JWT 实战

基本配置

最基础的Spring Security登陆校验

首先在项目文件路径下添加 config/securityConfig.java 文件

```
@Configuration
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .authorizeHttpRequests(conf -> conf
                        // 所有该uri下的请求都允许被放行
                        .requestMatchers("/api/auth/**").permitAll()
                        // 其他所有的请求都需要验证完之后才能访问
                        .anyRequest().authenticated()
                )
                .formLogin(conf -> conf
                        // 登录请求页面
                        .loginProcessingUrl("/api/auth/login")
                        // 登录成功的处理
                        .successHandler(this::onAuthenticationSuccess)
                        // 失败登录
                        .failureHandler(this::onAuthenticationFailure)
                )
                .logout(conf -> conf
                        // 登出的请求页面
                        .logoutUrl("/api/auth/logout")
                        .logoutSuccessHandler(this::onLogoutSuccess)
                )
                // 取消跨域检查
                .csrf(AbstractHttpConfigurer::disable)
                // 将session修改为无状态
                .sessionManagement(conf -> conf
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .build();

    }

    // 登录成功的处理
    public void onAuthenticationSuccess(HttpServletRequest request,
                                        HttpServletResponse response,
                                        Authentication authentication) throws IOException,
ServletException {
        // 告诉前端返回的内容格式是 json
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        response.getWriter().write(RestBean.success().asJsonString());
```

```
    }

    // 登录失败的处理
    public void onAuthenticationFailure(HttpServletRequest request,
                                        HttpServletResponse response,
                                        AuthenticationException exception) throws
IOException, ServletException {
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        // 登陆错误使用，讲捕获到的错误信息传入
        response.getWriter().write(RestBean.failure(401,
exception.getMessage()).asJsonString());
    }

    // 登出成功
    public void onLogoutSuccess(HttpServletRequest request,
                                HttpServletResponse response,
                                Authentication authentication) throws IOException,
ServletException {

    }

}
```

在这里为了让 response.getWriter.write() 的响应体的内容更加规范，方便在前端进行调用，可以直接创建一个专门的响应类 enetity/RestBean.java 用来进行格式化响应

```
public record RestBean<T>(int code, T data, String message) {

    public static <T> RestBean<T> success(T data) {
        return new RestBean<>(200, data, "请求成功");
    }

    public static <T> RestBean<T> success() {
        return success(null);
    }

    public static <T> RestBean<T> failure(int code, String message) {
        return new RestBean<>(code, null, message);
    }

    // 将对象转换成 json 的字符串格式
    public String asJsonString() {
        return JSONObject.toJSONString(this, JSONWriter.Feature.WriteNulls);
    }

}
```

JWT 令牌创建

分发 JWT 令牌，可以通过创建一个 utils/JwtUtils.java 工具类实现

JWT 校验的思维实际上就是在 Spring Security 对第一次登陆的用户进行校验之后，如果通过了认证那么就会生成一个带有用户名以及过期时间等信息的 JWT 令牌，然后次后该用户的每个请求都会带上这个令牌，此时对于拥有该令牌的用户不会再进行登录拦截。

这里的 UserDetail 是 spring security 框架中自带的用来存储登陆用户的信息

- `Authorities` 是 JWT 中的一个自定义声明（Claim），用于存储用户的权限信息（如角色、权限等）。
    - 在 JWT 的 Payload 部分，`Authorities` 通常是一个数组或列表，存储用户的权限信息

首先在 resources/application.yml 配置类中添加有关 jwt 生成需要的盐值以及过期时间，这里设置的过期时间是 7 天

```yaml
spring:
  security:
    jwt:
      key: "abcdefghick"
      expire: 7
```

然后编写 JWT 有关的工具类

```java
@Component
public class JwtUtils {

    // 用来生成JWT的密钥
    @Value("${spring.security.jwt.key}")
    String key;

    // 过期时间
    @Value("${spring.security.jwt.expire}")
    int expire;

    // 创建JWT令牌
    // UserDetail 里面的 name 是登录的邮箱，所以可以修改为后面更改的 username
    public String createJwt(UserDetails userDetails, int id, String username) {
        // 密钥加密的算法
        Algorithm algorithm = Algorithm.HMAC256(key);
        // 根据yml文件中写入的expire键的天数来决定过期的时间
        Date expire = this.expireTime();

        return JWT.create()
                // UserDetail 中没有id，所以需要单独传入
                .withClaim("id", id)
                .withClaim("name", username)
                .withClaim("Authorities",
userDetails.getAuthorities().stream().map(GrantedAuthority::getAuthority).toList())
                // 过期时间，按小时来算
                .withExpiresAt(expire)
                // 令牌颁发时间
```

```
                .withIssuedAt(new Date())
                .sign(algorithm);
    }

    // 计算过期的时间
    public Date expireTime() {
        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.HOUR, expire * 24);
        return calendar.getTime();
    }

}
```

修改 SecurityConfig.java 文件中的 onAuthenticationSuccess 为：

```
public void onAuthenticationSuccess(HttpServletRequest request,
                                    HttpServletResponse response,
                                    Authentication authentication) throws IOException,
ServletException {
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        // 获取当前登录的用户信息
        User user = (User) authentication.getPrincipal();
        // 使用当前登录的用户信息创建专属的JWT令牌
        String token = jwtUtils.createJwt(user, 1, "小明");
        // 创建一个返回信息的实体类
        AuthorizeVO vo = new AuthorizeVO();
        vo.setExpires(jwtUtils.expireTime());
        vo.setRole("");
        vo.setToken(token);
        vo.setUsername("小明");
        response.getWriter().write(RestBean.success(vo).asJsonString());
    }
```

JWT 请求头校验

因为 Spring Security 有自己的 FilterChain 来完成初始的用户登陆校验，所以我们可以把后续用户是否携带符合要求的 JWT 令牌的校验通过一个自定义的 filter/JwtAuthorizeFilter.java 来添加进去

- `Authorization` 是 HTTP 请求头的一部分，通常用于携带认证信息（如 JWT、Basic Auth 等），以便服务器验证客户端的身份。
    - 常见的格式是 `Bearer <token>`，其中 `<token>` 是 JWT 或其他类型的令牌，客户端在发起请求时，将 JWT 或其他认证信息放入 `Authorization` 请求头中，服务器通过解析该请求头来验证用户身份

```
@Component
// 这里继承的 OncePerRequestFilter 代表每次请求都会出发这个过滤器
public class JwtAuthorizeFilter extends OncePerRequestFilter {

    @Resource
    JwtUtils utils;
```

```
    // 在这编写自定义过滤器的认证逻辑，然后加入到Spring Security的FilterChain当中来进行登录的验证
    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException,
IOException {
        // 从请求头中读取中验证信息 (注意区分)
        String authorization = request.getHeader("Authorization");
        // 调用工具类解析token
        DecodedJWT jwt = utils.resolveJwt(authorization);
        // 如果token不为null
        if (jwt != null) {
            // 则获取JWT内部保存的SpringSecurity的UserDetails对象
            UserDetails user = utils.toUser(jwt);
            // Spring Security中Authentication接口的一个实现类。这个Authentication对象表示当前用户
已经通过认证。
            UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(user, null,
user.getAuthorities());
            // 这行代码为上面的对象设置额外的详细信息。
            // 会从当前请求中提取一些细节信息（如远程IP地址、Session ID等），并将其绑定到
Authentication对象中。
            authentication.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
            // 目的是为了在当前请求的上下文中标识用户的认证状态和权限信息。这样，Spring Security可以在后
续的请求处理过程中
            // （如方法调用、URL访问等）基于这些信息进行权限检查和访问控制
            SecurityContextHolder.getContext().setAuthentication(authentication);
            request.setAttribute("id", utils.toId(jwt));
        }
        // 继续执行后续的过滤器
        filterChain.doFilter(request, response);
    }
}
```

