Shop-flashSale

用户参数响应器

`org.springframework.web.method.support`包下有一个接口`HandlerMethodArgumentResolver`:

```
public interface HandlerMethodArgumentResolver {
    boolean supportsParameter(MethodParameter var1);

    @Nullable
    Object resolveArgument(MethodParameter var1, Message<?> var2) throws
Exception;
}
```

1. **建立用户参数解析器的类**

   定义一个自定义注解`@RequestUser`，用于标记需要注入用户信息的方法参数。

   ```
   @Target(ElementType.PARAMETER)
   @Retention(RetentionPolicy.RUNTIME)
   public @interface RequestUser {
   }
   ```

   创建一个实现`HandlerMethodArgumentResolver`接口的类`UserInfoMethodArgumentResolver`，用于解析带有`@RequestUser`注解的方法参数。

   ```
   public class UserInfoMethodArgumentResolver implements
   HandlerMethodArgumentResolver {
       @Autowired
       private StringRedisTemplate redisTemplate;

       @Override
       public boolean supportsParameter(MethodParameter parameter) {
           // 判断参数类型是否要处理
           return parameter.hasParameterAnnotation(RequestUser.class) &&
                   parameter.getParameterType() == UserInfo.class;
       }

       @Override
       public Object resolveArgument(MethodParameter parameter,
   ModelAndViewContainer modelAndViewContainer, NativeWebRequest webRequest,
   WebDataBinderFactory webDataBinderFactory) throws Exception {
           // 获取token
           String token = webRequest.getHeader("token");
           return
   JSON.parseObject(redisTemplate.opsForValue().get(CommonRedisKey.USER_TOKEN.ge
   tRealKey(token)), UserInfo.class);
       }
   }
   ```

- `supportsParameter`方法：判断参数是否带有`@RequestUser`注解，并且参数类型为`UserInfo`。
- `resolveArgument`方法：从请求头中获取 Token，然后从 Redis 中获取用户信息，并将其转换为`UserInfo`对象。

2. **在 Spring 配置类`WebConfig`中配置自定义参数解析器。**

```
    @Bean
    public UserInfoMethodArgumentResolver userInfoMethodArgumentResolver() {
        return new UserInfoMethodArgumentResolver();
    }
    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver>
resolvers) {
        resolvers.add(userInfoMethodArgumentResolver());
    }
```

3. **在控制器中使用自定义参数解析器**

```
@RestController
public class UserController {

    @GetMapping("/user/info")
    public String getUserInfo(@RequestUser UserInfo userInfo) {
        if (userInfo != null) {
            return "用户昵称: " + userInfo.getNickName();
        } else {
            return "未获取到用户信息";
        }
    }
}
```

秒杀

秒杀业务应当包含以下流程：

1. 检验用户是否登录
2. 基于秒杀商品id查询秒杀商品对象
3. 判断当前时间是否在秒杀时间范围内
4. 判断当前秒杀商品的库存是否足够
5. 判断用户是否在当前场次买过秒杀商品
6. 扣减秒杀商品库存
7. 创建秒杀商品订单信息

扣除库存

~~JVM内部锁~~

```
@CacheEvict(key = "'selectByIdAndTime:' + #time + ':' + #id")
@Override
public void decrStockCount(Long id, Integer time) {
    synchronized(this){
    // 先查询库存再扣库存
    int stockCount = seckillProductMapper.getStockCount(id);
    if (stockCount <= 0) {
        throw new BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
    }
    seckillProductMapper.decrStock(id);
    }
}
```

在应用开发中，使用 `synchronized` 关键字对扣除库存这类关键操作进行加锁控制，虽然是一种常见手段，但在现代复杂的软件架构下，其存在显著的局限性：

1. `synchronized` 所创建的锁，本质上是基于 JVM 进程内的同步机制。当在某个节点的程序中使用 `synchronized` 对扣除库存操作加锁时，该锁仅在当前 JVM 进程内生效。其他分布式节点上的程序完全感知不到这个锁的存在，它们可以自由地并发访问相同的业务逻辑代码。这就导致在分布式环境下，`synchronized` 无法有效阻止多个节点同时对库存进行扣除操作，引发超卖等数据不一致问题。
2. 在 Spring 框架中，Bean 默认采用单例模式进行管理当在一个单例 Bean 中使用 `synchronized(this)` 对扣除库存操作加锁时，由于 `this` 指向的是整个 Bean 实例，这意味着该 Bean 内的所有同步方法和代码块都会被这把锁所控制。假设该 Bean 除了扣除库存操作外，还包含数据库查询等其他业务方法，但由于 `synchronized(this)` 的使用，它们也被纳入了锁的范围，增加了锁的粒度。

分布式锁

实现思路

1. 对谁加锁：锁指定场次下的指定商品
2. 当多线程加锁时，只能有一个线程加锁成功：Redis的 `setnx` 命令
3. 锁存储的位置：利用redis的 `setnx` 命令存储在redis的String数据结构中
4. 线程获取不到锁时执行的策略：阻塞/自旋等待/抛出异常

```
@CacheEvict(key = "'selectByIdAndTime:' + #time + ':' + #id")
@Override
public void decrStockCount(Long id, Integer time) {
    String key = "seckill:product:stockcount:" + time + ":" + id;
    try {
        // 如果自旋五次，抛出异常
        int count = 0;
        boolean ret;
        do {
            // 对秒杀商品加锁
            // 设置过期时间，避免因分布式系统故障致使锁无法正常释放，造成死锁问题
```

```
            ret =
Boolean.TRUE.equals(redisTemplate.opsForValue().setIfAbsent(key,
String.valueOf(threadId), 10, TimeUnit.SECONDS));
            // 获取到锁继续往下执行
            if (ret) break;
            count++;
            if (count >= 5) throw new
BusinessException(SeckillCodeMsg.SERVER_BUSY);
            // 一段时间后再重新请求锁，防止CPU过于频繁
            Thread.sleep(20);
        } while (true);
        // 先查询库存再扣库存
        int stockCount = seckillProductMapper.getStockCount(id);
        if (stockCount <= 0) {
            throw new BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
        }
        seckillProductMapper.decrStock(id);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        // 释放锁
        redisTemplate.delete(key);
    }
}
```

`setIfAbsent(key, value, timeout, unit)` 方法会在 Redis 中执行类似 `SET key value NX PX timeout` 这样的原子命令，确保设置键值和过期时间是在一个原子操作中完成，现在支持过期时间的 `setIfAbsent(key, value, timeout, unit)` 是原子操作。

如果设置有效期不是原子操作，如以下这段代码

```
Boolean lock = Boolean.TRUE.equals(redisTemplate.opsForValue().setIfAbsent(key,
String.valueOf(threadId), 10, TimeUnit.SECONDS));
redisTemplate.expire(key,10,TimeUnit.SECONDS);
```

如果执行完setIfAbsent方法之后，expire方法之前程序宕机，死锁依旧会产生。

待解决的问题

现有 A、B、C 三个业务。A 执行逻辑耗时久，致其申请的锁 A 到期自动释放，B 随后申请到锁 B。在 B 执行时，A 执行完毕却误释放了锁 B，接着 C 申请到锁 C，B 执行完又误将锁 C 释放，造成锁释放错乱。

- **自己的锁自己释放**
    - 用雪花算法生成线程id标识当前线程，并作为redis锁的value

      ```
      // 雪花算法生成分布式唯一id
      threadId = IdGenerateUtil.get().nextId() + "";
      ret = Boolean.TRUE.equals(redisTemplate.opsForValue().setIfAbsent(key,
      String.valueOf(threadId), 10, TimeUnit.SECONDS));
      ```

    - 在释放锁时取出redis锁的value，判断当前value是否与threadId相同，相同时再释放锁

```
// 先获取value，判断当前value是否与threadId相同
String value = redisTemplate.opsForValue().get(key);
if (threadId.equals(value))
// 释放锁
redisTemplate.delete(key);
```

- **延长耗时长业务锁的有效期**
  - 问题：是否违背添加锁有效期的初衷？

    添加锁的有效期是避免因分布式系统故障（如崩溃、网络中断等）致使锁无法正常释放，造成死锁问题。续期机制是JVM完成的，当系统故障导致VM崩掉时，续期机制无法触发。当续期机制出发时，表明系统没有崩溃并且业务并没有执行完毕，所以并不违背添加锁有效期的初衷。

    但出现业务一直不执行完的情况，依旧可以导致死锁，亟待解决。

  - 看门狗解决思路

    一个线程负责监听业务是否执行完成，一个线程处理业务逻辑。当锁的有效期快到时如果业务还未完成，实现续期操作。如果业务完成，redis锁被释放，所以看门狗只需监听**redis锁是否释放**以此判断业务是否完成。看门狗在`finally`处释放。

```
    @CacheEvict(key = "'selectByIdAndTime:' + #time + ':' + #id")
    @Override
    public void decrStockCount(Long id, Integer time) {
        String key = "seckill:product:stockcount:" + time + ":" + id;
        String threadId = "";
        // 定义过期时间
        long timeout = 10;
        ScheduledFuture<?> future = null;
        try {
            // 如果自旋五次，抛出异常
            int count = 0;
            boolean ret;
            do {
                // 雪花算法生成分布式唯一id
                threadId = IdGenerateUtil.get().nextId() + "";
                // 对秒杀商品加锁
                ret =
Boolean.TRUE.equals(redisTemplate.opsForValue().setIfAbsent(key,
String.valueOf(threadId), timeout, TimeUnit.SECONDS));
                // 获取到锁继续往下执行
                if (ret) break;
                count++;
                if (count >= 5) throw new
BusinessException(SeckillCodeMsg.SERVER_BUSY);
                // 一段时间后再重新请求锁，防止CPU过于频繁
                Thread.sleep(20);
            } while (true);

            // 加锁成功 创建watchdog监听业务是否执行完成，实现续期操作
            long delayTime = (long) (timeout * 0.8);
            final String finalThreadId = threadId;
            future = scheduledExecutorService.scheduleAtFixedRate(() ->
{
                // 查询redis锁是否存在->业务执行未完成
```

```
                String value = redisTemplate.opsForValue().get(key);
                if (finalThreadId.equals(value)) {
                    // 将当前的redis锁续期
                    redisTemplate.expire(key, delayTime + 2,
TimeUnit.SECONDS);
                }
            }, delayTime, delayTime, TimeUnit.SECONDS);

            // 先查询库存再扣库存
            int stockCount = seckillProductMapper.getStockCount(id);
            if (stockCount <= 0) {
                throw new
BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
            }
            seckillProductMapper.decrStock(id);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            if (future != null) {
                future.cancel(true);
            }
            // 先获取value，判断当前value是否与threadId相同
            String value = redisTemplate.opsForValue().get(key);
            if (threadId.equals(value))
                // 释放锁
                redisTemplate.delete(key);
        }
    }
```

