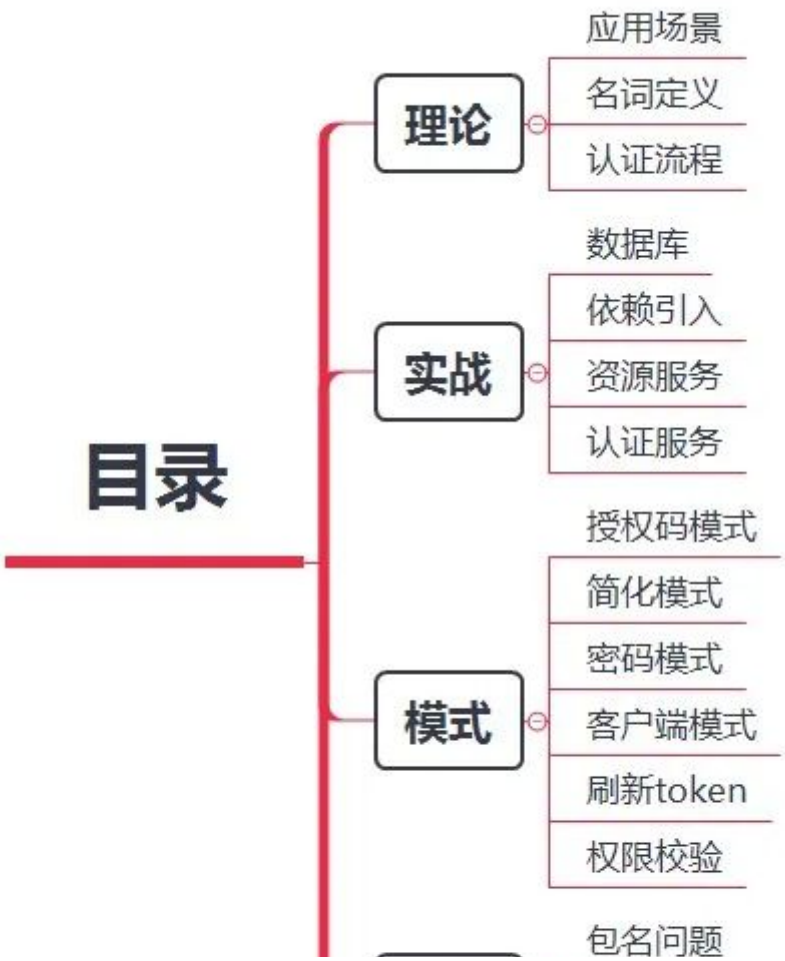
学会 OAuth2.0 授权登录，10 张图就够了

Java充电社 2023-02-10 08:06 发表于上海

以下文章来源于阿Q说代码 ，作者阿Q

您好，我是路人，更多优质文章见个人博客：http://itsoku.com

对于身份认证和用户授权，之前写过几篇关于Shiro和Security的文章。从发送口令获取源码的反馈来看，大家还是比较认可的。今天给大家带来一种新的授权方式：`oauth2` 。



理论

OAuth 是一个关于授权（ authorization ）的开放网络标准，用来授权第三方应用获取用户数据，是目前最流行的授权机制，它当前的版本是2.0。

应用场景

假如你正在“网站A”上冲浪，看到一篇帖子表示非常喜欢，当你情不自禁的想要点赞时，它会提示你进行登录操作。

打开登录页面你会发现，除了最简单的账户密码登录外，还为我们提供了微博、微信、QQ等快捷登录方式。假设选择了快捷登录，它会提示我们扫码或者输入账号密码进行登录。

登录成功之后便会将QQ/微信的昵称和头像等信息回填到“网站A”中，此时你就可以进行点赞操作了。

名词定义

在详细讲解 `oauth2` 之前，我们先来了解一下它里边用到的名词定义吧：

- Client：客户端，它本身不会存储用户快捷登录的账号和密码，只是通过资源拥有者的授权去请求资源服务器的资源，即例子中的网站A；
- Resource Owner：资源拥有者，通常是用户，即例子中拥有QQ/微信账号的用户；
- Authorization Server：认证服务器，可以提供身份认证和用户授权的服务器，即给客户端颁发 `token` 和校验 `token` ；
- Resource Server：资源服务器，存储用户资源的服务器，即例子中的QQ/微信存储的用户信息；

