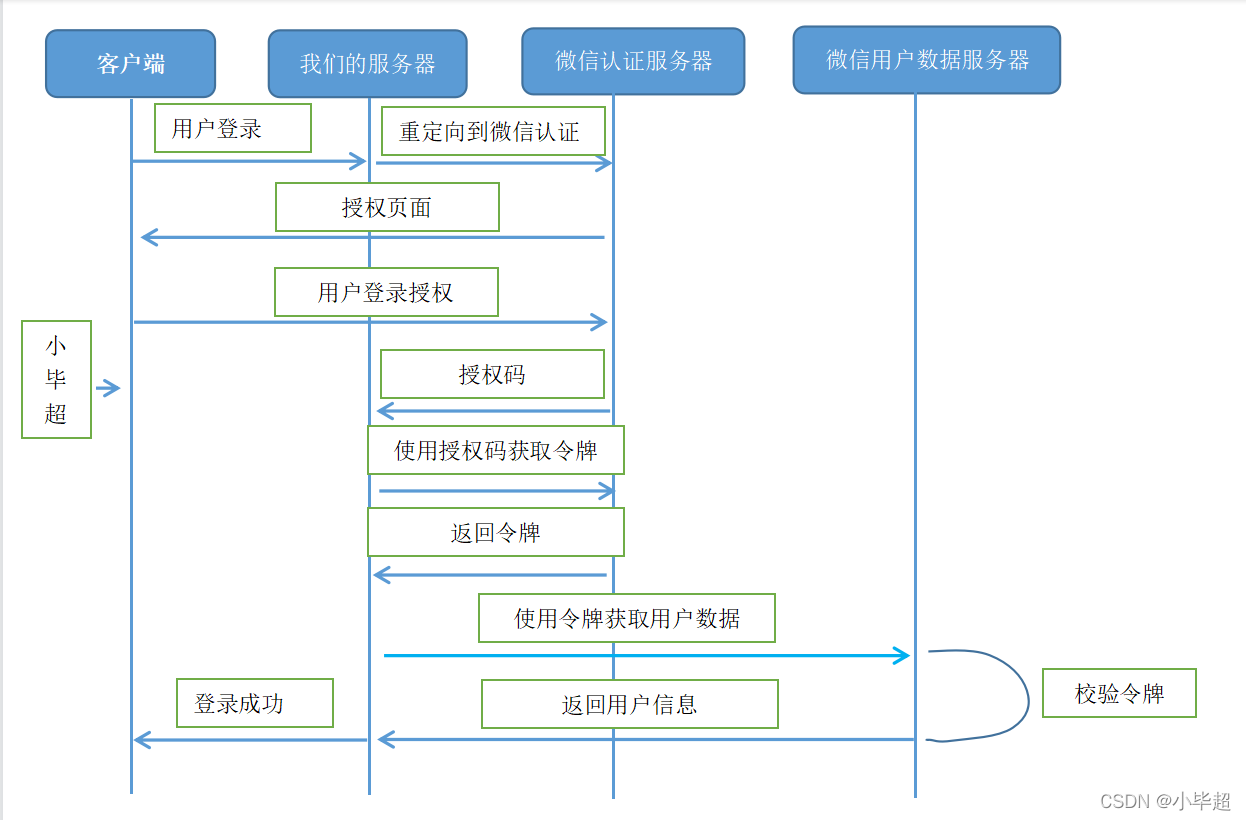
OAuth（开放授权）是一个开放标准，允许用户授权第三方应用访问他们存储在另外的服务提供者上的信息，而不 需要将用户名和密码提供给第三方应用或分享他们数据的所有内容。OAuth2.0是OAuth协议的延续版本，但不向 后兼容OAuth 1.0即完全废止OAuth1.0。微信登录的过程，就是Aauth2的授权码模式的一个实现：



上面这种是授权码的方式，这种方式安全级别较高，另外Oauth2还提供了密码模式、客户端模式、简化模式，但不仅限于这四种，用户完全可以自定义认证授权过程，要知道Oauth2只是一个协议，并不限于具体哪个框架，SpringOauth2是Oauth2的一个实现。

由于Oauth2可以解决分布式情况下的授权认证或第三方的授权认证，已经不是单体的架构，其中肯定有着某种的分工合作，其中在Oauth2中一般可以分为四个角色，包括：客户端、资源拥有者、授权服务器（也称认证服务器）、资源服务器。

