后端

1. 项目初始化

1. **创建springboot项目：2.7.8**
2. **pom依赖**

```
<!-- web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- mysql -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
</dependency>
<!-- mybatis-plus -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.2</version>
</dependency>
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.5.2</version>
</dependency>
<!-- freemarker -->
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
</dependency>
<!-- lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

3. **yml**

```yaml
server:
  port: 9999

spring:
  datasource:
    username: root
    password: 123456
    url: jdbc:mysql:///xdb

logging:
  level:
    com.lantu: debug
```

这里没有指定连接池的情况下使用默认的连接池：com.zaxxer:HikariCP:4.0.3，使用type:进行指定

更细节的参数没有配置，有些参数有默认值

4. **测试**

这里在测试的时候遇到报错：

```
org.springframework.beans.factory.BeanDefinitionStoreException: Failed to read
candidate component class: URL
[jar:file:/Users/brandon.gu/.m2/repository/org/springframework/boot/spring-boot-
autoconfigure/2.7.8/spring-boot-autoconfigure-
2.7.8.jar!/org/springframework/boot/autoconfigure/r2dbc/ConnectionFactoryConfigurations
$PoolConfiguration.class]; nested exception is java.lang.IllegalStateException: Could
not evaluate condition on
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryConfigurations$PoolConfig
uration due to io/r2dbc/spi/ValidationDepth not found. Make sure your own configuration
does not rely on that class. This can also happen if you are @ComponentScanning a
springframework package (e.g. if you put a @ComponentScan in the default package by
mistake)
  at
org.springframework.context.annotation.ClassPathScanningCandidateComponentProvider.scan
CandidateComponents(ClassPathScanningCandidateComponentProvider.java:457) ~[spring-
context-5.3.25.jar:5.3.25]
```

原因是没有加入ComponentScan进行包的扫描

```
@ComponentScan("com.lantu.*")
```

2. Mybatis-plus代码生成

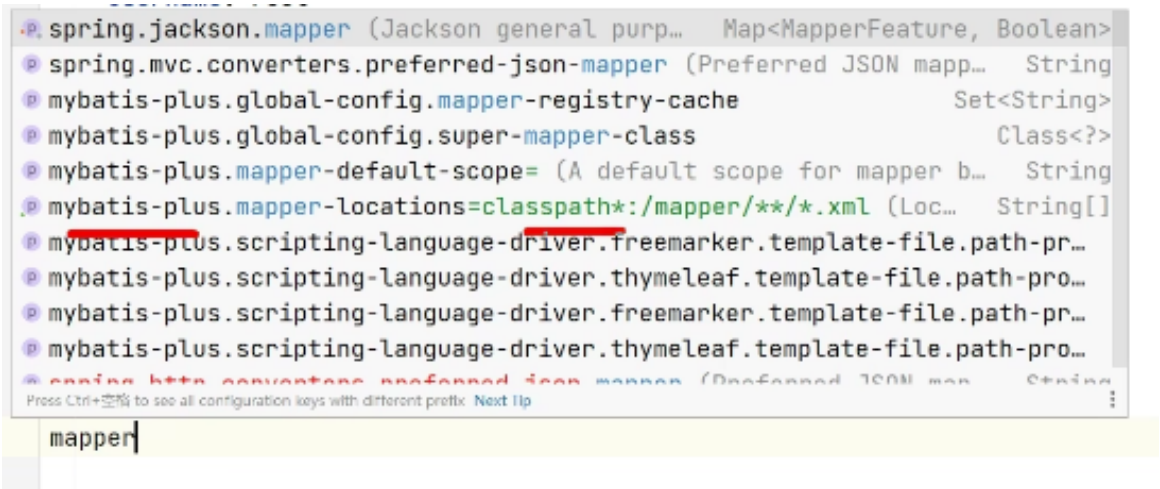
这里有相应的网站baomidou.com，可以先去查看

这段代码不需要打包到正式的目录下，因此我们放到test的目录下面

1. **编写代码生成器**

```
public static void main(String[] args) {
        String url = "jdbc:mysql:///xdb";
        String username = "root";
        String password = "123456";
        String author = "laocai";
        String outputDir = "D:\\tmp\\spring\\x-admin\\src\\main\\java";
        String basePackage = "com.lantu";
        String moduleName = "sys";
        String mapperLocation = "D:\\tmp\\spring\\x-
admin\\src\\main\\resources\\mapper\\" + moduleName;
        String tableName = "x_user,x_menu,x_role,x_role_menu,x_user_role";
        String tablePrefix = "x_";
        FastAutoGenerator.create(url, username, password)
                .globalConfig(builder -> {
                    builder.author(author) // 设置作者
                            //.enableSwagger() // 开启 swagger 模式
                            //.fileOverride() // 覆盖已生成文件
                            .outputDir(outputDir); // 指定输出目录
                })
                .packageConfig(builder -> {
                    builder.parent(basePackage) // 设置父包名
                            .moduleName(moduleName) // 设置父包模块名
                            .pathInfo(Collections.singletonMap(OutputFile.xml,
mapperLocation)); // 设置mapperXml生成路径
                })
                .strategyConfig(builder -> {
                    builder.addInclude(tableName) // 设置需要生成的表名
                            .addTablePrefix(tablePrefix); // 设置过滤表前缀
                })
                .templateEngine(new FreemarkerTemplateEngine()) // 使用Freemarker引
擎模板，默认的是Velocity引擎模板
                .execute();
    }
```

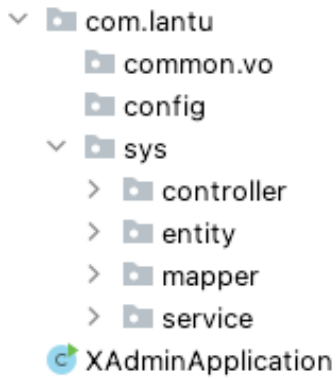
这里mybatis有一些默认的配置



这里我配置的生成代码为