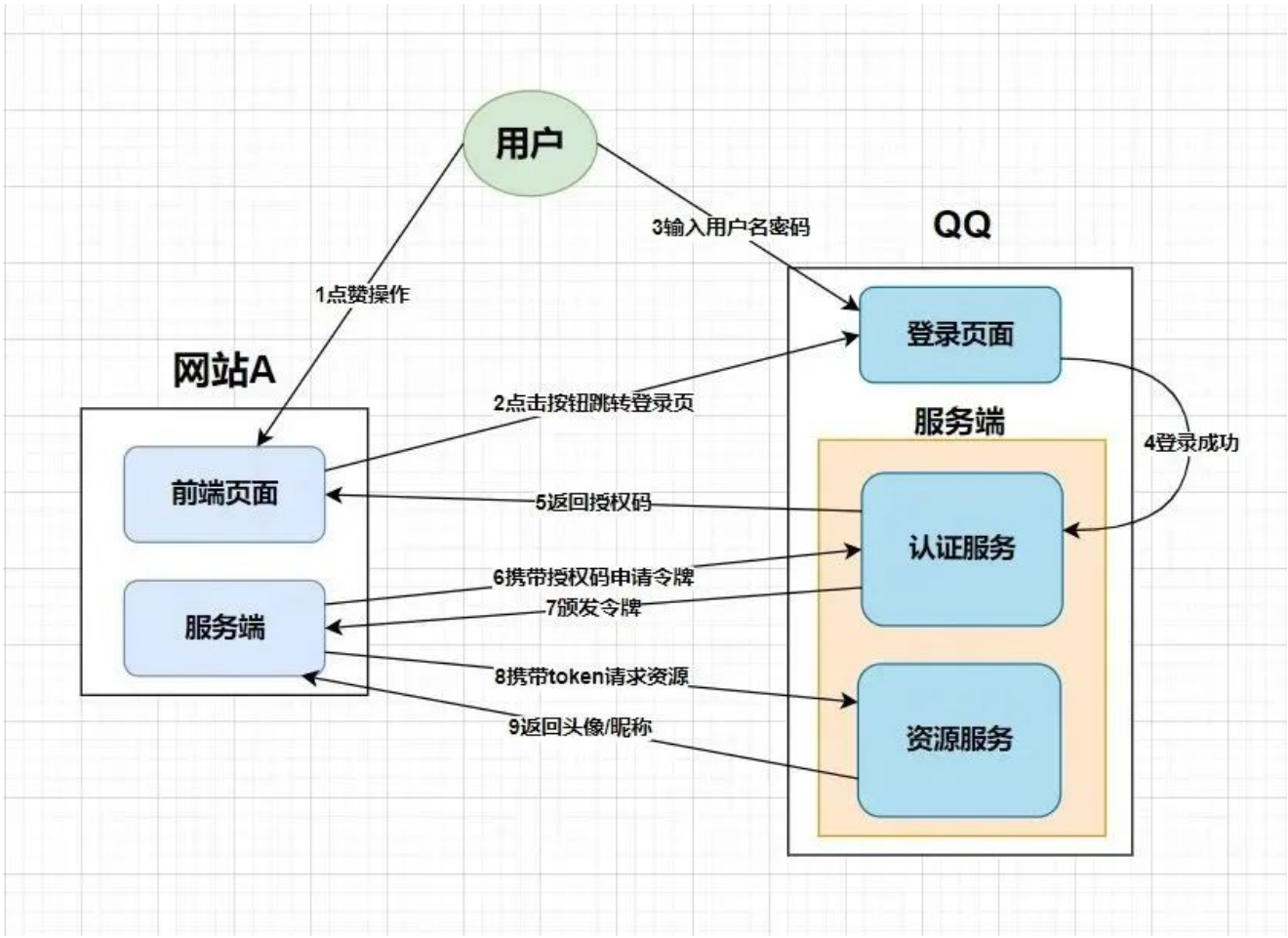
认证流程

```
|         |                                   +---------------+
|         |--(E)----- Access Token ------>|    Resource   |
|         |                               |     Server    |
|         |<-(F)--- Protected Resource ---|               |
+---------+                               +---------------+
```

如图是 `oauth2` 官网的认证流程图，我们来分析一下：

- A客户端向资源拥有者发送授权申请；
- B资源拥有者同意客户端的授权，返回授权码；
- C客户端使用授权码向认证服务器申请令牌 `token` ；
- D认证服务器对客户端进行身份校验，认证通过后发放令牌；
- E客户端拿着认证服务器颁发的令牌去资源服务器请求资源；
- F资源服务器校验令牌的有效性，返回给客户端资源信息；

