

程序设计进阶与实践

设计报告

PB20020586 叶子昂



2023 年 6 月 4 日

目录

1	需求分析	3
2	概要设计	3
3	详细设计介绍与系统设计	3
3.1	持久层	3
3.2	业务层	7
3.2.1	控制层主要功能类	7
3.2.2	业务层异常	9
3.3	控制层	10
3.3.1	控制层主要功能类	10
3.3.2	登录鉴权与操作鉴权	13
3.3.3	控制层异常处理	16
3.4	全局异常处理机制	16
3.5	前端与后端交互封装	19
3.6	第三方类引入	20
4	调试与测试	20
4.1	后端单元测试	20
4.1.1	数据库连接测试	20
4.1.2	管理员后端业务层功能测试	21
4.1.3	学生后端业务层功能测试	21
4.2	前端功能测试	22
5	源程序清单与执行结果	22
6	总结	23

1.

需求分析

2.

概要设计

3.

详细设计介绍与系统设计

我们的程序采用 Spring Boot 框架，JDBC 采用 MyBatis 管理，数据库使用 MySQL，前端使用 vue 构建，总体使用持久层-业务层-控制层-前端的架构，下面分别介绍各个模块的设计。

3.1 持久层

持久层采用 MyBatis 管理，使用 Mapper.xml 文件进行数据库操作，使用 Dao 接口进行调用，针对每一个表，我们都设计了一个 Dao 接口和一个 Mapper.xml(./resources/Mapper) 文件同时有相应的 entity 进行数据封装如下：

