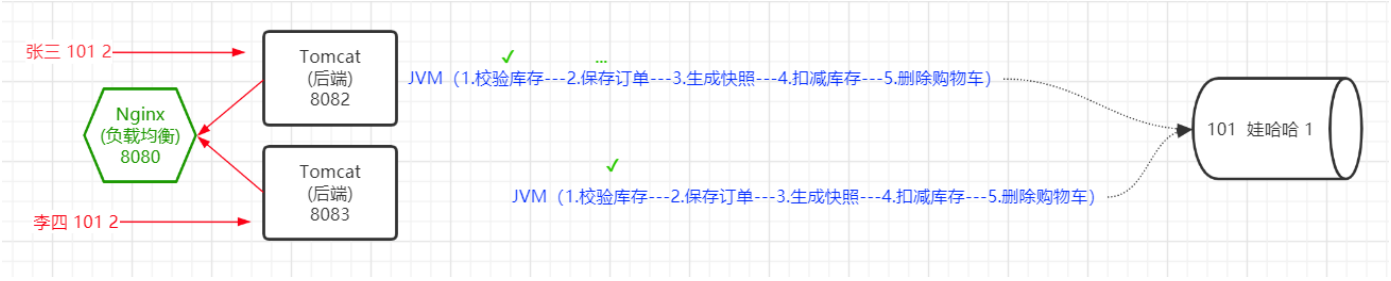
分布式锁

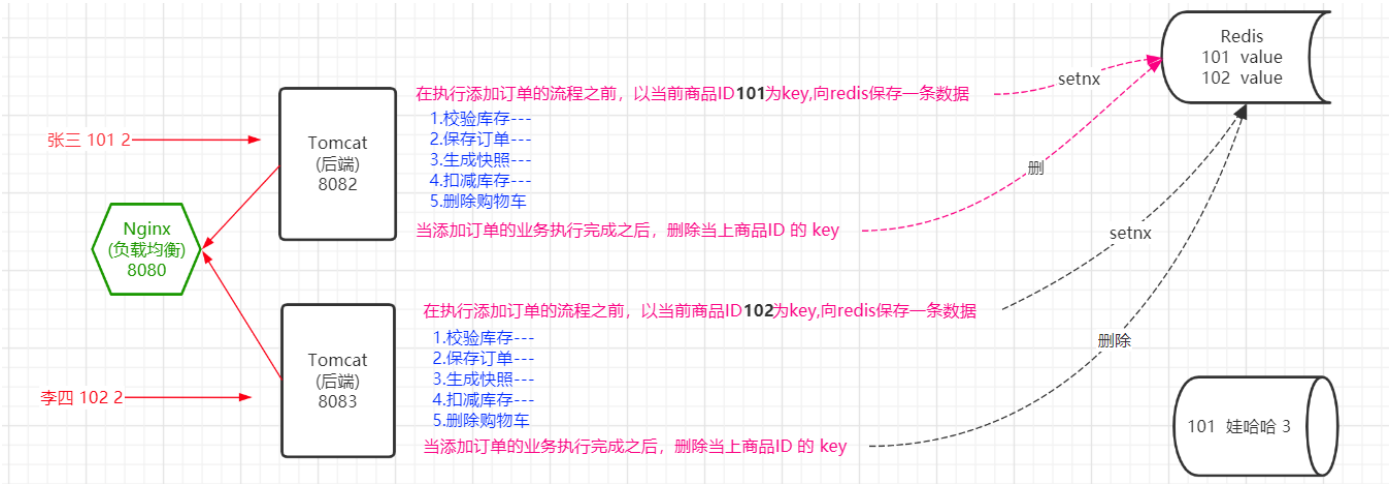
1.分布式并发问题

> 提交订单：商品超卖问题



2.如何解决分布式并发问题呢？

> 使用redis实现分布式锁



3.使用Redis实现分布式锁-代码实现

```
@Transactional
public Map<String,String> addOrder(String cids,Orders order) throws
SQLException {
    logger.info("add order begin...");
    Map<String, String> map = null;

    //1.校验库存：根据cids查询当前订单中关联的购物车记录详情（包括库存）
    String[] arr = cids.split(",");
    List<Integer> cidsList = new ArrayList<>();
    for (int i = 0; i < arr.length; i++) {
        cidsList.add(Integer.parseInt(arr[i]));
    }
```

```
12
13      //根据用户在购物车列表中选择的购物车记录的id 查询到对应的购物车记录
14      List<ShoppingCartVO> list =
shoppingCartMapper.selectShopcartByCids(cidsList);
15      //从购物车信息中获取到要购买的 skuId(商品ID) 以skuId为key写到redis中:   1
    2    3
16      boolean isLock = true;
17      String[] skuIds = new String[list.size()];  //记录已经锁定的商品的ID
18      for (int i = 0; i <list.size() ; i++) {
19          String skuId = list.get(i).getSkuId();  //订单中可能包含多个商品,
每个skuId表示一个商品
20          Boolean ifAbsent =
stringRedisTemplate.boundValueOps(skuId).setIfAbsent("fmmall");
21          if(ifAbsent){
22              skuIds[i] = skuId;
23          }
24          isLock = isLock && ifAbsent;
25      }
26
27      //如果isLock为true，表示“加锁”成功
28      if(isLock){
29          try{
30              //1.比较库存：当第一次查询购物车记录之后，在加锁成功之前，可能被其他
的并发线程修改库存
31              List<ShoppingCartVO> list =
shoppingCartMapper.selectShopcartByCids(cidsList);
32              boolean f = true;
