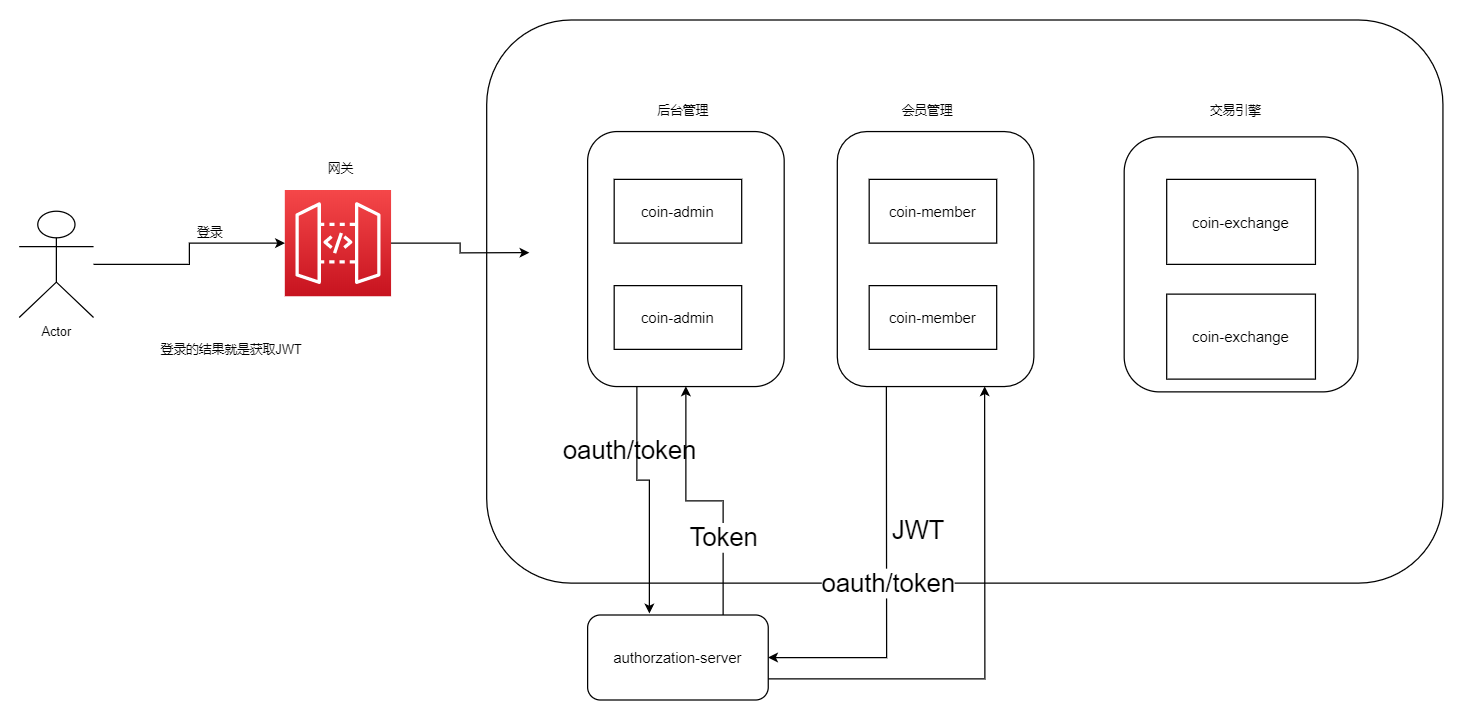
# 授权服务器的搭建

# 授权服务器的定位

一言而概之：就是为客户端产生一个Token

如图：



# 授权服务器的实现

## 2.1 添加依赖

|  |
| --- |
| *<!-- 服务发现-->* <dependency>  <groupId>com.alibaba.cloud</groupId>  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>  </dependency>   <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-oauth2</artifactId>  </dependency> |

## 2.2 配置文件

|  |
| --- |
| spring:  application:  name: authorization-server  cloud:  nacos:  discovery:  server-addr: nacos-server:8848 server:  port: 9999 |

## 2.3 启动类

|  |
| --- |
| @SpringBootApplication public class AuthorizationApplication {   public static void main(String[] args) {  SpringApplication.*run*(AuthorizationApplication.class, args);  } } |

## 2.4配置类

### 2.4.1 授权服务器的配置

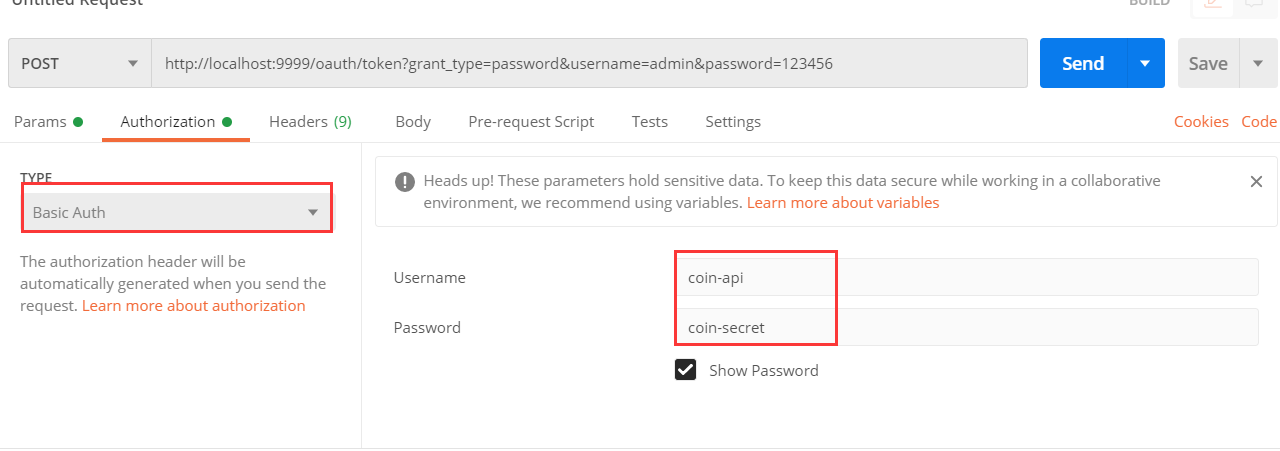
|  |
| --- |
| @EnableAuthorizationServer @Configuration public class AuthorizationConfig extends AuthorizationServerConfigurerAdapter {   @Autowired  public PasswordEncoder passwordEncoder ;   @Autowired  private AuthenticationManager authenticationManager ;   @Autowired  private UserDetailsService userDetailsService ;  */\*\*  \* 配置第三方客户端  \*/* @Override  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  clients.inMemory()  .withClient("coin-api")  .secret(passwordEncoder.encode("coin-secret"))  .scopes("all")  .authorizedGrantTypes("password","refresh")  .accessTokenValiditySeconds(24 \* 7200)  .refreshTokenValiditySeconds(7 \* 24 \* 7200) ;  }   */\*\*  \* 设置授权管理器和UserDetailsService  \* @param endpoints  \* @throws Exception  \*/* @Override  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  endpoints.tokenStore(new InMemoryTokenStore())  .authenticationManager(authenticationManager)  .userDetailsService(userDetailsService) ;  } } |

### 2.4.2 Web 安全的配置

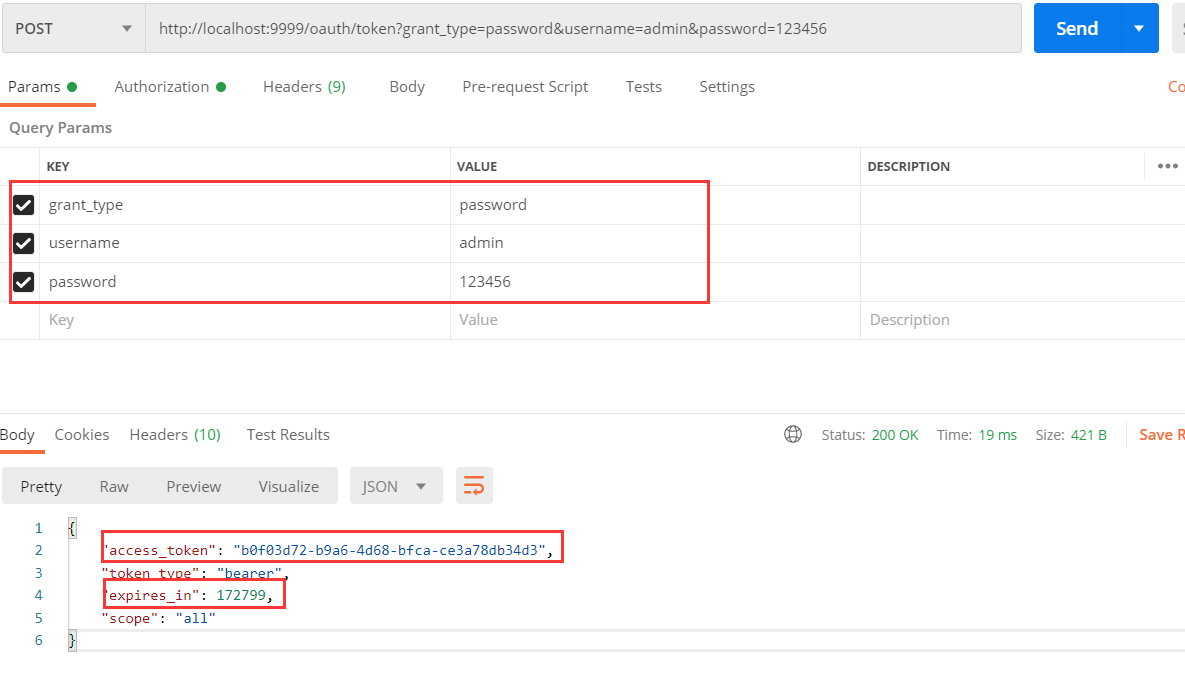
|  |
| --- |
| @Configuration public class WebSecurityConfig extends WebSecurityConfigurerAdapter {   */\*\*  \* 注入一个验证管理器  \*  \* @return  \* @throws Exception  \*/* @Bean  public AuthenticationManager authenticationManagerBean() throws Exception {  return super.authenticationManagerBean();  }   */\*\*  \* 资源的放行  \*/* @Override  protected void configure(HttpSecurity http) throws Exception {  http.csrf().disable(); *// 关闭scrf* http.authorizeRequests().anyRequest().authenticated();  }    */\*\*  \* 创建一个测试的UserDetail  \* @return  \*/* @Bean  public UserDetailsService userDetailsService() {  InMemoryUserDetailsManager inMemoryUserDetailsManager = new InMemoryUserDetailsManager();  User user = new User("admin", "123456", Arrays.*asList*(new SimpleGrantedAuthority("ROLE\_ADMIN"))) ;  inMemoryUserDetailsManager.createUser(user);  return inMemoryUserDetailsManager;  }   */\*\*  \* 注入密码的验证管理器  \* @return  \*/* @Bean  public PasswordEncoder passwordEncoder() {  return NoOpPasswordEncoder.*getInstance*();  } } |

