**《Web应用开发实战课程设计》报告**

题目——人员开发系统

1. **功能介绍**

该人员管理系统采用了vue-element技术搭建了前端框架，后端采用了springboot实现了用户登录操作、用户登陆状态的检查、文件上传、人员的增删改查、分页查询以及部门的增删改查。

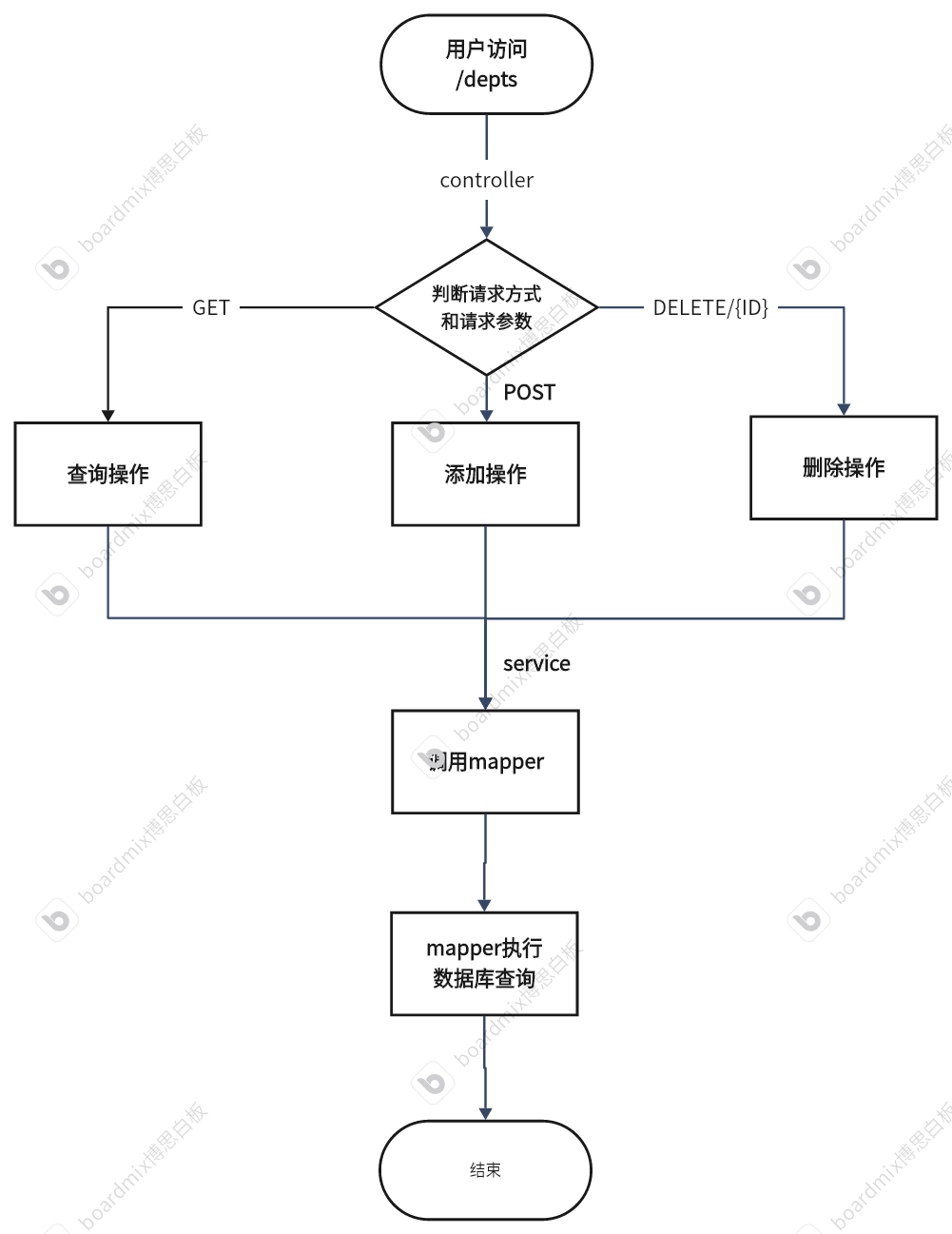
1. **开发技术**
2. vue-element
3. springboot
4. **项目设计**
5. depts流程图