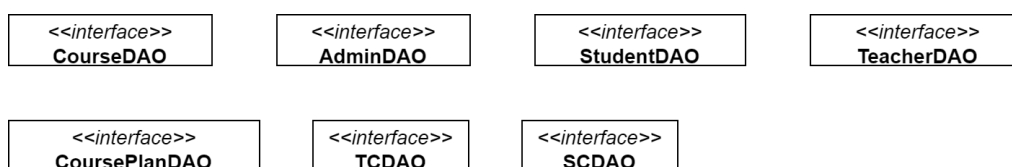


图 1: Dao 接口

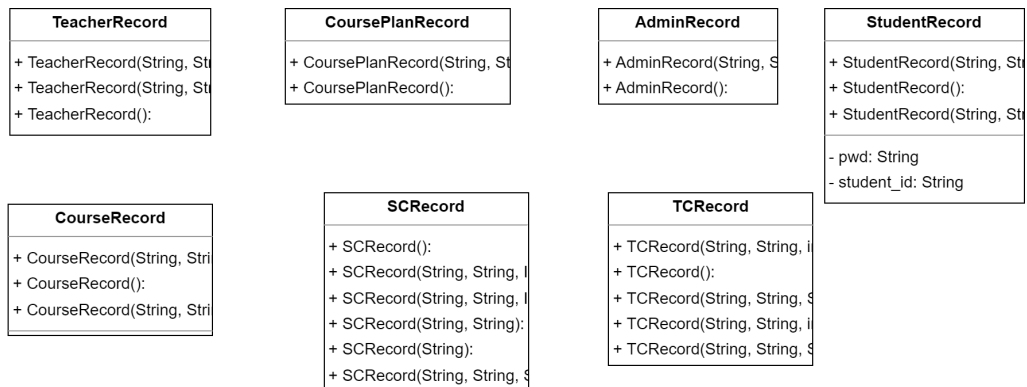


图 2: 数据库实体

下面以 TeacherMapper 为例，介绍持久层接口和 Mapper.xml 文件的设计。TeacherDao 接口中定义了如下的方法（功能见注释）：

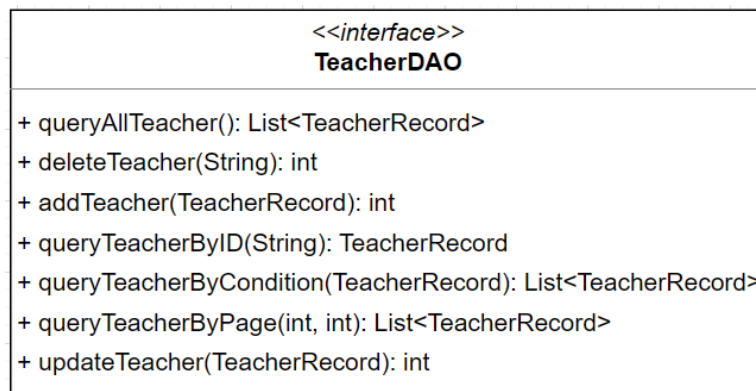


图 3: TeacherDao 接口

```

1 @Mapper //MyBatis 的注解，表示这是一个 Mapper 接口，会自动将其与 Mapper.xml 文件关联
2 @Repository(value = "TeacherDAO")
3 public interface TeacherDAO {
4     // 查询所有教师
5     List<TeacherRecord> queryAllTeacher();
6     // 分页查询教师
7     public List<TeacherRecord> queryTeacherByPage(@Param("page") int page,
8         @Param("size") int size);
9     // 查询指定教师的 ID
10    public TeacherRecord queryTeacherByID(@Param("teacher_id") String ID);
11    // 更新教师信息
12    public int updateTeacher(TeacherRecord record);
13    // 删除教师
14    public int deleteTeacher(String id);
15 }
    
```

```
13 public int deleteTeacher(@Param("teacher_id") String ID);
14 // 添加教师
15 public int addTeacher(TeacherRecord record);
16 // 根据条件查询教师
17 List<TeacherRecord> queryTeacherByCondition(TeacherRecord temp);
18 }
```

Mapper.xml 文件中定义了如下的 SQL 语句与之对应：

```
1 <select id="queryAllTeacher" resultType="com.cy.studentsel.entity.
  TeacherRecord">
2     select * from teacher
3 </select>
4 <select id="queryTeacherByPage" parameterType="int" resultType="
  TeacherRecord">
5     select * from teacher limit #{page}, #{size}
6 </select>
7 <select id="queryTeacherByID" resultType="TeacherRecord" parameterType="
  String">
8     select * from teacher where teacher.teacher_id = #{teacher_id}
9 </select>
10 <select id="queryTeacherByCondition" resultType="TeacherRecord">
11     select teacher_id, teacher_name, sex, age
12     from teacher
13     where
14 // 如果 teacher_id 不为空，动态拼接 SQL 语句
15     <if test="teacher_id != null and teacher_id != ''">
16         teacher_id like concat('%', #{teacher_id}, '%') and // 模糊查询
17     </if>
18     <if test="teacher_name != null and teacher_name != ''">
19         teacher_name like concat('%', #{teacher_name}, '%') and
20     </if>
21     <if test="sex != null and sex != ''">
22         sex = #{sex} and
23     </if>
24     <if test="age != null and age != -1">
25         age = #{age} and
26     </if>
```

```

27         true
28     </select>
29     <update id="updateTeacher" parameterType="TeacherRecord">
30         update teacher
31         <set>
32             // 如果 teacher_name 不为空, 动态拼接 SQL 语句
33             <if test="teacher_name != null and teacher_name != ''">
34                 teacher_name = #{teacher_name},
35             </if>
36             <if test="sex != null and sex != ''">
37                 sex = #{sex},
38             </if>
39             <if test="age != null and age != -1">
40                 age = #{age},
41             </if>
42             <if test="pwd != null and pwd != ''">
43                 pwd = #{pwd},
44             </if>
45             teacher_id = #{teacher_id}
46         </set>
47         <where>
48             teacher_id = #{teacher_id}
49         </where>
50     </update>
51     <delete id="deleteTeacher" parameterType="String">
52         call deleteTeacher(#{teacher_id})
53     </delete>
54     <insert id="addTeacher" parameterType="TeacherRecord">
55         insert into teacher values (#{teacher_id}, #{teacher_name}, #{sex}, #{
age}, #{pwd})
56     </insert>

```

其中 deleteTeacher 为保证数据一致性设计为一个存储过程（因为删去教师需要同时删去教师的授课信息）：

```

1  // 删除教师记录事务
2  drop procedure if exists deleteTeacher;
3  delimiter //

```

```
4 create procedure deleteTeacher(in teacher_id varchar(20))
5 begin
6     delete from TC where tc.teacher_id=teacher_id;
7     delete from Teacher where teacher.teacher_id=teacher_id;
8 end //
9 delimiter ;
```

