主讲老师：Fox

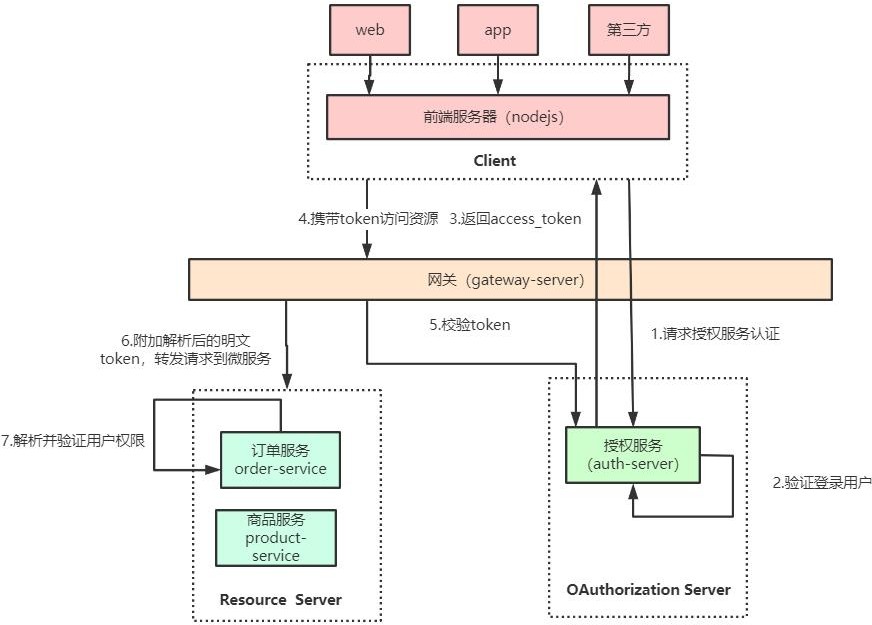
有道笔记地址：<https://note.youdao.com/s/AKMhjuXL>

# 微服务网关整合 OAuth2.0 设计思路分析

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）



# 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

* 1. **引入依赖**

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

* 1. **添加 yml 配置**

server:

port: 9999 spring:

application:

name: tulingmall-authcenter #配置 nacos 注册中心地址

cloud:

nacos:

discovery:

server-addr: 192.168.65.103:8848 #注册中心地址

namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618 #环境隔离 datasource:

url: jdbc:mysql://tuling.com:3306/tlmall\_oauth?serverTimezone=UTC&useSSL=false&useUnicode=tr ue&characterEncoding=UTF-8

username: root password: root druid:

initial-size: 5 #连接池初始化大小

min-idle: 10 #最小空闲连接数

max-active: 20 #最大连接数 web-stat-filter:

exclusions: "\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*" #不统计这些请求数据 stat-view-servlet: #访问监控网页的登录用户名和密码

login-username: druid

login-password: druid

* 1. **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**

@Configuration @EnableAuthorizationServer

public class TulingAuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Autowired

private DataSource dataSource;

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

// 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

clients.withClientDetails(clientDetails());

}

@Bean

public ClientDetailsService clientDetails(){

return new JdbcClientDetailsService(dataSource);

}

}

在 oauth\_client\_details 中添加第三方客户端信息（client\_id client\_secret scope 等等）

CREATE TABLE `oauth\_client\_details` (

`client\_id` varchar(128) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL,

`resource\_ids` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`client\_secret` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`scope` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`authorized\_grant\_types` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`web\_server\_redirect\_uri` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`authorities` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`access\_token\_validity` int(11) NULL DEFAULT NULL,

`refresh\_token\_validity` int(11) NULL DEFAULT NULL,

`additional\_information` varchar(4096) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

`autoapprove` varchar(256) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,

PRIMARY KEY (`client\_id`) USING BTREE

) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Dynamic;



**基于内存模式配置授权服务器存储第三方客户端的信息**

//TulingAuthorizationServerConfig.java @Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

// 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

// clients.withClientDetails(clientDetails());

/\*\*

\*授权码模式

\*http://localhost:9999/oauth/authorize?response\_type=code&client\_id=client&redirect\_uri=htt p://www.baidu.com&scope=all

\*

\* password 模式

\* http://localhost:8080/oauth/token?username=fox&password=123456&grant\_type=password&cl ient\_id=client&client\_secret=123123&scope=all

\*

\*/ clients.inMemory()

//配置 client\_id

.withClient("client")

//配置 client-secret

.secret(passwordEncoder.encode("123123"))

//配置访问 token 的有效期

.accessTokenValiditySeconds(3600)

//配置刷新 token 的有效期

.refreshTokenValiditySeconds(864000)

//配置 redirect\_uri，用于授权成功后跳转

.redirectUris("[http://www.baidu.com](http://www.baidu.com/)")

//配置申请的权限范围

.scopes("all")

/\*\*

* 配置 grant\_type，表示授权类型
* authorization\_code: 授权码
* password： 密码
* refresh\_token: 更新令牌

\*/

.authorizedGrantTypes("authorization\_code","password","refresh\_token");

}

### 配置 SpringSecurity

@Configuration

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Bean

public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder();

}

@Autowired

private TulingUserDetailsService tulingUserDetailsService;

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

// 实现 UserDetailsService 获取用户信息

auth.userDetailsService(tulingUserDetailsService);

}

@Bean @Override

public AuthenticationManager authenticationManagerBean() throws Exception {

// oauth2 密码模式需要拿到这个 bean return super.authenticationManagerBean();

}

@Override

protected void configure(HttpSecurity http) throws Exception { http.formLogin().permitAll()

.and().authorizeRequests()

.antMatchers("/oauth/\*\*").permitAll()

.anyRequest()

.authenticated()

.and().logout().permitAll()

.and().csrf().disable();

}

}

**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：**

@Slf4j

@Component

public class TulingUserDetailsService implements UserDetailsService {

@Override

public UserDetails

loadUserByUsername(String

username)

throws

UsernameNotFoundException {

// 加载用户信息

if(StringUtils.isEmpty(username)) {

log.warn("用户登陆用户名为空:{}",username);

throw new UsernameNotFoundException("用户名不能为空");

}

UmsMember umsMember = getByUsername(username);

if(null == umsMember) {

log.warn("根据用户名没有查询到对应的用户信息:{}",username);

}

log.info("根据用户名:{}获取用户登陆信息:{}",username,umsMember);

// 会员信息的封装 implements UserDetails

MemberDetails memberDetails = new MemberDetails(umsMember);

return memberDetails;

}

@Autowired

private UmsMemberFeignService umsMemberFeignService;

public UmsMember getByUsername(String username) {

// fegin 获取会员信息

CommonResult<UmsMember> umsMemberCommonResult = umsMemberFeignService.loadUserByUsername(username);

return umsMemberCommonResult.getData();

}

}

@FeignClient(value = "tulingmall-member",path="/member/center") public interface UmsMemberFeignService {

@RequestMapping("/loadUmsMember")

CommonResult<UmsMember> loadUserByUsername(@RequestParam("username") String username);

}

public class MemberDetails implements UserDetails { private UmsMember umsMember;

public MemberDetails(UmsMember umsMember) { this.umsMember = umsMember;

}

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

//返回当前用户的权限

return Arrays.asList(new SimpleGrantedAuthority("TEST"));

}

@Override

public String getPassword() {

return umsMember.getPassword();

}

@Override

public String getUsername() {

return umsMember.getUsername();

}

@Override

public boolean isAccountNonExpired() {

return true;

}

@Override

public boolean isAccountNonLocked() { return true;

}

@Override

public boolean isCredentialsNonExpired() { return true;

}

@Override

public boolean isEnabled() {

return umsMember.getStatus()==1;

}

public UmsMember getUmsMember() { return umsMember;

}

}

**修改授权服务配置，支持密码模式**

//TulingAuthorizationServerConfig.java @Autowired

private TulingUserDetailsService tulingUserDetailsService;

@Autowired

private AuthenticationManager authenticationManagerBean;

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception

{

//使用密码模式需要配置 endpoints.authenticationManager(authenticationManagerBean)

.reuseRefreshTokens(false) //refresh\_token 是否重复使用

.userDetailsService(tulingUserDetailsService) //刷新令牌授权包含对用户信

息的检查

.allowedTokenEndpointRequestMethods(HttpMethod.GET,HttpMethod.POS T); //支持 GET,POST 请求

}

/\*\*

* 授权服务器安全配置
* @param security
* @throws Exception

\*/ @Override

public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {

//第三方客户端校验 token 需要带入 clientId 和 clientSecret 来校验

security.checkTokenAccess("isAuthenticated()")

.tokenKeyAccess("isAuthenticated()");// 来获取我们的 tokenKey 需要带入

clientId,clientSecret

//允许表单认证 security.allowFormAuthenticationForClients();