## 2.5 获取token测试

第一步：



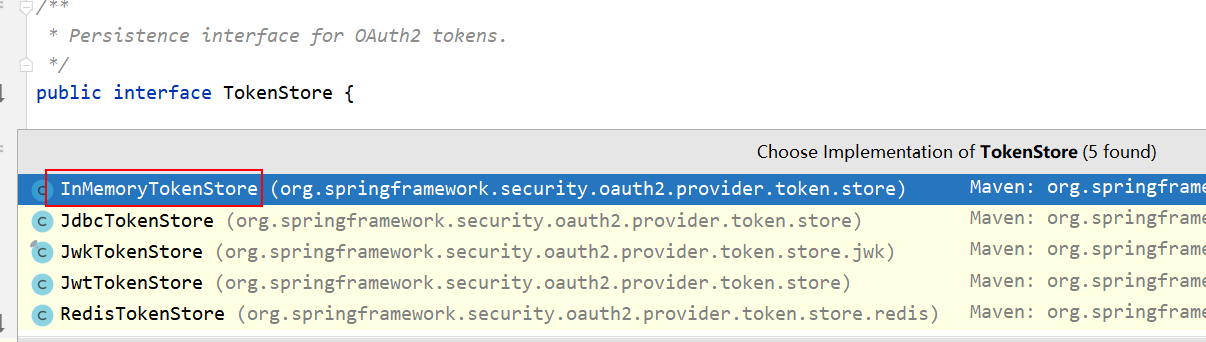
第二步：



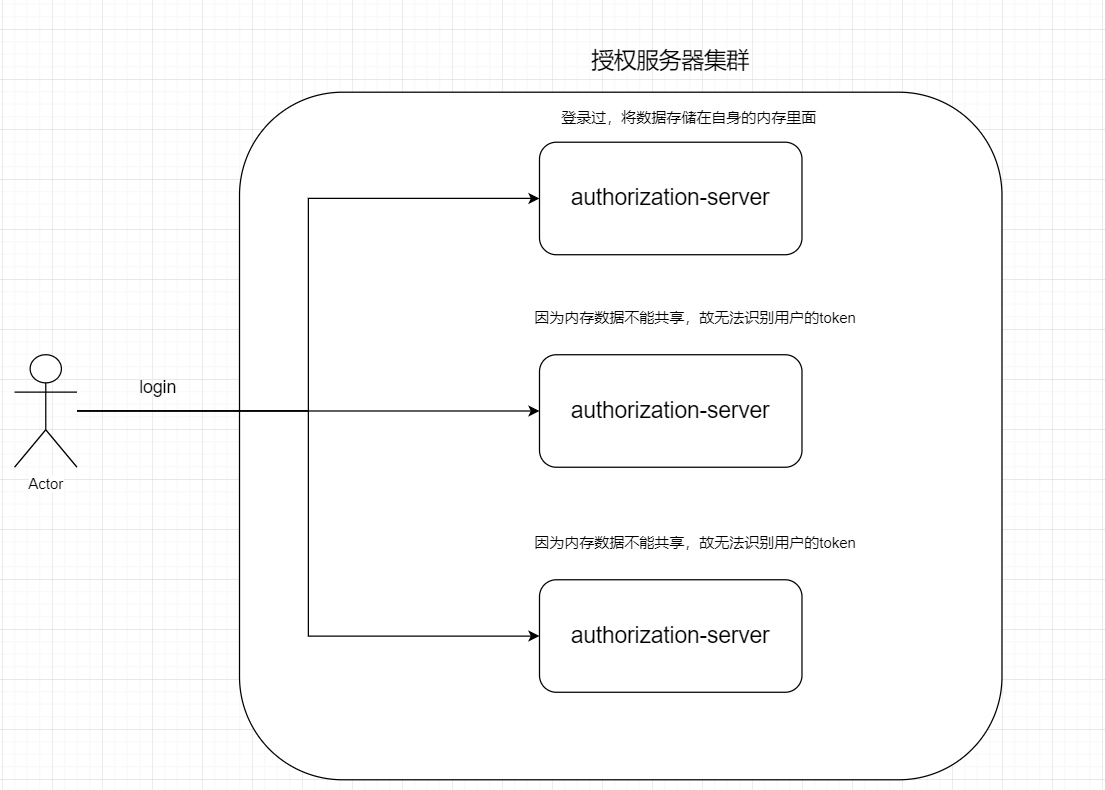
# JWT接入

## 3.1 Token共享问题

我的token 目前存储在内存里面：

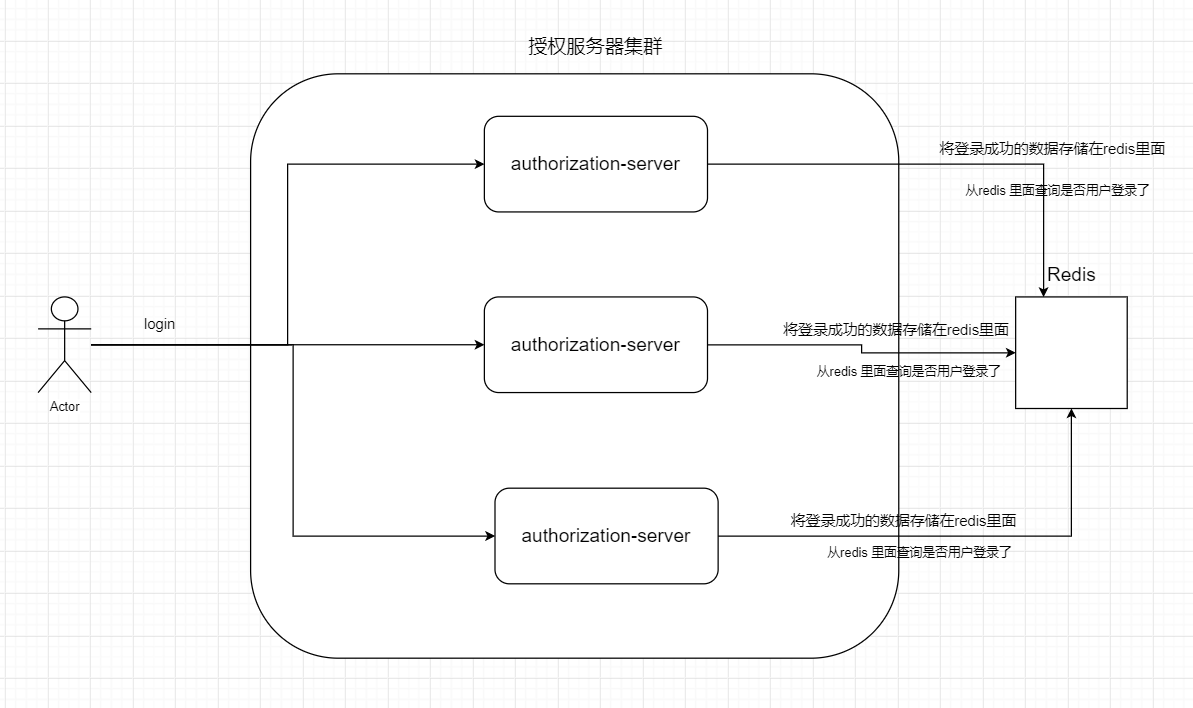


也就是说，当我们仅仅只有一台authorization-server 时，没有任何问题，但是当我们使用多台authorization-server时，由于内存数据无法共享，故用户登录的数据仅仅保存在一台服务器里面，这就会导致某台授权服务器会误判“是否用户登录”这个问题。



## 3.2 使用Redis 共享Token

将之前数据存储在内存里面的问题解决掉，现在直接把token 存储在内存里面：



### 3.2.1 添加依赖

|  |
| --- |
| *<!-- redis-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  </dependency> |