乐观锁

乐观锁一般会使用版本号机制或 CAS 算法实现，CAS 算法相对来说更多一些，这里需要格外注意。

版本号机制

一般是在数据表中加上一个数据版本号 `version` 字段，表示数据被修改的次数。当数据被修改时，`version` 值会加一。当线程 A 要更新数据值时，在读取数据的同时也会读取 `version` 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 `version` 值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子：假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ；而当前帐户余额字段（ `balance` ）为 $100 。

1. 操作员 A 此时将其读出（ `version`=1 ），并从其帐户余额中扣除 $50 （ $100-$50 ）。
2. 在操作员 A 操作的过程中，操作员 B 也读入此用户信息（ `version`=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。
3. 操作员 A 完成了修改工作，将数据版本号（ `version`=1 ），连同帐户扣除后余额（ `balance`=$50 ），提交至数据库更新，此时由于提交数据版本等于数据库记录当前版本，数据被更新，数据库记录 `version` 更新为 2 。
4. 操作员 B 完成了操作，也将版本号（ `version`=1 ）试图向数据库提交数据（ `balance`=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 1 ，数据库记录当前版本也为 2 ，不满足 “ 提交版本必须等于当前版本才能执行更新 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。

这样就避免了操作员 B 用基于 `version`=1 的旧数据修改的结果覆盖操作员 A 的操作结果的可能。

版本号实现机制关键点：1. 当作条件作为判断　2. 更新后值要改变 => `stock_count` 作为版本号

```
<update id="ol_decrStock">
    update t_seckill_product
    set stock_count = stock_count - 1
    where id = #{seckillId} and stock_count = stock_count
</update>
```

细致来讲，乐观锁的加入保证减库存操作的合理性，不会出现超卖问题。由于数据库的单个SQL语句的执行是原子性的，所以不管几个线程修改库存，只需保证库存数大于0即可避免超卖问题。

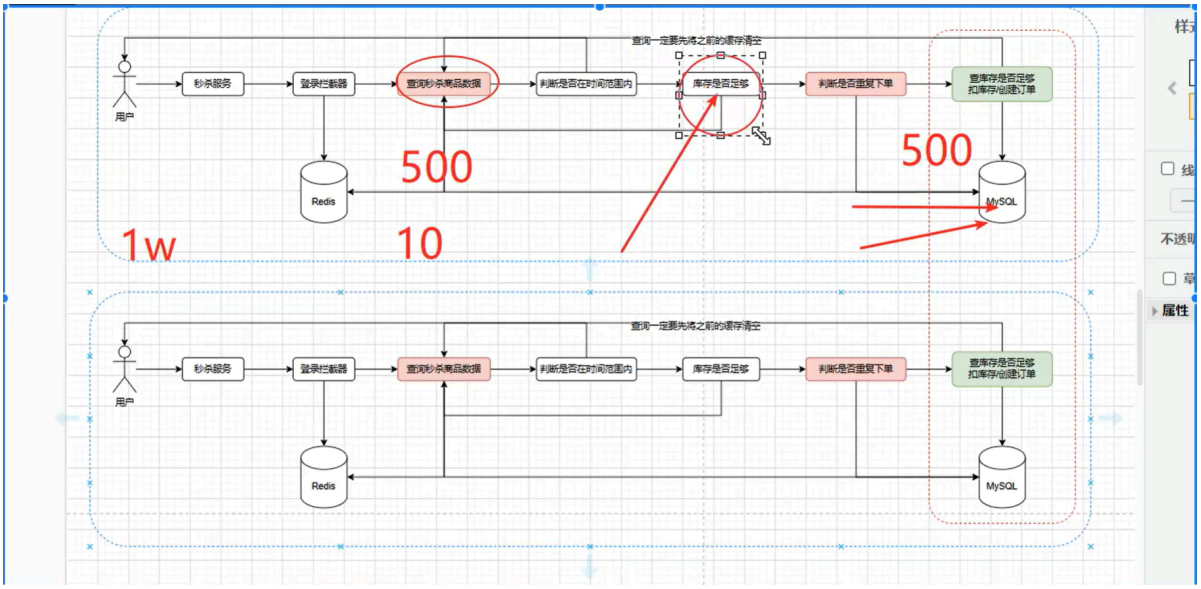
```
<update id="ol_decrStock">
    update t_seckill_product
    set stock_count = stock_count - 1
    where id = #{seckillId} and stock_count > 0
</update>
```

```
@CacheEvict(key = "'selectByIdAndTime'+#id")
@Override
public void decrStockCount(Long id) {
    int row = seckillProductMapper.ol_decrStock(id);
    if (row <= 0) throw new BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
}
```

库存预减

~~初始版本的判断库存是否充足：~~

```
// 判断库存是否充足
if (!(sp.getStockCount() > 0)) {
    throw new BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
}
```



当前这种库存判断方式仅能进行粗略估计，无法保证库存数量的准确。其根源在于，查询库存和扣除库存这两个操作并非原子性的。在分布式环境下，多个线程可能同时对库存进行访问，这就容易引发并发问题。具体来说，可能会出现这样的情况：线程 A 发起库存查询操作，在其获取到库存数量但尚未进行扣除操作时，线程 B 抢先完成了库存扣除。如此一来，线程 A 所依据的库存数据就已经过时。

为了避免在分布式环境下库存判断不准确的问题，我们可以将秒杀商品的库存提前缓存到 Redis 中，并确保库存的查询和扣除操作是原子性的。同时，为了防止因用户重复下单导致库存多扣而使商品无法正常售卖，需要调整逻辑顺序，**先检查用户是否已下单，再判断库存是否充足。**

```
// 判断库存是否充足
String hashKey = SeckillRedisKey.SECKILL_STOCK_COUNT_HASH.join(time + "");
// 返回剩余库存
Long remain = redisTemplate.opsForHash().increment(hashKey, seckillId + "", -1);
if (remain < 0) throw new BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
```

用户重复下单问题

~~初始版本的判断用户是否下单：~~

```
// 判断用户是否已经下过订单
OrderInfo orderInfo =
orderInfoService.selectByUserIdAndSeckillId(userInfo.getPhone(), seckillId,
time);
if (orderInfo != null) {
    throw new BusinessException(SeckillCodeMsg.REPEAT_SECKILL);
}
```

当前的问题时用户可以重复下单，因为查询用户订单和创建订单的操作不是原子操作。问题的解决思路同样是在redis中存储用户下单信息。`seckillOrderHash:phone:1`表示当前手机号的用户完成下单操作。

```
// 判断用户是否已经下过订单
String userOrderFlag = SeckillRedisKey.SECKILL_ORDER_HASH.join(seckillId + "");
Boolean absent = redisTemplate.opsForHash().putIfAbsent(userOrderFlag,
userInfo.getPhone() + "", 1);
if (!absent) throw new BusinessException(SeckillCodeMsg.REPEAT_SECKILL);
```

`putIfAbsent`方法的作用是：当指定的键不在哈希表中时，把该键值对插入到哈希表，并且返回`true`；若键已经存在于哈希表中，就不插入新的值，而是返回`false`。

此时还存在一个问题，用户下单后，redis已经存储用户下单标记，如果此时已经没有库存，但出现有人退款或者其他情况导致库存增加，但用户再次点击会报出请不要重复下单的提示导致秒杀失败。**当库存增加时，同时清除用户的下单标记，这样用户就可以再次参与秒杀。**

```
String orderNo = null;
try {
    // 判断库存是否充足
    String hashKey = SeckillRedisKey.SECKILL_STOCK_COUNT_HASH.join(time + "");
    // 返回剩余库存
    Long remain = redisTemplate.opsForHash().increment(hashKey, seckillId + "",
-1);
    if (remain < 0) throw new
BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
    // 创建订单，扣除库存，返回订单id
    orderNo = orderInfoService.doSeckill(userInfo, sp);
} catch (BusinessException e) {
    redisTemplate.opsForHash().delete(userOrderFlag, userInfo.getPhone() + "");
    return Result.error(e.getCodeMsg());
}
```

JVM缓存库存售完标记

进一步优化秒杀接口的性能，要求在当库存不够时，直接标记当前库存已经卖完。用线程安全的`ConcurrentHashMap`记录库存售完标记。

```
public static final Map<Long, Boolean> STOCK_OVER_FLOW_MAP = new
ConcurrentHashMap<>();
@RequireLogin
@RequestMapping("/doSeckill")
public Result<String> doSeckill(Integer time, Long seckillId, @RequestUser
UserInfo userInfo) {
    // 判断库存是否已经卖完
    Boolean flag = STOCK_OVER_FLOW_MAP.get(seckillId);
    if (flag != null && flag) {
        return Result.error(SeckillCodeMsg.SECKILL_STOCK_OVER);
    }

    // 基于秒杀id+场次查询秒杀商品对象
    // 判断当前时间是否在秒杀时间范围内
    // 判断用户是否已经下过订单
    ......
    try {
        // 判断库存是否充足
        // 创建订单，扣除库存，返回订单id
        ......
    } catch (BusinessException e) {
        // 当库存不够时，直接标记当前库存已经卖完
        STOCK_OVER_FLOW_MAP.put(seckillId, true);
        redisTemplate.opsForHash().delete(userOrderFlag, userInfo.getPhone() +
"");
        return Result.error(e.getCodeMsg());
    }
    return Result.success(orderNo);
    }
```

