

主讲老师：Fox

学习本课程的前提：

1. 了解[OAuth2协议及其密码授权模式](#)，熟悉[Spring Security OAuth2和JWT的使用](#)，熟悉Spring Cloud Gateway网关使用。
2. 对OAuth2协议不清楚的同学可以先学习微服务专题：Spring Security OAuth2两节课，再来学习本节课
3. 电商项目整合授权中心接入网关后服务的启动顺序：
tulingmall-member——》tulingmall-authcenter——》tulingmall-gateway

扩展知识：

Spring官方已经不在维护Spring Security OAuth，官方单独启动一个授权服务器项目Spring Authorization Server [Spring Authorization Server实战](#)

- 1 文档：2 电商项目微服务网关整合OAuth2.0授权...
- 2 链接：<http://note.youdao.com/noteshare?id=a5cbc586f2d43924ece4ec1121475d97&sub=ECEC0DC2F92347CD9A34DD454385D7CD>

1. 微服务网关整合 OAuth2.0 思路分析

2. 搭建微服务授权中心

2.1 引入依赖

2.2 添加yaml配置

2.3 配置授权服务器

2.5 测试模拟用户登录

2.6 配置资源服务器

2.7 Spring Security OAuth2整合JWT

2.8 优化：实现JWT非对称加密（公钥私钥）

3. 接入网关服务

1. 微服务网关整合 OAuth2.0 思路分析

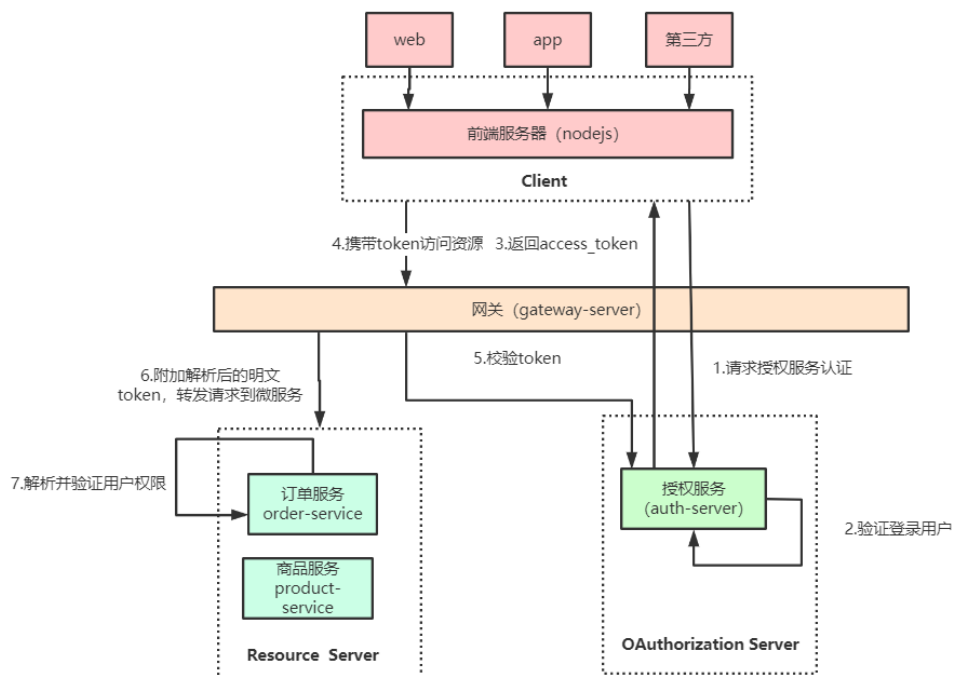
网关整合 OAuth2.0 有两种思路，一种是授权服务器生成令牌，所有请求统一在网关层验证，判断权限等操作；另一种是由各资源服务处理，网关只做请求转发。比较常用的是第一种，把API网关作为OAuth2.0的资源服务器角色，实现接入客户端权限拦截、令牌解析并转发当前登录用户信息给微服务，这样下游微服务就不需要关心令牌格式解析以及OAuth2.0相关机制了。

网关在认证授权体系里主要负责两件事：

- (1) 作为OAuth2.0的资源服务器角色，实现接入方访问权限拦截。
- (2) 令牌解析并转发当前登录用户信息（明文token）给微服务

微服务拿到明文token(明文token中包含登录用户的身份和权限信息)后也需要做两件事：

- (1) 用户授权拦截（看当前用户是否有权访问该资源）
- (2) 将用户信息存储进当前线程上下文（有利于后续业务逻辑随时获取当前用户信息）



2. 搭建微服务授权中心

授权中心的认证依赖：

- 第三方客户端的信息
- 微服务的信息
- 登录用户的信息

创建微服务tulingmall-authcenter

2.1 引入依赖

```
1 <dependency>
2   <groupId>com.alibaba</groupId>
3   <artifactId>druid-spring-boot-starter</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>mysql</groupId>
8   <artifactId>mysql-connector-java</artifactId>
9 </dependency>
10
11 <dependency>
12   <groupId>org.springframework.boot</groupId>
13   <artifactId>spring-boot-starter-web</artifactId>
14 </dependency>
15
```

```

16 <dependency>
17   <groupId>org.projectlombok</groupId>
18   <artifactId>lombok</artifactId>
19 </dependency>
20
21 <dependency>
22   <groupId>org.springframework.boot</groupId>
23   <artifactId>spring-boot-starter</artifactId>
24 </dependency>

```

2.2 添加yml配置

```

1 server:
2   port: 9999
3 spring:
4   application:
5     name: tulingmall-auth
6     #配置nacos注册中心地址
7   cloud:
8     nacos:
9     discovery:
10      server-addr: 192.168.65.103:8848 #注册中心地址
11      namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618 #环境隔离
12      datasource:
13        url: jdbc:mysql://tuling.com:3306/tlmall_oauth?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
14        username: root
15        password: root
16      druid:
17        initial-size: 5 #连接池初始化大小
18        min-idle: 10 #最小空闲连接数
19        max-active: 20 #最大连接数
20      web-stat-filter:
21        exclusions: "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*" #不统计这些请求数据
22        stat-view-servlet: #访问监控网页的登录用户名和密码
23        login-username: druid
24        login-password: druid
25

```

2.3 配置授权服务器

基于DB模式配置授权服务器存储第三方客户端的信息

```

1 @Configuration
2 @EnableAuthorizationServer
3 public class TulingAuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {
4
5   @Autowired
6   private DataSource dataSource;
7
8   @Override
9   public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

```

```

10 // 配置授权服务器存储第三方客户端的信息 基于DB存储 oauth_client_details
11 clients.withClientDetails(clientDetails());
12 }
13
14 @Bean
15 public ClientDetailsService clientDetails(){
16     return new JdbcClientDetailsService(dataSource);
17 }
18
19 }

```

在oauth_client_details中添加第三方客户端信息 (client_id client_secret scope等等)

```

1 CREATE TABLE `oauth_client_details` (
2   `client_id` varchar(128) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
3   `resource_ids` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
4   `client_secret` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
5   `scope` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
6   `authorized_grant_types` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
7   `web_server_redirect_uri` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
8   `authorities` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
9   `access_token_validity` int(11) NULL DEFAULT NULL,
10  `refresh_token_validity` int(11) NULL DEFAULT NULL,
11  `additional_information` varchar(4096) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
12  `autoapprove` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
13  PRIMARY KEY (`client_id`) USING BTREE
14 ) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;

```

client_id	resource_ids	client_secret	scope	authorized_grant_types	web_server_redirect_uri
client	(Null)	\$2a\$10\$CE1GKj9eBz\$NNMCZV2hpo.QBOz93ojy9mTd9YQaOy8H4JAYYKVM6	all	authorization_code,password,refresh_token	http://www.baidu.com
tulingmall-gateway	(Null)	\$2a\$10\$o.8XgLmnZi6RBRTbkj2z/u3ly6laiRI3eHIO\$O.IJfMn9pnKSM7i	read	password,refresh_token	(Null)
tulingmall-member	(Null)	\$2a\$10\$APF9tE9z9Z74rcFZUjvTeGpmH2XP1BdVTWYt6CLzT5UVDNt2uJW	read,write	password,refresh_token	(Null)

