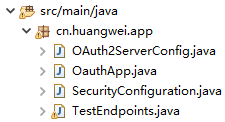
1. 工程结构

1.1项目代码



**pom文件设置**

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>cn.huangwei</groupId>  <artifactId>oauth-demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.7.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <dependencies>  <!-- 注意是starter,自动配置 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  <!-- 不是starter,手动配置 -->  <dependency>  <groupId>org.springframework.security.oauth</groupId>  <artifactId>spring-security-oauth2</artifactId>  <version>2.3.5.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies>  </project> |

**application.yml文件设置**

|  |
| --- |
| server:  port: 10011 |

**OauthApp.java文件设置**

|  |
| --- |
| package cn.huangwei.app;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  public class OauthApp {  public static void main(String[] args) {  SpringApplication.run(OauthApp.class, args);  }  } |

**Test EndPoint.java 文件配置**

|  |
| --- |
| **package** cn.huangwei.app;  **import** org.springframework.security.core.Authentication;  **import** org.springframework.security.core.context.SecurityContextHolder;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** TestEndpoints {  @GetMapping("/product/{id}")  **public** String getProduct(@PathVariable String id) {  //for debug  Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();  **return** "product id : " + id;  }  @GetMapping("/order/{id}")  **public** String getOrder(@PathVariable String id) {  //for debug  Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();  **return** "order id : " + id;  }  } |

**SecurityConfiguration.java文件设置**

|  |
| --- |
| **package** cn.huangwei.app;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.security.authentication.AuthenticationManager;  **import** org.springframework.security.config.annotation.web.builders.HttpSecurity;  **import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  **import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  **import** org.springframework.security.core.userdetails.User;  **import** org.springframework.security.core.userdetails.UserDetailsService;  **import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  **import** org.springframework.security.crypto.factory.PasswordEncoderFactories;  **import** org.springframework.security.crypto.password.PasswordEncoder;  **import** org.springframework.security.provisioning.InMemoryUserDetailsManager;  @Configuration  @EnableWebSecurity  **public** **class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {  @Bean  @Override  **protected** UserDetailsService userDetailsService() {  BCryptPasswordEncoder bCryptPasswordEncoder = **new** BCryptPasswordEncoder();  // password 方案三：支持多种编码，通过密码的前缀区分编码方式  String finalPassword = "{bcrypt}" + bCryptPasswordEncoder.encode("123456");  InMemoryUserDetailsManager manager = **new** InMemoryUserDetailsManager();  manager.createUser(User.*withUsername*("user\_1").password(finalPassword).authorities("USER").build());  manager.createUser(User.*withUsername*("user\_2").password(finalPassword).authorities("USER").build());  **return** manager;  }  // password 方案三：支持多种编码，通过密码的前缀区分编码方式,推荐  @Bean  PasswordEncoder passwordEncoder() {  **return** PasswordEncoderFactories.*createDelegatingPasswordEncoder*();  }  /\*\*  \* 这一步的配置是必不可少的，否则SpringBoot会自动配置一个AuthenticationManager,覆盖掉内存中的用户  \*/  @Bean  @Override  **public** AuthenticationManager authenticationManagerBean() **throws** Exception {  AuthenticationManager manager = **super**.authenticationManagerBean();  **return** manager;  }  @Override  **protected** **void** configure(HttpSecurity http) **throws** Exception {  http.requestMatchers().anyRequest().and().authorizeRequests().antMatchers("/oauth/\*").permitAll();  }  } |

**Oauth2ServerConfig.java文件设置**

|  |
| --- |
| **package** cn.huangwei.app;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.http.HttpMethod;  **import** org.springframework.security.authentication.AuthenticationManager;  **import** org.springframework.security.config.annotation.web.builders.HttpSecurity;  **import** org.springframework.security.config.http.SessionCreationPolicy;  **import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  **import** org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;  **import** org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;  **import** org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;  **import** org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;  **import** org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;  **import** org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;  **import** org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerSecurityConfigurer;  **import** org.springframework.security.oauth2.config.annotation.web.configurers.ResourceServerSecurityConfigurer;  **import** org.springframework.security.oauth2.provider.token.store.InMemoryTokenStore;  @Configuration  **public** **class** OAuth2ServerConfig {  **private** **static** **final** String ***DEMO\_RESOURCE\_ID*** = "order";  @Configuration  @EnableResourceServer  **protected** **static** **class** ResourceServerConfiguration **extends** ResourceServerConfigurerAdapter {  @Override  **public** **void** configure(ResourceServerSecurityConfigurer resources) {  resources.resourceId(***DEMO\_RESOURCE\_ID***).stateless(**true**);  }  @Override  **public** **void** configure(HttpSecurity http) **throws** Exception {  http  .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***IF\_REQUIRED***)  .and()  .requestMatchers().anyRequest()  .and()  .anonymous()  .and()  .authorizeRequests()  .antMatchers("/order/\*\*").authenticated();//配置order访问控制，必须认证过后才可以访问  }  }  @Configuration  @EnableAuthorizationServer  **protected** **static** **class** AuthorizationServerConfiguration **extends** AuthorizationServerConfigurerAdapter {  @Autowired  AuthenticationManager authenticationManager;    @Override  **public** **void** configure(ClientDetailsServiceConfigurer clients) **throws** Exception {  //配置两个客户端,一个用于password认证一个用于client认证  String finalSecret = "{bcrypt}"+**new** BCryptPasswordEncoder().encode("123456");  //配置两个客户端,一个用于password认证一个用于client认证  clients.inMemory().withClient("client\_1")  .resourceIds(***DEMO\_RESOURCE\_ID***)  .authorizedGrantTypes("client\_credentials", "refresh\_token")  .scopes("select")  .authorities("oauth2")  .secret(finalSecret)  .and().withClient("client\_2")  .resourceIds(***DEMO\_RESOURCE\_ID***)  .authorizedGrantTypes("password", "refresh\_token")  .scopes("select")  .authorities("oauth2")  .secret(finalSecret);  }  @Override  **public** **void** configure(AuthorizationServerEndpointsConfigurer endpoints) **throws** Exception {  endpoints  .tokenStore(**new** InMemoryTokenStore())  .authenticationManager(authenticationManager)  .allowedTokenEndpointRequestMethods(HttpMethod.***GET***, HttpMethod.***POST***);  }  @Override  **public** **void** configure(AuthorizationServerSecurityConfigurer oauthServer) **throws** Exception {  //允许表单认证  oauthServer.allowFormAuthenticationForClients();  }  }  } |

