目录

---

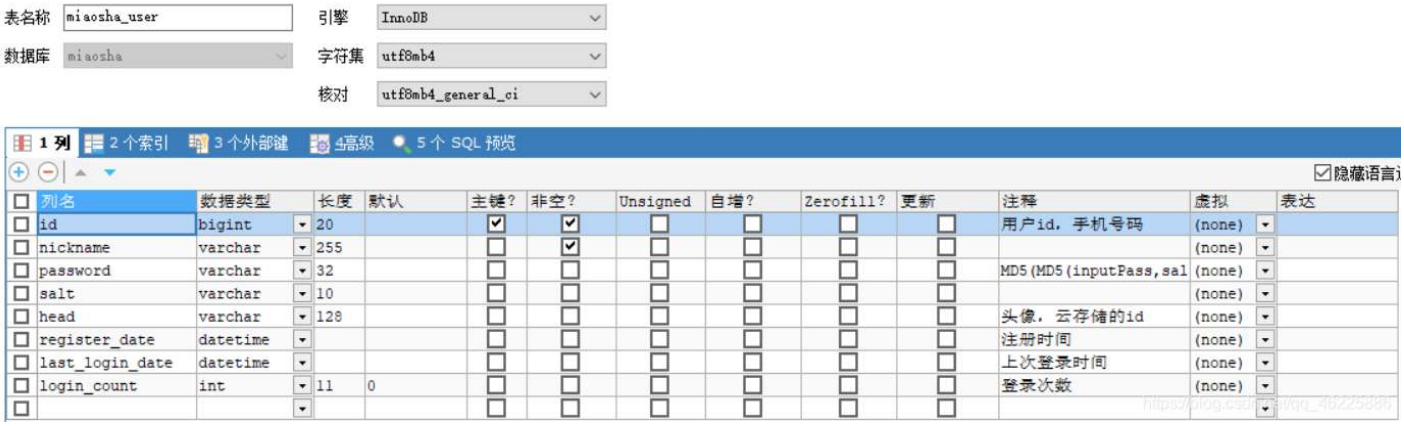
1. 登录过程中，密码两次MD5加密

1.1 为啥用两次MD5哇?

1. 第一次MD5，是针对 传输安全 做的MD5加密，因为 http是明文传递 ，如果不进行加密的话，密码就直接被劫持了。
(Password1 = MD5(inputPassword,固定的salt值)，salt为字符串)
2. 第二次MD5，是针对 数据库安全 做的MD5加密，保证数据库的防盗安全。若不进行二次加密，MD5值经数据库获取，可直接被MD5转换器直接转换为用户密码，不安全。
(Password2 = MD5(Password1,随机的salt值))

---

2. 构建数据库表



2.1 几个需要注意的点

- 字符集采用的是 utf8mb4 (most bytes 4)。简单来说，utf8mb4是 utf8的超集 ，能够用 4个字节 存储更多的字符。标准UTF-8字符集编码可以用1~4个字节取编码21位字符，但是在 MySQL中，utf8最多使用3个字节 ，像一些表情emoji和不常用的字符如“墅”需要用4个字节才能表示出来。用utf8mb4能解决以上问题。
- 数据库中存储了 "动态"salt值

---

3. 针对MD5加密功能，封装了专用工具类

以下MD5包的Maven依赖了解以下

```
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.6</version>
        </dependency>
```

3.1 工具类代码

```
public class MD5Util {
    //静态的salt，用于第一次MD5
    private static final String salt = "1a2b3c4d";

    private static String md5(String src){
        //调用DigestUtils，实现md5处理
```

```
        return DigestUtils.md5Hex(src);
    }

    /**
     * 第一次MD5处理
     * @param inputPass
     * @return
     */
    public static String inputPassToFormPass(String inputPass){
        //这里没加“”出现了问题？？？
        String pass ="" + salt.charAt(1) + salt.charAt(7) + inputPass
                + salt.charAt(3) + salt.charAt(5);
        //System.out.println(pass);
        return md5(pass);
    }

    ...
}
```