对于删去学生和课程的操作也是类似的。

3.2 业务层

3.2.1 控制层主要功能类

业务层主要负责对持久层的调用以及包装来定义业务逻辑以供控制层调用，是的控制层不需要关心具体的数据库实现使得将控制层与持久层解耦合。我们定义如下形式的业务层接口：

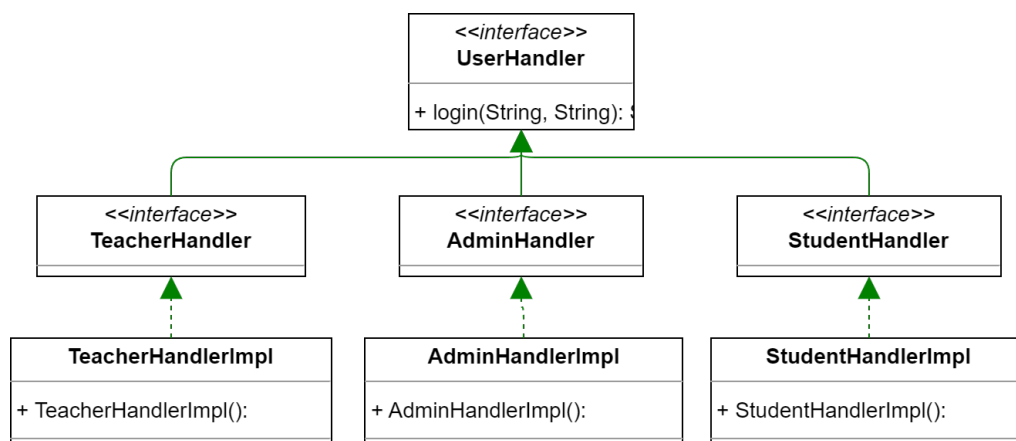


图 4: Handler 接口

下面以 TeacherHandler 为例，介绍业务层的设计。TeacherHandler 接口实现了 UserHandler 接口 (其中的 login 是每个 Handler 都需要提供的)，TeacherHandler 接口中定义了如下的方法：

<<interface>> TeacherHandler
+ update(TeacherRecord): boolean + getTeacher(String): TeacherRecord + queryTCByCondition(TCRecord): List<TCRecord> + querySCByCondition(SCRecord): List<SCRecord> + updateTC(TCRecord): void + queryTeacherByTeacherID(String): TeacherRecord + updateSC(SCRecord): void

图 5: TeacherHandler 接口

TeacherHandlerImpl 类中实现了 TeacherHandler 接口中的方法如下:

```
1  @Service // 标注为 Spring 的 Service 层
2  public class TeacherHandlerImpl implements TeacherHandler {
3      @Resource // 自动注入 TeacherDAO 对象
4      private TeacherDAO teacherDao;
5      @Resource // 自动注入 SCDAO 对象
6      private SCDAO scDao;
7      @Resource // 自动注入 TCDAO 对象
8      private TCDAO tcDao;
9
10     @Override
11     public String login(String ID, String pwd) {
12         TeacherRecord teacherRecord = teacherDao.queryTeacherByID(ID);
13         if (teacherRecord == null) {
14             throw new UserNameNotFoundException("教师工号不存在");
15         }
16         if (!Objects.equals(teacherRecord.getPwd(), pwd)){
17             throw new PasswordNoMatchException("密码错误");
18         }
19         return "登录成功";
20     }
21
22     // 查询教师信息
23     @Override
```



```
24     public List<TCRecord> queryTCByCondition(TCRecord record) {
25         return tcDao.queryTCByCondition(record);
26     }
27     // 通过 ID 查询教师信息
28     @Override
29     public TeacherRecord getTeacher(String ID) {
30         return teacherDao.queryTeacherByID(ID);
31     }
32     // 更新教师信息
33     @Override
34     public boolean update(TeacherRecord teacherRecord) {
35         teacherDao.updateTeacher(teacherRecord);
36         return true;
37     }
38     // 查询选本教师课的所有选课记录
39     @Override
40     public List<SCRecord> querySCByCondition(SCRecord record) {
41         return scDao.querySCByCondition(record);
42     }
43     // 更新授课信息
44     @Override
45     public void updateTC(TCRecord record) {
46         tcDao.updateTC(record);
47     }
48     // 更新选课信息（评分）
49     @Override
50     public void updateSC(SCRecord record) {
51         scDao.updateSC(record);
52     }
53
54 }
```