* 客户端可以理解为web浏览器、移动客户端、微信小程序等，就是用户看到的前端。
* 资源拥有者通常指的是用户，也可以是应用程序，即该资源的最终给了谁。
* 授权服务器，也称认证服务器，主要用户对用户的身份认证即权限的授权，以及全局令牌的颁发，就如上面微信认证的微信认证服务器。
* 资源服务器也就是提供资源的服务器，就比如上面微信认证的微信用户数据服务器，存储着我闷需要的资源。

# 基于授权码的认证授权

要通过继承AuthorizationServerConfigurerAdapter这个类来配制信息，并声明@EnableAuthorizationServer注解，表示这是认证服务。其中AuthorizationServerConfigurerAdapter配制类，有三个方法：

1. **public** **class** AuthorizationServerConfigurerAdapter **implements** AuthorizationServerConfigurer {
2. **public** AuthorizationServerConfigurerAdapter() {
3. }
5. **public** **void** configure(AuthorizationServerSecurityConfigurer security) **throws** Exception {
6. }
8. **public** **void** configure(ClientDetailsServiceConfigurer clients) **throws** Exception {
9. }
11. **public** **void** configure(AuthorizationServerEndpointsConfigurer endpoints) **throws** Exception {
12. }
13. }

ClientDetailsServiceConfigurer用来配置客户端详情服务，客户端详情信息在 这里进行初始化，能够把客户端详情信息写死在这里或者是通过数据库来存储调取详情信息。能够使用内存或者JDBC来实现客户端详情服务（ClientDetailsService），而ClientDetails有几个重要的属性如下：

* clientId：（必须的）用来标识客户的Id。
* secret：（需要值得信任的客户端）客户端安全码，如果有的话。
* scope：用来限制客户端的访问范围，如果为空（默认）的话，那么客户端拥有全部的访问范围。
* authorizedGrantTypes：此客户端可以使用的授权类型，默认为空。
* authorities：此客户端可以使用的权限（基于Spring Security authorities）。

AuthorizationServerEndpointsConfigurer用来配置令牌（token）的访问端点和令牌服务(token services)，可以通过设定以下属性决定支持的授权类型（Grant Types）：

* 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：如果你设置了这个属性的话，那说明你有一个自己的UserDetailsService 接口的实现， 或者你可以把这个东西设置到全局域上面去（例如 GlobalAuthenticationManagerConfigurer 这个配置对 象），当你设置了这个之后，那么 “refresh\_token” 即刷新令牌授权类型模式的流程中就会包含一个检查，用 来确保这个账号是否仍然有效，假如说你禁用了这个账户的话。
* authorizationCodeServices：这个属性是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。
* 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：当你设置了这个东西（即 TokenGranter 接口实现），那么授权将会交由你来完全掌控，并 且会忽略掉上面的这几个属性，这个属性一般是用作拓展用途的，即标准的四种授权模式已经满足不了你的 需求的时候，才会考虑使用这个。

## 认证服务器

AuthorizationServerSecurityConfigurer用来配置令牌端点的安全约束。permitAll()表示允许所有人。首先声明出Token的模式，这里先将Token存放在内存中：