为了大家更好的理解，阿Q特地画了一张图：



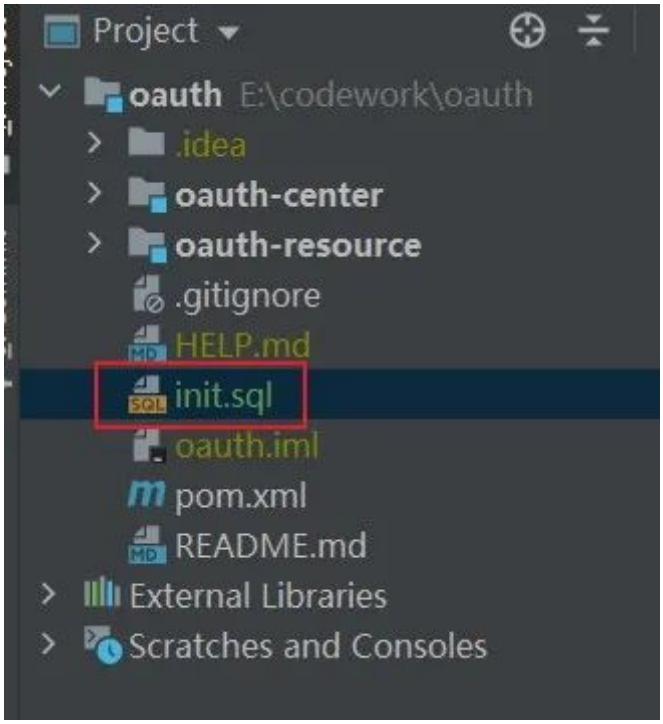
到这儿，相信大家对理论知识已经掌握的差不多了，接下来我们就进入实战训练吧。

实战

在正式开始搭建项目之前我们先来做一些准备工作：要想使用 `oauth2` 的服务，我们得先创建几张表。

数据库

`oauth2` 相关的建表语句可以参考官方初始化sql，也可以查看阿Q项目中的**init.sql**文件，回复“oauth2”获取源码。



至于表结构，大家可以先大体了解下，其中字段的含义，在**init.sql**文件中阿Q已经做了说明。

- oauth_client_details：存储客户端的配置信息，操作该表的类主要是 `JdbcClientDetailsService.java` ；
- oauth_access_token：存储生成的令牌信息，操作该表的类主要是 `JdbcTokenStore.java` ；
- oauth_client_token：在客户端系统中存储从服务端获取的令牌数据，操作该表的类主要是 `JdbcClientDetailsService.java` ；
- oauth_code：存储授权码信息与认证信息，即只有 `grant_type` 为 `authorization_code` 时，该表才会有数据，操作该表的类主要是 `JdbcAuthorizationCodeServices.java` ；
- oauth_approvals：存储用户的授权信息；
- oauth_refresh_token：存储刷新令牌的 `refresh_token` ，如果客户端的 `grant_type` 不支持 `refresh_token` ，那么不会用到这张表，操作该表的类主要是 `JdbcTokenStore` ；

在 `oauth_client_details` 表中添加一条数据

```
client_id:cheetah_one //客户端名称，必须唯一
resource_ids:product_api //客户端所能访问的资源id集合,多个资源时用逗号(,)分隔
client_secret:$2a$10$h/TmLPvXozJJHXDyJEN22ensJgaciomfpOc9js9OonwWIdAnRQeoi //客户端的访问密码
scope:read,write //客户端申请的权限范围,可选值包括read,write,trust。若有多个权限范围用逗号(,)分隔
authorized_grant_types:client_credentials,implicit,authorization_code,refresh_token,password //指
web_server_redirect_uri:http://www.baidu.com //客户端的重定向URI,可为空, 当grant_type为authorizatio
access_token_validity:43200 //设定客户端的access_token的有效时间值(单位:秒),可选, 若不设定值则使用默认
autoapprove:false //设置用户是否自动Approval操作, 默认值为 'false', 可选值包括 'true','false', 'read'
```

数据库中对密码进行了加密处理，大家可以在此路径下自行生成

用户角色相关的表也在**init.sql**文件中，表结构非常简单，大家自行查阅。我的初始化数据为

依赖引入

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
```

```
 <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.security</groupId>
 <artifactId>spring-security-jwt</artifactId>
</dependency>
```

至于其它依赖，大家可以根据需要自行引入，不再赘述，回复“oauth2”获取源码。

资源服务

配置文件对服务端口、应用名称、数据库、 `mybatis` 和日志进行了配置。

写了一个简单的控制层代码，用来模拟资源访问