基于内存模式配置授权服务器存储第三方客户端的信息

```

1 //TulingAuthorizationServerConfig.java
2 @Override
3 public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
4     // 配置授权服务器存储第三方客户端的信息 基于DB存储 oauth_client_details
5     // clients.withClientDetails(clientDetails());
6
7     /**
8      *授权码模式
9      *http://localhost:9999/oauth/authorize?
10     response_type=code&client_id=client&redirect_uri=http://www.baidu.com&scope=all
11
12     * password模式
13     * http://localhost:8080/oauth/token?username=fox&password=123456&grant_type=password&client_id=client&client_secret=123123&scope=all
14
15     */
16     clients.inMemory()
17     //配置client_id
18     .withClient("client")

```

```

18 //配置client-secret
19 .secret(passwordEncoder.encode("123123"))
20 //配置访问token的有效期
21 .accessTokenValiditySeconds(3600)
22 //配置刷新token的有效期
23 .refreshTokenValiditySeconds(86400)
24 //配置redirect_uri, 用于授权成功后跳转
25 .redirectUri("http://www.baidu.com")
26 //配置申请的权限范围
27 .scopes("all")
28 /**
29  * 配置grant_type, 表示授权类型
30  * authorization_code: 授权码
31  * password: 密码
32  * refresh_token: 更新令牌
33  */
34 .authorizedGrantTypes("authorization_code", "password", "refresh_token");
35
36 }

```

2.4 配置SpringSecurity

```

1 @Configuration
2 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
3
4     @Bean
5     public PasswordEncoder passwordEncoder() {
6         return new BCryptPasswordEncoder();
7     }
8
9     @Autowired
10    private TulingUserDetailsService tulingUserDetailsService;
11
12    @Override
13    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
14        // 实现UserDetailsService获取用户信息
15        auth.userDetailsService(tulingUserDetailsService);
16    }
17
18
19    @Bean
20    @Override
21    public AuthenticationManager authenticationManagerBean() throws Exception {
22        // oauth2 密码模式需要拿到这个bean
23        return super.authenticationManagerBean();
24    }
25
26    @Override
27    protected void configure(HttpSecurity http) throws Exception {
28        http.formLogin().permitAll()
29        .and().authorizeRequests()

```

```

30 .antMatchers("/oauth/**").permitAll()
31 .anyRequest()
32 .authenticated()
33 .and().logout().permitAll()
34 .and().csrf().disable();
35 }
36 }

```

获取会员信息，此处通过feign从tulingmall-member获取会员信息，需要配置feign，核心代码：

```

1  @Slf4j
2  @Component
3  public class TulingUserDetailsService implements UserDetailsService {
4
5      @Override
6      public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
7          // 加载用户信息
8          if(StringUtils.isEmpty(username)) {
9              log.warn("用户登陆用户名为空:{}, username);
10             throw new UsernameNotFoundException("用户名不能为空");
11         }
12
13         UmsMember umsMember = getByUsername(username);
14
15         if(null == umsMember) {
16             log.warn("根据用户名没有查询到对应的用户信息:{}, username);
17         }
18
19         log.info("根据用户名:{}获取用户登陆信息:{}, username, umsMember);
20
21         // 会员信息的封装 implements UserDetails
22         MemberDetails memberDetails = new MemberDetails(umsMember);
23
24         return memberDetails;
25     }
26
27     @Autowired
28     private UmsMemberFeignService umsMemberFeignService;
29
30     public UmsMember getByUsername(String username) {
31         // feign获取会员信息
32         CommonResult<UmsMember> umsMemberCommonResult = umsMemberFeignService.loadUserByUsername(username);
33
34         return umsMemberCommonResult.getData();
35     }
36 }
37
38 @FeignClient(value = "tulingmall-member", path="/member/center")
39 public interface UmsMemberFeignService {
40
41     @RequestMapping("/loadUmsMember")
42     CommonResult<UmsMember> loadUserByUsername(@RequestParam("username") String username);

```

```
43 }
44
45 public class MemberDetails implements UserDetails {
46     private UmsMember umsMember;
47
48     public MemberDetails(UmsMember umsMember) {
49         this.umsMember = umsMember;
50     }
51
52     @Override
53     public Collection<? extends GrantedAuthority> getAuthorities() {
54         //返回当前用户的权限
55         return Arrays.asList(new SimpleGrantedAuthority("TEST"));
56     }
57
58     @Override
59     public String getPassword() {
60         return umsMember.getPassword();
61     }
62
63     @Override
64     public String getUsername() {
65         return umsMember.getUsername();
66     }
67
68     @Override
69     public boolean isAccountNonExpired() {
70         return true;
71     }
72
73     @Override
74     public boolean isAccountNonLocked() {
75         return true;
76     }
77
78     @Override
79     public boolean isCredentialsNonExpired() {
80         return true;
81     }
82
83     @Override
84     public boolean isEnabled() {
85         return umsMember.getStatus()==1;
86     }
87
88     public UmsMember getUmsMember() {
89         return umsMember;
90     }
91 }
92
```

修改授权服务配置，支持密码模式

```
1 //TulingAuthorizationServerConfig.java
2 @Autowired
3 private TulingUserDetailsService tulingUserDetailsService;
4
5 @Autowired
6 private AuthenticationManager authenticationManagerBean;
7
8 @Override
9 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
10     //使用密码模式需要配置
11     endpoints.authenticationManager(authenticationManagerBean)
12     .reuseRefreshTokens(false) //refresh_token是否重复使用
13     .userDetailsService(tulingUserDetailsService) //刷新令牌授权包含对用户信息的检查
14     .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST); //支持GET, POST请求
15 }
16
17 /**
18  * 授权服务器安全配置
19  * @param security
20  * @throws Exception
21  */
22 @Override
23 public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {
24     //第三方客户端校验token需要带入 clientId 和clientSecret来校验
25     security.checkTokenAccess("isAuthenticated()")
26     .tokenKeyAccess("isAuthenticated()"); //来获取我们的tokenKey需要带入clientId, clientSecret
27
28     //允许表单认证
29     security.allowFormAuthenticationForClients();
30 }
```

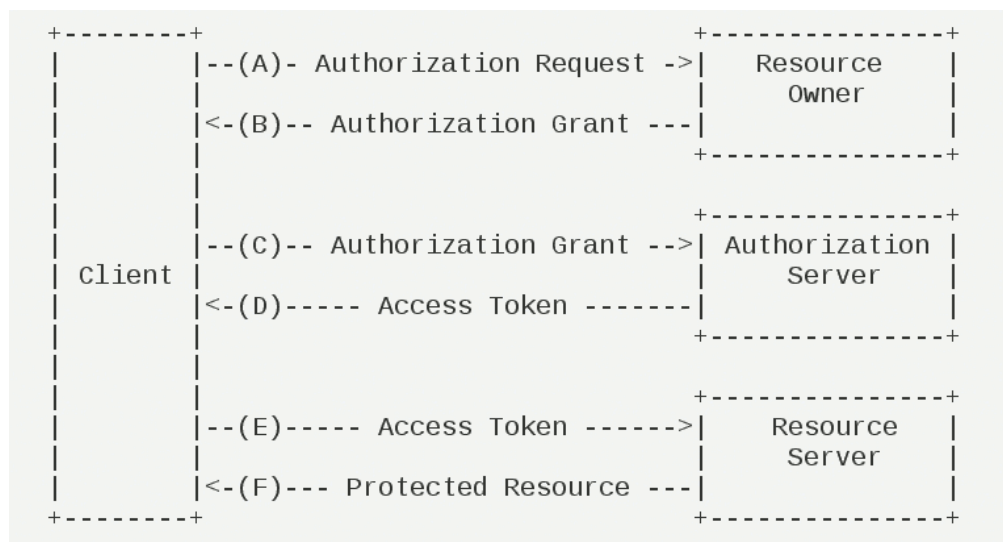
2.5 测试模拟用户登录

授权码模式

授权码（authorization code）方式，指的是第三方应用先申请一个授权码，然后再用该码获取令牌。

这种方式是最常用的流程，安全性也最高，它适用于那些有后端的 Web 应用。授权码通过前端传送，令牌则是储存在后端，而且所有与资源服务器的通信都在后端完成。这样的前后端分离，可以避免令牌泄漏。