图3.1

1. emp流程图

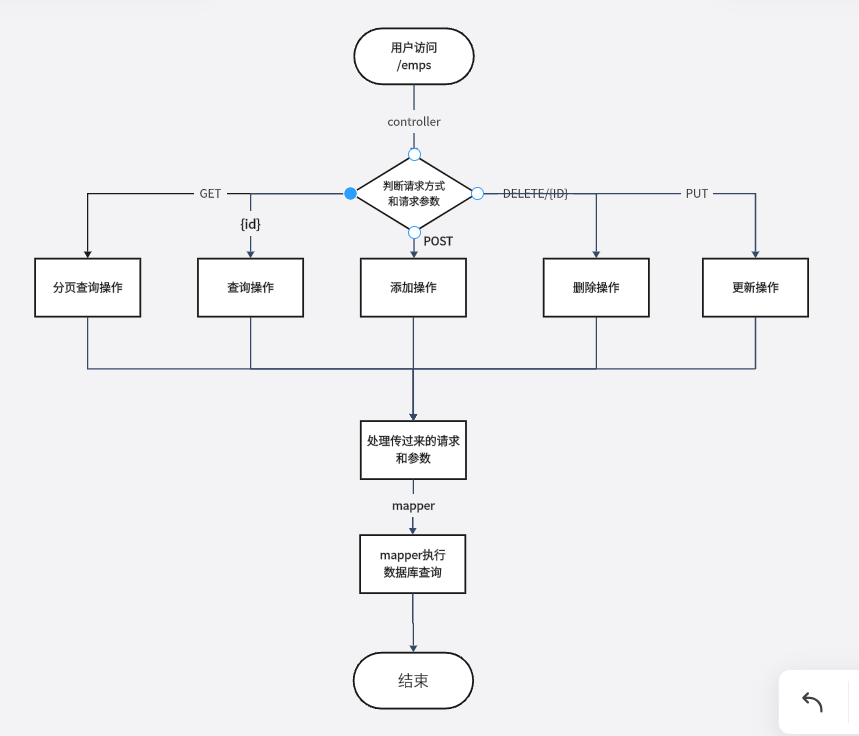


图3.2

1. 用户登录

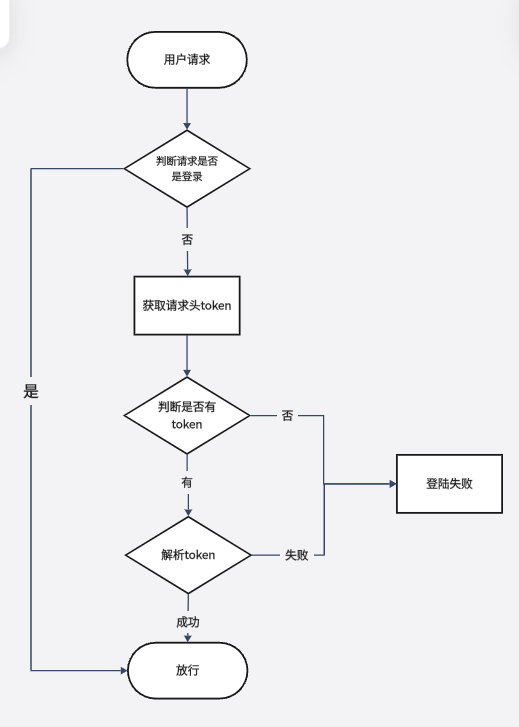


图3.3

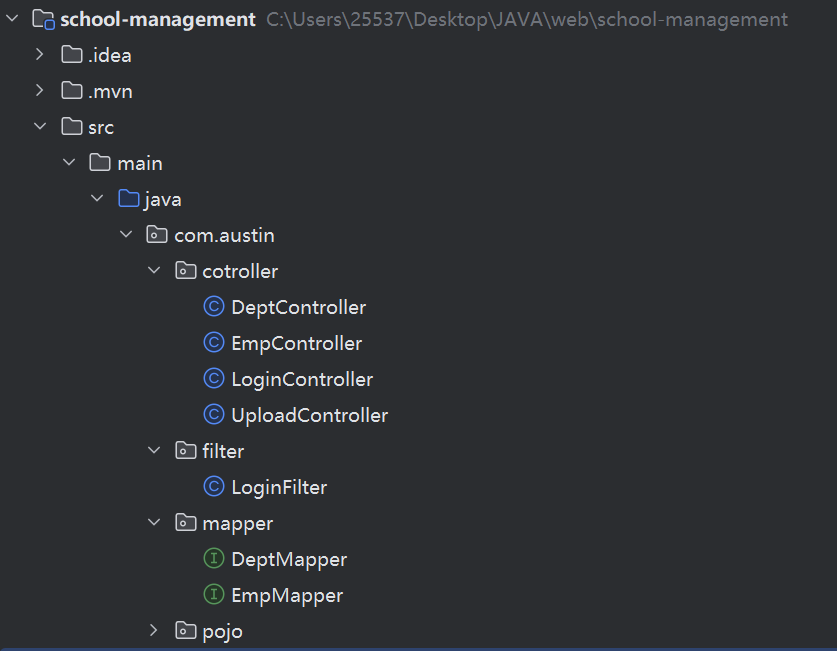
1. **项目文件结构**

图4.1

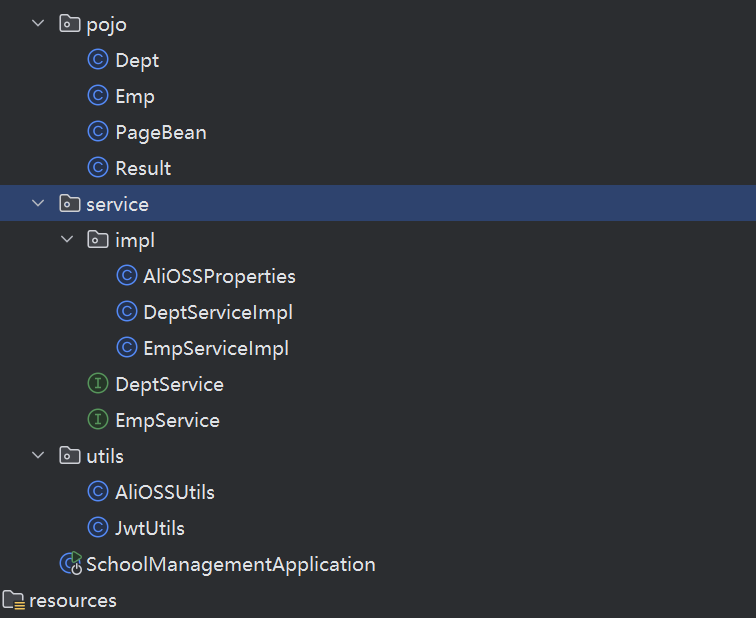


图4.2

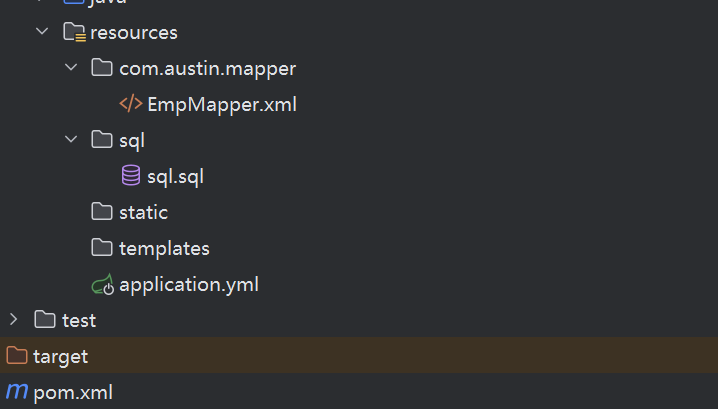


图4.3

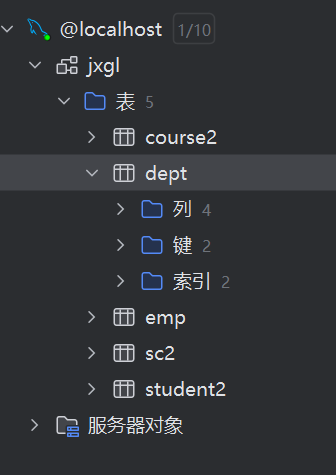
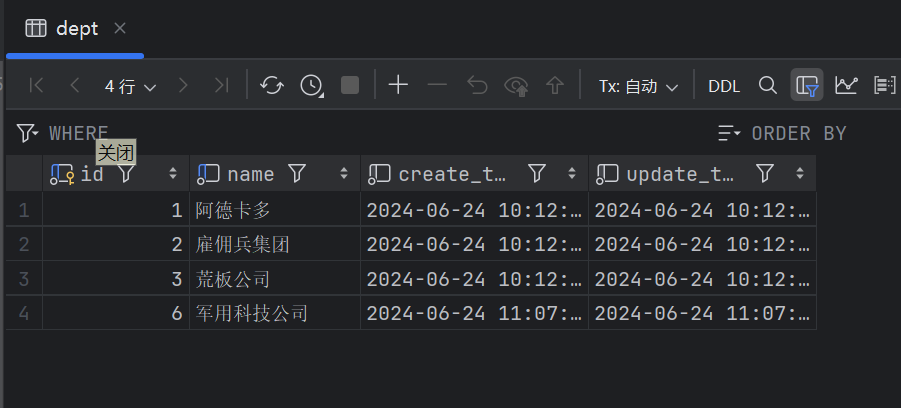