```
@RestController
@RequestMapping("/product")
public class ProductController {

    @GetMapping("/findAll")
    public String findAll(){
        return "产品列表查询成功";
    }
}
```

接着创建配置类继承 `ResourceServerConfigurerAdapter` 并增加 `@EnableResourceServer` 注解开启资源服务，重写两个 `configure` 方法

```
/**
 * 指定token的持久化策略
 * InMemoryTokenStore 表示将token存储在内存中
 * RedisTokenStore 表示将token存储在redis中
 * JdbcTokenStore 表示将token存储在数据库中
 * @return
 */
@Bean
public TokenStore jdbcTokenStore(){
```

```
    return new JdbcTokenStore(dataSource);
}

/**
 * 指定当前资源的id和token的存储策略
 * @param resources
 * @throws Exception
 */
@Override
public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
    //此处的id可以写在配置文件中，这里我们先写死
 resources.resourceId("product_api").tokenStore(jdbcTokenStore());
}


/**
 * 设置请求权限和header处理
 * @param http
 * @throws Exception
 */
@Override
public void configure(HttpSecurity http) throws Exception {
 //固定写法
 http.authorizeRequests()
   //指定不同请求方式访问资源所需的权限，一般查询是read，其余都是write
   .antMatchers(HttpMethod.GET,"/**").access("#oauth2.hasScope('read')")
   .antMatchers(HttpMethod.POST,"/**").access("#oauth2.hasScope('write')")
   .antMatchers(HttpMethod.PATCH,"/**").access("#oauth2.hasScope('write')")
   .antMatchers(HttpMethod.PUT,"/**").access("#oauth2.hasScope('write')")
   .antMatchers(HttpMethod.DELETE,"/**").access("#oauth2.hasScope('write')")
   .and()
   .headers().addHeaderWriter((request,response) -> {
    //域名不同或者子域名不一样并且是ajax请求就会出现跨域问题
    //允许跨域
    response.addHeader("Access-Control-Allow-Origin","*");
    //跨域中会出现预检请求，如果不能通过，则真正请求也不会发出
    //如果是跨域的预检请求，则原封不动向下传递请求头信息，否则预检请求会丢失请求头信息（主要是token信息）
    if(request.getMethod().equals("OPTIONS")){
     response.setHeader("Access-Control-Allow-Methods",request.getHeader("Access-Control-Allow-Me
     response.setHeader("Access-Control-Allow-Headers",request.getHeader("Access-Control-Allow-He
    }
 });
}
```

当然我们也可以配置忽略校验的 `url` ，在上边的 `public void configure(HttpSecurity http) throws Exception` 中进行配置

```
ExpressionUrlAuthorizationConfigurer<HttpSecurity>
  .ExpressionInterceptUrlRegistry config = http.requestMatchers().anyRequest()
  .and()
  .authorizeRequests();
properties.getUrls().forEach(e -> {
 config.antMatchers(e).permitAll();
});
```

因为我们是需要进行校验的，所以我把对应的代码给注释掉了，大家可以回复“oauth2”下载源码自行查看。

然后将实现了 `UserDetails` 的 `SysUser` 和实现了 `GrantedAuthority` 的 `SysRole` 放到项目中，当请求发过来时， `oauth2` 会帮我们自行校验。

认证服务

配置文件对服务端口、应用名称、数据库、 `mybatis` 和日志进行了配置。

Security配置

还是和之前**Security+JWT组合拳**的配置大同小异，不了解的可以先看下该文。

①将继承了 `UserDetailsService` 的 `ISysUserService` 的实现类 `SysUserServiceImpl` 重写 `loadUserByUsername` 方法

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
 return this.baseMapper.selectOne(new LambdaQueryWrapper<SysUser>().eq(SysUser::getUsername, user
}
```