### 3.2.2 添加配置文件

|  |
| --- |
| server:  port: 9999 spring:  application:  name: authorization-server  cloud:  nacos:  discovery:  server-addr: nacos-server:8848  redis:  host: redis-server  port: 6380  password: Ltd3411?? |

注意：redis-server 要事先在host文件里面配置。

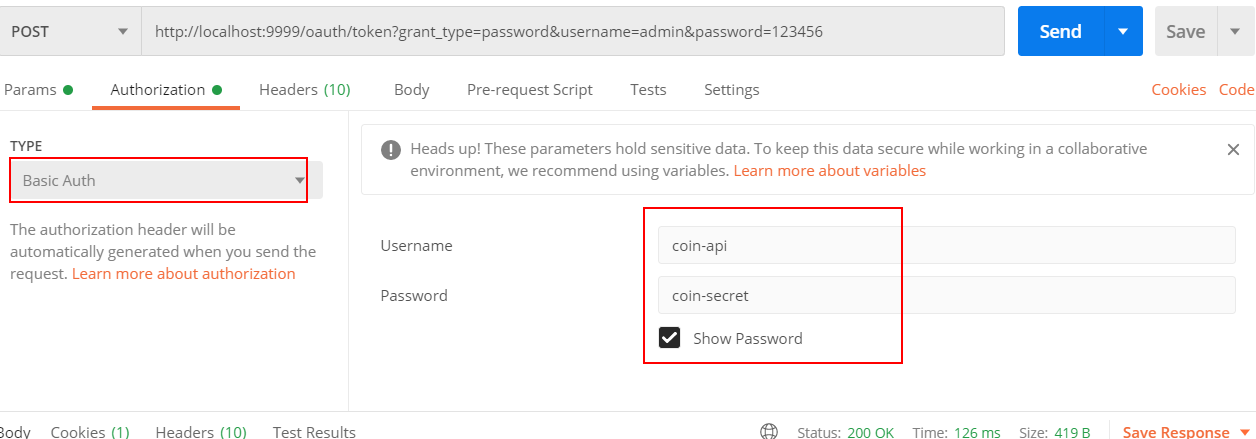
### 3.2.3 使用RedisTokenStore

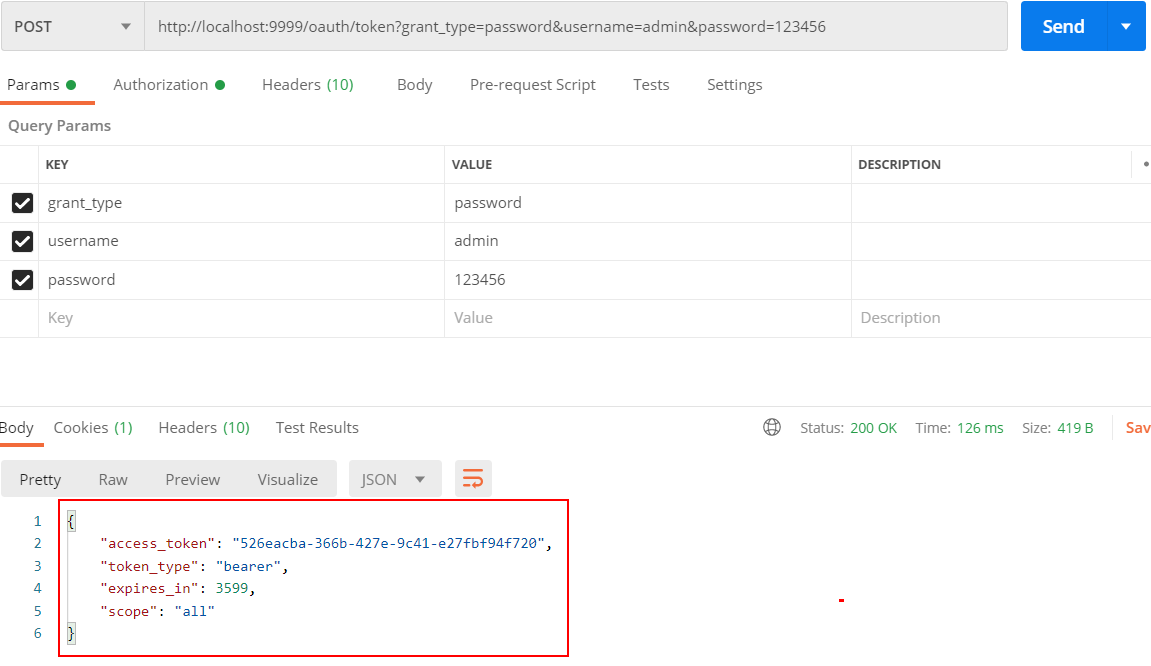
修改我们之前的AuthorizationServerConfig配置类：

|  |
| --- |
| @EnableAuthorizationServer *// 开启授权服务器的功能* @Configuration public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {   @Autowired  private PasswordEncoder passwordEncoder ;   @Autowired  private AuthenticationManager authenticationManager ;   @Autowired  private UserDetailsService userDetailsService ;   **@Autowired  private RedisConnectionFactory redisConnectionFactory ;**    */\*\*  \* 添加第三方的客户端  \*/* @Override  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  clients.inMemory()  .withClient("coin-api") *// 第三方客户端的名称* .secret(passwordEncoder.encode("coin-secret")) *// 第三方客户端的密钥* .scopes("all") *//第三方客户端的授权范围* .accessTokenValiditySeconds(3600) *// token的有效期* .refreshTokenValiditySeconds(7\*3600);*// refresh\_token的有效期* super.configure(clients);  }   */\*\*  \* 配置验证管理器，UserdetailService  \*/* @Override  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  endpoints.authenticationManager(authenticationManager)  .userDetailsService(userDetailsService)  **.tokenStore(redisTokenStore());**  super.configure(endpoints);  }    **public TokenStore redisTokenStore(){  return new RedisTokenStore(redisConnectionFactory) ;  }** } |

### 3.2.4 获取Token 测试

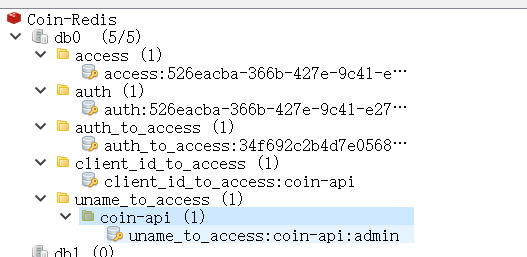
重启authorization-server，获取token测试



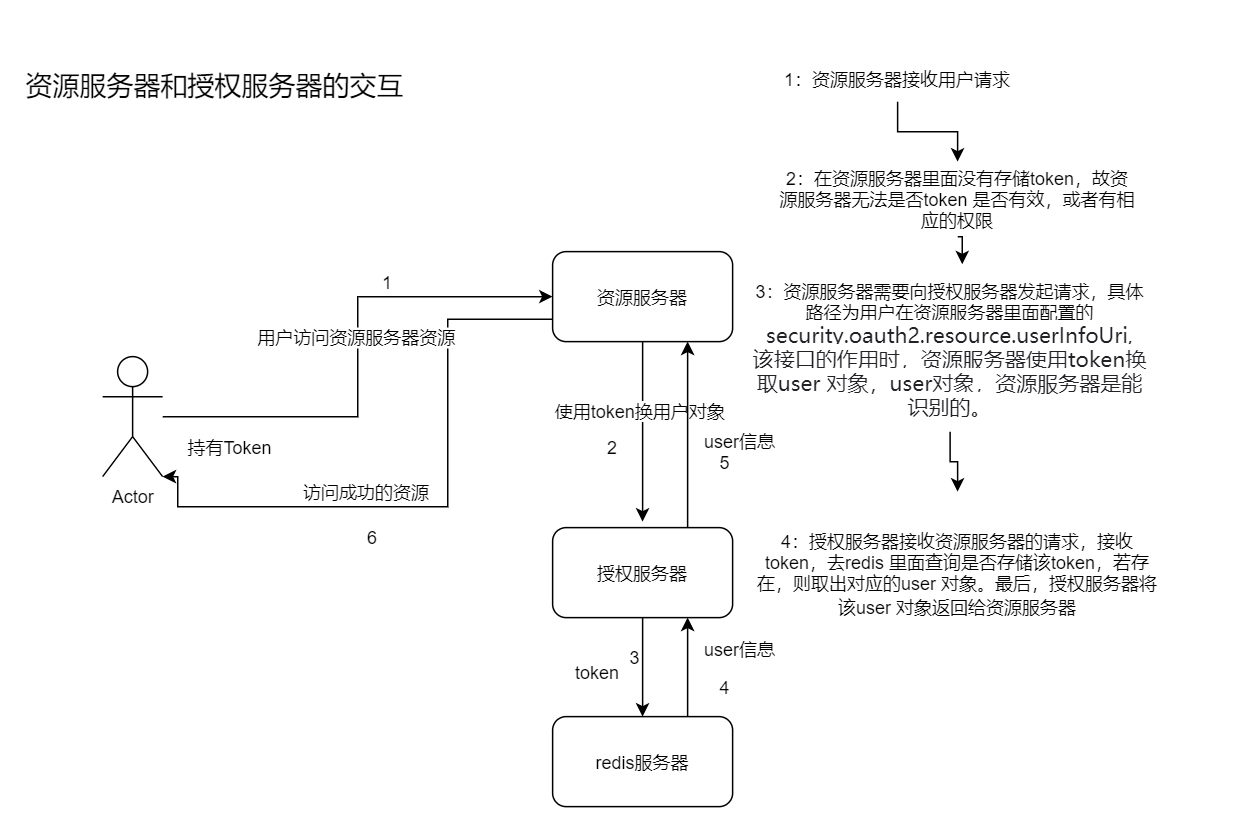


### 3.2.5 观察Redis数据

我们发现，在redis 里面已经保存了用户登录的数据了。



## 3.3 资源服务器和授权服务的交互



### 3.3.1 在授权服务器里面准备userinfo的接口

|  |
| --- |
| @RestController public class UserInfoController {   */\*\*  \* 获取该用户的对象  \* @param principal  \* @return  \*/* @GetMapping("/user/info")  public Principal usrInfo(Principal principal){ *// 此处的principal 由OAuth2.0 框架自动注入  // 原理就是：利用Context概念，将授权用户放在线程里面，利用ThreadLocal来获取当前的用户对象  // Authentication authentication = SecurityContextHolder.getContext().getAuthentication();* return principal ;  } } |

