高并发核心技术 - 订单与库存

2017-09-07

- 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

    1. 用户点击过快，重复提交两次。
    2. 网络延时，用户刷新或者点击下单重复提交。
    3. 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。
    4. 用户恶意行为。

解决办法

1. 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。
2. 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.       * 先生成 token 保存到 Redis
3.       * token 作为 key , 并设置过期时间 时间长度 根据任务需求
4.       * value 为数字 自增判断 是否使用过
5.       *
6.       * @param user
7.       * @return
8.       */
9.      public String createToken(User user) {
10.         String key = "placeOrder:token:" + user.getId();
11.         String token = UUID.randomUUID().toString();
12.         //保存到Redis
13.         redisService.set(key + token, 0, 1000L);
14.         return token;
15.     }
16.
17.     /**
18.      * 校验下单的token是否有效
19.      * @param user
20.      * @param token
21.      * @return
22.      */
23.     public boolean checkToken(User user, String token) {
24.         String key = "placeOrder:token:" + user.getId();
25.         if (null != redisService.get(key + token)) {
```

```
26.             long times = redisService.increment(key + token, 1);
27.             if (times == 1) {
28.                 //利用increment 原子性 判断是否 该token 是否使用
29.                 return true;
30.             } else {
31.                 // 已经使用过了
32.             }
33.             //删除
34.             redisService.remove(key + token);
35.         }
36.         return false;
37.     }
```

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

```
    reserveNum = reserveNum + ?2 where id = ?1 and  availableNum - ?2 >=
    0")
26.     int reduceStock2(Integer pid, Integer num);
27.
28. }
```

下单:

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

```
40.          //可用库存
41.          Integer availableNum = product.getAvailableNum();
42.          //可用预定
43.          if (availableNum >= num) {
44.              //减库存
45.              int count = productDao.reduceStock2(product.getId(), num);
46.              if (count == 1) {
47.                  //生成订单
48.                  createOrders(user, product, num);
49.              } else {
50.                  logger.info("库存不足 3");
51.              }
52.              return 1;
53.          } else {
54.              logger.info("库存不足 4");
55.              return -1;
56.          }
57.      }
```

方法1：

不考虑库存安全的写法:

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

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

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.       * 方法 3
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
4.       * <p>
5.       * 缺点并发不高,同时只能一个用户抢占操作,用户体验不好!
6.       *
7.       * @param req
8.       */
9.      @Override
10.     public void placeOrder2(PlaceOrderReq req) {
11.         String lockKey = "placeOrder:" + req.getProductId();
12.         boolean isLock = redisService.lock(lockKey);
13.         if (!isLock) {
```

```
14.                 logger.info("系统繁忙稍后再试!");
15.                 return 2;
16.             }
17.
18.             //place2(req);
19.             place1(req);
20.             //这两个方法都可以
21.             redisService.unLock(lockKey);
22.         }
```

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.       * 方法 4
 3.       * 商品的数量 等其他信息 先保存 到 Redis
 4.       * 检查库存 与 减少库存 不是原子性,    以 increment > 0 为准
 5.       *
 6.       * @param req
 7.       */
 8.      @Override
 9.      public void placeOrder3(PlaceOrderReq req) {
10.          String key = "product:" + req.getProductId();
11.          // 先检查 库存是否充足
12.          Integer num = (Integer) redisService.get(key);
13.          if (num < req.getNum()) {
14.              logger.info("库存不足 1");
15.          }else{
16.          //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;
17.          }
18.          //减少库存
19.          Long value = redisService.increment(key, -
    req.getNum().longValue());
20.          //库存充足
```

```
21.          if (value >= 0) {
22.              logger.info("成功抢购 ! ");
23.              //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
24.              place2(req);
25.          } else {
26.              //库存不足，需要增加刚刚减去的库存
27.              redisService.increment(key, req.getNum().longValue());
28.              logger.info("库存不足 2 ");
29.          }
30.      }
```

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。

如果你觉得不错就赞赏一下吧,您的支持是我的动力!