```
public class CodeGenerator {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/qingqing_bird?
useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&useSSL=true&s
erverTimezone=GMT%2B8";
        String username = "root";
        String password = "12345678";
        String author = "laocai";
        String outputDir = "/Users/brandon.gu/Desktop/学习代
码/qingqing_bird/qingqing_bird_maven/qingqingbird_project/src/main/java";
        String basePackage = "com.lantu";
        String moduleName = "sys";
        String mapperLocation = "/Users/brandon.gu/Desktop/学习代
码/qingqing_bird/qingqing_bird_maven/qingqingbird_project/src/main/resources/mapper" +
moduleName;
        String tableName = "x_user,x_menu,x_role,x_role_menu,x_user_role";
        String tablePrefix = "x_";
        //这里在起名字的时候过滤掉x_的名称前缀
        FastAutoGenerator.create(url, username, password)
                .globalConfig(builder -> {
                    builder.author(author) // 设置作者
                            //.enableSwagger() // 开启 swagger 模式
                            //.fileOverride() // 覆盖已生成文件
                            .outputDir(outputDir); // 指定输出目录
                })
                .packageConfig(builder -> {
                    builder.parent(basePackage) // 设置父包名
                            .moduleName(moduleName) // 设置父包模块名
                            .pathInfo(Collections.singletonMap(OutputFile.xml,
mapperLocation)); // 设置mapperXml生成路径
                })
                .strategyConfig(builder -> {
                    builder.addInclude(tableName) // 设置需要生成的表名
                            .addTablePrefix(tablePrefix); // 设置过滤表前缀
                })
                .templateEngine(new FreemarkerTemplateEngine()) // 使用Freemarker引擎模
板，默认的是Velocity引擎模板
                .execute();
    }
}
```

运行之后得到代码的配置文件



这里直接在UserServiceImpl等实现类中实现增删改查

2. **启动类加注解，在XAdminApplication中加入注解**

```
@MapperScan("com.lantu.*.mapper")
```

3. **测试**

```
@RestController
@RequestMapping("/sys/user")
public class UserController {
    @Autowired
    private IUserService userService;

    @GetMapping
    public Result<List<User>> getAll(){
        List<User> userList = userService.list();
        return Result.success(userList);
    }
}
```

!!!注意这里如果加@Controller会返回404，因为@Controller要求返回视图，但是现在找不到这个视图!!!

!!!注意这里的主程序必须要在sys目录下!!!

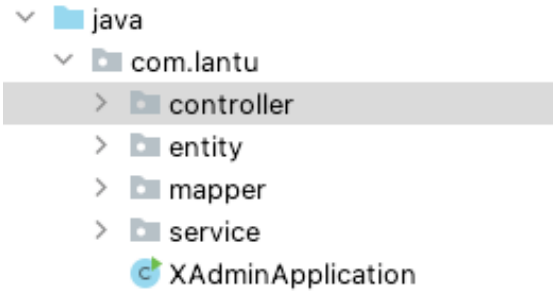
!!!注意!!!还不行把目录调整成跟它一样的目录结构，XAdminApplication上面不能套目录!!!

这里先在test中使用上述代码进行测试，判断在spring的环境中是否能跑mybatis的这段代码，然后再在端口中定义代码

```
@Controller
@RequestMapping("/user")
public class UserController {
    @Autowired
    private IUserService userService;

    @GetMapping("/all")
    public List<User> getAllUser(){
        List<User> list = userService.list();
        return list;
    }
}
```

这之后运行springboot，看localhost:8080/user/all是否能够成功地将内容搜索出来

搜索不到报404的情况从以下两个方面进行考虑，一个是@Controller是否改为@RestController，第二个是目录结构是否调整好了



3. 公共响应类(跟前端统一返回格式)

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Result<T> {
    //这里Result<T>定义范型为T，同时定义类型的时候
    //也需要把T具体的类放入进去
    //public Result<List<User>>
    private Integer code;
    private String message;
    private T data;
    //这里使用Object也可以，但是使用范型更好

    public static<T>  Result<T> success(){
        //这里static<T>定义了T为范型
        return new Result<>(20000,"success",null);
    }

    public static<T>  Result<T> success(T data){
        return new Result<>(20000,"success",data);
    }
```

```
    public static<T>  Result<T> success(T data, String message){
        return new Result<>(20000,message,data);
    }

    public static<T>  Result<T> success(String message){
        return new Result<>(20000,message,null);
    }

    public static<T>  Result<T> fail(){
        return new Result<>(20001,"fail",null);
    }

    public static<T>  Result<T> fail(Integer code){
        return new Result<>(code,"fail",null);
    }

    public static<T>  Result<T> fail(Integer code, String message){
        return new Result<>(code,message,null);
    }

    public static<T>  Result<T> fail( String message){
        return new Result<>(20001,message,null);
    }

}
```

成功三种情况：啥也不传，只传数据、只传message，既传数据又传message

失败两种情况：啥也不穿，只传数据、只传code，既传数据又传code

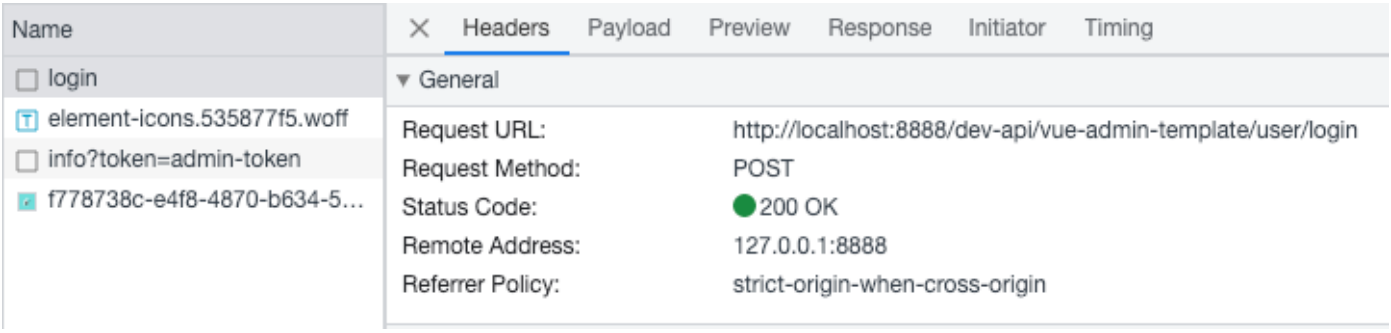
4. 登录相关接口

4.1 登录

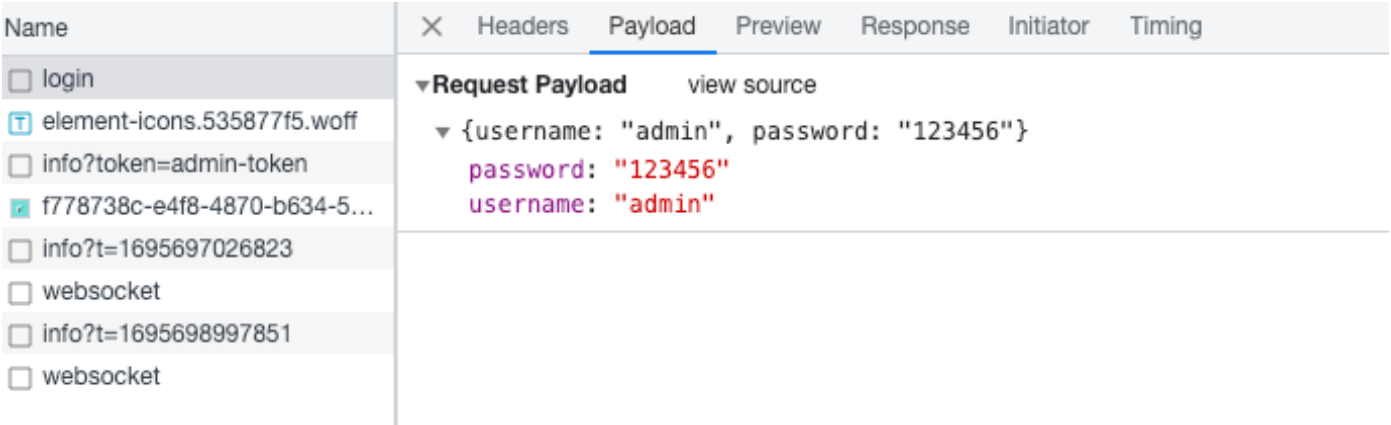
接口属性	值
url	/user/login
method	post
请求参数	username password