适用场景：目前市面上主流的第三方验证都是采用这种模式



它的步骤如下：

- (A) 用户访问客户端，后者将前者导向授权服务器。
- (B) 用户选择是否给予客户端授权。
- (C) 假设用户给予授权，授权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"，向授权服务器申请令牌。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) 授权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）。

[http://localhost:9999/oauth/authorize?](http://localhost:9999/oauth/authorize?response_type=code&client_id=client&redirect_uri=http://www.baidu.com&scope=all)

[response_type=code&client_id=client&redirect_uri=http://www.baidu.com&scope=all](http://localhost:9999/oauth/authorize?response_type=code&client_id=client&redirect_uri=http://www.baidu.com&scope=all)

获取到code

  baidu.com/?code=O3KwV

GET http://localhost:9999/oauth/token?grant_type=authorization_code&client_id=client&client_secret=123123&scope=all&code=O3KvvV&redi ...

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> grant_type	authorization_code	
<input checked="" type="checkbox"/> client_id	client	
<input checked="" type="checkbox"/> client_secret	123123	
<input checked="" type="checkbox"/> scope	all	
<input checked="" type="checkbox"/> code	O3KvvV	
<input checked="" type="checkbox"/> redirect_uri	http://www.baidu.com	

Body Cookies (1) Headers (9) Test Results Status: 200 OK Time: 94 ms Size: 501 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "access_token": "634d668e-7f28-4505-b2f9-34a3e5dd975b",
3   "token_type": "bearer",
4   "refresh_token": "0487b96a-5a27-47ec-8ac5-bd31b632d679",
5   "expires_in": 3600,
6   "scope": "all"
7 }
```

获取到access_token

密码模式

如果你高度信任某个应用，RFC 6749 也允许用户把用户名和密码，直接告诉该应用。该应用就使用你的密码，申请令牌，这种方式称为"密码式"（password）。

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。而授权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。

适用场景：自家公司搭建的授权服务器

测试获取token

<http://localhost:9999/oauth/token?>

[username=test&password=test&grant_type=password&client_id=client&client_secret=123123&scope=all](http://localhost:9999/oauth/token?username=test&password=test&grant_type=password&client_id=client&client_secret=123123&scope=all)

GET http://localhost:9999/oauth/token?username=test&password=test&grant_type=password&client_id=client&client_secret=123123&scope=...

Params Authorization Headers (8) Body Pre-request Script Tests Settings

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> username	test	
<input checked="" type="checkbox"/> password	test	
<input checked="" type="checkbox"/> grant_type	password	
<input checked="" type="checkbox"/> client_id	client	
<input checked="" type="checkbox"/> client_secret	123123	
<input checked="" type="checkbox"/> scope	all	
Key	Value	Description

Body Cookies (1) Headers (8) Test Results Status: 200 OK Time: 1723 ms Size: 426 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "access_token": "853967fb-6f9e-4813-b35e-cf4b0225e4cb",
3   "token_type": "bearer",
4   "refresh_token": "c419438c-5761-44a9-8d09-0ac9df4c1ab3",
5   "expires_in": 3599,
6   "scope": "all"
7 }
```

测试校验token接口

```

@Override
public void configure(AuthorizationServerSecurityConfigurer security) throws
Exception {
    // 第三方客户端校验token需要带入 client_id 和 clientSecret 来校验
    security.checkTokenAccess("isAuthenticated()")
        .tokenKeyAccess("isAuthenticated()"); // 来获取我们的 tokenKey 需要带入
    client_id, clientSecret
}

```

因为授权服务器的security配置需要携带clientId和clientSecret，可以采用basic Auth的方式发请求

GET http://localhost:9999/oauth/check_token?token=8e9858b4-30ab-4123-b86f-5608df287fe9 ...

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Type Basic Auth

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Username client_id client

Password client_secret 123123

☒ Show Password

Heads up! These parameters hold sensitive data. To keep this data secure while working in a c we recommend using variables. [Learn more about variables](#)

注意：传参是token

GET http://localhost:9999/oauth/check_token?token=8e9858b4-30ab-4123-b86f-5608df287fe9

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	token	8e9858b4-30ab-4123-b86f-5608df287fe9	
	Key	Value	Description

Body Cookies (1) Headers (10) Test Results

Status: 200 OK Time: 109 ms

Pretty Raw Preview Visualize JSON

```

1 {
2   "active": true,
3   "exp": 1618296434,
4   "user_name": "test",
5   "authorities": [
6     "TEST"
7   ],
8   "client_id": "client",
9   "scope": [
10    "all"
11  ]
12 }

```

2.6 配置资源服务器

```

1 @Configuration
2 @EnableResourceServer
3 public class TulingResourceServerConfig extends ResourceServerConfigurerAdapter {
4
5     @Override
6     public void configure(HttpSecurity http) throws Exception {
7         http.authorizeRequests()
8             .anyRequest().authenticated();
9
10    }
11 }
12
13 @RestController
14 @RequestMapping("/user")

```

```

15 public class UserController {
16
17
18     @RequestMapping("/getCurrentUser")
19     public Object getCurrentUser(Authentication authentication) {
20         return authentication.getPrincipal();
21     }
22 }

```

测试携带token访问资源

GET http://localhost:9999/user/getCurrentUser?access_token=853967fb-6f9e-4813-b35e-cf4b0225e4cb...

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	access_token	853967fb-6f9e-4813-b35e-cf4b0225e4cb	
	Key	Value	Description

Body Cookies (1) Headers (9) Test Results Status: 200 OK Time: 23 ms S

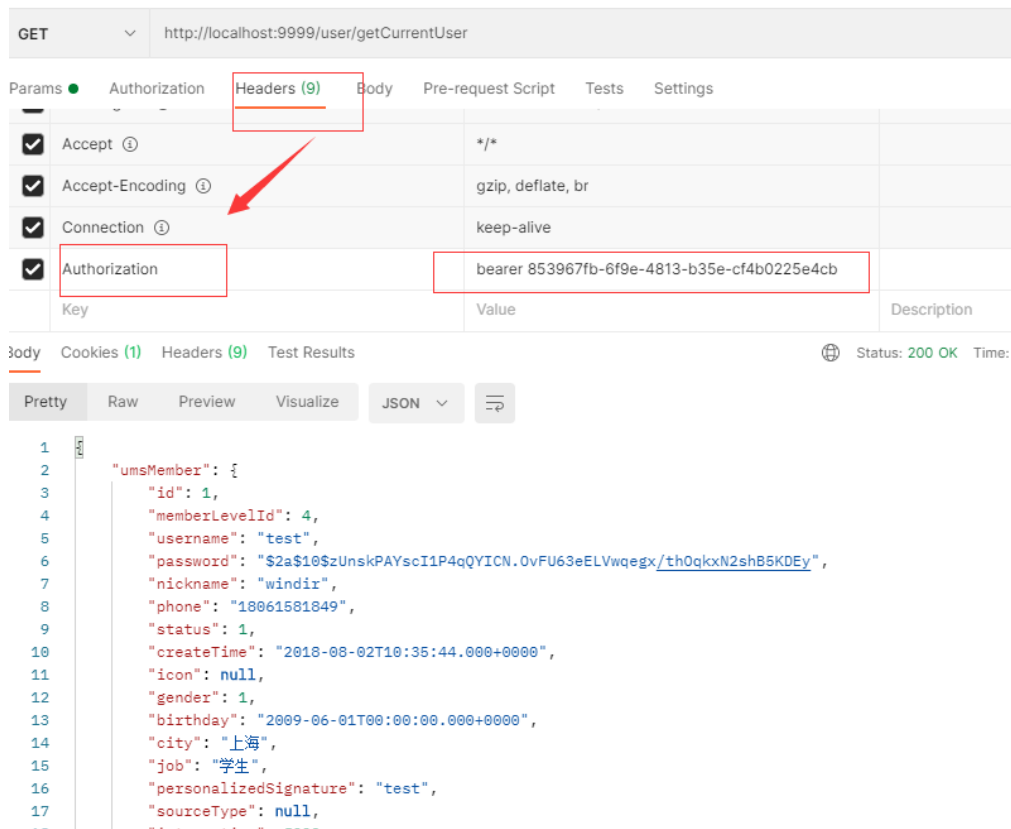
Pretty Raw Preview Visualize JSON

```

1  {
2    "umsMember": {
3      "id": 1,
4      "memberLevelId": 4,
5      "username": "test",
6      "password": "$2a$10$zUnskPAYscI1P4qQYICN.OvFU63eELVwqegx/th0qkxN2shB5KDEy",
7      "nickname": "windir",
8      "phone": "18061581849",
9      "status": 1,
10     "createTime": "2018-08-02T10:35:44.000+0000",
11     "icon": null,
12     "gender": 1,
13     "birthday": "2009-06-01T00:00:00.000+0000",
14     "city": "上海",
15     "job": "学生",
16     "personalizedSignature": "test",
17     "sourceType": null,
18     "integration": 5000,
19     "growth": null,