33              String untitled = "";
34              for (ShoppingCartVO sc : list) {
35                  if (Integer.parseInt(sc.getCartNum()) >
sc.getSkuStock()) {
36                      f = false;
37                  }
38                  untitled = untitled + sc.getProductName() + ",";
39              }
40              if (f) {
41                  //2.添加订单
42                  //3.保存快照
43                  //4.修改库存
44                  //5.删除购物车
45          map = new HashMap<>();
46                  logger.info("add order finished...");
```

```
                map.put("orderId", orderId);
                map.put("productNames", untitled);
            }
        }catch(Exception e){
            e.printStackTrance();
        }finally{
            //释放锁
            for (int m = 0; m < skuIds.length ; m++) {
                String skuId = skuIds[m];
                if(skuId!=null && !"".equals(skuId)){
                    stringRedisTemplate.delete(skuId);
                }
            }
        }
        return map;
    }else{
        //表示加锁失败，订单添加失败
        // 当加锁失败时，有可能对部分商品已经锁定，要释放锁定的部分商品
        for (int i = 0; i < skuIds.length ; i++) {
            String skuId = skuIds[i];
            if(skuId!=null && !"".equals(skuId)){
                stringRedisTemplate.delete(skuId);
            }
        }
        return null;
    }
}
```

问题：

1.如果订单中部分商品加锁成功，但是某一个加锁失败，导致最终加锁状态失败——需要对已经锁定的部分商品释放锁

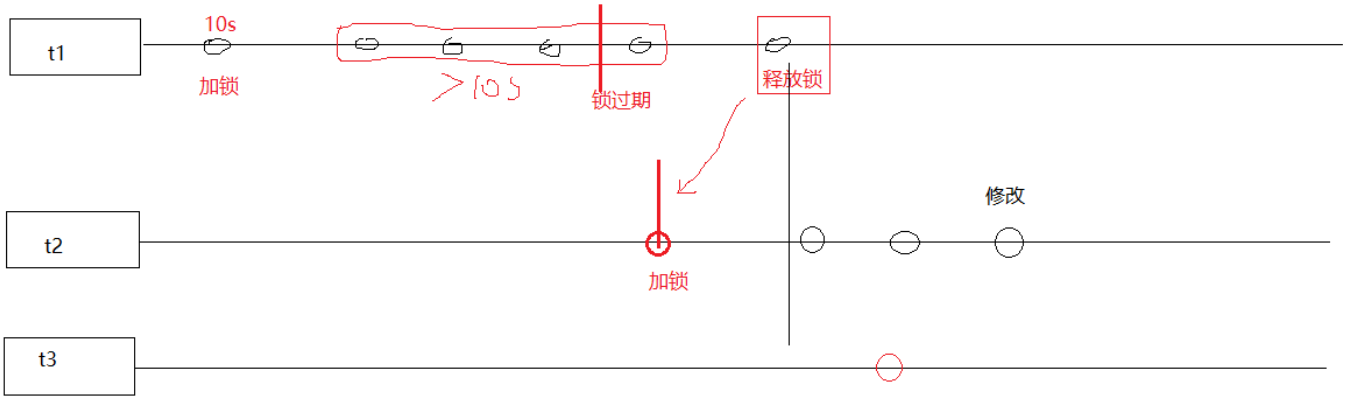
2.在成功加锁之前，我们根据购物车记录的id查询了购物车记录（包含商品库存），能够直接使用这个库存进行库存校验?