然后我们可以在 utils/JwtUtil.java 中添加一些有关 JWT 解析， 从 JWT 令牌中读取包含的用户信息，获取 JWT 中对应用户的id 和 单纯提取 token 部分的方法

```
// 解析Token
public DecodedJWT resolveJwt(String headerToken) {
    String token = this.convertToken(headerToken);
    if (token == null) {
        return null;
    }
    // 搭配密钥的解析算法
    Algorithm algorithm = Algorithm.HMAC256(key);
    JWTVerifier jwtVerifier = JWT.require(algorithm).build();
    try {
        // 解析token
        DecodedJWT verify = jwtVerifier.verify(token);
        // 判断当前token是否已经过期
```

```
        Date expiresAt = verify.getExpiresAt();
        // 如果超期则也返回null
        return new Date().after(expiresAt) ? null : verify;
    } catch (JWTVerificationException e) {
        // 验证失败，则返回空
        return null;
    }
}

// 解析token中包含的用户信息
// 这里的密码并不重要，因为有jwt令牌的人肯定已经通过了最初的登录校验
public UserDetails toUser(DecodedJWT jwt) {
    Map<String, Claim> claims = jwt.getClaims();
    return User
            // 提取到 JWT 在生成阶段设置的 name
            .withUsername(claims.get("name").asString())
            .password("******")
            .authorities(claims.get("Authorities").asArray(String.class))
            .build();
}

// 获取请求的id
public Integer toId(DecodedJWT jwt) {
    Map<String, Claim> claims = jwt.getClaims();
    return claims.get("id").asInt();
}

// 判断token
private String convertToken(String headerToken){
    // 如果请求头中的校验信息为空或者开头不是 Bearer 那么直接返回空
    if(headerToken == null || !headerToken.startsWith("Bearer "))
        return null;
    // 不然就把前面的开头去掉
    return headerToken.substring(7);
}
```

然后将自定义的 JWT 验证逻辑添加到 SecurityConfg.java 中

```
@Resource
JwtAuthorizeFilter jwtAuthorizeFilter;

.....

    .logout(conf -> conf
            .logoutUrl("/api/auth/logout")
            .logoutSuccessHandler(this::onLogoutSuccess)
    )
    // 捕获登录异常 (新添加)
    .exceptionHandling(conf -> conf
            .authenticationEntryPoint(this::onUnauthorized)  // 没有登陆
            .accessDeniedHandler(this::onAccessDeny)  // 没有访问权限
    )
```

```
    // 取消跨域检查
    .csrf(AbstractHttpConfigurer::disable)
    // 将session修改为无状态 (新添加)
    .sessionManagement(conf -> conf
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
    // 要把自定义的过滤器加在UsernamePasswordAuthenticationFilter前面进行校验 (新添加)
    .addFilterBefore(jwtAuthorizeFilter, UsernamePasswordAuthenticationFilter.class)
    .build();

...

// 新添加的捕获异常的方法
// 在没有验证的情况下，也就是没有登录的时候
public void onUnauthorized(HttpServletRequest request,
                           HttpServletResponse response,
                           AuthenticationException exception) throws IOException {
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    // 没有登录

 response.getWriter().write(RestBean.unauthorized(exception.getMessage()).asJsonString());
}

// 没有权限
public void onAccessDeny(HttpServletRequest request,
                         HttpServletResponse response,
                         AccessDeniedException exception) throws IOException {
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    response.getWriter().write(RestBean.forbidden(exception.getMessage()).asJsonString());
}
```

在标准返回类 enetity/RestBean.java 中加入新的范围类型

```
public static <T> RestBean<T> unauthorized(String message) {
    return failure(401, message);
}

public static <T> RestBean<T> forbidden(String message) {
    return failure(403, message);
}
```

JWT 退出登录

可以通过借用redis的方式来实现令牌的失效。正常情况下令牌失效之后不会被删除，所以为了节省空间应该在令牌失效之后也将令牌的记录删除掉

导入 redis 和 Mybatis-plus 依赖

```
<!--redis-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<!--mybatis-plus-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.3.1</version>
</dependency>
```

修改 JwtUtils.java 中的内容，添加删除令牌，以及为之前的方法添加判断当前传入的token是否已经在黑名单中的方法，以及在创建 JWT 的时候再单独创建一个 UUID。

```
@Resource
StringRedisTemplate template;

// 让令牌失效，也就是将令牌加入黑名单
public boolean invalidateJwt(String headerToken) {
    // 如果令牌为空则至直接返回false
    // 获取token，去掉开头的Bearer
    String token = this.convertToken(headerToken);
    if (token == null) {
        return false;
    }
    // 验证一下令牌
    Algorithm algorithm = Algorithm.HMAC256(key);
    JWTVerifier jwtVerifier = JWT.require(algorithm).build();
    // 拉黑提供的令牌，通过查找其对应的UUID
    try {
        // 这里的getId获取的是JWTID也就是我们的UUID而不是我们自定义的id，自定义id的获取方式是
jwt.getClaim("id").asInt();
        DecodedJWT jwt = jwtVerifier.verify(token);
        String id = jwt.getId();
        // 拉黑令牌
        return deleteToken(id, jwt.getExpiresAt());
    } catch (JWTVerificationException e) {
        return false;
    }
}

// 删除令牌（拉黑令牌）
private boolean deleteToken(String uuid, Date time) {
    // 如果再黑名单中则不需要添加
    if (this.isInvalidToken(uuid)) {
        return false;
    }
    // 这里在将token存放进黑名单中的时候，可以计算当前的token还有多久过期，然后可以只在redis中保存剩余的时
间，节约资源
    Date now = new Date();
    // 如果是负数则让时间设置为0
```

```
    long expire = Math.max(time.getTime() - now.getTime(), 0);
    template.opsForValue().set(Const.JWT_BLACK_LIST + uuid, "", expire,
TimeUnit.MILLISECONDS);
    return true;
}

// 判断当前提供的uuid的token是否已经在redis的黑名单中了
private boolean isInvalidToken(String uuid) {
    return Boolean.TRUE.equals(template.hasKey(Const.JWT_BLACK_LIST + uuid));
}

// 解析Token
public DecodedJWT resolveJwt(String headerToken) {
    String token = this.convertToken(headerToken);
    if (token == null) {
        return null;
    }
    Algorithm algorithm = Algorithm.HMAC256(key);
    JWTVerifier jwtVerifier = JWT.require(algorithm).build();
    try {
        // 解析token
        DecodedJWT verify = jwtVerifier.verify(token);
        // 判断当前token是否在黑名单中 （添加新功能）
        if (this.isInvalidToken(verify.getId()))
            return null;
        // 判断当前token是否已经过期
        Date expiresAt = verify.getExpiresAt();
        // 如果超期则也返回null
        return new Date().after(expiresAt) ? null : verify;
    } catch (JWTVerificationException e) {
        // 验证失败，则返回空
        return null;
    }
}

// 创建JWT令牌
public String createJwt(UserDetails userDetails, int id, String username) {
    Algorithm algorithm = Algorithm.HMAC256(key);
    // 根据yml文件中写入的expire键的天数来决定过期的时间
    Date expire = this.expireTime();
    // 创建令牌
    return JWT.create()
            // 为每一个JWT令牌生成一个UUID用来进行标识 （添加新功能）
            .withJWTId(UUID.randomUUID().toString())
            .withClaim("id", id)
            .withClaim("name", username)
            .withClaim("Authorities",
userDetails.getAuthorities().stream().map(GrantedAuthority::getAuthority).toList())
            // 过期时间，按小时来算
            .withExpiresAt(expire)
            .withIssuedAt(new Date())
            .sign(algorithm);
}
```

为了方便在不同的文件中调用相同的变量名，可以创建一个 utils/Const.java

```
// 存储经常要是用到的一些属性
public class Const {

    public static final String JWT_BLACK_LIST = "jwt:blacklist:";

}
```

修改 SecurityConfig.java 中的退出登录方法，

```
// 登出成功
public void onLogoutSuccess(HttpServletRequest request,
                            HttpServletResponse response,
                            Authentication authentication) throws IOException,
ServletException {
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    PrintWriter writer = response.getWriter();
    String headerToken = request.getHeader("Authorization");
    // 让当前的jwt失效，如果失效成功
    if (jwtUtils.invalidateJwt(headerToken)) {
        writer.write(RestBean.success().asJsonString());
    } else {
        writer.write(RestBean.failure(400, "退出登录失败").asJsonString());
    }
}
```