```

或者请求头配置Authorization



OAuth 2.0是当前业界标准的授权协议，它的核心是若干个针对不同场景的令牌颁发和管理流程；而JWT是一种轻量级、自包含的令牌，可用于在微服务间安全地传递用户信息。

2.7 Spring Security Oauth2整合JWT

JSON Web Token (JWT) 是一个开放的行业标准 (RFC 7519)，它定义了一种简介的、自包含的协议格式，用于在通信双方传递json对象，传递的信息经过数字签名可以被验证和信任。JWT可以使用HMAC算法或使用RSA的公钥/私钥对来签名，防止被篡改。

官网：<https://jwt.io/>

JWT令牌的优点：

- jwt基于json，非常方便解析。
- 可以在令牌中自定义丰富的内容，易扩展。
- 通过非对称加密算法及数字签名技术，JWT防止篡改，安全性高。
- 资源服务使用JWT可不依赖认证服务即可完成授权。

缺点：

JWT令牌较长，占存储空间比较大。

JWT：指的是 JSON Web Token，由 header.payload.signture 组成。不存在签名的JWT是不安全的，存在签名的JWT是不可篡改的。

JWS：指的是签过名的JWT，即拥有签名的JWT。

JWK：既然涉及到签名，就涉及到签名算法，对称加密还是非对称加密，那么就需要加密的 密钥或者公私钥对。此处我们将JWT的密钥或者公私钥对统一称为 JSON WEB KEY，即 JWK。

JWT组成

一个JWT实际上就是一个字符串，它由三部分组成，头部（header）、载荷（payload）与签名（signature）。

Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  fox
) ☐ secret base64 encoded
```

头部（header）

头部用于描述关于该JWT的最基本的信息：类型（即JWT）以及签名所用的算法（如HMACSHA256或RSA）等。

这也可以被表示成一个JSON对象：

```
1 {
2   "alg": "HS256",
3   "typ": "JWT"
4 }
```

然后将头部进行base64加密（该加密是可以对称解密的),构成了第一部分：

```
1 eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

载荷（payload）

第二部分是载荷，就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分：

- 标准中注册的声明（建议但不强制使用）

iss: jwt签发者

sub: jwt所面向的用户

aud: 接收jwt的一方

exp: jwt的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该jwt都是不可用的。

iat: jwt的签发时间

jti: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

- 公共的声明

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不

建议添加敏感信息，因为该部分在客户端可解密。

- 私有的声明

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个payload：

```
1 {
2   "sub": "1234567890",
3   "name": "John Doe",
4   "iat": 1516239022
5 }
```

然后将其进行base64加密，得到Jwt的第二部分：

```
1 eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
```

签名 (signature)

jwt的第三部分是一个签证信息，这个签证信息由三部分组成：

- header (base64后的)
- payload (base64后的)
- secret(盐，一定要保密)

这个部分需要base64加密后的header和base64加密后的payload使用. 连接组成的字符串，然后通过header中声明的加密方式进行加盐secret组合加密，然后就构成了jwt的第三部分：

```
1 var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);
2 var signature = HMACSHA256(encodedString, 'fox'); // khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME
```

将这三部分用. 连接成一个完整的字符串,构成了最终的jwt:

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME
```

注意：secret是保存在服务器端的，jwt的签生成成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

JWT应用场景

- 一次性验证

比如用户注册后需要发一封邮件让其激活账户，通常邮件中需要有一个链接，这个链接需要具备以下的特性:能够标识用户，该链接具有时效性（通常只允许几小时之内激活），不能被篡改以激活其他可能的账户...这种场景就和jwt的特性非常贴近.jwt的 payload 中固定的参数: iss签发者和exp过期时间正是为其做准备的。

- restful api的无状态认证

使用jwt来做restful api的身份认证也是值得推崇的一种使用方案。客户端和服务端共享secret;过期时间由服务端校验，客户端定时刷新;签名信息不可被修改。

- 使用jwt做单点登录+会话管理(不推荐)

jwt是无状态的，在处理注销，续约问题上会变得非常复杂

引入依赖

```
1 <!--spring security对jwt的支持 spring cloud oauth2已经依赖，可以不配置-->
```

```

2 <dependency>
3 <groupId>org.springframework.security</groupId>
4 <artifactId>spring-security-jwt</artifactId>
5 <version>1.0.9.RELEASE</version>
6 </dependency>

```

添加JWT配置

```

1 @Configuration
2 public class JwtTokenStoreConfig {
3
4     @Bean
5     public TokenStore jwtTokenStore(){
6         return new JwtTokenStore(jwtAccessTokenConverter());
7     }
8
9     @Bean
10    public JwtAccessTokenConverter jwtAccessTokenConverter(){
11        JwtAccessTokenConverter accessTokenConverter = new
12        JwtAccessTokenConverter();
13        //配置JWT使用的秘钥
14        accessTokenConverter.setSigningKey("123123");
15        return accessTokenConverter;
16    }
17 }

```

在授权服务器配置中指定令牌的存储策略为JWT

```

1 //TulingAuthorizationServerConfig.java
2
3 @Autowired
4 @Qualifier("jwtTokenStore")
5 private TokenStore tokenStore;
6
7 @Autowired
8 private JwtAccessTokenConverter jwtAccessTokenConverter;
9
10 @Autowired
11 private TulingUserDetailsService tulingUserDetailsService;
12
13 @Autowired
14 private AuthenticationManager authenticationManagerBean;
15
16 @Override
17 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
18     //使用密码模式需要配置
19     endpoints.authenticationManager(authenticationManagerBean)
20     .tokenStore(tokenStore) //指定token存储策略是jwt
21     .accessTokenConverter(jwtAccessTokenConverter)
22     .reuseRefreshTokens(false) //refresh_token是否重复使用
23     .userDetailsService(tulingUserDetailsService) //刷新令牌授权包含对用户信息的检查
24     .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST); //支持GET, POST请求

```


密码模式测试:

http://localhost:9999/oauth/token?

username=test&password=test&grant_type=password&client_id=client&client_secret=123123&scope=all

GET http://localhost:9999/oauth/token?username=test&password=test&grant_type=password&client_id=client&client_secret=123123&scope=...

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

KEY	VALUE	DESCRIPTION
username	test	
password	test	
grant_type	password	
client_id	client	
client_secret	123123	
scope	all	

Body Cookies (1) Headers (9) Test Results Status: 200 OK Time: 1098 ms Size: 1.05 KB Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MTgyMzc5NDU5InVzZXJfcmFtZSI6InRlc3QlLCJhdXRob3JpdG1lc2VzIHRmZTV1LThmZjgtNGYzZi1iMTE1LTk4ODBlYmNkNGY3YyIsImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3B1IjpjbImFsbCJdfQ.EN41WW
3   "token_type": "bearer",
4   "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MTgyMzc5NDU5InVzZXJfcmFtZSI6InRlc3QlLCJhdXRob3JpdG1lc2VzIHRmZTV1LThmZjgtNGYzZi1iMTE1LTk4ODBlYmNkNGY3YyIsImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3B1IjpjbImFsbCJdfQ.EN41WW
5   "expires_in": 3599,
6   "scope": "all",
7   "jti": "dca4fe5e-8ff8-4f3f-b115-9880ebcd4f7c"
8 }
```

