

OAuth2

一. OAuth2 介绍

OAuth 2.0是一种授权机制，用来授权第三方应用，获取用户数据。如第三方网站使用QQ、微信快捷登录。

1.1 Spring Security

使用OAuth2.0之前需要了解Spring Security的基本使用。

Spring Security 是 Spring 家族中的一个安全管理框架，实际上，在 Spring Boot 出现之前，Spring Security 就已经出现了，但是使用的并不多，一直都是 Shiro居多。

因为相比较Shiro，在SSM/SSH框架中整合Spring Security比较麻烦。虽然Spring Security功能比Shiro强大，但是使用却没有Shiro多。Spring Boot 出现之后，Spring Boot 对于 Spring Security 提供了 自动化配置方案，可以零配置使用 Spring Security。

一般来说，常见的安全管理技术栈的组合：

- SSM + Shiro
- Spring Boot/Spring Cloud + Spring Security

实际使用中，任意组合都是可以的。

1.1.1 Spring Security简单使用

使用Spring Boot框架。

1. Maven依赖

SpringBoot可以使用

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

SpringCloud可以使用

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

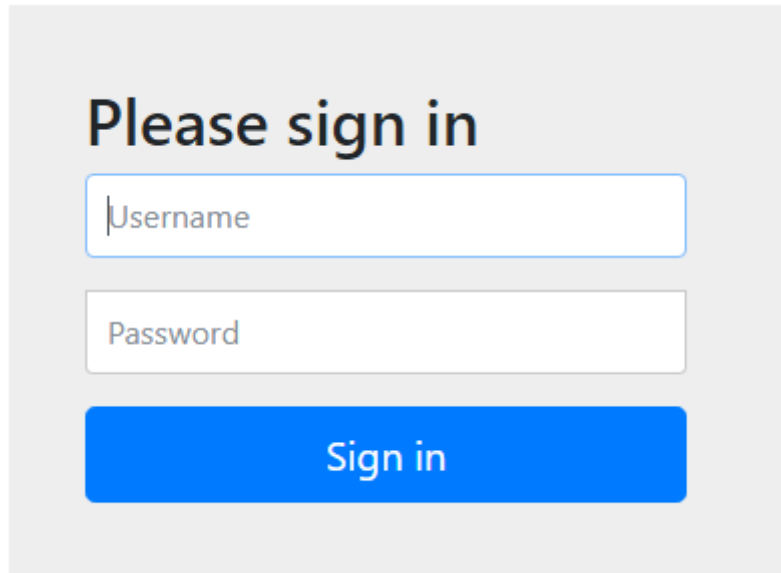
添加依赖后，默认情况下项目中所有接口都会被保护起来（如果用户自定义了 `WebSecurityConfigurerAdapter` 类则会按照自定义规则）。

2. 创建Controller

```
@RestController
public class HelloController {
    @GetMapping("/test")
    public String hello() {
        return "Spring Security!";
    }
}
```

Spring Security支持form表单和HttpBasic两种认证方式。

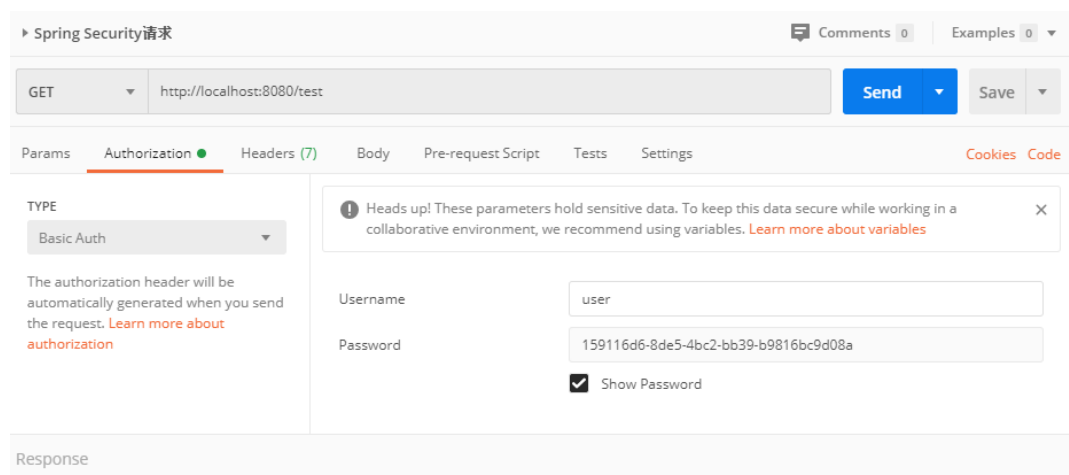
1. 浏览器直接访问 `/test`，显示需要登录。



访问 `/test` 时，服务端会返回 302 响应码，客户端重定向到 `/login` 页面，用户在登录页面登录成功后，会重新跳转回 `/test` 接口。

2. POSTMAN 使用：

请求头中添加Authorization认证，避免重定向到登录页面。



1.1.2 用户名配置

默认情况下，登陆的用户名是 `user`，密码在项目启动时随机生成，在控制台日志中查看。

```

2020-04-13 13:08:53.934 WARN 14200 --- [main] c.n.c.sources.URLConfigurationSource : No URLs will be polled as dynamic
2020-04-13 13:08:53.934 INFO 14200 --- [main] c.n.c.sources.URLConfigurationSource : To enable URLs as dynamic configu
2020-04-13 13:08:53.944 INFO 14200 --- [main] c.netflix.config.DynamicPropertyFactory : DynamicPropertyFactory is initial
2020-04-13 13:08:55.278 INFO 14200 --- [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath ba
2020-04-13 13:08:55.425 INFO 14200 --- [main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 159116d6-8de5-4bc2-bb39-b9816bc9d08a

2020-04-13 13:08:55.603 INFO 14200 --- [main] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: any request
2020-04-13 13:08:55.616 INFO 14200 --- [main] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: Ant [patte
2020-04-13 13:08:55.641 INFO 14200 --- [main] pertySourcedRequestMappingHandlerMapping : Mapped URL path [/v2/api-docs] or
2020-04-13 13:08:55.690 WARN 14200 --- [main] c.n.c.sources.URLConfigurationSource : No URLs will be polled as dynamic
2020-04-13 13:08:55.690 INFO 14200 --- [main] c.n.c.sources.URLConfigurationSource : To enable URLs as dynamic configu

```

随机生成的密码每次都会变更，实际开发中需要我们配置用户名密码。

1. 在application.yaml中配置

```

spring:
  security:
    user:
      name: user
      password: 123456

```

2. 通过Java代码配置（数据库中加载、储存在内存中）

创建一个 Spring Security 的配置类，继承 `WebSecurityConfigurerAdapter` 类

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        //下面这两行配置表示在内存中配置了两个用户
        auth.inMemoryAuthentication()
            .withUser("admin").roles("admin")

            .password("$2a$10$GstfEJEyoSHiSxnoP3SbD.R8XRowPlQKodi.N6/iFEWEJWtQqlSba")
                .and()
                .withUser("user").roles("user")

            .password("$2a$10$GstfEJEyoSHiSxnoP3SbD.R8XRowPlQKodi.N6/iFEWEJWtQqlSba");
    }
    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

```

注意：

1. 此处需要注入 `@Bean` `BCryptPasswordEncoder`，不然控制台会报错，无法注入 `BCryptPasswordEncoder` 类。
2. 配置多用户时，需要加入 `and()` 分隔。

关于 `and()`

在没有 Spring Boot 的时候，我们都是 SSM 中使用 Spring Security，这种时候都是在 XML 文件中配置 Spring Security，既然是 XML 文件，标签就有开始有结束，现在的 `and` 符号相当于就是 XML 标签的结束符，表示结束当前标签，这是个时候上下文会回到 `inMemoryAuthentication` 方法中，然后开启新用户的配置。

这样就配置了两个用户，密码是123456加密后的字符串。Spring5以后强制要求加密，可以使用过期的PasswordEncoder 的实例 NoOpPasswordEncoder来避免密码加密，但是一般情况下都会要求密码加密，即使用BCryptPasswordEncoder加密。

1.1.3 登录配置

对于登录接口，登录成功后的响应，登录失败后的响应，我们都可以在WebSecurityConfigurerAdapter 的实现类中进行配置。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    VerifyCodeFilter verifyCodeFilter;
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterBefore(verifyCodeFilter,
            UsernamePasswordAuthenticationFilter.class);
        http
            //开启登录配置
            .authorizeRequests()
            //表示访问 /hello 这个接口，需要具备 admin 这个角色
            .antMatchers("/test").hasRole("admin")
            //表示剩余的其他接口，登录之后就能访问
            .anyRequest().authenticated()
            .and()
            .formLogin()
            //自定义登录页面，未登录时，访问一个需要登录之后才能访问的接口，会自动跳转到该页面
            .loginPage("/login.html")
            //登录处理接口
            .loginProcessingUrl("/doLogin")
            //定义登录时，用户名的 key，默认为 username
            .usernameParameter("uname")
            //定义登录时，用户密码的 key，默认为 password
            .passwordParameter("passwd")
            //登录成功的处理器
            .successHandler(new AuthenticationSuccessHandler() {
                @Override
                public void onAuthenticationSuccess(HttpServletRequest req,
                    HttpServletResponse resp, Authentication authentication) throws IOException,
                    ServletException {
                    resp.setContentType("application/json;charset=utf-8");
                    PrintWriter out = resp.getWriter();
                    out.write("success");
                    out.flush();
                }
            })
            //登录失败的处理器
            .failureHandler(new AuthenticationFailureHandler() {
                @Override
                public void onAuthenticationFailure(HttpServletRequest req,
                    HttpServletResponse resp, AuthenticationException exception) throws IOException,
                    ServletException {
                    resp.setContentType("application/json;charset=utf-8");
                    PrintWriter out = resp.getWriter();
                    out.write("fail");
                    out.flush();
                }
            })
    }
}
```

```

    })
    .permitAll()//和表单登录相关的接口统统都直接通过
    .and()
    .logout()
    .logoutUrl("/logout")
    //登出成功的处理器
    .logoutSuccessHandler(new LogoutSuccessHandler() {
        @Override
        public void onLogoutSuccess(HttpServletRequest req,
        HttpServletResponse resp, Authentication authentication) throws IOException,
        ServletException {
            resp.setContentType("application/json;charset=utf-8");
            PrintWriter out = resp.getWriter();
            out.write("logout success");
            out.flush();
        }
    })
    .permitAll()
    .and()
    .httpBasic()
    .and()
    //关闭csrf跨域拦截
    .csrf().disable();
}
}