}

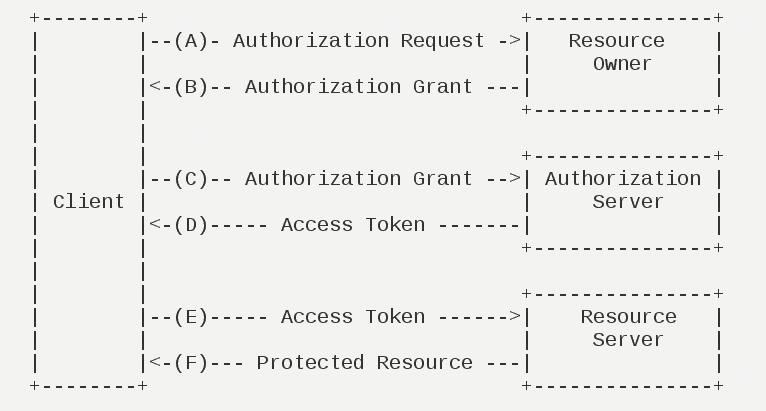
### 测试模拟用户登录

**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：

* + 1. 用户访问客户端，后者将前者导向授权服务器。
    2. 用户选择是否给予客户端授权。
    3. 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。

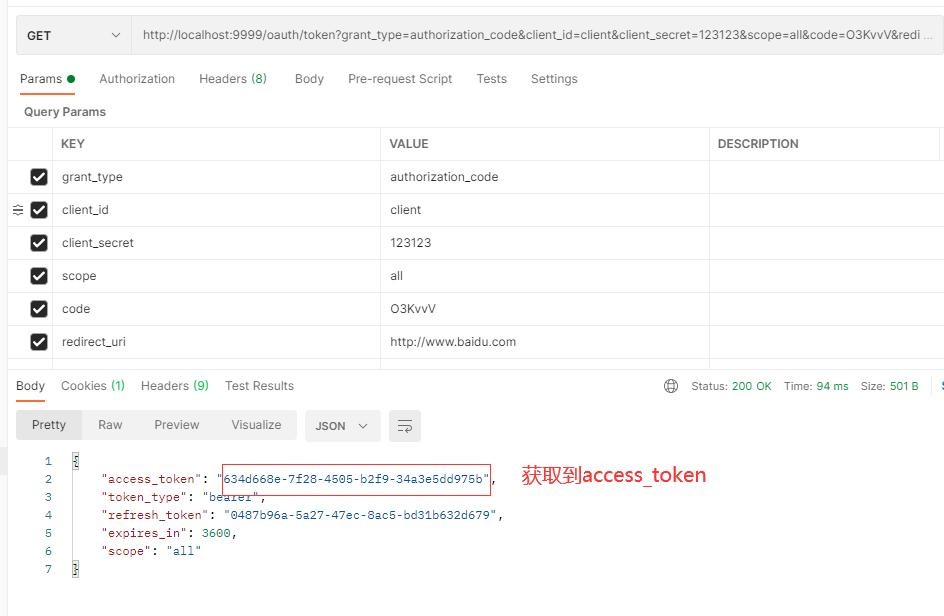
* + 1. 客户端收到授权码，附上早先的"重定向 URI"，向授权服务器申请令牌。这一步是在客户端的后台的服务器上完成的，对用户不可见。
    2. 授权服务器核对了授权码和重定向 URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。

## http://localhost:9999/oauth/authorize?response\_type=code&client\_id=client

&redirect\_uri=http://www.baidu.com&scope=all

获取到 code





**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"

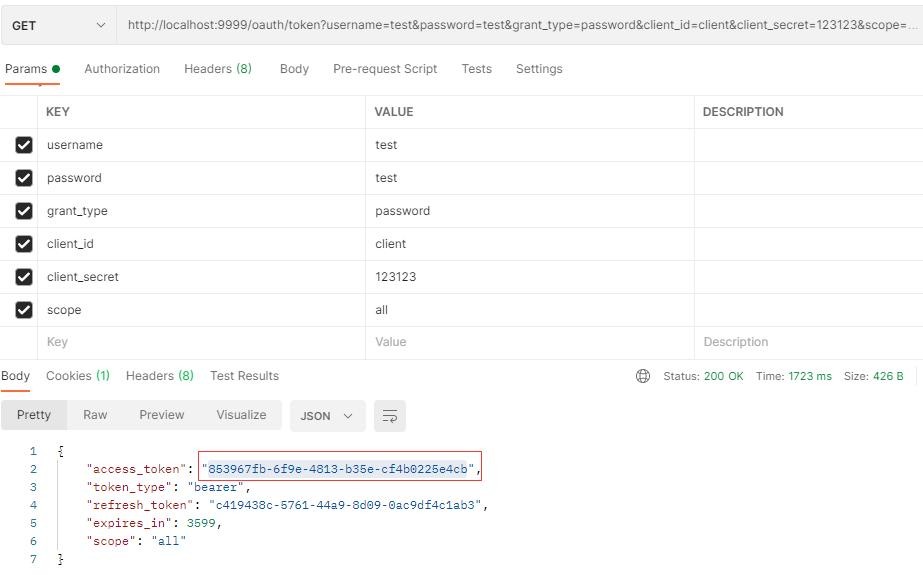
（password）。

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。

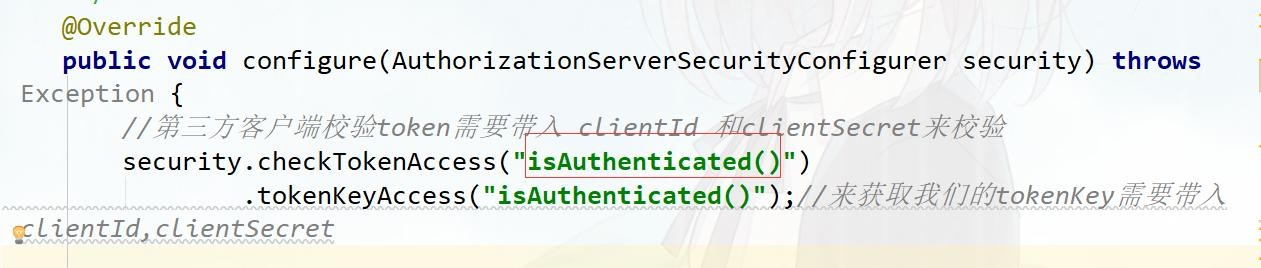
适用场景：自家公司搭建的授权服务器

测试获取 token

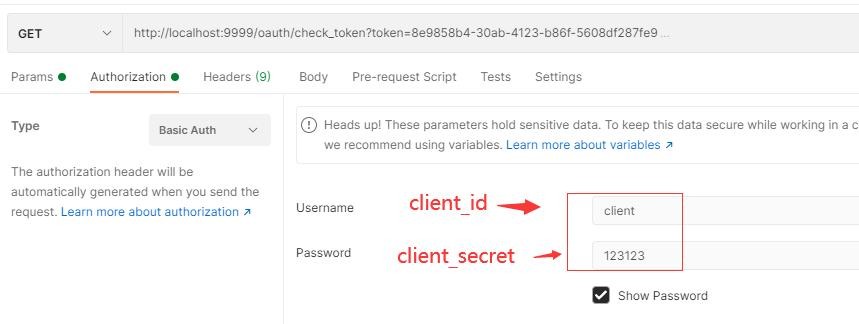
http://localhost:9999/oauth/token?username=test&password=test&grant\_type=password&client\_id=clien t&client\_secret=123123&scope=all



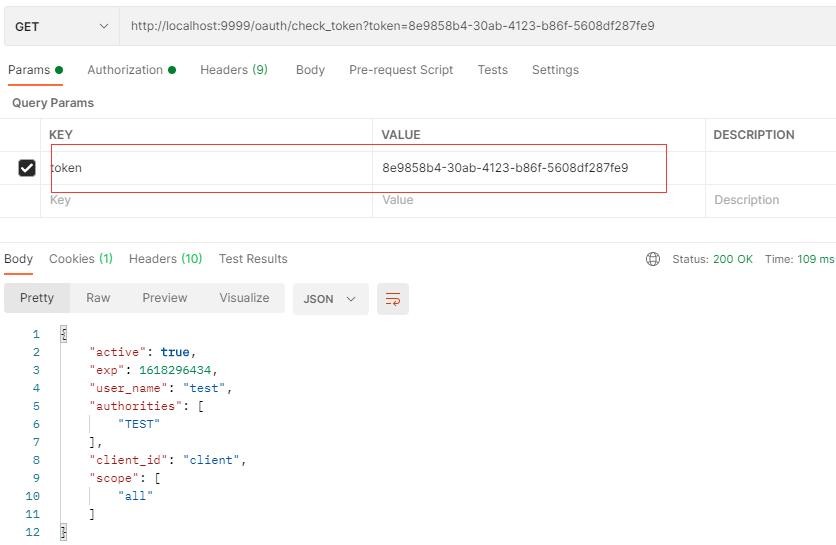
测试校验 token 接口



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



注意： 传参是 token



### 配置资源服务器

@Configuration @EnableResourceServer

public class TulingResourceServerConfig extends ResourceServerConfigurerAdapter {

