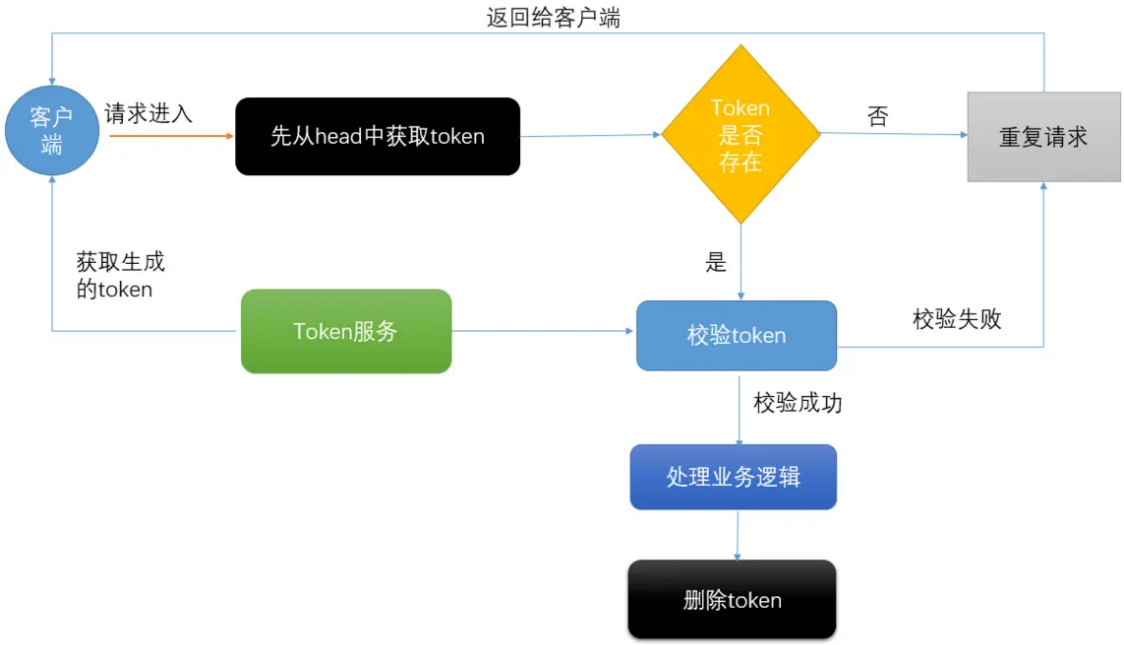
前言:

在实际的开发项目中,一个对外暴露的接口往往会面临很多次请求，我们来解释一下幂等的概念：**任意多次执行所产生的影响均与一次执行的影响相同。**按照这个含义，最终的含义就是 对数据库的影响只能是一次性的，不能重复处理。如何保证其幂等性，通常有以下手段：

1. 数据库建立唯一性索引，可以保证最终插入数据库的只有一条数据
2. token机制，每次接口请求前先获取一个token，然后再下次请求的时候在请求的header体中加上这个token，后台进行验证，如果验证通过删除token，下次请求再次判断token
3. 悲观锁或者乐观锁，悲观锁可以保证每次for update的时候其他sql无法update数据(在数据库引擎是innodb的时候,select的条件必须是唯一索引,防止锁全表)
4. 先查询后判断，首先通过查询数据库是否存在数据，如果存在证明已经请求过了，直接拒绝该请求，如果没有存在，就证明是第一次进来，直接放行。

redis实现自动幂等的原理图：



一：搭建redis的服务Api

1:首先是搭建redis服务器。

2:引入springboot中到的redis的stater，或者Spring封装的jedis也可以，后面主要用到的api就是它的set方法和exists方法,这里我们使用springboot的封装好的redisTemplate

```
/**
 * redis工具类
 */
@Component
public class RedisService {

    @Autowired
    private RedisTemplate redisTemplate;

    /**
```

```
     * 写入缓存
     * @param key
     * @param value
     * @return
     */
    public boolean set(final String key, Object value) {
        boolean result = false;
        try {
            ValueOperations<Serializable, Object> operations =
redisTemplate.opsForValue();
            operations.set(key, value);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }


    /**
     * 写入缓存设置时效时间
     * @param key
     * @param value
     * @return
     */
    public boolean setEx(final String key, Object value, Long expireTime) {
        boolean result = false;
        try {
            ValueOperations<Serializable, Object> operations =
redisTemplate.opsForValue();
            operations.set(key, value);
            redisTemplate.expire(key, expireTime, TimeUnit.SECONDS);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }


    /**
     * 判断缓存中是否有对应的value
     * @param key
     * @return
     */
    public boolean exists(final String key) {
        return redisTemplate.hasKey(key);
    }

    /**
     * 读取缓存
     * @param key
     * @return
     */
    public Object get(final String key) {
        Object result = null;
        ValueOperations<Serializable, Object> operations =
redisTemplate.opsForValue();
```

```
            result = operations.get(key);
            return result;
        }

        /**
         * 删除对应的value
         * @param key
         */
        public boolean remove(final String key) {
            if (exists(key)) {
                Boolean delete = redisTemplate.delete(key);
                return delete;
            }
            return false;

        }

    }
```