运行OauthApp工程文件，然后浏览器中输入client模式：

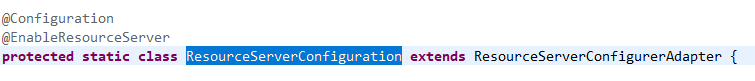
http://localhost:8080/oauth/token?grant\_type=client\_credentials&scope=select&client\_id=client\_1&client\_secret=123456

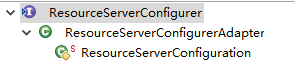
响应如下：

{"access\_token":"fbd5a92b-39b0-4330-b195-bd5518539f41","token\_type":"bearer","expires\_in":42102,"scope":"select"}

**1.2各个java 文件的作用**：

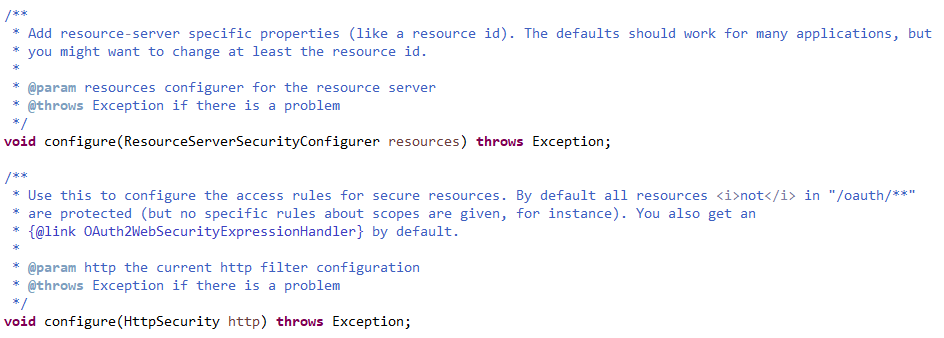
**资源服务器**OAuth2ServerConfig-ResourceServerConfiguration





使用@EnableResourceServer修饰的Configurer接口。实现此接口以调整受OAuth2安全性保护的访问规则和路径。应用程序可以提供此接口的多个实例，如果多个实例配置相同的属性，则使用最后一个配置的属性值。在应用之前，配置程序按{@link Order}排序。

该接口有两个方法



① 添加资源的相关属性

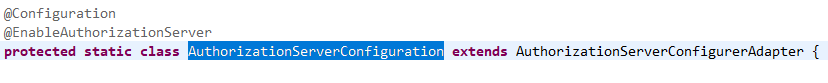
② 配置资源的安全访问规则，默认是所有在/oauth/\*\*的资源都应该被保护

一般去继承ResourceServerConfigurerAdapter这个接口类。然后实现相关方法。



@EnableResourceServer注解自动增加了一个类型为OAuth2AuthenticationProcessin-gFilter的过滤器链

**认证服务器**OAuth2ServerConfig-AuthorizationServerConfiguration



使用@EnableAuthorizationServer来修饰一个类，表明认证服务器配置类。

有三个方法：



①配置客户端的相关信息，id，secret，scope等信息

②配置认证endPoint的相关信息，tokenstore类型，authenticationManager,允许生成token的request method，例如get，most等。

③配置授权服务器/oauth/token endPoint的安全性，一般采用默认规则，即使不配置也能使用，创建ClientCredentialsTokenEndpointFilter核心过滤器