——不能，因为在查询之后加锁之前可能被并发的线程修改了库存；因此在进行库存比较之前需要重新查询库存。

3.当当前线程加锁成功之后，执行添加订单的过程中，如果当前线程出现异常导致无法释放锁，这个问题又该如何解决呢?

4.解决因线程异常导致无法释放锁的问题

> 解决方案：在对商品进行加锁时，设置过期时间，这样一来及时线程出现故障无法释放锁，在过期时间结束时也会自动“释放锁”



问题：当给锁设置了过期时间之后，如果当前线程t1因为特殊原因，在锁过期前没有完成业务执行，将会释放锁，同时其他线程（t2）就可以成功加锁了，当t2加锁成功之后，t1执行结束释放锁就会释放t2的锁,就会导致t2在无锁状态下执行业务。

5.解决因t1过期释放t2锁的问题

- 在加锁的时候，为每个商品设置唯一的value

```
boolean isLock = true;
String[] skuIds = new String[list.size()];  // ["1","2",null]
Map<String,String> values = new HashMap<>();
for (int i = 0; i <list.size() ; i++) {
    String skuId = list.get(i).getSkuId();  //订单中可能包含多个商品，每个skuId表示一个商品
    String value = UUID.randomUUID().toString();
    Boolean ifAbsent = stringRedisTemplate.boundValueOps(skuId).setIfAbsent(value, 10, TimeUnit.SECONDS);
    if(ifAbsent){
        skuIds[i] = skuId;
        values.put(skuId,value);
    }
    isLock = isLock && ifAbsent;
}
```

加锁

- 在释放锁的时候，先获取当前商品在redis中对应的value，如果获取的值与当前value相同，则释放锁

```
//释放锁
for (int m = 0; m < skuIds.length ; m++) {
    String skuId = skuIds[m];
    if(skuId!=null && !"".equals(skuId)){
        String s = stringRedisTemplate.boundValueOps(skuId).get(); //aaaa
        if(s!=null && s.equals(values.get(skuId))){
            stringRedisTemplate.delete(skuId);
        }
    }
}
```

释放锁之前，先查询当前商品的value

如果从redis中获取的值与生成的value是一致的，则表示此锁是当前线程加的锁，可以进行删除

问题：当释放锁的时候，在查询并判断“这个锁是当前线程加的锁”成功之后，正要进行删除时锁过期了，并且被其他线程成功加锁，一样会导致当前线程删除其他线程的锁。

- Redis的操作都是原子性的
- 要解决如上问题，必须保证查询操作和删除操作的原子性——使用lua脚本

使用lua脚本

- 在resources目录下创建unlock.lua,编辑脚本：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

- 配置Bean加载lua脚本

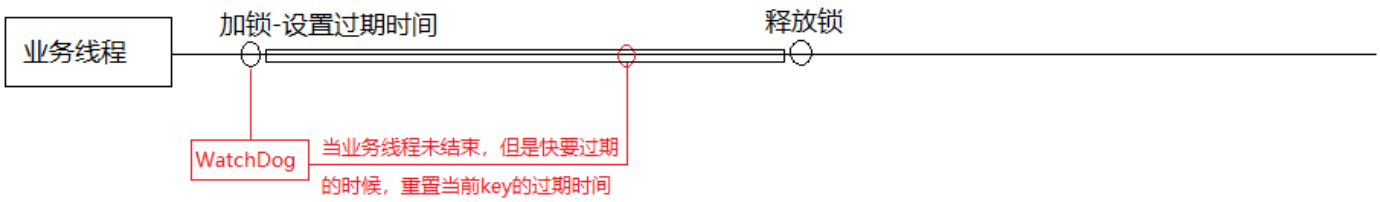
```
@Bean
public DefaultRedisScript<List> defaultRedisScript(){
    DefaultRedisScript<List> defaultRedisScript = new
DefaultRedisScript<>();
    defaultRedisScript.setResultType(List.class);
    defaultRedisScript.setScriptSource(new ResourceScriptSource(new
ClassPathResource("unlock.lua")));
    return  defaultRedisScript;
}
```

- 通过执行lua脚本解锁

```
@AutoWired
private DefaultRedisScript defaultRedisScript;

//执行lua脚本
List<String> keys = new ArrayList<>();
keys.add(skuId);
List rs = stringRedisTemplate.execute(defaultRedisScript,keys ,
values.get(skuId));
System.out.println(rs.get(0));
```