秒杀接口代码

```
@RequireLogin
@RequestMapping("/doSeckill")
public Result<String> doSeckill(Integer time, Long seckillId, @RequestUser
UserInfo userInfo) {
    // 判断库存是否已经卖完
    Boolean flag = STOCK_OVER_FLOW_MAP.get(seckillId);
    if (flag != null && flag) {
        return Result.error(SeckillCodeMsg.SECKILL_STOCK_OVER);
    }

    // 基于秒杀id+场次查询秒杀商品对象
    SeckillProductVo sp = seckillProductService.selectByIdAndTime(seckillId,
time);
    if (sp == null) {
        return Result.error(SeckillCodeMsg.OP_ERROR);
    }
    // 判断当前时间是否在秒杀时间范围内
    boolean isTime = betweenSeckillTime(sp);
    if (!isTime) {
        return Result.error(SeckillCodeMsg.OVER_TIME_ERROR);
    }


    // 判断用户是否已经下过订单
    String userOrderFlag = SeckillRedisKey.SECKILL_ORDER_HASH.join(seckillId +
"");
    Boolean absent = redisTemplate.opsForHash().putIfAbsent(userOrderFlag,
userInfo.getPhone() + "", "1");
    if (!absent) return Result.error(SeckillCodeMsg.REPEAT_SECKILL);

    String orderNo = null;
    try {
        // 判断库存是否充足
        String hashKey = SeckillRedisKey.SECKILL_STOCK_COUNT_HASH.join(time +
"");
        // 返回剩余库存
        Long remain = redisTemplate.opsForHash().increment(hashKey, seckillId +
"", -1);
        if (remain < 0) throw new
BusinessException(SeckillCodeMsg.SECKILL_STOCK_OVER);
        // 创建订单，扣除库存，返回订单id
        orderNo = orderInfoService.doSeckill(userInfo, sp);
    } catch (BusinessException e) {
        // 当库存不够时，直接标记当前库存已经卖完
        STOCK_OVER_FLOW_MAP.put(seckillId, true);
        redisTemplate.opsForHash().delete(userOrderFlag, userInfo.getPhone() +
"");
        return Result.error(e.getCodeMsg());
    }

    return Result.success(orderNo);
```

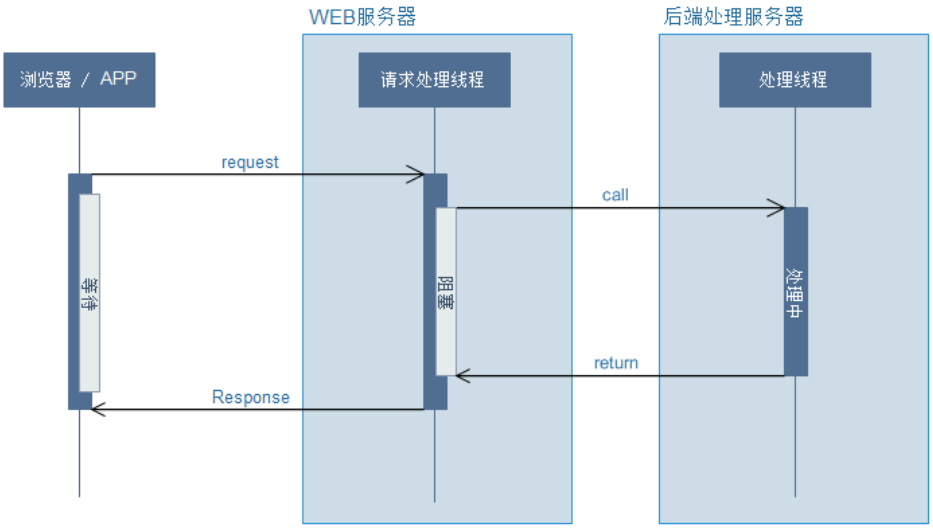
```
}
```

异步消息发送

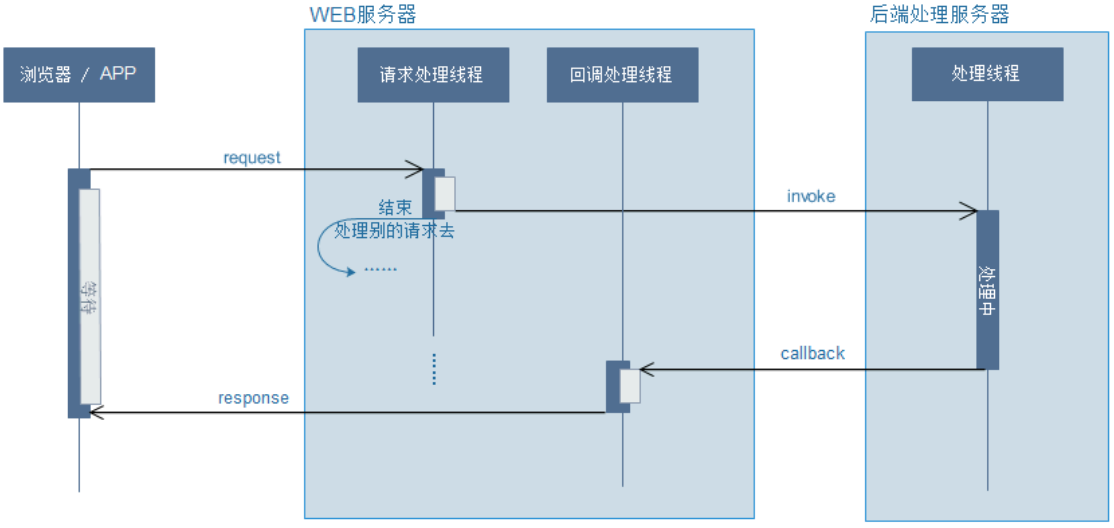
现在从数据库中查库存与扣库存，创建订单操作锁在一起，由于查询数据库是一个磁盘io操作，耗费时间，极大的占据主线程的时间。可以通过异步的方式查询数据库，解放主线程，提高系统的吞吐率。

异步请求

在`Servlet 3.0`之前，`Servlet`采用`Thread-Per-Request`的方式处理请求，即每一次`Http`请求都由某一个线程从头到尾负责处理。如果一个请求需要进行IO操作，比如访问数据库、调用第三方服务接口等，那么其所对应的线程将同步地等待IO操作完成， 而IO操作是非常慢的，所以此时的线程并不能及时地释放回线程池以供后续使用，在并发量越来越大的情况下，这将带来严重的性能问题。其请求流程大致为：



而在`Servlet3.0`发布后，提供了一个新特性：**异步处理请求**。可以**先释放**容器分配给请求的线程与相关资源，减轻系统负担，释放了容器所分配线程的请求，其**响应将被延后**，可以在耗时处理完成（例如长时间的运算）时再对客户端进行响应。其请求流程为：



在`Servlet 3.0`后，我们可以从`HttpServletRequest`对象中获得一个`AsyncContext`对象，该对象构成了异步处理的上下文，`Request`和`Response`对象都可从中获取。`AsyncContext`可以从当前线程传给另外的线程，并在新的线程中完成对请求的处理并返回结果给客户端，初始线程便可以还回给容器线程池以处理更多的请求。如此，通过将请求从一个线程传给另一个线程处理的过程便构成了`Servlet`

`3.0`中的异步处理。

Spring方式实现异步请求

> 在`Spring`中，有多种方式实现异步请求，比如`callable`、`DeferredResult`或者`WebAsyncTask`。每个的用法略有不同，可根据不同的业务场景选择不同的方式。

Callable

> 使用很简单，直接返回的参数包裹一层`callable`即可。

用法

```
@RequestMapping("/callable")
public Callable<String> callable() {
    log.info("外部线程：" + Thread.currentThread().getName());
    return new Callable<String>() {

        @Override
        public String call() throws Exception {
            log.info("内部线程：" + Thread.currentThread().getName());
            return "callable!";
        }
    };
}
```