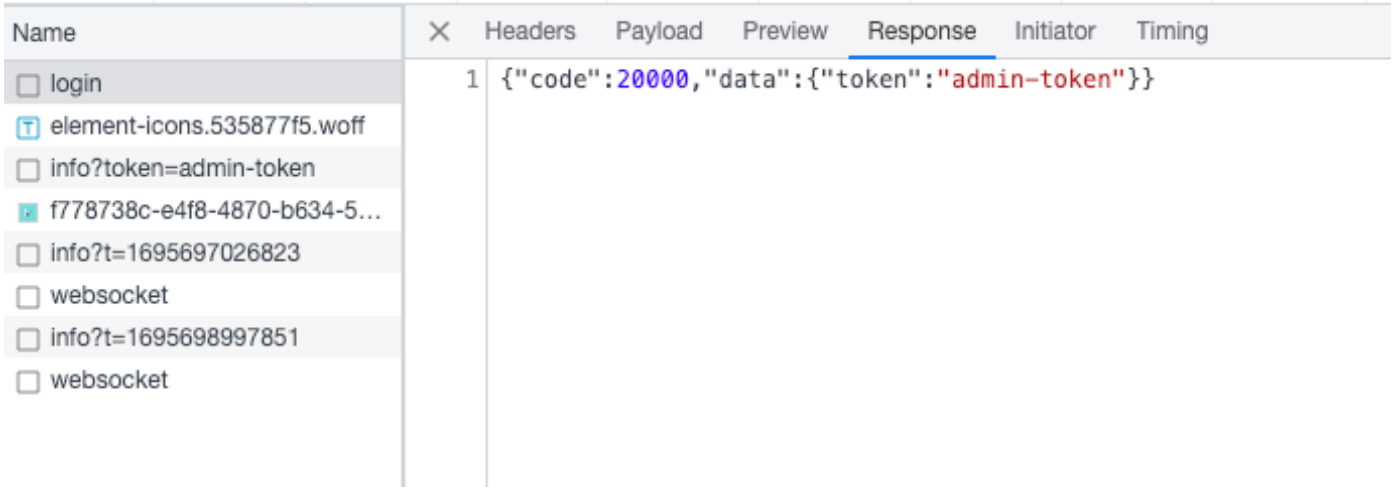
返回参数	<pre>{
	"code": 20000,
	"message": "success",
	"data": {
	"token": "846c7320-00d4-4532-9b97-031bf47bc51a"
	}
	}</pre>

先查看header登录头的数据



再查看payload传入过来之后携带的数据





先定义UserServiceImpl.java中的login函数

```
    @Override
    public Map<String, Object> login(User user) {
        //根据用户名和密码查询
        LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(User::getUsername,user.getUsername());
        //这里如果没有用户名会使查询结果为null
        wrapper.eq(User::getPassword,user.getPassword());
        User loginUser = this.baseMapper.selectOne(wrapper);
        if(loginUser != null)
        {
            //暂时用UUID，终极方案是jwt
            String key = "user:" + UUID.randomUUID();
            //返回数据
            Map<String,Object> data = new HashMap<>();
            data.put("token",key);
            return data;
        }
        return null;
    }
```

controller

```
@PostMapping("/login")
public Result<Map<String,Object>> login(@RequestBody User user){
    Map<String,Object> data = userService.login(user);
    //!!!这里验证的方法建议放到userService之中来做!!!
    //直接这样写，然后按alt+enter就会生成userService中的login方法
    if(data != null){
        return Result.success(data);
    }
    return Result.fail(20002,"用户名或密码错误");
}
```

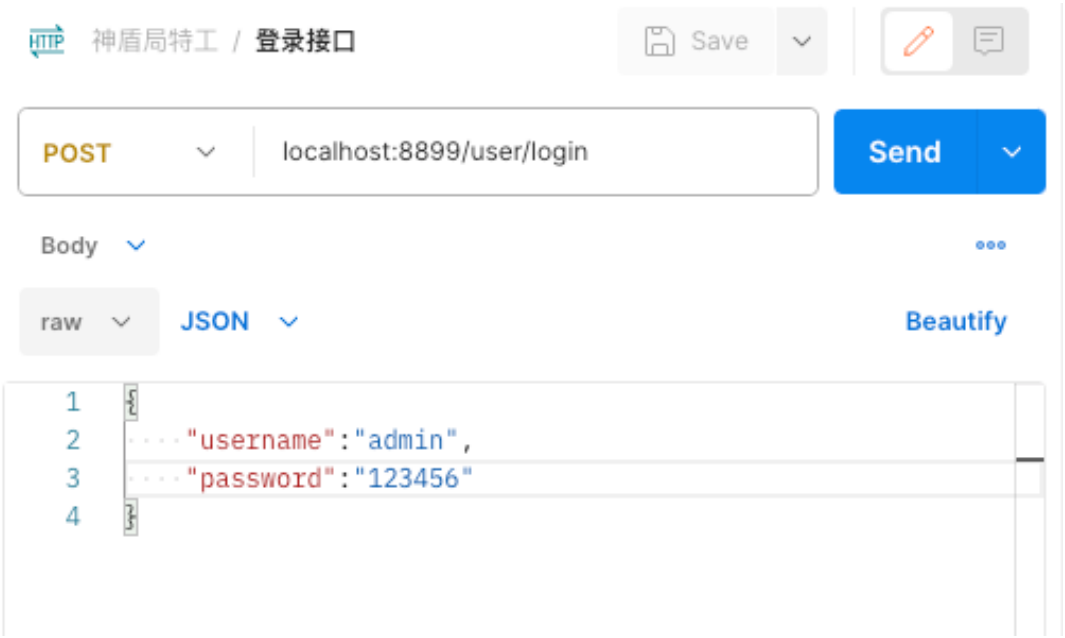
service

根据用户名和密码查询，如果结果 不为空，生成token

传统的单体是将存储结果放到session中，但是前后端分离的项目session是无效的，因为没有办法在前端访问后端的session，所以需要借助token，在UserServiceImpl中可以看到

```
public Map<String, Object> login(User user) {
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper();
    wrapper.eq(User::getUsername,user.getUsername());
    wrapper.eq(User::getPassword,user.getPassword());
    //查询条件：与传入的User用户名和密码都相同
    User loginUser = this.getOne(wrapper);
    if(loginUser != null){
        Map<String, Object> data = new HashMap<>();
        String key = "user::" + UUID.randomUUID();
        data.put("token", key);     // 待优化，最终方案jwt
        loginUser.setPassword(null);
        redisTemplate.opsForValue().set(key,loginUser,30, TimeUnit.MINUTES);
        return data;
    }
    return null;
}
```

