接口幂等和防抖还在傻傻分不清楚。。。

==========

最近想重温下接口幂等的相关知识，正所谓温故而知新，太久不CV的东西要是哪天CV起来都生疏了，哈哈哈

先从字面意思来温习下吧，幂等的官方概念来源于数学上幂等概念的衍生，如幂等函数，即使用相同的参数重复执行，可以得到相同的结果的函数，翻译成IT行业专业术语就是一个接口使用相同的入参，无论执行多少次，最后得到的结果且保存的数据和执行一次是完全一样的，所以，基于这个概念，分析我们的CRUD，首先，查询，它可以说是幂等的，但是如果更精细的说，它也可能不是幂等的，基于数据库数据不变的情况下，查询接口是幂等的，如果是变化的话那可能上一秒你查出来的数据，下一秒它就被人修改了也不是不可能，所以，基于这一点它不符合幂等概念

接下来是删除接口，它和查询一样，也是天然幂等的，但是如果你的接口提供范围删除，那么就破坏了幂等性原则，理论上这个删除接口就不应该存在，那如果是产品经理非要那也是可以存在滴，技术永远都是为业务服务的嘛

修改接口也是同样的道理，理论上都必须是幂等的，如果不是，那就要考虑接口幂等性了，比如你的修改积分接口里写修改积分，每次都使用i++这种操作,那么它就破坏了幂等原则，有一个好方法就是基于用户唯一性标识把积分变动通过一张表记录下来，最后统计这张表的积分数值，这里也就涉及到新增接口的知识点，其实到这里，我们会发现，所有的接口理论上都可以是幂等的，但是总是这个那个的原因导致不幂等，所以，总结起来就是，如果你的系统需求需要接口幂等，那么就实现它，现在让我们进入正题吧

刚开始温习幂等知识的时候，我百度了很多别人写的文章，发现另一个概念，叫防抖，防止用户重复点击的意思，有意思的是有些文章竟然认为防抖就是幂等，他们解决接口幂等的思路是每次调用需要实现幂等接口时，前端都需要调用后端的颁布唯一token的接口，后端颁布token后保存在缓存中，然后前端带着这个token的请求去请求我们的幂等接口，验证携带的token来实现接口幂等，按照这个思路每次请求的token都不一样，如何保证幂等中相同参数的条件呢，这显然和幂等南辕北辙了，这显然就是接口防抖或者接口加锁的思路嘛

还有一种是可以实现接口幂等性的思路，这里也可以分享一下，和上面的思路差不多，也是每次请求幂等接口的时候，先调用颁发唯一token的接口，唯一不同的是它颁发的token是基于入参生成的哈希值，后面的业务逻辑就是后端基于这个哈希值去校验，如果缓存中已经存在了，说明这个入参已经请求过了，那么直接拒绝请求并返回成功，这样，就从表面上实现了接口幂等性，因为执行

100次我只执行一次，剩余的99次我都拒绝，并直接返回成功，这样，我的接口就从表面上幂等了，但是这个方案有一个很大的问题就是每次调用都需要浪费一部分资源在请求颁发token上，这对需要大量的幂等接口的系统来说就是一个累赘，所以，接下来，我们基于这个思路实现一个不需要二次调用的实现接口幂等的方法。

我的思路是这样的，业务上有些接口是实现防抖功能，有些是实现幂等功能，其实这两个功能在业务上确实是相差不大，所以，我的思路是定义一个注解，包含防抖和幂等的功能，首先基于幂等如果要是把所有入参都哈希化作为唯一标识的话有点费劲，可以基于业务上的一些唯一标识来做，如用户id或者code，还需要一个开关，用于决定是否保存这个唯一标识，还要一个时间，保存多久，还有保存时间的单位，最后，还有一个返回提醒，就是拒绝之后的友好提示，基于这些差不多了，如果你的接口功能只需要实现防抖，那么你可以设置时间段内过期，这样就实现了防抖，如果你的接口没有唯一标识，那么可以基于路由来做防抖，这个不要忘了设置过期时间，不然你的接口就永远是拒绝了，好了，思路有了，接下来就是实操了，话不多数，上代码

```
@Retention(RetentionPolicy.RUNTIME)
//注解用于方法
@Target({ElementType.TYPE, ElementType.METHOD})
//注解包含在JavaDoc中
@Documented
public @interface Idempotent {

    /**
     * 幂等操作的唯一标识，使用spring el表达式 用#来引用方法参数
     *
     * @return Spring-EL expression
     */
    String key() default "";

    /**
     * 有效期 默认：1 有效期要大于程序执行时间，否则请求还是可能会进来
     *
     * @return expireTime
     */
    int expireTime() default 100;

    /**
     * 时间单位 默认：s
     *
     * @return TimeUnit
     */
    TimeUnit timeUnit() default TimeUnit.SECONDS;
```

```
    /**
     * 提示信息，可自定义
     *
     * @return String
     */
    String info() default "重复请求，请稍后重试";

    /**
     * 是否在业务完成后删除key  true:删除  false:不删除
     *
     * @return boolean
     */
    boolean delKey() default false;
```