@Override

public void configure(HttpSecurity http) throws Exception { http.authorizeRequests()

.anyRequest().authenticated();

}

}

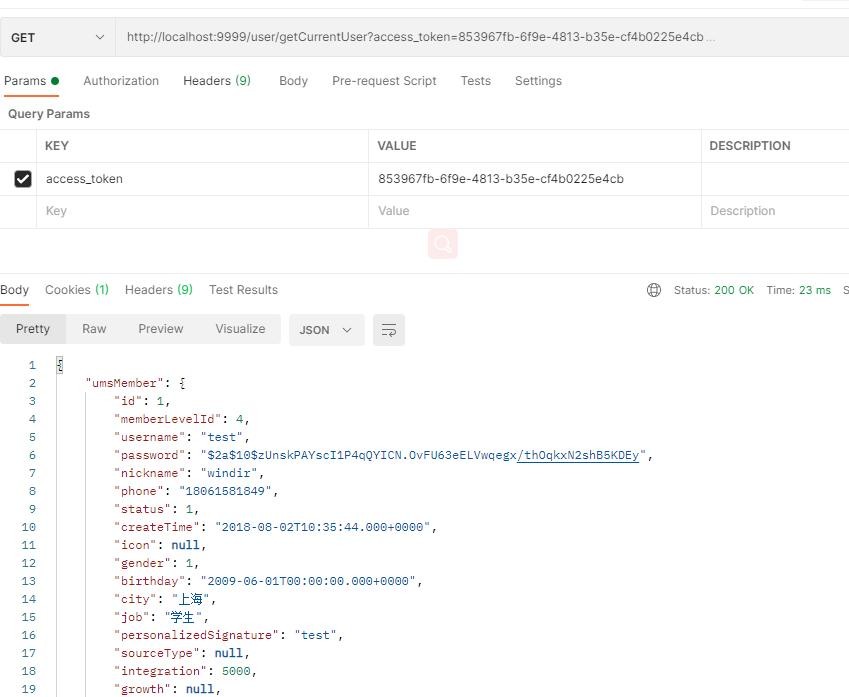
@RestController @RequestMapping("/user") public class UserController {

@RequestMapping("/getCurrentUser")

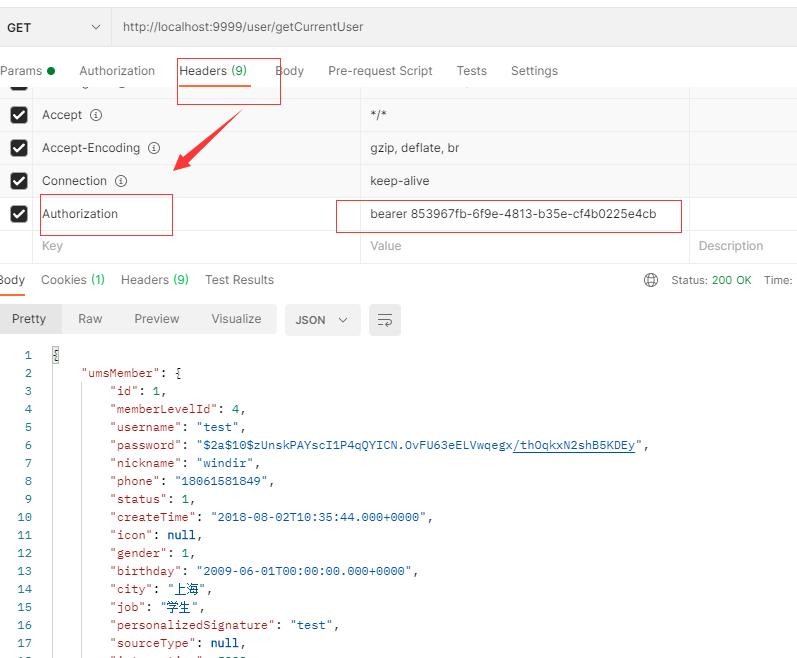
public Object getCurrentUser(Authentication authentication) { return authentication.getPrincipal();

}

}

测试携带 token 访问资源

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。**

### 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 是不可窜改的。

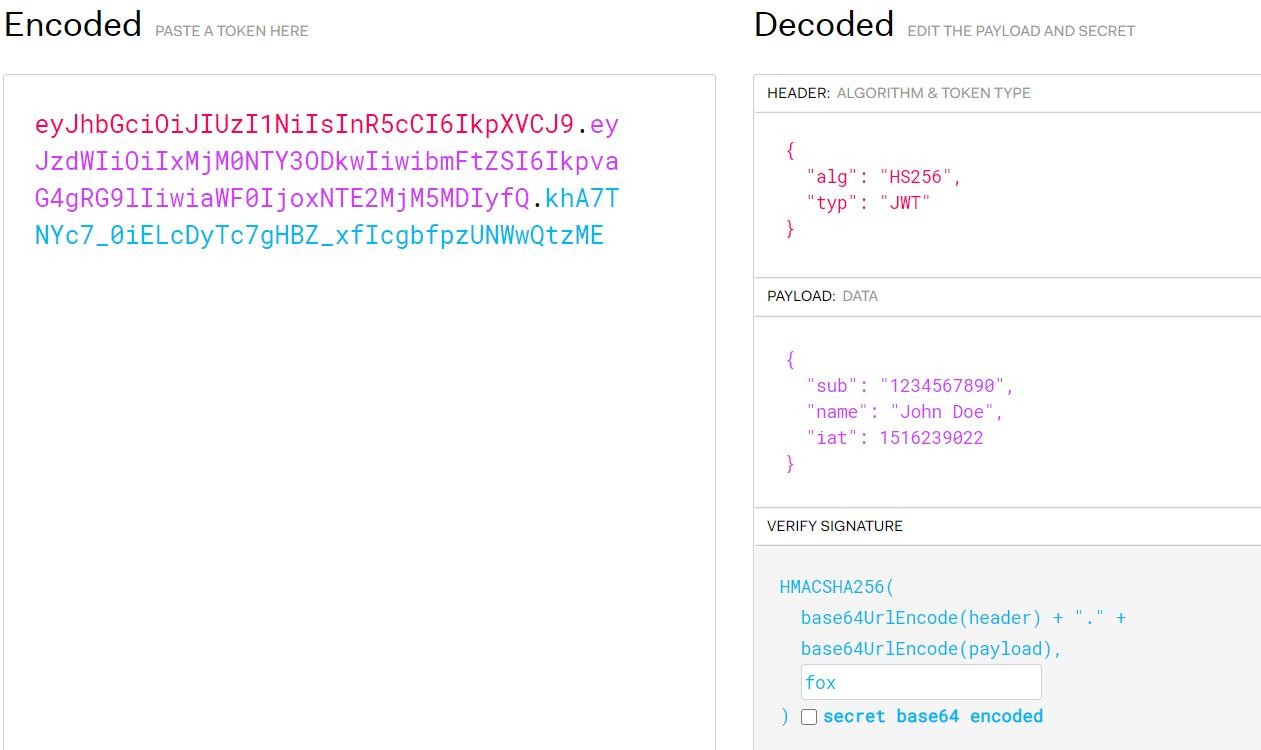
JWS：指的是签过名的 JWT，即拥有签名的 JWT。

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）。

**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：

{

"alg": "HS256",

"typ": "JWT"

}

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:

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：

{

"sub": "1234567890",

"name": "John Doe", "iat": 1516239022

}

然后将其进行 base64 加密，得到 Jwt 的第二部分:

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:

var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);

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:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIi

wiaWF0IjoxNTE2MjM5MDIyfQ.khA7TNYc7\_0iELcDyTc7gHBZ\_xfIcgbfpzUNWwQtzME

注意：secret 是保存在服务器端的，jwt 的签发生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) token+redis**

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

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

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-jwt</artifactId>

<version>1.0.9.RELEASE</version>

</dependency>

添加 JWT 配置

@Configuration

public class JwtTokenStoreConfig {

@Bean

public TokenStore jwtTokenStore(){

return new JwtTokenStore(jwtAccessTokenConverter());

}

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter(){ JwtAccessTokenConverter accessTokenConverter = new

JwtAccessTokenConverter();

//配置 JWT 使用的秘钥 accessTokenConverter.setSigningKey("123123"); return accessTokenConverter;

}

}

在授权服务器配置中指定令牌的存储策略为 JWT

//TulingAuthorizationServerConfig.java

@Autowired @Qualifier("jwtTokenStore") private TokenStore tokenStore;

@Autowired

private JwtAccessTokenConverter jwtAccessTokenConverter;

@Autowired

private TulingUserDetailsService tulingUserDetailsService;