②继承 `WebSecurityConfigurerAdapter` 类，增加 `@EnableWebSecurity` 注解并重写方法

```
/**
```

```
     * 指定认证对象的来源和加密方式
     * @param auth
     * @throws Exception
     */
    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder());
    }

    /**
     * 安全拦截机制（最重要）
     * @param httpSecurity
     * @throws Exception
     */
    @Override
    public void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
                //CSRF禁用，因为不使用session
                .csrf().disable()
                .authorizeRequests()
                //登录接口和静态资源不需要认证
                .antMatchers("/login*","/css/*").permitAll()
                //除上面的所有请求全部需要认证通过才能访问
                .anyRequest().authenticated()
                //返回HttpSecurity以进行进一步的自定义,证明是一次新的配置的开始
                .and()
                .formLogin()
                //如果未指定此页面，则会跳转到默认页面
    //                .loginPage("/login.html")
                .loginProcessingUrl("/login")
                .permitAll()
                //认证失败处理类
                .failureHandler(customAuthenticationFailureHandler);
    }

    /**
     * AuthenticationManager 对象在OAuth2.0认证服务中要使用，提前放入IOC容器中
     * @return
     * @throws Exception
     */
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
```

```
 return super.authenticationManagerBean();
}
```

AuthorizationServer配置

①继承 `AuthorizationServerConfigurerAdapter` 类，增加 `@EnableAuthorizationServer` 注解开启认证服务

②依赖注入，注入7个实例 `Bean` 对象

```
/**
 * 数据库连接池对象
 */
private final DataSource dataSource;

/**
 * 认证业务对象
 */
private final ISysUserService userService;

/**
 * 授权码模式专用对象
 */
private final AuthenticationManager authenticationManager;

/**
 * 客户端信息来源
 * @return
 */
@Bean
public JdbcClientDetailsService jdbcClientDetailsService(){
   return new JdbcClientDetailsService(dataSource);
}

/**
 * token保存策略
 * @return
 */
@Bean
public TokenStore tokenStore(){
 return new JdbcTokenStore(dataSource);
```

```
 return new JdbcTokenStore(dataSource);
}

/**
 * 授权信息保存策略
 * @return
 */
@Bean
public ApprovalStore approvalStore(){
 return new JdbcApprovalStore(dataSource);
}

/**
 * 授权码模式数据来源
 * @return
 */
@Bean
public AuthorizationCodeServices authorizationCodeServices(){
 return new JdbcAuthorizationCodeServices(dataSource);
}
```

③重写方法进行配置

```
/**
 * 用来配置客户端详情服务（ClientDetailsService）
 * 客户端详情信息在这里进行初始化
 * 指定客户端信息的数据库来源
 * @param clients
 * @throws Exception
 */
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
 clients.withClientDetails(jdbcClientDetailsService());
}

/**
 * 检测 token 的策略
 * @param security
 * @throws Exception
 */
@Override
```

```
public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {
 security
   //允许客户端以form表单的方式将token传达给我们
   .allowFormAuthenticationForClients()
   //检验token必须需要认证
   .checkTokenAccess("isAuthenticated()");
}


/**
 * OAuth2.0的主配置信息
 * @param endpoints
 * @throws Exception
 */
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
 endpoints
   //刷新token时会验证当前用户是否已经通过认证
   .userDetailsService(userService)
   .approvalStore(approvalStore())
   .authenticationManager(authenticationManager)
   .authorizationCodeServices(authorizationCodeServices())
   .tokenStore(tokenStore());
}
```

其它关于用户表和权限表的代码可参考源码，回复“oauth2”获取源码。

模式

授权码模式

我们前边所讲的内容都是基于授权码模式，授权码模式被称为最安全的一种模式，它获取令牌的操作是在两个服务端进行的，极大的减小了令牌泄漏的风险。

启动两个服务，当我们再次请求 `127.0.0.1:9002/product/findAll` 接口时会提示以下错误

```
{
    "error": "unauthorized",
    "error_description": "Full authentication is required to access this resource"
}
```

①调用接口获取授权码

发送 `127.0.0.1:9001/oauth/authorize?response_type=code&client_id=cheetah_one` 请求，前边的路径是固定形式的， `response_type=code` 表示获取授权码， `client_id=cheetah_one` 表示客户端的名称是我们数据库配置的数据。

该页面是 `oauth2` 的默认页面，输入用户的账户密码点击登录会提示我们进行授权，这是数据库 `oauth_client_details` 表我们设置 `autoapprove` 为 `false` 起到的效果。

选择 `Approve` 点击 `Authorize` 按钮，会发现我们设置的回调地址（ `oauth_client_details` 表中的 `web_server_redirect_uri` ）后边拼接了 `code` 值，该值就是授权码。