定义到这里的时候可以使用工具测试一下，



整合redis

1. pom

```
<!-- redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2. yml

```yaml
spring:
  redis:
    host: localhost
    port: 6379
```

3. 配置类

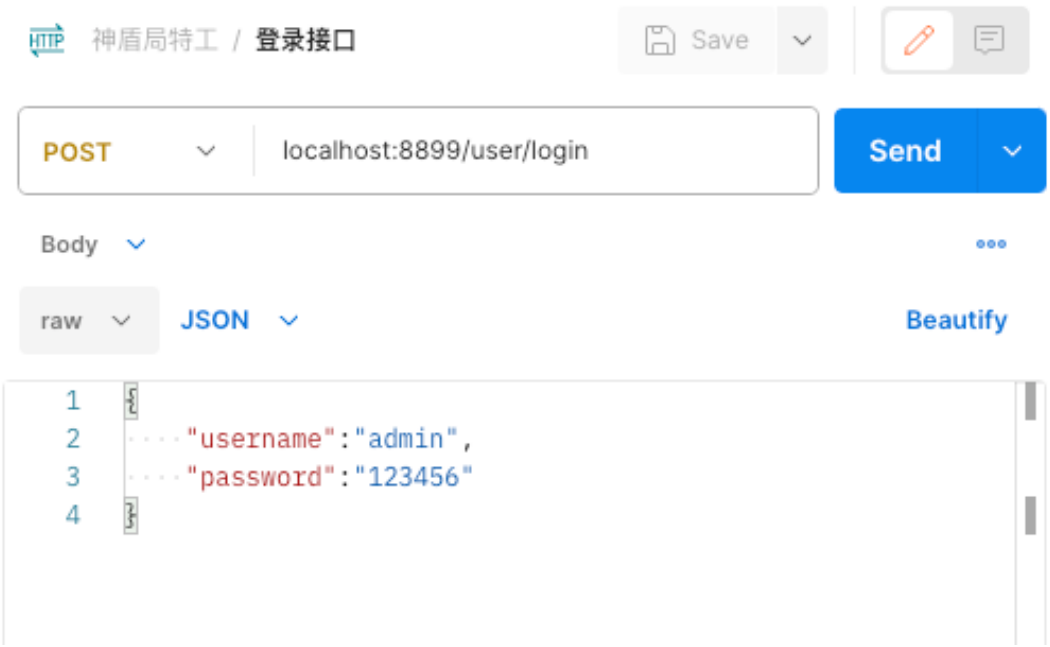
```
@Configuration
public class MyRedisConfig {
    @Resource
    private RedisConnectionFactory factory;

    @Bean
    public RedisTemplate redisTemplate(){
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
        redisTemplate.setConnectionFactory(factory);
        //设置连接工厂，这样才能跟application.yml中配置的redis连接起来
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        //!!!对redis进行序列化处理!!!
        //如果不做序列化处理，可能看不懂redis中保存的内容，看起来像乱码，这里需要看一下redis的教
程
        //setKeySerializer是针对键进行序列化处理

        Jackson2JsonRedisSerializer<Object> serializer = new
Jackson2JsonRedisSerializer<>(Object.class);
        redisTemplate.setValueSerializer(serializer);
        //setValueSerializer是针对值进行序列化处理
        //如果简单的处理成这样就可以，但是如果是包含了复杂的类型，比如集合、集合中又包含了对象，此
时会出现问题无法反序列化

        ObjectMapper om = new ObjectMapper();
        //ObjectMapper针对序列进行对象映射，对序列时序进行设置，方便反序列化
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
        om.setTimeZone(TimeZone.getDefault());
        om.configure(MapperFeature.USE_ANNOTATIONS, false);
        om.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
        om.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS, false);
        om.activateDefaultTyping(LaissezFaireSubTypeValidator.instance
,ObjectMapper.DefaultTyping.NON_FINAL, JsonTypeInfo.As.PROPERTY);
        om.setSerializationInclusion(JsonInclude.Include.NON_NULL);
        serializer.setObjectMapper(om);

        return redisTemplate;
    }
}
```

很多复杂的项目会针对redis封装工具类

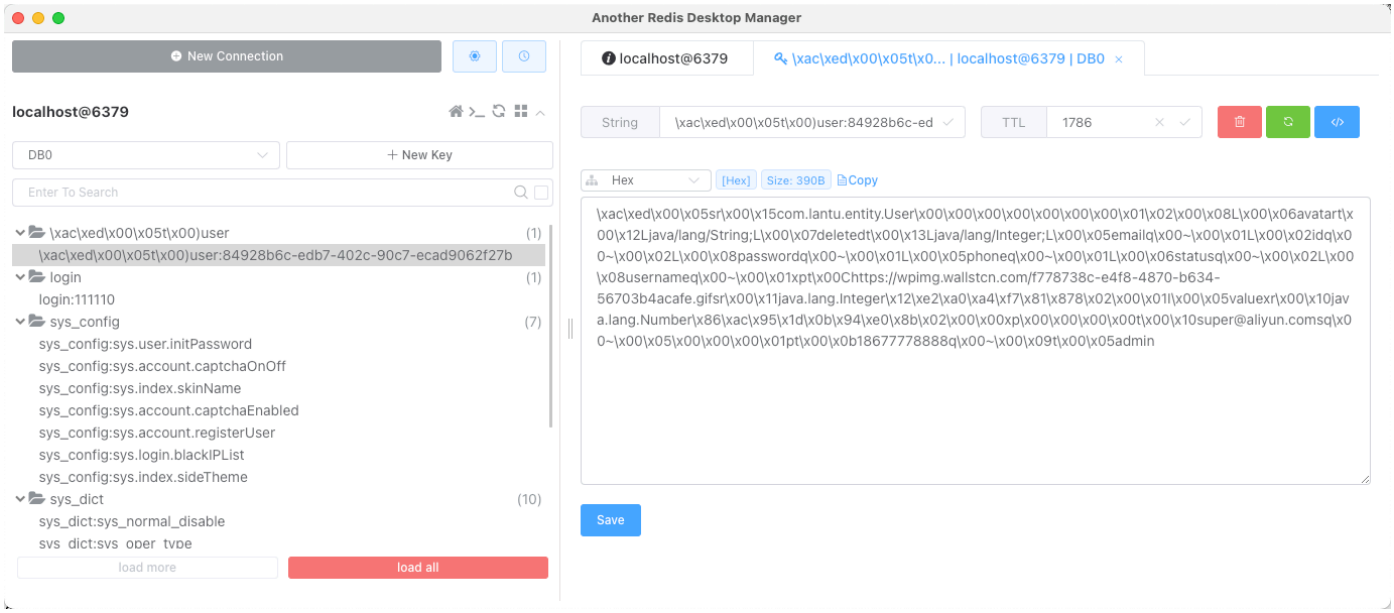
接下来修改UserServiceImpl类，将RedisTemplate注入进来

```
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
IUserService {

    @Autowired
    private RedisTemplate redisTemplate;
    ......