1. @Configuration
2. **public** **class** TokenConfig {
3. @Bean
4. **public** TokenStore tokenStore() {
5. **return** **new** InMemoryTokenStore();
6. }
7. }
9. @Configuration
10. @EnableAuthorizationServer
11. **public** **class** AuthorizationServerConfiguration **extends** AuthorizationServerConfigurerAdapter {
13. @Autowired
14. **private** AuthenticationManager authenticationManager;
16. @Autowired
17. **private** TokenStore tokenStore;
19. @Autowired
20. **private** ClientDetailsService clientDetailsService;
22. @Autowired
23. **private** AuthorizationCodeServices authorizationCodeServices;
25. /\*\*
26. \*  用来配置客户端详情服务（客户端详情信息在这里进行初始化）
27. \*
28. \* @param clients
29. \* @throws Exception
30. \*/
31. @Override
32. **public** **void** configure(ClientDetailsServiceConfigurer clients) **throws** Exception {
33. String finalSecret = "{bcrypt}" + **new** BCryptPasswordEncoder().encode("123456");
34. // 配置两个客户端，一个用于授权码认证
35. clients.inMemory().withClient("client\_1")
36. .resourceIds(Utils.RESOURCEIDS.ORDER)
37. // 该client允许的授权类型 authorization\_code,password,refresh\_token,implicit,client\_credentials
38. .authorizedGrantTypes("authorization\_code", "password", "client\_credentials", "implicit", "refresh\_token")
39. .scopes("all")
40. .secret(finalSecret)
41. .autoApprove(**false**) //加上验证回调地址
42. .authorities("admin")
43. .redirectUris("http://www.baidu.com");
44. }
46. @Bean
47. **public** AuthorizationServerTokenServices tokenService() {
48. DefaultTokenServices service = **new** DefaultTokenServices();
49. service.setClientDetailsService(clientDetailsService);
50. service.setSupportRefreshToken(**true**);
51. service.setTokenStore(tokenStore);
52. service.setAccessTokenValiditySeconds(7200); // 令牌默认有效期2小时
53. service.setRefreshTokenValiditySeconds(259200); // 刷新令牌默认有效期3天
54. **return** service;
55. }
57. //设置授权码模式的授权码如何存取，暂时采用内存方式
58. @Bean
59. **public** AuthorizationCodeServices authorizationCodeServices() {
60. **return** **new** InMemoryAuthorizationCodeServices();
61. }
63. /\*\*
64. \* 用来配置授权（authorization）以及令牌（token）的访问端点和令牌服务(token services)，还有token的存储方式(tokenStore)
65. \*
66. \* @param endpoints
67. \* @throws Exception
68. \*/
69. @Override
70. **public** **void** configure(AuthorizationServerEndpointsConfigurer endpoints) **throws** Exception {
71. endpoints
72. .authenticationManager(authenticationManager)//认证管理器
73. .authorizationCodeServices(authorizationCodeServices)//授权码服务
74. .tokenServices(tokenService())//令牌管理服务
75. .allowedTokenEndpointRequestMethods(HttpMethod.POST);
76. }
78. /\*\*
79. \* 用来配置令牌端点(Token Endpoint)的安全约束；
80. \* @param security
81. \* @throws Exception
82. \*/
83. @Override
84. **public** **void** configure(AuthorizationServerSecurityConfigurer security) **throws** Exception {
85. // 允许表单认证
86. security.tokenKeyAccess("permitAll()")                    //oauth/token\_key是公开
87. .checkTokenAccess("permitAll()")                  //oauth/check\_token公开
88. .allowFormAuthenticationForClients();
89. }
90. }
92. @Configuration
93. @EnableGlobalMethodSecurity(securedEnabled = **true**,prePostEnabled = **true**)
94. **public** **class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {
96. @Bean
97. **public** PasswordEncoder passwordEncoder() {
98. **return** PasswordEncoderFactories.createDelegatingPasswordEncoder();
99. }
101. //认证管理器
102. @Bean
103. @Override
104. **public** AuthenticationManager authenticationManagerBean() **throws** Exception {
105. AuthenticationManager manager = **super**.authenticationManagerBean();
106. **return** manager;
107. }
109. @Override
110. **public** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {
111. auth.inMemoryAuthentication().withUser("admin").password(**new** BCryptPasswordEncoder().encode("1234")).authorities("admin");
112. auth.inMemoryAuthentication().withUser("common").password(**new** BCryptPasswordEncoder().encode("1234")).authorities("common");
113. }

116. /\*\*
117. \* 中配置所有请求的安全验证
118. \*
119. \* @param http
120. \* @throws Exception
121. \*/
122. @Override
123. **protected** **void** configure(HttpSecurity http) **throws** Exception {
124. http.csrf().disable()
125. .authorizeRequests()
126. .antMatchers("/auth/\*").permitAll()
127. .anyRequest().authenticated()
128. .and()
129. .formLogin();
130. }
131. }

## 资源服务器

资源服务继承ResourceServerConfigurerAdapter，其中指定的TokenStore要和auth服务的一致，且依旧需要WebSecurityConfig控制授权服务：