查看数据库发现 `oauth_approvals` 和 `oauth_code` 表已经存入数据了。

拿着授权码去获取 `token`

获取到 `token` 之后 `oauth_access_token` 和 `oauth_refresh_token` 表中会存入数据以用于后边的认证。而 `oauth_code` 表中的数据被清除了，这是因为 `code` 值是直接暴漏在网页链接上的， `oauth2` 为了防止他人拿到 `code` 非法请求而特意设置为仅用一次。

拿着获取到的 `token` 去请求资源服务的接口，此时有两种请求方式

接下来我们再来看一下 `oauth2` 的其它模式。

简化模式

所谓简化模式是针对授权码模式进行的简化，它将授权码模式中获取授权码的步骤省略了，直接去请求获取 `token` 。

流程：发送请求 `127.0.0.1:9001/oauth/authorize?response_type=token&client_id=cheetah_one` 跳转到登录页进行登录， `response_type=token` 表示获取 `token` 。

输入账号密码登录之后会直接在浏览器返回 `token` ，我们就可以像授权码方式一样携带 `token` 去请求资源了。

该模式的弊端就是 `token` 直接暴漏在浏览器中，非常不安全，不建议使用。

密码模式

密码模式下，用户需要将账户和密码提供给客户端向认证服务器申请令牌，所以该种模式需要用户高度信任客户端。

流程：请求如下

获取成功之后可以去访问资源了。

客户端模式

客户端模式已经不太属于 `oauth2` 的范畴了，用户直接在客户端进行注册，然后客户端去认证服务器获取令牌时不需要携带用户信息，完全脱离了用户，也就不存在授权问题了。

发送请求如下

获取成功之后可以去访问资源了。

刷新**token**

权限校验

除了我们在数据库中为客户端配置资源服务外，我们还可以动态的给用户分配接口的权限。

①开启 `Security` 内置的动态配置

在开启资源服务时给 `ResourceServerConfig` 类增加注解 `@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)`

②给接口增加权限

```
@GetMapping("/findAll")
@Secured("ROLE_PRODUCT")
public String findAll(){
    return "产品列表查询成功";
}
```

③在用户登录时设置用户权限

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
 SysUser sysUser = this.baseMapper.selectOne(new LambdaQueryWrapper<SysUser>().eq(SysUser::getUse
 sysUser.setRoleList(AuthorityUtils.commaSeparatedStringToAuthorityList("ROLE_PRODUCT"));
 return sysUser;
}
```

然后测试会发现可以正常访问。

采坑

包名问题

当我在创建项目的时候，给 `product` 和 `server` 两个模块设置了不同的包名，导致发送请求获取资源时报错。

经过分析得知，在登录账号时会将用户的信息存储到 `oauth_access_token` 表的 `authentication` 中，在进行 `token` 校验时会根据 `token_id` 取出该字段进行反序列化，如果此时发现包名不一致便会导致解析 `token` 失败，因此请求资源失败。

解决思路

- 两个项目的包名改为一致；
- 可以将用户和权限的实体抽成单独的模块，供其它模块引用；
- `loadUserByUsername` 方法中使用的用户实体类不需要继承 `UserDetailsService` 类，每次返回时用 `user` 类包装一下即可；

数据库问题

当我在进行权限校验测试时，在设置权限时发现少打了一个单词，导致请求一直出错。修改完成之后继续请求，仍提示权限不足。

于是我将数据库中 `oauth_refresh_token` 和 `oauth_access_token` 的数据清除，重新开始测试就可以了。

个人认为是生成 `token` 时发现数据库中 `token` 存在，故不刷新 `token` ，但进行校验时却用带有权限标识的 `token` 前去校验导致失败。

至于其它的小坑在这不再赘述，如果遇到问题，建议按照流程对比我的源码仔细检查，回复“oauth2”获取源码。

小结

本文从原理、应用场景、认证流程出发，对 `oauth2` 进行了基本的讲解，并且手把手带大家完成了项目的搭建。大家在对授权码模式、简化模式、密码模式、客户端模式进行测试的同时要将重点放到授权码模式上。

•••••••••• **END** ••••••••••••

↓↓↓ 点击阅读原文，直达个人博客

你在看吗

阅读原文

喜欢此内容的人还喜欢

“劝吸毒情侣结婚”、“替弃养夫妇寻女”：这些“阴间新闻”，到底有多恶心？

桌子的生活观

小巧、零依赖的工具库！

前端实验室

瞧瞧人家，那后端API接口写得，那叫一个优雅

程序员不圆