将access_token复制到<https://jwt.io/>的Encoded中打开,可以看到会员认证信息

Encoded PASTE A TOKEN HERE

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "exp": 1618237941,
  "user_name": "test",
  "authorities": [
    "TEST"
  ],
  "jti": "dca4fe5e-8ff8-4f3f-b115-9880ebcd4f7c",
  "client_id": "client",
  "scope": [
    "all"
  ]
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) [ ] secret base64 encoded
```

测试校验token

GET

http://localhost:9999/oauth/check_token?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJlE2MTgyOTgwOTks

Params●Authorization●Headers(9)BodyPre-request ScriptTestsSettings

Type

Basic Auth

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

① Heads up! These parameters hold sensitive data. To keep this data secure while working, we recommend using variables. [Learn more about variables](#)

Username

client

Password

123123

☒ Show Password

GET

http://localhost:9999/oauth/check_token?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJlE2MTgyOTgwOTks

Params●Authorization●Headers(9)BodyPre-request ScriptTestsSettings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJlE2MTgyOTgwOTks	
	Key	Value	Description

body

Cookies(1)Headers(9)Test Results

Status: 200 OKTime: 1

PrettyRawPreviewVisualizeJSON

```
1  {
2    "user_name": "test",
3    "scope": [
4      "all"
5    ],
6    "active": true,
7    "exp": 1618298099,
8    "authorities": [
9      "TEST"
10   ],
11   "jti": "31f632b2-3130-4a92-8f2c-3360e4e7f42c",
12   "client_id": "client"
13 }
```

测试获取token_key

GET http://localhost:9999/oauth/token_key

Params Authorization Headers (11) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "umsMember": {
3     "id": 1,
4     "memberLevelId": 4,
5     "username": "test",
6     "password": "$2a$10$zUnskPAYscI1P4qQYICN.0vFU63eELVwqegx/th0qkxN2shB5KDEy",
7     "nickname": "windir",
8     "phone": "18061581849",
```

Body Cookies (1) Headers (9) Test Results Status

Pretty Raw Preview Visualize JSON

```
1 {
2   "alg": "HMACSHA256",
3   "value": "123123"
4 }
```

测试刷新token

GET http://localhost:9999/oauth/token?grant_type=refresh_token&client_id=client&client_secret=123123&refresh_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm9udC5kaW50IjoiImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3BlIjpbImFsbCJdfQ.CgXXG58qmlrPHXG-j3koGegsZIKvgQ54JTR3

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	grant_type	refresh_token	
<input checked="" type="checkbox"/>	client_id	client	
<input checked="" type="checkbox"/>	client_secret	123123	
<input checked="" type="checkbox"/>	refresh_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm9udC5kaW50IjoiImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3BlIjpbImFsbCJdfQ.CgXXG58qmlrPHXG-j3koGegsZIKvgQ54JTR3	
	Key	Value	Description

Body Cookies (1) Headers (8) Test Results Status: 200 OK Time: 92 ms Size: 1001

Pretty Raw Preview Visualize JSON

```
1 {
2   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm9udC5kaW50IjoiImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3BlIjpbImFsbCJdfQ.CgXXG58qmlrPHXG-j3koGegsZIKvgQ54JTR3",
3   "token_type": "bearer",
4   "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm9udC5kaW50IjoiImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3BlIjpbImFsbCJdfQ.CgXXG58qmlrPHXG-j3koGegsZIKvgQ54JTR3",
5   "expires_in": 3599,
6   "scope": "all",
7   "jti": "9fc3dce0-0d76-4092-bf63-1f3033a21d52"
8 }
```

2.8 优化：实现JWT非对称加密（公钥私钥）

第一步：生成jks 证书文件

我们使用jdk自动的工具生成

- 命令格式
- keytool
- genkeypair 生成密钥对
- alias jwt(别名)
- keypass 123456(别名密码)
- keyalg RSA(生证书的算法名称，RSA是一种非对称加密算法)
- keysize 1024(密钥长度,证书大小)
- validity 365(证书有效期，天单位)
- keystore D:/jwt/jwt.jks(指定生成证书的位置和证书名称)

-storepass 123456(获取keystore信息的密码)

-storetype (指定密钥仓库类型)

使用 "keytool -help" 获取所有可用命令

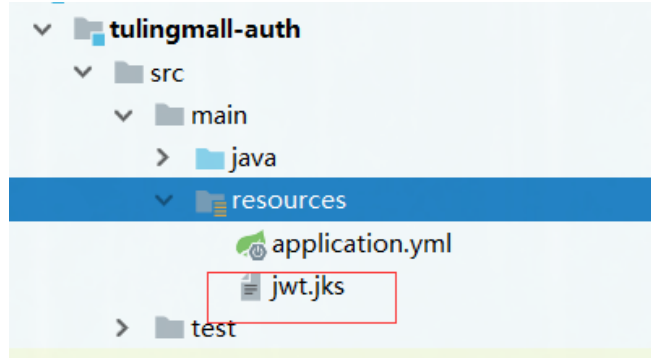
```
1 keytool -genkeypair -alias jwt -keyalg RSA -keysize 2048 -keystore D:/jwt/jwt.jks
```

```
C:\Users\chaos>keytool -genkeypair -alias jwt -keyalg RSA -keysize 2048 -keystore D:/jwt/jwt.jks
输入密钥库口令:
再次输入新口令: 输入密钥: 123123
您的名字与姓氏是什么?
[Unknown]: fox
您的组织单位名称是什么?
[Unknown]: tuling
您的组织名称是什么?
[Unknown]: tuling
您所在的城市或区域名称是什么?
[Unknown]: changsha
您所在的省/市/自治区名称是什么?
[Unknown]: hunan
该单位的双字母国家/地区代码是什么?
[Unknown]: china
CN=fox, OU=tuling, O=tuling, L=changsha, ST=hunan, C=china是否正确?
[否]: y

输入 <jwt> 的密钥口令
(如果和密钥库口令相同, 按回车):

Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore D:/jwt/jwt.jks -destkeystore D:/jwt/
jwt.jks -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。
```

将生成的jwt.jks文件拷贝到授权服务器的resource目录下



查看公钥信息

```
1 keytool -list -rfc --keystore jwt.jks | openssl x509 -inform pem -pubkey
```

```
$ keytool -list -rfc --keystore jwt.jks | openssl x509 -inform pem -pubkey
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAI6f8aDMN20en1BfCORax
+IN5z8MSq1FzwX+oLLjkvqr1wktXELMnON6+Tk/4WgX51tm4Xd7VHNSwQHaTmICk
inbJHHbEKk/OmbjwqDdIuuVcV7rQnMsgoXDFdeMXmifHuJfKdF+R1+3ey10ISi4B
bD4KBi+ZCMzVGu3mEQC0cnso+iKytgdC7qYhOwNehR9r/cYfvXHBf0bh1DGtj0bB
ArpKtbvKjOvaEtyfW/CiJe9wT+RT+mDBHYr3jotXC1DvdLKLWSW7pJBGLuifiCaY
znhqmbKGkyIEQVLxeU0wCPzNiYBaU0PG2cswNZ6D7G4S3g73mVkJDZHkzVd649
NwIDAQAB
-----END PUBLIC KEY-----
-----BEGIN CERTIFICATE-----
MIIDZTCCAk2gAwIBAgIEd/Py2DANBgkqhkiG9w0BAQsFADBJMQ4wDAYDVQQGEwVj
aGluYTEOMAwGA1UECBFaHVVuYW4xETAPBgNVBACTCGNoYW55nc2hhMQ8wDQYDVQQK
-----END CERTIFICATE-----
```

