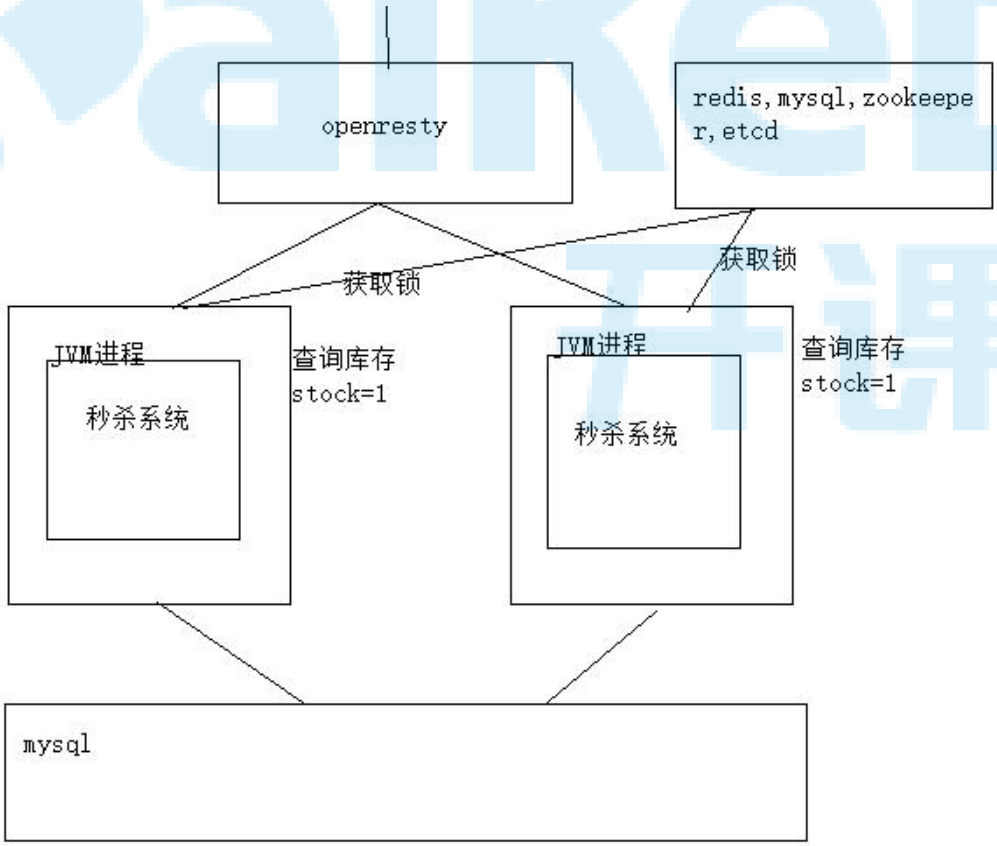
秒杀下单优化&最终消息一致&流量洪峰限流

今日课程内容主题：

1、分布式锁在下单业务中应用
2、分布式锁应用优化
3、下单业务进行优化处理（异步实现）
4、下单业务优化—缓存优化
5、数据一致性问题（缓存，数据库一致性）
6、最终消息一致性解决数据库一致性问题