实现根据数据库用户信息进行登录校验

首先在数据库中添加 entity/dto/Account.java 实体类，用来方便提取并存放从数据库 account 表中提取到的有关用户的信息，这里的字段也分别代表着该表在数据库中的字段。

```
@Data
// 对应的数据库中表的名称
@TableName("db_account")
// 全参构造器
@AllArgsConstructor
public class Account {

    // 不需要专门指定id，在存入数据库的时候会自动根据自增生成id
    @TableId(type = IdType.AUTO)
    Integer id;
    String username;
    String password;
    String email;
    String role;
    Date registerTime;

}
```

然后在 resources/application.yml 中添加如下配置，datasource 与上面的 security 同级。url就是你数据库连接后面跟上你使用的数据库名字

```
  datasource:
    url: jdbc:mysql://localhost:3306/test
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
```

然后添加 mapper/AccountMapper.java，这个类因为继承了 Mybatis-Plus 的BaseMapper，所以针对数据库的基本操作可以直接通过调用方法来实现。然后这里指定的类型为 Account

```
@Mapper
public interface AccountMapper extends BaseMapper<Account>{
}
```

接下来创建 service 层，首先是 service/AccountService.java 接口

```
public interface AccountService extends IService<Account>, UserDetailsService {

    // 根据提供的用户名或者邮箱查找对应的用户
    Account findAccountByNameOrEmail(String text);

}
```

然后编写实现类 service/impl/AccountServiceImpl.java

注意，这里的 loadUserByUsername 是由于在 AccountService 中继承了 UserDetailService，然后需要实现这个抽象方法，然后 Spring Security 在进行登录校验的时候会自动调用这个方法根据登录表单中传入的用户名的信息来获取到 User 类并用于接下来的校验操作。

```
@Service
public class AccountServiceImpl extends ServiceImpl<AccountMapper, Account> implements
AccountService {

    // 自定义查询用户信息
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
{

        // 通过用户名获取用户信息
        Account account = this.findAccountByNameOrEmail(username);
        // 如果传入的内容为空
        if (account == null) {
            throw new UsernameNotFoundException("用户名或者密码错误");
        }
        // 成功则返回对应username的用户
        return User
                .withUsername(username)
                .password(account.getPassword())
                .roles(account.getRole())
```

```
                .build();
    }

    // 实现service接口中的通过用户名或者邮箱来查找对应的用户
    public Account findAccountByNameOrEmail(String text) {
        return this.query()
                .eq("username", text).or()
                .eq("email", text)
                .one();
    }
}
```

此外，还需要添加一个指定密码校验的类，因为在数据库中存放的密码都是加密过后的密文，所以在登录的时候，当 Spring Security 获取到用户在登录表单中输入的明文密码之后，会通过这个指定的密码转换方式以及盐值加密然后和数据库中读取到的密码对比校验。下面的代码可以单独写在一个类中，然后通过 @Bean 自动注入
config/WebConfiguration.java

```
@Configuration
public class WebConfiguration {

    @Bean
    BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

}
```

然后就是修改 SecurityConfig.java 中有关第一次登陆成功后生成 JWT 部分的代码

```
// 登录成功的处理
public void onAuthenticationSuccess(HttpServletRequest request,
                                    HttpServletResponse response,
                                    Authentication authentication) throws IOException,
ServletException {
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    // 获取当前登录的用户信息
    User user = (User) authentication.getPrincipal();
    // 查找到当前User中username的信息并找到数据库中的实体类，方便设置下面的json返回信息
    Account account = accountService.findAccountByNameOrEmail(user.getUsername());
    // 使用当前登录的用户信息创建专属的JWT令牌
    String token = jwtUtils.createJwt(user, account.getId(), account.getUsername());
    // 创建一个返回信息的实体类
    AuthorizeVO vo = new AuthorizeVO();
    vo.setExpires(jwtUtils.expireTime());
    vo.setRole(account.getRole());
    vo.setToken(token);
    vo.setUsername(account.getUsername());
    response.getWriter().write(RestBean.success(vo).asJsonString());
}
```

前端登陆页面设计

我这里使用的是element-plus ui，实际上也有很多其他的前端框架可以选择

在 IDEA 或者 WebStorm 中创建了 VUE 项目之后进入，可以先讲 src/asset 和 src/component 下面的文件都先删除掉，方便后续加入自己的设计

首先在终端输入

```
npm install vue-router
```

这样可以实现通过路由来访问不同的页面

然后就在项目文件夹下创建 router/index.js 用来填写前端不同页面的路由地址，这里设置的默认路由是"/"，也就是当访问默认的localhost:5173 之后会渲染 WelcomeView 的页面，该路由还有一个子路由，也就是可以在 WelcomeView 中再通过添加 展示

```
import {createRouter, createWebHistory} from "vue-router";

const router = createRouter({
    history: createWebHistory(import.meta.env.BASE_URL),
    routes: [
        {
            path: "/",
            name: "welcome",
            component: () => import('@/views/WelcomeView.vue'),
            children: [
                {
                    path: '',
                    name: 'welcome-login',
                    component: () => import('@/views/welcome/LoginPage.vue')
                }
            ]
        }
    ]
})

export default router
```

然后修改 main.js， 这是 Vue 应用的入口文件，负责初始化应用并挂载到 DOM 中

```
import { createApp } from 'vue'
import App from './App.vue'
import router from "@/router/index.js";
import axios from "axios";
import 'element-plus/theme-chalk/dark/css-vars.css'

axios.defaults.baseURL = 'http://localhost:8080'

// 创建 Vue 应用实例，并将 App.vue 作为根组件
const app = createApp(App)
```

```
app.use(router)

app.mount('#app')
```

上面的axios是后面会用到想后端发送请求用的工具，可以直接通过npm下载。这里是将其默认的传输路径设置为localhost:8080，因为后端的本地请求地址是这个

```
npm install axios
```

然后需要修改 App.vue 文件，这个文件是是 Vue 应用的根组件，所有其他组件（包括路由组件）都会在它的上下文中渲染。

```
<script setup>
</script>

<template>
  <div>
    <!-- 路由匹配的组件会渲染在这里， 当访问 / 路径时，WelcomeView.vue 组件会渲染在 <router-view> 的位置 -->
    <!-- 这里实际上就是路由渲染展示的地方，访问某个路由，那么对应的页面就会展示在这里 -->
    <router-view/>
  </div>
</template>

<style scoped>

</style>
```

然后就是编写默认的请求页面，也就是 WelcomeView.vue

```
<script setup>

</script>

<template>
  <div style="width: 100vw;height: 100vh;overflow: hidden;display: flex">
    <!--左侧图片-->
    <div style="flex: 1;background-color: black">
      <el-image style="width: 100%;height: 100%" fit="cover"

 src="https://img1.baidu.com/it/u=4097856652,4033702227&fm=253&fmt=auto&app=120&f=JPEG?w=1422&h=800"/>
    </div>
    <!-- 左侧图片中的标题 -->
    <div class="welcome-title">
      <div style="font-size: 30px;font-weight: bold ">欢迎来到我们的管理系统</div>
      <div style="margin-top: 10px">在这里你可以</div>
      <div style="margin-top: 10px">唱，跳，rap，篮球</div>
    </div>
    <!-- 右侧登陆与注册页面 -->
    <div class="right-card">
```

```
      <router-view/>
    </div>
  </div>

</template>

<style scoped>
.welcome-title {
  position: absolute;
  bottom: 30px;
  left:30px;
  color: white;
  text-shadow: 0 0 10px black;
}

.right-card {
  width: 400px;
  z-index: 1;
  background-color: white;
}
</style>
```