**TestEndpoints为controller用于验证安全访问的请求，不再赘述**

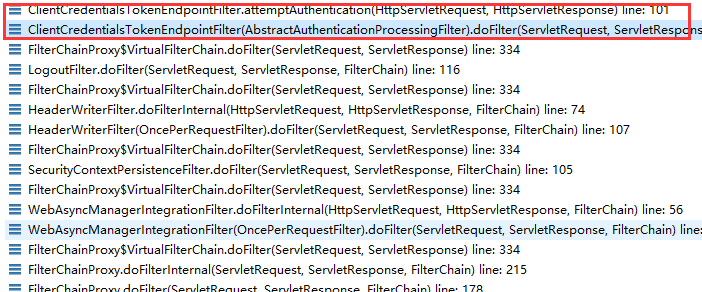
**SecurityConfiguration文件**



配置用户的信息，以及一些需要在认证服务器中使用的bean实例，config(HttpSecurity http)用户设置哪些方法能够通行，不需要经过安全认证

**2. Oauth2认证流程源码分析**

**2.1客户端认证**

从浏览器中输入网址之后，首先会通过Spring的相关过滤器进行拦截，filterchains中的过滤器一层一层执行doFilter， 

到ClientCredentialsTokenEndPointFilter，首先会调用其父类AbstractAuthenticationProcessingFilter的doFilter方法。



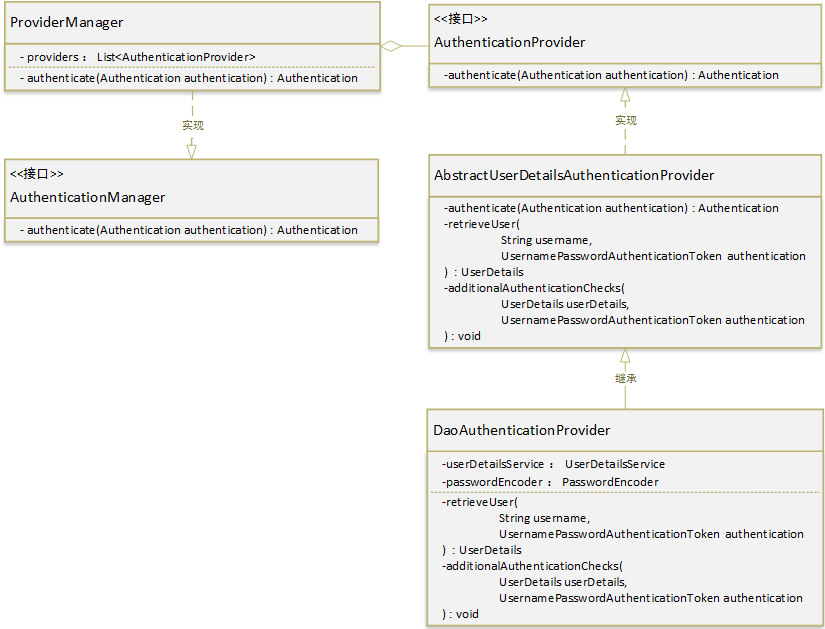
然后就会进入子类ClientCredentialsTokenEndPointFilter的attemptAuthentication方法。

|  |
| --- |
| **public** Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)  **throws** AuthenticationException, IOException, ServletException {  **if** (allowOnlyPost && !"POST".equalsIgnoreCase(request.getMethod())) {  **throw** **new** HttpRequestMethodNotSupportedException(request.getMethod(), **new** String[] { "POST" });  }  String clientId = request.getParameter("client\_id");  String clientSecret = request.getParameter("client\_secret");  // If the request is already authenticated we can assume that this  // filter is not needed  Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();  **if** (authentication != **null** && authentication.isAuthenticated()) {  **return** authentication;  }  **if** (clientId == **null**) {  **throw** **new** BadCredentialsException("No client credentials presented");  }  **if** (clientSecret == **null**) {  clientSecret = "";  }  clientId = clientId.trim();  UsernamePasswordAuthenticationToken authRequest = **new** UsernamePasswordAuthenticationToken(clientId,  clientSecret);  **return** **this**.getAuthenticationManager().authenticate(authRequest);  } |

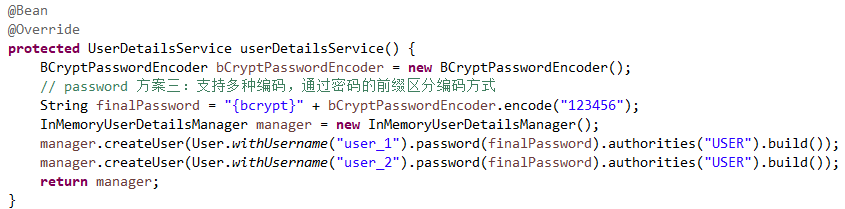
解析：从request中获取clientId和clientsecret，**Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();检查当前的request，实质应该为client，是否已经被认证过，如果认证过，直接返回authentication，否则进行下面的认证。**将clientId和clientsecret封装成UsernamePasswordAuthenticationToken；使用AuthenticationManager的authenticate方法去对UsernamePasswordAuthenticationToken进行客户端的认证。