1 分布式锁应用

1.1 使用分布式锁原因？



在集群模式下部署（分布式部署模式），必须使用分布式锁解决客户端对共享资源互斥访问，否则将会存在共享资源脏读的现象，从而出现超卖的现象；

定义：分布式锁定义： 第三方加锁的东西（锁就是状态标识符），标识客户端获取到了资源的执行权限；在分布式部署的模式下，有多个客户端访问共享资源，此时多个客户端就

必须争用一个标识符，获取到标识符，就表示客户端获取到这把锁，就具有业务执行权限；

分布式应用场景：保证多个客户端对共享资源进行互斥访问（修改）

1、秒杀

2、抢火车票

3、退款—防止重复退款—接口幂等性设计

1.2 锁和事务区别？

事务： 事务底层就是锁，保持事务和事务之间隔离性，多个事务之间是使用锁进行隔离的；

事务： 保证一次操作要么都成功，要么都失败 ----- 一次回话 ，叫做一个事务单元

锁： 应对在多线程模式下，防止共享资源出现并发修改的情况，锁是为多线程而生的；

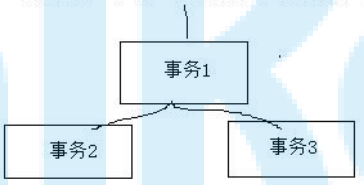
区别：

1、一次（一个线程）回话，就是一个事务

2、锁必须存在与多个线程情况下，对共享资源的保护，防止并发修改

分布式事务：

事务特性：保证一次回话要么全部成功，要么全部失败



就是保证事务在夸服务的事务要么都成功，要么都失败

进程之间的事务

分布式锁：

1、第三方加锁的服务

2、保证不同客户端（进程）之间对共享资源互斥访问，防止多个客户端的并发修改

1.3 Mysql 分布式锁

1）悲观锁 ： for update

使用 for update 方式，实现分布式条件下库存控制；保证多客户端之间访问的互斥；

```
@Select(value = "select * from tb_seckill_goods where id = #{seckillId} for update")
@Results(value = {
        @Result(column = "id",property = "id"),
        @Result(column = "product_id",property = "productId"),
        @Result(column = "item_id",property = "itemId"),
        @Result(column = "image",property = "image"),
        @Result(column = "images",property = "images"),
        @Result(column = "title",property = "title"),
        @Result(column = "info",property = "info"),
        @Result(column = "price",property = "price"),
        @Result(column = "cost_price",property = "costPrice"),
        @Result(column = "unit_name",property = "unitName"),
        @Result(column = "postage",property = "postage"),
        @Result(column = "add_time",property = "addTime"),
        @Result(column = "status",property = "status"),
        @Result(column = "start_time_date",property = "startTimeDate"),
        @Result(column = "end_time_date",property = "endTimeDate"),
        @Result(column = "stock",property = "stock"),
        @Result(column = "mark",property = "mark"),
        @Result(column = "stock_count",property = "stockCount"),
        @Result(column = "description",property = "description")
})
TbSeckillGoods selectByPrimaryKeyBySQLLock(Long seckillId);
```

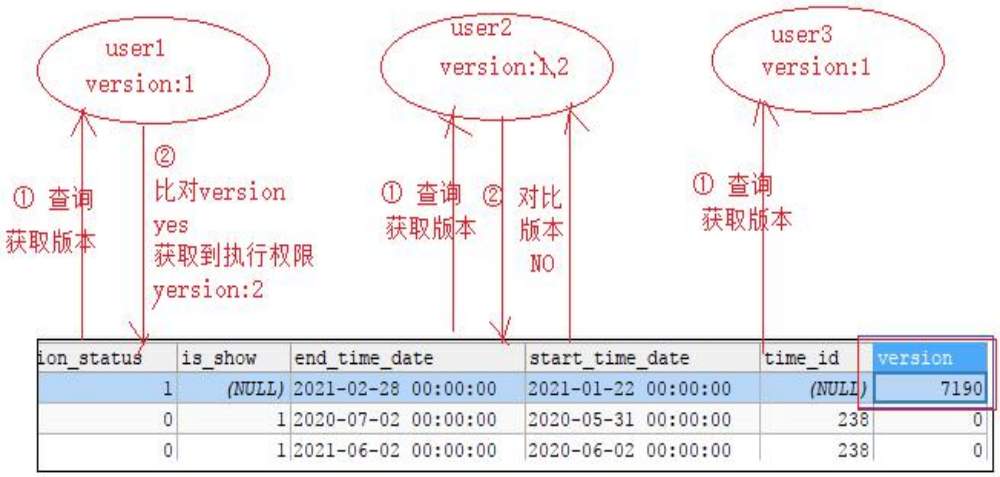
Mysql 分布式锁：使用结束，在分布式环境下进行一个测试，验证分布式是否可以在分布式模式下控制库存

验证结构： 数据库库存完美的实现控制，在 1000 个 users threads 测试下，1000 个库存被销售一空，说明么出现一个超卖的商品；

	ot_price	give_integral	sort	stock	stock_count	sales	unit
0	(NULL)	(NULL)	1	100	0	66720	节

MySQL 悲观锁分布式模式下实现的部署，TPS 吞吐能力：（2 台集群模式下）

2）乐观锁（版本模式）



ion_status	is_show	end_time_date	start_time_date	time_id	version
1	(NULL)	2021-02-28 00:00:00	2021-01-22 00:00:00	(NULL)	7190
0	1	2020-07-02 00:00:00	2020-05-31 00:00:00	238	0
0	1	2021-06-02 00:00:00	2020-06-02 00:00:00	238	0

