返回数据解析成Map
    return  WXPayUtil.xmlToMap(content);
}
```

5.1 下单延时消息发送

5.1.1 集成RabbitMQ

修改qingcheng_service_seckill工程，在该工程中添加applicationContext-rabbitmq-producer.xml,用于配置消息发送对象，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/rabbit
       http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">

    <!--配置connection-factory，指定连接rabbit server参数 -->
    <rabbit:connection-factory id="connectionFactory" username="guest" password="guest"
host="127.0.0.1" port="5672" publisher-confirms="true"/>
```

```
    <rabbit:admin connection-factory="connectionFactory"></rabbit:admin>

    <!-- 延时队列 -->
    <rabbit:direct-exchange id="exchange.delay.order.begin"
name="exchange.delay.order.begin"  durable="false" auto-delete="false" >
        <rabbit:bindings>
            <rabbit:binding queue="queue.delay.order.begin" key="delay" />
        </rabbit:bindings>
    </rabbit:direct-exchange>

    <rabbit:queue name="queue.delay.order.begin" durable="false">
        <rabbit:queue-arguments>
            <!--  队列过期时间 -->
            <entry key="x-message-ttl" value="10000" value-type="java.lang.Long" />
            <entry key="x-dead-letter-exchange" value="exchange.delay.order.done" />
            <entry key="x-dead-letter-routing-key" value="delay" />
        </rabbit:queue-arguments>
    </rabbit:queue>

    <rabbit:direct-exchange id="exchange.delay.order.done" name="exchange.delay.order.done"
durable="false" auto-delete="false" >
        <rabbit:bindings>
            <rabbit:binding queue="queue.delay.order.done" key="delay" />
            <!--  binding key 相同为 【delay】exchange转发消息到多个队列 -->
            <!--<rabbit:binding queue="queue.delay.order.done.two" key="delay" />-->
        </rabbit:bindings>
    </rabbit:direct-exchange>

    <rabbit:queue name="queue.delay.order.done" durable="false"/>

    <rabbit:template id="rabbitTemplate" connection-factory="connectionFactory" />

    <!-- 消息接收者 -->
    <rabbit:listener-container connection-factory="connectionFactory" channel-
transacted="false" >
        <rabbit:listener queues="queue.delay.order.done" ref="orderMessageListener" />
    </rabbit:listener-container>
</beans>
```

5.1.2 下单延时消息发送

在下单的时候，实现消息发送，这里采用延时消息队列，代码如下：

```
/***
 * 延时消息发送
 * @param seckillStatus
 */
public void sendDelayMessage(SeckillStatus seckillStatus){
    rabbitTemplate.convertAndSend(
            "exchange.delay.order.begin",
            "delay",
```

```
            JSON.toJSONString(seckillStatus),
            new MessagePostProcessor() {
                @Override
                public Message postProcessMessage(Message message) throws AmqpException {
                    //消息有效期30分钟
                    //message.getMessageProperties().setExpiration(String.valueOf(1800000));
                    message.getMessageProperties().setExpiration(String.valueOf(10000));
                    return message;
                }
            });
}
```

订单创建完成后记得调用上面发送延时消息的方法，代码如下：

```
@Async
public void createOrder(){
    //从队列中获取排队信息
    SeckillStatus seckillStatus = (SeckillStatus) redisTemplate.boundListOps( key: "SeckillOrderQueue").rightPop();
    try {
        if(seckillStatus!=null){
            //...略
            //发送MQ消息
            sendDelayMessage(seckillStatus);
        }
    } catch (Exception e) {
        e.printStackTrace();
        //发生错误，记录日志，交由人工处理
    }
}
```

上图代码如下：

```
//发送MQ消息
sendDelayMessage(seckillStatus);
```

5.2.2 延时消息消费

创建一个com.qingcheng.consumer.OrderMessageListener类用于消费延时消息，并在该方法中实现数据回滚等操作，代码如下：

```
@Component
public class OrderMessageListener implements MessageListener {

    /***
     * 消息监听
     * @param message
     */
    @Override
    public void onMessage(Message message) {
        try {
            //获取消息
            String content = new String(message.getBody());
```

```
            //将消息转换成SeckillStatus
            SeckillStatus seckillStatus = JSON.parseObject(content,SeckillStatus.class);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.2.3 库存回滚

创建一个方法，实现库存回滚，并在消息消费后调用该方法，完整代码如下：

```
@Component
public class OrderMessageListener implements MessageListener {
    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private SeckillOrderMapper seckillOrderMapper;

    @Autowired
    private SeckillGoodsMapper seckillGoodsMapper;

    @Reference
    private WeixinPayService weixinPayService;

    /***
     * 订单处理以及回滚库存处理
     * @param seckillStatus
     */
    public void rollbackOrder(SeckillStatus seckillStatus) throws Exception {
        //获取Redis中订单信息
        String username = seckillStatus.getUsername();
        SeckillOrder seckillOrder = (SeckillOrder)
redisTemplate.boundHashOps("SeckillOrder").get(username);

        //如果Redis中有订单信息，说明用户未支付
        if(seckillOrder!=null){
            //关闭支付
            Map<String,String> closeResult =
weixinPayService.closePay(seckillStatus.getOrderId());
            if(closeResult.get("return_code").equalsIgnoreCase("success") &&
                    closeResult.get("result_code").equalsIgnoreCase("success") ){
                //删除订单
                redisTemplate.boundHashOps("SeckillOrder").delete(username);
                //回滚库存
                //1)从Redis中获取该商品
                SeckillGoods seckillGoods = (SeckillGoods)
redisTemplate.boundHashOps("SeckillGoods_"+seckillStatus.getTime()).get(seckillStatus.getGoo
dsId());
```

```
                //2)如果Redis中没有，则从数据库中加载
                if(seckillGoods==null){
                    seckillGoods =
seckillGoodsMapper.selectByPrimaryKey(seckillStatus.getGoodsId());
                }

                //3)数量+1  (递增数量+1，队列数量+1)
                Long surplusCount =
redisTemplate.boundHashOps("SeckillGoodsCount").increment(seckillStatus.getGoodsId(), 1);
                seckillGoods.setStockCount(surplusCount.intValue());
                redisTemplate.boundListOps("SeckillGoodsCountList_" +
seckillStatus.getGoodsId()).leftPush(seckillStatus.getGoodsId());

                //4)数据同步到Redis中

redisTemplate.boundHashOps("SeckillGoods_"+seckillStatus.getTime()).put(seckillStatus.getGoo
dsId(),seckillGoods);

                //清理排队标示

redisTemplate.boundHashOps("UserQueueCount").delete(seckillStatus.getUsername());

                //清理抢单标示

redisTemplate.boundHashOps("UserQueueStatus").delete(seckillStatus.getUsername());
            }
        }
    }

    /***
     * 消息监听
     * @param message
     */
    @Override
    public void onMessage(Message message) {
        try {
            //获取消息
            String content = new String(message.getBody());

            //将消息转换成SeckillStatus
            SeckillStatus seckillStatus = JSON.parseObject(content,SeckillStatus.class);

            //订单处理以及回滚库存处理
            rollbackOrder(seckillStatus);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

由于消息消费 类中使用到了Dubbo的服务提供对象，所以需要在applicationContext-service.xml中新增一个dubbo的包扫描，代码如下：

```
<dubbo:annotation package="com.qingcheng.consumer" />
<dubbo:consumer check="false" retries="0" timeout="8000"/>
```