图4.4数据库结构

图4.5 dept表结构

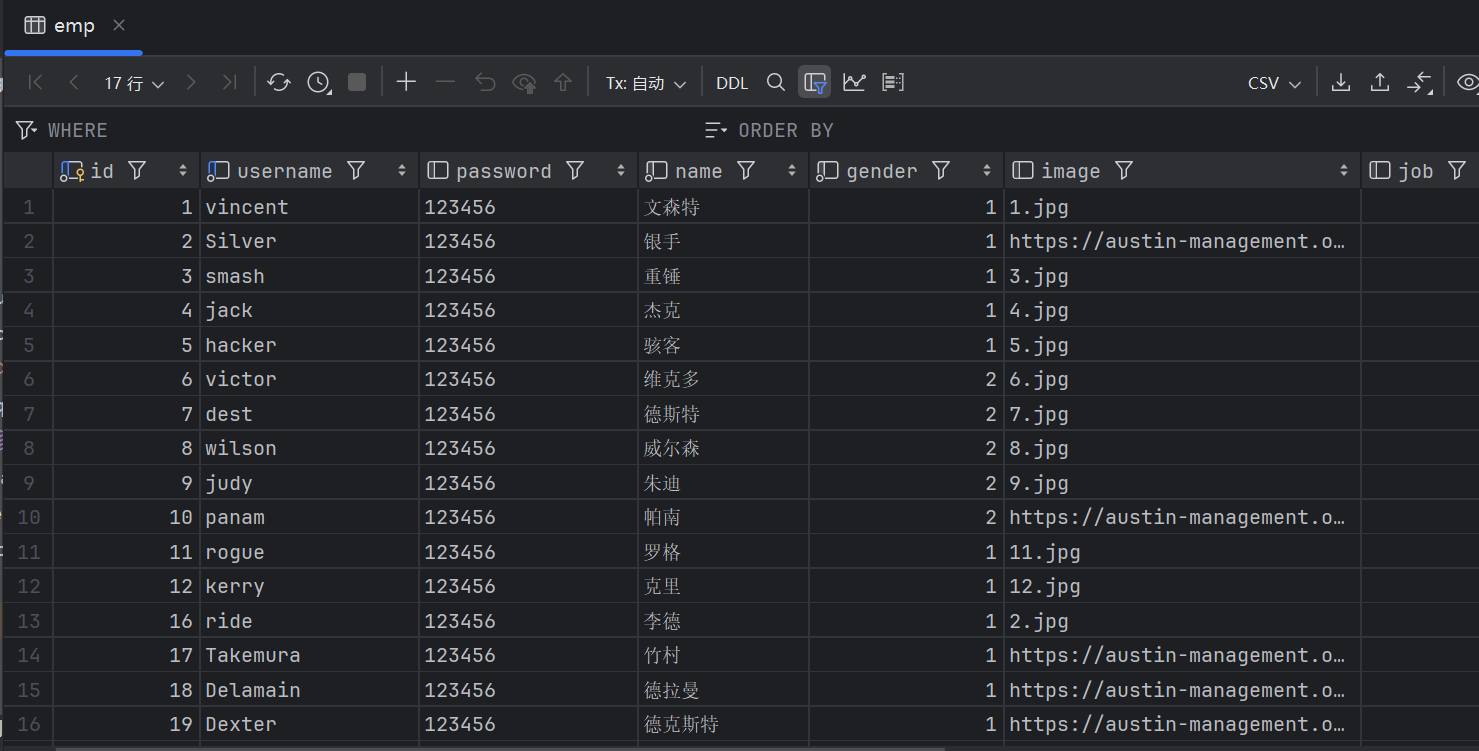


图4.6 emp表结构

1. **设计工作**
2. 设计了数据库的数据和表结构
3. 设计了后端depts和emp处理访问请求
4. 设计了login登录处理操作和登陆状态请求操作
5. 设计了文件上传处理机制
6. **核心代码**
7. **Dept**

**Controller**

@RestController

@RequestMapping("/depts")

public class DeptController {

private static final Logger log = LoggerFactory.getLogger(DeptController.class);

@Autowired

private DeptService deptService;

@GetMapping

public Result getDepts() {

log.info("select all depts");

//调用service层

List<Dept> deptList=deptService.list();

return Result.success(deptList);

}

@DeleteMapping("/{id}")

public Result deleteDept(@PathVariable int id) {

log.info("delete dept by id: {}", id);

deptService.delete(id);

return Result.success();

}

@PostMapping

public Result addDept(@RequestBody Dept dept) {

log.info("add dept : {}", dept);

deptService.adddept(dept);

return Result.success();

}

@PutMapping

public Result updateDept(@RequestBody Dept dept) {

log.info("update dept : {}", dept);

deptService.update(dept);

return Result.success();

}

}

**解释：**

1. getDepts()方法用于获取所有部门列表。它通过调用DeptService层的list()方法来获取数据,并将结果封装到Result对象中返回。
2. deleteDept()方法用于删除指定ID的部门。它通过调用DeptService层的delete()方法来删除数据,并将删除结果封装到Result对象中返回。
3. addDept()方法用于新增一个部门。它通过调用DeptService层的adddept()方法来新增数据,并将新增结果封装到Result对象中返回。

4.update()方法用于修改部门它通过调用DeptService层的update ()方法来新增数据,并将新增结果封装到Result对象中返回。

**Service**

@Slf4j

@Service

public class DeptServiceImpl implements DeptService {

@Autowired

private DeptMapper deptMapper;

@Override

public List<Dept> list() {

//调用mapper层

return deptMapper.list();

}

@Override

public void delete(int id) {

deptMapper.delete(id);

}

@Override

public void adddept(Dept dept) {

dept.setCreateTime(LocalDateTime.now());

dept.setUpdateTime(LocalDateTime.now());

deptMapper.adddept(dept);

}

@Override

public void update(Dept dept) {

dept.setUpdateTime(LocalDateTime.now());

deptMapper.update(dept);

}

}

**解释：**

1. list()方法用于获取所有部门列表。它通过调用DeptMapper层的list()方法来获取数据。
2. delete(int id)方法用于删除指定ID的部门。它通过调用DeptMapper层的delete()方法来删除数据。
3. adddept(Dept dept)方法用于新增一个部门。它首先设置部门的创建时间和更新时间,然后调用DeptMapper层的adddept()方法来新增数据。
4. update方法用于新增一个部门。它首先设置部门的创建时间和更新时间,然后调用DeptMapper层的update()方法来新增数据。

**Mapper**

@Mapper

public interface DeptMapper {

@Select("select id, name, create\_time, update\_time from dept")

List<Dept> list();

@Delete("delete from dept where id=#{id}")

void delete(int id);

@Insert("insert into dept (name, create\_time, update\_time) values (#{name},#{createTime},#{updateTime})")

void adddept(Dept dept);

@Update("update dept set name=#{name} where id=#{id}")

void update(Dept dept);

}

**解释：**

1. list()方法使用@Select注解定义了一条SQL语句,用于查询dept表中的所有记录,并返回Dept对象列表。
2. delete(int id)方法使用@Delete注解定义了一条SQL语句,用于根据给定的id删除dept表中的记录。
3. adddept(Dept dept)方法使用@Insert注解定义了一条SQL语句,用于向dept表中插入新的记录。insert语句中引用了Dept对象的name、createTime和updateTime属性作为参数。
4. update方法使用@update注解定义了一条SQL语句,用于向dept表中更改记录。。