6.看门狗机制



看门口线程：用于给当前key延长过期时间，保证业务线程正常执行的过程中，锁不会过期。

7.分布式锁框架-Redisson

> 基于Redis+看门狗机制的分布式锁框架

7.1 Redisson介绍

Redisson在基于NIO的Netty框架上，充分的利用了Redis键值数据库提供的一系列优势，在Java实用工具包中常用接口的基础上，为使用者提供了一系列具有分布式特性的常用工具类。使得原本作为协调单机多线程并发程序的工具包获得了协调分布式多机多线程并发系统的能力，大大降低了设计和研发大规模分布式系统的难度。同时结合各富特色的分布式服务，更进一步简化了分布式环境中程序相互之间的协作

7.2 在SpringBoot应用中使用Redisson

- 添加依赖

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.12.0</version>
</dependency>
```

- 配置yml

```
redisson:
  addr:
    singleAddr:
      host: redis://47.96.11.185:6370
      password: 12345678
      database: 0
```

- 配置RedissonClient

```
@Configuration
public class RedissonConfig {

    @Value("${redisson.addr.singleAddr.host}")
    private String host;

    @Value("${redisson.addr.singleAddr.password}")
    private String password;

    @Value("${redisson.addr.singleAddr.database}")
    private int database;


    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();

 config.useSingleServer().setAddress(host).setPassword(password).setDatabase(database);
        return Redisson.create(config);
    }

}
```

- 在秒杀业务实现中注入RedissonClient对象

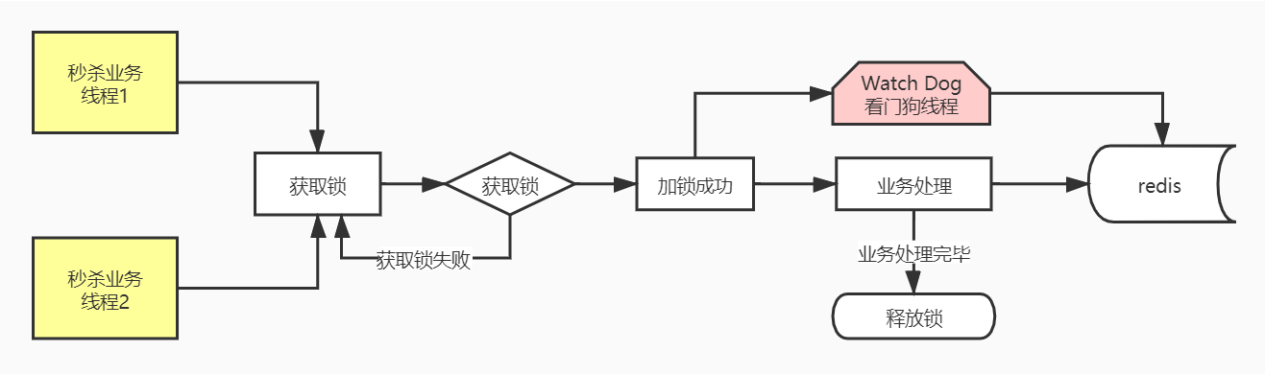
```

```

7.3 Redisson工作原理

- “看门狗”

Redisson工作原理图



7.4 Redisson使用扩展

7.4.1 Redisson单机连接

- application.yml

```yaml
redisson:
  addr:
    singleAddr:
      host: redis://47.96.11.185:6370
      password: 12345678
      database: 0
```

- RedissonConfig

```java
@Configuration
public class RedissonConfig {

    @Value("${redisson.addr.singleAddr.host}")
    private String host;

    @Value("${redisson.addr.singleAddr.password}")
    private String password;

    @Value("${redisson.addr.singleAddr.database}")
    private int database;


    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();
```

```
17
 config.useSingleServer().setAddress(host).setPassword(password).se
tDatabase(database);
18          return Redisson.create(config);
19      }
20
21  }
```

7.4.2 Redisson集群连接

- application.yml

```
1  redisson:
2    addr:
3      cluster:
4        hosts: redis://47.96.11.185:6370,...,redis://47.96.11.185:6373
5        password: 12345678
```

- RedissonConfig——RedissonClient对象