1. @Configuration
2. @EnableResourceServer
3. **public** **class** ResourceServerConfig **extends** ResourceServerConfigurerAdapter {
4. **public** **static** **final** String RESOURCE\_ID = "res1";
6. @Autowired
7. **private** TokenStore tokenStore;
9. @Bean
10. **public** TokenStore tokenStore() {
11. **return** **new** InMemoryTokenStore();
12. }
14. @Override
15. **public** **void** configure(ResourceServerSecurityConfigurer resources) {
16. resources.resourceId(RESOURCE\_ID)//资源 id
17. .tokenStore(tokenStore)
18. .tokenServices(tokenService())//验证令牌的服务
19. .stateless(**true**);
20. }
22. @Override
23. **public** **void** configure(HttpSecurity http) **throws** Exception {
24. http.authorizeRequests()
25. .antMatchers("/admin/\*\*").hasAuthority("admin")
26. .antMatchers("/common/\*\*").hasAuthority("common")
27. .and().csrf().disable()
28. .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
29. }
31. //资源服务令牌解析服务
32. @Bean
33. **public** ResourceServerTokenServices tokenService() {
34. //使用远程服务请求授权服务器校验token,必须指定校验token 的url、client\_id，client\_secret
35. RemoteTokenServices service = **new** RemoteTokenServices();
36. service.setCheckTokenEndpointUrl("http://localhost:9098/oauth/check\_token");
37. service.setClientId("client\_1");
38. service.setClientSecret("secret");
39. **return** service;
40. }
42. }
44. @Configuration
45. @EnableGlobalMethodSecurity(securedEnabled = **true**,prePostEnabled = **true**)
46. **public** **class** WebSecurityConfig **extends** WebSecurityConfigurerAdapter {
48. //安全拦截机制（最重要）
49. @Override
50. **protected** **void** configure(HttpSecurity http) **throws** Exception {
51. http.csrf().disable()
52. .authorizeRequests()
53. .antMatchers("/\*\*").authenticated()
54. .anyRequest().permitAll();
55. }
57. }

## 测试

直接访问<http://localhost:8081/provider/depart/list>是不行的



第一步就是先获取授权码，向浏览器访问下面路径：

<http://localhost:9098/oauth/authorize?client_id=client_1&response_type=code&scope=all&redirect_uri=http://www.baidu.com>

然后输入密码点击授权。重定向到百度，获取到授权码。



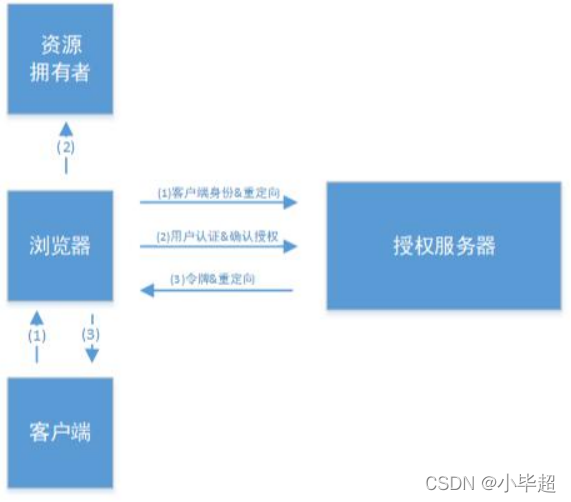
使用接口调用软件用POST方式获取Token：http://localhost:9098/oauth/token?client\_id=client\_1&client\_secret=secret&grant\_type=authorization\_code&code=F2XAWK&redirect\_uri=http://www.baidu.com



然后使用上面的access\_token访问即可：



# 简化模式



简化模式是相对于授权码模式来说，减少了通过授权码换取Token的步骤。

* 资源拥有者打开客户端，客户端要求资源拥有者给予授权，它将浏览器被重定向到授权服务器，重定向时会附加客户端的身份信息。
* 浏览器出现向授权服务器授权页面，之后将用户同意授权。
* 授权服务器将授权码将令牌（access\_token）以Hash的形式存放在重定向uri的fargment中发送给浏览器。

一般来说，简化模式用于没有服务器端的第三方单页面应用，因为没有服务器端就无法接收授权码。特点：

* 简单。流程简单
* 适用于纯前端应用
* 不安全。稍有不慎，Token可以被恶意脚本获取
* Token有效期短，浏览器关闭即失效

浏览器访问：

http://localhost:9098/oauth/authorize?client\_id=client\_1&response\_type=token&scope=all&redirect\_uri=http://www.baidu.com

将token返回至浏览器：



# 密码模式



这种模式十分简单，但是却意味着直接将用户敏感信息泄漏给了client，因此这就说明这种模式只能用于client是我们自己开发的情况下。因此密码模式一般用于我们自己开发的，第一方原生App或第一方单页面应用。