@Autowired

private AuthenticationManager authenticationManagerBean;

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

//使用密码模式需要配置 endpoints.authenticationManager(authenticationManagerBean)

.tokenStore(tokenStore) //指定 token 存储策略是 jwt

.accessTokenConverter(jwtAccessTokenConverter)

.reuseRefreshTokens(false) //refresh\_token 是否重复使用

.userDetailsService(tulingUserDetailsService) //刷新令牌授权包含对用户信息的

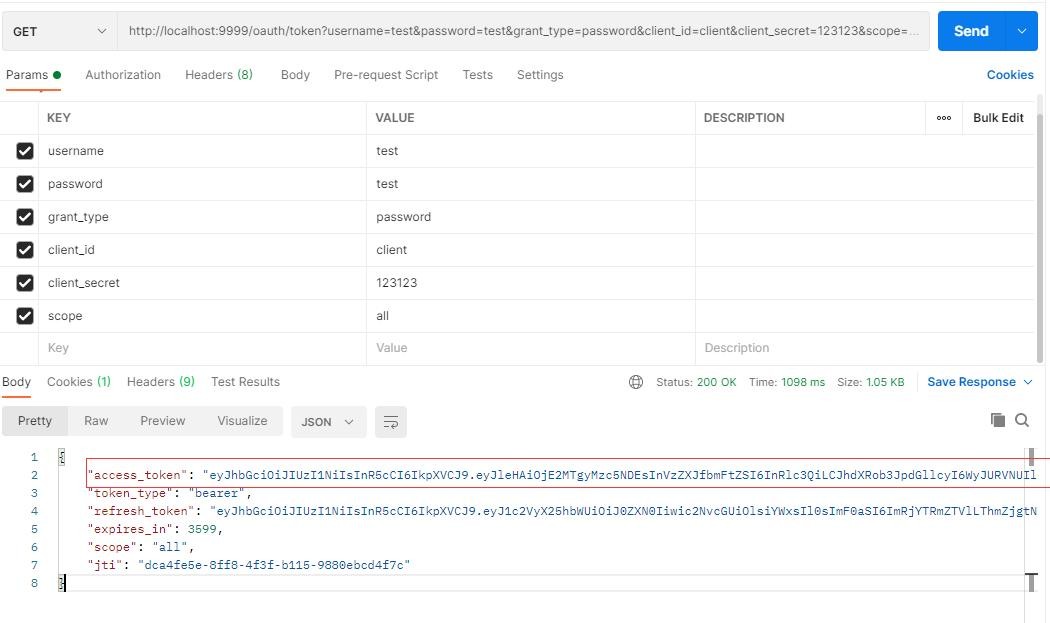
检查

.allowedTokenEndpointRequestMethods(HttpMethod.GET,HttpMethod.POST); //

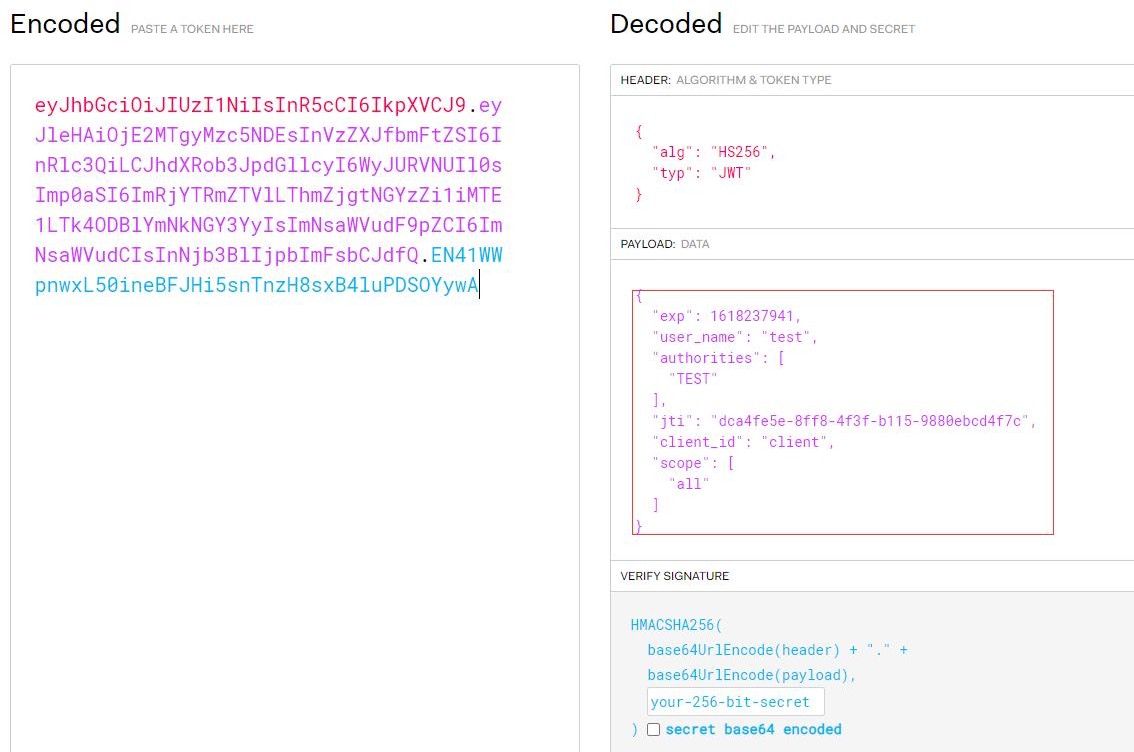
支持 GET,POST 请求

}

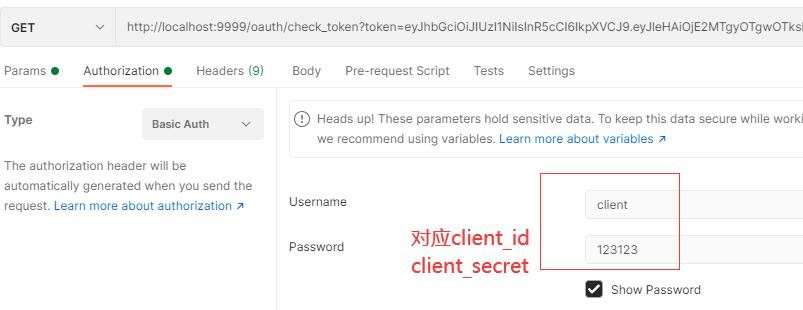
密码模式测试：

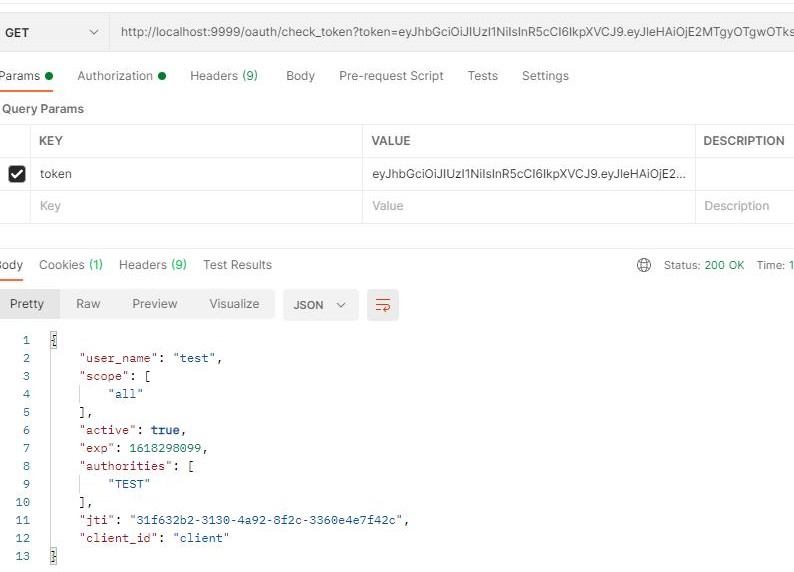
http://localhost:9999/oauth/token?username=test&password=test&grant\_type=password&client\_id=clien t&client\_secret=123123&scope=all

