压测库存超卖-问题跟进

问题描述:

0427 下午压测提交订单接口时,测试发现100并发下库存的订单占用数会变成负数

问题追踪:

写法一（原始写法）

针对sku和订单code加锁,然后进行查询,在内存里计算库存,再写入数据库,再释放锁

Java

```
//开启事务
//从redis拿订单里的商品信息
String records = InventoryStorageRedisManager.getObjectValue(StockCache.OCCUPIED_STOCK + inventoryGoodsOutDTO.getSourceCode());
//对占用商品skuid上锁
InventoryDistributedLockManager.lock(RedisKey.STOCK_UPDATE_LOCK_KEY + inventorySkuDTO.getSkuId());
//查询实时库存信息
Inventory inventory = inventoryDao.selectBySkuIdAndChannelId(inventorySkuDTO.getSkuId(), inventoryLockDTO.getChannelId());
//计算渠道扣库存
lock(Integer quantity) {
    orderLockedCount += quantity;
    avaliableCount -= quantity;
}
//更新库存信息
inventoryDao.batchUpdateInventory(inventoryList);
//释放锁
InventoryDistributedLockManager.unlock(RedisKey.STOCK_UPDATE_LOCK_KEY + inventorySkuDTO.getSkuId());
//提交事务
```

0427 22:00 - 0428 09:30 怀疑删除redis里的订单的库存数据失败,验证排除

0427 22:00 - 0428 09:30 打印首次出现库存变负的日志,找到第一条数据,但是无法帮助解决问题

0427 22:00 - 0428 09:30 针对订单号加redis锁,无法解决问题

0427 22:00 - 0428 09:30 针对sku加锁时先进行去重,无法解决问题

0427 22:00 - 0428 09:30 去除针对订单号加的redis锁,将redission对skuid的加锁方式改成循环 *tryLock,无法*

~~0609 通过redis加锁，分析发现如果redis锁在事务之内，则redis锁会在事务提交之前释放~~

~~*解决问题*~~

0427 22:00 - 0428 09:30 加锁后重新查询sku的剩余渠道库存,再进行计算,无法解决问题

0427 22:00 - 0428 09:30 怀疑失败,捕捉redission加/释放锁异常,并打印,验证排除

0427 22:00 - 0428 09:30 修复父渠道的子渠道可用数的计算错误

0427 22:00 - 0428 09:30 修改出库方法的事务隔离级别为序列化,无法解决问题,回退

0428 09:30 - 0428 13:47 将库存变动的计算改到数据库进行,库存字段添加非负约束,并删除skuid锁,问题(被规避)解决,写法二

写法二（改进写法，目前线上版本）

Java

```
1  //开启事务
2  //锁定订单号
3  InventoryDistributedLockManager.lock(RedisKey.STOCT_SOURCE_CODE_LOCK +
   inventoryGoodsOutDTO.getSourceCode());
4  //再次校验redis里的订单是否存在
5  if
   (ObjectUtils.isEmpty(InventoryStorageRedisManager.getObjectValue(StockCache.OC
   CUPIED_STOCK + inventoryGoodsOutDTO.getSourceCode()))){
6      return new ApiResp(ErrorCodes.OutStockErr.OUT_RESOURCE_NOT_EXIT,"出库资源不
   存在");
7  }
8  //把库存变动量传入数据库,计算并替换
9  inventoryDao.updateInventoryForOut(inventory,updateCount);
10 //释放订单号锁
11 InventoryDistributedLockManager.unlock(RedisKey.STOCT_SOURCE_CODE_LOCK +
   inventoryGoodsOutDTO.getSourceCode());
12 //释放事务
13
```