**AuthenticationManager：**

使用容器中的顶级身份管理器AuthenticationManager去进行身份认证（AuthenticationManager的实现类一般是ProviderManager。而ProviderManager内部维护了一个List,真正的身份认证是由一系列AuthenticationProvider去完成。而AuthenticationProvider的常用实现类则是DaoAuthenticationProvider。

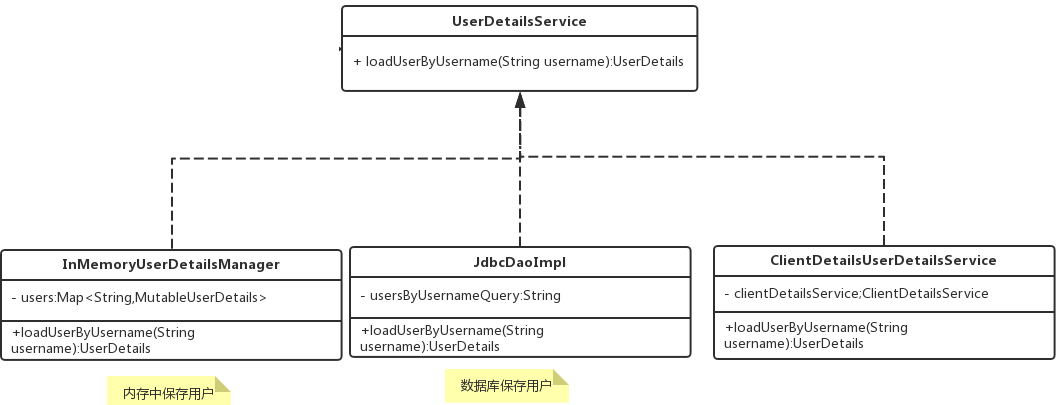


DaoAuthenticationProvider内部又聚合了一个UserDetailsService接口，UserDetailsService才是获取用户详细信息的最终接口，而我们上面的文章中在内存中配置用户，就是使用了UserDetailsService的一个实现类InMemoryUserDetailsManager）。



client模式是没有用户的概念的，运行中会将UserDetailsService适配成ClientDetailsUserDetailsService

UserDetailsService的继承关系如下



**AuthenticationManager的认证流程**

AuthenticationManager是接口类，因此分析它的实现类 ProviderManager，ProviderManager 的 authenticate() 方法代码如下：

|  |
| --- |
| **public** Authentication authenticate(Authentication authentication)  **throws** AuthenticationException {  。。。  。。。  **for** (AuthenticationProvider provider : getProviders()) {  **。。。**  **try** {  **result = provider.authenticate(authentication);**  **if** (result != **null**) {  copyDetails(authentication, result);  **break**;  }  }  。。。  **if** (result == **null** && parent != **null**) {  // Allow the parent to try.  **try** {  **result = parentResult = parent.authenticate(authentication);**  }  **。。。**  } |

主要是**result = provider.authenticate(authentication);**

AuthenticationProvider是个接口，主要分析他的实现类AbstractUserDetailsAuthenticationProvider的子类DaoAuthenticationProvider，它调用父类AbstractUserDetailsAuthenticationProvider中的**authenticate方法。**

|  |
| --- |
| **public** Authentication authenticate(Authentication authentication)  **throws** AuthenticationException {  。。。  // Determine username  String username = (authentication.getPrincipal() == **null**) ? "NONE\_PROVIDED"  : authentication.getName();  **boolean** cacheWasUsed = **true**;  UserDetails user = **this**.userCache.getUserFromCache(username);  **if** (user == **null**) {  cacheWasUsed = **false**;  **try** {  user = retrieveUser(username,  (UsernamePasswordAuthenticationToken) authentication);  }  **。。。**  **try** {  preAuthenticationChecks.check(user);  additionalAuthenticationChecks(user,  (UsernamePasswordAuthenticationToken) authentication);  }  **catch** (AuthenticationException exception) {  **if** (cacheWasUsed) {  // There was a problem, so try again after checking  // we're using latest data (i.e. not from the cache)  cacheWasUsed = **false**;  user = retrieveUser(username,  (UsernamePasswordAuthenticationToken) authentication);  preAuthenticationChecks.check(user);  additionalAuthenticationChecks(user,  (UsernamePasswordAuthenticationToken) authentication);  }  **else** {  **throw** exception;  }  }  postAuthenticationChecks.check(user);  **if** (!cacheWasUsed) {  **this**.userCache.putUserInCache(user);  }  Object principalToReturn = user;  **if** (forcePrincipalAsString) {  principalToReturn = user.getUsername();  }  **return** createSuccessAuthentication(principalToReturn, authentication, user);  } |

user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);

调用的就是DaoAuthenticationProvider的方法，具体如下

|  |
| --- |
| **protected** **final** UserDetails retrieveUser(String username,  UsernamePasswordAuthenticationToken authentication)  **throws** AuthenticationException {  prepareTimingAttackProtection();  **try** {  UserDetails loadedUser = **this**.getUserDetailsService().loadUserByUsername(username);  。。。  **return** loadedUser;  。。。 |