```

如果某一个请求地址不需要拦截的话，可以配置忽略

```

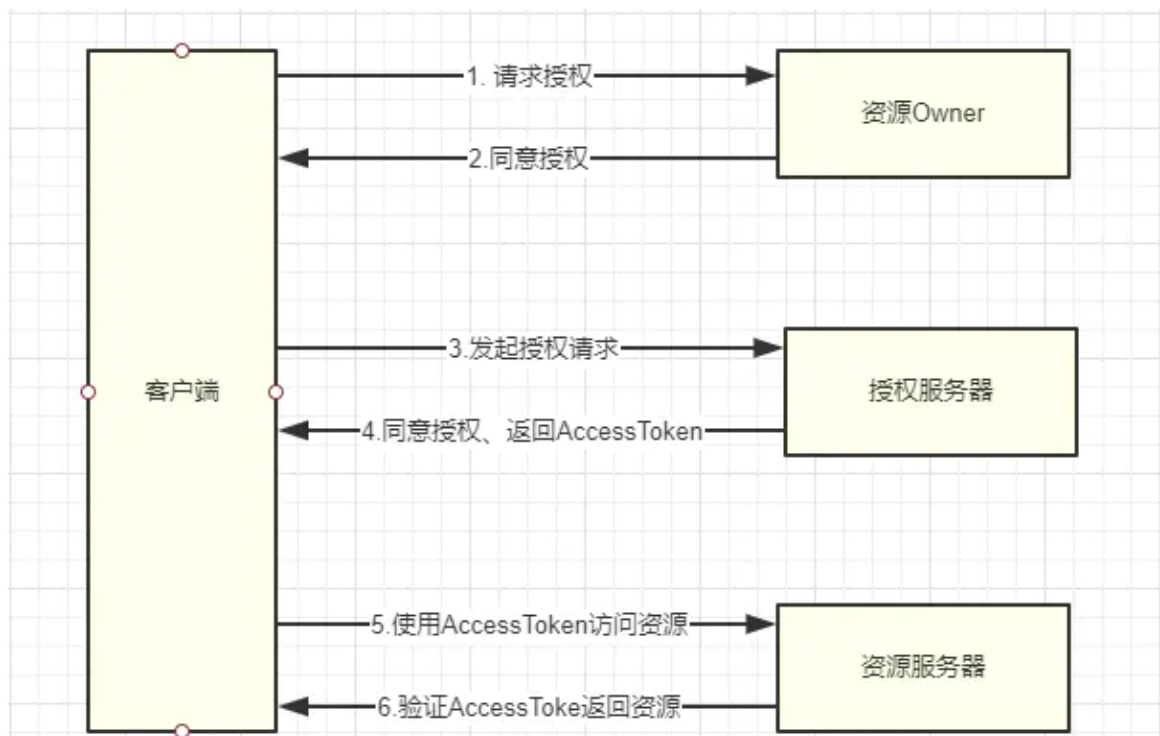
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/swagger-ui");
    }
}

```

1.2 OAuth2 角色

OAuth 2.0 中有四种类型的角色分别为：资源Owner、授权服务、客户端、资源服务

授权流程



资源 Owner (资源所有者)

资源 Owner (拥有者) 可以理解为一个用户。

如使用QQ登录网易云音乐，用户使用QQ账号登录网易云音乐，网易就需要知道用户在QQ中的头像、用户名等信息，这些账户信息都是属于用户的。

这样就是资源 Owner了。在网易请求从QQ中获取想要的用户信息时，QQ为了安全起见，需要通过用户（资源 Owner）的同意。

资源服务器

资源服务器存放受保护资源，要访问这些资源，需要获得访问令牌

用户账号的信息都存放在QQ的服务器中，所以这里的资源服务器就是QQ服务器。QQ服务器负责保存、保护用户的资源，任何其他第三方系统想到使用这些信息的系统都需要经过资源 Owner授权，同时依照 OAuth 2.0 授权流程进行交互。

客户端

客户端代表请求资源服务器资源的第三程序，**客户端同时也可能是一个资源服务器**

客户端就是想要获取资源的系统，如使用QQ登录网易时，网易就是OAuth中的客户端。客户端主要负责发起授权请求、获取AccessToken、获取用户资源。

授权服务器0000

授权服务器用于发放访问令牌给客户端

有了资源 Owner、资源服务器、客户端还不能完成OAuth授权的，还需要有授权服务器。在 OAuth中授权服务器除了负责与用户（资源 Owner）、客户端（网易）交互外，还要生成 AccessToken、验证AccessToken等功能，它是OAuth授权中的非常重要的一环，在例子中授权服务器就是GitHub的服务器。

1.3 授权类型

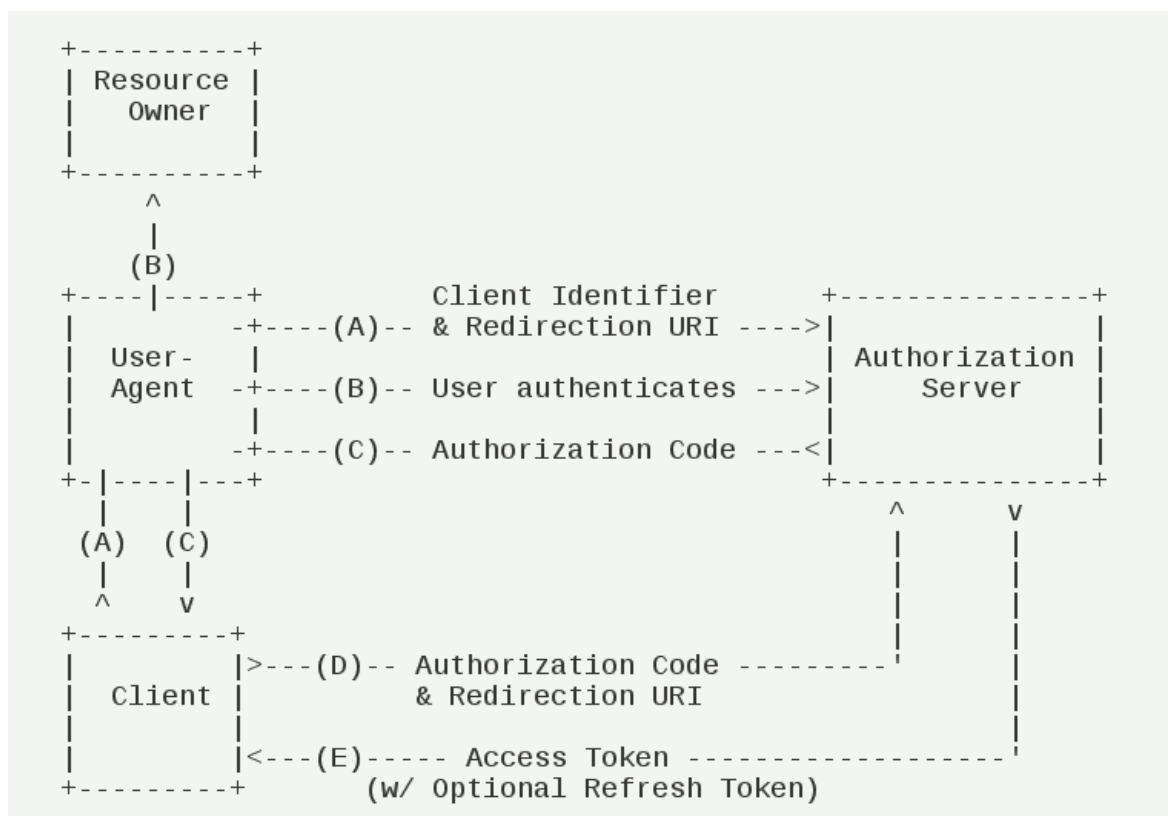
客户端必须得到用户的授权（authorization grant），才能获得令牌（access token）。OAuth 2.0定义了四种授权方式。

- 授权码模式（authorization code）

- 简化模式 (implicit)
- 密码模式 (resource owner password credentials)
- 客户端模式 (client credentials)

1.3.1 授权码授权

授权码模式是功能最完整、流程最严密的授权模式，它的特点是通过客户端的后台服务器，与“服务器提供”的认证服务器进行互动。



A. 用户访问客户端，后者将前者导向认证服务器。

B. 用户选择是否给予客户端授权。

C. 假设用户给予授权，认证服务器将用户导向客户端事先指定的“重定向URI” (redirection URI)，同时附上一个授权码。

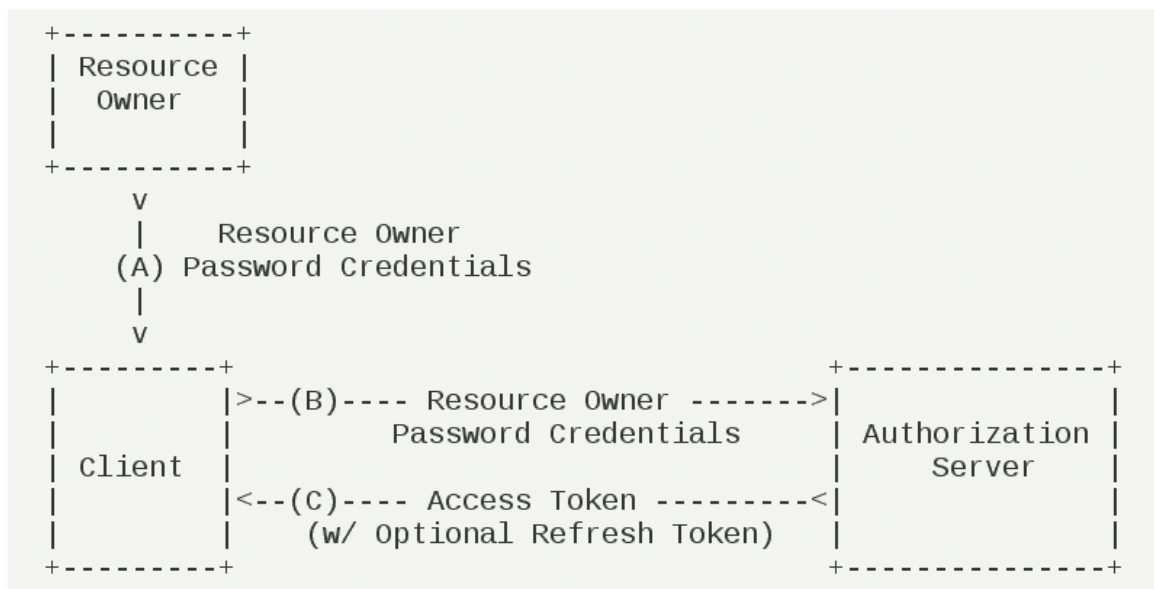
D. 客户端收到授权码，附上早先的“重定向URI”，向认证服务器申请令牌。这一步是在客户端的后台的服务器上完成的，对用户不可见。