其中 TeacherHandlerImpl 类中的 TeacherDAO、SCDAO、TCDAO 为持久层接口。

3.2.2 业务层异常

业务层执行部分持久层操作前，需要对传入的参数进行检查，如果参数不合法，需要抛出异常。业务层的异常自定义为 RuntimeException 的子类，如下：

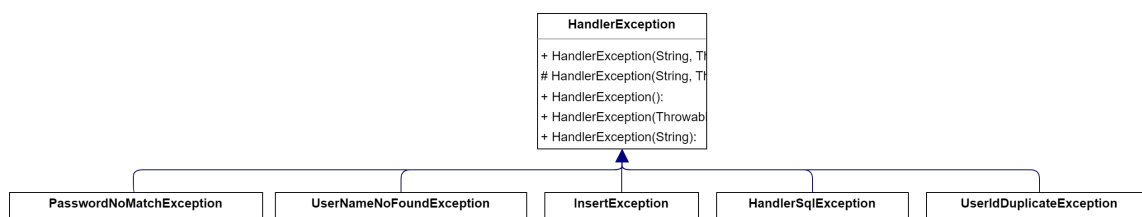


图 6: 业务层异常

3.3 控制层

3.3.1 控制层主要功能类

控制层主要负责接收用户的请求，直接面向前端提供方法，整合业务层的方法。我们定义如下控制层类：

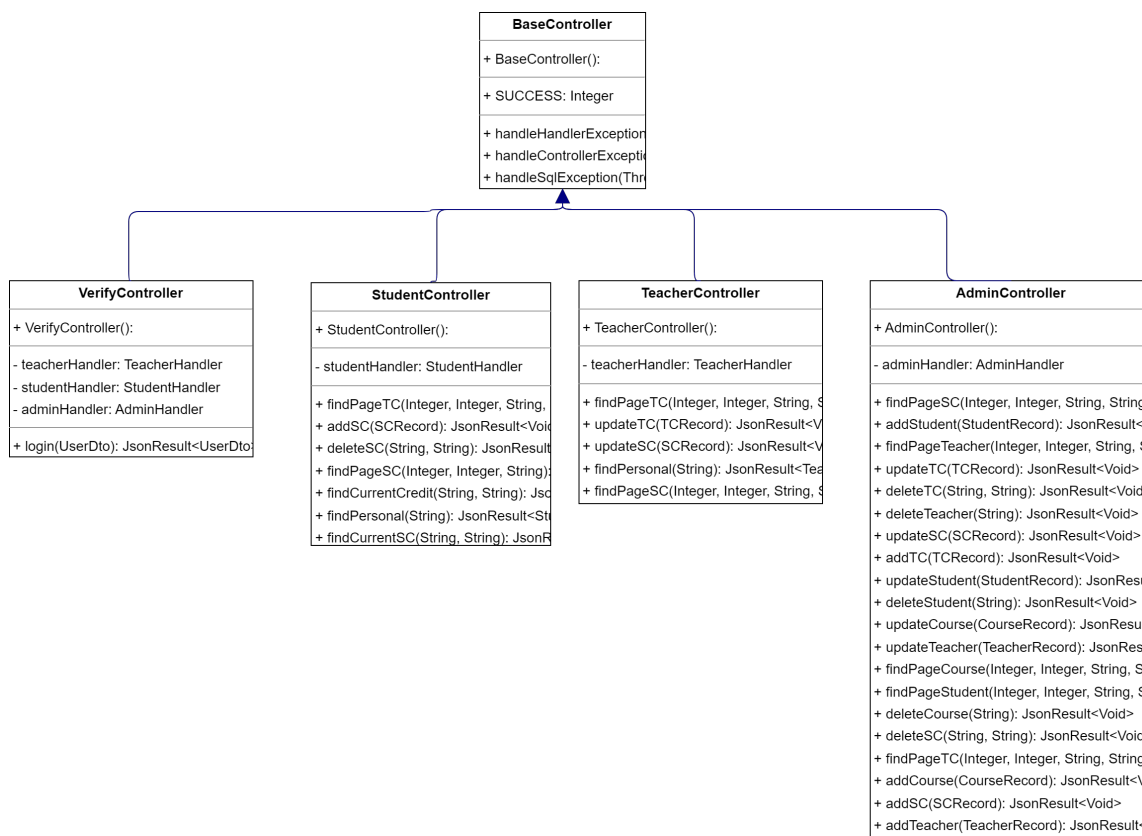


图 7: 控制层类

其中：

- **BaseController** 类：提供所有控制层类的基类，提供统一的全局异常处理机制

- **VerifyController** 类：提供针对所有用户的统一登录验证
- **AdminController** 类：提供管理员前端界面所需的所有方法
- **TeacherController** 类：提供教师前端界面所需的所有方法
- **StudentController** 类：提供学生前端界面所需的所有方法

以 **TeacherController** 类为例，介绍控制层的设计。**TeacherController** 类中定义了如下方法：

TeacherController
+ TeacherController():
- teacherHandler: TeacherHandler
+ findPageTC(Integer, Integer, String, String, String, String, String, int, String): JsonResult<pageInfo<TCRecord>>
+ updateTC(TCRecord): JsonResult<Void>
+ updateSC(SCRecord): JsonResult<Void>
+ findPersonal(String): JsonResult<TeacherRecord>
+ findPageSC(Integer, Integer, String, String, String, String, String, String, int, String): JsonResult<pageInfo<SCRecord>>