**从数据库中获取user，像mybatis的调用格式**

preAuthenticationChecks.check(user);

additionalAuthenticationChecks(user,

(UsernamePasswordAuthenticationToken) authentication);

这两个方法是对获得的user进行验证

其中addtionalAuthenticationChecks代码如下：

|  |
| --- |
| **protected** **void** additionalAuthenticationChecks(UserDetails userDetails,  UsernamePasswordAuthenticationToken authentication)  **throws** AuthenticationException {  **。。。**  String presentedPassword = authentication.getCredentials().toString();  **if** (!passwordEncoder.matches(presentedPassword, userDetails.getPassword())) {  。。。  } |

使用passwordEncoder的校验方法。这样authenticationManager的认证就结束了。

**2.2 token的生成**

通过过滤器和认证管理器之后，客户端的前置认证基本完成了，现在分析token的产生

涉及到**TokenEndpoint**

|  |
| --- |
| **public** **class** TokenEndpoint **extends** AbstractEndpoint {  **。。。**  @RequestMapping(value = "/oauth/token", method=RequestMethod.***GET***)  **public** ResponseEntity<OAuth2AccessToken> getAccessToken(Principal principal, @RequestParam  Map<String, String> parameters) **throws** HttpRequestMethodNotSupportedException {  **。。。**  **return** **postAccessToken(principal, parameters);**  }    @RequestMapping(value = "/oauth/token", method=RequestMethod.***POST***)  **public ResponseEntity<OAuth2AccessToken> postAccessToken(Principal principal, @RequestParam**  **Map<String, String> parameters) throws HttpRequestMethodNotSupportedException** {  。。。  String clientId = getClientId(principal);  ClientDetails authenticatedClient = getClientDetailsService().loadClientByClientId(clientId);**---①**  TokenRequest tokenRequest = getOAuth2RequestFactory().createTokenRequest(parameters, authenticatedClient);**----②**  **。。。**  OAuth2AccessToken token = getTokenGranter().grant(tokenRequest.getGrantType(), tokenRequest);**----③**  **。。。**  **return** getResponse(token);  }  。。。  } |

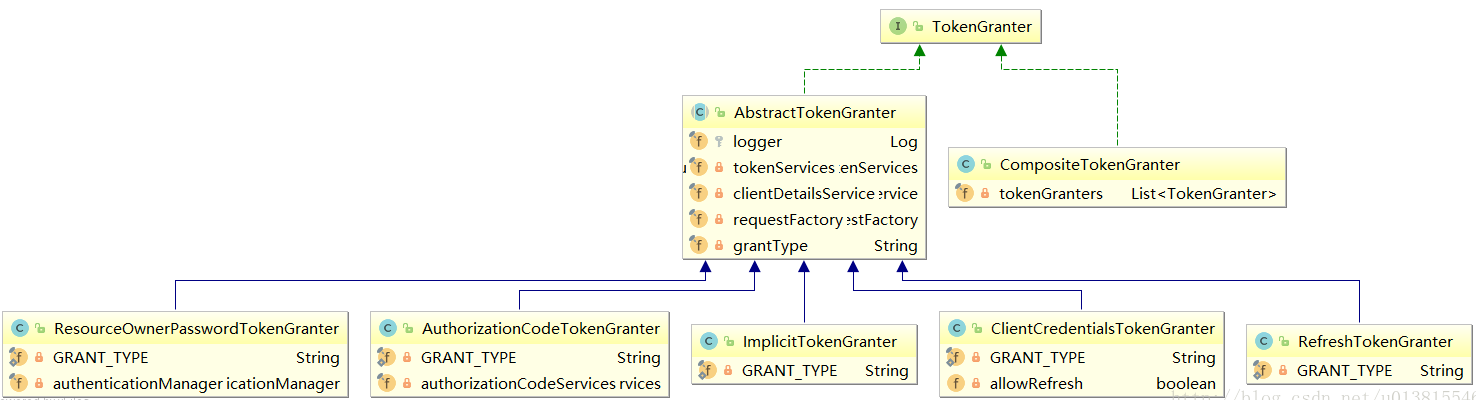
①加载客户端信息

②结合请求信息clientId，scope，grantType，创建TokenRequest

③将TokenRequest传递给TokenGranter颁发token

getTokenGranter().grant(tokenRequest.getGrantType())；

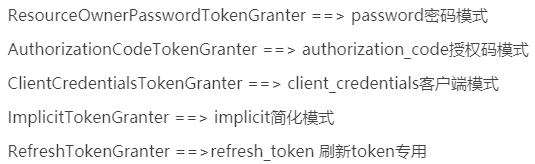
TokenGranter来生成AccessToken，具体继承关系如下



TokenGranter的设计思路是使用CompositeTokenGranter管理一个List列表，每一种grantType对应一个具体的真正授权者，在debug过程中可以发现CompositeTokenGranter 内部就是在循环调用五种TokenGranter实现类的grant方法，而granter内部则是通过grantType来区分是否是各自的授权类型。

|  |
| --- |
| **public** **class** CompositeTokenGranter **implements** TokenGranter {  **。。。**    **public** OAuth2AccessToken grant(String grantType, TokenRequest tokenRequest) {  **for** (TokenGranter granter : tokenGranters) {  OAuth2AccessToken grant = granter.grant(grantType, tokenRequest);  **if** (grant!=**null**) {  **return** grant;  }  }  **return** **null**;  }    **。。。**  } |