* 资源拥有者将用户名、密码发送给客户端。
* 客户端拿着资源拥有者的用户名、密码向授权服务器请求令牌（access\_token）。

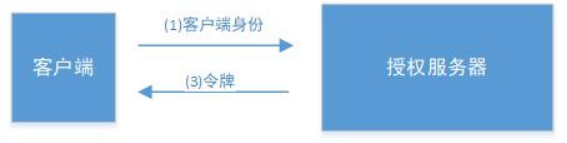
特点：

* 需要输入账号密码，极度不安全，需要高度信任第三方应用
* 适用于其他授权模式都无法采用的情况

发送POST请求：

<http://localhost:9098/oauth/token?client_id=client_1&client_secret=secret&grant_type=password&redirect_uri=http://www.baidu.com&username=admin&password=123456>

# 客户端模式



这种模式是最方便但最不安全的模式。因此这就要求我们对client完全的信任，而client本身也是安全的。因此这种模式一般用来提供给我们完全信任的服务器端服务。比如，合作方系统对接，拉取一组用户信息。

* 客户端向授权服务器发送自己的身份信息，并请求令牌（access\_token）
* 确认客户端身份无误后，将令牌（access\_token）发送给client

特点：

* 授权维度为应用维度，而不是用户维度。因此有可能多个用户共用一个Token的情况
* 适用于应用维度的共享资源

发送POST请求：

http://localhost:9098/oauth/token?client\_id=client\_1&client\_secret=secret&grant\_type=client\_credentials

# 认证服务使用JWT

当资源服务和授权服务不在一起时资源服务使用RemoteTokenServices 远程请求授权服务验证token，如果访问量较大将会影响系统的性能 。因此我们可以采用JWT生成令牌，用户认证通过会得到一个JWT令牌，JWT令牌中已经包括了用户相关的信息，客户端只需要携带JWT访问资源服务，资源服务根据事先约定的算法自行完成令牌校验，无需每次都请求认证服务完成授权。

## 整合JWT模式

JWT的签名模式，这里采用对称加密的方式，密钥设为bxc123，这个在资源服务器也是这么修改：

1. @Configuration
2. **public** **class** TokenConfig {
4. **private** String SIGNING\_KEY = "bxc123";
6. @Bean
8. **public** TokenStore tokenStore() {
9. **return** **new** JwtTokenStore(accessTokenConverter());
11. }
13. @Bean
14. **public** JwtAccessTokenConverter accessTokenConverter() {
15. JwtAccessTokenConverter converter = **new** JwtAccessTokenConverter();
16. converter.setSigningKey(SIGNING\_KEY); //对称秘钥，资源服务器使用该秘钥来验证
17. **return** converter;
18. }
19. }

修改AuthorizationServer配制类，主要AuthorizationServerTokenServices的配制，增加一个TokenEnhancerChain，将上面声明出的JwtAccessTokenConverter 配制进去。

1. @Autowired
2. **private** JwtAccessTokenConverter accessTokenConverter;
4. //令牌增强
5. TokenEnhancerChain tokenEnhancerChain = **new** TokenEnhancerChain();
6. List<TokenEnhancer> tokenEnhancers = **new** ArrayList<>();
7. tokenEnhancers.add(accessTokenConverter);
8. tokenEnhancerChain.setTokenEnhancers(tokenEnhancers);

将上面写的TokenConfig类，覆盖到资源服务中，并修改ResouceServerConfig配制类，将前面写的远程访问认证服务校验令牌的逻辑可以去除。

## 测试模式



## JWT内容增强

想要存放其他东西，只需使用DefaultOAuth2AccessToken的setAdditionalInformation方法，传递一个Map即可放置自定义的数据，其中可以获取用户的信息，也可以再查找其他自定义的信息放置，下面修改认证服务的AuthorizationServer配制类，添加TokenEnhancer tokenEnhancer()方法，在构建TokenEnhancerChain对象时，将上面配制的设置进去：：

1. /\*\*
2. \* JWT内容增强
3. \*/
4. @Bean
5. **public** TokenEnhancer tokenEnhancer() {
6. **return** (accessToken, authentication) -> {
7. Map<String, Object> additionalInfo = **new** HashMap<>();
8. User principal = (User)authentication.getUserAuthentication().getPrincipal();
9. String username = principal.getUsername();
10. additionalInfo.put("three","额外的内容！");
11. ((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(additionalInfo);
12. **return** accessToken;
13. };
14. }
16. //内容增强
17. tokenEnhancers.add(tokenEnhancer());