图 8: TeacherController 类

- **findPageTC** 方法：处理教师分页查询授课信息的请求
- **updateTC** 方法：处理教师更新授课信息的请求
- **updateSC** 方法：处理教师更新学生选课信息（评分）的请求
- **findPersonal** 方法：处理登录时查询个人信息
- **findPageSC** 方法：处理教师分页查询选自己课学生信息的请求

```

1  @RestController// 标注为 Spring 的 Controller 层，自动将返回值转换为 JSON 对象
2  @RequestMapping("api/teacher")
3  public class TeacherController extends BaseController{
4      @Resource
5      private TeacherHandler teacherHandler;
6
7      /**
8       * get teacher personal information
9       */

```

```
10 @GetMapping("/personal")
11 @ResponseBody
12 public JsonResult<TeacherRecord> findPersonal(@RequestParam String teacher_id
    ) {
13     JsonResult<TeacherRecord> jsonResult = new JsonResult<>();
14     TeacherRecord record = teacherHandler.queryTeacherByTeacherID(teacher_id);
15     jsonResult.setData(record);
16     jsonResult.setStatus(SUCCESS);
17     jsonResult.setMsg("查询成功");
18     return jsonResult;
19 }
20
21 /**
22  * All the student who choose the course in page range
23  */
24 @GetMapping("/page/sc")
25 @ResponseBody
26 public JsonResult<pageInfo<SCRecord>> findPageSC(@RequestParam Integer page,
27     @RequestParam Integer pageSize,
28     @RequestParam String student_id,
29     @RequestParam String student_name,
30     @RequestParam String course_id,
31     @RequestParam String course_name,
32     @RequestParam String teacher_id,
33     @RequestParam String teacher_name,
34     @RequestParam String type,
35     @RequestParam int credit,
36     @RequestParam String place) {
37     JsonResult<pageInfo<SCRecord>> jsonResult = new JsonResult<>();
38     pageInfo<SCRecord> pageInfo = new pageInfo<>();
39     SCRecord record = new SCRecord(student_id, course_id, teacher_id,
40         student_name, course_name, teacher_name, type, credit, place);
41     List<SCRecord> list = teacherHandler.querySCByCondition(record);
42     int pageStart = (page - 1) * pageSize;
43     int limit = pageSize;
44     pageInfo.list = list.subList(pageStart, min(pageStart + limit, list.size())
    );
```

```

45     pageInfo.total = list.size();
46     jsonResult.setData(pageInfo);
47     jsonResult.setStatus(SUCCESS);
48     jsonResult.setMsg("查询成功");
49     return jsonResult;
50 }
51 .....
52 部分方法实现与上面方法类似且相对较长，不再赘述见源码
53
54 }

```

3.3.2 登录鉴权与操作鉴权

登录鉴权是指用户在访问系统资源时，系统对用户进行身份验证的过程。本系统中，用户登录后，系统会为用户生成一个 token (限制有效时间为 15 分钟) 返回给前端，前端将得到的 token 存储在本地，每次请求时将 token 放在请求头中，后端会对 token 进行不同权限的操作进行鉴权，如果 token 过期或者 token 不合法，后端会返回错误信息，前端会跳转到登录界面。登录鉴权的实现通过如下类：

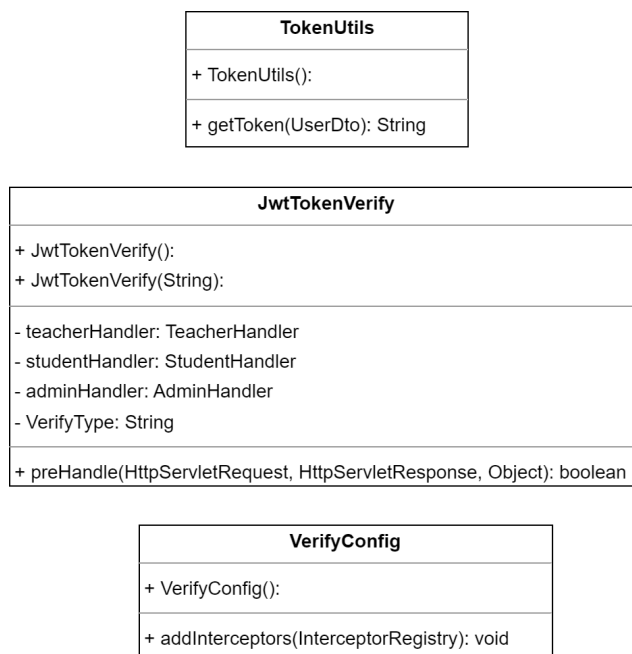


图 9: 登录鉴权

其中：

- TokenUtil 类：提供依据用户名，用户类型，用户签名生成 token 的方法（使用 HMAC256 算法）
- JwtTokenVerify 类：提供基于不同用户类型的 token 验证方法
- VerifyConfig 类：提供 token 验证的拦截器配置