五种类型分别是



以客户端模式为例，分析如何产生token，从他们的父类AbstractTokenGranter入手。

|  |
| --- |
| **public** **abstract** **class** AbstractTokenGranter **implements** TokenGranter {  。。  **private** **final** AuthorizationServerTokenServices tokenServices;  **private** **final** ClientDetailsService clientDetailsService;    **private** **final** OAuth2RequestFactory requestFactory;    **private** **final** String grantType;  **。。。**  **public** OAuth2AccessToken grant(String grantType, TokenRequest tokenRequest) {  **if** (!**this**.grantType.equals(grantType)) {  **return** **null**;  }    String clientId = tokenRequest.getClientId();  ClientDetails client = clientDetailsService.loadClientByClientId(clientId);  validateGrantType(grantType, client);    logger.debug("Getting access token for: " + clientId);  **return** getAccessToken(client, tokenRequest);  }  grant方法通过tokenRequest获取client和用户的相关信息，通过getAcessToken方法获取token，其中getOAuth2Authentication根据client和tokenRequest获得OAuth2Request，其中包含用户的相关信息。生成token  **protected** OAuth2AccessToken getAccessToken(ClientDetails client, TokenRequest tokenRequest) {  **return** tokenServices.createAccessToken(getOAuth2Authentication(client, tokenRequest));  }  **protected** OAuth2Authentication getOAuth2Authentication(ClientDetails client, TokenRequest tokenRequest) {  OAuth2Request storedOAuth2Request = requestFactory.createOAuth2Request(client, tokenRequest);  **return** **new** OAuth2Authentication(storedOAuth2Request, **null**);  }  **protected** **void** validateGrantType(String grantType, ClientDetails clientDetails) {  。。。  }  **。。。**  } |

重点是tokenServices的createAccessToken方法。

|  |
| --- |
| public interface AuthorizationServerTokenServices {  //创建token  OAuth2AccessToken createAccessToken(OAuth2Authentication authentication) throws AuthenticationException;  //刷新token  OAuth2AccessToken refreshAccessToken(String refreshToken, TokenRequest tokenRequest)  throws AuthenticationException;  //获取token  OAuth2AccessToken getAccessToken(OAuth2Authentication authentication);  } |

它的实现类为



|  |
| --- |
| public class **DefaultTokenServices** implements AuthorizationServerTokenServices, ResourceServerTokenServices,  ConsumerTokenServices, InitializingBean {  。。。  。。。  @Transactional  public OAuth2AccessToken createAccessToken(OAuth2Authentication authentication) throws AuthenticationException {  OAuth2AccessToken existingAccessToken = tokenStore.getAccessToken(authentication);  OAuth2RefreshToken refreshToken = null;  if (existingAccessToken != null) {  if (existingAccessToken.isExpired()) {  if (existingAccessToken.getRefreshToken() != null) {  refreshToken = existingAccessToken.getRefreshToken();  // The token store could remove the refresh token when the  // access token is removed, but we want to  // be sure...  tokenStore.removeRefreshToken(refreshToken);  }  tokenStore.removeAccessToken(existingAccessToken);  }  else {  // Re-store the access token in case the authentication has changed  tokenStore.storeAccessToken(existingAccessToken, authentication);  return existingAccessToken;  }  }  。。。  if (refreshToken == null) {  refreshToken = createRefreshToken(authentication);  }  。。。  else if (refreshToken instanceof ExpiringOAuth2RefreshToken) {  ExpiringOAuth2RefreshToken expiring = (ExpiringOAuth2RefreshToken) refreshToken;  if (System.currentTimeMillis() > expiring.getExpiration().getTime()) {  refreshToken = createRefreshToken(authentication);  }  }  OAuth2AccessToken accessToken = createAccessToken(authentication, refreshToken);  tokenStore.storeAccessToken(accessToken, authentication);  // In case it was modified  refreshToken = accessToken.getRefreshToken();  if (refreshToken != null) {  tokenStore.storeRefreshToken(refreshToken, authentication);  }  return accessToken;  }  。。。  } |

获取当前授权的token作为existingAccessToken，如果当前token不为空过期了，就将该token对应的refrestToken一起删除；

如果没有过期，就继续使用当前的token；如果当前的existingAccessToken是null说明没有token，如果过期token没有一个关联的refreshToken，就只能创建一个refreshToken。