**截图：**

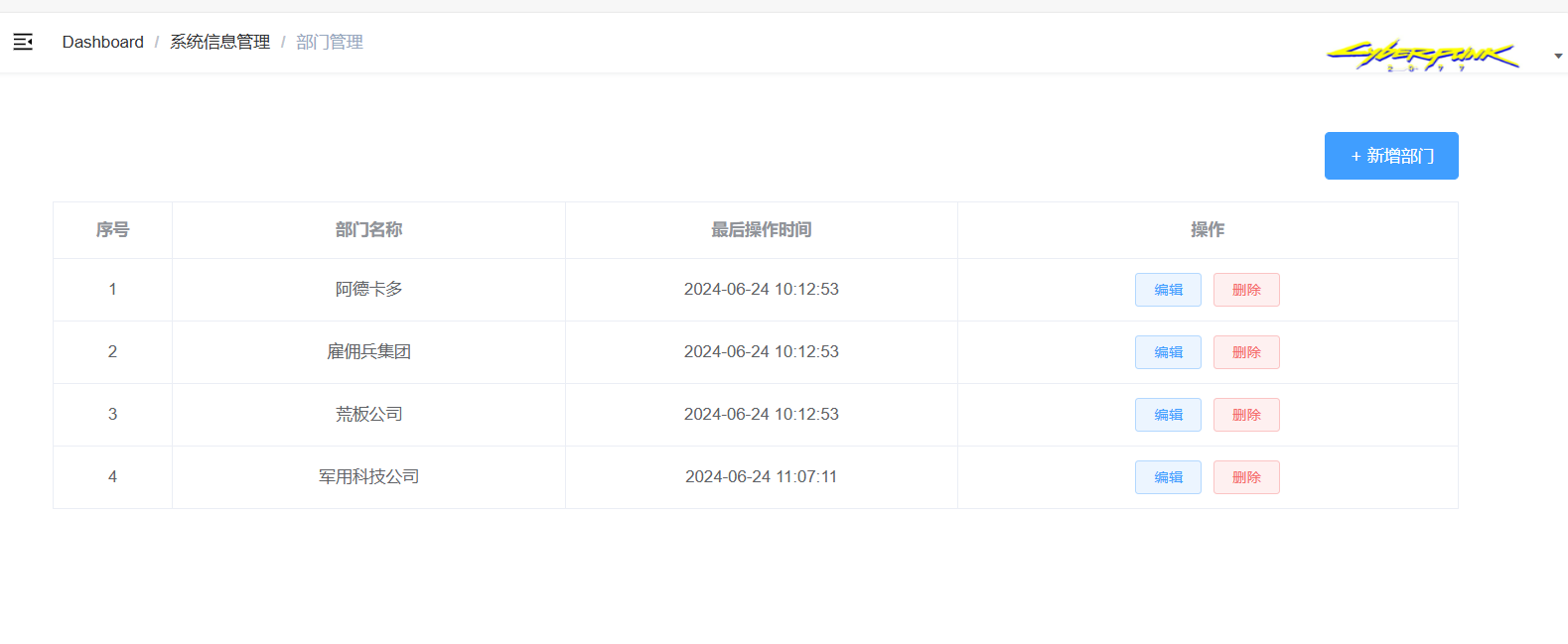
****

图 5.1

****

图5.2 删除

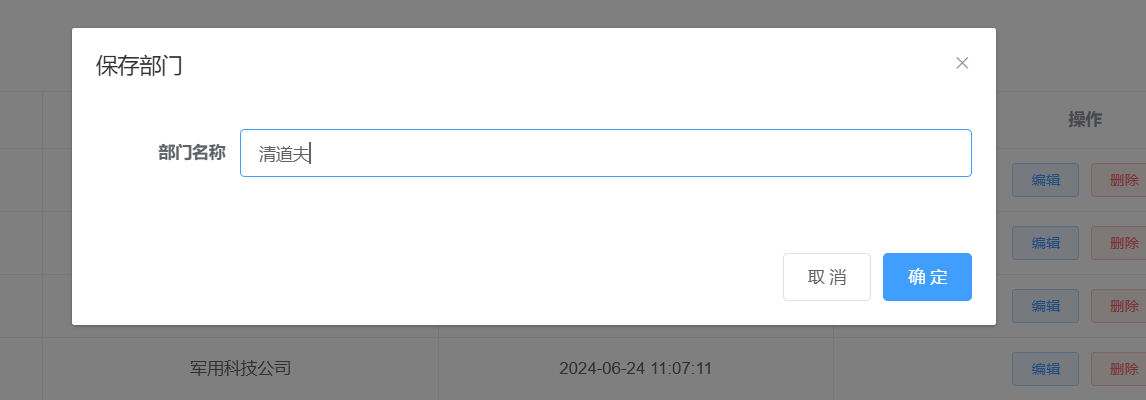
图5.3 删除

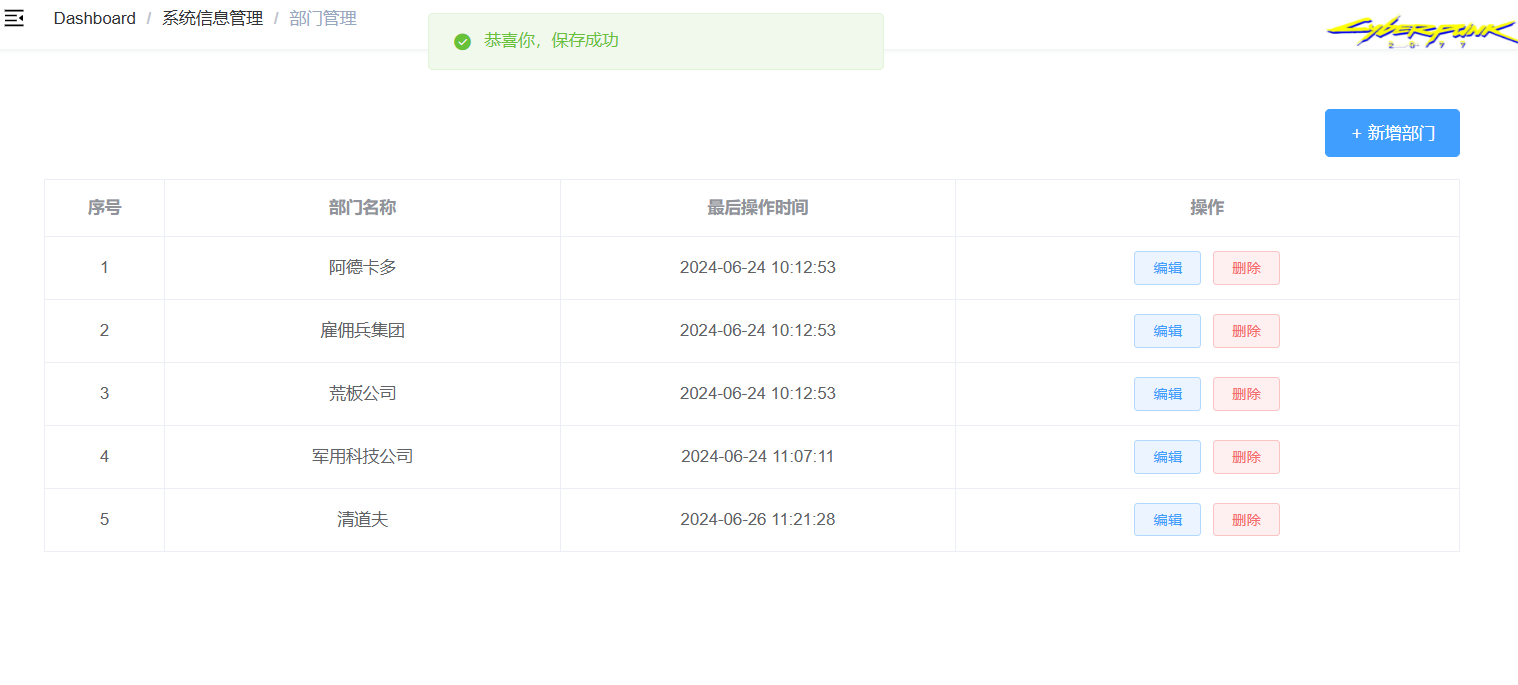
图5.3

图5.4

1. **emp**

**controller**

@RequestMapping("/emps")

public class EmpController {

@Autowired

private EmpService empService;

//条件分页查询

@GetMapping

public Result page(@RequestParam(defaultValue = "1") Integer page,

@RequestParam(defaultValue = "10") Integer pageSize,

String name, Short gender,

@DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate begin,

@DateTimeFormat(pattern = "yyyy-MM-dd")

LocalDate end) {

log.info("分页查询，参数：{},{},{},{},{},{}", page,

pageSize,name, gender, begin, end);

//调用业务层分页查询功能

PageBean pageBean = empService.page(page, pageSize, name, gender, begin, end);

//响应

return Result.success(pageBean);

}

@DeleteMapping("/{ids}")

public Result delete(@PathVariable List<Integer> ids) {

empService.delet(ids);

return Result.success();

}

@PostMapping

public Result add(@RequestBody Emp emp) {

empService.add(emp);

return Result.success();

}

@GetMapping("/{id}")

public Result get(@PathVariable Integer id) {

Emp emp=empService.get(id);

return Result.success(emp);

}

@PutMapping

public Result update(@RequestBody Emp emp) {

empService.update(emp);

return Result.success();

}

}

**解释：**

1. page()方法用于分页查询员工信息。它接受页码、每页数量、姓名、性别、入职时间范围等参数,并调用EmpService层的page()方法来获取分页数据。最后将结果封装到Result对象中返回。
2. delete()方法用于删除指定ID的员工。它接受一个List<Integer>类型的ids参数,并调用EmpService层的delet()方法来删除数据,最后返回Result对象。
3. add()方法用于新增一个员工。它接受一个Emp对象作为参数,并调用EmpService层的add()方法来新增数据,最后返回Result对象。
4. get()方法用于根据ID获取员工详情。它接受一个Integer类型的id参数,并调用EmpService层的get()方法来获取数据,最后将结果封装到Result对象中返回。
5. update()方法用于更新员工信息。它接受一个Emp对象作为参数,并调用EmpService层的update()方法来更新数据,最后返回Result对象。