依次实现如下：

TokenUtil.java

```
1  /**
2  * 生成 Token
3  */
4  public static String getToken(UserDto userDto) {
5      String token="";
6      token= JWT.create().withClaim("username", userDto.getUsername())// 保存用户
      信息
7      .withClaim("userType", userDto.getUserType())// 保存用户信息
8      .withExpiresAt(DateUtil.offsetMinute(new Date(), 15))// 设置过期时间为 15
      分钟
9      .sign(Algorithm.HMAC256(userDto.getPassword()));// 以 password 作为 token
      的密钥
10     return token;
11 }
```

JwtTokenVerify.java

```
1  @Override
2  public boolean preHandle(HttpServletRequest request, HttpServletResponse
      response, Object handler) {
3      String token = request.getHeader("token");
4      if (!(handler instanceof HandlerMethod))
5          return true;
6      if (token == null || token.equals("")) {
7          throw new NoTokenException("未登录");
8      }
9
10     String username;
11     String userType;
12     String password;
13     try {
```

```
14         username = JWT.decode(token).getClaim("username").asString();
15         userType = JWT.decode(token).getClaim("userType").asString();
16         // if the token is not for the requested user type, then throw
exception
17         if (!userType.equals(VerifyType)) {
18             throw new VerifyTokenFailException("token 验证失败");
19         }
20         // get the password from database
21         switch (userType) {
22             case "admin" -> {
23                 AdminRecord adminRecord = adminHandler.getAdmin(username);
24                 password = adminRecord.getPwd();
25             }
26             case "teacher" -> {
27                 TeacherRecord teacherRecord = teacherHandler.getTeacher(
username);
28                 password = teacherRecord.getPwd();
29             }
30             case "student" -> {
31                 StudentRecord studentRecord = studentHandler.getStudent(
username);
32                 password = studentRecord.getPwd();
33             }
34             default -> throw new VerifyTokenFailException("token 验证失败");
35         }
36         // verify the token
37         JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256(password)).
build();
38         jwtVerifier.verify(token);
39     } catch (Exception e) {
40         throw new VerifyTokenFailException("token 验证失败");
41     }
42     return true;
43
```

VerifyConfig.java

```
1  /**
```

```
2  * 拦截请求, 验证 token, 登录请求不必验证
3  * @param registry
4  */
5  @Override
6  public void addInterceptors(InterceptorRegistry registry) {
7      // if /api/admin is requested, then verify the token with admin role
8      registry.addInterceptor(jwtTokenVerifyAdmin())
9          .addPathPatterns("/api/admin/**")
10         .excludePathPatterns("/api/verify");
11     // if /api/teacher is requested, then verify the token with teacher role
12     registry.addInterceptor(jwtTokenVerifyTeacher())
13         .addPathPatterns("/api/teacher/**")
14         .excludePathPatterns("/api/verify");
15     // if /api/student is requested, then verify the token with student role
16     registry.addInterceptor(jwtTokenVerifyStudent())
17         .addPathPatterns("/api/student/**")
18         .excludePathPatterns("/api/verify");
19 }
```

3.3.3 控制层异常处理

控制层在请求业务层操作前, 需要对传入的参数进行一定的检查 (包括权限检查, 登录请求类型检查), 当登录参数不合法以及请求的权限不足时, 需要抛出异常。控制层的异常自定义为 `RuntimeException` 的子类, 如下:

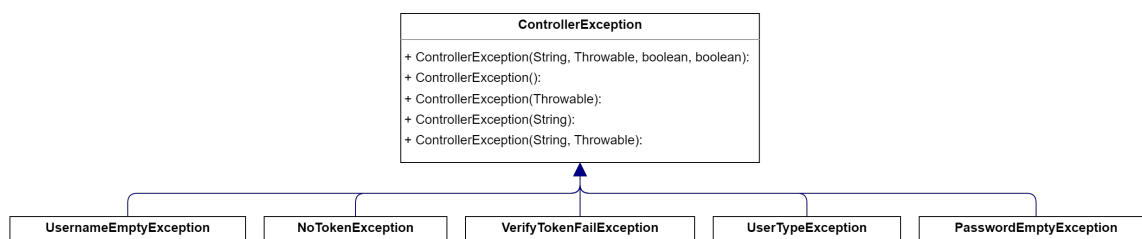


图 10: 控制层异常

3.4 全局异常处理机制

在项目中我们在业务层和控制层抛出自定义异常, 数据库在操作时也可能可能由于服务器宕机等各种原因抛出异常, 在将问题传递给前端前需要将异常信息进行处理避免造成

程序崩溃。我们选择在控制层统一捕获各类异常进行处理。异常的处理在 `BaseController` 类中实现，所有的控制层 `Controller` 都继承自 `BaseController`，拥有统一的异常处理机制。