以及其子路由 LoginPage.vue，这个直接渲染在 WelcomeView.vue 的 中

```
<script setup>
import { User, Lock } from '@element-plus/icons-vue';  // 导入 Element Plus 的用户icon图标
import {reactive, ref} from "vue";  // 导入 Vue 的 reactive 函数，用于创建响应式对象。
import {login} from "@/net/index.js";  // 导入 Vue 的 reactive 函数，用于创建响应式对象。

// 做校验
const formRef = ref()

// 使用 reactive 创建一个响应式对象 form，用于存储表单数据
const form = reactive( {
  username: '',
  password: '',
  remember: false
})

// 校验规则
const rule = {
  username: [
    {required: true, message: '请输入用户名'}
  ],
  password: [
    {required: true, message: '请输入密码'}
  ]
}

// 登陆函数
// 这里会通过上面自定义的 rule 方法来校验合规性，因为下面的表单中已经指定了 rules
function userLogin() {
  formRef.value.validate((valid) => {
```

```
    if (valid) {
      login(form.username, form.password, form.remember, () => {})
    }
  })
}
</script>

<template>
  <div style="text-align: center;margin: 0 20px">
    <div style="margin-top: 150px">
      <div style="font-size: 25px;font-weight: bold">登陆</div>
      <div style="font-size: 14px;color: grey">在进入系统之前，请先输入用户名和密码进行登陆</div>
    </div>

    <!-- 登陆页面 -->
    <div style="margin-top: 50px">
      <!-- 将表单与 form 响应式对象绑定。 -->
      <el-form :model="form" :rules="rule" ref="formRef">
        <el-form-item prop="username">
          <!-- 将输入框的值与 form.username 绑定 -->
          <el-input v-model="form.username" maxlength="10" placeholder="用户名/邮箱">
            <template #prefix>
              <el-icon><User/></el-icon>
            </template>
          </el-input>
        </el-form-item>

        <el-form-item prop="password">
          <el-input v-model="form.password" type="password" maxlength="20" placeholder="密
码">
            <template #prefix>
              <el-icon><Lock/></el-icon>
            </template>
          </el-input>
        </el-form-item>

        <el-row>
          <el-col :span="12" style="text-align: left">
            <el-form-item prop="remember">
              <el-checkbox v-model="form.remember" label="记住我"/>
            </el-form-item>
          </el-col>

          <el-col :span="12" style="text-align: right">
            <el-link>忘记密码？</el-link>
          </el-col>
        </el-row>
      </el-form>
    </div>

    <!-- 登陆按钮 -->
    <div style="margin-top: 40px">
```

```
      <el-button @click="userLogin" style="width: 270px" type="success" plain>立即登录</el-
button>
    </div>

    <!-- 注册按钮 -->
    <el-divider>
      <span style="font-size: 13px;color: grey">没有账号</span>
    </el-divider>
    <div>
      <el-button style="width: 270px" type="warning" plain>立即注册</el-button>
    </div>
  </div>
</template>

<style scoped>

</style>
```

修改根目录同级的index.html

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="UTF-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="//unpkg.com/element-plus/dist/index.css">
      <style>
          body {
              margin: 0;
          }
      </style>
    <title>Vite App</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="./src/main.js"></script>
  </body>
</html>
```

使用 AXIOS 发送请求

创建 sec/net/index.js 文件，这里主要是一些使用axios发送信息的方法模板

```
import axios from 'axios'
import {ElMessage} from "element-plus";

const authItemName = "access_token"
```

```
// 默认的错误处理
const defaultFailure = (message, code, url) => {
    console.warn(`请求地址: ${url}, 状态码: ${code}, 错误信息: ${message}`)
    ElMessage.warning(message)
}

// 默认异常处理
const defaultError = (err) => {
    console.error(err)
    ElMessage.warning("发生了一些错误，请联系管理员")
}

// 获取请求头中的token
function takeAccessToken() {
    const str = localStorage.getItem(authItemName) || sessionStorage.getItem(authItemName);
    if (!str) return null;
    // 将字符串解析为JSON
    const authObj = JSON.parse(str);
    // 当前token的时间是否超时
    if (authObj.expire <= new Date()) {
        deleteAccessToken()
        ElMessage.warning("登录状态过期，请重新登录")
        return null
    }
    return authObj.token
}

// remember为是否记住登录状态
// 保存token
function storeAccessToken(token, remember, expire) {
    // 将 token 和 expire 封装成一个对象 authObj，方便存储和读取
    const authObj = {token: token, expire: expire}
    // 将对象转换为字符串
    const str = JSON.stringify(authObj)
    if (remember)
        localStorage.setItem(authItemName, str)
    else
        // 不勾选记住我，则保存在session中
        sessionStorage.setItem(authItemName, str)
}

// 从存储中删除访问的token
function deleteAccessToken() {
    localStorage.removeItem(authItemName)
    sessionStorage.removeItem(authItemName)
}

function internalPost(url, data, header, success, failure, error = defaultError) {
    axios.post(url, data, {
        headers: header
    }).then(({data}) => {
        if (data.code === 200) {
```

```
            success(data.data)
        } else {
            failure(data.message, data.code, url)
        }
    }).catch(err => error(err))
}

function internalGet(url, data, header, success, failure, error = defaultError) {
    axios.get(url, {headers: header}).then(({data}) => {
        if (data.code === 200) {
            success(data.data)
        } else {
            failure(data.message, data.code, url)
        }
    }).catch(err => error(err))
}

function login(username, password, remember, success, failure = defaultFailure) {
    internalPost('/api/auth/login', {
        username: username,
        password: password,
    }, {
        'Content-Type': 'application/x-www-form-urlencoded'
    }, (data) => {
        storeAccessToken(remember, data.token, data.expire)
        ElMessage.success(`登录成功，欢迎${data.username}来到我们的系统`)
        success(data)
    }, failure)
}

export {login}
```

然后这个时候再回过头来看上面 LoginPage.vue 中的 js 代码，可以发现登陆的方法已经写好了，然后基本上就可以通过直接使用axios来想后端发送登录请求了

跨域手动配置

因为当前是前后端分离项目，两个站点的地址不一样

在后端添加 filter/CorsFilter.java

```
// 解决跨域问题
@Component
@Order(Const.ORDER_CORS)
public class CorsFilter extends HttpFilter {

    @Value("${spring.web.cors.origin}")
    String origin;

    @Value("${spring.web.cors.credentials}")
    boolean credentials;

    @Value("${spring.web.cors.methods}")
```

```
    String methods;

    @Override
    protected void doFilter(HttpServletRequest request, HttpServletResponse response,
FilterChain chain) throws IOException, ServletException {
        this.addCorsHeader(request, response);
        chain.doFilter(request, response);
    }

    private void addCorsHeader(HttpServletRequest request,
                               HttpServletResponse response) {
        // 配置跨域，在这里可以只允许前端的地址可以访问
//        response.addHeader("Access-Control-Allow-Origin", "http://localhost:5173");
        // 这样是只要是从请求头中获取到的所有的请求地址都被允许通过
        response.addHeader("Access-Control-Allow-Origin", this.resolveOrigin(request));
        response.addHeader("Access-Control-Allow-Methods", this.resolveMethod());
        response.addHeader("Access-Control-Allow-Headers", "Authorization, Content-Type");
        if(credentials) {
            response.addHeader("Access-Control-Allow-Credentials", "true");
        }
    }

    /**
     * 解析配置文件中的请求方法
     * @return 解析得到的请求头值
     */
    private String resolveMethod(){
        return methods.equals("*") ? "GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE, PATCH" :
methods;
    }

    /**
     * 解析配置文件中的请求原始站点
     * @param request 请求
     * @return 解析得到的请求头值
     */
    private String resolveOrigin(HttpServletRequest request){
        return origin.equals("*") ? request.getHeader("Origin") : origin;
    }
}
```

在 utils/Const.java 中添加需要使用的参数

```
    public final static int ORDER_FLOW_LIMIT = -101;
    public final static int ORDER_CORS = -102;
```

退出登录

修改 net/index.js 添加新的请求封装方法，以及登出的方法，并将需要的方法都抛出