**Service**

public class EmpServiceImpl implements EmpService {

@Autowired

private EmpMapper empMapper;

@Override

public PageBean page(Integer page, Integer pageSize, String

name, Short gender, LocalDate begin, LocalDate end) {

//设置分页参数

PageHelper.startPage(page, pageSize);

//执行条件分页查询

List<Emp> empList = empMapper.list(name, gender, begin, end);

//获取查询结果

Page<Emp> p = (Page<Emp>) empList;//封装PageBean

PageBean pageBean = new PageBean(p.getResult(),p.getTotal());

return pageBean;

}

@Override

public void delet(List<Integer> ids) {

for (Integer id : ids) {

empMapper.delete(id);

}

}

@Override

public void add(Emp emp) {

emp.setCreateTime(LocalDateTime.now());

emp.setUpdateTime(LocalDateTime.now());

emp.setPassword("123456");

empMapper.add(emp);

}

@Override

public Emp get(Integer id) {

Emp emp=empMapper.get(id);

return emp;

}

@Override

public void update(Emp emp) {

emp.setUpdateTime(LocalDateTime.now());

empMapper.update(emp);

}

@Override

public Emp login(Emp emp) {

Emp e=empMapper.login(emp);

return e;

**解释：**

1. page()方法用于分页查询员工信息。它首先使用PageHelper工具设置分页参数,然后调用EmpMapper的list()方法查询满足条件的员工列表。最后将查询结果封装到PageBean对象中返回。
2. delet()方法用于删除指定ID的员工。它遍历传入的id列表,并逐个调用EmpMapper的delete()方法删除对应的员工记录。
3. add()方法用于新增一个员工。它为新增的Emp对象设置创建时间、更新时间和默认密码,然后调用EmpMapper的add()方法来保存数据。
4. get()方法用于根据ID获取员工详情。它调用EmpMapper的get()方法,并将查询结果直接返回。
5. update()方法用于更新员工信息。它设置Emp对象的更新时间,然后调用EmpMapper的update()方法来更新数据。
6. login()方法用于登录验证。它调用EmpMapper的login()方法,传入Emp对象作为参数,并返回查询结果。

**Mapper**

public interface EmpMapper {  
 //获取当前页结果，使用xml映射文件  
 public List<Emp> list(String name, Short gender, LocalDate begin, LocalDate end);  
 @Delete("delete from emp where id=#{id}")  
 void delete(Integer id);

@Insert("insert into emp (username, name, gender, image, job, entrydate, dept\_id, create\_time, update\_time,password) values (#{username}, #{name}, #{gender}, #{image},#{job},#{entrydate}, #{deptId}, #{createTime}, #{updateTime},#{password});")  
 void add(Emp emp);  
  
 @Select(("select id, username, password, name, gender, image, job, entrydate, dept\_id, create\_time, update\_time from emp where id = #{id}"))  
 Emp get(Integer id);  
  
 //使用xml映射文件  
 void update(Emp emp);  
  
 @Select("select \* from emp where username=#{username} and password=#{password}")  
 Emp login(Emp emp);  
}

**解释：**

1. list()方法用于根据条件分页查询员工列表。它接受姓名、性别、入职时间范围等参数,并在XML映射文件中定义具体的SQL语句。
2. delete()方法用于删除指定ID的员工。它使用@Delete注解直接定义SQL语句。
3. add()方法用于新增一个员工。它使用@Insert注解直接定义SQL语句,并接受一个Emp对象作为参数。
4. get()方法用于根据ID获取员工详情。它使用@Select注解直接定义SQL语句,并接受一个Integer类型的id参数。
5. update()方法用于更新员工信息。它在XML映射文件中定义具体的SQL语句,并接受一个Emp对象作为参数。
6. login()方法用于登录验证。它使用@Select注解直接定义SQL语句,并接受一个Emp对象作为参数。

**XML**

<mapper namespace="com.austin.mapper.EmpMapper">  
 <!-- 条件分页查询 -->  
 <select id="list" resultType="com.austin.pojo.Emp">  
 select \* from emp  
 <where>  
 <if test="name != null and name != ''">  
 name like concat('%',#{name},'%')  
 </if>  
 <if test="gender != null">  
 and gender = #{gender}  
 </if>  
 <if test="begin != null and end != null">  
 and entrydate between #{begin} and #{end}  
 </if>  
 </where>  
 order by update\_time desc  
 </select>  
  
<!-- 更新-->  
 <update id="update">  
 update emp  
 <set>  
 <if test="username != null and username != ''">  
 username = #{username},  
 </if>  
 <if test="password != null and password != ''">  
 password = #{password},  
 </if>  
 <if test="name != null and name != ''">  
 name = #{name},  
 </if>  
 <if test="gender != null">  
 gender = #{gender},  
 </if>  
 <if test="image != null and image != ''">  
 image = #{image},  
 </if>  
 <if test="job != null">  
 job = #{job},  
 </if>  
 <if test="entrydate != null">  
 entrydate = #{entrydate},  
 </if>  
 <if test="deptId != null">  
 dept\_id = #{deptId},  
 </if>  
 <if test="updateTime != null">  
 update\_time = #{updateTime}  
 </if>  
 </set>  
 where id = #{id}  
 </update>  
</mapper>

**解释：**

1. list()方法的SQL定义:
   * 这个SQL语句用于根据条件分页查询员工列表。
   * 它使用动态SQL的<where>和<if>标签来拼接查询条件,包括员工姓名、性别、入职时间范围等。
   * 最后按照更新时间降序排序结果集。
2. update()方法的SQL定义:
   * 这个SQL语句用于更新员工信息。
   * 它使用动态SQL的<set>和<if>标签来动态拼接更新字段,包括用户名、密码、姓名、性别、头像、职位、入职日期、部门ID、更新时间等。
   * 最后根据员工ID来更新对应的记录。