```
1  @Configuration
2  public class RedissonConfig {
3
4      @Value("${redisson.addr.cluster.hosts}")
5      private String hosts;
6      @Value("${redisson.addr.cluster.password}")
7      private String password;
8
9      /**
10      * 集群模式
11      * @return
12      */
13     @Bean
14     public RedissonClient redissonClient(){
15         Config config = new Config();
16         config.useClusterServers().addNodeAddress(hosts.split("
[,]"))
17                 .setPassword(password)
18                 .setScanInterval(2000)
19                 .setMasterConnectionPoolSize(10000)
20                 .setSlaveConnectionPoolSize(10000);
21         return Redisson.create(config);
```

```
    }

}
```

7.4.3 Redisson主从连接

- application.yml

```
redisson:
  addr:
    masterAndSlave:
      masterhost: redis://47.96.11.185:6370
      slavehosts:
redis://47.96.11.185:6371,redis://47.96.11.185:6372
      password: 12345678
      database: 0
```

- RedissonConfig --- RedissonClient

```
@Configuration
public class RedissonConfig3 {

    @Value("${redisson.addr.masterAndSlave.masterhost}")
    private String masterhost;
    @Value("${redisson.addr.masterAndSlave.slavehosts}")
    private String slavehosts;
    @Value("${redisson.addr.masterAndSlave.password}")
    private String password;
    @Value("${redisson.addr.masterAndSlave.database}")
    private int database;

    /**
     * 主从模式
     * @return
     */
    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();
        config.useMasterSlaveServers()
                .setMasterAddress(masterhost)
                .addSlaveAddress(slavehosts.split("[,]"))
                .setPassword(password)
```

```
                .setDatabase(database)
                .setMasterConnectionPoolSize(10000)
                .setSlaveConnectionPoolSize(10000);
        return Redisson.create(config);
    }

}
```

7.5 分布式锁总结

7.5.1 分布式锁特点

1、互斥性
和我们本地锁一样互斥性是最基本，但是分布式锁需要保证在不同节点的不同线程的互斥。

2、可重入性
同一个节点上的同一个线程如果获取了锁之后那么也可以再次获取这个锁。

3、锁超时
和本地锁一样支持锁超时，加锁成功之后设置超时时间，以防止线程故障导致不释放锁，防止死锁。

4、高效，高可用
加锁和解锁需要高效，同时也需要保证高可用防止分布式锁失效，可以增加降级。

redission是基于redis的，redis的故障就会导致redission锁的故障，因此redission支持单节点redis、reids主从、reids集群

5、支持阻塞和非阻塞
和 ReentrantLock 一样支持 lock 和 trylock 以及 tryLock(long timeOut)。

7.5.2 锁的分类

1、乐观锁与悲观锁

- 乐观锁
- 悲观锁

2、可重入锁和非可重入锁

- 可重入锁：当在一个线程中第一次成功获取锁之后，在此线程中就可以再次获取
- 非可重入锁

3、公平锁和非公平锁

- 公平锁：按照线程的先后顺序获取锁
- 非公平锁：多个线程随机获取锁

4、阻塞锁和非阻塞锁

- 阻塞锁：不断尝试获取锁，直到获取到锁为止
- 非阻塞锁：如果获取不到锁就放弃，但可以支持在一定时间段内的重试

——在一段时间内如果没有获取到锁就放弃

7.5.3 Redission的使用

1、获取锁——公平锁和非公平锁

```
//获取公平锁
RLock lock = redissonClient.getFairLock(skuId);

//获取非公平锁
RLock lock = redissonClient.getLock(skuId);
```

2、加锁——阻塞锁和非阻塞锁

```
//阻塞锁（如果加锁成功之后，超时时间为30s；加锁成功开启看门狗，剩5s延长过期时间）
lock.lock();
//阻塞锁（如果加锁成功之后，设置自定义20s的超时时间）
lock.lock(20,TimeUnit.SECONDS);

//非阻塞锁（设置等待时间为3s；如果加锁成功默认超时间为30s）
boolean b = lock.tryLock(3,TimeUnit.SECONDS);
//非阻塞锁（设置等待时间为3s；如果加锁成功设置自定义超时间为20s）
boolean b = lock.tryLock(3,20,TimeUnit.SECONDS);
```

3、释放锁

```
lock.unlock();
```

4、应用示例

```
//公平非阻塞锁
RLock lock = redissonClient.getFairLock(skuId);
boolean b = lock.tryLock(3,20,TimeUnit.SECONDS);
```

8.分布式锁释放锁代码优化

- 伪代码

