# 高并发下的订单与库存的处理

2018-10-22 22:16:47 [林老师带你学编程](https://me.csdn.net/linzhiqiang0316" \t "https://blog.csdn.net/linzhiqiang0316/article/details/_blank) 阅读数 850 文章标签： [架构](https://so.csdn.net/so/search/s.do?q=%E6%9E%B6%E6%9E%84&t=blog" \t "https://blog.csdn.net/linzhiqiang0316/article/details/_blank) 更多

分类专栏： [架构](https://blog.csdn.net/linzhiqiang0316/article/category/7655582" \t "https://blog.csdn.net/linzhiqiang0316/article/details/_blank)

问题：一件商品只有100个库存，现在有1000或者更多的用户来购买，每个用户计划同时购买1个到几个不等商品。如何保证库存在高并发的场景下是安全的。  
1.不多发  
2.不少发

下单涉及的一些步骤  
1.下单  
2.下单同时预占库存  
3.支付  
4.支付成功真正减扣库存  
5.取消订单  
6.回退预占库存

什么时候进行预占库存  
方案一：加入购物车的时候去预占库存。  
方案二：下单的时候去预占库存。  
方案三：支付的时候去预占库存。  
分析：  
方案一：加入购物车并不代表用户一定会购买,如果这个时候开始预占库存，会导致想购买的无法加入购物车。而不想购买的人一直占用库存。显然这种做法是不可取的。  
方案二：商品加入购物车后，选择下单，这个时候去预占库存。用户选择去支付说明了，用户购买欲望是比 方案一 要强烈的。订单也有一个时效，例如半个小时。超过半个小时后，系统自动取消订单，回退预占库存。  
方案三：下单成功去支付的时候去预占库存。只有100个用户能支付成功，900个用户支付失败。用户体验不好，就像你走了一条光明大道，一路通畅，突然被告知此处不通行。而且支付流程也是一个比较复杂的流程，如果和减库存放在一起，将会变的更复杂。

所以综上所述：  
选择方案二比较合理。

重复下单问题

用户点击过快，重复提交两次。

网络延时，用户刷新或者点击下单重复提交。

网络框架重复请求，某些网络框架，在延时比较高的情况下会自动重复请求。

用户恶意行为。

解决办法

在UI拦截，点击后按钮置灰，不能继续点击，防止用户，连续点击造成的重复下单。

在下单前获取一个下单的唯一token，下单的时候需要这个token。后台系统校验这个 token是否有效，才继续进行下单操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | /\*\*       \* 先生成 token 保存到 Redis       \* token 作为 key , 并设置过期时间 时间长度 根据任务需求       \* value 为数字 自增判断 是否使用过       \*       \* @param user       \* @return       \*/      public String createToken(User user) {          String key = "placeOrder:token:" + user.getId();          String token = UUID.randomUUID().toString();          //保存到Redis          redisService.set(key + token, 0, 1000L);          return token;      }        /\*\*       \* 校验下单的token是否有效       \* @param user       \* @param token       \* @return       \*/      public boolean checkToken(User user, String token) {          String key = "placeOrder:token:" + user.getId();          if (null != redisService.get(key + token)) {              long times = redisService.increment(key + token, 1);              if (times == 1) {                  //利用increment 原子性 判断是否 该token 是否使用                  return true;              } else {                  // 已经使用过了              }              //删除              redisService.remove(key + token);          }          return false;      } |

如何安全的减扣库存

同一个用户或者多个用户同时抢购一个商品的时候，我们如何做到并发安全减扣库存？

数据库操作商品库存：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | /\*\*   \* Created by Administrator on 2017/9/8.   \*/  public interface ProductDao extends JpaRepository<Product, Integer> {        /\*\*       \* @param pid 商品ID       \* @param num 购买数量       \* @return       \*/        @Transactional      @Modifying      @Query("update Product set availableNum = availableNum - ?2 , reserveNum = reserveNum + ?2 where id = ?1")      int reduceStock1(Integer pid, Integer num);        /\*\*       \* @param pid 商品ID       \* @param num 购买数量       \* @return       \*/        @Transactional      @Modifying      @Query("update Product set availableNum = availableNum - ?2 , reserveNum = reserveNum + ?2 where id = ?1 and  availableNum - ?2 >= 0")      int reduceStock2(Integer pid, Integer num);    } |

下单：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | /\*\*       \* 下单操作1       \*       \* @param req       \*/      private int place(PlaceOrderReq req) {          User user = userDao.findOne(req.getUserId());          Product product = productDao.findOne(req.getProductId());          //下单数量          Integer num = req.getNum();          //可用库存          Integer availableNum = product.getAvailableNum();          //可用预定          if (availableNum >= num) {              //减库存              int count = productDao.reduceStock1(product.getId(), num);              if (count == 1) {                  //生成订单                  createOrders(user, product, num);              } else {                  logger.info("库存不足 3");              }              return 1;          } else {              logger.info("库存不足 4");              return -1;          }      }         /\*\*       \* 下单操作2       \*       \* @param req       \*/      private int place2(PlaceOrderReq req) {          User user = userDao.findOne(req.getUserId());          Product product = productDao.findOne(req.getProductId());          //下单数量          Integer num = req.getNum();          //可用库存          Integer availableNum = product.getAvailableNum();          //可用预定          if (availableNum >= num) {              //减库存              int count = productDao.reduceStock2(product.getId(), num);              if (count == 1) {                  //生成订单                  createOrders(user, product, num);              } else {                  logger.info("库存不足 3");              }              return 1;          } else {              logger.info("库存不足 4");              return -1;          }      } |