E. 认证服务器核对了授权码和重定向URI，确认无误后，向客户端发送访问令牌 (access token) 和更新令牌 (refresh token)。

1.3.2 用户的密码授权

密码模式 (Resource Owner Password Credentials Grant) 中，用户向客户端提供自己的用户名和密码。客户端使用这些信息，向“服务商提供商”索要授权。

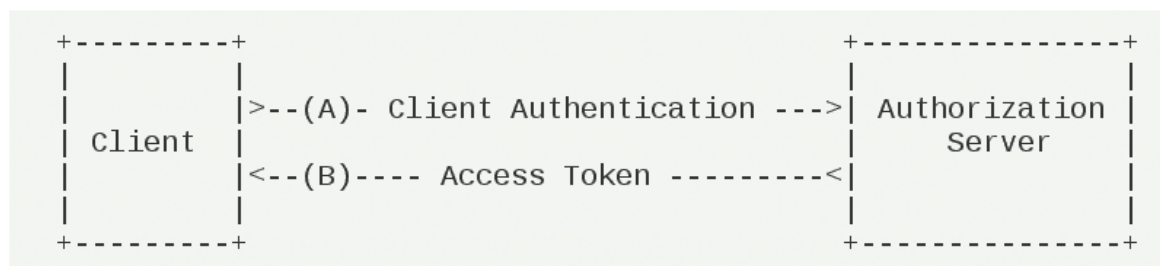
在这种模式中，用户必须把自己的密码给客户端，但是客户端不得储存密码。这通常用在用户对客户端高度信任的情况下，比如客户端是操作系统的一部分，或者由一个著名公司出品。而认证服务器只有在其他授权模式无法执行的情况下，才能考虑使用这种模式。



- A. 用户向客户端提供用户名和密码。
- B. 客户端将用户名和密码发给认证服务器，向后者请求令牌。
- C. 认证服务器确认无误后，向客户端提供访问令牌。

1.3.3 客户端凭证授权

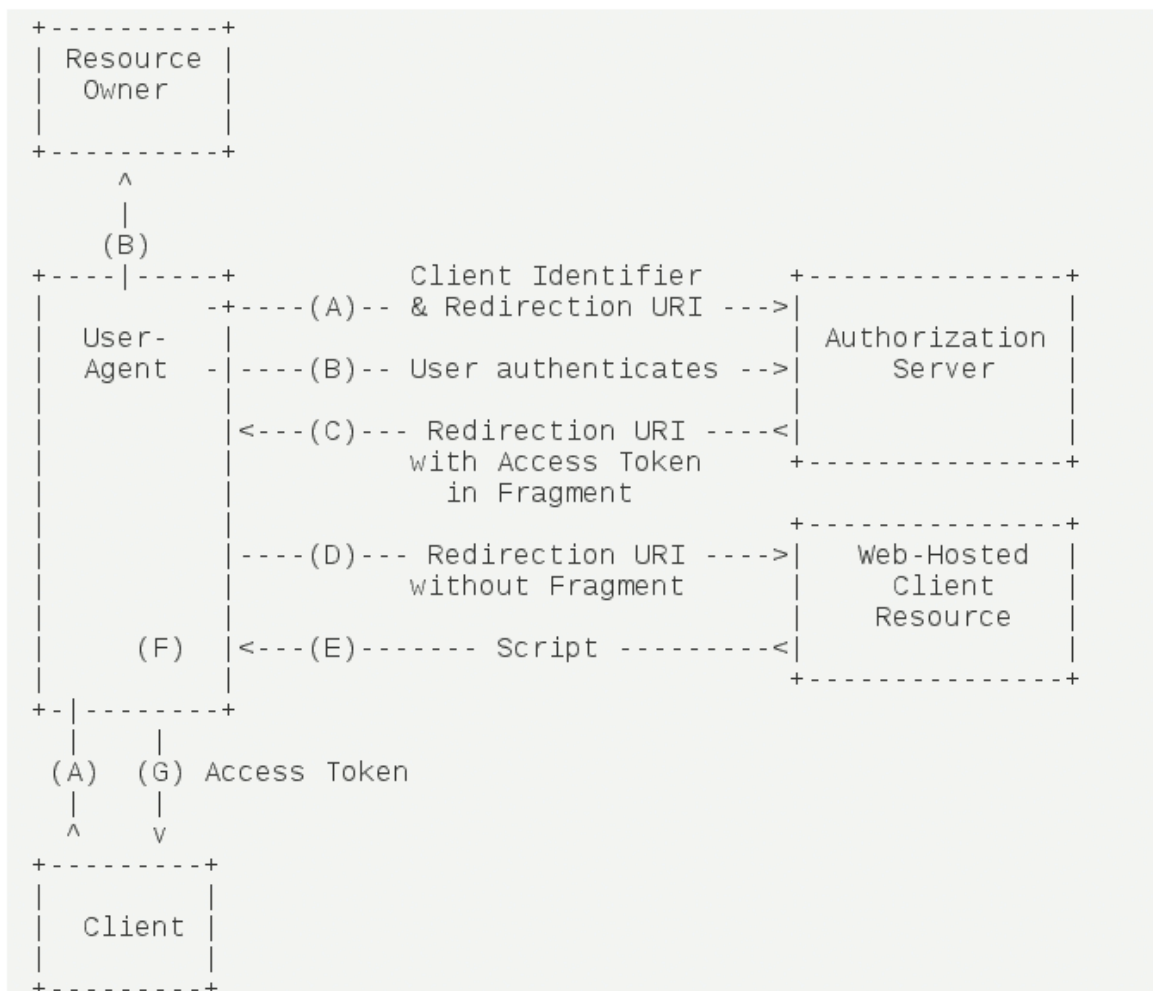
客户端模式（Client Credentials Grant）指客户端以自己的名义，而不是以用户的名义，向“服务提供商”进行认证。严格地说，客户端模式并不属于OAuth框架所要解决的问题。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求“服务提供商”提供服务，其实不存在授权问题。



- A. 客户端向认证服务器进行身份认证，并要求一个访问令牌。
- B. 认证服务器确认无误后，向客户端提供访问令牌。

1.3.4 简化授权

简化模式（implicit grant type）不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了“授权码”这个步骤，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且客户端不需要认证。



- 客户端将用户导向认证服务器。
- 用户决定是否给予客户端授权。
- 假设用户给予授权，认证服务器将用户导向客户端指定的"重定向URI"，并在URI的Hash部分包含了访问令牌。
- 浏览器向资源服务器发出请求，其中不包括上一步收到的Hash值。
- 资源服务器返回一个网页，其中包含的代码可以获取Hash值中的令牌。
- 浏览器执行上一步获得的脚本，提取出令牌。
- 浏览器将令牌发给客户端。

不同的授权类型可以使用在不同的场景中。

1.4 更新令牌

如果用户访问的时候，客户端的"访问令牌"已经过期，则需要使用"更新令牌"申请一个新的访问令牌。

客户端发出更新令牌的HTTP请求，包含以下参数：

- grant_type：表示使用的授权模式，此处的值固定为"refresh_token"，必选项。
- refresh_token：表示早前收到的更新令牌，必选项。
- scope：表示申请的授权范围，不可以超出上一次申请的范围，如果省略该参数，则表示与上一次一致。

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=tGzv3J0kFOXG5Qx2TlKWIA
```

二、OAuth2 授权服务器

在OAuth2中，一般分为授权服务器和资源服务器。

引入依赖

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

想要启动授权服务器需要实现以下几类配置。

2.1 Authorization Server 配置

只要你配置了授权服务器，那么你应该考虑

1. 客户端用于获取access token的授权类型（例如，授权码，用户凭证，刷新token）。
2. 服务器的配置是用来提供client detail服务和token服务的，并且可以启用或者禁用全局的某些机制。
3. 每个客户端可以配置不同的权限

@EnableAuthorizationServer注解被用来配置授权服务器，继承AuthorizationServerConfigurerAdapter类

```
@Configuration
//配置授权服务器
@EnableAuthorizationServer
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {
```

2.1.1 配置客户端详情 (Client Details)

ClientDetailsServiceConfigurer 能够使用内存或 JDBC 方式实现获取已注册的客户端详情：

- clientId：客户端标识 ID
- secret：客户端安全码
- scope：客户端访问范围，默认为空则拥有全部范围（相当于一个标记，标记客户端可以访问那些东西）
- authorizedGrantTypes：客户端使用的授权类型，默认为空（password密码模式,authorization_code收授权码模式,refresh_token刷新token）

名	类型	长度	小数点	不是 null	虚拟	键	注释
client_id	varchar	128	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	
resource_ids	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		
client_secret	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		
scope	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		
authorized_grant_types	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		
web_server_redirect_uri	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		
authorities	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		
access_token_validity	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		
refresh_token_validity	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		
additional_information	varchar	4096	0	<input type="checkbox"/>	<input type="checkbox"/>		
autoapprove	varchar	256	0	<input type="checkbox"/>	<input type="checkbox"/>		

client_id	resource_ids	client_secret	scope	authorized_grant_types	web_server_redirect_uri	authorities	access_token_validity	refresh_token_validity	additional_information	autoapprove
browser	(Null)		read	password,refresh_token	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)
client1	(Null)	\$2a\$10\$GStfEJEyoSHiSnc all	all	password,authorization_code	http://www.baidu.com	(Null)	(Null)	(Null)	(Null)	(Null)
webapp	(Null)	(Null)	read	implicit	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)

上文中AuthorizationServerConfig类添加

```
//注入数据源
@Autowired
private DataSource dataSource;

//自定义请求认证客户端详情
@Autowired
private OAuthClientDetailsServiceImp1 oauthClientDetailsService;

/**
 * 配置客户端详情
 * 内存/JDBC方式储存客户端详情信息
 * @param clients
 * @throws Exception
 */
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
    //自定义储存-->mysql
    clients.withClientDetails(oauthClientDetailsService);
}
@Bean
public ClientDetailsService clientDetails() {
    return new JdbcClientDetailsService(dataSource);
}
```

自定义OAuthClientDetailsServiceImp1类

```
/**
 * 自定义OAuthClientDetailsService
 * 查询oauth_client_details表
 * @author CQ
 * @date 2020/4/14 20:13
 */
@Slf4j
@Configuration
public class OAuthClientDetailsServiceImp1 implements ClientDetailsService {