业务层异常处理

```
1  /**
2   * Handle the exception thrown by the service layer
3   * @param e
4   * @return
5   */
6  @ExceptionHandler({HandlerException.class}) // Java 注释，自动捕获所有业务层异常
7  @ResponseBody
8  public JsonResult<Void> handleHandlerException(Throwable e) {
9      JsonResult<Void> jsonResult = new JsonResult<>();
10     switch (e.getClass().getSimpleName()) {
11         case "UserNameNoFoundException" -> {
12             jsonResult.setStatus(1000);
13             jsonResult.setMsg(e.getMessage());
14         }
15         case "PasswordNoMatchException" -> {
16             jsonResult.setStatus(1001);
17             jsonResult.setMsg(e.getMessage());
18         }
19         case "UserIdDuplicateException" -> {
20             jsonResult.setStatus(1002);
21             jsonResult.setMsg(e.getMessage());
22         }
23         case "HandlerSqlException" -> {
24             jsonResult.setStatus(5000);
25             jsonResult.setMsg(e.getMessage());
26         }
27         default -> {
28             jsonResult.setStatus(5001);
29             jsonResult.setMsg("未知错误");
30         }
31     }
32     return jsonResult;
33 }
```

控制层异常处理

```
1  /**
2   * Handle the exception thrown by the controller layer
3   * @param e
4   * @return
5   */
6  @ExceptionHandler(ControllerException.class) //Java 注释，自动捕获所有控制层异常
7  @ResponseBody
8  public JsonResult<Void> handleControllerException(Throwable e) {
9      JsonResult<Void> jsonResult = new JsonResult<>();
10     switch (e.getClass().getSimpleName()) {
11         case "UsernameEmptyException" -> {
12             jsonResult.setStatus(2000);
13             jsonResult.setMsg(e.getMessage());
14         }
15         case "PasswordEmptyException" -> {
16             jsonResult.setStatus(2001);
17             jsonResult.setMsg(e.getMessage());
18         }
19         case "UserTypeEmptyException" -> {
20             jsonResult.setStatus(2002);
21             jsonResult.setMsg(e.getMessage());
22         }
23         case "NoTokenException" -> {
24             jsonResult.setStatus(401);
25             jsonResult.setMsg("未登录");
26         }
27         case "VerifyTokenFailException" -> {
28             jsonResult.setStatus(401);
29             jsonResult.setMsg("权限不足，重新登录");
30         }
31         default -> {
32             jsonResult.setStatus(6004);
33             jsonResult.setMsg("控制层未知错误");
34         }
35     }
36     return jsonResult;
```

```
37 }
```

持久层异常处理

```
1  /**
2   * Handle the exception thrown by the mybatis
3   * @param e
4   * @return
5   */
6  @ExceptionHandler(SQLException.class) // Java 注释, 自动捕获所有数据库操作异常
7  @ResponseBody
8  public JsonResult<Void> handleSQLException(Throwable e) {
9      JsonResult<Void> jsonResult = new JsonResult<>();
10     jsonResult.setStatus(6000);
11     jsonResult.setMsg("Sql 错误");
12     return jsonResult;
13 }
```

3.5 前端与后端交互封装

前端与后端交互使用 Ajax 技术（进一步使用 axios 进行封装），我们约定使用 JSON 格式传递数据，后端通过 Spring Boot 框架提供的 `@ResponseBody` `@RequestBody` 注解分别将传出传入数据转换为 JSON 格式。包装的格式如下：

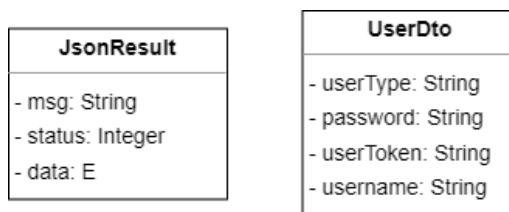


图 11: JSON 封装

其中：

- JsonResult 是一个泛型类，用于封装后端传递给前端的数据，包括状态码和信息
- UserDto 由于封装登录时前端传递的数据，包括用户名，密码，用户类型
- 在持久层提到的数据库实体类用于封装从数据库中查询出的数据以及封装前端传递的数据