方法1 ：

不考虑库存安全的写法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*           \* 方法 1           \* 减可用           \* 加预占           \* 库存数据不安全           \*           \* @param req           \*/          @Override          @Transactional          public void placeOrder(PlaceOrderReq req) {              place1(req);          } |

分析：  
在高并的场景下，假设库存只有 2 件 ，两个请求同时进来，抢购改商品，购买数量都是 2.  
A请求 此时去获取库存，发现库存刚好足够，执行扣库存下单操作。  
在 A 请求为完成的时候（事务未提交），B请求 此时也去获取库存，发现库存还有2. 此时也去执行扣库存，下单操作。

库存剩 2  ，但是卖出了 4 。最终数据库库存数量将变为 -2 ，所以库存是不安全的。

方法2 ：

这个操作可以保证库存数据是安全的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*           \* 方法 2           \* 减可用           \* 加预占           \* 库存数据不安全           \*           \* @param req           \*/          @Override          @Transactional          public void placeOrder(PlaceOrderReq req) {              place2(req);          } |

分析： 在方法1 的基础上 ，更新库存的语句，增加了可用库存数量 大于 0, availableNum - num >= 0 ;实质是使用了数据库的乐观锁来控制库存安全，在并发量不是很大的情况下可以这么做。但是如果是秒杀，抢购，瞬时流量很高的话，压力会都到数据库，可能拖垮数据库。

方法3：

该方法也可以保证库存数量安全。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\*\*       \* 方法 3       \* 采用 Redis 锁  通一个时间 只能一个 请求修改 同一个商品的数量       \* <p>       \* 缺点并发不高,同时只能一个用户抢占操作,用户体验不好！       \*       \* @param req       \*/      @Override      public void placeOrder2(PlaceOrderReq req) {          String lockKey = "placeOrder:" + req.getProductId();          boolean isLock = redisService.lock(lockKey);          if (!isLock) {              logger.info("系统繁忙稍后再试!");              return 2;          }          //place2(req);          place1(req);          //这两个方法都可以          redisService.unLock(lockKey);      } |

分析：

利用Redis 分布式锁， 强制控制 同一个商品，同时只能一个请求处理下单。 其他请求返回 ‘系统繁忙稍后再试！’；  
强制把处理请求串行化，缺点并发不高 ，处理比较慢，不适合抢购等方案 。  
用户体验也不好，明明看到库存是充足的，就是强不到。  
相比方案2减轻了数据库的压力。

方法4 ：

可以保证库存安全，满足高并发处理，但是相对复杂一点。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | /\*\*           \* 方法 4           \* 商品的数量 等其他信息 先保存 到 Redis           \* 检查库存 与 减少库存 不是原子性，  以 increment > 0 为准           \*           \* @param req           \*/          @Override          public void placeOrder3(PlaceOrderReq req) {              String key = "product:" + req.getProductId();              // 先检查 库存是否充足              Integer num = (Integer) redisService.get(key);              if (num < req.getNum()) {                  logger.info("库存不足 1");              }else{              //不可在这里下单减库存，否则导致数据不安全， 情况类似 方法1；              }              //减少库存              Long value = redisService.increment(key, -req.getNum().longValue());              //库存充足              if (value >= 0) {                  logger.info("成功抢购 ! ");                  //TODO 真正减 扣 库存 等操作 下单等操作  ,这些操作可用通过 MQ 或 其他方式                  place2(req);              } else {                  //库存不足，需要增加刚刚减去的库存                  redisService.increment(key, req.getNum().longValue());                  logger.info("库存不足 2 ");              }          } |

分析：  
利用Redis increment 的原子操作，保证库存安全。 事先需要把库存的数量等其他信息保存到Redis，并保证更新库存的时候，更新Redis。  
进来的时候 先 get 库存数量是否充足，再执行 increment。以 increment > 0 为准。  
检查库存 与 减少库存 不是原子性的。  
检查库存的时候技术库存充足也不可下单；否则造成库存不安全，原来类似 方法1.  
increment 是个原子操作，已这个为准。

redisService.increment(key, -req.getNum().longValue()) >= 0 说明库存充足，可以下单。

redisService.increment(key, -req.getNum().longValue()) < 0 的时候 不能下单，次数库存不足。并且需要 回加刚刚减去的库存数量，否则会导致刚才减扣的数量 一直买不出去。数据库与缓存的库存不一致。

次方法可以满足 高并抢购等一些方案，真正减扣库存和下单可以异步执行。

订单时效问题，订单取消等  
为保证商家利益，同时把商品卖给有需要的人，订单下单成功后，往往会有个有效时间。超过这个时间，订单取消，库存回滚。

为每笔订单设置 有效时间 可用参考这个：<http://jblog.top/article/details/254951>

订单取消后，可利用MQ 回退库存等，参考：http://jblog.top/article/details/255453