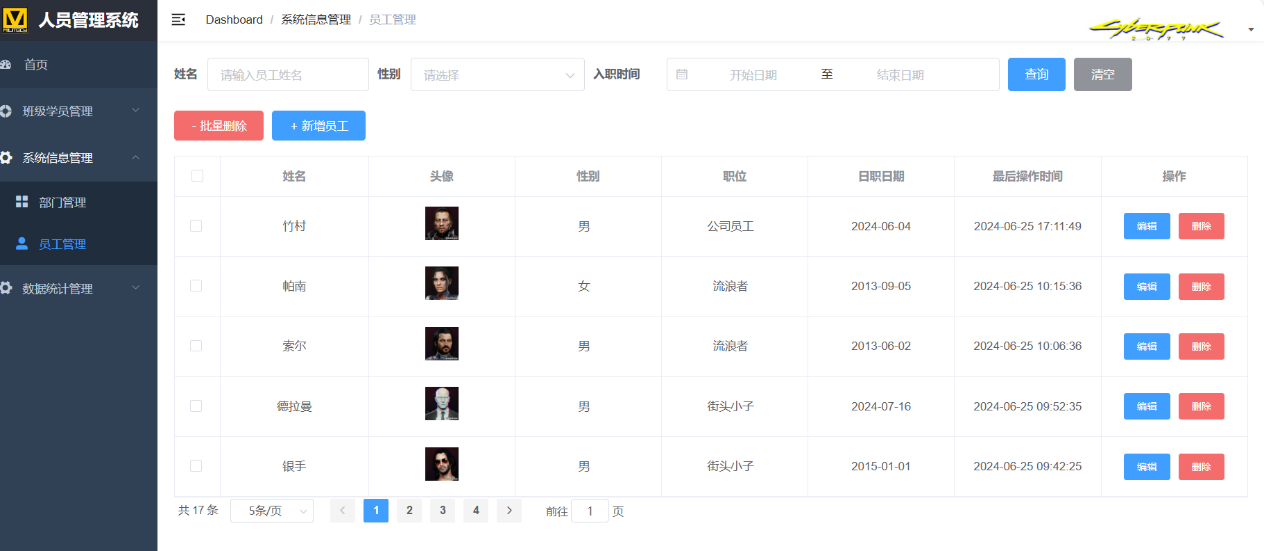
**截图：**

图5.5



图5.6 查询

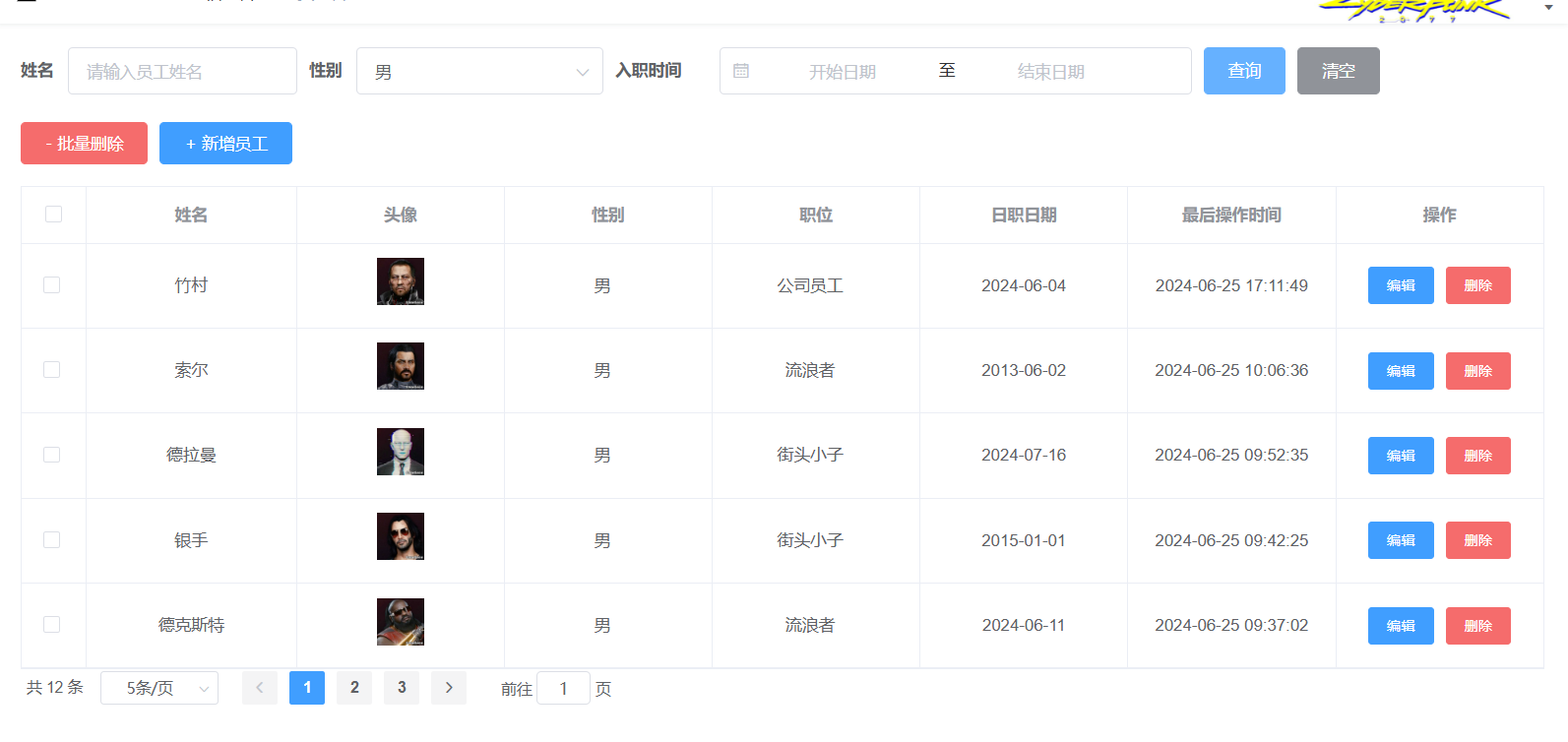


图5.7 根据性别查询



 图5.8 编辑员工

图5.9 编辑员工

图5.10 编辑员工

图5.11 新增员工



图5.12 新增员工

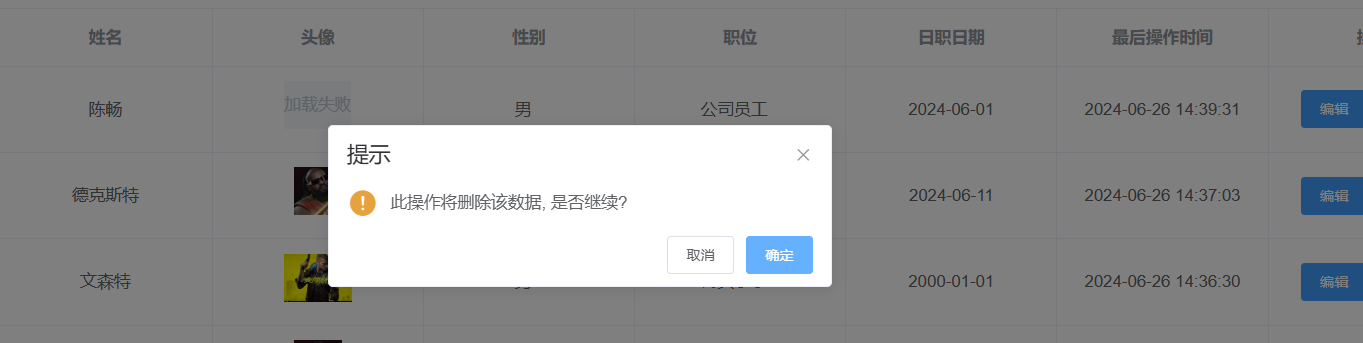
图5.13 删除员工

图5.14 删除员工

1. **Login**

**Controller**

public class LoginController {

//依赖业务层对象

@Autowired

private EmpService empService;

@PostMapping("/login")

public Result login(@RequestBody Emp emp) {

//调用业务层：登录功能

Emp loginEmp = empService.login(emp);

//判断：登录用户是否存在

if(loginEmp !=null ){

//自定义信息

Map<String , Object> map = new HashMap<>();

map.put("id", loginEmp.getId());

map.put("username",loginEmp.getUsername());

map.put("name",loginEmp.getName());

//使用JWT工具类，生成身份令牌

String token = JwtUtils.generateJwt(map);

return Result.success(token);

}

return Result.error("用户名或密码错误");

}

}

**解释：**

1. login方法使用@PostMapping注解映射到 /login 端点。该方法接受一个Emp对象作为输入,这个对象表示员工实体。
2. 在login方法内部,调用了empService.login(emp)方法进行用户认证。这个方法会根据提供的用户名和密码进行验证。
3. 如果用户登录成功,即loginEmp不为空,则创建一个包含用户ID、用户名和名称的Map对象。
4. 接着使用JWT工具类JwtUtils.generateJwt(map)生成身份令牌(token)。
5. 最后返回包含token的成功响应结果。
6. 如果登录失败,即loginEmp为空,则返回错误响应,提示"用户名或密码错误"。

**Filter:**

@WebFilter(urlPatterns = "/\*")

public class LoginFilter implements Filter {

@Override

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

//获取请求对象

HttpServletRequest request = (HttpServletRequest) servletRequest;

//获取响应对象

HttpServletResponse response = (HttpServletResponse) servletResponse;

//获取请求url

String requestURl = request.getRequestURL().toString();

log.info("requestURl = {}", requestURl);

//判断是否是login

if (requestURl.contains("/login")){

filterChain.doFilter(request, response);

return;

}

//获取请求令牌

String token=request.getHeader("token");

//令牌不存在，未登录

if (token==null){

log.info("token is null");

Result responseResult = Result.error("NOT\_LOGIN");