乐观锁的模式，有失败的一个现象（如果版本匹配失败，没有执行权限，返回没有执行相关的业务操作）

例如： 1000 threads , 300 个比对版本成功的，700 个下单失败了!!（秒杀情况下，可以允许设置）

乐观锁的实现方式：

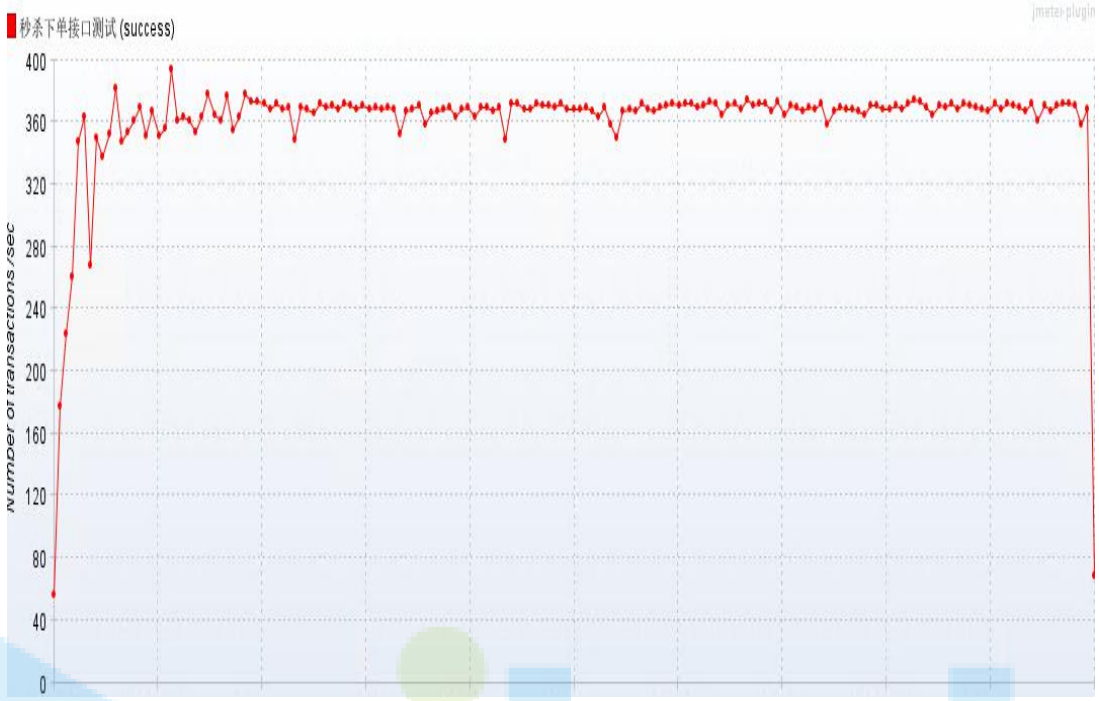
```
@Update(value = "UPDATE tb_seckill_goods SET stock_count=
stock_count-1,version=version+1 WHERE id = #{seckillId} AND
version = #{version}")
int updateSeckillGoodsByPrimaryKeyByVersion(@Param("seckillId")
Long seckillId, @Param("version") Integer version);
```

在乐观锁模式下：TPS 性能 是 悲观锁的 10 倍以上；

1.4 Redis 分布式锁

Redis 分布式锁：Redisson 框架实现分布式锁；

Redis 分布式锁单机上（单机版 redis 分布式锁）：



2 下单性能优化

2.1 业务改造思考

```
// 从数据库查询商品数据
// 优化一： 从缓存中获取数据
TbSeckillGoods seckillGoods =
seckillGoodsMapper.selectByPrimaryKey(killId);
//库存扣减

// 优化二：操作缓存，先不考虑数据一致性问题
seckillGoods.setStockCount(seckillGoods.getStockCount() - 1);
//更新库存
seckillGoodsMapper.updateByPrimaryKeySelective(seckillGoods);


//下单
// 优化三：写操作，异步的操作
TbSeckillOrder order = new TbSeckillOrder();
order.setSeckillId(killId);
order.setUserId(userId);
```

```
order.setCreateTime(new Date());
order.setStatus("0");
order.setMoney(seckillGoods.getCostPrice());

//保存订单
seckillOrderMapper.insertSelective(order);
```

2.2 缓存改造

首先需求把秒杀商品存储在 Redis 缓存中，同时把秒杀商品库存单独存储在 Redis 服务器中；

Redis

```
# redis 商品
key: seckill_goods_1
value: {"id":1,name:vivo,stock:2}

# 库存
key: seckill_goods_stock_1
value:2
```

实现下单业务优化改造工作：

```
// 1、 从缓存中获取秒杀商品数据
TbSeckillGoods seckillGoods = (TbSeckillGoods)
redisTemplate.opsForValue().get("seckill_goods_"+killId);
//2、利用 redis 原子性操作扣减库存，不需要上锁
boolean res = reduceStock(killId);


//3、异步实现（blockingQueue,disruptor,rocketMQ 队列实现异步）
//下单
TbSeckillOrder order = new TbSeckillOrder();
// 队列实现异步下单操作
Boolean produce = SeckillQueue.getMailQueue().produce(order);

if(!produce){
   return HttpResult.error("秒杀失败");
```