3.6 第三方类引入

为支持项目一些特殊功能，我们引入了一些第三方类库，它们分别是：

- **Spring Boot 框架**：用于管理项目，提供控制反转特性的容器，托管项目的控制层，业务层，持久层，以及其他组件的注入。
- **MyBatis 框架**：用于管理数据库，提供持久层接口与 mySql 数据库的映射。
- **Jwt**：用于生成 token 以及验证 token 的有效性。
- **hutool**：提供方便有效的时间管理机制（设置 token 的有效时间）。

4.

调试与测试

项目的调试与测试主要分为两个部分，前端的调试与测试，后端的调试与测试。前端的调试通过 Postman 进行，后端的调试基于 Junit 进行单元测试。下面举例介绍部分功能的单元测试。

4.1 后端单元测试

4.1.1 数据库连接测试

```
1  @SpringBootTest
2  class StudentSelApplicationTests {
3      @Autowired
4      private DataSource DataS;
5
6      @Test
7      void getConnection() throws SQLException {
8          System.out.println(DataS.getConnection());
9      }
10 }
```

4.1.2 管理员后端业务层功能测试

```
1  @SpringBootTest // 注释, 自动创建 Spring Boot 容器
2  public class TestAdminHandlerImpl {
3      @Autowired // 注释, 自动注入 AdminHandlerImpl 对象
4      private AdminHandler adminHandler;
5
6      @Test
7      public void testQueryStudentByCondition() {
8          StudentRecord record = new StudentRecord("", "", "", 20, "", "");
9          System.out.println(adminHandler.queryStudentByCondition(record));
10     }
11
12     @Test
13     public void testUpdateStudent() {
14         StudentRecord record = new StudentRecord("201600001", "", "女", -1, "
15         123", "");
16         adminHandler.updateStudent(record);
17     }
18 }
```

4.1.3 学生后端业务层功能测试

```
1  @SpringBootTest\\ 注释, 自动创建 Spring Boot 容器
2  public class TestStudentHandlerImpl {
3      @Autowired\\ 注释, 自动注入 StudentHandlerImpl 对象
4      private StudentHandler studentHandler;
5
6      @Test
7      public void testlogin(){System.out.println(studentHandler.login("
8      PB420000001","123456"));}
9
10     @Test
11     public void testgetstudent(){System.out.println(studentHandler.getStudent
12     ("PB420000001"));}
13
14     @Test
15     public void testupdate(){
```

```
11         StudentRecord a=new StudentRecord("PB420000001","11f","男",21,"cs","  
12         123456");  
13         System.out.println(studentHandler.update(a));  
14     }  
15  
16 }
```

4.2 前端功能测试

前端主要通过浏览器开发者模式，Postman 进行测试，主要检查点击交互时前端是否向后端发送了预期的请求，以及后端是否返回了预期的数据。下面以登录功能为例进行测试。

5.

源程序清单与执行结果

本项目的源代码托管在华为云 CodeArt 上，源代码遵循如下目录结构：



未标明的文件夹一般为项目自动生成框架，标出的文件夹中未注明的文件其作用与文件名相同，不再赘述。

执行结果见 report 文件夹下的 result.mp4.

6.

总结

这次课程设计，我们综合运用了课程提到的工程化开发方法，在一个具有一定规模的

项目开发中我们深刻体会到了工程项目与个人项目的区别以及规范化开发的重要性。扭转了我们平时一般先写代码别写边想的开发习惯，在开发前我们先进行了详细的需求分析，然后进行了详细的规划设计，最后才根据规划进行编码。

这次课程设计由我们整个小组一起完成，虽然大多数成员没有接触过前后端分离的开发，以及项目用到的很多技术（包括 **Spring Boot** 框架，**Mybatis** 接口，**Vue** 前端脚手架，**Jwt** 鉴权），但是我们通过查阅资料、跟着晚上的视频根据预先的设计需求学习相应的技术，明确分工共同协作，遇到问题积极询问，互相讨论不断迭代完善需求，最终完成了这个项目。

因为能力有限，这个项目还有很多不足之处，比如前端页面的美观程度，后端代码的优化程度，以及更多个性化需求的实现以及代码的健壮性。但是我们会在以后的学习中不断完善这个项目，同时也会在以后的学习中不断提高自己的能力积累高质量代码开发经验，为以后的工作打下坚实的基础。