### 3.3.2 将该授权服务器变成资源服务器

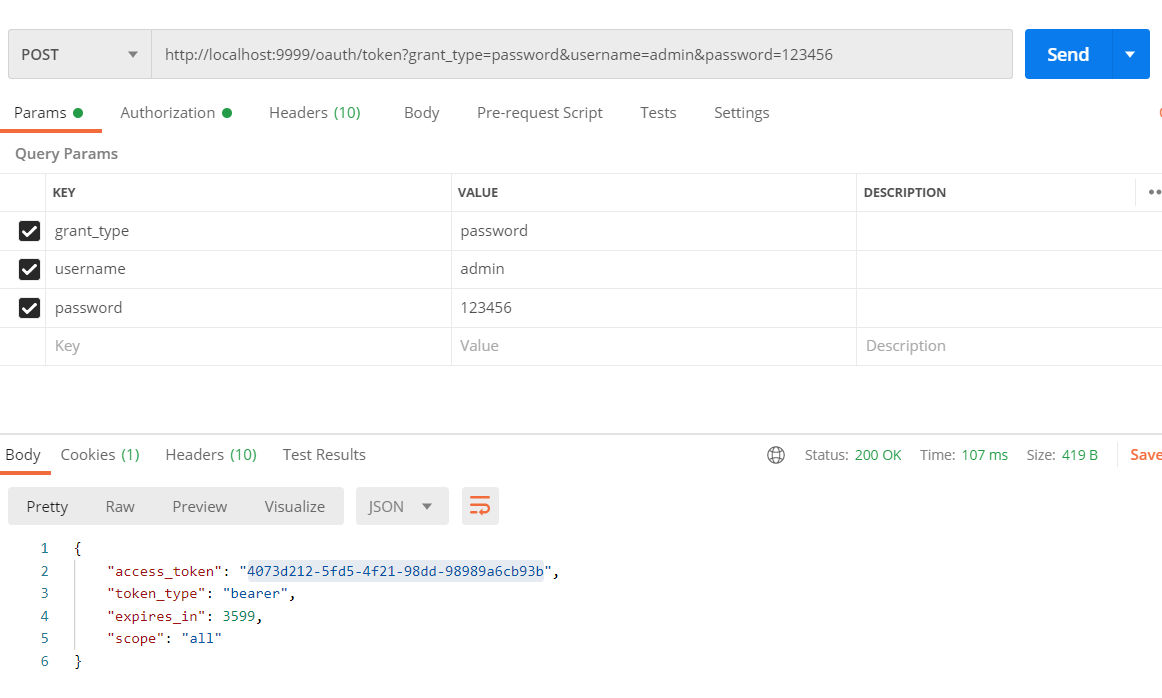
因为授权服务器里面提供了userinfo 该资源，所以我们也将它认为是授权服务器。

添加配置类就可以了。

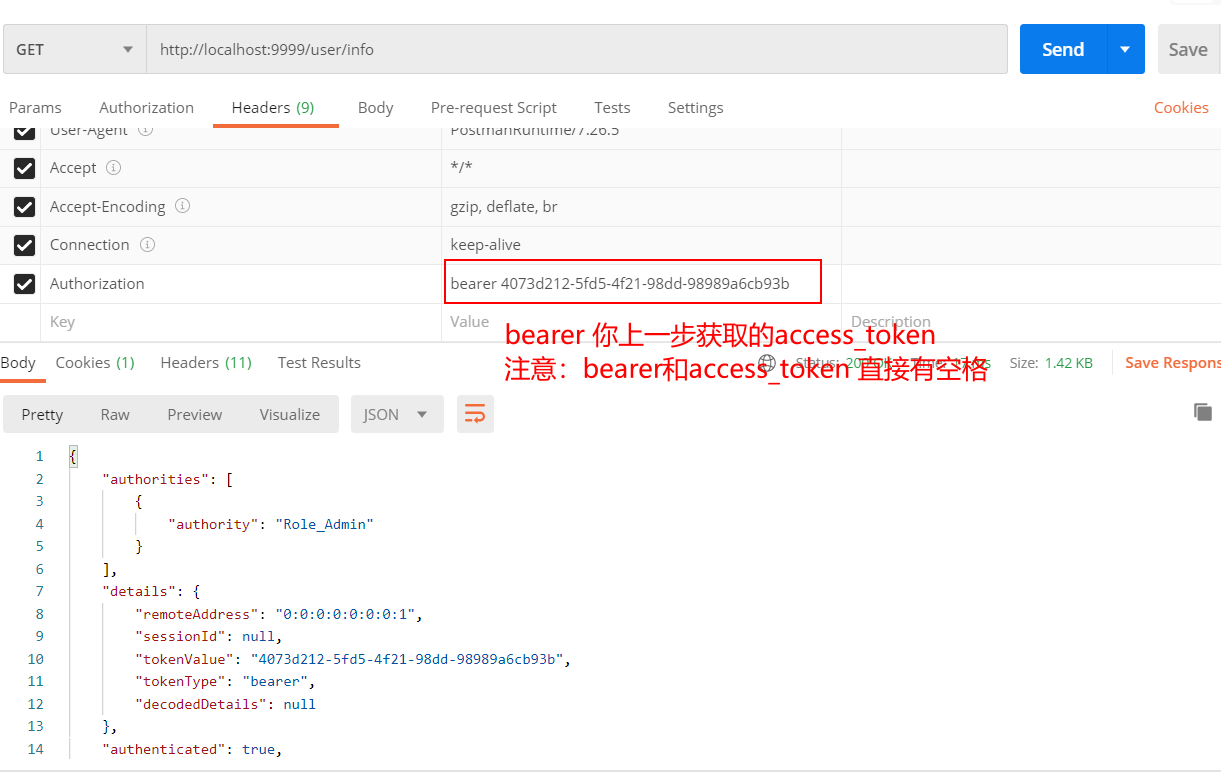
|  |
| --- |
| @EnableResourceServer @Configuration public class ResourceServerConfig extends ResourceServerConfigurerAdapter { } |

### 3.3.2使用token 换取user对象

第一步：获取一个token



第二步：使用Token 换用户对象



我们可以看见，user 对象已经被获取到了。

### 3.3.3 存在的问题

我们在理解资源服务器和授权服务的交互后会发现，授权服务器有巨大的压力：当用户访问每一个受保护的资源时（无论该资源散落在那个微服务里面），资源服务器都要和授权服务器交互一次，这样，授权服务器的压力将非常的大。我们必须找解决方案。