0428 09:30 - 0512 12:00 去除相关锁,提高压测性能

0513 09:30 - 0517 14:00 排查业务联路上下游,发现有定时任务做库存释放操作,怀疑是数据问题,经排查排除

0518 09:30 - 0615 19:12 定位超卖问题为分布式锁失效,失效原因排查并验证为事务对锁的影响,改成写法三,问题解决

写法三（最新写法）

为保证业务方法的事务起效，加锁方法中通过applicationContext.getBean的方式调用业务逻辑方法

也可在类中注入本类,使用变量名调用

Java

```java
@Override
public ApiResp outOfInventoryOccupiedByOrderLock(InventoryGoodsOutDTO inventoryGoodsOutDTO){
    try {
        //对所有释放商品上锁
        for (OrderInventory record : orderInventoryList) {
            InventoryDistributedLockManager.lock(RedisKey.STOCK_UPDATE_LOCK_KEY + record.getSkuId());
        }
        //商品出库
        //为使事务生效,使用applicationContext.getBean调用
        //也可在类中注入本类,使用变量名调用
        resp = applicationContext.getBean(InventoryCmdServiceImpl.class).outOfInventoryOccupiedByOrder(inventoryGoodsOutDTO, orderInventoryList);
    }catch (ApiException e){
        throw new ApiException(e.getCode(),e.getMessage());
    } finally {
        //释放锁
        for (OrderInventory record : orderInventoryList) {
            InventoryDistributedLockManager.unlock(RedisKey.STOCK_UPDATE_LOCK_KEY + record.getSkuId());
        }
    }
     return resp;
 }


@Override
@Transactional(rollbackFor = ShouldRollbackException.class)
public ApiResp outOfInventoryOccupiedByOrder(InventoryGoodsOutDTO inventoryGoodsOutDTO, List<OrderInventory> orderInventoryList) {
    //出库
    return orderInventoryCmdService.outOfInventoryOccupiedBySourcecodeAndType(inventoryGoodsOutDTO,orderInventoryList);
}
```

问题分析:

写法一

针对sku和订单code加锁,然后进行查询,在内存里计算库存,再写入数据库,再释放锁

Java

```
//开启事务
//从redis拿订单里的商品信息
String records = InventoryStorageRedisManager.getObjectValue(StockCache.OCCUPIED_STOCK + inventoryGoodsOutDTO.getSourceCode());
//对占用商品上锁
InventoryDistributedLockManager.lock(RedisKey.STOCK_UPDATE_LOCK_KEY + inventorySkuDTO.getSkuId());
//查询实时库存信息
Inventory inventory = inventoryDao.selectBySkuIdAndChannelId(inventorySkuDTO.getSkuId(), inventoryLockDTO.getChannelId());
//计算渠道扣库存
lock(Integer quantity) {
    orderLockedCount += quantity;
    avaliableCount -= quantity;
}
//更新库存信息
inventoryDao.batchUpdateInventory(inventoryList);
//释放锁
InventoryDistributedLockManager.unlock(RedisKey.STOCK_UPDATE_LOCK_KEY + inventorySkuDTO.getSkuId());
//释放事务
```

问题

1、订单里面的商品循环加锁,可能导致锁等待,极端情况会死锁,所以需要按照商品id大小顺序加锁,但是并不能完全解决问题,最好使用乐观锁并重试;

2、查询库存信息在内存里计算数量的变动,锁失效情况下会导致数量错误,CAS或者放在数据库里计算比较好;

3、商品上锁之后未重新查询redis里的商品信息,无法保证“单例”

写法二

Java

```java
//开启事务
//锁定订单号
InventoryDistributedLockManager.lock(RedisKey.STOCT_SOURCE_CODE_LOCK + inventoryGoodsOutDTO.getSourceCode());
//再次校验redis里的订单是否存在
if (ObjectUtils.isEmpty(InventoryStorageRedisManager.getObjectValue(StockCache.OCCUPIED_STOCK + inventoryGoodsOutDTO.getSourceCode()))){
    return new ApiResp(ErrorCodes.OutStockErr.OUT_RESOURCE_NOT_EXIT,"出库资源不存在");
}
//把库存变动量传入数据库,计算并替换
inventoryDao.updateInventoryForOut(inventory,updateCount);
//释放订单号锁
InventoryDistributedLockManager.unlock(RedisKey.STOCT_SOURCE_CODE_LOCK + inventoryGoodsOutDTO.getSourceCode());
//提交事务

```

好处

1、去除代码里商品的锁,免去内存里的库存变动计算,将线程的安全交给数据库,并且把库存的表字段设置成unsign作为超卖的兜底.