- 我在第一次处理加密时，拼接 字符 时没有添加 ""，出现了登录验证失败的问题

---

4. 加入JSR参数校验

4.1 JSR参数校验

- 我们看如下，代码，在登录处理过程中，我们要用 代码 实现对前端传过来的id和password进行校验（我们这里是验证 非空），引入JSR参数校验之后，能够将这些代码省去

```
    @PostMapping("/do_login")
    @ResponseBody
    public Result<Boolean> doLogin(LoginVo loginVo){
        log.info(loginVo.toString());

        //参数校验
        String mobile = loginVo.getMobile();
        String password = loginVo.getPassword();
        if(StringUtils.isEmpty(password)){
            return Result.error(CodeMsg.PASSWORD_EMPTY);
        }
        if(! ValidatorUtil.isMobile(mobile)){
            return Result.error(CodeMsg.MOBILE_ERROR);
        }

        CodeMsg msg = miaoShaUserService.login(loginVo);

```

```
        if(msg.getCode() == 0){
            return Result.success(true);
        }else {
            return Result.error(msg);
        }
    }
```

- 进行JSR参数校验升级

我们先看一下导入的包

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
```

我们在doLogin()方法上，加上JSR验证， `@Valid` 注解

```
public Result<CodeMsg> doLogin(@Valid LoginVo loginVo)
```

被标注的参数，我们进入它的实现类中，对其中的字段进行约束，如下（ `@NotNull` ， `@Length` ， `@IsMobile` ，其中@IsMobile是我们自定义的注解）

```
@Data
public class LoginVo {

    @IsMobile
    @NotNull
    private String mobile;

    @NotNull
    @Length(min = 32)
    private String password;
}
```

4.2 @IsMobile自定义注解

我们看一下它的代码（这个注解的写法，根据已有注解@NotNull，仿写而来），它实现的是对手机号码的验证

```
@Target({ElementType.FIELD, ElementType.ANNOTATION_TYPE, ElementType.CONSTRUCTOR, Element
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(
        validatedBy = {IsMobileValidator.class}
)
public @interface IsMobile {

    boolean required() default true;

        //以下三条语句，足矣
        //我们在其中添加错误信息
    String message() default "手机号码格式错误";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

- `@Target`：表示的是能够标注的范围
- `@Constraint`：这个注解帮助我们处理逻辑，其中有 `IsMobileValidator.class` 是真正处理逻辑的类，我们看看它的代码

```
public class IsMobileValidator implements ConstraintValidator<IsMobile, String> {

    private boolean required = false;

    @Override
    public boolean isValid(String s, ConstraintValidatorContext constraintValidatorContex
        if(required){
            //在必须有值的情况下
            return ValidatorUtil.isMobile(s);
        }else {
            //在不要求有值的情况下
            if(StringUtils.isEmpty(s)){
                //空值是允许的
                return true;
            }else {
                //有值就给它判断判断
                return ValidatorUtil.isMobile(s);
            }
        }
    }

```

```
    @Override
    public void initialize(IsMobile constraintAnnotation) {
        required = constraintAnnotation.required();
    }
}
```

- 先看类的声明部分，public class IsMobileValidator implements `ConstraintValidator<IsMobile, String>`，它有两个泛型，第一个是 自定义的注解类，第二个是要 验证的参数类型，另外实现该接口的逻辑类， 被spring管理成bean，可以在需要的地方进行装配
- 其中有一个 `initialize`，初始化方法，它调用的是我们自定义注解中写的 `required()` 方法，默认需要有值
- 另一个方法 `isValid`，则对逻辑进行验证，true验证通过，false验证失败

---

5. 全局异常处理器

5.1 我们为什么要引入全局异常处理器?