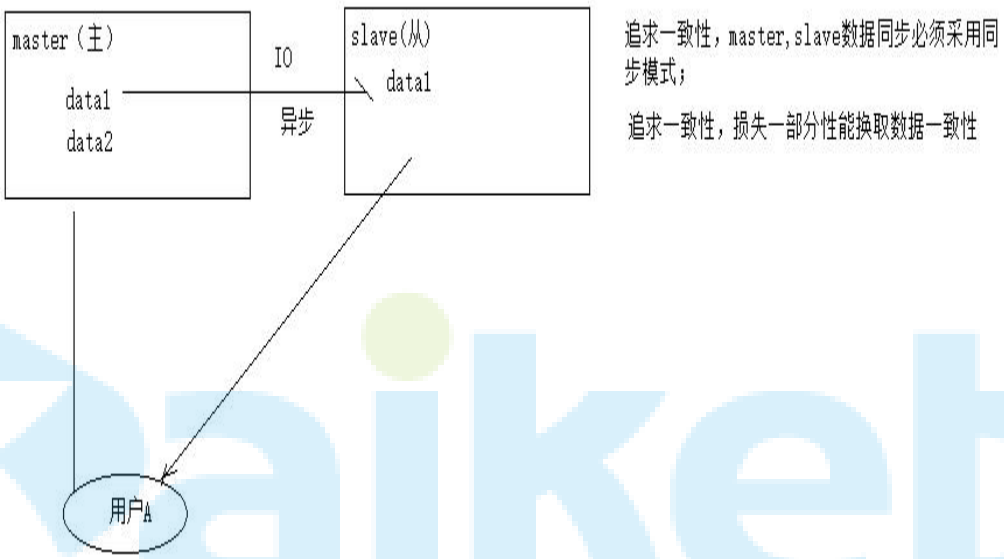
```
}

return HttpResult.ok("秒杀成功");
```

下单操作经过 3 个步骤的优化，吞吐能力显著提升：

3 数据一致性问题

3.1 CAP 定理



因此在分布式模式下，CAP 理论要求不做一个平衡，不能同时要求可用性，一致性；

3.2 数据一致性问题

1、扣减库存是扣减的数据库的库存

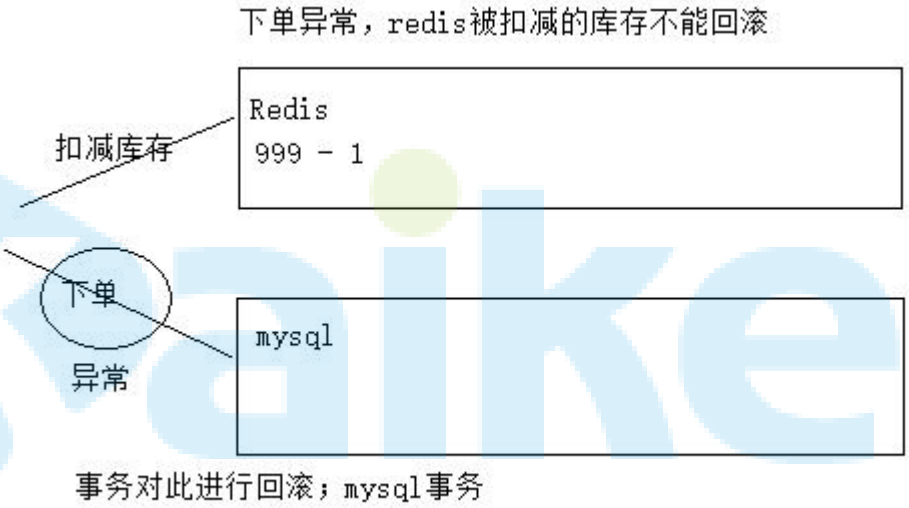
```
//2、利用redis原子性操作扣减库存，不需要上锁
boolean res = reduceStock(killId);
```

缓存数据 ，数据库库存数据一致性问题； 此时数据库的库存和缓存的库存不一致；

2、下单（操作的数据库）

```
//下单
TbSeckillOrder order = new TbSeckillOrder();
order.setSeckillId(killId);
order.setUserId(userId);
order.setCreateTime(new Date());
order.setStatus("0");
order.setMoney(seckillGoods.getCostPrice());
// 队列实现异步下单操作
Boolean produce = SeckillQueue.getMailQueue().produce(order);
```

下单（出现异常）失败了（下单操作是 MySQL 事务，可以回滚），扣减库存确成功了（扣减库存是 redis 操作，无法回滚）！！！



解决问题：缓存数据库库存一致性问题