## JWT非对称密匙

修改TokenConfig，修改为使用非对称密钥：

1. @Configuration
2. **public** **class** TokenConfig {
4. @Bean
5. **public** TokenStore tokenStore() {
6. **return** **new** JwtTokenStore(accessTokenConverter());
7. }
9. @Bean
10. **public** JwtAccessTokenConverter accessTokenConverter() {
11. JwtAccessTokenConverter converter = **new** JwtAccessTokenConverter();
12. converter.setKeyPair(keyPair()); //非对称秘钥
13. **return** converter;
14. }
16. @Bean
17. **public** KeyPair keyPair() {
18. KeyStoreKeyFactory factory = **new** KeyStoreKeyFactory(**new** ClassPathResource("bxc.jks"), "123456".toCharArray());
19. **return** factory.getKeyPair("bxc", "123456".toCharArray());
20. }
22. }

编写一个公钥的接口，可以使用keytool和openssl生成公钥放在资源服务的resources下，通过本地读取的方式获取公钥，这里先提供一个公钥的接口（将该接口放行，不需要认证，修改WebSecurityConfig的configure(HttpSecurity http)，将/key/\*\*规则的直接permitAll()即可）：

1. @RestController
2. **public** **class** KeyController {
4. @Autowired
5. KeyPair keyPair;
7. //获取公钥
8. @GetMapping("/key/public-key")
9. **public** Map<String, Object> getPublicKey() {
10. RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
11. RSAKey key = **new** RSAKey.Builder(publicKey).build();
12. **return** **new** JWKSet(key).toJSONObject();
13. }
14. }

# 自定义授权模式

SpringOauth2.0虽说只有四种授权模式，但是他提供了非常高的扩展性，我们可以很方便的进行授权模式的扩展，而无需大量的改动，在SpringOauth2.0中扩展授权模式，只需继承AbstractTokenGranter类，并在构造方法中super父类时，指定自己的授权标识，当用户认证传入grant\_type为该标识，则会触发该生成器的getOAuth2Authentication方法，我们可以在这里获取客户端传入的参数，比如手机号和验证码，然后我们再和预先的验证码对比，如果OK，可以根据手机号获取用户的信息组成UsernamePasswordAuthenticationToken对象再交给Oauth2即可。

SmsCodeTokenGranter这里只是演示功能，直接将用户和验证码写死了：

1. **public** **class** SmsCodeTokenGranter **extends** AbstractTokenGranter {
3. **private** **static** **final** String SMS\_GRANT\_TYPE = "sms\_code";
5. **private** UserService userService;
7. **public** SmsCodeTokenGranter(AuthorizationServerTokenServices tokenServices,
8. ClientDetailsService clientDetailsService,
9. OAuth2RequestFactory requestFactory,
10. UserService userService) {
11. **super**(tokenServices, clientDetailsService, requestFactory, SMS\_GRANT\_TYPE);
12. **this**.userService = userService;
13. }

16. @Override
17. **protected** OAuth2Authentication getOAuth2Authentication(ClientDetails client,
18. TokenRequest tokenRequest) {
20. Map<String, String> parameters = **new** LinkedHashMap<String, String>(tokenRequest.getRequestParameters());
22. System.out.println(parameters.toString());
23. // 客户端提交的手机号码
24. String phoneNumber = parameters.get("phone");
25. **if** (StringUtils.isBlank(phoneNumber)) {
26. **throw** **new** AccessDeniedException("get phone is null ！");
27. }
29. // 客户端提交的验证码
30. String smsCode = parameters.get("code");
31. **if** (!smsCode.equals("9876")) {
32. **throw** **new** AccessDeniedException("code err!");
33. }
35. UserDetails user = userService.loadUserByUsername("admin");
37. AbstractAuthenticationToken userAuth = **new** UsernamePasswordAuthenticationToken(user, **null**, user.getAuthorities());
39. userAuth.setDetails(parameters);
40. OAuth2Request oAuth2Request = getRequestFactory().createOAuth2Request(client, tokenRequest);
41. **return** **new** OAuth2Authentication(oAuth2Request, userAuth);
42. }
43. }

修改AuthorizationServer将上面的生成器配制进AuthorizationServerEndpointsConfigurer中。