- 一边想，一边看一下下面这个方法

```
    public CodeMsg login(LoginVo loginVo){
        if(loginVo == null){
            return CodeMsg.SERVER_ERROR;
        }

        String mobile = loginVo.getMobile();
        String password = loginVo.getPassword();
        //判断手机号是否存在
        MiaoShaUser user = getById(Long.parseLong(mobile));
        if(user == null){
            return CodeMsg.MOBILE_NOT_EXIST;
        }

        //验证密码
        String DBPass = user.getPassword();
        //这里对前端来的密码第二次MD5处理
        String formPassToDBPass = MD5Util.formPassToDBPass(password, user.getSalt());
        if(!formPassToDBPass.equals(DBPass)){
            return CodeMsg.PASSWORD_ERROR;
        }


```

```
        return CodeMsg.SUCCESS;
    }
```

它的返回值是CodeMsg，而在业务中，方法对应的返回值应该是 确切的 ，我们登陆，返回应该为 true 或 false，所以，我们要对这里进行优化

5.2 优化

代码如下

```
    public boolean login(LoginVo loginVo){
        if(loginVo == null){
            throw new GlobalException(CodeMsg.SERVER_ERROR);
        }

        String mobile = loginVo.getMobile();
        String password = loginVo.getPassword();
        //判断手机号是否存在
        MiaoShaUser user = getById(Long.parseLong(mobile));
        if(user == null){
            throw new GlobalException(CodeMsg.MOBILE_NOT_EXIST);
        }

        //验证密码
        String DBPass = user.getPassword();
        //这里对前端来的密码第二次MD5处理
        String formPassToDBPass = MD5Util.formPassToDBPass(password, user.getSalt());
        if(!formPassToDBPass.equals(DBPass)){
            throw new GlobalException(CodeMsg.PASSWORD_ERROR);
        }

        return true;
    }
```

我们可以发现，对应的参数验证，并没有返回值，而是直接 抛出异常 ，而且我们也将 返回值进行了修改 ，执行到方法的最后，能够返回ture

5.3 全局异常

```
public class GlobalException extends RuntimeException {
    private CodeMsg codeMsg;

    public GlobalException(CodeMsg codeMsg){
        this.codeMsg = codeMsg;
```

```
    }

    public CodeMsg getCodeMsg() {
        return codeMsg;
    }
}
```

- 全局异常就比较简单了，它 继承了RuntimeException类 ，其中包含我们需要返回的信息CodeMsg的字段

5.4 全局异常处理器

这个处理器可就值得说一说了！

```
@ControllerAdvice
@ResponseBody
public class GlobalExceptionHandler {

    @ExceptionHandler(value = Exception.class)
    public Result<String> exceptionHandler(HttpServletRequest request,Exception e){
        if(e instanceof GlobalException){
            GlobalException ge = (GlobalException) e;

            CodeMsg codeMsg = ge.getCodeMsg();
            return Result.error(codeMsg);
        } else if(e instanceof BindException){
            //获取错误列表，拿取其中的第一个
            BindException be = (BindException) e;
            List<ObjectError> allErrors = be.getAllErrors();
            ObjectError error = allErrors.get(0);

            String message = error.getDefaultMessage();
            return Result.error(CodeMsg.BIND_ERROR.fillArgs(message));
        }else {
            return Result.error(CodeMsg.SERVER_ERROR);
        }

    }
}
```

- `@ControllerAdvice`： 它是增强的Controller，能够实现 全局异常处理 和全局数据绑定
- 配合 `@ExceptionHandler(value = Exception.class)` ，它能够实现对所有异常的接受，而在方法中，对不同的异常进行处理

6. 关注一下参数替换的方法

```
    public static CodeMsg BIND_ERROR = new CodeMsg(500101,"参数校验异常：%s");

    public CodeMsg fillArgs(Object... args){
        int code = this.code;
        String message = String.format(this.msg, args);
        return new CodeMsg(code,message);
    }
```

- 其中 `String.format()` 能够根据传入的字符串格式，比如"参数校验异常：%s"，其中 `%s` ，能被第二个传入的参数进行替换，从而形成 动态的字符串