```
// 获取完整的请求头，用作在发送请求的时候用
```

```
function accessHeader() {
    const token = takeAccessToken();
    // 如果拿到了token就返回完整的token，如果没拿到就什么都不返回
    return token ? {'Authorization': `Bearer ${takeAccessToken()}`} : {}
}

// get封装
function get(url, success, failure = defaultFailure) {
    internalGet(url, accessHeader(), success, failure)
}

// post封装
function post(url, data, success, failure = defaultFailure) {
    internalPost(url, data, accessHeader(), success, failure)
}

// 登出
function logout(success, failure = defaultFailure) {
    get('/api/auth/logout', () => {
        // 删除当前颁发的 jwt
        deleteAccessToken()
        ElMessage.success("退出登录成功，欢迎您再次使用")
        success()
    }, failure)
}

// 判断是否未验证
function unauthorized() {
    return !takeAccessToken()
}

export {login, logout, get, post, unauthorized}
```

在 router/index.js 中添加新的主页路由 以及添加新的路由守卫来拒绝一些不合理的请求

```
import {createRouter, createWebHistory} from 'vue-router'
import {unauthorized} from "@/net/index.js";

// 定义路由规则，配置根路径 / 对应的组件为 WelcomeView.vue。
const router = createRouter({
    history: createWebHistory(import.meta.env.BASE_URL),
    routes: [
        {
            // 根路由
            path: '/',
            name: "welcome",
            component: () => import('@/views/WelcomeView.vue'),
            children: [
                {
                    path: '',
                    name: 'welcome-login',
                    component: () => import('@/views/welcome/LoginPage.vue')
                }
```

```
            ]
        } , {
            path: '/index',
            name: 'index',
            component: () => import('@/views/IndexView.vue')
        }
    ]
})

// 设置路由守卫，判断是否完成验证（已经登录了）
router.beforeEach((to, from, next) => {
    const isUnauthorized = unauthorized()
    // 如果用户已经登录了，并且又想进行登录界面
    // 已经登录并且已经被验证过
    if (to.name.startsWith("welcome-") && !isUnauthorized) {
        next("/index")
    } else if (to.fullPath.startsWith("/index") && isUnauthorized) {
        // 用户没有登录过，则不能访问/index
        next("/")
    } else {
        next()
    }

})

export default router
```

在 views 的目录下创建主页视图文件 IndexView.vue

```
<script setup>
import {logout} from "@/net/index.js";
import router from "@/router/index.js";

function userLogout() {
  logout(() => router.push('/'))
}
</script>

<template>
  <div>
    <el-button @click="userLogout">退出登录</el-button>
  </div>
</template>

<style scoped>

</style>
```

修改之前的 LoginPage.vue 中的登陆函数，在登录成功之后跳转到主页面

```
function userLogin() {
  formRef.value.validate((valid) => {
    if (valid) {
      login(form.username, form.password, form.remember, () => {router.push("/index")})
    }
  })
}
```

用户注册

在 application.yml 中配置发送邮件相关的信息，同时配置消息队列，放在 spring 级内：

```
  mail:
    host: smtp.163.com
    username: gustavhuang@163.com
    password: VBNXTLAZGZEUFQSN
```

在 AccountService.java 中添加有关发送注册邮件的方法

```
/**
 * 注册发送邮件
 * @param type 要发送的邮件的类型
 * @param email 邮箱
 * @param ip ip地址
 * @return
 */
String registerEmailVerifyCode(String type, String email, String ip);
```

然后在 AccountServiceImpl.java 中实现发送邮件的方法

```
    @Resource
    AmqpTemplate amqpTemplate;

    @Resource
    StringRedisTemplate stringRedisTemplate;

    @Resource
    FlowUtils flowUtils;

    // 读取配置文件中的用户名
    @Value("${spring.mail.username}")
    String username;

    @Resource
    JavaMailSender sender;



    // 注册，将验证码添加到redis当中，并直接发送邮件
    @Override
    public String registerEmailVerifyCode(String type, String email, String ip) {
```

```
        // 添加锁，防止同一时间内多个请求同时访问
        synchronized (ip.intern()) {
            // 如果当前请求的ip在等待时间内
            if (!this.verifyLimit(ip)) {
                // 如果没有通过验证
                return "请求频繁，请稍候再尝试!";
            }

            // 随机生成验证码
            Random random = new Random();
            int code = random.nextInt(899999) + 100000;

            // 将验证码存入 Redis，设置过期时间为 3 分钟
            stringRedisTemplate.opsForValue()
                    .set(Const.VERIFY_EMAIL_DATA + email, String.valueOf(code), 3,
TimeUnit.MINUTES);

            // 直接发送邮件
            SimpleMailMessage message = switch (type) {
                case "register" -> createMessage(
                        "欢迎注册我们的网站",
                        "您的邮件注册验证码为: " + code + ", 有效时间3分钟，为了保证您的安全，请勿向他
人泄露验证码",
                        email
                );
                case "reset" -> createMessage(
                        "您的密码重置邮件",
                        "您好，您正在进行重置密码操作，验证码为:  " + code + ", 有效时间3分钟，如非本
人操作，请无视! ",
                        email
                );
                default -> null;
            };

            if (message != null) {
                sender.send(message);
            }

            return null;
        }
    }

    // 要发送的消息实体
    private SimpleMailMessage createMessage(String title, String content, String email) {
        // 创建邮件发送对象
        SimpleMailMessage message = new SimpleMailMessage();
        // 发送的邮件的主题
        message.setSubject(title);
        // 发送的邮件的内容
        message.setText(content);
        // 接收人邮箱
        message.setTo(email);
        // 发送人
```

```
        message.setFrom(username);
        return message;
    }

    // 限制ip访问
    private boolean verifyLimit(String ip) {
        String key = Const.VERIFY_EMAIL_LIMIT + ip;
        return flowUtils.limitOnceCheck(key, 60);  // 限制60秒
    }
```

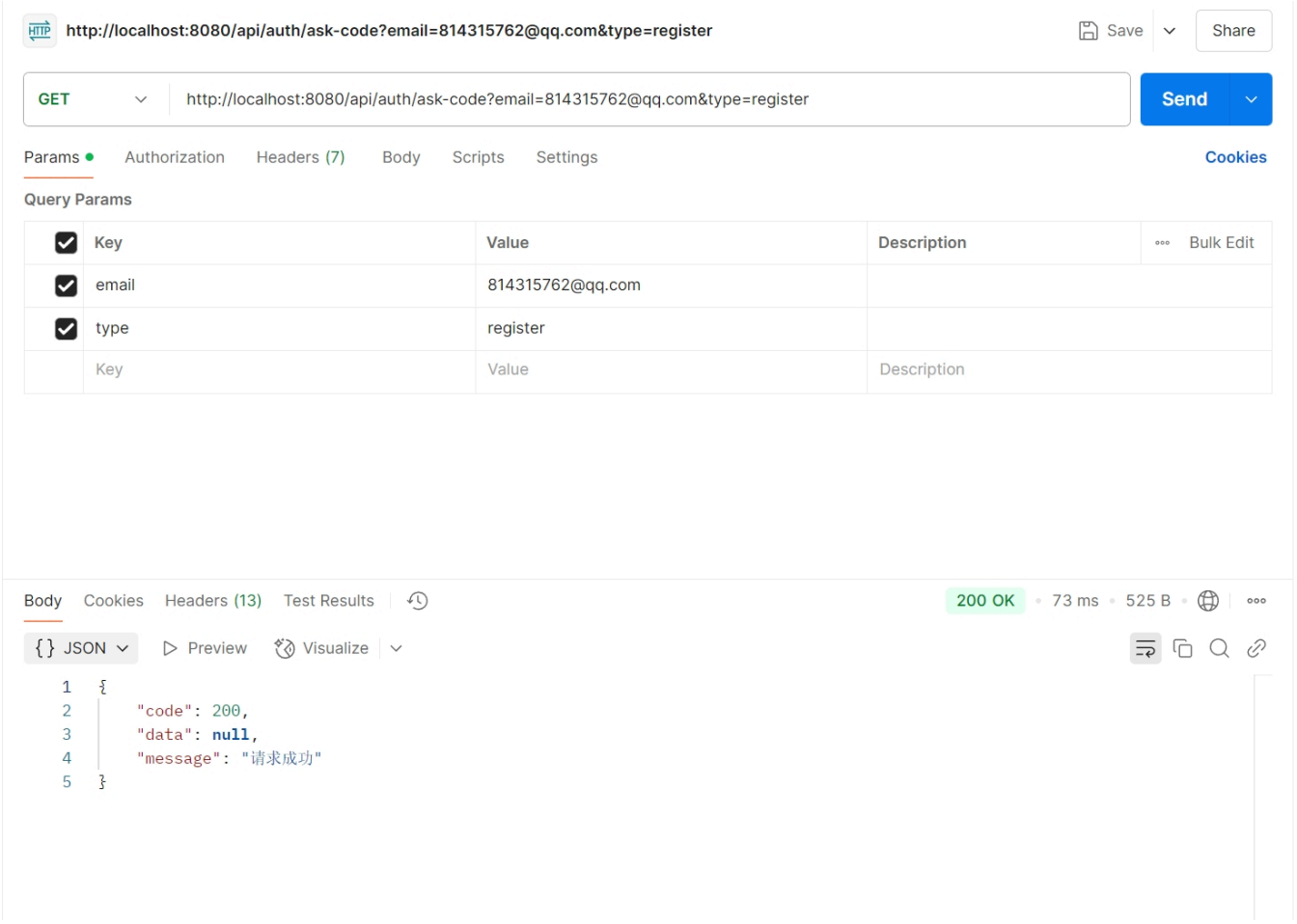
添加 config/FlowUtil.java 限流控制类