//转换为json串

String json = JSONObject.toJSONString(responseResult);

response.setContentType("application/json;charset=utf8");

//响应

response.getWriter().write(json);

return;

}

//解析jwt

try {

JwtUtils.parseJWT(token);

}catch (Exception exception) {

log.info("fail to check token");

Result responseResult = Result.error("NOT\_LOGIN");

//转换为json串

String json = JSONObject.toJSONString(responseResult);

response.setContentType("application/json;charset=utf8");

//响应

response.getWriter().write(json);

return;

}

filterChain.doFilter(request, response);

log.info("success to login...");

}

**解释：**

1. 映射URL
   * 这个过滤器使用@WebFilter注解将其映射到应用程序的所有URL("/\*")。这意味着在用户访问任何URL时,该过滤器都会被调用进行权限校验。
2. doFilter方法
   * doFilter方法是过滤器的核心实现,它会在每个请求到达目标资源之前被调用。
   * 在该方法内部,首先获取当前的HTTP请求和响应对象。
3. 获取请求URL
   * 接下来,获取当前请求的完整URL并打印到日志中,便于后续排查问题。
4. 检查登录状态
   * 首先判断请求URL是否包含"/login"。如果是登录请求,则直接放行,不进行任何权限验证。因为登录请求本身就是为了获取登录身份。
   * 如果不是登录请求,则从请求头中获取"token"参数,即用户的身份令牌。
5. 验证身份令牌
   * 如果token为null,表示用户未登录,则返回"NOT\_LOGIN"错误响应,提示用户需要先登录。
   * 如果token存在,则尝试使用JwtUtils.parseJWT(token)方法解析token。
   * 如果token解析失败,也返回"NOT\_LOGIN"错误响应,说明token无效或已过期。
6. 放行请求
   * 如果token解析成功,说明用户已登录,则放行请求,让其继续访问目标资源。
   * 同时打印"success to login..."日志,表示登录验证通过。

**jwtUtils：**

public class JwtUtils {  
  
 private static String *signKey* = "austin";  
 private static int *expire* = 300000;  
  
 //生成JWT令牌  
 //@param claims JWT第二部分负载 payload 中存储的内容  
 public static String generateJwt(Map<String, Object> claims){  
 String jwt = Jwts.*builder*()  
 .addClaims(claims)  
 .signWith(SignatureAlgorithm.*HS256*, *signKey*)  
 .setExpiration(new Date(System.*currentTimeMillis*() + *expire*))  
 .compact();  
 return jwt;  
 }  
  
 //解析JWT令牌  
 //@param jwt JWT令牌  
 //@return JWT第二部分负载 payload 中存储的内容  
 public static Claims parseJWT(String jwt){  
 Claims claims = Jwts.*parser*()  
 .setSigningKey(*signKey*)  
 .parseClaimsJws(jwt)  
 .getBody();  
 return claims;  
 }  
}

**解释：**

1. 生成JWT令牌
   * generateJwt方法接受一个Map<String, Object>类型的claims参数,表示要存储在JWT令牌负载(payload)中的数据。
   * 该方法使用JJWT库的Jwts.builder()构建一个JWT令牌。
   * 它将claims数据添加到令牌的负载部分,使用HS256算法对令牌进行签名,并设置令牌的过期时间(300秒/5分钟)。
   * 最后,将签名后的JWT令牌字符串返回。
2. 解析JWT令牌
   * parseJWT方法接受一个jwt参数,表示要解析的JWT令牌字符串。
   * 该方法使用JJWT库的Jwts.parser()解析JWT令牌。
   * 它使用与生成JWT令牌时相同的签名密钥(signKey)对令牌进行验签,以确保令牌的合法性。
   * 如果令牌验签通过,则提取并返回负载(payload)部分的数据,即之前存储的claims内容。

**截图：**

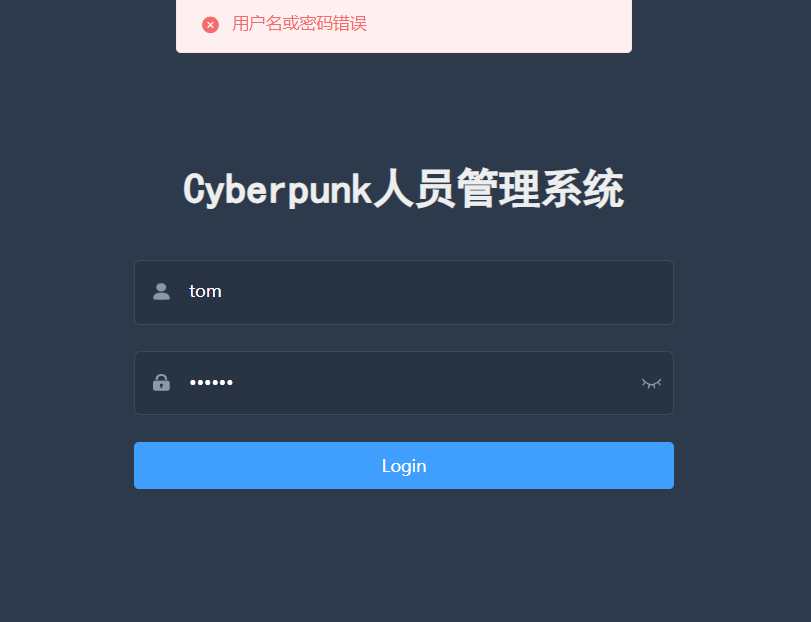


图5.15 登陆失败截图

1. **上传文件**

**Controller：**

public class UploadController {  
 @Autowired  
 private AliOSSUtils aliOSSUtils;  
  
 @RequestMapping("/upload")  
 public Result upload(MultipartFile image) throws IOException {  
 //调用oss工具类上传图片  
 String url=aliOSSUtils.upload(image);  
 //返回url给浏览器  
 return Result.*success*(url);  
 }  
  