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

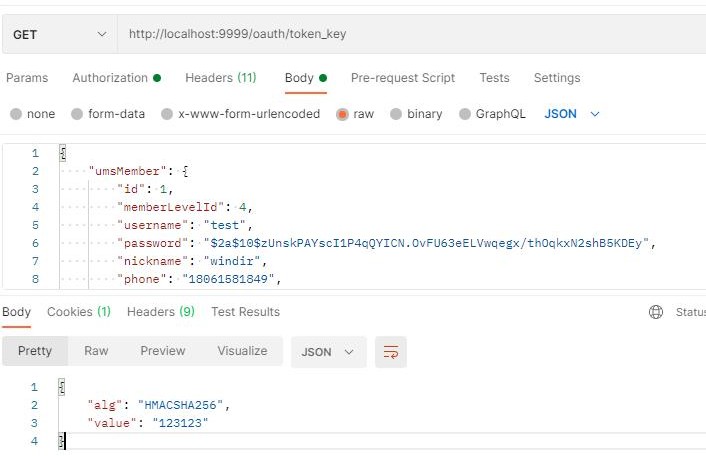


测试校验 token

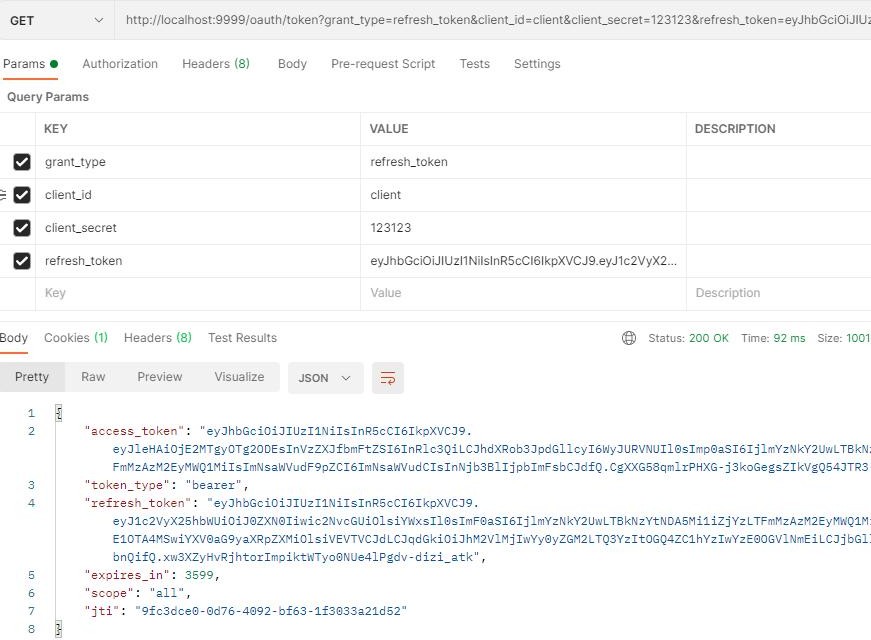




测试获取 token\_key



测试刷新 token



### 优化：实现 JWT 非对称加密（公钥私钥）

安全，利用 publickey 校验 token 可以减少一次远程调用

**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)

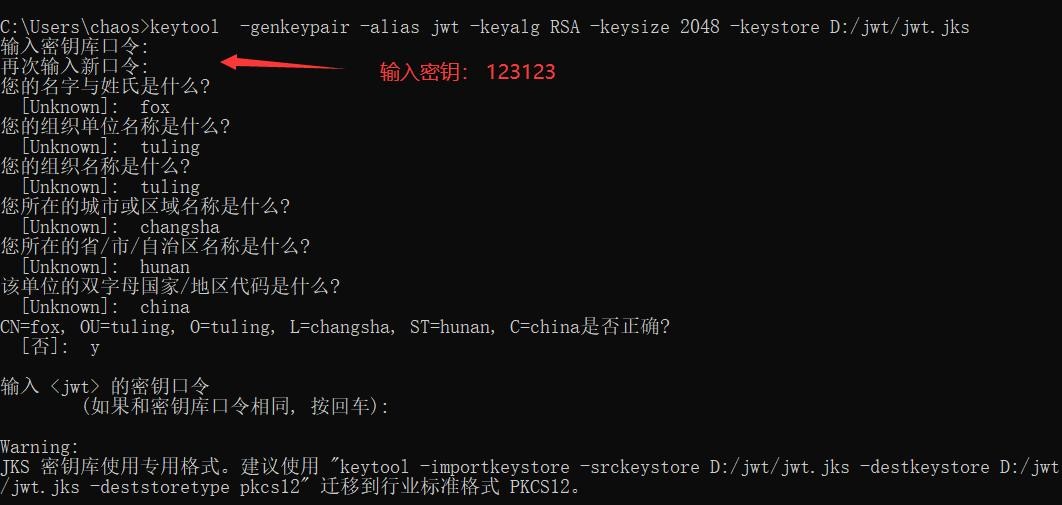
-keystore D:/jwt/jwt.jks(指定生成证书的位置和证书名称)

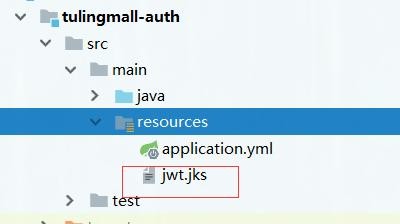
-storepass 123456(获取 keystore 信息的密码)

-storetype (指定密钥仓库类型)

使用 "keytool -help" 获取所有可用命令

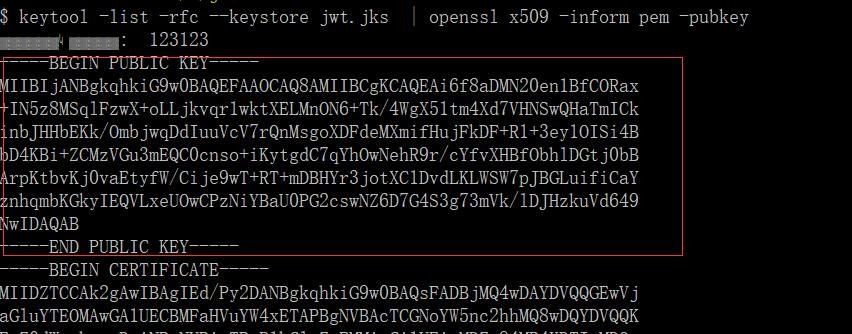
keytool -genkeypair -alias jwt -keyalg RSA -keysize 2048 -keystore D:/jwt/jwt.jks



将生成的 jwt.jks 文件 cope 到授权服务器的 resource 目录下

查看公钥信息

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



**第二步：授权服务中增加jwt 的属性配置类**

@Data

@ConfigurationProperties(prefix = "tuling.jwt") public class JwtCAProperties {

/\*\*

* 证书名称

\*/

private String keyPairName;

/\*\*

* 证书别名

\*/

private String keyPairAlias;

/\*\*

* 证书私钥

\*/

private String keyPairSecret;

/\*\*

* 证书存储密钥

\*/

private String keyPairStoreSecret;

}

@Configuration

// 指定属性配置类

@EnableConfigurationProperties(value = JwtCAProperties.class) public class JwtTokenStoreConfig {

。。。。。。

}

yml 中添加 jwt 配置

tuling:

jwt:

keyPairName: jwt.jks keyPairAlias: jwt keyPairSecret: 123123

keyPairStoreSecret: 123123

**第三步：修改JwtTokenStoreConfig 的配置，支持非对称加密**

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter(){

JwtAccessTokenConverter accessTokenConverter = new

JwtAccessTokenConverter();

//配置 JWT 使用的秘钥

//accessTokenConverter.setSigningKey("123123");

//配置 JWT 使用的秘钥 非对称加密

accessTokenConverter.setKeyPair(keyPair()); return accessTokenConverter;

}

@Autowired

private JwtCAProperties jwtCAProperties;

@Bean

public KeyPair keyPair() { KeyStoreKeyFactory

keyStoreKeyFactory

=

new

KeyStoreKeyFactory(new

ClassPathResource(jwtCAProperties.getKeyPairName()),

jwtCAProperties.getKeyPairSecret().toCharArray());

return keyStoreKeyFactory.getKeyPair(jwtCAProperties.getKeyPairAlias(), jwtCAProperties.getKeyPairStoreSecret().toCharArray());

}

**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

public class TulingTokenEnhancer implements TokenEnhancer {

@Override

public OAuth2AccessToken enhance(OAuth2AccessToken

accessToken,

OAuth2Authentication authentication) {

MemberDetails memberDetails = (MemberDetails) authentication.getPrincipal();

final Map<String, Object> additionalInfo = new HashMap<>(); final Map<String, Object> retMap = new HashMap<>();

//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

additionalInfo.put("memberId",memberDetails.getUmsMember().getId());

additionalInfo.put("nickName",memberDetails.getUmsMember().getNickname()); additionalInfo.put("integration",memberDetails.getUmsMember().getIntegration());

retMap.put("additionalInfo",additionalInfo);

((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(retMap);

return accessToken;

}

}

在 JwtTokenStoreConfig 中配置 TulingTokenEnhancer

//JwtTokenStoreConfig.java

/\*\*

* token 的增强器根据自己业务添加字段到 Jwt 中
* @return

\*/ @Bean

public TulingTokenEnhancer tulingTokenEnhancer() { return new TulingTokenEnhancer();

}

在授权服务器配置中配置 JWT 的内容增强器

// TulingAuthorizationServerConfig.java @Autowired

private TulingTokenEnhancer tulingTokenEnhancer;