使用新的refreshToken和authentication创建新的accessToken

使用tokenStore将token和authentication一起存储起来。

|  |
| --- |
| **private** OAuth2AccessToken createAccessToken(OAuth2Authentication authentication, OAuth2RefreshToken refreshToken) {  DefaultOAuth2AccessToken token = **new** DefaultOAuth2AccessToken(UUID.*randomUUID*().toString());  **int** validitySeconds = getAccessTokenValiditySeconds(authentication.getOAuth2Request());  **if** (validitySeconds > 0) {  token.setExpiration(**new** Date(System.*currentTimeMillis*() + (validitySeconds \* 1000L)));  }  token.setRefreshToken(refreshToken);  token.setScope(authentication.getOAuth2Request().getScope());  **return** accessTokenEnhancer != **null** ? accessTokenEnhancer.enhance(token, authentication) : token;  }  新的token的生成，使用uuId生成，然后设置refreshToken和scope。 |

上面就是token 生成的大致流程。

**3.携带token访问受限资源**

携带token访问资源，这涉及到@EnableResourceServer相关的资源服务的配置。

|  |
| --- |
| @Configuration  @EnableResourceServer  **protected** **static** **class** ResourceServerConfiguration **extends** ResourceServerConfigurerAdapter {  @Override  **public** **void** configure(ResourceServerSecurityConfigurer resources) {  resources.resourceId(***DEMO\_RESOURCE\_ID***).stateless(**true**);  }  @Override  **public** **void** configure(HttpSecurity http) **throws** Exception {  http  .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***IF\_REQUIRED***)  .and()  .requestMatchers().anyRequest()  .and()  .anonymous()  .and()  .authorizeRequests()  .antMatchers("/order/\*\*").authenticated();//配置order访问控制，必须认证过后才可以访问  }  } |

涉及到ResourceServerSecurityConfigurer和HttpSecurity；前者与资源安全配置相关，后者与http安全配置相关。

ResourceServerSecurityConfigurer主要是这个方法configure

|  |
| --- |
| @Override  **public** **void** configure(HttpSecurity http) **throws** Exception {  AuthenticationManager oauthAuthenticationManager = oauthAuthenticationManager(http);  resourcesServerFilter = **new** OAuth2AuthenticationProcessingFilter();  resourcesServerFilter.setAuthenticationEntryPoint(authenticationEntryPoint);  resourcesServerFilter.setAuthenticationManager(oauthAuthenticationManager);  **if** (eventPublisher != **null**) {  resourcesServerFilter.setAuthenticationEventPublisher(eventPublisher);  }  **if** (tokenExtractor != **null**) {  resourcesServerFilter.setTokenExtractor(tokenExtractor);  }  **if** (authenticationDetailsSource != **null**) {  resourcesServerFilter.setAuthenticationDetailsSource(authenticationDetailsSource);  }  resourcesServerFilter = postProcess(resourcesServerFilter);  resourcesServerFilter.setStateless(stateless);  // @formatter:off  http  .authorizeRequests().expressionHandler(expressionHandler)  .and()  .addFilterBefore(resourcesServerFilter, AbstractPreAuthenticatedProcessingFilter.**class**)  .exceptionHandling()  .accessDeniedHandler(accessDeniedHandler)  .authenticationEntryPoint(authenticationEntryPoint);  // @formatter:on  } |

注意点：

-- OAuth2AuthenticationProcessingFilter

--为OAuth2AuthenticationProcessingFilter提供固定的AuthenticationManager即OAuth2AuthenticationManager，它并没有将OAuth2AuthenticationManager添加到spring的容器中，不然可能会影响spring security的普通认证流程（非oauth2请求），只有被OAuth2AuthenticationProcessingFilter拦截到的oauth2相关请求才被特殊的身份认证器处理

--设置了TokenExtractor默认的实现—-BearerTokenExtractor

-- 相关的异常处理器，可以重写相关实现，自定义异常。

通过注解和配置，使得我们在访问资源的时候，请求会被OAuth2AuthenticationProcessingFilter拦截。

<http://localhost:8080/order/1?access_token=950a7cc9-5a8a-42c9-a693-40e817b1a4b0>

正常访问这个url，会进入OAuth2AuthenticationProcessingFilter的过滤器中。

**3.1 OAuth2AuthenticationProcessingFilte**

|  |
| --- |
| **public** **void** doFilter(ServletRequest req, ServletResponse res, FilterChain chain) **throws** IOException, ServletException {  。。  **final** HttpServletRequest request = (HttpServletRequest) req;  **final** HttpServletResponse response = (HttpServletResponse) res;  **try** {  //**从请求中取出身份信息，即access\_token**  Authentication authentication = tokenExtractor.extract(request);    **if** (authentication == **null**) {  **。。。**  **else** {  request.setAttribute(OAuth2AuthenticationDetails.***ACCESS\_TOKEN\_VALUE***, authentication.getPrincipal());  **if** (authentication **instanceof** AbstractAuthenticationToken) {  AbstractAuthenticationToken needsDetails = (AbstractAuthenticationToken) authentication;  needsDetails.setDetails(authenticationDetailsSource.buildDetails(request));  }  **//身份认证**  Authentication authResult = authenticationManager.authenticate(authentication);  。。。  eventPublisher.publishAuthenticationSuccess(authResult);  **//将身份认证绑定到SecurityContextHolder**  SecurityContextHolder.*getContext*().setAuthentication(authResult);  }  }  **catch** (OAuth2Exception failed) {  。。。  **return**;  }  chain.doFilter(request, response);  } |