第二步：授权服务中增加jwt的属性配置类

```
1 @Data
2 @ConfigurationProperties(prefix = "tuling.jwt")
3 public class JwtCAProperties {
4
5     /**
6      * 证书名称
```

```

7  */
8  private String keyPairName;
9
10
11  /**
12   * 证书别名
13   */
14  private String keyPairAlias;
15
16  /**
17   * 证书私钥
18   */
19  private String keyPairSecret;
20
21  /**
22   * 证书存储密钥
23   */
24  private String keyPairStoreSecret;
25
26  }
27
28  @Configuration
29  // 指定属性配置类
30  @EnableConfigurationProperties(value = JwtCAProperties.class)
31  public class JwtTokenStoreConfig {
32      . . . . .
33  }

```

yml中添加jwt配置

```

1  tuling:
2    jwt:
3      keyPairName: jwt.jks
4      keyPairAlias: jwt
5      keyPairSecret: 123123
6      keyPairStoreSecret: 123123

```

第三步：修改JwtTokenStoreConfig的配置，支持非对称加密

```

1  @Bean
2  public JwtAccessTokenConverter jwtAccessTokenConverter(){
3      JwtAccessTokenConverter accessTokenConverter = new
4      JwtAccessTokenConverter();
5      //配置JWT使用的密钥
6      //accessTokenConverter.setSigningKey("123123");
7      //配置JWT使用的密钥 非对称加密
8      accessTokenConverter.setKeyPair(keyPair());
9      return accessTokenConverter;
10 }
11
12 @Autowired
13 private JwtCAProperties jwtCAProperties;

```

```

14
15 @Bean
16 public KeyPair keyPair() {
17     KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(new ClassPathResource(jwtCAPr
18     operties.getKeyPairName()), jwtCAProperties.getKeyPairSecret().toCharArray());
19     return keyStoreKeyFactory.getKeyPair(jwtCAProperties.getKeyPairAlias(), jwtCAProperties.gett
20     eyPairStoreSecret().toCharArray());
21 }

```

第四步：扩展JWT中的存储内容

有时候我们需要扩展JWT中存储的内容，根据自己业务添加字段到Jwt中。

继承TokenEnhancer实现一个JWT内容增强器

```

1 public class TulingTokenEnhancer implements TokenEnhancer {
2     @Override
3     public OAuth2AccessToken enhance(OAuth2AccessToken accessToken, OAuth2Authentication authent
4     ication) {
5         MemberDetails memberDetails = (MemberDetails) authentication.getPrincipal();
6         final Map<String, Object> additionalInfo = new HashMap<>();
7         final Map<String, Object> retMap = new HashMap<>();
8
9         //todo 这里暴露memberId到Jwt的令牌中,后期可以根据自己的业务需要 进行添加字段
10        additionalInfo.put("memberId",memberDetails.getUmsMember().getId());
11        additionalInfo.put("nickName",memberDetails.getUmsMember().getNickname());
12        additionalInfo.put("integration",memberDetails.getUmsMember().getIntegration());
13
14        retMap.put("additionalInfo",additionalInfo);
15
16        ((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(retMap);
17
18        return accessToken;
19    }
20 }
21
22

```

在JwtTokenStoreConfig中配置TulingTokenEnhancer

```

1 //JwtTokenStoreConfig.java
2 /**
3  * token的增强器 根据自己业务添加字段到Jwt中
4  * @return
5  */
6 @Bean
7 public TulingTokenEnhancer tulingTokenEnhancer() {
8     return new TulingTokenEnhancer();
9 }

```

在授权服务器配置中配置JWT的内容增强器

```

1 // TulingAuthorizationServerConfig.java
2 @Autowired
3 private TulingTokenEnhancer tulingTokenEnhancer;
4
5 @Override

```

```

6 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
7     //配置JWT的内容增强器
8     TokenEnhancerChain enhancerChain = new TokenEnhancerChain();
9     List<TokenEnhancer> delegates = new ArrayList<>();
10    delegates.add(tulingTokenEnhancer);
11    delegates.add(jwtAccessTokenConverter);
12    enhancerChain.setTokenEnhancers(delegates);
13
14    //使用密码模式需要配置
15    endpoints.authenticationManager(authenticationManagerBean)
16        .tokenStore(tokenStore) //指定token存储策略是jwt
17        .accessTokenConverter(jwtAccessTokenConverter)
18        .tokenEnhancer(enhancerChain) //配置tokenEnhancer
19        .reuseRefreshTokens(false) //refresh_token是否重复使用
20        .userDetailsService(tulingUserDetailsService) //刷新令牌授权包含对用户信息的检查
21        .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST); //支持GET, POST请求
22 }
23

```

1) 通过密码模式测试获取token

[illegible]

<https://jwt.io/>中校验token，可以获取到增强的用户信息，传入私钥和公钥可以校验通过。

```

    "alg": "RS256",
    "typ": "JWT"
  }
}

PAYLOAD: DATA

{
  "user_name": "test",
  "scope": [
    "all"
  ],
  "additionalInfo": {
    "nickName": "windir",
    "integration": 5000,
    "memberId": 1
  },
  "exp": 1618810609,
  "authorities": [
    "TEST"
  ],
  "jti": "e65e3c1a-5d29-46ac-8669-79a253b3094b",
  "client_id": "client"
}

VERIFY SIGNATURE

RSASHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
zkuVd649
NwIDAQAB
-----END PUBLIC KEY-----
123123
)

```

2) 测试校验token

GET

⌵

http://localhost:9999/oauth/check_token?token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUOIj0ZXN0Iiwic2Nv...

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	token	eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2...	
	Key	Value	Description

body Cookies (1) Headers (9) Test Results  Status: 200 OK Time: 94 ms Size: 512 B

3. 接入网关服务

在网关服务tulingmall-gateway中配置tulingmall-authcenter

1) yml中添加对tulingmall-authcenter的路由

```
1 server:
2   port: 9999
3   spring:
4     application:
5       name: tulingmall-gateway
6       #配置nacos注册中心地址
7     cloud:
8       nacos:
9         discovery:
10          server-addr: 192.168.65.232:8848 #注册中心地址
11          namespace: 80a98d11-492c-4008-85aa-32d889e9b0d0 #环境隔离
12
13     gateway:
14       routes:
15         - id: tulingmall-member #路由ID, 全局唯一
16           uri: lb://tulingmall-member
17           predicates:
18             - Path=/member/**,/sso/**
19         - id: tulingmall-promotion
20           uri: lb://tulingmall-promotion
21           predicates:
22             - Path=/coupon/**
23         - id: tulingmall-authcenter
24           uri: lb://tulingmall-authcenter
25           predicates:
26             - Path=/oauth/**
```

2) 编写GateWay的全局过滤器进行权限的校验拦截

认证过滤器AuthenticationFilter#filter中需要实现的逻辑

```
1 //1. 过滤不需要认证的url, 比如/oauth/**
2
3 //2. 获取token
4 // 从请求头中解析 Authorization value: bearer xxxxxxxx
5 // 或者从请求参数中解析 access_token
6
7 //3. 校验token
8 // 拿到token后, 通过公钥 (需要从授权服务获取公钥) 校验
9 // 校验失败或超时抛出异常
10
11 //4. 校验通过后, 从token中获取的用户登录信息存储到请求头中
```