* 方案一：使用负载均衡的概念，多部署几台授权服务器
* 方案二：让资源服务器不再访问授权服务器

## 3.4 使用Jwt来做token的存储

上面的方案里面，我们提到了让资源服务器不再访问授权服务器，那会存在什么问题呢？

资源服务器访问授权服务的本质在于2点：

第一点：资源服务器无法验证token的正确性，因为它没有存储token

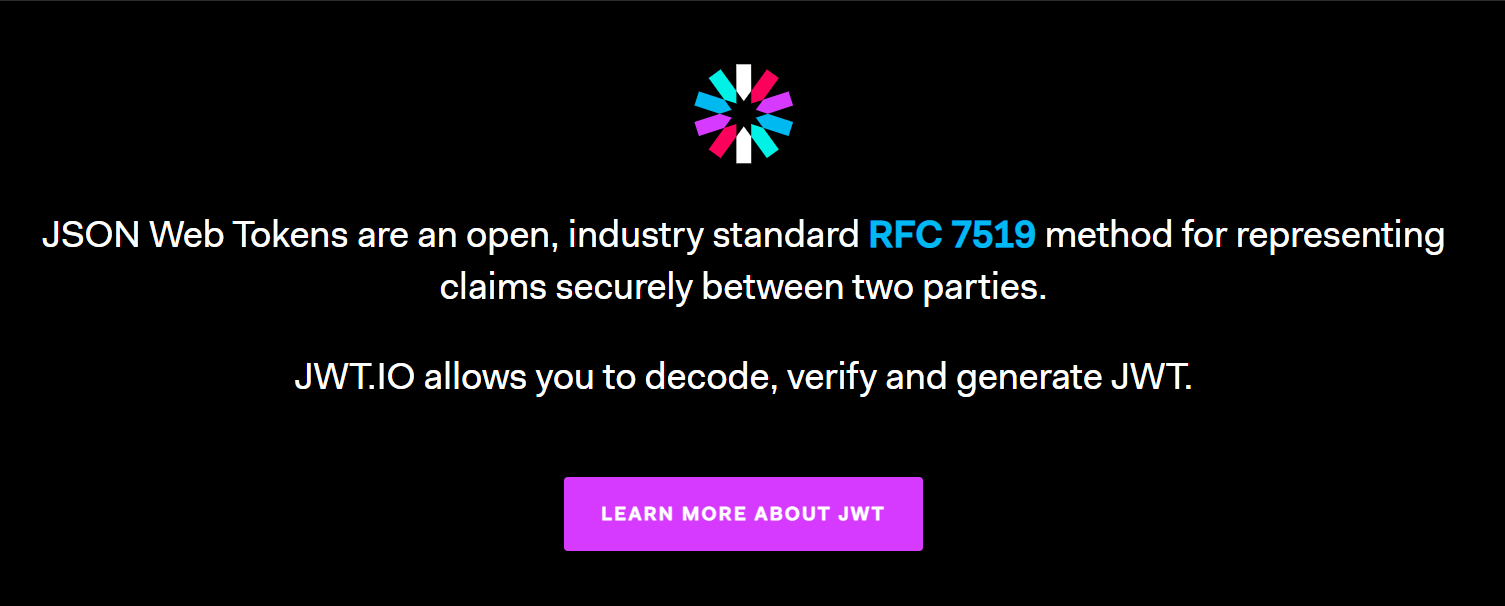
第二点：资源服务要通过授权服务器来换取用户（token 换 user）。

我们来推演：资源服务器当前只能得到用户给他的token，我们能做的改造有限：

第一步：若我们将**用户的基本信息存储在token** 里面呢？

第二步：定义一种加密规则，让资源服务器也能去判断该token的正确性。

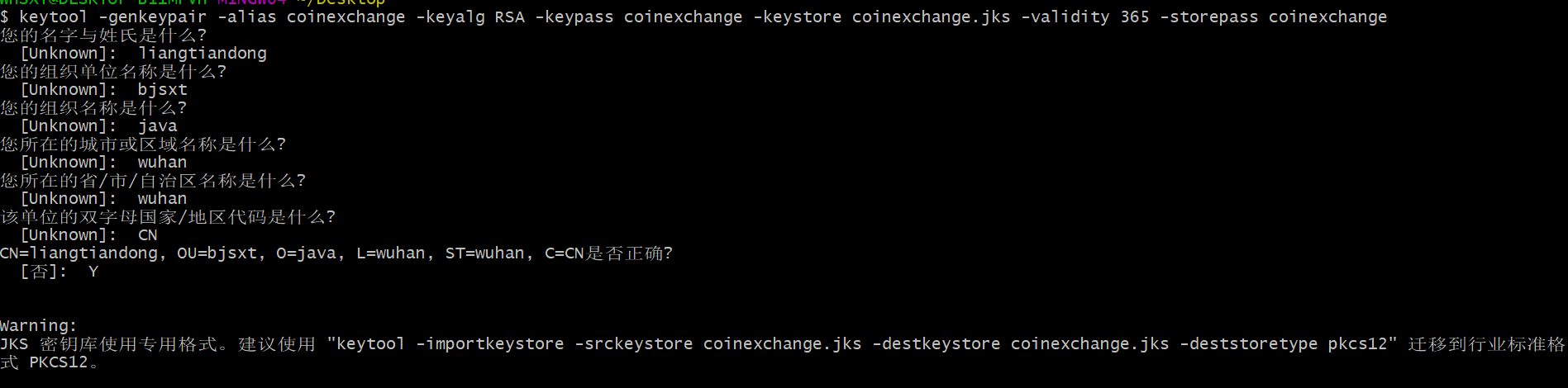
这样，我们的JWT就上场了。看看JWT的定义：



### 3.4.1 生成私钥和公钥

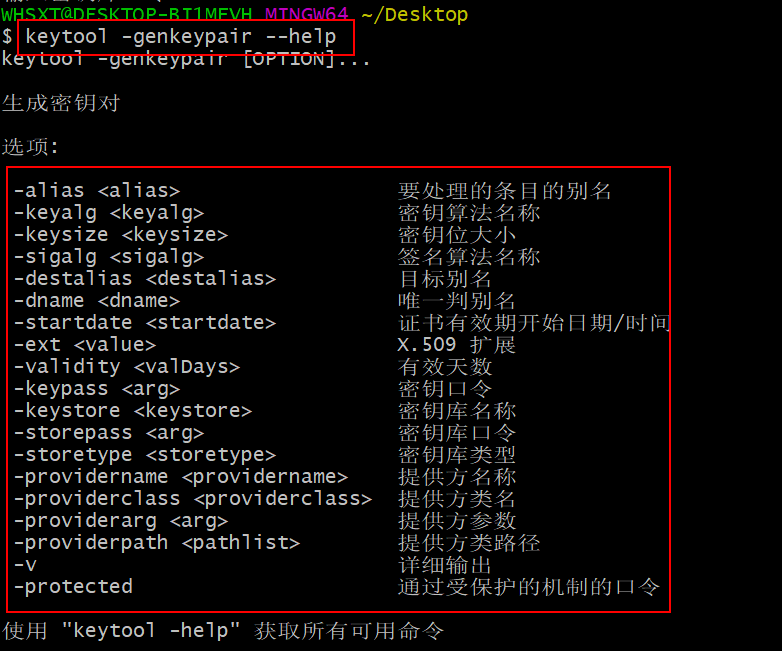
#### 生成私钥：

|  |
| --- |
| keytool -genkeypair -alias coinexchange -keyalg RSA -keypass coinexchange -keystore coinexchange.jks -validity 365 -storepass coinexchange |

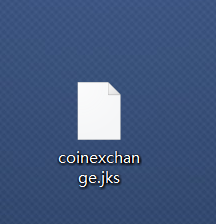


具体命令和参数：

Keytool 是一个java提供的证书管理工具



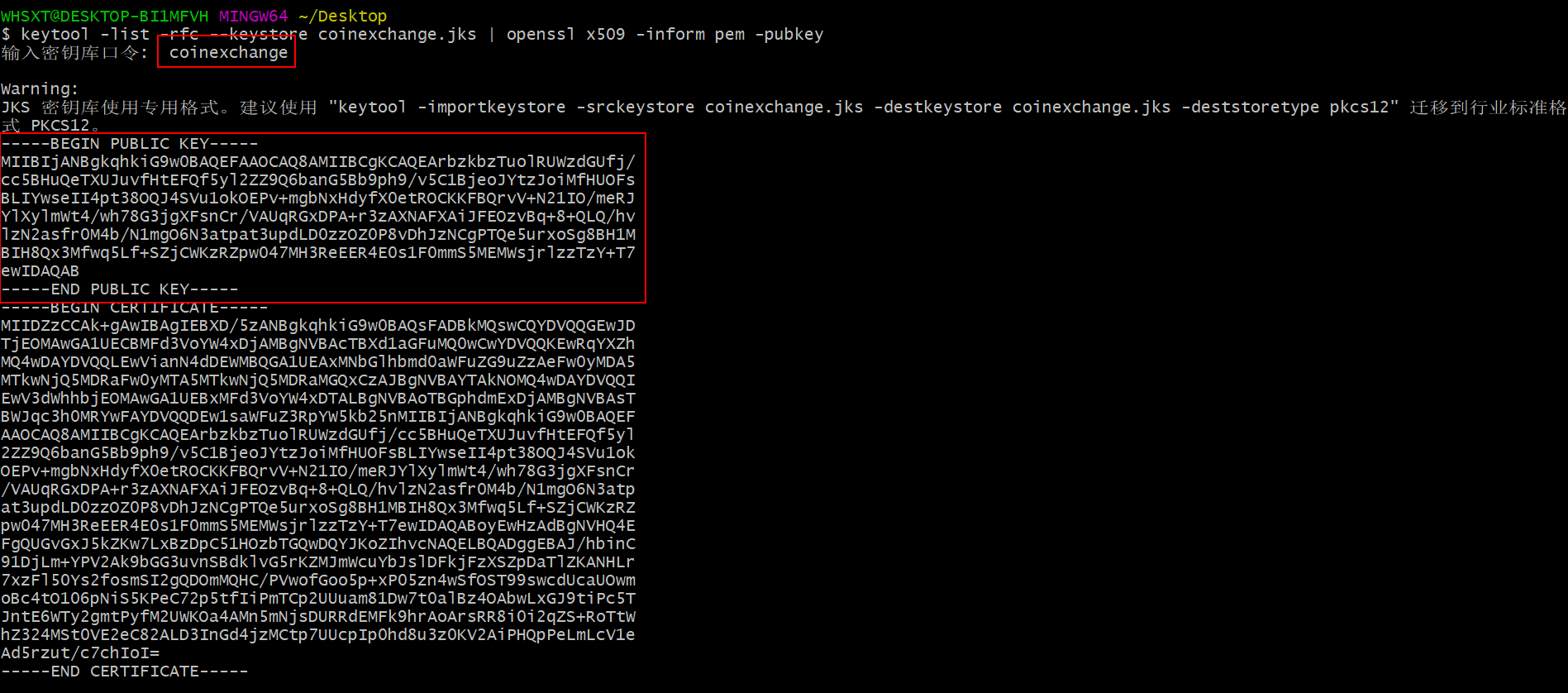
现在，桌面上已经有一个jks 文件了。



该文件里面保存的就是私钥信息。

#### 解析公钥：

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



将解析出来的公钥放在一个文件的文件里面:

|  |
| --- |
| -----BEGIN PUBLIC KEY-----  MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEArbzkbzTuolRUWzdGUfj/  cc5BHuQeTXUJuvfHtEFQf5yl2ZZ9Q6banG5Bb9ph9/v5C1BjeoJYtzJoiMfHUOFs  BLIYwseII4pt38OQJ4SVu1okOEPv+mgbNxHdyfX0etROCKKFBQrvV+N21IO/meRJ  YlXylmWt4/wh78G3jgXFsnCr/VAUqRGxDPA+r3zAXNAFXAiJFEOzvBq+8+QLQ/hv  lzN2asfr0M4b/N1mgO6N3atpat3updLD0zzOZ0P8vDhJzNCgPTQe5urxoSg8BH1M  BIH8Qx3Mfwq5Lf+SZjCWKzRZpw047MH3ReEER4E0s1F0mmS5MEMWsjrlzzTzY+T7  ewIDAQAB  -----END PUBLIC KEY----- |