异步处理下单请求

异步请求版本

配置线程池

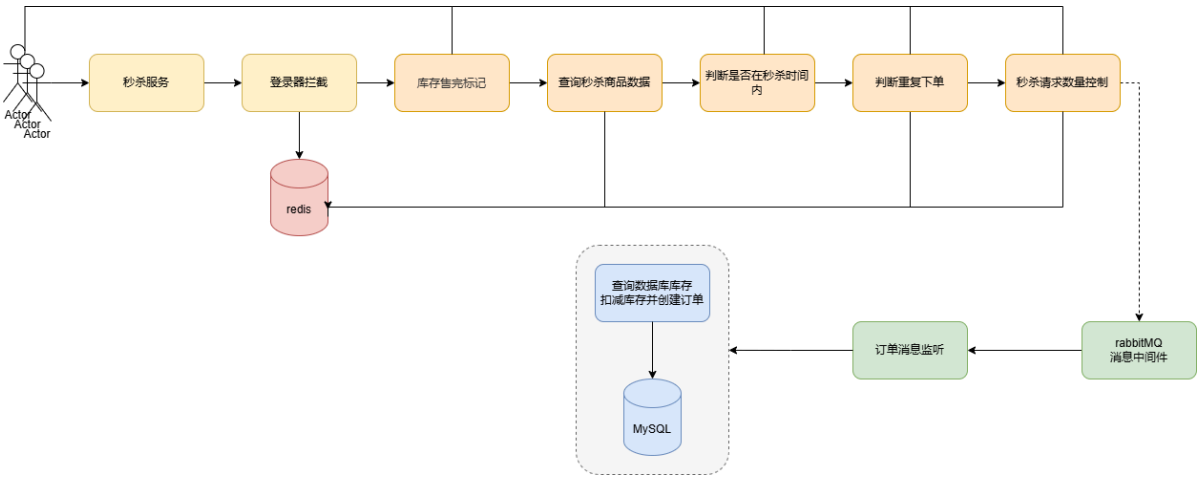
```
/**
 * 配置一个用于异步任务处理的线程池执行器
 * @return 配置好的 ThreadPoolTaskExecutor 实例
 */
@Bean
public ThreadPoolTaskExecutor webAsyncThreadPoolExecutor() {
    // 创建一个 ThreadPoolTaskExecutor 实例，用于管理线程池
    ThreadPoolTaskExecutor threadPoolTaskExecutor = new ThreadPoolTaskExecutor();
    // 设置线程池的核心线程数为 16，即线程池初始创建并保持活动的线程数量
    threadPoolTaskExecutor.setCorePoolSize(16);
    // 设置线程池的最大线程数为 200，当任务队列满时，线程池可以创建的最大线程数量
    threadPoolTaskExecutor.setMaxPoolSize(200);
    // 设置线程的空闲存活时间为 3 秒，当线程空闲超过该时间，会被销毁
    threadPoolTaskExecutor.setKeepAliveSeconds(3);
    // 返回配置好的线程池执行器实例
    return threadPoolTaskExecutor;
}

/**
 * 配置 Spring MVC 的异步支持
 * @param configurer 用于配置异步支持的配置器
 */
@Override
public void configureAsyncSupport(AsyncSupportConfigurer configurer) {
```

```
    // 设置异步任务的默认超时时间为 3000 毫秒，即异步任务如果在 3 秒内未完成，将被视为超时
    configurer.setDefaultTimeout(3000);
    // 将之前配置好的线程池执行器设置为异步任务的默认执行器
    configurer.setTaskExecutor(webAsyncThreadPoolExecutor());
}

/**
 * 创建一个用于处理 Callable 类型异步任务超时的拦截器
 * @return TimeoutCallableProcessingInterceptor 实例
 */
@Bean
public TimeoutCallableProcessingInterceptor
timeoutCallableProcessingInterceptor() {
    // 返回一个 TimeoutCallableProcessingInterceptor 实例，用于在异步任务超时时进行相应
处理
    return new TimeoutCallableProcessingInterceptor();
}
```

异步处理下单操作

```
@RequireLogin
@RequestMapping("/doSeckill")
public Callable<Result<String>> doSeckill(Integer time, Long seckillId,
@RequestUser UserInfo userInfo) {
    return () -> {
        // 判断库存是否已经卖完
        // 基于秒杀id+场次查询秒杀商品对象
        // 判断当前时间是否在秒杀时间范围内
        // 判断用户是否已经下过订单
        ......
        try {
            // 判断库存是否充足
            // 返回剩余库存
            // 异步创建订单，扣除库存，返回订单id
            CompletableFuture<String> future = CompletableFuture.supplyAsync(() -
> orderInfoService.doSeckill(userInfo, sp));
            return Result.success(future.get());

        } catch (BusinessException e) {
            // 当库存不够时，直接标记当前库存已经卖完
            ......
        } catch (Exception e) {
            e.printStackTrace();
        }
        return Result.defalutError();
    };
}
```

消息队列



在秒杀接口中发送mq异步消息

```
// 发送mq异步消息
OrderMessage orderMessage = new OrderMessage(time, seckillId, token,
userInfo.getPhone());
log.info("[秒杀下单] 发送订单创建消息: {}", JSON.toJSONString(orderMessage));
try {
    rabbitTemplate.convertAndSend(
            MQConstant.ORDER_PEDDING_TOPIC,
            "",
            orderMessage
    );
    log.info("[秒杀下单] 订单创建消息发送成功");
    return Result.success("订单创建中，请稍后...");
} catch (Exception e) {
    log.error("[秒杀下单] 订单创建消息发送失败: {}", e.getMessage(), e);
    return Result.error(SeckillCodeMsg.SECKILL_ERROR);
}
```

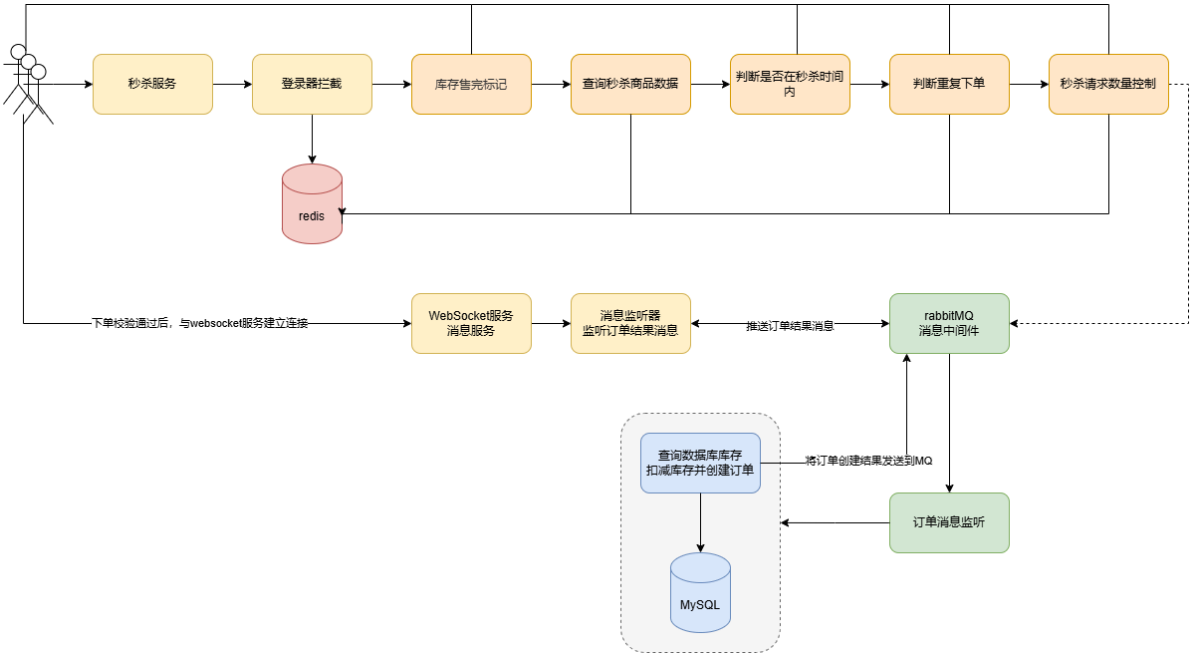
订单消息监听

```
@Component
@Slf4j
public class OrderPendingMessageListener {
    @Autowired
    private IOrderInfoService orderInfoService;
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = MQConstant.ORDER_PEDDING_TOPIC + "_QUEUE")
    public void onMessage(Message message, OrderMessage orderMessage, Channel
channel) throws IOException {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        log.info("[创建订单] 收到创建订单消息，准备开始创建订单： {}",
JSON.toJSONString(orderMessage));
        OrderMQResult result = new OrderMQResult();
        try {
            result.setTime(orderMessage.getTime());
            result.setSeckillId(orderMessage.getSeckillId());
```

```
            result.setToken(orderMessage.getToken());
            String orderNo =
orderInfoService.doSeckill(orderMessage.getUserPhone(),
orderMessage.getSeckillId(), orderMessage.getTime());
            result.setOrderNo(orderNo);
            result.setCode(Result.SUCCESS_CODE);
            result.setMsg("订单创建成功");
            // 手动确认消息
            channel.basicAck(deliveryTag, false);
        } catch (Exception e) {
            log.error("[创建订单] 处理订单消息失败: {}", e.getMessage(), e);
            // 订单创建失败
            SeckillCodeMsg codeMsg = SeckillCodeMsg.SECKILL_ERROR;
            result.setCode(codeMsg.getCode());
            result.setMsg(codeMsg.getMsg());
            // 消息处理失败
            channel.basicNack(deliveryTag, false, false);
        }
        // 发送订单创建结果消息
        rabbitTemplate.convertAndSend(
                MQConstant.ORDER_RESULT_TOPIC,
                "",
                result
        );
    }
}
```

服务端消息推送

普通的HTTP请求只能客户端向服务端发起请求，服务端响应请求，现在需要实现服务端主动的向指定的客户端发送消息。



- 短链接

  默认的HTTP协议是短链接，连接只保持一个响应/请求，结束后连接就会断开。

    - 客户端轮询： 客户端循环发送http请求或询问服务器是否有数据，实现简单，但会给服务端造成太大的压力。
- 长连接

  服务器/客户端双方都可以基于该链接对象，互相发送消息（全双工通信）。相对短链接来说效率更高，实现更优雅。

    - WebSocket：**web场景**下的tcp/ip通信，通信效率高，性能好。

要确保消息队列想消息监听器发送消息时WebSocket已经建立连接。

- 用户进入秒杀服务时便建立连接
    - 建立的很多不必要的连接（如用户秒杀失败就不需要连接），增加了开销
- 重试机制

```
// 尝试重新获取会话
int count = 0;
do {
    count++;
    log.info("[订单结果] 第{}次查询session对象获取失败，准备下一次获取...", count);
    // 等待前客户端可能的连接
    Thread.sleep(500);

    // 再次尝试获取session
    session = WebsocketServer.SESSION_MAP.get(token);
    if (session != null && session.isOpen()) {
        try {
            log.info("[订单结果] 重试后找到会话，发送消息: token={}", token);
            session.getBasicRemote().sendText(JSON.toJSONString(result));
            log.info("[订单结果] 重试后消息发送成功");
            channel.basicAck(deliveryTag, false);
            return;
        } catch (Exception e) {
            log.error("[订单结果] 重试后发送消息失败: {}", e.getMessage(), e);
            WebsocketServer.SESSION_MAP.remove(token);
        }
    }
} while (count < 10); // 增加重试次数以提高成功率
```