    @Autowired
```

```

private OAuthClientDetailsDao oauthClientDetailsDao;

/**
 * 根据id client_id获取客户端信息
 * @param s
 * @return
 * @throws ClientRegistrationException
 */
public ClientDetails loadClientByClientId(String s) throws
ClientRegistrationException {
    // 根据Id查询
    OAuthClientDetails oauthClientDetails =
oauthClientDetailsDao.selectByPrimarykey(s);
    try {
        return translateClient(oauthClientDetails);
    } catch (Exception e) {
        log.error("==> {}", e);
        throw new ClientRegistrationException("无效client");
    }
}

public ClientDetails translateClient(OAuthClientDetails details) {
    BaseClientDetails clientDetails = new
BaseClientDetails(details.getClientId(), details.getResourceIds(),
details.getScope(),
        details.getAuthorizedGrantTypes(), details.getAuthorities(),
details.getWebServerRedirectUri());
    clientDetails.setClientSecret(details.getClientSecret());

    clientDetails.setScope(StringUtils.commaDelimitedListToSet(details.getScope()))
;
    clientDetails.setAutoApproveScopes(new ArrayList<String>());

    clientDetails.setRefreshTokenValiditySeconds(details.getRefreshTokenValidity())
;
    clientDetails.setAccessTokenValiditySeconds(details.getAccessTokenValidity());
    return clientDetails;
}
}

```

相关Mybatis映射

```

public interface OAuthClientDetailsDao {
    /**
     * @author CQ
     * @param s
     * @return
     */
    OAuthClientDetails selectByPrimarykey(String s);
}

```

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.precision.auth.dao.OauthClientDetailsDao">
    <select id="selectByPrimaryKey"
resultType="com.precision.common.entity.OauthClientDetails"
parameterType="java.lang.String">
        select * from oauth_client_details where client_id = #
{clientId,jdbcType=VARCHAR}
    </select>
</mapper>

```

实体类

```

@Data
public class OauthClientDetails {
    String clientId;
    String resourceIds;
    String scope;
    String authorizedGrantTypes;
    String authorities;
    String webServerRedirectUri;
    String clientSecret;
    String autoApproveScopes;
    Integer refreshTokenValidity;
    Integer accessTokenValidity;
    String additional_information;
}

```

2.1.2 管理令牌 (Managing Token)

- ResourceServerTokenServices 接口定义了令牌加载、读取方法
 - AuthorizationServerTokenServices 接口定义了令牌的创建、获取、刷新方法
 - ConsumerTokenServices 定义了令牌的撤销方法
- 一般直接使用下述DefaultTokenServices接口
- DefaultTokenServices 实现了上述三个接口,它包含了一些令牌业务的实现,如创建令牌、读取令牌、刷新令牌、获取客户端ID,除了持久化令牌是委托一个 TokenStore 接口实现以外。默认的当尝试创建一个令牌时,是使用 UUID 随机值进行填充的。
 - TokenStore 接口的实现:
 - InMemoryTokenStore: 默认采用该实现,将令牌信息保存在内存中
 - jdbcTokenStore: 令牌会被保存在关系型数据库,可以在不同服务器之间共享令牌
 - JwtTokenStore: 使用 JWT 方式保存令牌

使用JDBC储存令牌

```

/**
 * access_token 存储 mysql
 * @return
 */
@Primary
@Bean
public TokenStore jdbcTokenStore(){
    return new JdbcTokenStore(dataSource);
}

```

自定义令牌

```

/**
 * pmp
 * 自定义令牌
 * @author : CQ
 * @date : 2020-04-14 19:22
 */
public class MyTokenEnhancer implements TokenEnhancer {
    @Override
    public OAuth2AccessToken enhance(OAuth2AccessToken accessToken,
    OAuth2Authentication authentication) {
        User user = (User) authentication.getPrincipal();
        final Map<String, Object> additionalInfo = new HashMap<>();
        additionalInfo.put("username", user.getUsername());
        ((DefaultOAuth2AccessToken)
        accessToken).setAdditionalInformation(additionalInfo);
        return accessToken;
    }
}

```

DefaultTokenServices类

```

/**
 * 把认证的token保存到JDBC
 * 注意，自定义TokenServices的时候，需要设置@Primary，否则报错
 * @return
 */
@Primary
@Bean
public DefaultTokenServices defaultTokenServices(){
    DefaultTokenServices tokenServices = new DefaultTokenServices();
    //设置jdbc
    tokenServices.setTokenStore(jdbcTokenStore());
    //令牌刷新
    tokenServices.setSupportRefreshToken(true);
    //自定义令牌
    tokenServices.setTokenEnhancer(tokenEnhancer());
    // token有效期自定义设置，默认12小时
    tokenServices.setAccessTokenValiditySeconds(60*60*12);
    // refresh_token默认30天
    tokenServices.setRefreshTokenValiditySeconds(60 * 60 * 24 * 7);
    return tokenServices;
}

/**

```

```

    * 自定义令牌
    * @return
    */
@Bean
public TokenEnhancer tokenEnhancer() {
    return new MyTokenEnhancer();
}

```

2.1.3 配置授权类型 (Grant Types)

- 授权是使用 AuthorizationEndpoint 这个端点来进行控制的，使用 AuthorizationServerEndpointsConfigurer 这个对象实例来进行配置，默认是支持除了密码授权外所有标准授权类型，它可配置以下属性：
 - authenticationManager：认证管理器，当需要密码授权类型的时候，注入一个 AuthenticationManager 对象
 - userDetailsService：可定义自己的 UserDetailsService 接口实现
 - authorizationCodeServices：用来设置授权码服务的（即 AuthorizationCodeServices 的实例对象），主要用于 "authorization_code" 授权码类型模式，可以使用JDBC储存
 - implicitGrantService：这个属性用于简化（隐式）模式，用来管理隐式授权模式的状态
 - tokenGranter：自定义授权服务实现（TokenGranter 接口实现），当标准的四种授权模式已无法满足需求时使用

自定义UserDetailsService

```

/**
 * 自定义UserDetailsService
 *
 * @author cq
 */
@Service("userDetailsService")
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private UserDao userDao;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        MyUser myUser = userDao.findByUserName(username);
        if (myUser == null) {
            throw new UsernameNotFoundException(username);
        }
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        // 可用性 :true:可用 false:不可用
        boolean enabled = true;
        // 过期性 :true:没过期 false:过期
        boolean accountNonExpired = true;
        // 有效性 :true:凭证有效 false:凭证无效
        boolean credentialsNonExpired = true;
        // 锁定性 :true:未锁定 false:已锁定
        boolean accountNonLocked = true;
        for (Role role : myUser.getRoles()) {
            //角色必须是ROLE_开头，可以在数据库中设置
            GrantedAuthority grantedAuthority = new
SimpleGrantedAuthority(role.getRoleName());
            grantedAuthorities.add(grantedAuthority);
        }
    }
}

```

```

        //获取权限
        for (Permission permission : role.getPermissions()) {
            GrantedAuthority authority = new
SimpleGrantedAuthority(permission.getUri());
            grantedAuthorities.add(authority);
        }
    }
    User user = new User(myUser.getUserName(), myUser.getPassword(),
        enabled, accountNonExpired, credentialsNonExpired,
accountNonLocked, grantedAuthorities);
    return user;
}

}

```

实体类

```

/**
 * 用户实体
 *
 * @author CQ
 *
 */
@Data
public class MyUser {
    //用户id
    private int id;
    //用户姓名
    private String userName;
    //用户密码
    private String password;
    //手机号
    private String mobile;
    //邮箱
    private String email;
    private short sex;
    private Date birthday;
    private Date createTime;
    //角色
    private Set<Role> roles;
}

```

相关Mybatis映射

```

public interface UserDao {
    /**
     * 用户名查用户
     * @param userName 用户名
     * @return 用户
     */
    MyUser findByUserName(String userName);
}

```

```
<?xml version="1.0" encoding="UTF-8" ?>
```



```

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.precision.auth.dao.UserDao">

    <resultMap id="baseResultMap" type="com.precision.common.entity.MyUser">
        <id property="id" column="id" jdbcType="INTEGER" />
        <result property="userName" column="user_name" jdbcType="VARCHAR" />
        <result property="password" column="password" jdbcType="VARCHAR" />
        <result property="mobile" column="mobile" jdbcType="VARCHAR" />
        <result property="email" column="email" jdbcType="VARCHAR" />
        <result property="sex" column="sex" jdbcType="TINYINT" />
        <result property="birthday" column="birthday" jdbcType="TIMESTAMP" />
        <result property="createTime" column="createTime" jdbcType="TIMESTAMP"
    />

        <collection property="roles" fetchType="eager" column="id"
select="com.precision.auth.dao.RoleDao.findById"></collection>
    </resultMap>

    <sql id="normalItems">
        id,user_name,email,mobile,sex
    </sql>

    <select id="findByUserName" parameterType="java.lang.String"
resultMap="baseResultMap">
        select * from user where user_name = #{userName,jdbcType=VARCHAR}
    </select>

</mapper>

```

```

//开启密码模式->注入authenticationManager
@Autowired
private AuthenticationManager authenticationManager;

/**
 * 授权码code 储存mysql
 * @return
 */
@Bean
public AuthorizationCodeServices authorizationCodeServices(){
    JdbcAuthorizationCodeServices services=new
JdbcAuthorizationCodeServices(dataSource);
    return services;
}

//自定义userService用户详情
@Autowired
private MyUserService userService;

/**
 * 配置授权服务器端点
 * 告诉Spring Security Token的生成方式
 * 默认支持除了密码授权外所有标准授权类型
 * @param endpoints
 * @throws Exception
 */
@Override

```

```

    public void configure(AuthorizationServerEndpointsConfigurer endpoints)
    throws Exception {
        endpoints
            .allowedTokenEndpointRequestMethods(HttpMethod.GET,
            HttpMethod.POST)
            //开启密码模式->注入authenticationManager
            .authenticationManager(authenticationManager)
            //开启认可储存->oauth_approvals表
            .approvalStore(approvalStore())
            //开启自定义用户详情
            .userDetailsService(userDetailsService)
            //使用JDBC存储令牌
            .tokenStore(jdbcTokenStore())
            //设置授权码模式
            .authorizationCodeServices(authorizationCodeServices());
            //设置tokenServices令牌服务
        endpoints.tokenServices(defaultTokenServices());
        //认证异常翻译
        //endpoints.exceptionTranslator(webResponseExceptionHandler());
        //自定义确认授权页面
        endpoints.pathMapping("/oauth/confirm_access", "/oauth/confirm_access");
        //自定义错误页面
        endpoints.pathMapping("/oauth/error", "/oauth/error");
    }