1) 过滤不需要认证的url, 可以通过yml设置不需要认证的url。

```
1 /**
2   * @author Fox
3   *
4   * 认证过滤器: 实现认证逻辑
```

```

5  *
6  */
7  @Component
8  @Order(0)
9  @EnableConfigurationProperties(value = NotAuthUrlProperties.class)
10 public class AuthenticationFilter implements GlobalFilter, InitializingBean {
11
12     /**
13      * 请求各个微服务 不需要用户认证的URL
14      */
15     @Autowired
16     private NotAuthUrlProperties notAuthUrlProperties;
17
18     @Override
19     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
20
21         String currentUrl = exchange.getRequest().getURI().getPath();
22
23         //过滤不需要认证的url
24         if(shouldSkip(currentUrl)) {
25             //log.info("跳过认证的URL:{},currentUrl);
26             return chain.filter(exchange);
27         }
28         //log.info("需要认证的URL:{},currentUrl);
29
30
31         return chain.filter(exchange);
32     }
33
34
35     @Override
36     public void afterPropertiesSet() throws Exception {
37         //获取公钥 TODO
38
39     }
40
41
42     /**
43      * 方法实现说明:不需要授权的路径
44      * @author:smlz
45      * @param currentUrl 当前请求路径
46      * @return:
47      * @exception:
48      * @date:2019/12/26 13:49
49      */
50     private boolean shouldSkip(String currentUrl) {
51         //路径匹配器(简介SpringMvc拦截器的匹配器)
52         //比如/oauth/** 可以匹配/oauth/token /oauth/check_token等
53         PathMatcher pathMatcher = new AntPathMatcher();
54         for(String skipPath: notAuthUrlProperties.getShouldSkipUrls()) {
55             if(pathMatcher.match(skipPath, currentUrl)) {

```


[illegible]

Body

Cookies

Headers (3)

Test Results

200 OK

18.96 s

1.61 KB

Pretty

Raw

Preview

Visualize

JSON

```
1 {  
2   "code": 200,  
3   "message": "操作成功",  
4   "data": {  
5     "tokenHead": "bearer",  
6     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
           eyJlc2VyZXZhbmVudj0iOiJ0ZXN0Iiwic2NvcGUoIj0lcmVhZCJldCJhZGRpdGlmYmFScW5mbyI6eyJuYWxzTmF1  
           RpciIsImuldGVncmF0aW9uIjo1MDAwLCJtZW11ZXJJZCJiMXosImV4cCI6MTYxODk0NTk1NywiYXV0aG9ya2  
           VEVTVGVhZCJqdGkiOiJ0ZGU0YjIwM0YyIiwiaWF0IjEwM0YyM0QwLTMzM0AtOTBzYS1mMjkwMGZmYjhlYmQ1ClCjbGllbnRfaWQi  
           dTYwXSwiOiJlbWJlcj0.  
           aXgsAlhiqsmYv21De5kvJKLC5jL5piFqhXTDpJy0t7gxyboGmAHHNBK_o01KFznIADgyzX8TTbtvePX  
           nazRay94812rKzrSx12sZil57WNMA6siH7qyO4-sYcqW5_iNTar_zqu02UfR8e0kc1AZ_jFE1-WLAzBjreme  
           MVRKMdYnA7cFOVMRLku9mhArTYMvz015NLgmPMPuzalduOC3MPS_vlw7ztuI22nxvBRdz0c0C1sAknpQ
```

2) 解析请求, 获取token

从请求头中解析 Authorization value: bearer xxxxxxxx 或者 从请求参数中解析 access token

在AuthenticationFilter#filter中实现获取token的逻辑

```
1 //2. 获取token
2 // 从请求头中解析 Authorization value: bearer xxxxxxxx
3 // 或者从请求参数中解析 access_token
4 //第一步:解析出我们Authorization的请求头 value为: "bearer XXXXXXXXXXXXXXXX"
5 String authHeader = exchange.getRequest().getHeaders().getFirst("Authorization");
6
7 //第二步:判断Authorization的请求头是否为空
8 if(StringUtils.isEmpty(authHeader)) {
9     log.warn("需要认证的url,请求头为空");
10     throw new GatewayException(StatusCode.AUTHORIZATION_HEADER_IS_EMPTY);
11 }
```

测试：通过网关获取用户优惠券信息，因为请求头中不带token信息，所以会抛出异常

GET localhost:8888/member/center/coupons ...

Params Authorization Headers (8) Body Pre-request Script Tests Settings

<input checked="" type="checkbox"/>	Cache-Control ⓘ	no-cache	
<input checked="" type="checkbox"/>	Postman-Token ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/>	Host ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/>	User-Agent ⓘ	PostmanRuntime/7.26.10	
<input checked="" type="checkbox"/>	Accept ⓘ	*/*	
<input checked="" type="checkbox"/>	Accept-Encoding ⓘ	gzip, deflate, br	
<input checked="" type="checkbox"/>	Connection ⓘ	keep-alive	
<input checked="" type="checkbox"/>	memberId	1	
	Key	Value	Des

Body Cookies Headers (2) Test Results Status: 500 Internal Server Error

Pretty Raw Preview Visualize JSON

```

1  {
2    "timestamp": "2021-04-20T08:42:02.756+0000",
3    "path": "/member/center/coupons",
4    "status": 500,
5    "error": "Internal Server Error",
6    "message": "请求头中的token为空"
7  }

```

3) 校验token

拿到token后，通过公钥（需要从授权服务获取公钥）校验，校验失败或超时抛出异常引入依赖

```

1  <!-- 添加jwt相关的包 -->
2  <dependency>
3    <groupId>io.jsonwebtoken</groupId>
4    <artifactId>jjwt-api</artifactId>
5    <version>0.10.5</version>
6  </dependency>
7  <dependency>
8    <groupId>io.jsonwebtoken</groupId>
9    <artifactId>jjwt-impl</artifactId>
10   <version>0.10.5</version>
11   <scope>runtime</scope>
12 </dependency>
13 <dependency>
14   <groupId>io.jsonwebtoken</groupId>
15   <artifactId>jjwt-jackson</artifactId>
16   <version>0.10.5</version>
17   <scope>runtime</scope>
18 </dependency>

```

在AuthenticationFilter#filter中实现校验token的逻辑

```

1 //3. 校验token
2 // 拿到token后，通过公钥（需要从授权服务获取公钥）校验
3 // 校验失败或超时抛出异常
4 //第三步 校验我们的jwt 若jwt不对或者超时都会抛出异常
5 Claims claims = JwtUtils.validateJwtToken(authHeader,publicKey);
6

```

校验token逻辑

```

1 // AuthenticationFilter.java
2 /**
3  * 请求头中的 token的开始
4  */
5 private static final String AUTH_HEADER = "bearer ";
6
7 public static Claims validateJwtToken(String authHeader,PublicKey publicKey) {
8     String token =null ;
9     try{
10         token = StringUtils.substringAfter(authHeader, AUTH_HEADER);
11
12         Jwt<JwsHeader, Claims> parseClaimsJwt = Jwts.parser().setSigningKey(publicKey).parseClaimsJws(token);
13
14         Claims claims = parseClaimsJwt.getBody();
15
16         //log.info("claims:{}",claims);
17
18         return claims;
19     }catch(Exception e){
20
21
22         log.error("校验token异常:{},异常信息:{}", token,e.getMessage());
23
24         throw new GatewayException(ErrorCode.JWT_TOKEN_EXPIRE);
25     }
26 }

```

工具类

```

1 @Slf4j
2 public class JwtUtils {
3
4     /**
5      * 认证服务器许可我们的网关的clientId(需要在oauth_client_details表中配置)
6      */
7     private static final String CLIENT_ID = "tulingmall-gateway";
8
9     /**
10      * 认证服务器许可我们的网关的client_secret(需要在oauth_client_details表中配置)
11      */
12     private static final String CLIENT_SECRET = "123123";
13
14     /**
15      * 认证服务器暴露的获取token_key的地址