2、将库存的变动改成消费模型,规避了分布式锁失效的问题

问题

由于数据库承担了所有的并发,并发高的时候会压垮数据库,所以只能临时处理

写法三

为保证业务方法的事务起效，加锁方法中通过applicationContext.getBean的方式调用业务逻辑方法

也可在类中注入本类,使用变量名调用

Java

```java
@Override
public ApiResp outOfInventoryOccupiedByOrderLock(InventoryGoodsOutDTO inventoryGoodsOutDTO){
    try {
        //对所有释放商品上锁
        for (OrderInventory record : orderInventoryList) {
            InventoryDistributedLockManager.lock(RedisKey.STOCK_UPDATE_LOCK_KEY + record.getSkuId());
        }
        //商品出库
        //为使事务生效,使用applicationContext.getBean调用
        //也可在类中注入本类,使用变量名调用
        resp = applicationContext.getBean(InventoryCmdServiceImpl.class).outOfInventoryOccupiedByOrder(inventoryGoodsOutDTO, orderInventoryList);
    }catch (ApiException e){
        throw new ApiException(e.getCode(),e.getMessage());
    } finally {
        //释放锁
        for (OrderInventory record : orderInventoryList) {
            InventoryDistributedLockManager.unlock(RedisKey.STOCK_UPDATE_LOCK_KEY + record.getSkuId());
        }
    }
     return resp;
 }


@Override
@Transactional(propagation = Propagation.REQUIRES_NEW)
public ApiResp outOfInventoryOccupiedByOrder(InventoryGoodsOutDTO inventoryGoodsOutDTO, List<OrderInventory> orderInventoryList) {
    //出库
    return orderInventoryCmdService.outOfInventoryOccupiedBySourcecodeAndType(inventoryGoodsOutDTO,orderInventoryList);
}
```

好处

1、正确使用了分布式锁,对数据库的压力加了一层过滤

2、可以将数据放在内存计算,减少数据库压力

解决方案:

1、用数据库事务保证并发安全,库存变更的计算在数据库完成

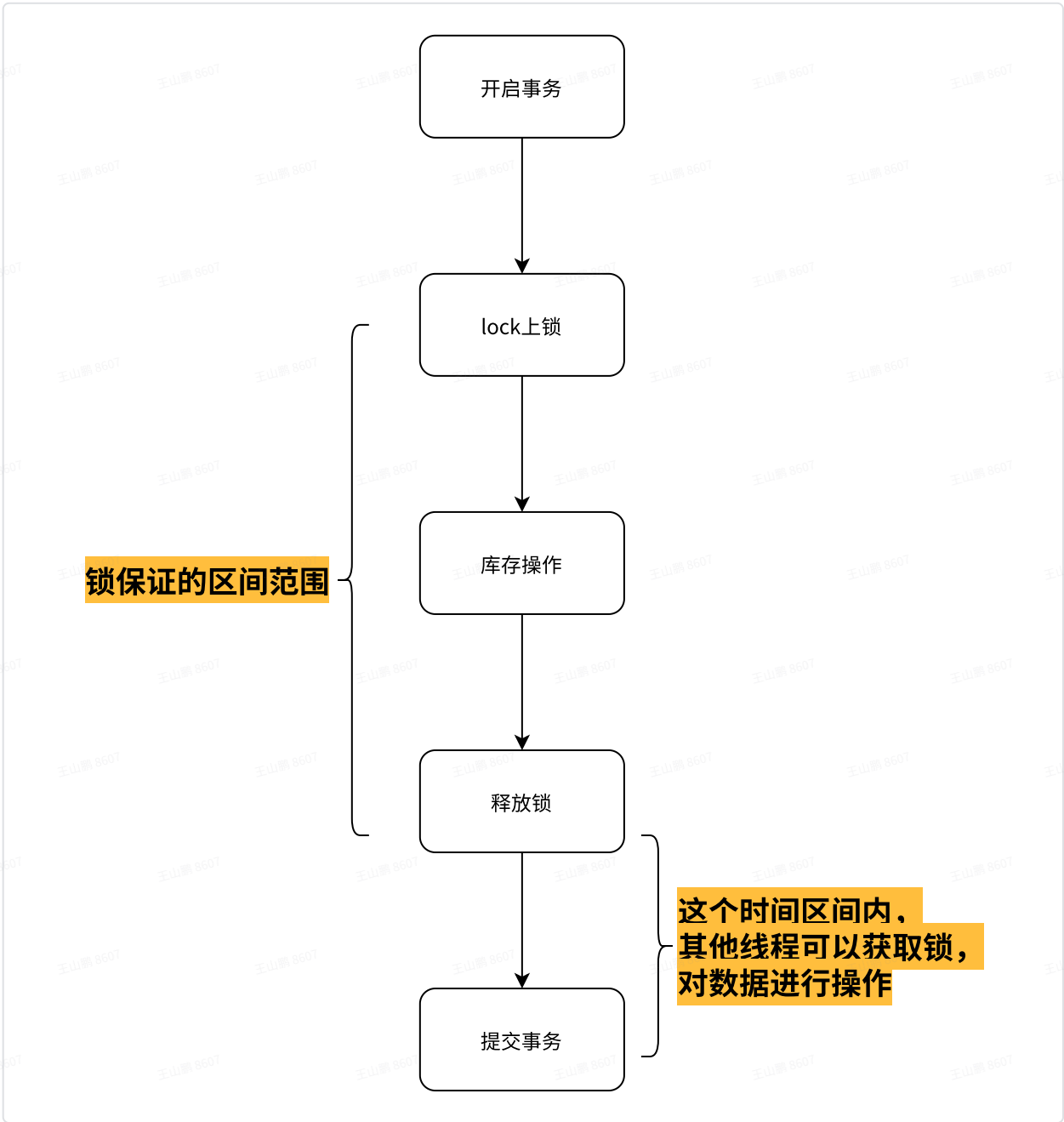
2、修改表字段为unsigned,作为保险

3、加分布式锁放在业务的事务之外,保证业务逻辑事务提交之前，redis锁不会被释放。

4、事务只能针对mysql的操作,不操作第三方

锁失效原因分析

现有逻辑，通过redis锁保证业务安全，分析发现，如果redis锁放在事务内，则会在事务提交之前进行锁的释放，导致数据不安全，应该把redis锁提到事务外边。



解决方案：

通过将redis锁提到事务的外边，保证业务逻辑事务提交之前，redis锁不会被释放。

知识点

@Transcational的生效过程及原理

Plain Text

```
@Nullable
protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass, TransactionAspectSupport.InvocationCallback invocation) throws Throwable {

    .......

    TransactionAspectSupport.TransactionInfo txInfo = this.createTransactionIfNecessary(ptm, txAttr, joinpointIdentification);

    Object retVal;
    try {
        retVal = invocation.proceedWithInvocation();
    } catch (Throwable var20) {
        this.completeTransactionAfterThrowing(txInfo, var20);
        throw var20;
    } finally {
        this.cleanupTransactionInfo(txInfo);
    }

    if (retVal != null && vavrPresent && TransactionAspectSupport.VavrDelegate.isVavrTry(retVal)) {
        TransactionStatus status = txInfo.getTransactionStatus();
        if (status != null && txAttr != null) {
            retVal = TransactionAspectSupport.VavrDelegate.evaluateTryFailure(retVal, txAttr, status);
        }
    }
    this.commitTransactionAfterReturning(txInfo);
    return retVal;
}
```

@Transcational事务失效的几种场景

- @Transcational是通过动态代理实现的, 本类方法的直接调用或者this调用不走代理,会导致事务失效
- @Transcational是通过动态代理实现的,事务方法访问修饰符非public不走代理，导致事务失效
- @Transactional注解的方法抛出的异常不是spring的事务支持的异常，导致事务失效
- 数据表本身是不支持事务，导致事务失效
- @Transactional注解所在的类没有被spring管理，导致事务失效
- catch掉异常之后，没有再次抛出异常，导致事务失效
- 多线程调用导致事务失效(新起线程操作数据库)