基本和我们上面的思路一样，唯一key，有效期，有效期时间单位，提示信息，是否删除，注解有了，那么我们就要基于注解写我们的逻辑了，这里我们需要用到aop，引用注解应该都知道吧，这里我们直接上代码了

```
@Aspect
@Slf4j
public class IdempotentAspect {
    @Resource
    private RedisUtil redisUtil;

    private static final SpelExpressionParser PARSER = new SpelExpressionParser();

    private static final LocalVariableTableParameterNameDiscoverer DISCOVERER = new LocalVariableTableParameterNameDiscoverer();

    /**
     * 线程私有map
     */
    private static final ThreadLocal<Map<String, Object>> THREAD_CACHE = ThreadLocal.withInitial(HashMap::new);

    private static final String KEY = "key";

    private static final String DEL_KEY = "delKey";

    // 以自定义 @Idempotent 注解为切点
```

```
    @Pointcut("@annotation(com.liuhui.demo_core.spring.Idempotent)")
    public void idempotent() {
    }

    @Before("idempotent()")
    public void before(JoinPoint joinPoint) throws Throwable {
        //获取到当前请求的属性，进而得到HttpServletRequest对象，以便后续获取请求URL和参数信息。
        ServletRequestAttributes requestAttributes = (ServletRequestAttributes) RequestContextHolder
                .getRequestAttributes();
        HttpServletRequest request = requestAttributes.getRequest();
        //从JoinPoint中获取方法签名，并确认该方法是否被@Idempotent注解标记。如果是，则继续执行幂等性检查逻辑；如果不，则直接返回，不进行幂等处理。。
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        Method method = signature.getMethod();
        if (!method.isAnnotationPresent(Idempotent.class)) {
            return;
        }
        Idempotent idempotent = method.getAnnotation(Idempotent.class);
        String key;
        // 若没有配置 幂等 标识编号，则使用 url + 参数列表作为区分;如果提供了key规则，则利用keyResolver根据提供的规则和切点信息生成键
        if (!StringUtils.hasLength(idempotent.key())) {
            String url = request.getRequestURL().toString();
            String argString = Arrays.asList(joinPoint.getArgs()).toString();
            key = url + argString;
        } else {
            // 使用jstl 规则区分
            key = resolver(idempotent, joinPoint);
        }
        //从注解中读取并设置幂等操作的过期时间、描述信息、时间单位以及是否删除键的标志。
        long expireTime = idempotent.expireTime();
        String info = idempotent.info();
        TimeUnit timeUnit = idempotent.timeUnit();
        boolean delKey = idempotent.delKey();
        String value = LocalDateTime.now().toString().replace("T", " ");
        Object valueResult = redisUtil.get(key);
        synchronized (this) {
            if (null == valueResult) {
                redisUtil.set(key, value, expireTime, timeUnit);
            } else {
                throw new IdempotentException(info);
            }
        }
```

```
        Map<String, Object> map = THREAD_CACHE.get();
        map.put(KEY, key);
        map.put(DEL_KEY, delKey);
    }


    /**
     * 从注解的方法的参数中解析出用于幂等性处理的键值（key）
     *
     * @param idempotent
     * @param point
     * @return
     */
    private String resolver(Idempotent idempotent, JoinPoint point) {
        //获取被拦截方法的所有参数
        Object[] arguments = point.getArgs();
        //从字节码的局部变量表中解析出参数名称
        String[] params =
DISCOVERER.getParameterNames(getMethod(point));
        //SpEL表达式执行的上下文环境，用于存放变量
        StandardEvaluationContext context = new
StandardEvaluationContext();
        //遍历方法参数名和对应的参数值，将它们一一绑定到
StandardEvaluationContext中。
        //这样SpEL表达式就可以引用这些参数值
        if (params != null && params.length > 0) {
            for (int len = 0; len < params.length; len++) {
                context.setVariable(params[len], arguments[len]);
            }
        }
        //使用SpelExpressionParser来解析Idempotent注解中的key属性，将
其作为SpEL表达式字符串
        Expression expression =
PARSER.parseExpression(idempotent.key());
        //转换结果为String类型返回
        return expression.getValue(context, String.class);
    }

    /**
     * 根据切点解析方法信息
     *
     * @param joinPoint 切点信息
     * @return Method    原信息
     */
    private Method getMethod(JoinPoint joinPoint) {
        //将joinPoint.getSignature()转换为MethodSignature
        //Signature是AOP中表示连接点签名的接口，而MethodSignature是它
的具体实现，专门用于表示方法的签名。
```

```
        MethodSignature signature = (MethodSignature)
joinPoint.getSignature();
        //获取到方法的声明。这将返回代理对象所持有的方法声明。
        Method method = signature.getMethod();

        //判断获取到的方法是否属于一个接口
        //因为在Java中，当通过Spring    AOP或其它代理方式调用接口的方法
时，实际被执行的对象是一个代理对象，直接获取到的方法可能来自于接口声
明而不是实现类。
        if (method.getDeclaringClass().isInterface()) {
            try {
                //通过反射获取目标对象的实际类
 (joinPoint.getTarget().getClass()) 中同名且参数类型相同的方法
                //这样做是因为代理类可能对方法进行了增强，直接调用实现类的
方法可以确保获取到最准确的实现细节
                method =
joinPoint.getTarget().getClass().getDeclaredMethod(joinPoint.getSignatur
e().getName(),
                        method.getParameterTypes());
            } catch (SecurityException | NoSuchMethodException e) {
                throw new RuntimeException(e);
            }
        }
        return method;
    }


    @After("idempotent()")
    public void after() throws Throwable {
        Map<String, Object> map = THREAD_CACHE.get();
        if (CollectionUtils.isEmpty(map)) {
            return;
        }

        String key = map.get(KEY).toString();
        boolean delKey = (boolean) map.get(DEL_KEY);
        if (delKey) {
            redisUtil.delete(key);
            log.info("[idempotent]:has removed key={}", key);
        }
        //无论是否移除了键，最后都会清空当前线程局部变量
THREAD_CACHE中的数据，避免内存泄漏
        THREAD_CACHE.remove();
    }
 }