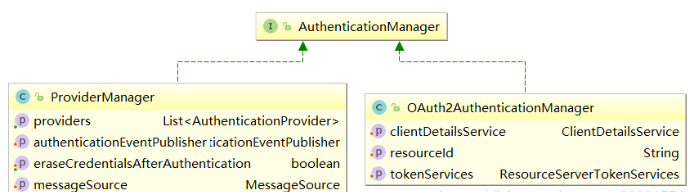
OAuth2保护资源的预先认证过滤器。如果与OAuth2AuthenticationManager结合使用，则会从到来的请求之中提取一个OAuth2 token

其中涉及到两个关键的类TokenExtractor，AuthenticationManager。

OAuth2的身份管理器-**OAuth2AuthenticationManager**

|  |
| --- |
| **private** AuthenticationManager oauthAuthenticationManager(HttpSecurity http) {  OAuth2AuthenticationManager oauthAuthenticationManager = **new** OAuth2AuthenticationManager();  **if** (authenticationManager != **null**) {  **if** (authenticationManager **instanceof** OAuth2AuthenticationManager) {  oauthAuthenticationManager = (OAuth2AuthenticationManager) authenticationManager;  }  **else** {  **return** authenticationManager;  }  }  oauthAuthenticationManager.setResourceId(resourceId);  oauthAuthenticationManager.setTokenServices(resourceTokenServices(http));  oauthAuthenticationManager.setClientDetailsService(clientDetails());  **return** oauthAuthenticationManager;  } |

从这里可以看出，OAuth2AuthenticationManager也是AuthenticationManager的实现类，而且这里的验证token就是用OAuth2AuthenticationManager，与之前的providerManager不同，providerManager是用来获取token的。



该类的认证方法如下。

|  |
| --- |
| **public** Authentication authenticate(Authentication authentication) **throws** AuthenticationException {  **if** (authentication == **null**) {  **throw** **new** InvalidTokenException("Invalid token (token not found)");  }  String token = (String) authentication.getPrincipal();  OAuth2Authentication auth = **tokenServices.loadAuthentication(token);**  **。。。**  **//核对客户端以及用户的相关信息**  checkClientDetails(auth);  **if** (authentication.getDetails() **instanceof** OAuth2AuthenticationDetails) {  。。。  }  auth.setDetails(authentication.getDetails());  auth.setAuthenticated(**true**);  **return** auth;  } |

这个tokenService是ResourceServerTokenServices类型，根据用户携带的token码去加载token；

TokenExtract的实现类BearerTokenExtractor，调用extractToken方法，

|  |
| --- |
| **protected** String extractToken(HttpServletRequest request) {  // first check the header...  String token = extractHeaderToken(request);  // bearer type allows a request parameter as well  **if** (token == **null**) {  ***logger***.debug("Token not found in headers. Trying request parameters.");  token = request.getParameter(OAuth2AccessToken.***ACCESS\_TOKEN***);  **if** (token == **null**) {  ***logger***.debug("Token not found in request parameters. Not an OAuth2 request.");  }  **else** {  request.setAttribute(OAuth2AuthenticationDetails.***ACCESS\_TOKEN\_TYPE***, OAuth2AccessToken.***BEARER\_TYPE***);  }  }  **return** token;  }  **protected** String extractHeaderToken(HttpServletRequest request) {  Enumeration<String> headers = request.getHeaders("Authorization");  **while** (headers.hasMoreElements()) { // typically there is only one (most servers enforce that)  String value = headers.nextElement();  **if** ((value.toLowerCase().startsWith(OAuth2AccessToken.***BEARER\_TYPE***.toLowerCase()))) {  String authHeaderValue = value.substring(OAuth2AccessToken.***BEARER\_TYPE***.length()).trim();  // Add this here for the auth details later. Would be better to change the signature of this method.  request.setAttribute(OAuth2AuthenticationDetails.***ACCESS\_TOKEN\_TYPE***,  value.substring(0, OAuth2AccessToken.***BEARER\_TYPE***.length()).trim());  **int** commaIndex = authHeaderValue.indexOf(',');  **if** (commaIndex > 0) {  authHeaderValue = authHeaderValue.substring(0, commaIndex);  }  **return** authHeaderValue;  }  }  **return** **null**;  } |

它的作用在于分离出请求中包含的token。也启示了我们可以使用多种方式携带token。

1 在Header中携带

http://localhost:8080/order/1

Header：

Authentication：Bearer f732723d-af7f-41bb-bd06-2636ab2be135

2 拼接在url中作为requestParam

http://localhost:8080/order/1?access\_token=f732723d-af7f-41bb-bd06-2636ab2be135

3 在form表单中携带

http://localhost:8080/order/1

form param：

access\_token=f732723d-af7f-41bb-bd06-2636ab2be135