事务的传递

- propagation = Propagation.REQUIRES_NEW：无论当前存不存在事务，都创建新事务进行执行。
- propagation = Propagation.SUPPORTS：如果当前存在事务，就加入该事务；如果当前不存在事务，就以非事务执行
- propagation = Propagation.NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
- propagation = Propagation.NESTED：如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则按REQUIRED属性执行。
- propagation = Propagation.MANDATORY：如果当前存在事务，就加入该事务；如果当前不存在事务，就抛出异常。
- propagation = Propagation.NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

事务对分布式锁的影响

@Transcational注解的方法执行完毕之后才会提交事务, 锁的释放放在@Transcational注解的方法里的话,会导致先释放锁后提交事务,并发情况下可能有别的事务抢到锁,读到未提交事务前的数据

分布式锁的注意事项

- 加锁的顺序要一致
- 释放时“先进后出”
- 加锁要放在共享数据的查询及操作之前,释放要放在之后
- 加锁要放在事务之前,释放要在事务提交之后

事务导致锁失效扩展

将redis上锁提取到业务逻辑代码之外，进行压测，线程依然不安全，发现service类上加了@transaction注解，redis加锁过程和业务逻辑依然在一个事务之内。所以增加了propagation = Propagation.REQUIRES_NEW.

Java

```
1
2  @Slf4j
3  @Service
4  @Transactional(rollbackFor = ShouldRollbackException.class)
5  public class InventoryCmdServiceImpl implements InventoryCmdService {
6  }
```

~~推荐方案一~~

~~方案一：依然将事务委托spring管理~~

~~将类上的@Transaction注解去掉，需要事务的方法上单独加~~

Java

```
1
2  @Slf4j
3  @Service
4  //@Transactional(rollbackFor = ShouldRollbackException.class)
5  public class InventoryCmdServiceImpl implements InventoryCmdService {
6  }
```

~~方案二：自己手动管理事务~~

JavaScript

```
DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();
DefaultTransactionDefinition transDefinition = new DefaultTransactionDefinition();
 //开启新事务
 transDefinition.setPropagationBehavior(DefaultTransactionDefinition.PROPAGATION_REQUIRES_NEW);
 TransactionStatus transStatus = transactionManager.getTransaction(transDefinition);
 try {
          //库存操作
          .......
          transactionManager.commit(transStatus);
      } catch (Exception e) {
          transactionManager.rollback(transStatus);
      }finally {
          if (transStatus != null && transStatus.isNewTransaction()
                  && !transStatus.isCompleted()) {
              transactionManager.commit(transStatus);
           }
      }
  }
```

循环加锁的解决方案 TODO

事务只能针对mysql的操作,不操作第三方

多次操作数据库需要保证原子性的时候才需要使用事务

压测性能优化-问题跟进

问题描述:

压测下单业务时，库存服务耗时长，库存占用400多毫秒，出库400多毫秒,总计1100多毫秒。

问题追踪:

1、删除对skuid的加锁,总耗时降低为900多毫秒,业务正常

2、代码层面通过对库存批量更新的方式，时间提升到20多毫秒，但是出现死锁问题严重，出库时，数据安全无法得到保证。

问题分析：

经过排查发现，由于mysql锁竞争严重，导致耗费资源和时间过多。

后续计划:

1、从设计层面，子渠道的库存更新不再和父渠道库存联动，可减少mysql锁竞争，此方案待验证。

2、将商品的库存放在redis,把并发的压力转移到redis

20220615评审

- [ ] 不同端操作库存的优先级
- [x] ~~库存的变动时发消息使用canal~~
- [x] ~~事务只能针对mysql的操作,不操作第三方~~
- [ ] 将渠道的库存及变动拆表
- [ ] 循环加锁的解决方案
- [ ] 将需要加事务的方法放在manager里,或者是一个私有方法
- [ ] 渠道放一起的时候不同操作可能会造成碰撞
- [ ] 扣成负的也是库存的一个基本问题
- [ ] 事务只能保证操作的一致性,不能保证数据一致性,及扣成负的
- [ ] 事务里面不能操作第三方
- [ ] 什么时候使用锁什么时候使用事务?