}

**解释：**

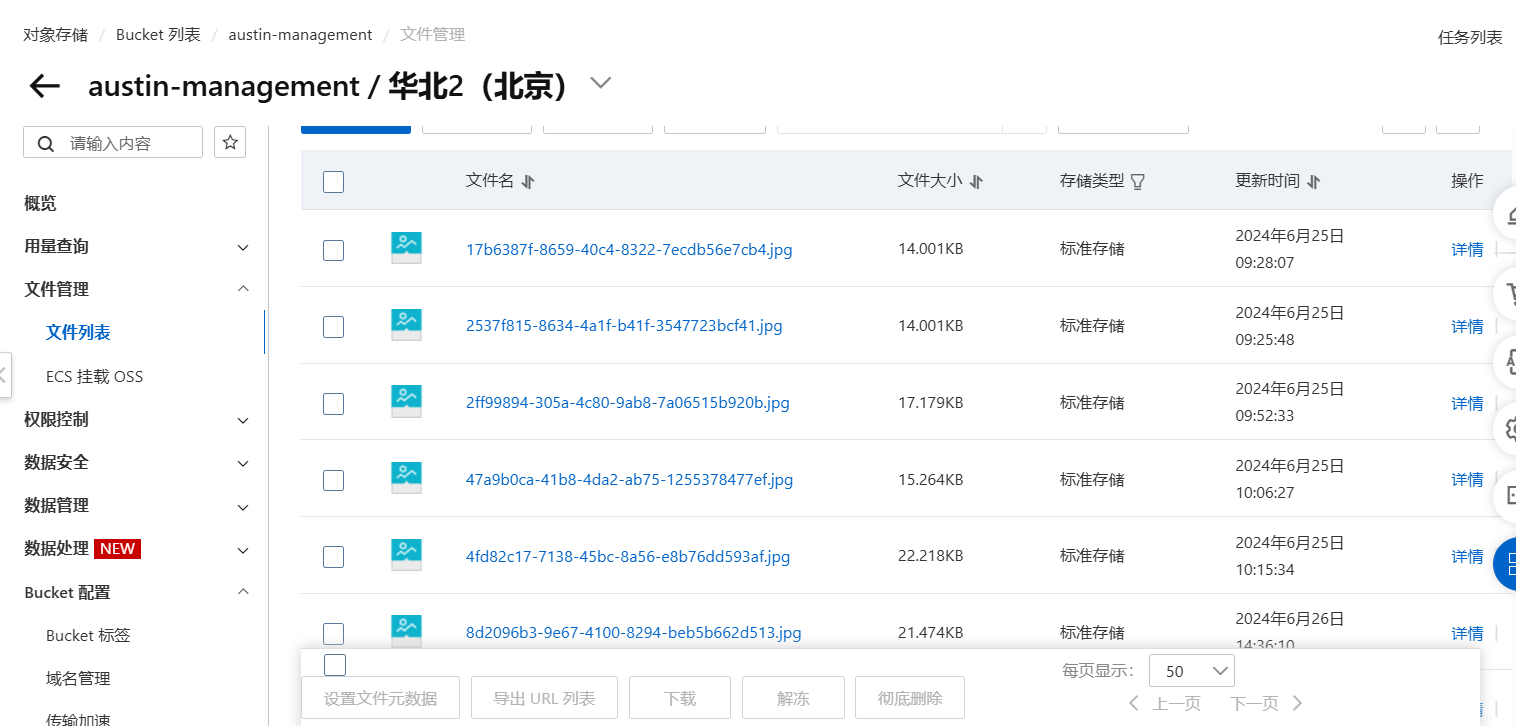
1. upload方法
   * upload方法使用@RequestMapping("/upload")注解将其映射到"/upload"这个URL。
   * 该方法接受一个MultipartFile类型的参数image,表示前端上传的图片文件。
2. 上传图片
   * 在upload方法内部,调用了aliOSSUtils.upload(image)方法上传图片到阿里云OSS。
   * aliOSSUtils.upload(image)方法会实现以下逻辑:
     + 生成一个唯一的文件名
     + 将图片文件上传到阿里云OSS
     + 返回图片在OSS上的访问URL
3. 返回结果
   * 上传成功后,将返回的图片URL封装到Result.success(url)对象中,作为响应返回给前端。
   * Result是一个自定义的响应结果类,用于统一响应格式。

**Aliossutils：**

public String upload(MultipartFile file) throws IOException {  
 // 获取上传的文件的输入流  
 InputStream inputStream = file.getInputStream();  
  
 // 避免文件覆盖  
 String originalFilename = file.getOriginalFilename();  
 //为文件随机取名  
 String fileName = UUID.*randomUUID*()+ originalFilename.substring(originalFilename.lastIndexOf("."));  
  
 //上传文件到 OSS  
 OSS ossClient = new OSSClientBuilder().build(aliOSSProperties.getEndpoint(),aliOSSProperties.getAccessKeyId(), aliOSSProperties.getAccessKeySecret());  
 ossClient.putObject(aliOSSProperties.getBucketName(), fileName, inputStream);  
  
 //文件访问路径  
 String url = aliOSSProperties.getEndpoint().split("//")[0] + "//" +aliOSSProperties.getBucketName() + "." + aliOSSProperties.getEndpoint().split("//")[1] + "/" + fileName;  
 // 关闭ossClient  
 ossClient.shutdown();  
 return url;// 把上传到oss的路径返回  
}

**解释：**

1. 获取上传文件的输入流
   * 通过file.getInputStream()获取上传文件的输入流。
2. 生成文件名
   * 获取上传文件的原始文件名(file.getOriginalFilename())。
   * 使用UUID.randomUUID()生成一个随机的文件名,并拼接上原始文件名的后缀。这样可以避免文件名重复导致的覆盖问题。
3. 上传文件到OSS
   * 创建一个OSS客户端实例(new OSSClientBuilder().build(...))。
   * 使用 ossClient.putObject(...) 将文件输入流上传到指定的OSS Bucket和文件名。
4. 生成文件访问URL
   * 根据OSS的访问域名规则,拼接出文件的访问URL。通常由Bucket名称和文件名组成。
5. 关闭OSS客户端
   * 上传完成后,调用ossClient.shutdown()关闭OSS客户端。
6. 返回文件访问URL
   * 最后,返回上传成功后的文件访问URL。

**截图：**

1. **Pojo**

**（1）Page bean**

public class PageBean {  
 //当前页数据表  
 private List rows;  
 //总记录数  
 private long total;  
}

**（2）result**

public class Result {  
 private Integer code;//响应码，1 代表成功; 0 代表失败  
 private String msg; //响应信息 描述字符串  
 private Object data; //返回的数据  
 //增删改 成功响应  
 public static Result success(){  
 return new Result(1,"success",null);  
 }  
 //查询 成功响应  
 public static Result success(Object data){  
 return new Result(1,"success",data);  
 }  
 //失败响应  
 public static Result error(String msg){  
 return new Result(0,msg,null);  
 }  
}

**解释：**

1. 类定义和属性:
   * Result是一个Java类,用于封装接口请求的返回结果。
   * 它包含三个属性:
     + code: 响应码,1表示成功,0表示失败。
     + msg: 响应信息,用于描述请求结果。
     + data: 返回的数据,可以是任意对象类型。
2. 静态方法:
   * success(): 创建一个成功的响应结果,code为1,msg为"success",data为null。
   * success(Object data): 创建一个成功的响应结果,code为1,msg为"success",data为传入的数据对象。
   * error(String msg): 创建一个失败的响应结果,code为0,msg为传入的错误信息,data为null。
3. 使用场景:
   * 在处理接口请求时,可以使用这些静态方法来创建标准化的响应结果。
   * 成功的请求可以使用success()或success(data)方法,返回成功码和数据。
   * 失败的请求可以使用error(msg)方法,返回失败码和错误信息。
4. **总结与收获**
5. JWT 身份验证机制:
   * 了解 JWT 的结构,包括头部、有效载荷和签名三个部分。
   * 掌握使用 JJWT 库生成和解析 JWT 令牌的具体实现,包括设置签名密钥、过期时间等。
   * 理解 JWT 在身份认证和授权中的应用场景,以及相比传统 session/cookie 方式的优势,如更好的扩展性和安全性。
   * 学会如何检查 JWT 令牌的合法性和过期状态,维护应用程序的安全性。
6. 文件上传到云存储:
   * 学会使用 Java 的 MultipartFile 接口处理客户端上传的文件,获取文件输入流等信息。
   * 掌握如何生成唯一的文件名,避免文件覆盖问题。可以使用 UUID 或者结合原文件名的方式。
   * 熟悉如何使用阿里云 OSS SDK 上传文件,包括创建 OSS 客户端、设置 Bucket 信息等。
   * 了解如何根据 OSS 的域名规则,拼接出文件的访问 URL。这样可以方便其他系统访问上传的文件。
   * 学会在使用完 OSS 客户端后及时关闭,以释放资源,体现良好的资源管理habit。
7. Java 编程技巧:
   * 掌握使用 try-with-resources 语法确保资源的正确关闭,避免资源泄漏。
   * 学会合理地封装方法,提高代码的可读性和可维护性。比如将文件上传的逻辑封装在一个独立的方法中。
   * 理解使用 UUID 生成唯一 ID 的技巧,解决文件名重复的问题。
   * 了解如何通过配置属性灵活地与云服务交互,增强代码的可配置性。
8. 云服务的使用方式:
   * 学会如何获取和配置阿里云 OSS 的相关属性,如 endpoint、accessKeyId、accessKeySecret、bucketName 等。
   * 掌握如何创建和使用 OSS 客户端与云服务交互,完成文件上传等操作。
   * 理解云服务在企业级应用中的作用,如将文件存储和管理交给专业的云服务提供商,提高应用程序的性能和可扩展性。
9. 学会了如何处理服务器出现乱码，Java项目编码相关知识。学会了应该即使输出日志排查错误。