推送订单结果消息： 在`OrderPendingMessageListener`中实现

```
// 发送订单创建结果消息
rabbitTemplate.convertAndSend(
        MQConstant.ORDER_RESULT_TOPIC,
        "",
        result
);
```

回滚

下单失败回滚

要注意的是，项目部署在分布式环境，意味着每一个秒杀接口都会有一个**库存售罄标识**，在删除本地库存售完标识时应当考虑到其他节点的**库存售罄标识**也应该一起删除。

但用户下单失败，应当包含以下三个操作：

1. **回补redis库存数量控制**
    - 由于下单失败的订单并没有修改数据库的库存信息，只修改了Redis中的缓存值
    - 回补操作直接从数据库查询当前最新的库存数量(selectStockById)
    - 然后将该数量直接写入Redis缓存，覆盖之前减少后的值
    - 这样可以保证Redis中的库存数据与数据库保持一致，无需计算增量

```
// 回补redis库存
Long stock = seckillProductService.selectStockById(msg.getSeckillId());
String hashKey = SeckillRedisKey.SECKILL_STOCK_COUNT_HASH.join(msg.getTime()
+ "");
redisTemplate.opsForHash().put(hashKey, msg.getSeckillId() + "", stock + "");
```

2. **删除用户下单标识**

    为了防止用户重复下单，系统在用户发起秒杀请求时会在Redis中记录一个标识。当订单创建失败时，需要删除这个标识，以允许用户重新下单。

```
// 删除用户下单标识
String userOrderFlag =
SeckillRedisKey.SECKILL_ORDER_HASH.join(msg.getSeckillId() + "");
redisTemplate.opsForHash().delete(userOrderFlag, msg.getUserPhone() + "");
```

3. **删除本地售罄标识**

    秒杀系统通常会在本地内存中维护一个商品售罄标识Map，用于快速过滤库存已空的商品请求。当需要回滚时，也需要清除这个标识。代码实现分为两部分：

    - 发送广播消息：

```
// 使用广播交换机进行广播，清除所有节点的售罄标识
log.info("[订单回滚] 发送清除售罄标识广播消息，商品ID: {}", msg.getSeckillId());
rabbitTemplate.convertAndSend(
        MQConstant.CLEAR_STOCK_OVER_BROADCAST_EXCHANGE,
        "",
        msg.getSeckillId()
);
```

    - 监听广播消息并处理：

```
@RabbitListener(queues = MQConstant.CLEAR_STOCK_OVER_BROADCAST_QUEUE)
public void onMessage(Message message, Long seckillId, Channel channel)
throws IOException {
    long deliveryTag = message.getMessageProperties().getDeliveryTag();
```

```
        try {
            log.info("[库存回滚] 接收清除售罄标识广播消息，商品ID: {}", seckillId);
            // 清除本地售罄标识缓存
            OrderInfoController.deleteKey(seckillId);
            // 手动确认消息
            channel.basicAck(deliveryTag, false);
        } catch (Exception e) {
            log.error("[库存回滚] 处理清除售罄标识消息失败: {}", e.getMessage(), e);
            // 消息处理失败，拒绝消息并不重新入队
            channel.basicNack(deliveryTag, false, false);
        }
    }
```

超时未支付

在订单创建成功后，`OrderPendingMessageListener`会发送一个延迟消息

```
// 当下单成功后 发送延迟消息 检查订单支付状态 超时未支付就直接取消订单
orderMessage.setOrderNo(orderNo);
log.info("[创建订单] 开始发送订单支付超时检查延迟消息，订单号: {}，延迟时间: {}秒",
orderNo, MQConstant.ORDER_PAY_TIMEOUT_DELAY_SECONDS);
try {
    // 发送到延迟队列，通过TTL和死信队列实现延迟
    rabbitTemplate.convertAndSend(
            MQConstant.ORDER_PAY_TIMEOUT_TOPIC + "_DELAY",
            "",
            orderMessage
    );
    log.info("[创建订单] 订单支付超时检查延迟消息发送成功，订单号: {}", orderNo);
} catch (Exception e) {
    log.error("[创建订单] 订单支付超时检查延迟消息发送失败: {}", e.getMessage(), e);
}
```

监听超时订单消息并处理

```
@Component
@Slf4j
public class OrderPayTimeoutMsgListener {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private IOrderInfoService orderInfoService;

    @RabbitListener(queues = MQConstant.ORDER_PAY_TIMEOUT_TOPIC + "_QUEUE")
    public void onMessage(Message message, OrderMessage msg, Channel channel)
throws IOException {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        log.info("[订单超时未支付检查] 收到超时检查订单状态消息: {}",
JSON.toJSONString(msg));
        try {
            orderInfoService.checkPayTimeout(msg);
            // 手动确认消息
            channel.basicAck(deliveryTag, false);
```

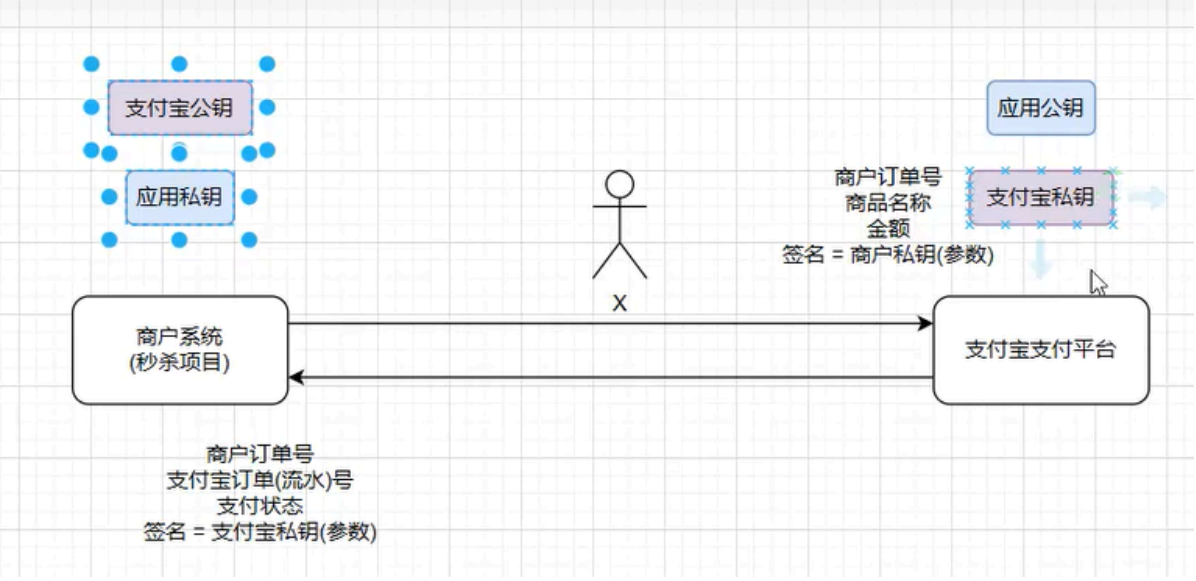
```
            log.info("[订单超时未支付检查] 处理成功，订单号: {}", msg.getOrderNo());
        } catch (Exception e) {
            log.error("[订单超时未支付检查] 处理失败: {}", e.getMessage(), e);
            // 消息处理失败，拒绝消息并不重新入队
            channel.basicNack(deliveryTag, false, false);
        }
    }
}
```

超时订单的处理逻辑

```
@Transactional(rollbackFor = Exception.class)
@Override
public void checkPayTimeout(OrderMessage msg) {
    // 查询订单对象
    // 判断状态 未支付-> 取消订单
    int row = orderInfoMapper.changePayStatus(msg.getOrderNo(),
OrderInfo.STATUS_CANCEL, OrderInfo.PAY_TYPE_ONLINE);
    if (row > 0) {
        // MySQL秒杀商品库存数量+1
        seckillProductService.incryStockCount(msg.getSeckillId());
    }
    // 失败订单信息回滚
    this.failedRollback(msg);
}
```