二：自定义注解AutoIdempotent

自定义一个注解，定义此注解的主要目的是把它添加在需要实现幂等的方法上，凡是某个方法注解了它，都会实现自动幂等。后台利用反射如果扫描到这个注解，就会处理这个方法实现自动幂等，使用元注解ElementType.METHOD表示它只能放在方法上，etentionPolicy.RUNTIME表示它在运行时

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface AutoIdempotent {

}
```

三：token创建和检验

1:token服务接口

我们新建一个接口，创建token服务，里面主要是两个方法，一个用来创建token，一个用来验证token。创建token主要产生的是一个字符串，检验token的话主要是传达request对象，为什么要传request对象呢？主要作用就是获取header里面的token,然后检验，通过抛出的Exception来获取具体的报错信息返回给前端

```
public interface TokenService {

    /**
     * 创建token
     * @return
     */
    public  String createToken();

    /**
     * 检验token
```

```
     * @param request
     * @return
     */
    public boolean checkToken(HttpServletRequest request) throws Exception;

}
```

2:token的服务实现类

token引用了redis服务，创建token采用随机算法工具类生成随机uuid字符串,然后放入到redis中(为了防止数据的冗余保留,这里设置过期时间为10000秒,具体可视业务而定)，如果放入成功，最后返回这个token值。checkToken方法就是从header中获取token到值(如果header中拿不到，就从paramter中获取)，如若不存在,直接抛出异常。这个异常信息可以被拦截器捕捉到，然后返回给前端。

```
@Service
public class TokenServiceImpl implements TokenService {

    @Autowired
    private RedisService redisService;


    /**
     * 创建token
     *
     * @return
     */
    @Override
    public String createToken() {
        String str = RandomUtil.randomUUID();
        StrBuilder token = new StrBuilder();
        try {
            token.append(Constant.Redis.TOKEN_PREFIX).append(str);
            redisService.setEx(token.toString(), token.toString(),10000L);
            boolean notEmpty = StrUtil.isNotEmpty(token.toString());
            if (notEmpty) {
                return token.toString();
            }
        }catch (Exception ex){
            ex.printStackTrace();
        }
        return null;
    }


    /**
     * 检验token
     *
     * @param request
     * @return
     */
    @Override
    public boolean checkToken(HttpServletRequest request) throws Exception {

        String token = request.getHeader(Constant.TOKEN_NAME);
        if (StrUtil.isBlank(token)) {// header中不存在token
```

```
            token = request.getParameter(Constant.TOKEN_NAME);
            if (StrUtil.isBlank(token)) {// parameter中也不存在token
                throw new
ServiceException(Constant.ResponseCode.ILLEGAL_ARGUMENT, 100);
            }
        }

        if (!redisService.exists(token)) {
            throw new
ServiceException(Constant.ResponseCode.REPETITIVE_OPERATION, 200);
        }

        boolean remove = redisService.remove(token);
        if (!remove) {
            throw new
ServiceException(Constant.ResponseCode.REPETITIVE_OPERATION, 200);
        }
        return true;
    }
}
```

四：拦截器的配置

1:web配置类，实现WebMvcConfigurerAdapter，主要作用就是添加autoIdempotentInterceptor到配置类中，这样我们到拦截器才能生效，注意使用@Configuration注解，这样在容器启动是时候就可以添加进入context中

```
@Configuration
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Resource
   private AutoIdempotentInterceptor autoIdempotentInterceptor;

    /**
     * 添加拦截器
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(autoIdempotentInterceptor);
        super.addInterceptors(registry);
    }
}
```

2:拦截处理器：主要的功能是拦截扫描到AutoIdempotent到注解到方法,然后调用tokenService的checkToken()方法校验token是否正确，如果捕捉到异常就将异常信息渲染成json返回给前端

```
/**
 * 拦截器
 */
@Component
```

```
public class AutoIdempotentInterceptor implements HandlerInterceptor {

    @Autowired
    private TokenService tokenService;

    /**
     * 预处理
     *
     * @param request
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

        if (!(handler instanceof HandlerMethod)) {
            return true;
        }
        HandlerMethod handlerMethod = (HandlerMethod) handler;
        Method method = handlerMethod.getMethod();
        //被ApiIdempotment标记的扫描
        AutoIdempotent methodAnnotation =
method.getAnnotation(AutoIdempotent.class);
        if (methodAnnotation != null) {
            try {
                return tokenService.checkToken(request);// 幂等性校验, 校验通过则放
行, 校验失败则抛出异常, 并通过统一异常处理返回友好提示
            }catch (Exception ex){
                ResultVo failedResult = ResultVo.getFailedResult(101,
ex.getMessage());
                writeReturnJson(response, JSONUtil.toJsonStr(failedResult));
                throw ex;
            }
        }
        //必须返回true,否则会被拦截一切请求
        return true;
    }