```
HashMap map = null;
加锁
try{
    if(isLock){
        校验库存
        if(库存充足){
            保存订单
            保存快照
            修改库存
            删除购物车
            map = new HashMap();
            ...
        }
    }
}catch(Exception e){
  e.printStackTrace();
}finally{
    释放锁
}
return map;
```

- Java代码实现

```
/**
* 保存订单业务
*/
@Transactional
public Map<String, String> addOrder(String cids, Orders order)
throws SQLException {

    logger.info("add order begin...");
    Map<String, String> map = null;

```

```
    //1.校验库存：根据cids查询当前订单中关联的购物车记录详情（包括库存）
    String[] arr = cids.split(",");
    List<Integer> cidsList = new ArrayList<>();
    for (int i = 0; i < arr.length; i++) {
        cidsList.add(Integer.parseInt(arr[i]));
    }
    //根据用户在购物车列表中选择的购物车记录的id 查询到对应的购物车记录
    List<ShoppingCartVO> list =
shoppingCartMapper.selectShopcartByCids(cidsList);

    //加锁
    boolean isLock = true;
    String[] skuIds = new String[list.size()];
    Map<String, RLock> locks = new HashMap<>();  //用于存放当前订单的
锁
    for (int i = 0; i < list.size(); i++) {
        String skuId = list.get(i).getSkuId();
        boolean b = false;
        try {
            RLock lock = redissonClient.getLock(skuId);
            b = lock.tryLock(10, 3, TimeUnit.SECONDS);
            if (b) {
                skuIds[i] = skuId;
                locks.put(skuId, lock);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        isLock = isLock & b;
    }

    //如果isLock为true，表示“加锁”成功
    try {
        if (isLock){
            //1.检验库存
            boolean f = true;
            String untitled = "";
            list =
shoppingCartMapper.selectShopcartByCids(cidsList);
            for (ShoppingCartVO sc : list) {
                if (Integer.parseInt(sc.getCartNum()) >
sc.getSkuStock()) {
```

```
                    f = false;
                }
                untitled = untitled + sc.getProductName() + ",";
            }
            if (f) {
                //如果库存充足，则进行下订单操作
                logger.info("product stock is OK...");
                //2.保存订单
                order.setUntitled(untitled);
                order.setCreateTime(new Date());
                order.setStatus("1");
                //生成订单编号
                String orderId =
UUID.randomUUID().toString().replace("-", "");
                order.setOrderId(orderId);
                int i = ordersMapper.insert(order);

                //3.生成商品快照
                for (ShoppingCartVO sc : list) {
                    int cnum = Integer.parseInt(sc.getCartNum());
                    String itemId = System.currentTimeMillis() +
"" + (new Random().nextInt(89999) + 10000);
                    OrderItem orderItem = new OrderItem(itemId,
orderId, sc.getProductId(), sc.getProductName(),
sc.getProductImg(), sc.getSkuId(), sc.getSkuName(), new
BigDecimal(sc.getSellPrice()), cnum, new
BigDecimal(sc.getSellPrice() * cnum), new Date(), new Date(), 0);
                    orderItemMapper.insert(orderItem);
                    //增加商品销量
                }

                //4.扣减库存：根据套餐ID修改套餐库存量
                for (ShoppingCartVO sc : list) {
                    String skuId = sc.getSkuId();
                    int newStock = sc.getSkuStock() -
Integer.parseInt(sc.getCartNum());

                    ProductSku productSku = new ProductSku();
                    productSku.setSkuId(skuId);
                    productSku.setStock(newStock);

 productSkuMapper.updateByPrimaryKeySelective(productSku);
```

```
                }

                //5.删除购物车：当购物车中的记录购买成功之后，购物车中对应做删除操作
                for (int cid : cidsList) {
                    shoppingCartMapper.deleteByPrimaryKey(cid);
                }
                map = new HashMap<>();
                logger.info("add order finished...");
                map.put("orderId", orderId);
                map.put("productNames", untitled);
            }
        }
    }catch (Exception e){
        e.printStackTrace();
    }finally {
        //释放锁
        for (int i = 0; i < skuIds.length; i++) {
            String skuId = skuIds[i];
            if (skuId != null && !"".equals(skuId)) {
                locks.get(skuId).unlock();
                System.out.println("-----------------------unlock");
            }
        }
    }
    return map;
}
```

千锋教育Java教研院 关注公众号【Java架构栈】下载所有课程代码课件及工具 让技术回归本该有的纯静!