    @Override
    public Map<String, Object> login(User user) {
        ......
        if(loginUser != null)
        {
            ......
            //存入redis
            loginUser.setPassword(null);
            redisTemplate.opsForValue().set(key,loginUser);
            ......
        }
    }
}
```

接下来再次调用一下登录接口，然后去another redis manager中查看存储到数据



redis存储到数据部分



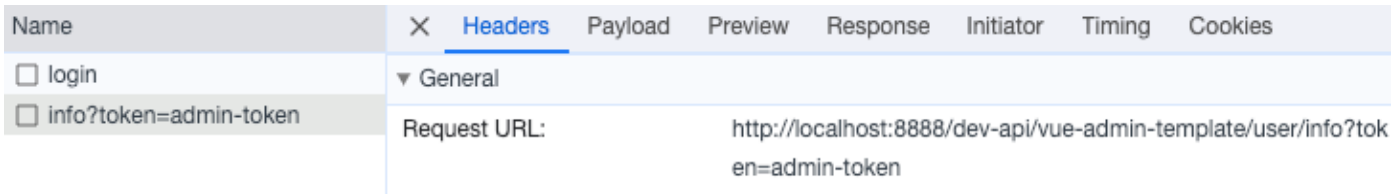
4.2 获取用户信息

接口属性	值
url	/user/info?token=xxx
method	get
请求参数	token

返回参数	{
	"code": 20000,
	"message": "success",
	"data": {
	"roles": [
	"admin"
	],
	"name": "admin",
	"avatar": "https://wpimg.wallstcn.com/f778738c-e4f8-4870-b634-56703b4acafe.gif"
	}
	}

这里发送info请求的时候需要在请求头上加上token

这里发送到info请求需要在请求头上加上token



而返回的有数据列表

avatar为头像的地址

controller(UserController.java)

@Param("token")是获取请求头中的token参数

```
@GetMapping("/info")
    public Result<?> getUserInfo(@Param("token") String token){
        Map<String,Object> data = userService.getUserInfo(token);
        if(data != null){
            return Result.success(data);
        }
        return Result.fail(20003,"用户信息获取失败");
    }
```

service(UserServiceImpl.java)

```
@Override
public Map<String, Object> getUserInfo(String token) {
        //根据token从redis中获取用户信息
        Object obj = redisTemplate.opsForValue().get(token);

        //反序列化，因为存入的时候存在序列化操作
        /***
         * 之前放入的时候序列化操作
         * Jackson2JsonRedisSerializer<Object> serializer = new
Jackson2JsonRedisSerializer<>(Object.class);
         *          redisTemplate.setValueSerializer(serializer);
         *          //setValueSerializer是针对值进行序列化处理
         *          //如果简单的处理成这样就可以，但是如果是包含了复杂的类型，比如集合、集合中又包含了
对象，此时会出现问题无法反序列化
         *
         *          ObjectMapper om = new ObjectMapper();
         *          //ObjectMapper针对序列进行对象映射，对序列时序进行设置，方便反序列化
         *          om.setVisibility(PropertyAccessor.ALL,
JsonAutoDetect.Visibility.ANY);
```

```
         *         om.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
         *         om.setTimeZone(TimeZone.getDefault());
         *         om.configure(MapperFeature.USE_ANNOTATIONS, false);
         *         om.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false);
         *         om.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS, false);
         *         om.activateDefaultTyping(LaissezFaireSubTypeValidator.instance
,ObjectMapper.DefaultTyping.NON_FINAL, JsonTypeInfo.As.PROPERTY);
         *         om.setSerializationInclusion(JsonInclude.Include.NON_NULL);
         *         serializer.setObjectMapper(om);
         *
         */
        if(obj != null){
            //1.将对象转成JSON字符串 2.外面套JSON.parseObject反序列化成User对象
            User loginUser = JSON.parseObject(JSON.toJSONString(obj),User.class);
            //这里需要进行反序列化操作

            /***
             * x_user_role中被插入了数据
             *  id  user_id  role_id
             *  1      1        1
             *  2      1        2
             * 1号用户为1号角色，然后查看x_role中角色表对应信息
             *  role_id  role_name   role_desc
             *     1      admin      超级管理员
             *     2       hr        人事管理员
             *     3     normal      普通员工
             * 这里角色名称需要进行关联查询，目前能单表查的还是尽量单表查
             */
            Map<String, Object> data = new HashMap<>();
            data.put("name",loginUser.getUsername());
            data.put("avatar",loginUser.getAvatar());

            //角色
            List<String> roleList =
this.baseMapper.getRoleNameByUserId(loginUser.getId());
            data.put("roles",roleList);

            return data;
        }
        return null;
    }
```

mapper.xml

parameterType为传入参数，resultType为返回参数

```
<select id="getRoleNamesByUserId" parameterType="Integer" resultType="String">
    SELECT
    b.role_name
    FROM x_user_role a,x_role b
    WHERE a.`user_id` = #{userId}
    AND a.`role_id` = b.`role_id`
</select>
```

一、这里写的时候先在DBeaver中试写一下sql语句，能够跑起来再在mapper.xml中使用上

1.内连接

```
select
   b.role_name
from qingqing_bird.x_user_role a,qingqing_bird.x_role b
where
   a.role_id = b.role_id
and a.user_id = 1;
```

2.外连接

```
select
   b.role_name
from qingqing_bird.x_user_role a left join qingqing_bird.x_role b
on a.role_id = b.role_id
where a.user_id = 1;
```

二、然后需要在resources/mapper.sys/UserMapper.xml中把sql语句写出来

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.lantu.mapper.UserMapper">
    <select id="getRoleNameByUserId" parameterType="Integer" resultType="string">
        SELECT
        b.role_name
        FROM x_user_role a,x_role b
        WHERE a.`user_id` = #{userId}
        AND a.`role_id` = b.`role_id`
    </select>
</mapper>
```