```

2.1.4配置授权端点 URL (Endpoint URLs)

- AuthorizationServerEndpointsConfigurer 配置对象有一个 pathMapping() 方法用来配置端点的 URL，它有两个参数：
 - 参数一：端点 URL 默认链接
 - 参数二：替代的 URL 链接
- 下面是一些默认的端点 URL：
 - /oauth/authorize：授权端点
 - /oauth/token：令牌端点
 - /oauth/confirm_access：用户确认授权提交端点
 - /oauth/error：授权服务错误信息端点
 - /oauth/check_token：用于资源服务访问的令牌解析端点
 - /oauth/token_key：提供公有密匙的端点，如果你使用JWT令牌的话
- 授权端点的 URL 应该被 Spring Security 保护起来只供授权用户访问

```

@Override
    public void configure(AuthorizationServerSecurityConfigurer security) throws
    Exception {
        security
            //允许客户端发送表单来进行权限认证获取令牌
            .allowFormAuthenticationForClients()
            //允许所有资源服务器访问公钥端点 (/oauth/token_key)
            .tokenKeyAccess("permitAll()")
            //只允许验证用户访问令牌解析端点 (/oauth/check_token)
            .checkTokenAccess("isAuthenticated()");
    }

```

2.2 Resource Server配置

很多情况下授权服务器同时也提供着一些关于用户信息查询的功能，所以授权服务器也可以作为一个资源服务器，为其他需要使用用户信息的客户端提供服务。

此时ResourceServerConfig继承ResourceServerConfigurerAdapter，对资源服务器进行相应的配置，同时加上@EnableResourceServer注解。

如下，对服务器上前缀为api的接口进行保护，必须获取授权之后使用。

```
/**
 * 资源认证服务器
 * 资源服务器，配置受保护的资源，用于保护oauth受限资源
 *
 * @author cq
 */
@Configuration
@EnableResourceServer
@Order(3)
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .exceptionHandling()
            .authenticationEntryPoint((request, response, authException) ->
                response.sendError(HttpStatus.UNAUTHORIZED))
            .and()
            .requestMatchers().antMatchers("/api/**")
            .and()
            .authorizeRequests()
            .antMatchers("/api/**").authenticated()
            .and()
            .authorizeRequests()
            .and()
            .httpBasic();
    }
}
```

2.3 WebSecurity配置

WebSecurityConfigurerAdapter存在的目的是提供一个方便开发人员配置WebSecurity的基类。它提供了一组全方位配置WebSecurity的缺省方法实现。开发人员只要继承WebSecurityConfigurerAdapter提供自己的实现类，哪怕不覆盖WebSecurityConfigurerAdapter的任何一个方法，都得到了一个配置WebSecurity的安全配置器WebSecurityConfigurer实例。

如下，我们对授权服务器的一些接口进行配置，保证暴露的接口的安全性。同时为了项目中的swagger2可以正常使用，放行有关swagger2的接口。

```
/**
 * security配置
 *
 * @author cq
 */
@Configuration
@EnableWebSecurity
@Order(2)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```

@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder(){
    return new BCryptPasswordEncoder();
}

@Autowired
private MyUserDetailService userDetailsService;

@Autowired
private BCryptPasswordEncoder passwordEncoder;

/**
 * 安全过滤器链配置
 * 用来配置 HttpSecurity
 * @param http
 * @throws Exception
 */
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        //session只在security需要的时候创建session

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
        .and()
        //端点配置组,等同于多个.antMatchers()
        .requestMatchers()
        .antMatchers("/oauth/**",
"/login", "/test", "/loginSuccess", "/logout")
        .and()
        //允许基于使用HttpServletRequest限制访问, 配置权限hasAnyRole、
access(...)

        .authorizeRequests()
        // /login所有人都可以访问, 表示登录表单页面不拦截
        .antMatchers("/login").permitAll()
        .antMatchers("/logout").permitAll()
        // /oauth需要授权
        .antMatchers("/oauth/**").authenticated()// 保护url, 需要用户登录
        .antMatchers("/test").authenticated()
        .antMatchers("/loginSuccess").authenticated()
        .and()
        //没有自定义loginpage 则不要写上loginPage("/xxxx") 否则404
        // 指定支持基于表单的身份验证。如果未指定
FormLoginConfigurer#loginPage(String), 则将生成默认登录页面
        .formLogin().permitAll()
        //登录成功后默认处理页
        .defaultSuccessUrl("/loginSuccess")
        .and()
        //任何人都可以注销, 默认/logout
        .logout().permitAll()
        // /logout退出清除cookie
        .addLogoutHandler(new CookieClearingLogoutHandler("token",
"remember-me"))
        .and()
        //禁用csrf跨域, 默认开启
        .csrf().disable()
        // 禁用httpBasic
        .httpBasic().disable();
}

```

```

/**
 * 认证管理器配置AuthenticationManager
 * 所有和UserDetails相关的
 * userDetailsService提供登录用户的账密信息供springsecurity框架校验
 */
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth.userDetailsService(userDetailsService)
        .passwordEncoder(passwordEncoder);
}

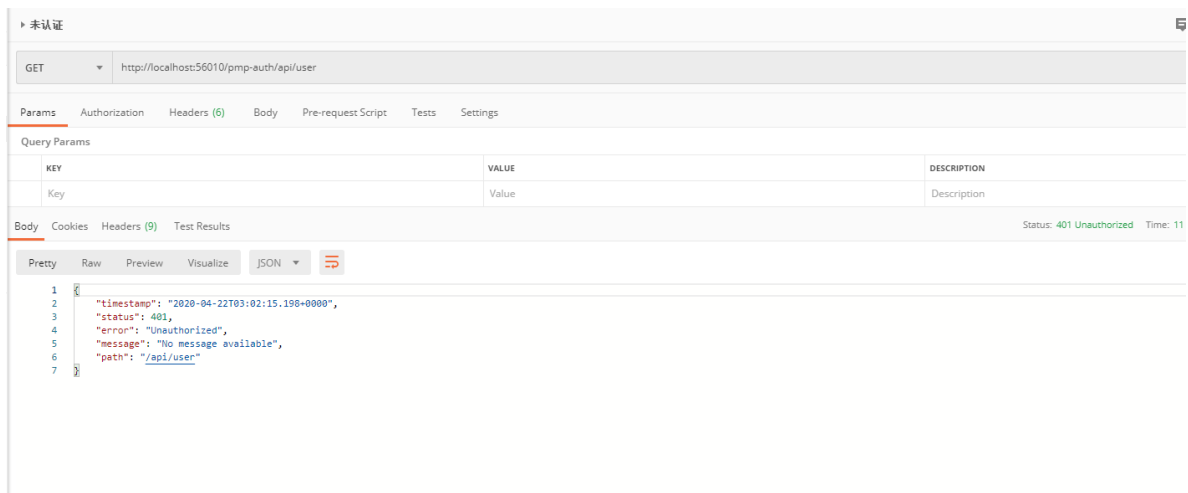
/**
 * 不定义没有password grant_type,密码模式需要AuthenticationManager支持
 * 需要声明bean, 不声明不能注入
 * @return
 * @throws Exception
 */
@Override
@Bean
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}

/**
 * 过滤器配置方法
 * 来配置WebSecurity
 * 一般交给WebSecurityConfiguration
 * 只会在这里ignoring()方法用来忽略 Spring Security 对静态资源的控制
 * @param web
 * @throws Exception
 */
@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers("/v2/api-docs", "/swagger-
resources/configuration/ui",
        "/swagger-resources", "/swagger-
resources/configuration/security",
        "/swagger-ui.html", "/css/**", "/js/**", "/images/**",
        "/webjars/**", "**/favicon.ico", "/index");
}
}