支付退款服务

RSA算法-签名机制实现数据防篡改



1. **数据加密**：公钥加密，私钥解密
2. **签名机制（防篡改）**：私钥加密，公钥解密

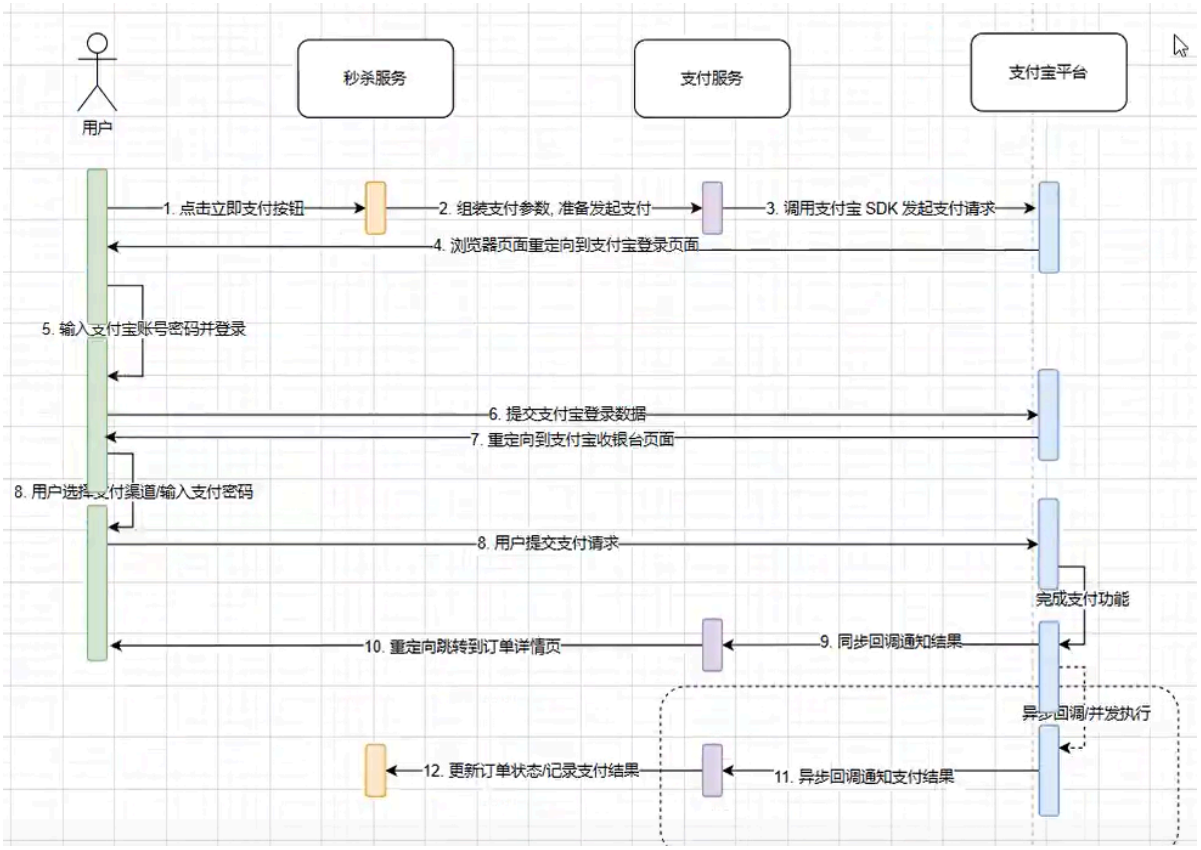
幂等性问题

> 幂等性问题指的是一个操作无论被执行一次还是多次，其产生的结果都是相同的，不会因为多次执行而导致系统状态出现不一致或产生额外的副作用。

重复多次调用下单成功接口会导致数据混乱，这就需要考虑**幂等性问题**。在分布式场景下的重试后需要做DML语句的操作，都有可能产生幂等性问题。如**消息队列中的ack消息因为网络故障导致传输失败，订单创建逻辑执行成功，但因为没有收到确认消息导致订单创建消息重新投递，造成了重复消费问题。**

解决幂等性问题的关键在于使用唯一编号，具体做法有：一是在数据库中设置唯一索引；二是基于锁或乐观锁，借助状态值来处理 。

支付宝支付退款服务



支付服务实现

1. `seckill-server` 模块提供支付选择接口，作为用户支付入口
2. 通过 `Feign` 客户端实现服务间通信，将支付请求转发至 `pay-server` 模块
3. `pay-server` 模块集成支付宝沙箱 API，处理具体的支付逻辑
4. 支付结果通过回调接口通知到系统，并更新订单状态

支付宝支付接口

```
@PostMapping("/prepay")
public Result<String> prepay(@RequestBody PayVo payVo) {
    // 支付宝SDK API向支付宝发起支付请求
    AlipayTradePagePayRequest request = new AlipayTradePagePayRequest();
    // 必传参数
    JSONObject bizContent = new JSONObject();
    // 商户订单号
    bizContent.put("out_trade_no", payVo.getOutTradeNo());
```

```
    // 支付金额
    bizContent.put("total_amount", payVo.getTotalAmount());
    // 订单标题
    bizContent.put("subject", payVo.getSubject());
    bizContent.put("body", payVo.getBody());
    // 电脑网站支付场景固定传值
    bizContent.put("product_code", "FAST_INSTANT_TRADE_PAY");

    request.setBizContent(bizContent.toString());
    // 异步接收地址 仅支持http/https
    request.setNotifyUrl(payVo.getNotifyUrl() != null ? payVo.getNotifyUrl() :
alipayProperties.getNotifyUrl());
    // 同步跳转地址 仅支持http/https
    request.setReturnUrl(payVo.getReturnUrl() != null ? payVo.getReturnUrl() :
alipayProperties.getReturnUrl());

    // 记录请求参数，用于调试
    log.info("支付宝支付请求参数: bizContent={}, notifyUrl={}, returnUrl={}",
            bizContent.toString(), request.getNotifyUrl(),
request.getReturnUrl());
    log.info("支付宝配置信息: appId={}, signType={}, charset={}, gatewayUrl={}",
            alipayProperties.getAppId(), alipayProperties.getSignType(),
            alipayProperties.getCharset(), alipayProperties.getGatewayUrl());

    try {
        AlipayTradePagePayResponse response = alipayClient.pageExecute(request);
        log.info("支付宝响应完整信息: {}", JSON.toJSONString(response));

        if (response.isSuccess()) {
            log.info("支付宝预支付请求成功: {}", payVo.getOutTradeNo());
            return Result.success(response.getBody());
        } else {
            log.error("支付宝预支付请求失败: {}, 错误码: {}, 错误信息: {}",
                    payVo.getOutTradeNo(), response.getCode(),
response.getMsg());
            return Result.defalutError();
        }
    } catch (AlipayApiException e) {
        log.error("支付宝预支付请求异常", e);
        return Result.defalutError();
    }
```

支付服务实现

```
@Override
public String onlinePay(String orderNo) {
    // 基于订单号查询订单对象
    OrderInfo orderInfo = orderInfoMapper.find(orderNo);
    // 判断订单状态
    if(!OrderInfo.STATUS_ARREARAGE.equals(orderInfo.getStatus())){
        throw new BusinessException(PayCodeMsg.PAY_ERROR);
    }
    // 封装支付参数
    PayVo vo = new PayVo();
    vo.setBody("秒杀: " + orderInfo.getProductName());
```

```
    vo.setSubject(orderInfo.getProductName());
    vo.setOutTradeNo(orderNo);
    vo.setTotalAmount(orderInfo.getSeckillPrice().toString());
    // 远程调用支付服务发起支付
    Result<String> result = paymentFeignApi.prepay(vo);
    if (result.hasError()) {
        throw new BusinessException(PayCodeMsg.PAY_ERROR);
    }
    return result.getData();
}
```

支付宝异步回调支付结果

> 支付异步回调是在用户完成支付操作后，支付宝服务器会通过异步方式把支付结果发送给商户服务器。

- 支付服务

  定义一个接口，接受支付宝异步通知结果，更新订单状态

  1. 定义一个`POST`请求接口，接收支付宝传递的参数
  2. 验证参数签名是否正确
  3. 远程调用秒杀服务，更新订单状态，记录支付日志
  4. 返回成功

```
@PostMapping("/notify_url")
public String notifyUrl(HttpServletRequest request) {
    // 接收支付宝参数
    Map<String, String> params = new HashMap<>();
    Map<String, String[]> requestParams = request.getParameterMap();
    for (Iterator iter = requestParams.keySet().iterator(); iter.hasNext(); )
{
        String name = (String) iter.next();
        String[] values = requestParams.get(name);
        String valueStr = "";
        for (int i = 0; i < values.length; i++) {
            valueStr = (i == values.length - 1)? valueStr + values[i]
                    : valueStr + values[i] + ",";
        }
        //乱码解决。这段代码在出现乱码时使用。如果mysign和sign不相等也可以使用这段代码转
化
        //valueStr = new String(valueStr.getBytes("ISO-8859-1"), "gbk");
        params.put(name, valueStr);
    }
    try {
        log.info("[异步回调] 收到交易已完成消息:{}", params);
        boolean verify_result = AlipaySignature.rsaCheckV1(params,
alipayProperties.getAlipayPublicKey(), alipayProperties.getCharset(),
"RSA2");
        if (verify_result) {
            // 商户订单号
            String out_trade_no = request.getParameter("out_trade_no");
            // 支付宝交易号
            String trade_no = request.getParameter("trade_no");
            // 交易状态
```

```
            String trade_status = request.getParameter("trade_status");
            // 支付金额
            String totalAmount = request.getParameter("total_amount");

            if (trade_status.equals("TRADE_FINISHED")) {
                log.info("[异步回调] {}订单已完成", out_trade_no);
            } else if (trade_status.equals("TRADE_SUCCESS")) {
                log.info("[异步回调]{}订单已支付", out_trade_no);
                // 远程调用秒杀服务更新订单状态
                PayResult payResult = new PayResult(out_trade_no, trade_no,
totalAmount);
                Result<?> result =
seckillFeignService.updateOrderPaySuccess(payResult);
                if (result.hasError()) {
                    throw new BusinessException(PayCodeMsg.PAY_ERROR);
                }
            }
            return "success";
        }
    } catch (AlipayApiException e) {
        e.printStackTrace();
    }
    return "fail";
}
```