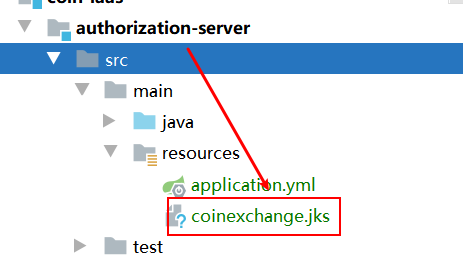
保存好，以后备用。

### 3.4.2 修改配置文件

|  |
| --- |
| server:  port: 9999 spring:  application:  name: authorization-server  cloud:  nacos:  discovery:  server-addr: nacos-server:8848 |

将之前的redis配置删除就可以了

### 3.4.3 将私钥文件复制到resource下



### 3.4.4 修改配置类载入私钥文件

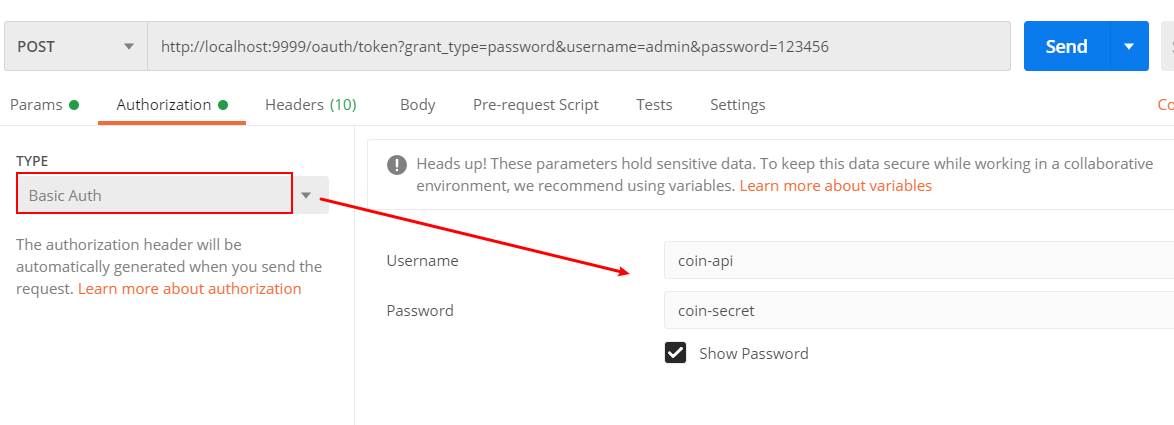
|  |
| --- |
| @EnableAuthorizationServer *// 开启授权服务器的功能* @Configuration public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {   @Autowired  private PasswordEncoder passwordEncoder ;   @Autowired  private AuthenticationManager authenticationManager ;   @Autowired  private UserDetailsService userDetailsService ;  *// @Autowired // private RedisConnectionFactory redisConnectionFactory ;    /\*\*  \* 添加第三方的客户端  \*/* @Override  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  clients.inMemory()  .withClient("coin-api") *// 第三方客户端的名称* .secret(passwordEncoder.encode("coin-secret")) *// 第三方客户端的密钥* .scopes("all") *//第三方客户端的授权范围* .accessTokenValiditySeconds(3600) *// token的有效期* .refreshTokenValiditySeconds(7\*3600);*// refresh\_token的有效期* super.configure(clients);  }   */\*\*  \* 配置验证管理器，UserdetailService  \*/* @Override  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  endpoints.authenticationManager(authenticationManager)  .userDetailsService(userDetailsService)  .tokenStore(jwtTokenStore()) *//设置token 存储在哪里* .tokenEnhancer(jwtAccessTokenConverter()) ;  super.configure(endpoints);  }   *// public TokenStore redisTokenStore(){ // return new RedisTokenStore(redisConnectionFactory) ; // }   /\*\*  \* jwtTokenStore  \* @return  \*/* **public TokenStore jwtTokenStore(){  JwtTokenStore jwtTokenStore = new JwtTokenStore(jwtAccessTokenConverter());  return jwtTokenStore ;  }**   **public JwtAccessTokenConverter jwtAccessTokenConverter(){  JwtAccessTokenConverter tokenConverter = new JwtAccessTokenConverter() ;  *// 读取classpath 下面的密钥文件* ClassPathResource classPathResource = new ClassPathResource("coinexchange.jks");  *// 获取KeyStoreFactory* KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(classPathResource,"coinexchange".toCharArray()) ;  *// 给JwtAccessTokenConverter 设置一个密钥对* tokenConverter.setKeyPair(keyStoreKeyFactory.getKeyPair("coinexchange","coinexchange".toCharArray()));  return tokenConverter ;  }** } |

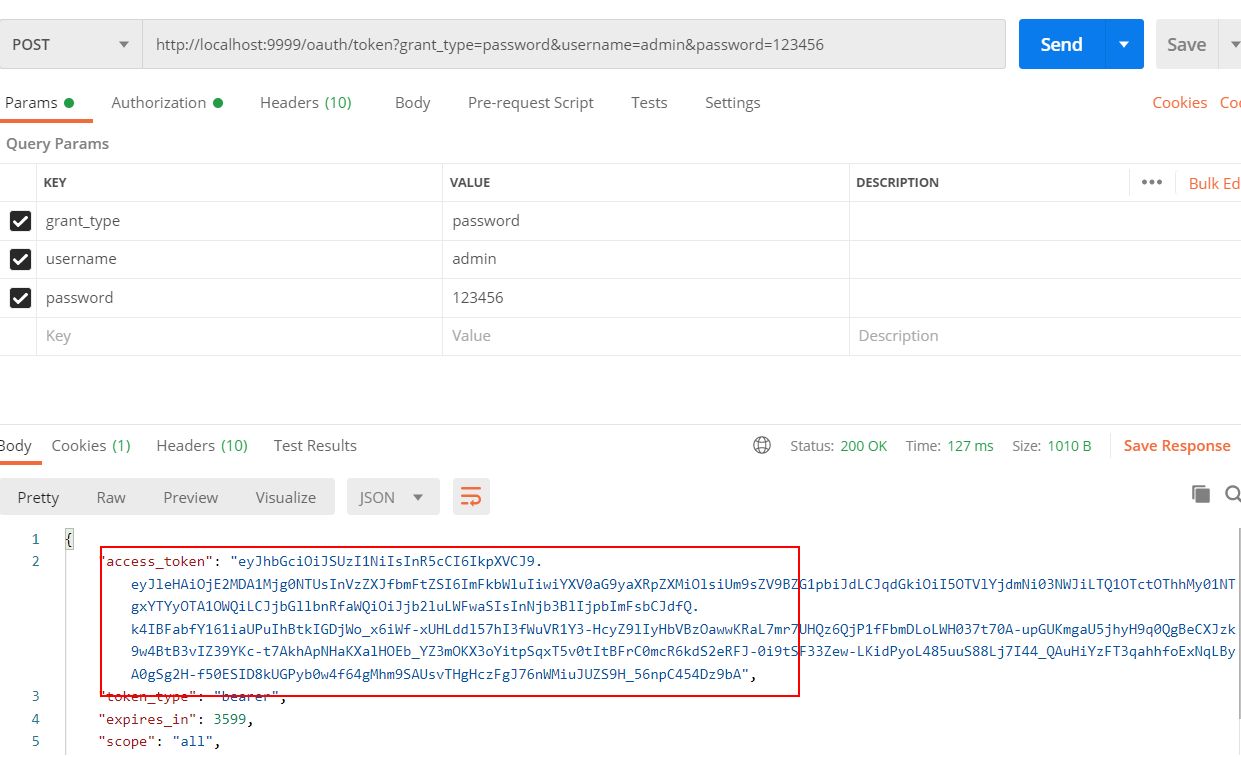
### 3.4.5 修改pom.xml 文件

将之前的redis 依赖去掉

|  |
| --- |
| <dependencies>  *<!-- 服务的发现和注册-->* <dependency>  <groupId>com.alibaba.cloud</groupId>  <artifactId>spring-cloud-alibaba-nacos-discovery</artifactId>  </dependency>  *<!-- OAuth2.0-->* <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-oauth2</artifactId>  </dependency>  *<!-- web-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  ***<!-- &lt;!&ndash; redis&ndash;&gt;--> <!-- <dependency>--> <!-- <groupId>org.springframework.boot</groupId>--> <!-- <artifactId>spring-boot-starter-data-redis</artifactId>--> <!-- </dependency>-->***</dependencies> |