    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {

    }

    /**
     * 返回的json值
     * @param response
     * @param json
```

```
     * @throws Exception
     */
    private void writeReturnJson(HttpServletResponse response, String json)
throws Exception{
        PrintWriter writer = null;
        response.setCharacterEncoding("UTF-8");
        response.setContentType("text/html; charset=utf-8");
        try {
            writer = response.getWriter();
            writer.print(json);

        } catch (IOException e) {
        } finally {
            if (writer != null)
                writer.close();
        }
    }

}
```

五：测试用例

1:模拟业务请求类

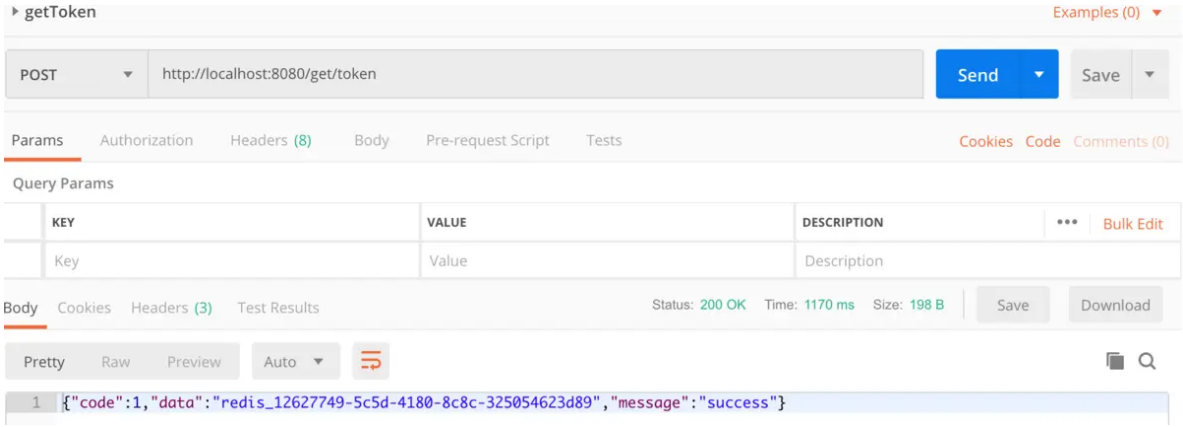
首先我们需要通过/get/token路径通过getToken()方法去获取具体的token，然后我们调用testIdempotence方法，这个方法上面注解了@AutoIdempotent，拦截器会拦截所有的请求，当判断到处理的方法上面有该注解的时候，就会调用TokenService中的checkToken()方法，如果捕获到异常会将异常抛出调用者，下面我们来模拟请求一下：

```
@RestController
public class BusinessController {


    @Resource
    private TokenService tokenService;

    @Resource
    private TestService testService;