```
@Component
public class FlowUtils {

    // Redis
    @Resource
    StringRedisTemplate stringRedisTemplate;

    /**
     * 防止用户短时间内发送大量请求
     * @param key 存放在redis中的邮箱键
     * @param blockTime 阻拦重复请求的时间
     * @return
     */
    public boolean limitOnceCheck(String key, int blockTime) {
        if (Boolean.TRUE.equals(stringRedisTemplate.hasKey(key))) {
            return false;  // 当前用户在冷却时间中，阻止请求
        } else {
            // 不在冷却状态,则添加一个封禁的标志
            stringRedisTemplate.opsForValue().set(key, "", blockTime, TimeUnit.SECONDS);
        }
        return true;
    }
}
```

然后在 util/Const.java 中补全新引入的常量名

```
    public static final String VERIFY_EMAIL_LIMIT = "verify:email:limit";
    // 存储邮箱信息
    public static final String VERIFY_EMAIL_DATA = "verify:email:data";
```

因为我们在 SecurityConfig.java 中已经设置了允许所有 /api/auth/** 路径下的请求直接通过验证不需要 jwt，所以我们可以直接使用 postman 来进行测试



然后提供的邮箱中就能收到对应 registry 信息的消息

注册接口实现

上面的功能还有点小欠缺就是即使我们传入的邮箱地址是错误的，程序也会执行然后报错，所以最好是再加一个校验的步骤

在 pom.xml 中添加新的校验依赖

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

然后在 AuthorizeController.java 中添加新的校验注解

```
// 添加校验功能
@Validated
@RestController
@RequestMapping("/api/auth")
public class AuthorizeController {

    @Resource
    AccountService accountService;
```

```
    /**
     * 请求验证码
     * @param email 要发往的邮箱
     * @param type 要发送的邮件的类型
     * @param request 以及请求头信息
     * @return
     */
    // 现在需要额外校验输入的邮箱是否是合法的邮箱地址，以及当前输入的操作类型只能是注册或者重置密码
    @GetMapping("/ask-code")
    public RestBean<Void> askVerify(@RequestParam @Email String email,
                                    @RequestParam @Pattern(regexp = "(register | reset)")
String type,
                                    HttpServletRequest request) {
        // 根据提取到的ip地址进行请求次数限制
        String message = accountService.registerEmailVerifyCode(type, email,
request.getRemoteAddr());
        return message == null ? RestBean.success() : RestBean.failure(400, message);
    }
}
```

但是现在虽然错误能够拦截，但是仍然会在控制台输入大量的报错信息，所以最好的情况是能够添加一个捕捉错误的类，然后统一展示错误信息

不过在此之前，可以先将 "/error" 请求也从 SecurityConfig.java 中放行

```
.requestMatchers("/api/auth/**", "/error").permitAll()
```

然后新建一个 controller/exception/ValidationController.java 的类用来捕捉异常

```
// 日志打印
@Slf4j
// 对异常进行捕获，然后规范控制台输出格式
@RestControllerAdvice
public class ValidationController {

    // 捕获所有的 ValidationException ，然后进行统一的错误打印
    @ExceptionHandler(ValidationException.class)
    public RestBean<Void> validationException(ValidationException e) {
        log.warn("Resolve [{}: {}]", e.getClass().getName(), e.getMessage());
        return RestBean.failure(400, "请求参数有误");
    }

}
```

然后就是完成注册请求的接口设置

首先添加一个专门用来接受从前端发送的登陆请求的实体类，用来包含登录请求的信息，这个类我放在了 entity/request/EmailRegistryVO.java 中

```
// 请求的实体类
@Data
```

```
public class EmailRegisterVO {

    @Email
    @Length(min = 4)
    String email;

    @Length(min = 6, max = 6)
    String code;

    @Pattern(regexp = "^[a-zA-Z0-9\\u4e00-\\u9fa5]+$")
    @Length(min = 1, max = 10)
    String username;

    @Length(min = 6, max = 20)
    String password;

}
```

接下来在 AccountService.java 中添加操作方法

```
    /**
     * 后端接口调用注册邮件账户
     * @param vo 请求注册的信息实体类
     * @return
     */
    String registerEmailAccount(EmailRegisterVO vo);
```

然后在 AccountServiceImpl.java 中实现

```
    // 根据请求实体类的内容搭配邮箱和验证码进行校验
    @Override
    public String registerEmailAccount(EmailRegisterVO vo) {
        // 判断邮箱有没有被注册过
        // 判断是不是唯一的
        String email = vo.getEmail();
        String username = vo.getUsername();
        // 从redis中提取验证码
        String code = stringRedisTemplate.opsForValue().get(Const.VERIFY_EMAIL_DATA +
email);

        // 然后使用从redis中取出来的验证码和当前请求中的验证码进行比较
        if (code == null) {
            return "请先获取验证码";
        }
        if (!code.equals(vo.getCode())) {
            return "验证码输入错误";
        }
        if (this.existsAccountByEmail(email)) {
            return "此邮箱已经被其他用户注册，请重新选择邮箱";
        }
        if (this.existsAccountByUsername(username)) {
            return "此用户名已被其他人注册，请更换一个用户名";
```

```
        }
        // 将从前端获得的明文密码通过已经定义好的密码加密方式进行加密
        String password = passwordEncoder.encode(vo.getPassword());
        // 生成账号实体
        // 这里的id是自增的在数据库中
        Account account = new Account(null, vo.getUsername(), password, email, "user", new
Date());
        if (this.save(account)) {
            // 如果登录成功，可以将code从redis中删掉
            stringRedisTemplate.delete(Const.VERIFY_EMAIL_DATA + email);
            return null;
        } else {
            return "内部错误，请联系管理员!";
        }
    }

    // 判断当前提供的邮箱是否在数据库中存在
    private boolean existsAccountByEmail(String email) {
        return this.baseMapper.exists(Wrappers.<Account>query().eq("email", email));
    }

    // 判断当前提供的邮箱是否在数据库中存在
    private boolean existsAccountByUsername(String username) {
        return this.baseMapper.exists(Wrappers.<Account>query().eq("username", username));
    }
```

当然为了能够做到双重的保证，保证用户名和邮箱都不是重复地，可以额外在数据库中添加唯一键

最后在 AuthorizeController.java 中添加请求的接口