```

2.4 密码模式获取token

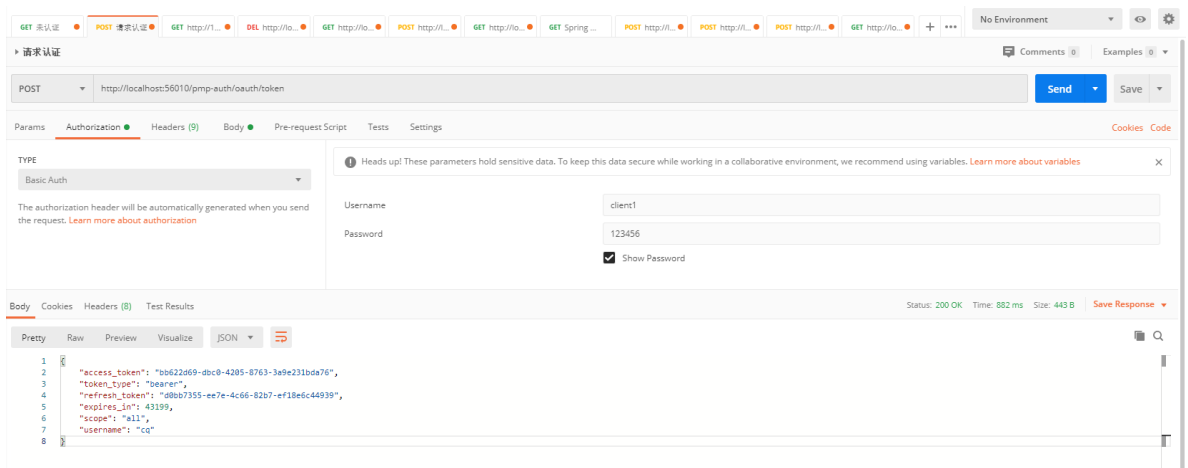
未认证模式下请求接口，会返回401未授权。



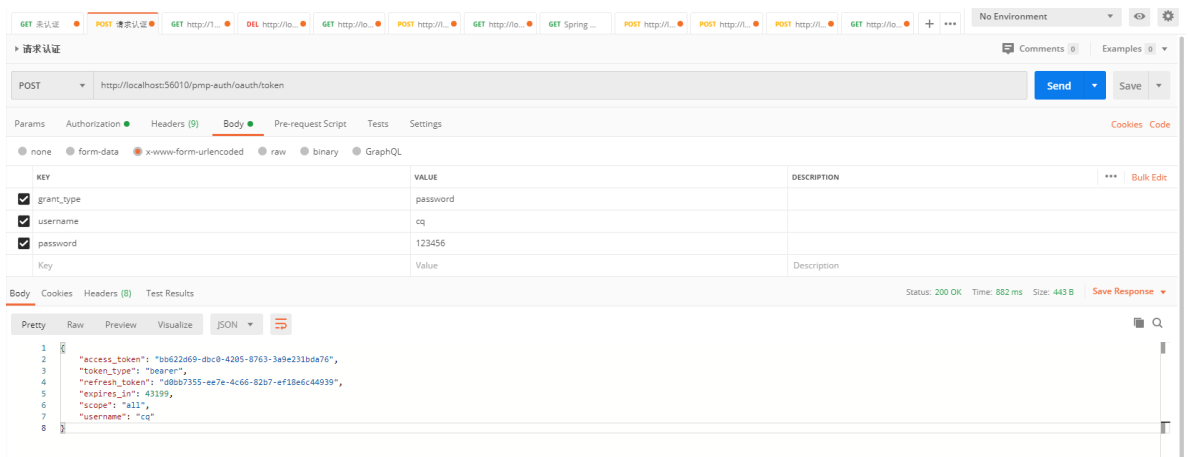
所以我们需要获取token，才能请求接口。

获取token

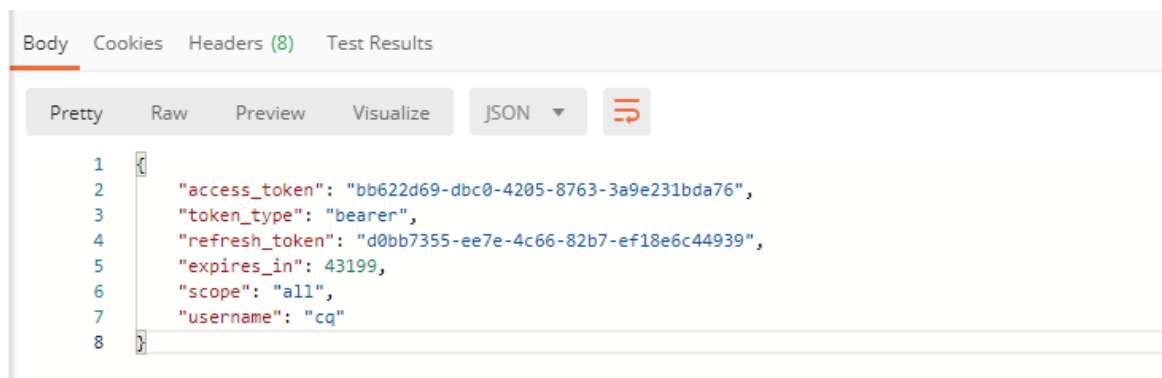
1. postman中需要设置为post模式，Authorization中选择Basic Auth，填写客户端id和密码。



2. Body中选择x-www-form-urlencoded，填写grant_type为password，username和password字段。



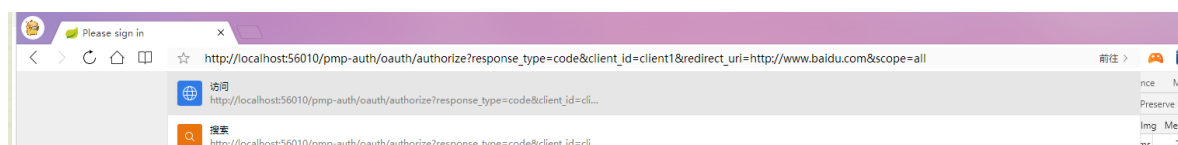
3. 点击send，会返回token



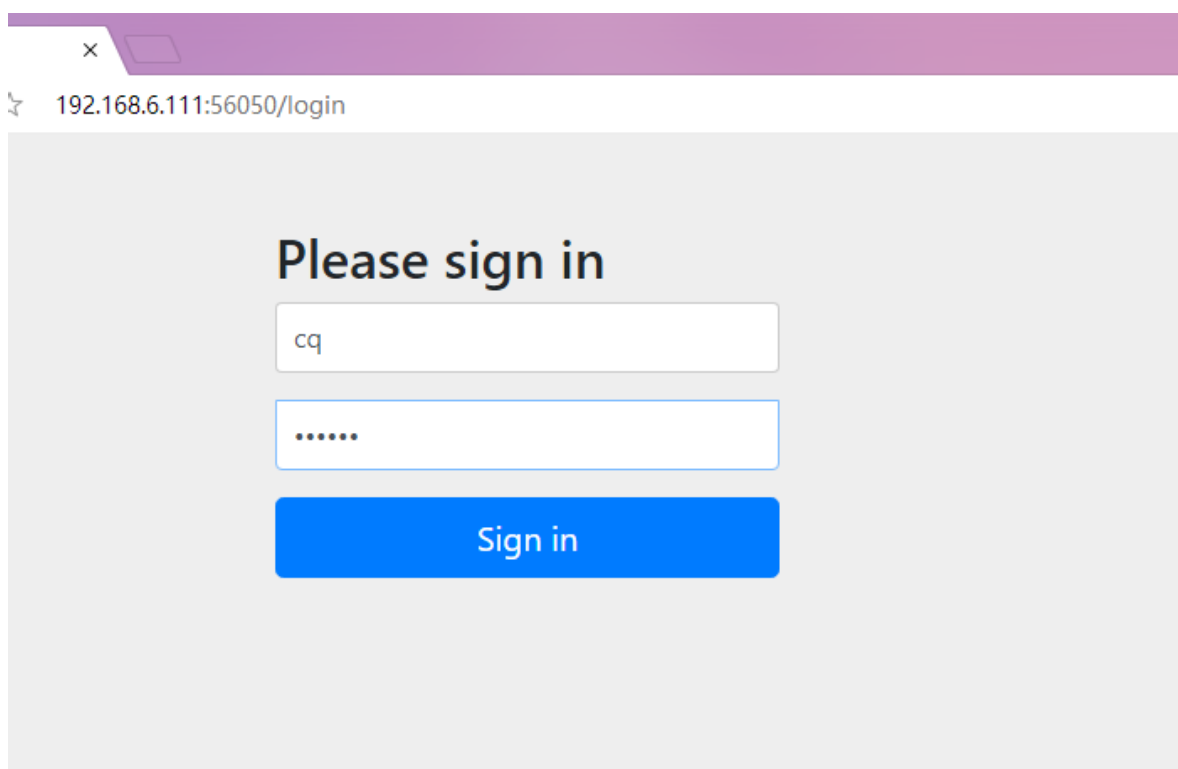
2.5 授权码模式获取token

授权码模式使用浏览器模拟

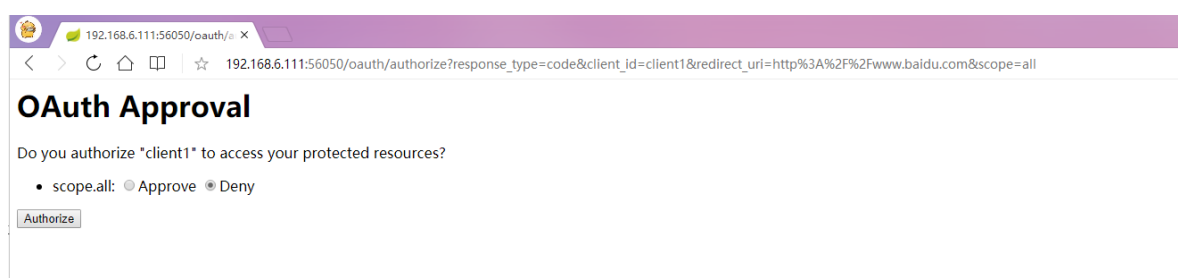
http://192.168.6.111:56010/pmp-auth/oauth/authorize?response_type=code&client_id=client1&redirect_uri=http://www.baidu.com&scope=all



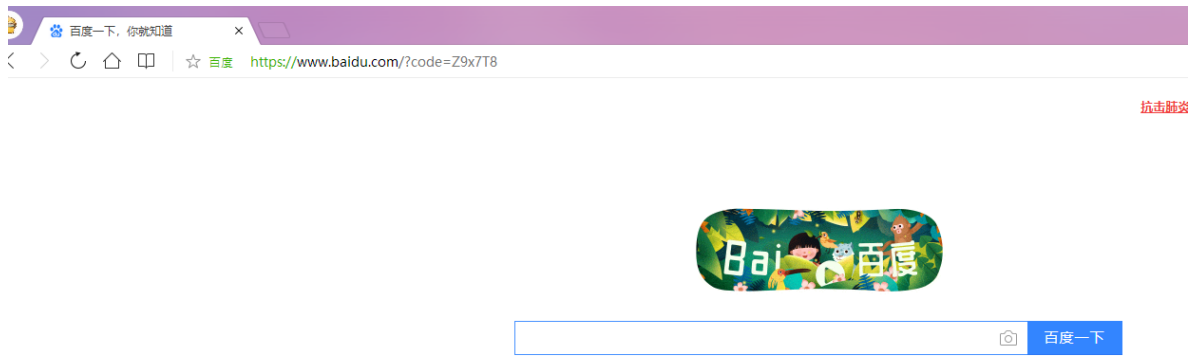
会提示需要登录



登录之后，会显示是否授权



点击Approve，会重定向到指定页面，并带上授权码（此处对于用户来说不可见）



此处会在数据库中oauth_approvals表中生成记录，下次不需要点击授权，有效期默认1个月

对象

oauth_approvals @test (loc...oauth_code @test (localhos...