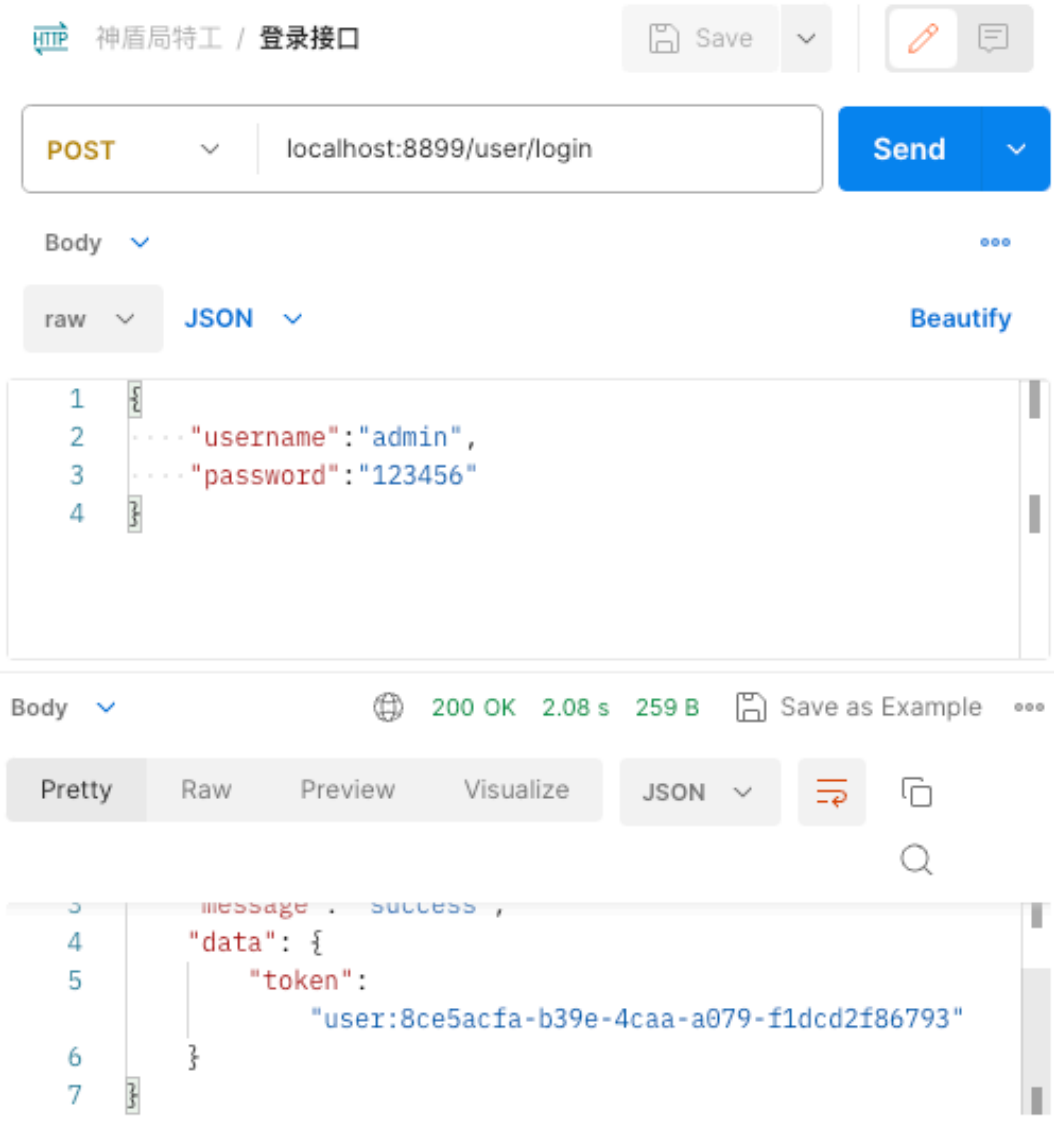
三、接下来需要在mapper/UserMapper中定义接口

```
public interface UserMapper extends BaseMapper<User> {
    public List<String> getRoleNameByUserId(Integer userId);
}
```

四、角色列表的调用，在UserServiceImpl的getUserInfo.java中加入

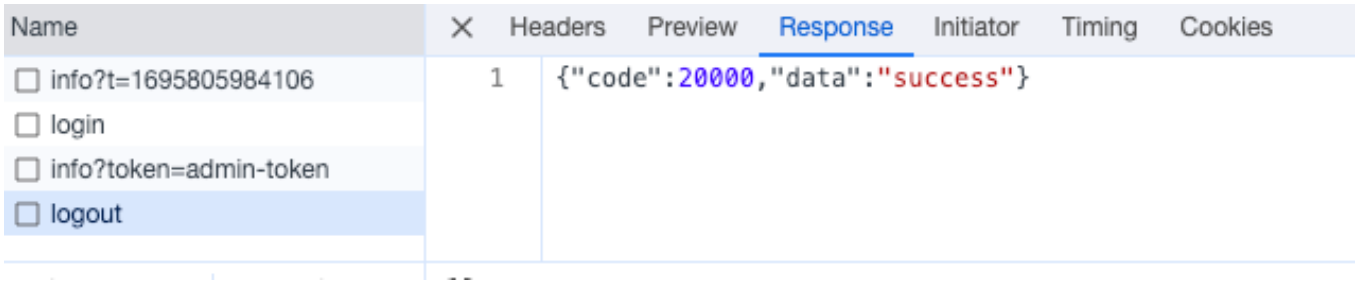
```
List<String> roleList = this.baseMapper.getRoleNameByUserId(loginUser.getId());
data.put("roles",roleList);
```

重启测试一下接口，这里需要先打开登录接口，因为我们需要先获得一个token，然后才能将token传入

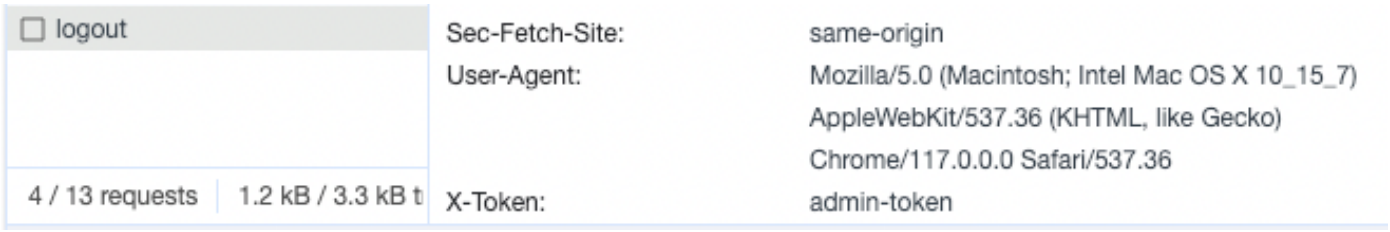


4.3 注销

点击一下注销之后，调用接口logout



注意这里在请求头上也带了token



登录之后所有操作在请求头上都带有token，因为请求拦截器塞了一个X-token进来

```
// request interceptor
service.interceptors.request.use(
  config => {
    // do something before request is sent

    if (store.getters.token) {
      // let each request carry token
      // ['X-Token'] is a custom headers key
      // please modify it according to the actual situation
      config.headers['X-Token'] = getToken()
    }
    return config
  },
```