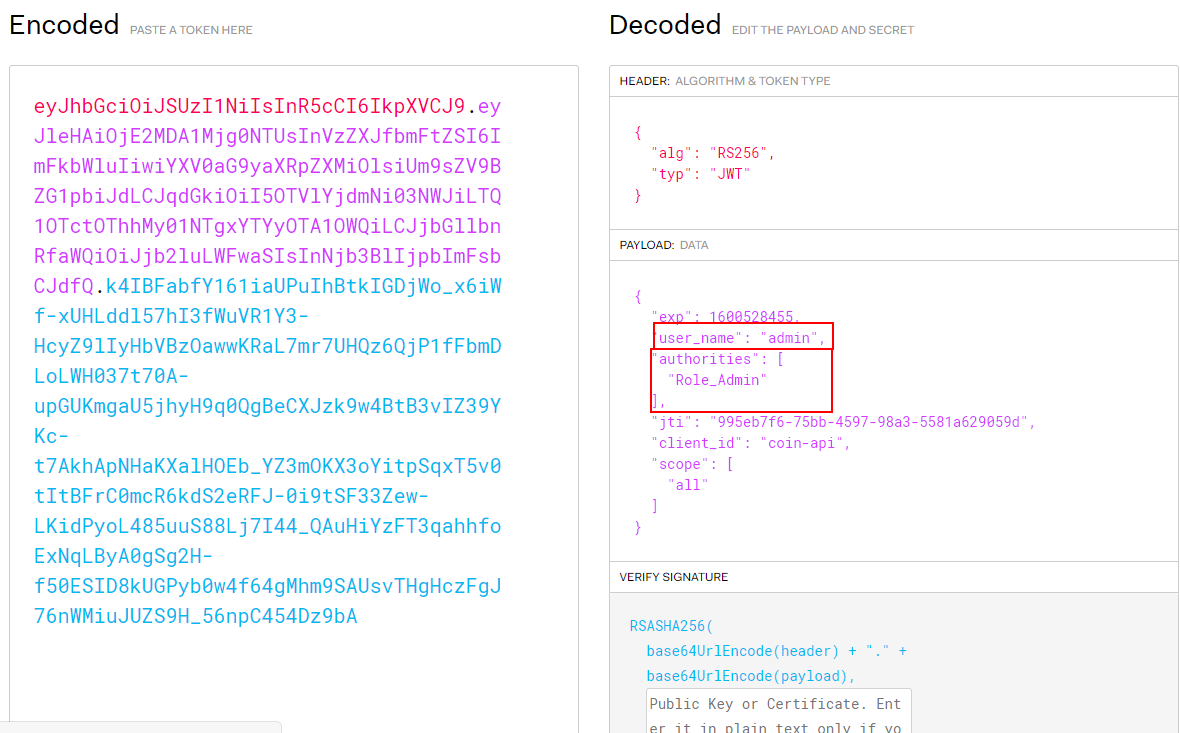
### 3.4.6 获取token 测试





### 3.4.7 使用jwt.io 校验token

|  |
| --- |
| https://jwt.io/ |

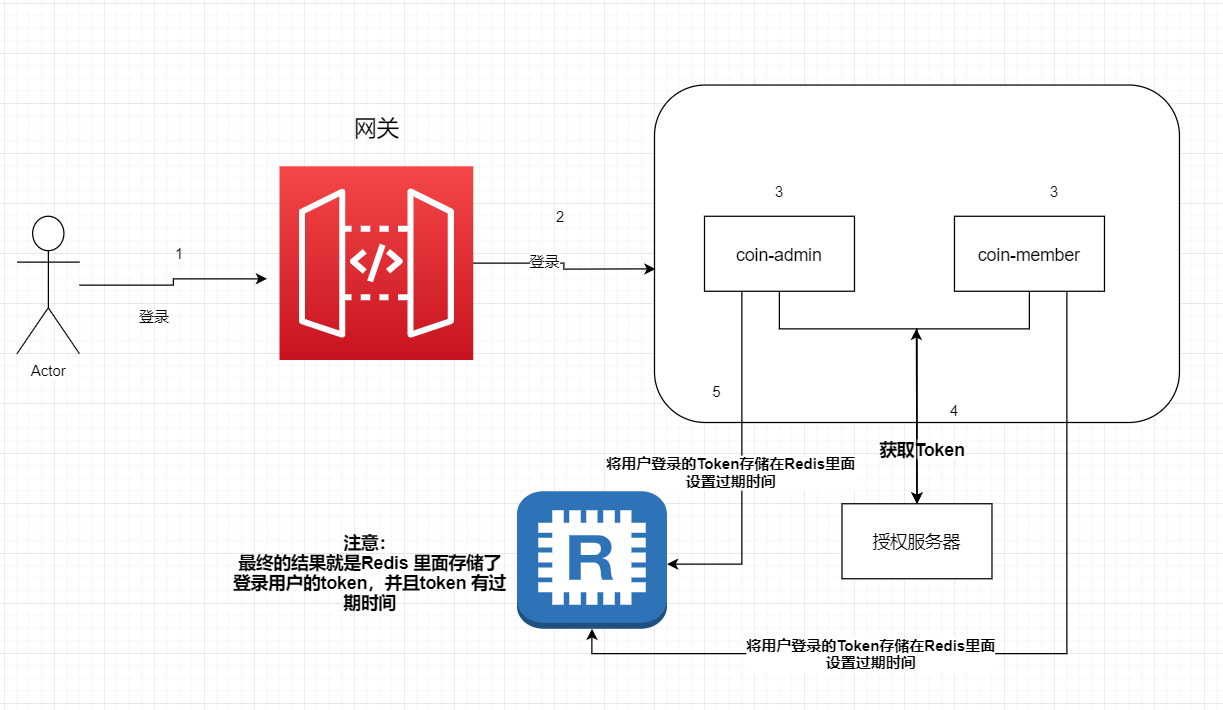


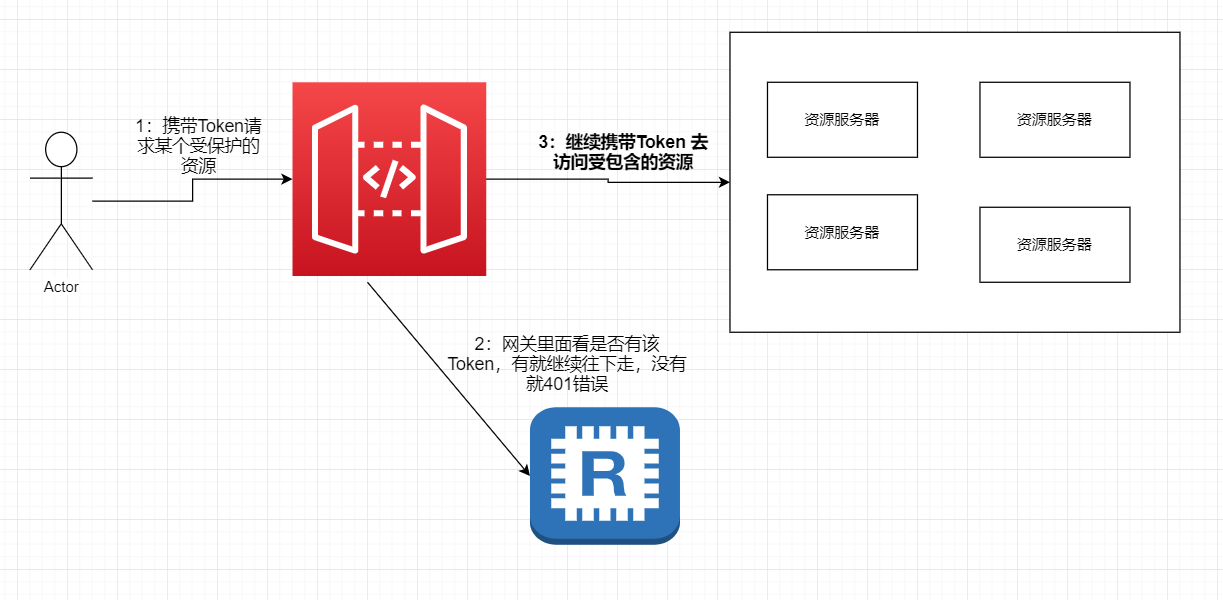
# JWT的登出问题

Jwt 使用起来不难，而且让我们将“无状态”的概念更贴切的展示出来了，但是实践就真的这么完美吗？不是，因为jwt的登出问题。

**何为登出：**就是用户自己点击登出后，或用户的角色/权限改变后，该token 仍然是有效的。你可以选择在前端清除该token，但是，如果用户是有技术背景的黑客呢？之前的token他保存一边，在没有过期（时间过期）时，他仍然可以使用该token。