```
    // 前端发送注册请求
    @PostMapping("/register")
    public RestBean<Void> register(@RequestBody @Valid EmailRegisterVO vo) {
        // 通过将请求中携带的实体类传入后端的注册方法来进行消息处理
        return this.messageHandler(() ->
                accountService.registerEmailAccount(vo));

    }

    // 统一的消息处理
    private RestBean<Void> messageHandler(Supplier<String> action) {
        String message = action.get();
        // 如果是null代表成功，否则输出错误信息
        return message == null ? RestBean.success() : RestBean.failure(400, message);
    }
```

添加前端注册页面

在前端路由中添加注册页面

```
...
        {
```

```
            path: "/",
            name: "welcome",
            component: () => import('@/views/WelcomeView.vue'),
            children: [
                {
                    path: '',
                    name: 'welcome-login',
                    component: () => import('@/views/welcome/LoginPage.vue')
                },
                {
                    path: 'register',
                    name: 'welcome-register',
                    component: () => import('@/views/welcome/RegisterPage.vue')
                }
            ]
        },
...
```

然后在 views/welcome/RegistryPage.vue 中添加登陆页面的视图

```
<script setup>
import {computed, reactive, ref} from "vue";
import {EditPen, Lock, Message, User} from "@element-plus/icons-vue";
import router from "@/router/index.js";
import {ElMessage} from "element-plus";
import {get, post} from "@/net/index.js";

// 冷却时间
const coldTime = ref(0);

const formRef = ref()

const form = reactive({
  username: '',
  password: '',
  password_repeat: '',
  email: '',
  code: ''
})

// value是输入框内填的值
const validateUsername = (rule, value, callback) => {
  if (value === '') {
    callback(new Error('请输入用户名'))
  } else if (!/^[a-zA-Z0-9\u4e00-\u9fa5]+$/.test(value)) {
    callback(new Error('用户名不能包含特殊字符，只能是中英文'))
  } else {
    callback()
  }
}

const validatePassword = (rule, value, callback) => {
  if (value === '') {
```

```
    callback(new Error('请再次输入密码'))
  } else if (value !== form.password) {
    callback(new Error('两次输入的密码不一致'))
  } else {
    callback()
  }
}

// 校验规则
const rule = {
  // 判断用户名是否满足正则表达式
  username: [
    { validator: validateUsername, trigger: ['blur', 'change'] },
    { min: 1, max: 10, message: '用户名的长度必须在2-8个字符之间', trigger: ['blur', 'change'] }
  ],
  password: [
    { required: true, message: '请输入密码', trigger: ['blur'] },
    { min: 6, max: 20, message: '密码长队须在6-16个字符之间', trigger: ['blur', 'change'] }
  ],
  password_repeat: [
    { validator: validatePassword, trigger: ['blur', 'change'] }
  ],
  email: [
    { required: true, message: '请输入邮件地址', trigger: ['blur'] },
    { type: "email", message: '请输入合法的邮箱地址', trigger: ['blur', 'change'] }
  ],
  code: [
    { required: true, message: '请输入验证码', trigger: ['blur'] },
  ]
}

// 在请求验证码的时候要保证一分钟内只能发送一次
function askCode() {
  if (isEmailValid) {
    coldTime.value = 60;
    get(`/api/auth/ask-code?email=${form.email}&type=register`, () => {
      ElMessage.success(`验证码已经发送到邮箱: ${form.email}, 请注意查收`)
      setInterval(() => coldTime.value--, 1000)
    }, (message) => {
      ElMessage.warning(message)
      coldTime.value = 0;
    })
  } else {
    ElMessage.warning("请输入正确的电子邮件")
  }

}

// 判断邮箱是否正确
const isEmailValid = computed(() => /^[\w\.-]+@[\w\.-]+\.\w+$/.test(form.email))

function register() {
  formRef.value.validate((valid) => {
```

```
    if (valid) {
      post('/api/auth/register', {...form}, () => {
        ElMessage.success("注册成功，欢迎加入")
        router.push('/')
      })
    } else {
      ElMessage.warning("请完善表单内容")
    }
  })
}
</script>

<template>
  <div style="text-align: center;margin: 0 20px">
    <div style="margin-top: 100px">
      <div style="font-size: 25px;font-weight: bold">注册新用户</div>
      <div>欢迎注册我们的学习平台，请在下方填写相关信息</div>
    </div>
<!--表单-->
    <div style="margin-top: 50px">
      <el-form :model="form" :rules="rule" ref="formRef">
<!--        用户名-->
        <el-form-item prop="username">
          <el-input v-model="form.username" maxlength="10" type="text" placeholder="用户名">
            <template #prefix>
              <el-icon><User/></el-icon>
            </template>
          </el-input>
        </el-form-item>
<!--        密码-->
        <el-form-item prop="password">
          <el-input v-model="form.password" maxlength="20" type="password" placeholder="密
码">
            <template #prefix>
              <el-icon><Lock/></el-icon>
            </template>
          </el-input>
        </el-form-item>
<!--        重复密码-->
        <el-form-item prop="password_repeat">
          <el-input v-model="form.password_repeat" maxlength="20" type="password"
placeholder="重复密码">
            <template #prefix>
              <el-icon><Lock/></el-icon>
            </template>
          </el-input>
        </el-form-item>
<!--        邮箱-->
        <el-form-item prop="email">
          <el-input v-model="form.email" type="email" placeholder="邮箱账号">
            <template #prefix>
              <el-icon><Message/></el-icon>
            </template>
```

```
          </el-input>
        </el-form-item>

        <el-form-item prop="code">
          <el-row :gutter="10" style="width: 100%">
            <el-col :span="17">
              <el-input v-model="form.code" maxlength="6" type="text" placeholder="请输入验证
码">
                <template #prefix>
                  <el-icon><EditPen/></el-icon>
                </template>
              </el-input>
            </el-col>
            <el-col :span="7">
              <el-button @click="askCode" :disable="isEmailValid || coldTime"
type="success">
                {{ coldTime > 0 ? `请稍候${coldTime}秒`: '获取验证码' }}
              </el-button>
            </el-col>
          </el-row>
        </el-form-item>
      </el-form>

      <!-- 注册按钮 -->
      <div style="margin-top: 80px">
        <el-button style="width: 270px" type="warning" @click="register" plain>立即注册</el-
button>
      </div>

      <div style="margin-top: 20px">
        <span style="font-size: 14px;line-height: 15px;color:grey">已有账号?</span>
        <el-link style="translate: 0 -2px" @click="router.push('/')">立即登录</el-link>
      </div>
    </div>
  </div>


</template>

<style scoped>

</style>
```

并且修改登陆页面的注册按钮的请求方法

```
    <!-- 注册按钮 -->
    <el-divider>
      <span style="font-size: 13px;color: grey">没有账号</span>
    </el-divider>
    <div>
        <!-- 跳转到注册页面 -->
      <el-button @click="router.push('/register')" style="width: 270px" type="warning"
plain>立即注册</el-button>
    </div>
```

最后可以在 WelcomeView.vue 中设置页面跳转时呈现渐变效果

```
<!-- 右侧登陆与注册页面 -->
    <div class="right-card">
      <!--  添加淡入淡出的动态效果    -->
      <router-view v-slot="{ Component }">
        <transition name="el-fade-in-linear" mode="out-in">
          <component :is="Component"></component>
        </transition>
      </router-view>
    </div>
```

重置密码

首先为重置密码页面添加一个路由

```
routes: [
        {
            path: "/",
            name: "welcome",
            component: () => import('@/views/WelcomeView.vue'),
            children: [
                {
                    path: '',
                    name: 'welcome-login',
                    component: () => import('@/views/welcome/LoginPage.vue')
                },
                {
                    path: 'register',
                    name: 'welcome-register',
                    component: () => import('@/views/welcome/RegisterPage.vue')
                },
                {
                    path: 'reset',
                    name: 'welcome-reset',
                    component: () => import('@/views/welcome/ResetPage.vue')
                }
            ]
        },
...
```

然后编写重置密码的页面视图，在 views/welcome/ResetPage.vue 中