接口属性	值
url	/user/logout
method	post
请求参数	

返回参数	{
	"code": 20000,
	"message": "注销成功",
	"data": null
	}

controller

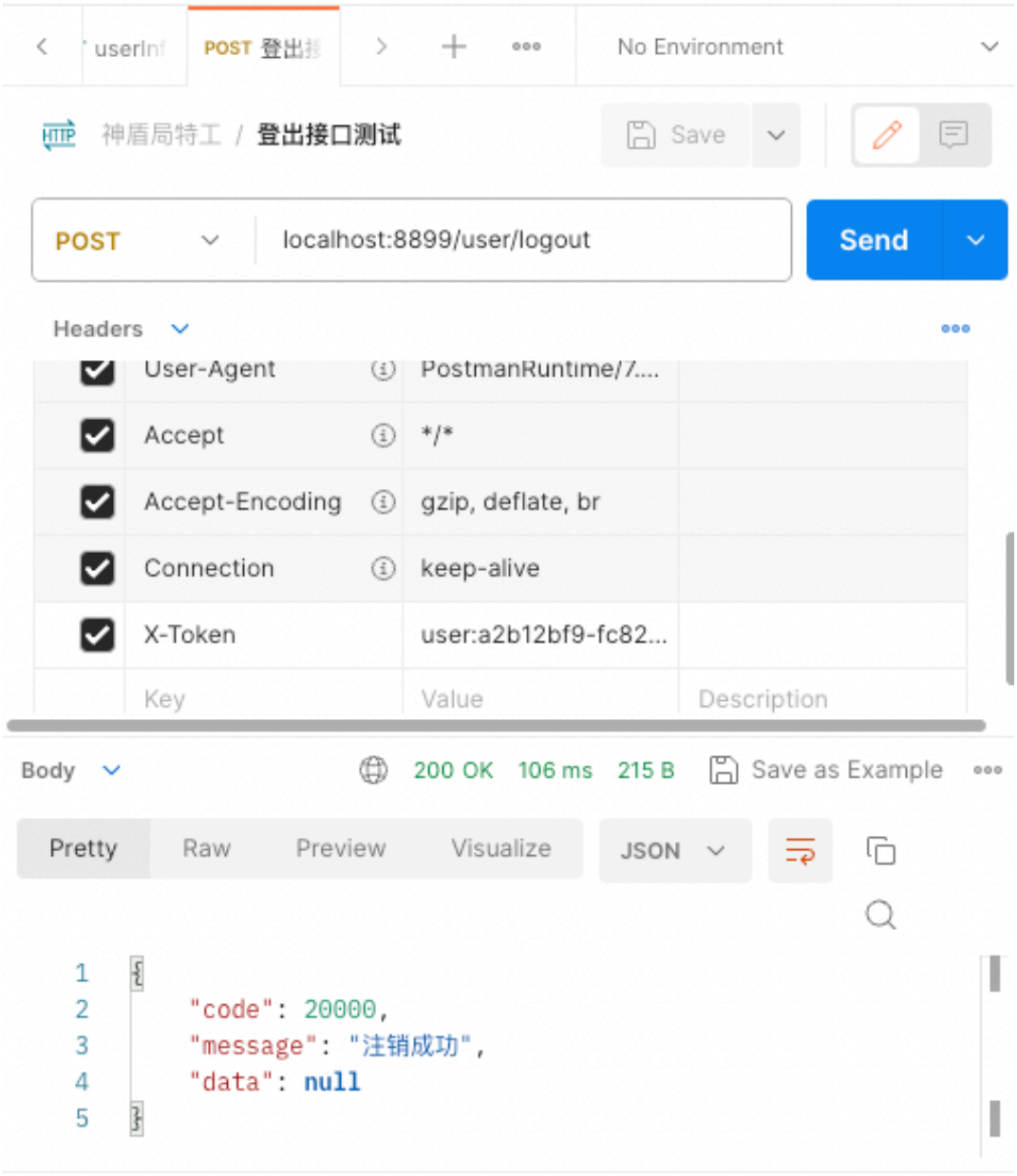
```
@PostMapping("/logout")
public Result<?> logout(@RequestHeader("X-Token") String token){
    userService.logout(token);
    return Result.success("注销成功");
}
```

与之前RequestParam区别：RequestParams是从参数中提取，直接放到url中，而RequestHeader则是从Header头中抽取

service

```
public void logout(String token) {
    redisTemplate.delete(token);
}
```

如果使用springsecurity这里的逻辑都需要改动，暂时这么做比较合理



注意如果报405错误，可能是get、post请求写错了

5.删除前端部分代码

进入到src/api/user.js之中，修改一下url部分，将/vue-admin-template删除掉

```
export function login(data) {
  return request({
    url: '/user/login',
    method: 'post',
    data
  })
}
export function getInfo(token) {
  return request({
    url: '/user/info',
    method: 'get',
    params: { token }
  })
}
export function logout() {
  return request({
    url: '/user/logout',
    method: 'post'
  })
}
```

修改.env.development文件

```
VUE_APP_BASE_API = 'http://localhost:8888'
```

在使用.env.development的时候

6. 跨域处理

推荐的跨域解决方法：

开发环境 生产环境

cors cors

Proxy nginx

方法一：在接口上加@CrossOrigin

方法二：进行全局跨域处理

注意这里的CorsFilter使用springboot中的

```
@Configuration
public class MyCorsConfig {
```

```
    @Bean
    public CorsFilter corsFilter(){
        System.out.println("corsFilter");
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedOrigin("http://localhost:8888");
        //config.addAllowedOrigin("*");
        //config.addAllowedOriginPattern("*");
        //如果是多个url可能需要调用多次
        config.setAllowCredentials(true);
        config.addAllowedMethod("*");
        config.addAllowedHeader("*");

        UrlBasedCorsConfigurationSource configurationSource = new
UrlBasedCorsConfigurationSource();
        configurationSource.registerCorsConfiguration("/**",config);

        return new CorsFilter(configurationSource);
    }
}
```

!!!addAllowedOrigin、setAllowCredentials、addAllowedMethod、addAllowedHeader缺一不可，否则就会报错，还是无法解决跨域问题!!!

注意点：这里的config.addAllowedOriginPattern("*")这样的描写形式才能加入 * ， 8888为前端定义的端口，如果报500的时候证明redis没有被打开!!!