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

//配置 JWT 的内容增强器

TokenEnhancerChain enhancerChain = new TokenEnhancerChain(); List<TokenEnhancer> delegates = new ArrayList<>(); delegates.add(tulingTokenEnhancer); delegates.add(jwtAccessTokenConverter); enhancerChain.setTokenEnhancers(delegates);

//使用密码模式需要配置 endpoints.authenticationManager(authenticationManagerBean)

.tokenStore(tokenStore) //指定 token 存储策略是 jwt

.accessTokenConverter(jwtAccessTokenConverter)

.tokenEnhancer(enhancerChain) //配置 tokenEnhancer

.reuseRefreshTokens(false) //refresh\_token 是否重复使用

.userDetailsService(tulingUserDetailsService) //刷新令牌授权包含对用户信息的

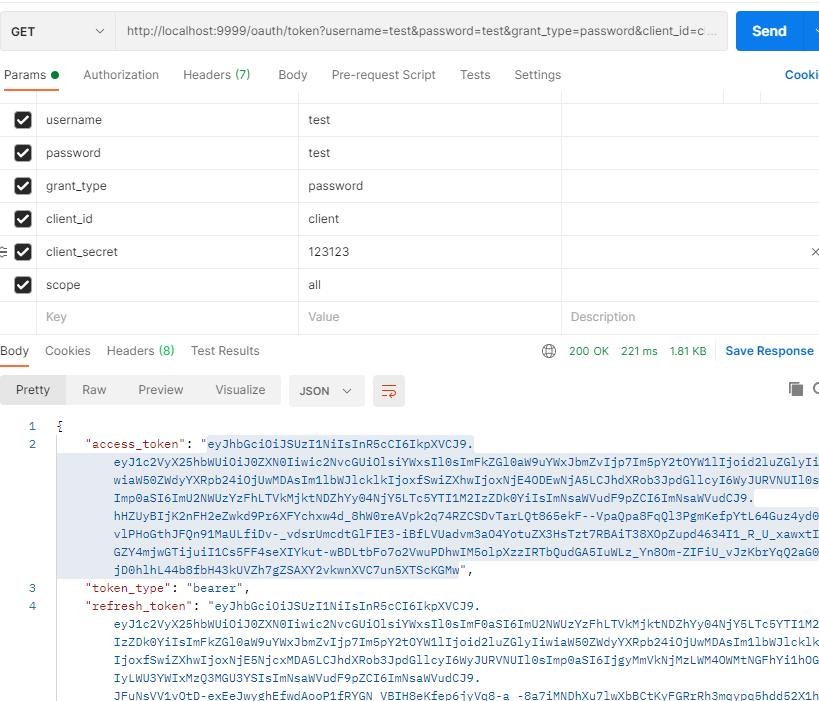
检查

.allowedTokenEndpointRequestMethods(HttpMethod.GET,HttpMethod.POST); //

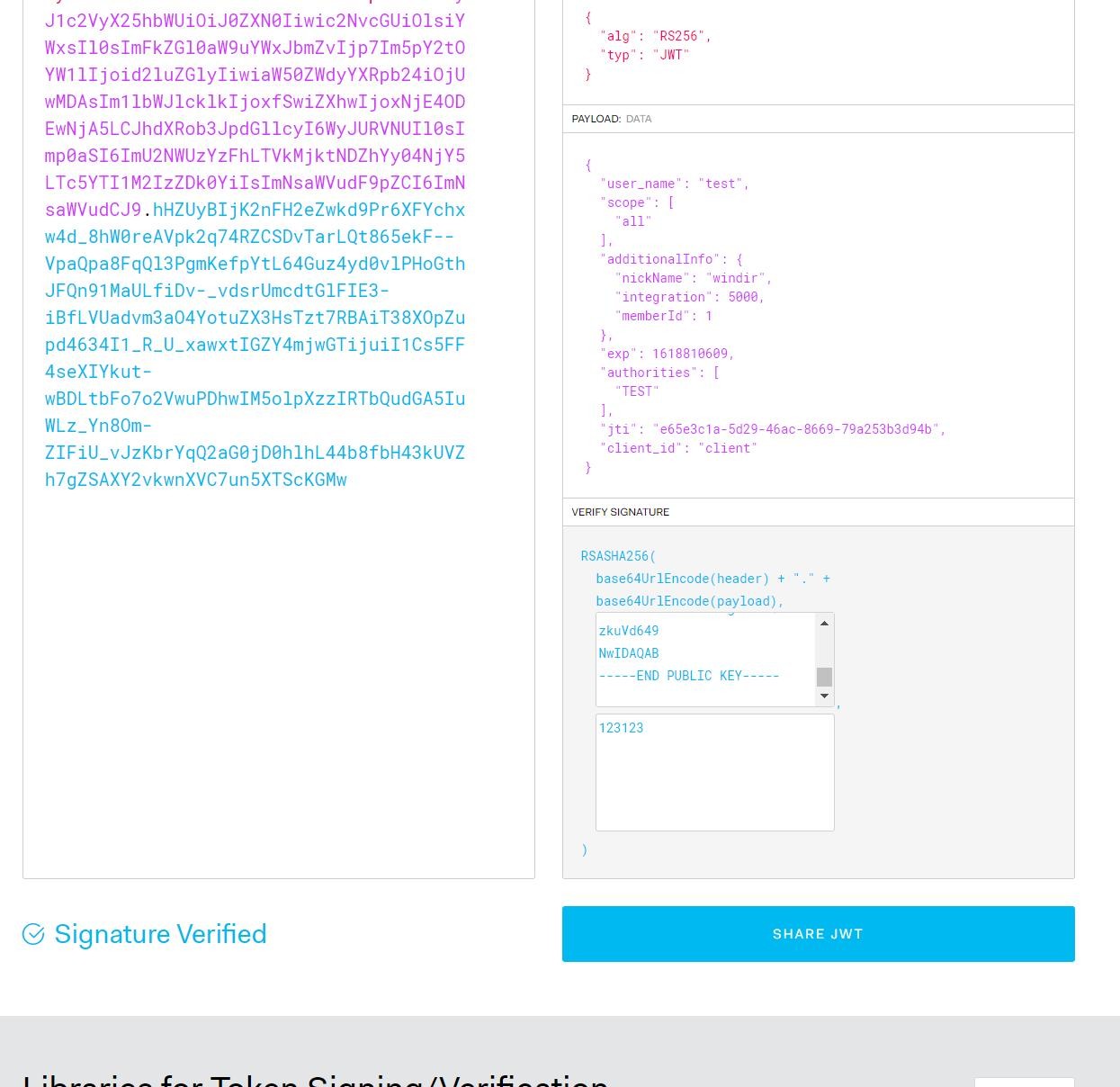
支持 GET,POST 请求

}

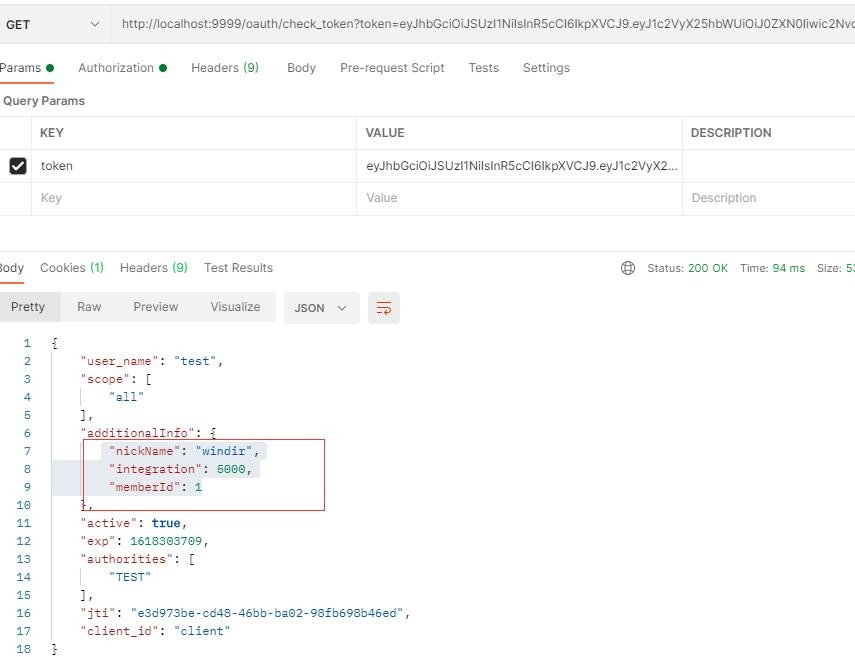
1. 通过密码模式测试获取 token



<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。



1. 测试校验 token



# 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

server:

port: 9999 spring:

application:

name: tulingmall-gateway #配置 nacos 注册中心地址 cloud:

nacos:

discovery:

server-addr: 192.168.65.232:8848 #注册中心地址

namespace: 80a98d11-492c-4008-85aa-32d889e9b0d0 #环境隔离

gateway: routes:

- id: tulingmall-member #路由 ID，全局唯一 uri: lb://tulingmall-member

predicates:

- Path=/member/\*\*,/sso/\*\*

* id: tulingmall-promotion

uri: lb://tulingmall-promotion predicates:

* + Path=/coupon/\*\*
* id: tulingmall-authcenter

uri: lb://tulingmall-authcenter predicates:

* + Path=/oauth/\*\*

1. 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.过滤不需要认证的 url,比如/oauth/\*\*

//2. 获取 token

// 从请求头中解析 Authorization value: bearer xxxxxxx

// 或者从请求参数中解析 access\_token

//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

//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。

/\*\*

* @author Fox

\*

* 认证过滤器: 实现认证逻辑

\*/ @Component @Order(0)

@EnableConfigurationProperties(value = NotAuthUrlProperties.class) public class AuthenticationFilter implements GlobalFilter, InitializingBean {

/\*\*

\* 请求各个微服务 不需要用户认证的 URL

\*/ @Autowired

private NotAuthUrlProperties notAuthUrlProperties;

@Override

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

String currentUrl = exchange.getRequest().getURI().getPath();

//过滤不需要认证的 url if(shouldSkip(currentUrl)) {

//log.info("跳过认证的 URL:{}",currentUrl); return chain.filter(exchange);

}

//log.info("需要认证的 URL:{}",currentUrl);

return chain.filter(exchange);

}

@Override

public void afterPropertiesSet() throws Exception {

//获取公钥 TODO

}

/\*\*

* 方法实现说明:不需要授权的路径
* @author:smlz
* @param currentUrl 当前请求路径
* @return:
* @exception:
* @date:2019/12/26 13:49

\*/

private boolean shouldSkip(String currentUrl) {

//路径匹配器(简介 SpringMvc 拦截器的匹配器)

//比如/oauth/\*\* 可以匹配/oauth/token /oauth/check\_token 等

PathMatcher pathMatcher = new AntPathMatcher();

for(String skipPath:notAuthUrlProperties.getShouldSkipUrls()) { if(pathMatcher.match(skipPath,currentUrl)) {

return true;

}

}

return false;

}

}

@Data @ConfigurationProperties("tuling.gateway") public class NotAuthUrlProperties {

private LinkedHashSet<String> shouldSkipUrls;

}

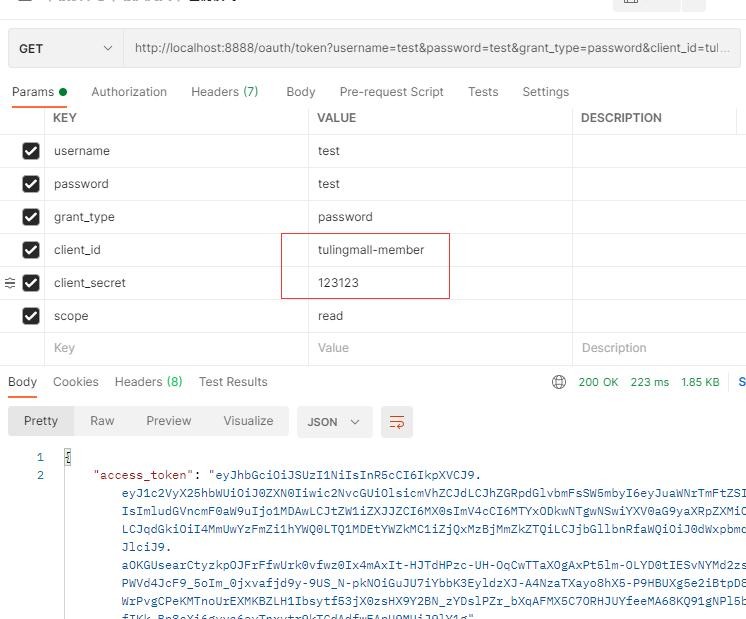
//application.yml tuling:

gateway: shouldSkipUrls:

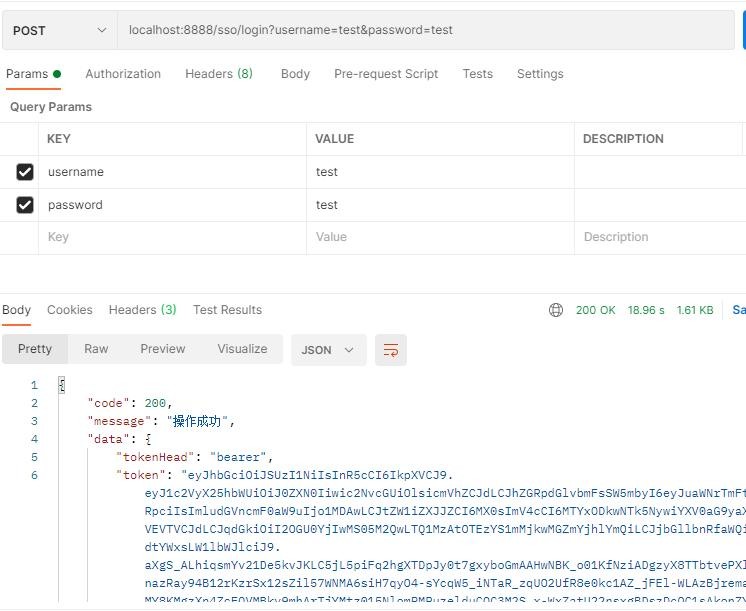
- /oauth/\*\*

- /sso/\*\*

测试： 密码模式 client\_id 为会员微服务，能够获取到 token 信息



测试： 会员微服务会员登录逻辑



3） 解析请求，获取 token

从请求头中解析 Authorization value: bearer xxxxxxx 或者 从请求参数中解析

access\_token

在 AuthenticationFilter#filter 中实现获取 token 的逻辑

//2. 获取 token

// 从请求头中解析 Authorization value: bearer xxxxxxx

// 或者从请求参数中解析 access\_token

//第一步:解析出我们 Authorization 的请求头 value 为: “bearer XXXXXXXXXXXXXX” String authHeader = exchange.getRequest().getHeaders().getFirst("Authorization");

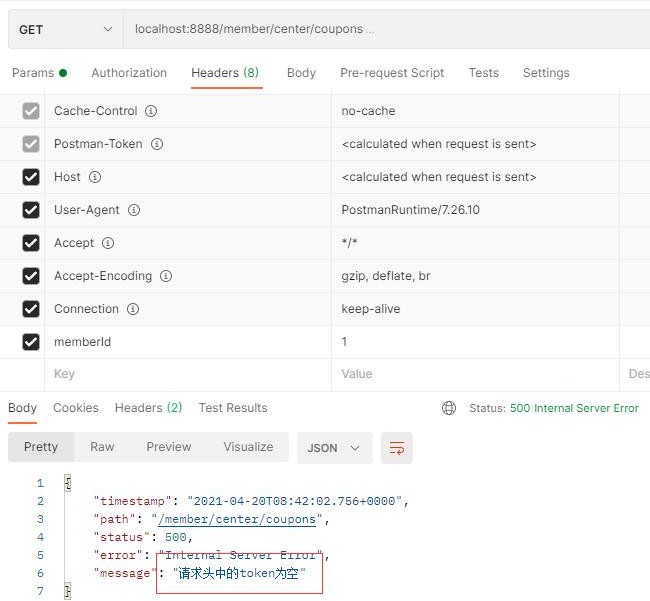
//第二步:判断 Authorization 的请求头是否为空

if(StringUtils.isEmpty(authHeader)) { log.warn("需要认证的 url,请求头为空");

throw new GateWayException(ResultCode.AUTHORIZATION\_HEADER\_IS\_EMPTY);

}

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



1. 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

<!--添加 jwt 相关的包-->

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.10.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.10.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.10.5</version>

<scope>runtime</scope>

</dependency>

在 AuthenticationFilter#filter 中实现校验 token 的逻辑

//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

//第三步 校验我们的 jwt 若 jwt 不对或者超时都会抛出异常

Claims claims = JwtUtils.validateJwtToken(authHeader,publicKey);

校验 token 逻辑

// AuthenticationFilter.java

/\*\*

\* 请求头中的 token 的开始

\*/

private static final String AUTH\_HEADER = "bearer ";

public static Claims validateJwtToken(String authHeader,PublicKey publicKey) {

String token =null ; try{

token = StringUtils.substringAfter(authHeader, AUTH\_HEADER);

Jwt<JwsHeader,

Claims>

parseClaimsJwt

=

Jwts.parser().setSigningKey(publicKey).parseClaimsJws(token);

Claims claims = parseClaimsJwt.getBody();

//log.info("claims:{}",claims);

return claims;

}catch(Exception e){

log.error("校验 token 异常:{},异常信息:{}",token,e.getMessage());

throw new GateWayException(ResultCode.JWT\_TOKEN\_EXPIRE);