```
<script setup>
import {computed, reactive, ref} from "vue";
import {EditPen, Lock, Message} from "@element-plus/icons-vue";
import {ElMessage} from "element-plus";
import {get, post} from "@/net/index.js";
import router from "@/router/index.js";

const active = ref(0)
// 冷却时间
const coldTime = ref(0);

const formRef = ref()

const form = reactive({
  email: '',
  code: '',
  password: '',
  password_repeat: '',
})

// 在请求验证码的时候要保证一分钟内只能发送一次
function askCode() {
  if (isEmailValid) {
    coldTime.value = 60;
    get(`/api/auth/ask-code?email=${form.email}&type=register`, () => {
      ElMessage.success(`验证码已经发送到邮箱: ${form.email}, 请注意查收`)
      setInterval(() => coldTime.value--, 1000)
    }, (message) => {
      ElMessage.warning(message)
      coldTime.value = 0;
    })
  } else {
    ElMessage.warning("请输入正确的电子邮件")
  }

}

// 邮箱是否有效
const isEmailValid = computed(() => /^[\w\.-]+@[\w\.-]+\.\w+$/.test(form.email))

// value是输入框内填的值
const validatePassword = (rule, value, callback) => {
  if (value === '') {
    callback(new Error('请再次输入密码'))
  } else if (value !== form.password) {
    callback(new Error('两次输入的密码不一致'))
  } else {
    callback()
  }
}
```

```
// 校验规则
const rules = {
  email: [
    {required: true, message: '请输入邮件地址', trigger: 'blur'},
    {type: 'email', message: '请输入合法的电子邮件地址', trigger: ['blur', 'change']}
  ],
  code: [
    {required: true, message: '请输入获取的验证码', trigger: 'blur'},
  ],
  password: [
    {required: true, message: '请输入密码', trigger: 'blur'},
    {min: 6, max: 16, message: '密码的长度必须在6-16个字符之间', trigger: ['blur']}
  ],
  password_repeat: [
    {validator: validatePassword, trigger: ['blur', 'change']},
  ],
}


// 判断验证码是否正确
function confirmReset() {
  formRef.value.validate((valid) => {
    if (valid) {
      post('/api/auth/reset-confirm', {
        email: form.email,
        code: form.code
      }, () => active.value++)
    }
  })
}

function doReset() {
  formRef.value.validate((valid) => {
    if (valid) {
      post('/api/auth/reset-password', {...form}, () => {
        ElMessage.success("密码充重置成功，请重新登录")
        router.push('/')
      })
    }
  })
}

</script>

<template>
  <div style="text-align:center;">
    <div style="margin: 30px 20px">
      <el-steps :active="active" finish-status="success" align-center>
        <el-step title="验证电子邮件"/>
        <el-step title="重新设定密码"/>
      </el-steps>
    </div>
```

```
    <div style="margin: 0 20px" v-if="active === 0">
      <div style="margin-top: 80px">
        <div style="font-size: 25px;font-weight: bold">重置密码</div>
        <div style="font-size: 14px;color: grey">请输入需要重置密码的电子邮件地址</div>
      </div>

      <div style="margin-top: 50px">
        <el-form :model="form" :rules="rules" ref="formRef">
          <el-form-item prop="email">
            <el-input v-model="form.email" type="email" placeholder="邮箱账号">
              <template #prefix>
                <el-icon>
                  <Message/>
                </el-icon>
              </template>
            </el-input>
          </el-form-item>


          <el-form-item prop="code">
            <el-row :gutter="10" style="width: 100%">
              <el-col :span="17">
                <el-input v-model="form.code" :maxlength="6" type="text" placeholder="请输入
验证码">
                  <template #prefix>
                    <el-icon>
                      <EditPen/>
                    </el-icon>
                  </template>
                </el-input>
              </el-col>
              <el-col :span="5">
                <el-button @click="askCode" :disable="isEmailValid || coldTime"
type="success">
                  {{ coldTime > 0 ? `请稍候${coldTime}秒` : '获取验证码' }}
                </el-button>
              </el-col>
            </el-row>
          </el-form-item>
        </el-form>
      </div>
      <div style="margin-top: 70px">
        <el-button @click="confirmReset" style="width: 270px;" type="danger" plain>开始重置密
码</el-button>
      </div>
    </div>
    <div style="margin: 0 20px" v-if="active === 1">
      <div style="margin-top: 80px">
        <div style="font-size: 25px;font-weight: bold">重置密码</div>
        <div style="font-size: 14px;color: grey">请填写您的新密码，务必牢记，防止丢失</div>
      </div>

      <div style="margin-top: 50px">
```

```
        <el-form :model="form" :rules="rules" ref="formRef">
          <el-form-item prop="password">
            <el-input v-model="form.password" :maxlength="16" type="password"
placeholder="新密码">
              <template #prefix>
                <el-icon>
                  <Lock/>
                </el-icon>
              </template>
            </el-input>
          </el-form-item>
          <el-form-item prop="password_repeat">
            <el-input v-model="form.password_repeat" :maxlength="16" type="password"
placeholder="重复新密码">
              <template #prefix>
                <el-icon>
                  <Lock/>
                </el-icon>
              </template>
            </el-input>
          </el-form-item>
        </el-form>
      </div>
      <div style="margin-top: 70px">
        <el-button @click="doReset" style="width: 270px;" type="danger" plain>立即重置密码
</el-button>
      </div>
    </div>
  </div>
</template>

<style scoped>

</style>
```

然后在 LoginPage.vue 中修改忘记密码的链接请求

```
<el-link @click="router.push('/reset')">忘记密码？</el-link>
```

然后去修改后端，完成密码重置请求

具体方案：

- 在一阶段中，用户带着验证码请求对应接口，然后后端仅对验证码是否正确进行验证。第二阶段中用户填写新的密码之后，请求重置密码接口，不仅需要带上密码还要带上之前的验证码一起，然后再次验证验证码，如果正确，则重置密码

在 AccountService.java 中添加新的处理方法

```
    /**
     * 第一阶段，判断验证码是否正确
     * @param vo 请求
     * @return
     */
    String resetConfirm(ConfirmResetVO vo);

    /**
     * 第二阶段，重置密码
     * @param vo
     * @return
     */
    String resetEmailAccountPassword(EmailResetVO vo);
```

然后在 AccountServiceImpl.java 中实现

```
    // 确认验证码输入正确
    @Override
    public String resetConfirm(ConfirmResetVO vo) {
        String email = vo.getEmail();
        String code = stringRedisTemplate.opsForValue().get(Const.VERIFY_EMAIL_DATA +
email);
        if (code == null) return "请先获取验证码";
        if (!code.equals(vo.getCode())) {return "验证码错误，请重新输入!";}
        return null;
    }

    // 进行重置密码
    @Override
    public String resetEmailAccountPassword(EmailResetVO vo) {
        String email = vo.getEmail();
        String verify = this.resetConfirm(new ConfirmResetVO(vo.getEmail(), vo.getCode()));
        if (verify != null) return verify;
        String password = passwordEncoder.encode(vo.getPassword());
        boolean update = this.update().eq("email", email).set("password",
password).update();
        if (update) {
            stringRedisTemplate.delete(Const.VERIFY_EMAIL_DATA + email);
        }
        return null;
    }
```

添加接口

```
    // 注册
    @PostMapping("/register")
    public RestBean<Void> register(@RequestBody @Valid EmailRegisterVO vo) {
        return this.messageHandler(vo, accountService::registerEmailAccount);
    }

    @PostMapping("/reset-confirm")
```

```java
    public RestBean<Void> resetConfirm(@RequestBody @Valid ConfirmResetVO vo) {
        return this.messageHandler(vo, accountService::resetConfirm);
    }

    @PostMapping("/reset-password")
    public RestBean<Void> resetConfirm(@RequestBody @Valid EmailResetVO vo) {
        return this.messageHandler(vo, accountService::resetEmailAccountPassword);
    }

    private <T> RestBean<Void> messageHandler(T vo, Function<T, String> function) {
        return messageHandler(() -> function.apply(vo));
    }
```