    @PostMapping("/get/token")
    public String  getToken(){
        String token = tokenService.createToken();
        if (StrUtil.isNotEmpty(token)) {
            ResultVo resultVo = new ResultVo();
            resultVo.setCode(Constant.code_success);
            resultVo.setMessage(Constant.SUCCESS);
            resultVo.setData(token);
            return JSONUtil.toJsonStr(resultVo);
        }
        return StrUtil.EMPTY;
    }
```

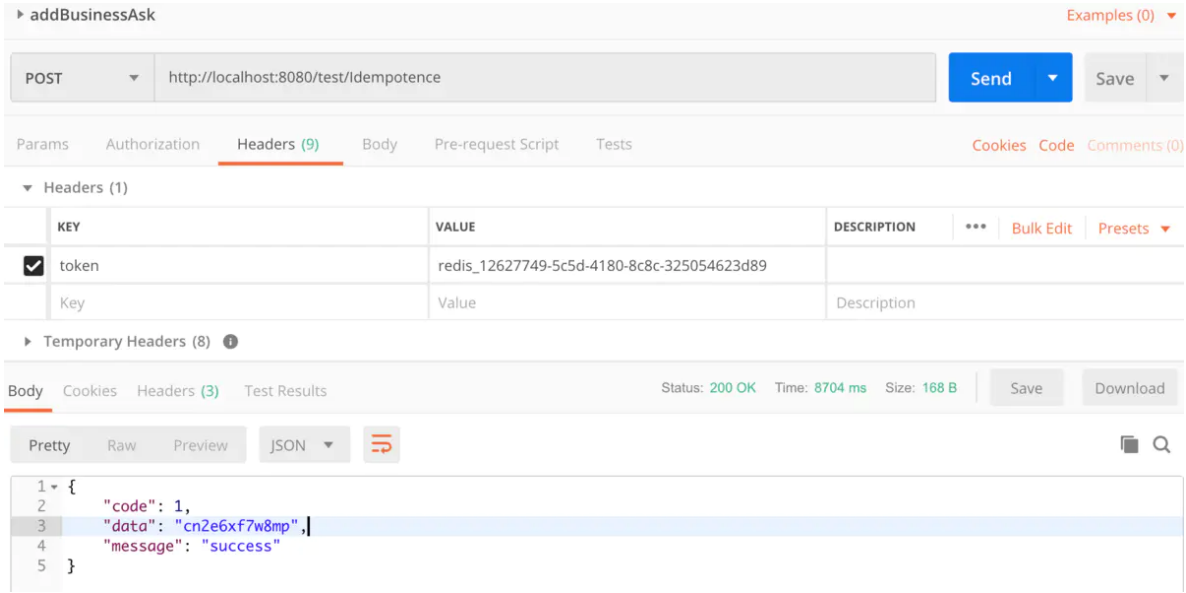
```
    @AutoIdempotent
    @PostMapping("/test/Idempotence")
    public String testIdempotence() {
        String businessResult = testService.testIdempotence();
        if (StrUtil.isNotEmpty(businessResult)) {
            ResultVo successResult = ResultVo.getSuccessResult(businessResult);
            return JSONUtil.toJsonStr(successResult);
        }
        return StrUtil.EMPTY;
    }
}
```

2:使用postman请求

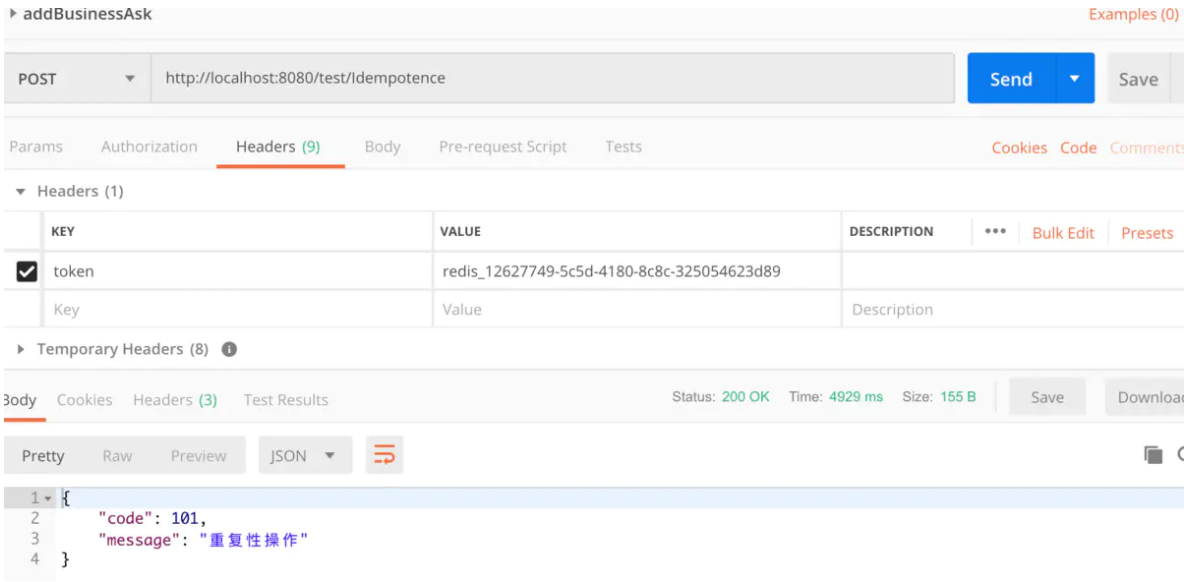
首先访问get/token路径获取到具体到token：



利用获取到到token,然后放到具体请求到header中,可以看到第一次请求成功，接着我们请求第二次：



第二次请求，返回到是重复性操作，可见重复性验证通过，再多次请求到时候我们只让其第一次成功，第二次就是失败：



六：总结

本篇博客介绍了使用springboot和拦截器、redis来优雅的实现接口幂等，对于幂等在实际的开发过程中是十分重要的，因为一个接口可能会被无数的客户端调用，如何保证其不影响后台的业务处理，如何保证其只影响数据一次是非常重要的，它可以防止产生脏数据或者乱数据，也可以减少并发量，实乃十分有益的一件事。而传统的做法是每次判断数据，这种做法不够智能化和自动化，比较麻烦。而今天的这种自动化处理也可以提升程序的伸缩性。

来自：https://www.jianshu.com/p/c806003a8530