```

上面redisUtil是基于RedisTemplate封装的工具类，可以直接替换哈，这里我们定义一个切入点，也就是我们定义的注解，然后在调用接口之前获取到接口的入参以及注解的参数，获取到这些之后，判断是否有唯一标识，没有就用路由，保存到reids当中，然后设置过期时间，最后需要把删除的标识放到线程私有变量THREAD_CACHE中在接口处理完之后判断是否需要删除redis当中保存的key，这里，我们的逻辑就写完了，接下来是使用了，使用这个就很简单，直接在你需要实现防抖和幂等的接口上打上我们的注解

```
/**
 * 测试接口添加幂等校验
 *
 * @return
 */
@PostMapping("/redis")
@Idempotent(key = "#user.id", expireTime = 10, delKey = true, info = "重复请求，请稍后再试")
public Result<?> getRedis(@RequestBody User user) throws InterruptedException {
    return Result.success(true);
}
```

这里key的定义方式我们使用了SpEL表达式，如果不指定这个表达式的话就会使用路由作为key了

到这里，接口幂等和防抖功能就顺利完成了，以后，别再防抖和幂等傻傻分不清楚了哈哈哈

最后，还是要送上一位名人曾说的一句话：手上没有剑和有剑不用是两回事！

原文链接: https://juejin.cn/post/7380274613185970195