- 秒杀服务

  定义一个接口，接受支付服务传递的参数，更新订单状态，记录支付流水

  1. 定义一个`POST`请求接口，接收支付服务传递的参数
  2. 基于订单编号查询订单信息，判断订单信息是否有误
  3. 更新订单状态（保证幂等性）
  4. 创建支付流水对象并保存
  5. 返回成功

```
@Override
public void alipaySuccess(PayResult result) {
    // 查询订单信息
    OrderInfo orderInfo = orderInfoMapper.find(result.getOutTradeNo());
    if(orderInfo==null) throw new BusinessException(PayCodeMsg.ORDER_ERROR);
    // 判断订单信息是否正确
    if (!orderInfo.getSeckillPrice().toString().equals(result.getTradeNo()))
{
        throw new BusinessException(PayCodeMsg.ORDER_ERROR);
    }
    // 更新订单状态
    int row = orderInfoMapper.changePayStatus(result.getOutTradeNo(),
OrderInfo.STATUS_ACCOUNT_PAID, OrderInfo.PAY_TYPE_ONLINE);
    if (row <= 0) throw new BusinessException(PayCodeMsg.PAY_ERROR);
    // 记录支付流水
    PayLog paylog = new PayLog();
    paylog.setOrderNo(result.getOutTradeNo());
    paylog.setPayType(PayLog.PAY_TYPE_ONLINE);
    paylog.setTotalAmount(result.getTotalAmount());
    paylog.setTradeNo(result.getTradeNo());
```

```
        paylog.setNotifyTime(System.currentTimeMillis() + "");
        payLogMapper.insert(paylog);
    }
```

支付宝同步回调支付结果

> 支付同步回调主要用于在支付完成后，支付平台将支付结果实时返回给商户网站或应用程序。

```
@GetMapping("/return_url")
public String returnUrl(HttpServletRequest request) {
    Map<String, String> params = new HashMap<>();
    Map requestParams = request.getParameterMap();
    for (Iterator iter = requestParams.keySet().iterator(); iter.hasNext(); ) {
        String name = (String) iter.next();
        String[] values = (String[]) requestParams.get(name);
        String valueStr = "";
        for (int i = 0; i < values.length; i++) {
            valueStr = (i == values.length - 1)? valueStr + values[i]
                    : valueStr + values[i] + ",";
        }
        //乱码解决，这段代码在出现乱码时使用。如果mysign和sign不相等也可以使用这段代码转化
        params.put(name, valueStr);
    }

    try {
        log.info("[同步回调] 收到交易消息：{}", params);
        boolean verify_result = AlipaySignature.rsaCheckV1(params,
alipayProperties.getAlipayPublicKey(), alipayProperties.getCharset(), "RSA2");
        if (!verify_result) throw new BusinessException(PayCodeMsg.PAY_ERROR);
        // 商户订单号
        String outTradeNo = request.getParameter("out_trade_no");
        return outTradeNo;
    } catch (AlipayApiException e) {
        e.printStackTrace();
    }
    return "签名验证失败";
}
```

退款服务实现

1. `seckill-server` 模块提供退款接口，作为用户退款入口
2. 通过 `Feign` 客户端实现服务间通信，将退款请求转发至 `pay-server` 模块
3. `pay-server` 模块集成支付宝沙箱 API，处理具体的退款逻辑
4. 退款结果通过返回值通知到系统，并更新订单状态

退款服务退款接口

```
@PostMapping("/refund")
public Result<Boolean> refund(@RequestBody RefundVo refund) {
    log.info("[支付宝退款] 准备退款请求: {}", JSON.toJSONString(refund));
    AlipayTradeRefundRequest alipay_request = new AlipayTradeRefundRequest();

    AlipayTradeRefundModel model = new AlipayTradeRefundModel();
    model.setOutTradeNo(refund.getOutTradeNo());
```

```
    model.setRefundAmount(refund.getRefundAmount());
    model.setRefundReason(refund.getRefundReason());
    alipay_request.setBizModel(model);

    try {
        AlipayTradeRefundResponse response =
alipayClient.execute(alipay_request);
        log.info("[支付宝退款] 收到支付宝退款响应结果: {}",
JSON.toJSONString(response));
        if (response.isSuccess()) {
            // 判断是否支付成功
            if ("Y".equalsIgnoreCase(response.getFundChange())) {
                return Result.success(true);
            }
            // 如果 fund_change = N 表示退款没有成功/需要等待结果确认
            return Result.success(false);
        } else {
            // 处理响应不成功的情况
            log.error("[支付宝退款] 退款请求失败: {}", response.getSubMsg());
            return Result.error(new CodeMsg(500001, response.getSubMsg()));
        }
    } catch (AlipayApiException e) {
        e.printStackTrace();
        return Result.error(new CodeMsg(500001, e.getMessage()));
    }
}
```

退款服务实现

退款服务的具体实现在秒杀服务中完成，通过feign接口调用pay-server中的退款接口辅以实现。

```
@Override
@Transactional(rollbackFor = Exception.class)
public void refund(String orderNo) {
    // 查询订单对象
    OrderInfo orderInfo = orderInfoMapper.find(orderNo);
    // 判断订单状态是否为已支付
    if (!OrderInfo.STATUS_ACCOUNT_PAID.equals(orderInfo.getStatus())) {
        throw new BusinessException(PayCodeMsg.UNPAID);
    }
    // 判断订单支付类型
    Result<Boolean> result = null;
    RefundVo refundVo = new RefundVo(orderNo,
orderInfo.getSeckillPrice().toString(), "不想要了");
    if (orderInfo.getPayType() == OrderInfo.PAY_TYPE_ONLINE) {
        //支付宝退款
        result = paymentFeignApi.refund(refundVo);
    } else {
        //积分退款
    }
    // 判断是否退款成功
    if (result == null || result.hasError() || !result.getData()) {
        throw new BusinessException(PayCodeMsg.REFUND_ERROR);
    }
    // 退款成功 更新订单状态为已退款
```

```
    int row = orderInfoMapper.changeRefundStatus(orderNo,
OrderInfo.STATUS_REFUND);
    if(row<=0) throw new BusinessException(PayCodeMsg.REFUND_ERROR);
    // 库存回补
    seckillProductService.incryStockCount(orderInfo.getSeckillId());
    this.rollbackRedis(orderInfo.getSeckillId(), orderInfo.getSeckillTime());
    // 使用广播交换机进行广播，清除所有节点的售罄标识
    log.info("[订单回滚] 发送清除售罄标识广播消息，商品ID: {}",
orderInfo.getSeckillId());
    rabbitTemplate.convertAndSend(
            MQConstant.CLEAR_STOCK_OVER_BROADCAST_EXCHANGE,
            "",
            orderInfo.getSeckillId()
    );
    // 创建退款日志对象
    RefundLog refundLog = new RefundLog();
    refundLog.setRefundReason("用户申请退款" + orderInfo.getProductName());
    refundLog.setRefundTime(new Date());
    refundLog.setRefundType(orderInfo.getPayType());
    refundLog.setRefundAmount(orderInfo.getSeckillPrice().toString());
    refundLog.setOutTradeNo(orderNo);
    refundLogMapper.insert(refundLog);
}
```

积分支付退款服务

积分支付操作由 `seckill-server` 发起，通过 `Feign` 客户端接口调用 `integral-server 中的服务来实现。

幂等性问题

在分布式系统环境下，网络波动、服务重启等因素可能导致 Feign 请求重传，这就会带来重复扣减积分的风险，因此需要实现严格的幂等性控制。

数据库级别的幂等性设计：将 t_account_log 表中的订单号和操作类型字段 (out_trade_no, type) 设置为复合唯一索引。这种设计使得每个订单只能进行一次特定类型的积分操作：

- type=0 表示积分减少操作，对应支付行为
- type=1 表示积分增加操作，对应退款行为

通过这种方式，系统可以保证同一订单的积分支付或退款操作不会因为重复请求而被多次执行。当重复请求到达时，数据库会因违反唯一约束而拒绝插入重复记录，从而防止积分被多次扣减或返还。

```
ALTER TABLE `shop_intergral`.`t_account_log` ADD UNIQUE INDEX
`uk_out_trade_no_type` (`out_trade_no`, `type`);
```

积分支付服务实现

在`seckill-server`中的接口为

```
@Transactional(rollbackFor = Exception.class)
@Override
public String integralPay(String orderNo, Long phone) {
    // 基于订单号查询订单对象
    OrderInfo orderInfo = orderInfoMapper.find(orderNo);
```

```
    // 判断订单状态
    if (!OrderInfo.STATUS_ARREARAGE.equals(orderInfo.getStatus())) {
        throw new BusinessException(PayCodeMsg.PAY_ERROR);
    }
    // 判断当前用户是否是创建但钱订单的用户 =》 确保自己订单用自己的积分消费
    if (!orderInfo.getUserId().equals(phone)) throw new
BusinessException(PayCodeMsg.NOT_THIS_USER);
    // 封装支付参数
    OperateIntergralVo operateIntergralVo = new OperateIntergralVo();
    operateIntergralVo.setInfo("积分秒杀:" + orderInfo.getProductName());
    operateIntergralVo.setValue(orderInfo.getIntegral());
    operateIntergralVo.setUserId(phone);
    operateIntergralVo.setOutTradeVo(orderNo);
    // 远程调用支付服务发起支付
    Result<String> result = integralFeignApi.prepay(operateIntergralVo);
    String tradeNo = result.getData();
    // 更新订单支付状态和记录支付流水日志
    int row = orderInfoMapper.changePayStatus(orderNo,
OrderInfo.STATUS_ACCOUNT_PAID, OrderInfo.PAY_TYPE_INTEGRAL);
    if(row<0) throw new BusinessException(PayCodeMsg.PAY_FAIL);

    PayLog payLog = new PayLog();
    payLog.setPayType(PayLog.PAY_TYPE_INTEGRAL);
    payLog.setTotalAmount(operateIntergralVo.getValue() + "");
    payLog.setOutTradeNo(orderNo);
    payLog.setTradeNo(tradeNo);
    payLog.setNotifyTime(System.currentTimeMillis() + "");
    payLogMapper.insert(payLog);
    return "积分支付成功!\n" + result.toString();
}
```

其中通过`integralFeignApi`调用`integral-server`的`prepay`接口。

```
// 远程调用支付服务发起支付
Result<String> result = integralFeignApi.prepay(operateIntergralVo);
String tradeNo = result.getData();
```