开始事务

文本 筛选 排序

导入 导出

userId	clientId	scope	status	expiresAt	lastModifiedAt
cq	client1	all	APPROVED	2020-05-22 10:51:43	2020-04-22 10:51:43

然后使用上面生成的code去请求token

使用postman，参数：

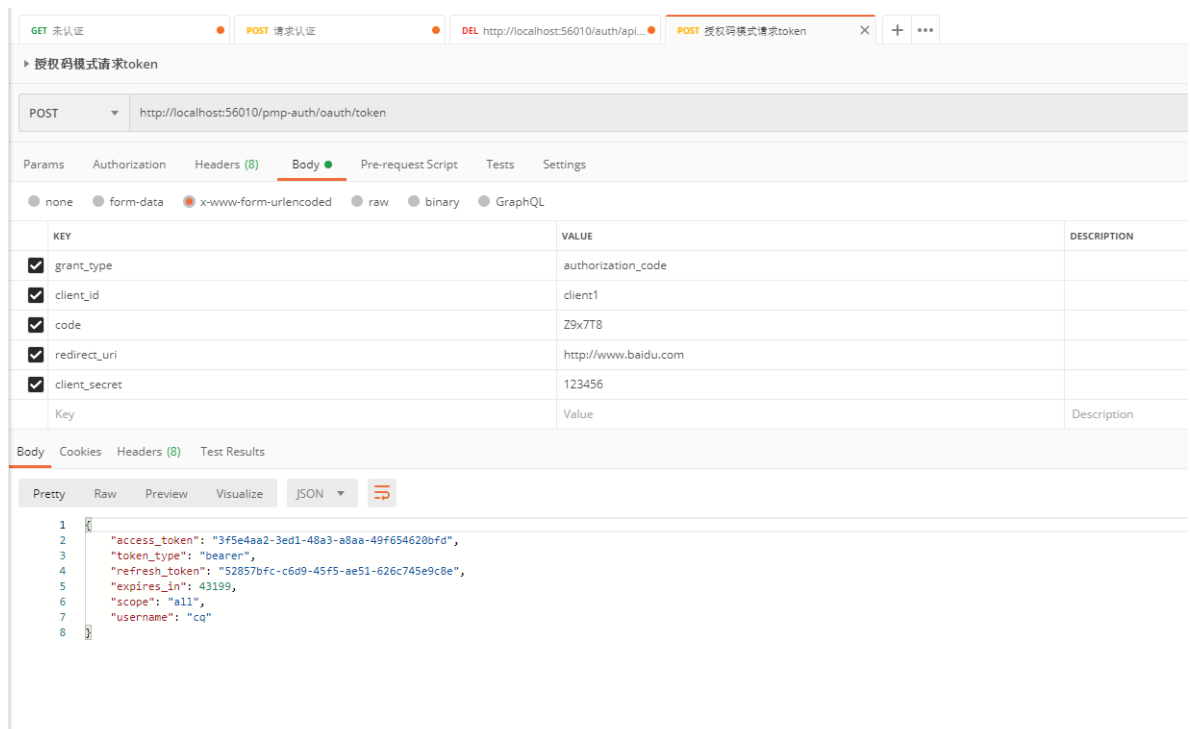
grant_type:authorization_code（指定授权码模式）

client_id:client1（客户端id）

code:Z9x7T8（授权码）

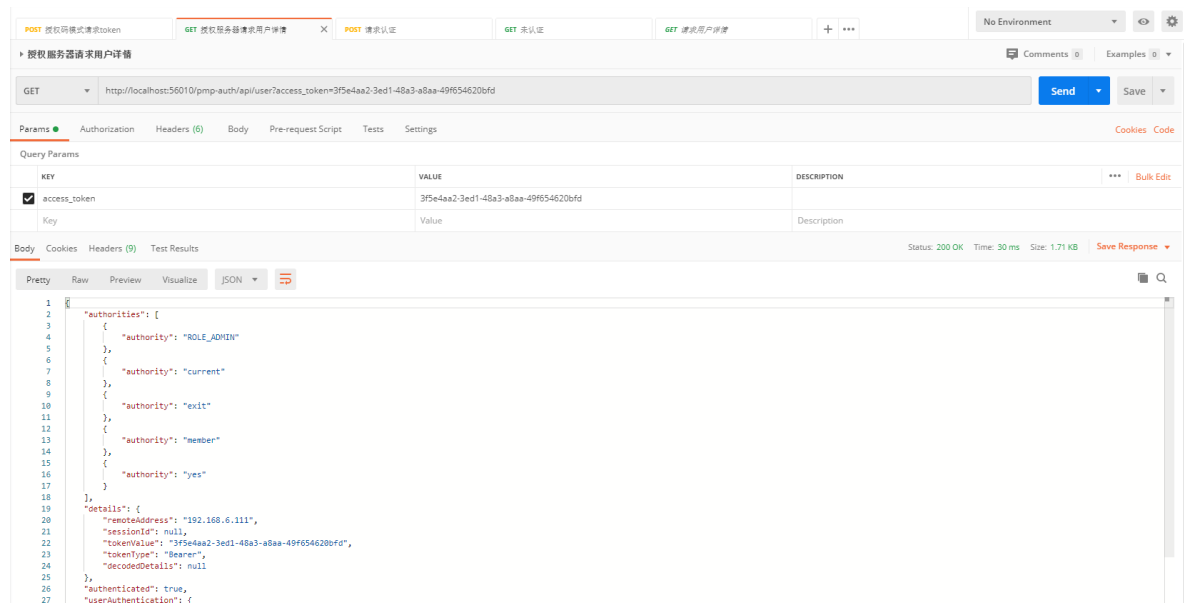
redirect_uri:<http://www.baidu.com>（重定向连接）

client_secret:123456（客户端密钥）



即可获得token

2.6 使用token获取资源



三、OAuth2 资源服务器

资源服务器其实就是我们微服务中一个个的服务，他们提供其他服务器所需要的资源。要访问资源服务器受保护的资源需要携带令牌，令牌从授权服务器获取。

客户端往往同时也是一个资源服务器，各个服务之间的通信（访问需要权限的资源）时需携带访问令牌

3.1 Resource Server配置

资源服务器通过 `@EnableResourceServer` 注解来添加配置，通过继承 `ResourceServerConfigurerAdapter` 类来配置资源服务器。

如下配置保护服务器中的api接口，需要授权之后才能进行访问。同时放行swagger2相关接口，保证swagger正常使用。

```
/**
 * OAuth资源服务配置
 *
 * @author cq
 */
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .exceptionHandling()
            .authenticationEntryPoint((request, response, authException) ->
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED))
            .and()
            .requestMatchers().antMatchers("/api/**")
            .and()
            .authorizeRequests()
```

```

        .antMatchers("/api/**").authenticated()
        .antMatchers(
            "/webjars/**",
            "/resources/**",
            "/swagger-ui.html",
            "/swagger-resources/**",
            "/v2/api-docs").permitAll()

        .and()
        .httpBasic();
    }
}

```

3.2 资源服务设置用户信息地址

资源服务一定要设置用户信息地址（OAuth2授权认证服务的用户信息）

```

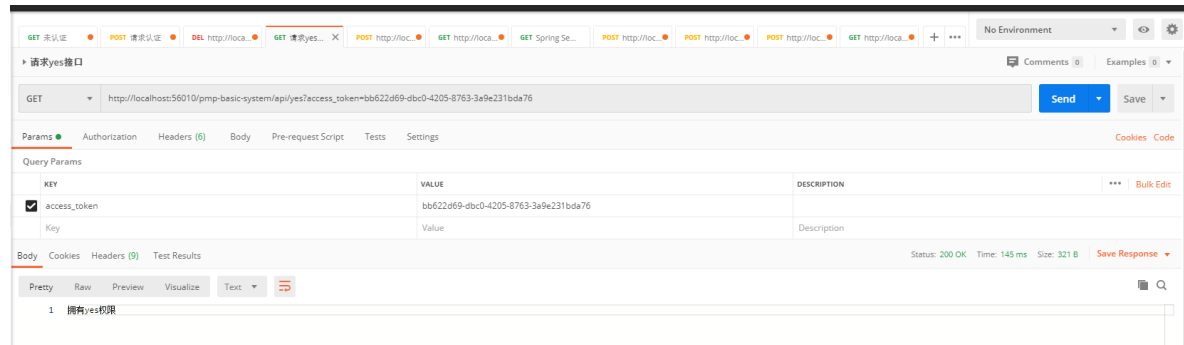
security:
  oauth2:
    resource:
      id: pmp-basic-system
      user-info-uri: http://localhost:56010/pmp-auth/api/user
      prefer-token-info: false