}

}

工具类

@Slf4j

public class JwtUtils {

/\*\*

\* 认证服务器许可我们的网关的 clientId(需要在 oauth\_client\_details 表中配置)

\*/

private static final String CLIENT\_ID = "tulingmall-gateway";

/\*\*

\* 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)

\*/

private static final String CLIENT\_SECRET = "123123";

/\*\*

\* 认证服务器暴露的获取 token\_key 的地址

\*/

private static final String AUTH\_TOKEN\_KEY\_URL = "[http://tulingmall-](http://tulingmall-/) auth/oauth/token\_key";

/\*\*

\* 请求头中的 token 的开始

\*/

private static final String AUTH\_HEADER = "bearer ";

/\*\*

* 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
* @author:smlz
* @param restTemplate 远程调用的操作类
* @return: tokenKey 解析 jwt 的 tokenKey
* @exception:
* @date:2020/1/22 11:31

\*/

private static String getTokenKeyByRemoteCall(RestTemplate restTemplate) throws GateWayException {

//第一步:封装请求头

HttpHeaders headers = new HttpHeaders(); headers.setContentType(MediaType.APPLICATION\_FORM\_URLENCODED); headers.setBasicAuth(CLIENT\_ID,CLIENT\_SECRET); HttpEntity<MultiValueMap<String, String>> entity = new HttpEntity<>(null, headers);

//第二步:远程调用获取 token\_key try {

ResponseEntity<Map> response = restTemplate.exchange(AUTH\_TOKEN\_KEY\_URL, HttpMethod.GET, entity, Map.class);

String tokenKey = response.getBody().get("value").toString();

log.info("去认证服务器获取 Token\_Key:{}",tokenKey);

return tokenKey;

}catch (Exception e) {

log.error("远程调用认证服务器获取 Token\_Key 失败:{}",e.getMessage());

throw new GateWayException(ResultCode.GET\_TOKEN\_KEY\_ERROR);

}

}

/\*\*

* 方法实现说明:生成公钥
* @author:smlz
* @param restTemplate:远程调用操作类
* @return: PublicKey 公钥对象
* @exception:
* @date:2020/1/22 11:52

\*/

public static PublicKey genPulicKey(RestTemplate restTemplate) throws GateWayException

{

String tokenKey = getTokenKeyByRemoteCall(restTemplate);

try{

//把获取的公钥开头和结尾替换掉

String dealTokenKey =tokenKey.replaceAll("\\-\*BEGIN PUBLIC KEY\\-\*", "").replaceAll("\\-\*END PUBLIC KEY\\-\*", "").trim();

java.security.Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());

X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(Base64.decodeBase64(dealTokenKey));

KeyFactory keyFactory = KeyFactory.getInstance("RSA");

PublicKey publicKey = keyFactory.generatePublic(pubKeySpec);

log.info("生成公钥:{}",publicKey);

return publicKey;

}catch (Exception e) {

log.info("生成公钥异常:{}",e.getMessage());

throw new GateWayException(ResultCode.GEN\_PUBLIC\_KEY\_ERROR);

}

}

public static Claims validateJwtToken(String authHeader,PublicKey publicKey) {

String token =null ; try{

token = StringUtils.substringAfter(authHeader, AUTH\_HEADER);

Jwt<JwsHeader,

Claims>

parseClaimsJwt

=

Jwts.parser().setSigningKey(publicKey).parseClaimsJws(token);

Claims claims = parseClaimsJwt.getBody();

//log.info("claims:{}",claims);

return claims;

}catch(Exception e){

log.error("校验 token 异常:{},异常信息:{}",token,e.getMessage());

throw new GateWayException(ResultCode.JWT\_TOKEN\_EXPIRE);

}

}

}

需要从 tulingmall-authcenter 获取公钥，实现公钥获取逻辑

// AuthenticationFilter.java

/\*\*

\* jwt 的公钥,需要网关启动,远程调用认证中心去获取公钥

\*/

private PublicKey publicKey;

@Autowired

private RestTemplate restTemplate;

@Override

public void afterPropertiesSet() throws Exception {

//获取公钥 TODO

this.publicKey = JwtUtils.genPulicKey(restTemplate);

}

@Configuration

public class RibbonConfig {

@Autowired

private LoadBalancerClient loadBalancer;

@Bean

public RestTemplate restTemplate(){

RestTemplate restTemplate = new RestTemplate(); restTemplate.setInterceptors(

Collections.singletonList(

new LoadBalancerInterceptor(loadBalancer)));

return restTemplate;

}

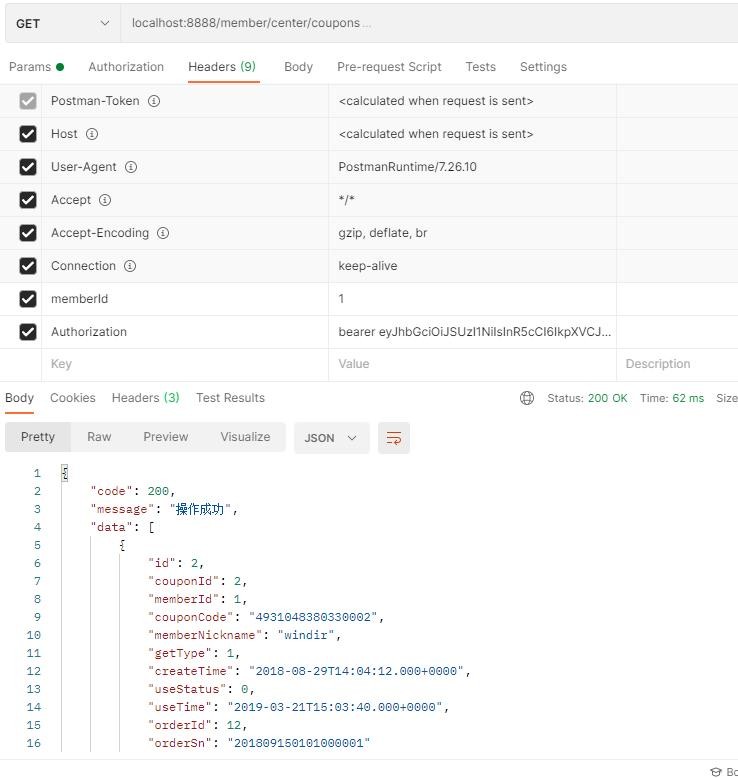
}

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

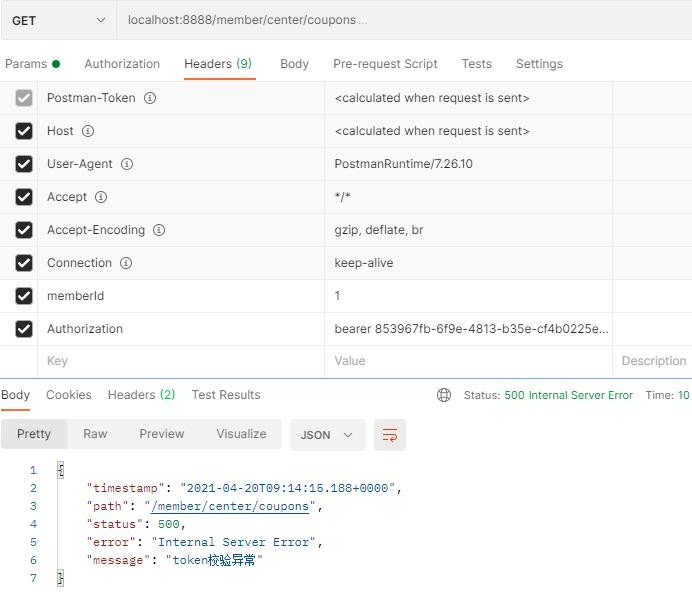
源码参考：

org.springframework.cloud.client.loadbalancer.LoadBalancerAutoConfiguration org.springframework.beans.factory.support.DefaultListableBeanFactory#preInstantiateSingleton s

测试： 正确的 token，通过网关获取用户优惠券信息



错误的 token，抛出异常



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

//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

//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

ServerWebExchange webExchange = wrapHeader(exchange,claims);

解析用户登录信息存储到请求头中

// AuthenticationFilter.java

private ServerWebExchange wrapHeader(ServerWebExchange serverWebExchange,Claims claims) {

String loginUserInfo = JSON.toJSONString(claims);

//log.info("jwt 的用户信息:{}",loginUserInfo);

String memberId = claims.get("additionalInfo", Map.class).get("memberId").toString(); String nickName = claims.get("additionalInfo",Map.class).get("nickName").toString();

//向 headers 中放文件，记得 build

ServerHttpRequest request = serverWebExchange.getRequest().mutate()

.header("username",claims.get("user\_name",String.class))

.header("memberId",memberId)

.header("nickName",nickName)

.build();

//将现在的 request 变成 change 对象

return serverWebExchange.mutate().request(request).build();

}