该接口实现了**积分支付**的功能，积分支付服务代码如下：

```
@Transactional(rollbackFor = Exception.class)
@Override
public String doPay(OperateIntergralVo vo) {
    // 1. 获取当前用户的积分账户，判断账户余额是否足够
    // 2. 扣除积分，更新时间
    int row = usableIntegralMapper.decrIntegral(vo.getUserId(), vo.getValue());
    if (row <= 0) {
        throw new BusinessException(IntergralCodeMsg.INTERGRAL_NOT_ENOUGH);
    }
    // 3. 记录账户变动日志
    AccountLog log = new AccountLog();
    log.setAmount(vo.getValue());
    log.setInfo(vo.getInfo());
    log.setGmtTime(new Date());
    log.setOrderNo(vo.getOutTradeVo());
```

```
    // 生成交易流水号
    log.setTradeNo(IdGenerateUtil.get().nextId() + "");
    accountLogMapper.insert(log);
    return log.getTradeNo();
}
```

积分退款

在`seckill-server`中延续支付宝退款接口

```
if (orderInfo.getPayType() == OrderInfo.PAY_TYPE_ONLINE) {
    //支付宝退款
    result = paymentFeignApi.refund(refundVo);
} else {
    //积分退款
    result = integralFeignApi.refund(refundVo);
}
```

`integral-server`中对积分退款服务的具体实现

```
@Transactional(rollbackFor = Exception.class)
@Override
public boolean refund(RefundVo vo) {
    // 1.基于订单编号查询支付流水
    AccountLog decrLog =
accountLogMapper.selectByOutTradeNoAndType(vo.getOutTradeNo(),
AccountLog.TYPE_DECR);
    if (decrLog == null) {
        throw new BusinessException(IntergralCodeMsg.INTERGRAL_UNPAID);
    }
    // 2.通过支付流水中的支付金额，退回到用户积分账户
    BigDecimal refundAmount = new BigDecimal(vo.getRefundAmount());
    if (refundAmount.compareTo(new BigDecimal(decrLog.getAmount()))>0) {
        // 退款金额大于支付金额
        throw new BusinessException(IntergralCodeMsg.AMOUNT_ERROR);
    }
    // 注意数值转换
    usableIntegralMapper.addIntegral(decrLog.getUserId(),
refundAmount.longValue());
    // 3. 记录新的退款流水记录
    AccountLog incrLog = new AccountLog();
    incrLog.setInfo(vo.getRefundReason());
    // 幂等性问题 => type与OutTradeNo保证了唯一索引，用户再次退款会报SQL错误
    incrLog.setType(AccountLog.TYPE_INCR);
    incrLog.setOutTradeNo(vo.getOutTradeNo());
    incrLog.setAmount(refundAmount.longValue());
    incrLog.setTradeNo(IdGenerateUtil.get().nextId() + "");
    incrLog.setUserId(decrLog.getUserId());
    incrLog.setGmtTime(new Date());
    accountLogMapper.insert(incrLog);
    return true;
}
```

积分支付退款分布式事务分析

上述设计中，`integralPay`方法虽然使用了`@Transactional(rollbackFor = Exception.class)`注解进行事务管理，但确实存在明显的分布式事务问题。这个问题的核心在于，本地事务无法跨越微服务边界来保证整体操作的原子性。当`integralPay`方法通过**Feign**客户端调用积分服务`integralFeignApi.prepay()`时，实际上已经跨越了服务边界。Spring的本地事务管理器只能管理当前数据源的操作，无法控制远程服务的事务。

正常流程中的三个操作：**远程调用积分服务，扣减用户积分；更新订单状态为已支付；插入支付日志记录**。这三个操作应该是一个原子操作，但目前的设计无法保证这一点。

不一致状态的风险：

- 如果积分服务成功扣减积分后，订单服务在更新订单状态时发生异常，本地事务会回滚订单状态变更
- 但积分服务中已执行的扣减操作不会回滚，因为它在另一个服务的事务中
- 这就导致用户积分已扣除，但订单状态仍为未支付，造成数据不一致

与支付过程类似，积分退款也需要作为一个原子单元执行，但由于跨越了服务边界，本地事务无法保证一致性。

分布式事务

> 分布式事务是指跨越多个独立服务或资源的事务操作，需要保证这些操作要么全部成功，要么全部失败回滚，以维护数据的一致性。

分布式事务的场景

- 单体系统访问多个数据库

  > 一个服务需要调用多个数据库实例完成数据的增删改查

- 多个微服务访问同一个数据库

  > 多个服务需要调用一个数据库实例完成数据的增删改查

- 多个微服务访问多个数据库

  > 多个服务需要调用一个数据库实例完成数据的增删改查

分布式系统理论

CAP理论

> 在分布式系统中，最多只能同时满足一致性、可用性和分区容错性中的两项。由于网络分区在分布式系统中不可避免，因此实际上我们只能在一致性(C)和可用性(A)之间做选择：
>
> - CP系统：在网络分区时保证一致性，牺牲可用性
> - AP系统：在网络分区时保证可用性，牺牲一致性

- **C:一致性**

  一个分布式系统在任意情况下，所有实例都返回最新一致的数据。主从节点数据的一致性。

- **A:可用性**

一个分布式系统在任意情况下，所有实例都能随时响应成功给客户端，哪怕是旧数据。集群，一个节点挂掉后有备选节点。

- **P:分区容错性**

  一个分布式系统在任意情况下，至少有一个网络分区是可用的。存在多个网络分区时，能够容忍有部分网络分区不可用。

  由于每个系统都基本要满足分区容错性，一旦在多个不同的网络分区时，A/C之间就产生了互斥性。真正的系统设计考虑的特性只有CP/AP两种类型的系统。

  > 假设一个分布式系统有两个节点A和B，存储着相同的数据X=1。现在发生了网络分区，这两个节点之间无法通信：
  >
  > - 如果优先保证一致性(C)：
  >   - 当客户端向节点A发送写请求，将X改为2
  >   - 由于网络分区，节点A无法将新值同步到节点B
  >   - 为了保证一致性，系统必须阻止节点B上的读操作，或者拒绝节点A上的写操作
  >   - 这意味着某些操作请求会被拒绝，无法保证可用性(A)
  > - 如果优先保证可用性(A)：
  >   - 当客户端向节点A发送写请求，将X改为2
  >   - 节点A接受写入，节点B继续服务读请求，返回旧值X=1
  >   - 系统保持可用，但节点A和B返回的数据不一致
  >   - 这意味着系统牺牲了一致性(C)

BASE理论

> BASE理论是对CAP定理的延伸和补充，为了解决强一致性和高可用性不可兼得的矛盾，提出的一种适用于分布式系统的设计理念。BASE由三个概念的首字母组成：Basically Available, Soft state, Eventually consistent。

- BA 基本可用：一个分布式系统中，只要满足基本可用即可，无需所有节点都可用。
- S 中间状态：一个分布式系统中，无需满足强一致性，允许存在中间状态
- E 最终一致性：允许存在中间状态，但最终（时效性）要达到一致。

解决方案

全局事务

全局事务基于DTP模型实现，DTP是由X/Open组织提出的一种分布式事务模型——X/Open Distributed Transaction Processing Reference Model，它规定了实现分布式事务，需要三种角色：

• AP: Application 应用系统 (微服务)

• TM: Transaction Manager 事务管理器 (全局事务管理)

• RM: Resource Manager 资源管理器 (数据库)

整个事务分成两个阶段 (2PC)：

• 阶段一: 表决阶段，所有参与者都将本事务执行预提交，并将能否成功的信息反馈给协调者。

• 阶段二: 执行阶段，协调者根据所有参与者的反馈，通知所有参与者，步调一致地执行提交或者回滚

TCC事务

TCC 即为 Try Confirm Cancel，它属于补偿型分布式事务。TCC 实现分布式事务一共有三个步骤：

- Try：尝试执行业务

  这个过程并未执行业务，只是完成所有业务的一致性检查，并预留好执行所需的全部资源

- Confirm：确认执行业务

  确认执行业务操作，不做任何业务检查，只使用Try阶段预留的业务资源。通常情况下，采用TCC则认为 Confirm 阶段是不会出错的，即：只要Try成功，Confirm一定成功。若Confirm阶段真的出错了，需引入重试机制或人工处理。

- Cancel：取消待执行的业务

  取消Try阶段预留的业务资源。通常情况下，采用TCC则认为Cancel阶段也是一定成功的。若Cancel阶段真的出错了，需引入重试机制或人工处理。

可靠消息服务

pass

Seata

Seata-AT模式实现2PC与传统2PC的差别:

- 架构层次方面，传统2PC方案的 RM 实际上是在数据库层，RM本质上就是数据库自身，通过XA协议实现，而 Seata 的 RM 是以 jar 包的形式作为中间件层部署在应用程序这一侧的。
- 两阶段提交方面，传统2PC无论第二阶段的决议是commit还是rollback，事务性资源的锁都要保持到Phase2完成才释放。而 Seata 的做法是在 Phase1 就将本地事务提交，这样就可以省去Phase2持锁的时间，整体提高效率。

AT模式代码实现

- 分布式事务发起方贴`@GlobalTransactional`注解

  积分支付事务的发起方：`seckill-server`中的`integralPay`方法

  ```
  @GlobalTransactional
  @Transactional(rollbackFor = Exception.class)
  @Override
  public String integralPay(String orderNo, Long phone){...}
  ```

- 分布式事务贴上`@Transactional`即可

  积分支付涉及到的分布式事务

  ```
  /*integral-service*/
  @Transactional(rollbackFor = Exception.class)
  @Override
  public String doPay(OperateIntergralVo vo) {...}
  ```