```

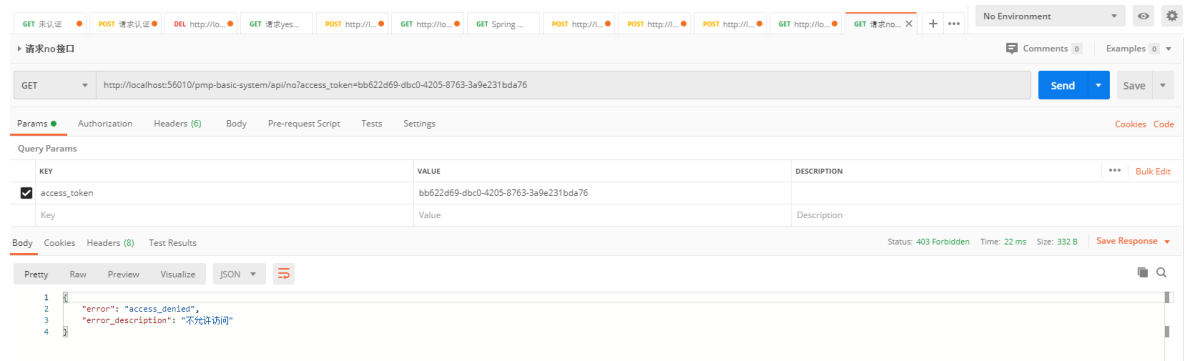
3.3 使用token在资源服务器上获取资源

有了token之后，只需要把token添加到请求的url之后就可以请求对应的接口。

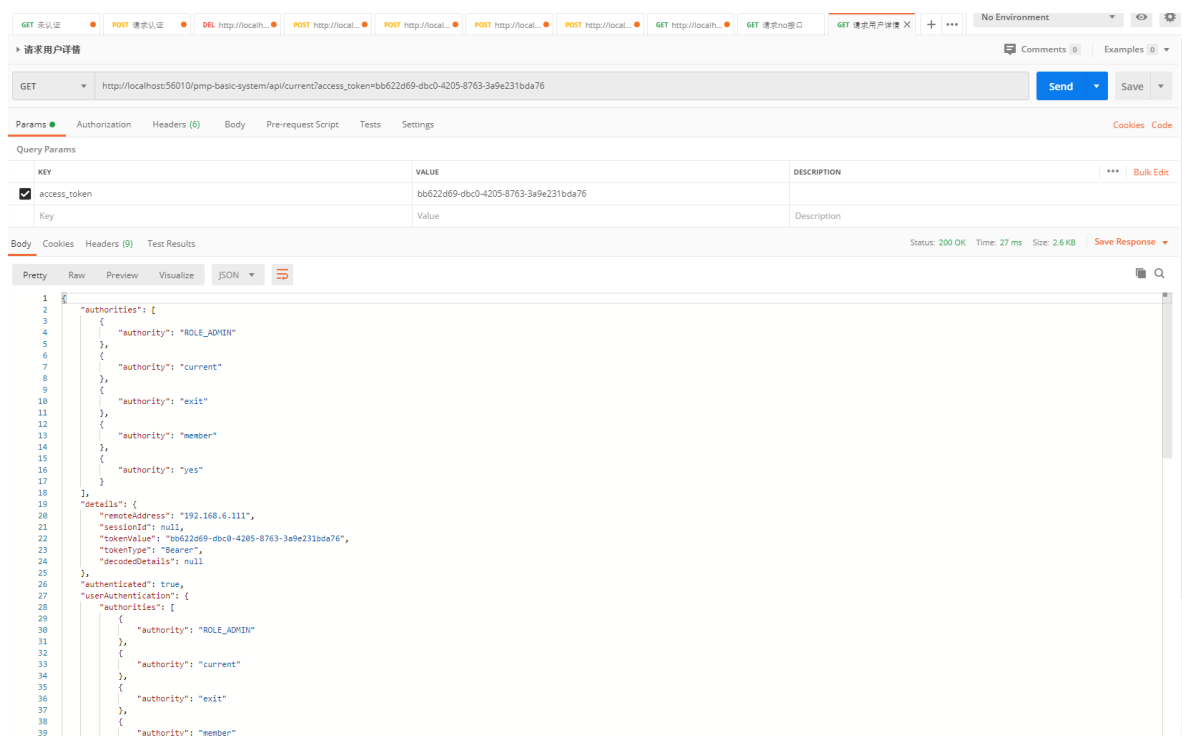
拥有yes接口权限的



没有权限访问no接口的



查询用户权限的接口



四、相关类汇总

AuthorizationServerConfigurerAdapter

- 通过继承该类并重写 `configure` 方法来配置授权服务器：
 - `configure(AuthorizationServerEndpointsConfigurer endpoints)`: 配置授权服务器端点，如令牌存储，令牌自定义，用户批准和授权类型，不包括端点安全配置
 - `configure(AuthorizationServerSecurityConfigurer oauthServer)`: 配置授权服务器端点的安全
 - `configure(ClientDetailsServiceConfigurer clients)`: 配置 `ClientDetailsService` 也就是客户端属性信息

ResourceServerConfigurerAdapter

- 通过继承该类并重写 `configure` 方法来配置资源服务器：
 - `configure(org.springframework.security.config.annotation.web.builders.HttpSecurity http)`: 配置资源的访问规则
 - `configure(ResourceServerSecurityConfigurer resources)`: 添加资源服务器特定的属性（如资源ID），默认值适用于很多情况，但至少需要修改下ID

TokenStore

- OAuth2 Token（令牌）持久化接口，用于定义 Token 如何存储，它有几个实现类：
 - `InMemoryTokenStore`: 实现了在内存中存储令牌
 - `JdbcTokenStore`: 通过 JDBC 方式存储令牌
 - `JwtTokenStore`: 通过 JWT 方式存储令牌

TokenEnhancer

- 可用于自定义令牌策略，在令牌被 `AuthorizationServerTokenServices` 的实现存储之前增强令牌的策略，它有两个实现类：
 - `JwtAccessTokenConverter`: 用于令牌 JWT 编码与解码

- TokenEnhancerChain：一个令牌链，可以存放多个令牌，并循环的遍历令牌并将结果传递给下一个令牌

TokenEndpoint

- 该类根据 OAuth2 规范实现了令牌端点，客户端必须使用 Spring Security 身份验证来访问此端点，并从身份验证令牌客户端 ID，根据 OAuth2 规范，使用标准的 Spring Security 的 HTTP Basic 验证

AuthorizationServerTokenServices

- 该接口定义了一些操作可以对令牌进行一些管理，包含了三个方法声明：
 - createAccessToken(OAuth2Authentication authentication)：从 OAuth2Authentication 对象中创建令牌
 - getAccessToken(OAuth2Authentication authentication)：从 OAuth2Authentication 对象中获取令牌
 - refreshAccessToken(String refreshToken, TokenRequest tokenRequest)：刷新令牌

OAuth2AuthenticationDetails

- HTTP 请求中资源所有者（用户）相关的 OAuth2 Authentication（身份验证）信息，它可通过 SecurityContext 获得

```
Authentication authentication =  
SecurityContextHolder.getContext().getAuthentication();  
Object details = authentication.getDetails();
```

- 它有几个方法：
 - getDecodedDetails()：
 - getRemoteAddress()：Authentication（身份验证）请求的 TCP/IP 地址
 - getSessionId()：Authentication（身份验证）请求的 SessionID
 - getTokenValue()：请求中包含的令牌信息（通常在请求头中）
 - setDecodedDetails(Object decodedDetails)：设置令牌
 - getDecodedDetails()：解码请求中包含的令牌
 - getTokenType()：获取 Token 类型，例如 Bearer

ResourceServerTokenServices

- 令牌业务类接口，它定义了两个方法：
 - loadAuthentication(String accessToken)：加载 Authentication 验证信息
 - readAccessToken(String accessToken)：读取令牌
- 它有两个实现类：
 - DefaultTokenServices：详情看下面
 - RemoteTokenServices：通过访问授权服务器 /check_token 端点来获取、解析令牌，如果端点返回 400 相应则表示该令牌无效

DefaultTokenServices

- 该类主要实现了 ResourceServerTokenServices、AuthorizationServerTokenServices、ConsumerTokenServices 三个接口
- 它包含了令牌业务的实现，如创建令牌、读取令牌、刷新令牌、撤销令牌、获取客户端ID。默认的当尝试创建一个令牌时，是使用 UUID 随机值进行填充的
- 除了持久化令牌是委托一个 TokenStore 接口实现以外，这个类几乎帮你做了所有事情

AuthorizationEndpoint

- 该类根据 OAuth2 规范实现了授权端点，如果授权类型是授权码模式，则处理用户批准，令牌是从令牌端点获得的，默认是支持除了密码授权外所有标准授权类型
- 它使用 AuthorizationServerEndpointsConfigurer 该类实例对象来进行配置，它可配置以下属性：
 - authenticationManager：认证管理器，当你选择了资源所有者密码（password）授权类型的时候，请设置这个属性注入一个 AuthenticationManager 对象
 - userDetailsService：可定义自己的 UserDetailsService 接口实现
 - authorizationCodeServices：用来设置收取码服务的（即 AuthorizationCodeServices 的实例对象），主要用于 "authorization_code" 授权码类型模式
 - implicitGrantService：这个属性用于设置隐式授权模式，用来管理隐式授权模式的状态
 - tokenGranter：完全自定义授权服务实现（TokenGranter 接口实现），只有当标准的四种授权模式已无法满足需求时

OAuth2AuthenticationProcessingFilter

- OAuth2 受保护资源的预设认证过滤器，从请求中提取 OAuth2 令牌，并将令牌传入 OAuth2Authentication，（如果与 OAuth2AuthenticationManager 结合使用）传入 Spring Security 上下文

ClientDetailsService

- 提供有关 OAuth2 客户端的详细信息的服务，该接口仅声明了一个方法：
 - loadClientByClientId(String clientId)：通过客户端名字加载客户端详情
- 它有两个实现类：
 - InMemoryClientDetailsService：客户端详细信息服务的内存实现
 - JdbcClientDetailsService：客户端详细信息服务的 JDBC 实现，它会从数据库中读取客户端详情

AccessTokenConverter

- 令牌服务实现的转换器接口，用于将令牌数据存储在令牌中
- 它有两个实现类：
 - JwtAccessTokenConverter：用于令牌 JWT 编码与解码
 - DefaultAccessTokenConverter：AccessTokenConverter 的默认实现

OAuth2AccessToken

- OAuth2 Token（令牌）实体类，包含了令牌、类型（Bearer）、失效时间等

OAuth2ClientProperties

- OAuth2 Client（客户端）属性配置实体类

OAuth2ProtectedResourceDetails

- OAuth2 受保护资源（资源服务器）的配置实体类

AuthorizationServerProperties

- OAuth2 Authentication（授权服务器）属性配置实体类

ResourceServerProperties

- OAuth2 为资源服务器配置提供了 ResourceServerProperties 类，该类会读取配置文件中对资源服务器得配置信息（如授权服务器公钥访问地址）

ClientDetails

- 客户端详情接口，声明了获取客户端详情所需的一些方法，比如获取访问令牌失效时间、获取客户端权限、获取客户端 ID、获取客户端访问范围等方法

AuthorizationServerProperties

- OAuth2授权服务器的配置属性

五、数据库表资料汇总

官方sql语句

```
-- used in tests that use HSQL
create table oauth_client_details (
  client_id VARCHAR(256) PRIMARY KEY,
  resource_ids VARCHAR(256),
  client_secret VARCHAR(256),
  scope VARCHAR(256),
  authorized_grant_types VARCHAR(256),
  web_server_redirect_uri VARCHAR(256),
  authorities VARCHAR(256),
  access_token_validity INTEGER,
  refresh_token_validity INTEGER,
  additional_information VARCHAR(4096),
  autoapprove VARCHAR(256)
);

create table oauth_client_token (
  token_id VARCHAR(256),
  token LONGVARBINARY,
  authentication_id VARCHAR(256) PRIMARY KEY,
  user_name VARCHAR(256),
  client_id VARCHAR(256)
);

create table oauth_access_token (
  token_id VARCHAR(256),
  token LONGVARBINARY,
  authentication_id VARCHAR(256) PRIMARY KEY,
  user_name VARCHAR(256),
  client_id VARCHAR(256),
  authentication LONGVARBINARY,
  refresh_token VARCHAR(256)
);

create table oauth_refresh_token (
  token_id VARCHAR(256),
  token LONGVARBINARY,
  authentication LONGVARBINARY
);

create table oauth_code (
  code VARCHAR(256), authentication LONGVARBINARY
);

create table oauth_approvals (
  userId VARCHAR(256),
```

```
    clientId VARCHAR(256),
    scope VARCHAR(256),
    status VARCHAR(10),
    expiresAt TIMESTAMP,
    lastModifiedAt TIMESTAMP
);

-- customized oauth_client_details table
create table ClientDetails (
    appId VARCHAR(256) PRIMARY KEY,
    resourceIds VARCHAR(256),
    appSecret VARCHAR(256),
    scope VARCHAR(256),
    grantTypes VARCHAR(256),
    redirectUrl VARCHAR(256),
    authorities VARCHAR(256),
    access_token_validity INTEGER,
    refresh_token_validity INTEGER,
    additionalInformation VARCHAR(4096),
    autoApproveScopes VARCHAR(256)
);
```

包含6张oauth需要的表 (oauth_client_details, oauth_client_token, oauth_access_token, oauth_refresh_token, oauth_code, oauth_approvals)

1张用户自定义表 (ClientDetails)

六、OAuth2 项目结构