```

```

16  */
17  private static final String AUTH_TOKEN_KEY_URL = "http://tulingmall-auth/oauth/token_key";
18
19  /**
20   * 请求头中的 token的开始
21   */
22  private static final String AUTH_HEADER = "bearer ";
23
24  /**
25   * 方法实现说明：通过远程调用获取认证服务器颁发jwt的解析的key
26   * @author:smlz
27   * @param restTemplate 远程调用的操作类
28   * @return: tokenKey 解析jwt的tokenKey
29   * @exception:
30   * @date:2020/1/22 11:31
31   */
32  private static String getTokenKeyByRemoteCall(RestTemplate restTemplate) throws GatewayException {
33
34      //第一步:封装请求头
35      HttpHeaders headers = new HttpHeaders();
36      headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
37      headers.setBasicAuth(CLIENT_ID, CLIENT_SECRET);
38      HttpEntity<MultiValueMap<String, String>> entity = new HttpEntity<>(null, headers);
39
40      //第二步:远程调用获取token_key
41      try {
42
43          ResponseEntity<Map> response = restTemplate.exchange(AUTH_TOKEN_KEY_URL, HttpMethod.GET, entity, Map.class);
44
45          String tokenKey = response.getBody().get("value").toString();
46
47          log.info("去认证服务器获取Token_Key:{}", tokenKey);
48
49          return tokenKey;
50
51      } catch (Exception e) {
52
53          log.error("远程调用认证服务器获取Token_Key失败:{}", e.getMessage());
54
55          throw new GatewayException(ResultCode.GET_TOKEN_KEY_ERROR);
56      }
57  }
58
59  /**
60   * 方法实现说明:生成公钥
61   * @author:smlz
62   * @param restTemplate:远程调用操作类
63   * @return: PublicKey 公钥对象
64   * @exception:

```

```

65  * @date:2020/1/22 11:52
66  */
67  public static PublicKey genPulicKey(RestTemplate restTemplate) throws GateWayException {
68
69      String tokenKey = getTokenKeyByRemoteCall(restTemplate);
70
71      try{
72
73          //把获取的公钥开头和结尾替换掉
74          String dealTokenKey =tokenKey.replaceAll("\\\\-*BEGIN PUBLIC KEY\\\\-*", "").replaceAll("\\\\-*END
PUBLIC KEY\\\\-*", "").trim();
75
76          java.security.Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider());
77
78          X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(Base64.decodeBase64(dealTokenKey));
79
80          KeyFactory keyFactory = KeyFactory.getInstance("RSA");
81
82          PublicKey publicKey = keyFactory.generatePublic(pubKeySpec);
83
84          log.info("生成公钥:{}",publicKey);
85
86          return publicKey;
87
88      }catch (Exception e) {
89
90          log.info("生成公钥异常:{}",e.getMessage());
91
92          throw new GateWayException(ResultCode.GEN_PUBLIC_KEY_ERROR);
93      }
94  }
95
96  public static Claims validateJwtToken(String authHeader,PublicKey publicKey) {
97      String token =null ;
98      try{
99          token = StringUtils.substringAfter(authHeader, AUTH_HEADER);
100
101          Jwt<JwsHeader, Claims> parseClaimsJwt = Jwts.parser().setSigningKey(publicKey).parseClaimsJ
ws(token);
102
103          Claims claims = parseClaimsJwt.getBody();
104
105          //log.info("claims:{}",claims);
106
107          return claims;
108
109      }catch(Exception e){
110
111          log.error("校验token异常:{}",异常信息:{}",token,e.getMessage());
112
113          throw new GateWayException(ResultCode.JWT_TOKEN_EXPIRE);

```



```
114 }
115 }
116 }
```

需要从tulingmall-authcenter获取公钥，实现公钥获取逻辑

```
1 // AuthenticationFilter.java
2 /**
3  * jwt的公钥,需要网关启动,远程调用认证中心去获取公钥
4  */
5 private PublicKey publicKey;
6
7 @Autowired
8 private RestTemplate restTemplate;
9
10 @Override
11 public void afterPropertiesSet() throws Exception {
12     //获取公钥 TODO
13     this.publicKey = JwtUtils.genPulicKey(restTemplate);
14 }
15
16 @Configuration
17 public class RibbonConfig {
18
19     @Autowired
20     private LoadBalancerClient loadBalancer;
21
22     @Bean
23     public RestTemplate restTemplate(){
24         RestTemplate restTemplate = new RestTemplate();
25         restTemplate.setInterceptors(
26             Collections.singletonList(
27                 new LoadBalancerInterceptor(loadBalancer)));
28
29         return restTemplate;
30     }
31
32 }
```

注意： 此处不能直接通过@LoadBalancer配置RestTemplate去获取公钥，思考为什么？

源码参考：

org.springframework.cloud.client.loadbalancer.LoadBalancerAutoConfiguration

org.springframework.beans.factory.support.DefaultListableBeanFactory#preInstantiateSingletons

测试： 正确的token，通过网关获取用户优惠券信息

GET

localhost:8888/member/center/coupons...

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

<input checked="" type="checkbox"/>	Postman-Token ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/>	Host ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/>	User-Agent ⓘ	PostmanRuntime/7.26.10	
<input checked="" type="checkbox"/>	Accept ⓘ	*/*	
<input checked="" type="checkbox"/>	Accept-Encoding ⓘ	gzip, deflate, br	
<input checked="" type="checkbox"/>	Connection ⓘ	keep-alive	
<input checked="" type="checkbox"/>	memberId	1	
<input checked="" type="checkbox"/>	Authorization	bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ...	
	Key	Value	Description

Body

Cookies

Headers (3)

Test Results

Status: 200 OKTime: 62 msSize

Pretty

Raw

Preview

Visualize

JSON

```
1  {
2    "code": 200,
3    "message": "操作成功",
4    "data": [
5      {
6        "id": 2,
7        "couponId": 2,
8        "memberId": 1,
9        "couponCode": "4931048380330002",
10       "memberNickname": "windir",
11       "getType": 1,
12       "createTime": "2018-08-29T14:04:12.000+0000",
13       "useStatus": 0,
14       "useTime": "2019-03-21T15:03:40.000+0000",
15       "orderId": 12,
16       "orderSn": "201809150101000001"
```

Bc

错误的token，抛出异常

GET localhost:8888/member/center/coupons ...

Params Authorization Headers (9) Body Pre-request Script Tests Settings

<input checked="" type="checkbox"/>	Postman-Token ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/>	Host ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/>	User-Agent ⓘ	PostmanRuntime/7.26.10	
<input checked="" type="checkbox"/>	Accept ⓘ	*/*	
<input checked="" type="checkbox"/>	Accept-Encoding ⓘ	gzip, deflate, br	
<input checked="" type="checkbox"/>	Connection ⓘ	keep-alive	
<input checked="" type="checkbox"/>	memberId	1	
<input checked="" type="checkbox"/>	Authorization	bearer 853967fb-6f9e-4813-b35e-cf4b0225e...	
	Key	Value	Description

Body Cookies Headers (2) Test Results Status: 500 Internal Server Error Time: 10

Pretty Raw Preview Visualize JSON

```

1  {
2    "timestamp": "2021-04-20T09:14:15.188+0000",
3    "path": "/member/center/coupons",
4    "status": 500,
5    "error": "Internal Server Error",
6    "message": "token校验异常"
7  }

```

4) 校验通过后，从token中获取的用户登录信息存储到请求头中

在AuthenticationFilter#filter中，将从token中获取的用户登陆信息存储到请求头中

```

1 //4. 校验通过后，从token中获取的用户登录信息存储到请求头中
2 //第四步 把从jwt中解析出来的 用户登陆信息存储到请求头中
3 ServerWebExchange webExchange = wrapHeader(exchange,claims);

```

解析用户登录信息存储到请求头中

```

1 // AuthenticationFilter.java
2
3 private ServerWebExchange wrapHeader(ServerWebExchange serverWebExchange,Claims claims) {
4
5     String loginUserInfo = JSON.toJSONString(claims);
6
7     //log.info("jwt的用户信息:{",loginUserInfo);
8
9     String memberId = claims.get("additionalInfo", Map.class).get("memberId").toString();
10
11     String nickName = claims.get("additionalInfo",Map.class).get("nickName").toString();
12
13     //向headers中放文件，记得build
14     ServerHttpRequest request = serverWebExchange.getRequest().mutate()
15     .header("username",claims.get("user_name",String.class))
16     .header("memberId",memberId)
17     .header("nickName",nickName)
18     .build();
19
20     //将现在的request 变成 change对象

```

```
21 return serverWebExchange.mutate().request(request).build();  
22 }
```