7. 用户管理接口

接口	说明
查询用户列表	分页查询
新增用户	
根据id查询用户	
修改用户	
删除用户	逻辑删除

7.1 查询用户列表

1. controller

```
@GetMapping("/list")
public Result<Map<String,Object>> addUser(@RequestParam(value = "username",required
= false) String username,
                                          @RequestParam(value = "phone",required =
false) String phone,
                                          @RequestParam(value = "pageNo",required =
false) Long pageNo,
                                          @RequestParam(value = "pageSize",required
= false)Long pageSize){
    //除此之外还需要pageNo和pageSize
    LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();
    //这里lambdaQueryWrapper进行了一次重载，Children eq(boolean condition, R column,
Object val);
    //满足condition条件的才进行，可参照Children函数
    // default Children eq(R column, Object val) {
    //    return this.eq(true, column, val);
    // }

 lambdaQueryWrapper.eq(StringUtils.hasLength(username),User::getUsername,username);
    //hasLength相当于!=null&&!=""
    lambdaQueryWrapper.eq(StringUtils.hasLength(phone),User::getPhone,phone);
    Page<User> page = new Page<>(pageNo,pageSize);
    //注意这个Page是baomidou中的Page
    /***
     * 这里可以点进去看Page的构造函数
     * public Page(long current,long size) { this(current, size, 0L);}
     */
    userService.page(page,lambdaQueryWrapper);
    //把page和查询的lambdaQueryWrapper放入进去

    //将前端需要传入的数据放进去，这里前后端的数据必须保持统一
    Map<String,Object> data = new HashMap<>();
    data.put("total",page.getTotal());
    data.put("rows",page.getRecords());
    return Result.success(data);
}
```

注意在运行完成这段内容之后，返回的值中data中的total值为0，说明此时并没有进行分页拦截，并且控制台中没有显示出select count(*)这段的内容，因此我们需要在config下面加入MpConfig分页拦截器

2. 分页拦截器

```
@Configuration
public class MpConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new
PaginationInnerInterceptor(DbType.MYSQL));
        return interceptor;
    }
}
```

加入分页拦截器之后，total不为零了，并且控制台上出现了select count(*)部分，查询就能够成功的运行

7.2 新增用户

密码加密处理，用BCryptPasswordEncoder，涉及登录逻辑改动

```
@PostMapping
public Result<?> addUser(@RequestBody User user){
    //使用@RequestBody的原因是前端传过来的是一个json数据
    userService.save(user);
    return Result.success("新增用户成功");
}
```

7.2.1 密码加密处理

1.在pom.xml中加入spring-security-core的依赖

2.在XAdminApplication.java中配置Bean

```
@Bean
public PasswordEncoder passwordEncoder(){
    return new BCryptPasswordEncoder();
}
```

然后在UserController中注入并配置

```
@Autowired
private PasswordEncoder passwordEncoder;

......
@PostMapping
public Result<?> addUser(@RequestBody User user){
    //使用@RequestBody的原因是前端传过来的是一个json数据
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    //这里做了加盐的处理，因此每一次传入的密码都是不一样的，这会影响我们的登录逻辑
    userService.save(user);
    return Result.success("新增用户成功");
}
```

接下来数据库中所有的密码都改成加密后的字符串

id	username	password
1	admin	$2a$10$BfYVPaCL
2	zhangsan	$2a$10$BfYVPaCL
3	lisi	$2a$10$BfYVPaCL
4	wangwu	$2a$10$BfYVPaCL
5	zhaoer	$2a$10$BfYVPaCL
6	songliu	$2a$10$BfYVPaCL
7	aaaa	$2a$10$BfYVPaCL
8	aaa1	$2a$10$BfYVPaCL
9	aaa11	$2a$10$BfYVPaCL
10	aaa12	$2a$10$BfYVPaCL

然后需要修改登录的判断密码逻辑，变成根据用户查询，然后比对密码

旧的逻辑

```
@Override
public Map<String, Object> login(User user) {
    //根据用户名和密码查询
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(User::getUsername,user.getUsername());
    //这里如果没有用户名会使查询结果为null
    wrapper.eq(User::getPassword,user.getPassword());
    User loginUser = this.baseMapper.selectOne(wrapper);
    if(loginUser != null)
    {
        //暂时用UUID，终极方案是jwt
        String key = "user:" + UUID.randomUUID();


        //存入redis
        loginUser.setPassword(null);
        redisTemplate.opsForValue().set(key,loginUser,30, TimeUnit.MINUTES);
```

```
        //返回数据
        Map<String,Object> data = new HashMap<>();
        data.put("token",key);
        return data;
    }
    return null;
}
```

新的逻辑

```
    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    public Map<String, Object> login(User user) {
        //根据用户名和密码查询
        LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(User::getUsername,user.getUsername());
        User loginUser = this.baseMapper.selectOne(wrapper);
        if(loginUser != null &&
passwordEncoder.matches(user.getPassword(),loginUser.getPassword()))
        {
            //暂时用UUID，终极方案是jwt
            String key = "user:" + UUID.randomUUID();

            //存入redis
            loginUser.setPassword(null);
            redisTemplate.opsForValue().set(key,loginUser,30, TimeUnit.MINUTES);

            //返回数据
            Map<String,Object> data = new HashMap<>();
            data.put("token",key);
            return data;
        }
        return null;
    }
```

7.3 修改用户

首先写后端的接口代码，修改使用put请求

!!!注意：修改用户的时候不返回密码!!!

```
 @PutMapping
  public Result<?> updateUser(@RequestBody User user){
      user.setPassword(null);
      //修改的时候不展示密码
      userService.updateById(user);
      return Result.success("修改用户成功");
  }
```

接着由于每一个用户旁边需要增加一个修改/删除按钮，因此需要查询到对应id的信息，再写一个通过id去查找用户获取用户信息的方法

```
@GetMapping("/{id}")
public Result<User> getUserById(@PathVariable("id") Integer id){
    User user = userService.getById(id);
    return Result.success(user);
}
//同时每一个属性旁边还有一个按钮：修改、删除，点开之后要能显示信息，
//因此这里再写一个接口接收返回的信息
```

7.4 删除用户

```
@DeleteMapping("/{id}")
public Result<User> deleteUserById(@PathVariable("id") Integer id){
  userService.removeById(id);
  return Result.success("删除用户成功");
}
```

上面的删除属于物理删除，但是很多时候公司的数据都很重要，不允许直接删除，所以此时需要逻辑删除，因此在数据表中出现一个deleted字段进行逻辑删除

mybatisplus中帮我们集成了逻辑删除的功能，这样子其他逻辑就不用动了

利用MyBatisPlus做逻辑删除处理，在application.yml中配置

```
mybatis-plus:
  global-config:
    db-config:
      logic-delete-field: deleted
      logic-delete-value: 1
      logic-not-delete-value: 0
```

这里标记了逻辑删除使用deleted字段，1为被删除，0为没有被删除，然后回到前端。