**解决方案：**

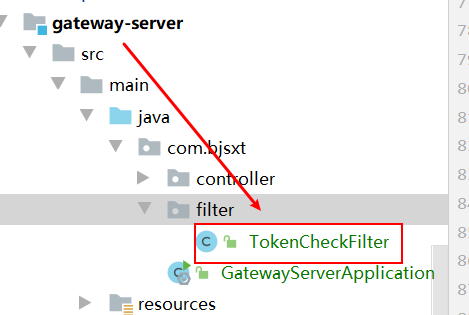




登出怎么做？聪明的你是否有答案吗？

就是删除该用户存储在redis里面登录的token数据，so easy 。

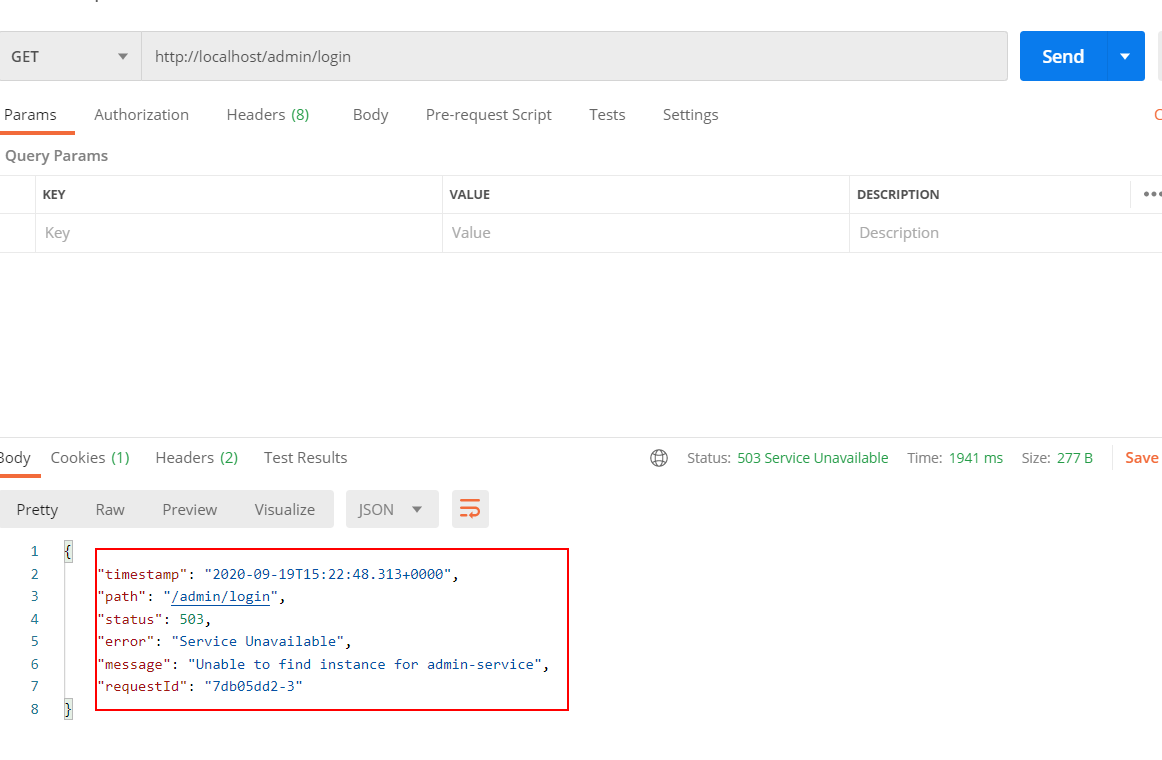
## 4.1 在网关里面判断该token是否存在



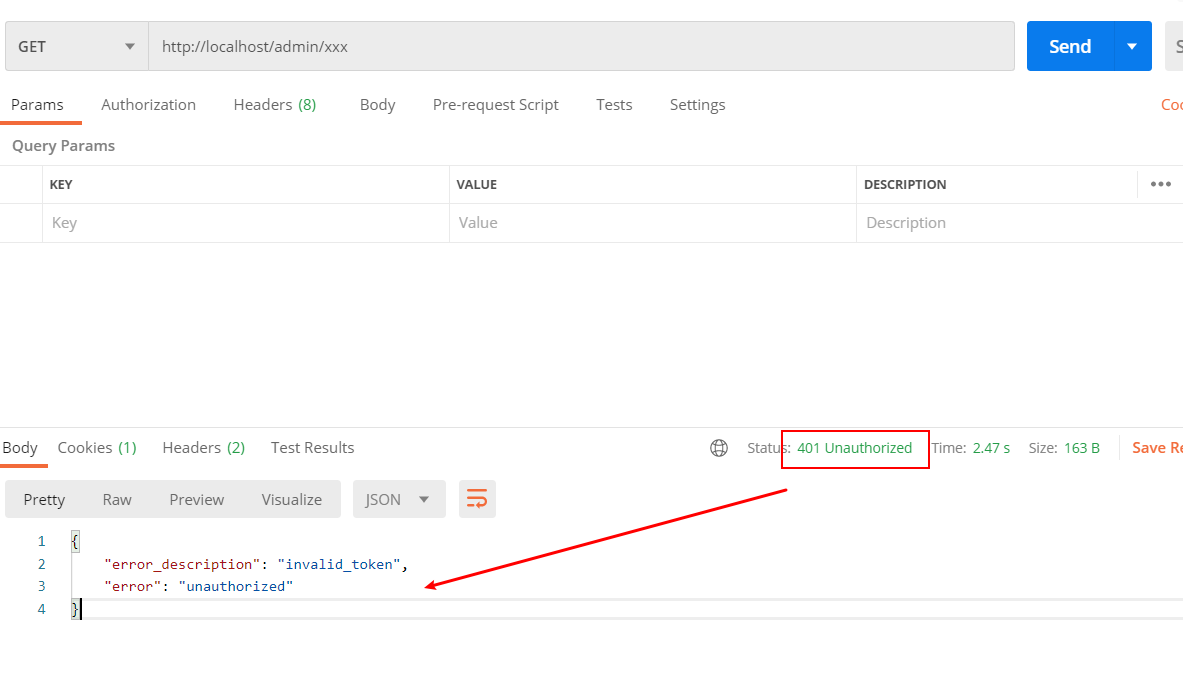
|  |
| --- |
| @Component public class TokenCheckFilter implements GlobalFilter, Ordered {   @Value("${no.token.access.urls:/admin/login,/admin/validate/code}")  private Set<String> noTokenAccessUrls; *// private Redis* @Override  public int getOrder() {  return 0;  }   */\*\*  \* 实现判断用户是否携带token ，或token 错误的功能  \*  \* @param exchange  \* @param chain  \* @return  \*/* @Override  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  *// 不需要token 就能访问* if (allowNoTokenAccess(exchange)) {  return chain.filter(exchange);  }  *// 获取用户的token* String token = getToken(exchange);   if (StringUtils.*isEmpty*(token)) { *// token 为 Empty* return buildUNAuthorizedResult(exchange);  }  if (false) {  return buildUNAuthorizedResult(exchange);  }   return chain.filter(exchange);  }   private boolean allowNoTokenAccess(ServerWebExchange exchange) {  String path = exchange.getRequest().getURI().getPath();  if (noTokenAccessUrls.contains(path)) {  return true;  }  return false;  }   */\*\*  \* 从头里面获取  \*  \* @param exchange  \* @return  \*/* private String getToken(ServerWebExchange exchange) {  ServerHttpRequest request = exchange.getRequest();  HttpHeaders headers = request.getHeaders();  String authorization = headers.getFirst(HttpHeaders.*AUTHORIZATION*);  if (Objects.*isNull*(authorization) || authorization.trim().isEmpty()) {  return null;  }  return authorization.replace("bearer ", "");  }   private Mono<Void> buildUNAuthorizedResult(ServerWebExchange exchange) {  ServerHttpResponse response = exchange.getResponse();  response.setStatusCode(HttpStatus.*UNAUTHORIZED*);  response.getHeaders().set("Content-Type", "application/json;charset=UTF-8");  JSONObject jsonObject = new JSONObject();  jsonObject.put("error", "unauthorized");  jsonObject.put("error\_description", "invalid\_token");  DataBuffer dataBuffer = response.bufferFactory().wrap(jsonObject.toJSONString().getBytes());  return response.writeWith(Flux.*just*(dataBuffer));  } } |

## 4.2 模拟访问

访问 /admin/login 这样的不需要token的资源时：

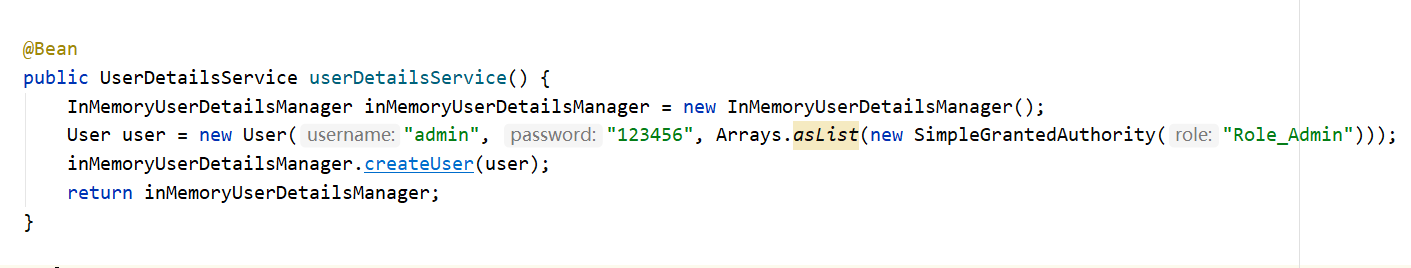


访问启动的资源时：



# 授权服务器的接入

之前我们仅仅在authorizaiton-server 里面添加了一个模拟的用户:



还没有接入到我们的系统的用户数据,本节课我们来接入一下我们的用户数据

## 5.1 添加登录常量

在该登录常量里面，我们可以定义受支持的登录类型

|  |
| --- |
| public class LoginConstant {   */\*\*  \* 管理员登录  \*/* public static final String *ADMIN\_TYPE* = "admin\_type" ;   */\*\*  \* 用户/会员登录  \*/* public static final String *MEMBER\_TYPE* = "member\_type" ;   } |

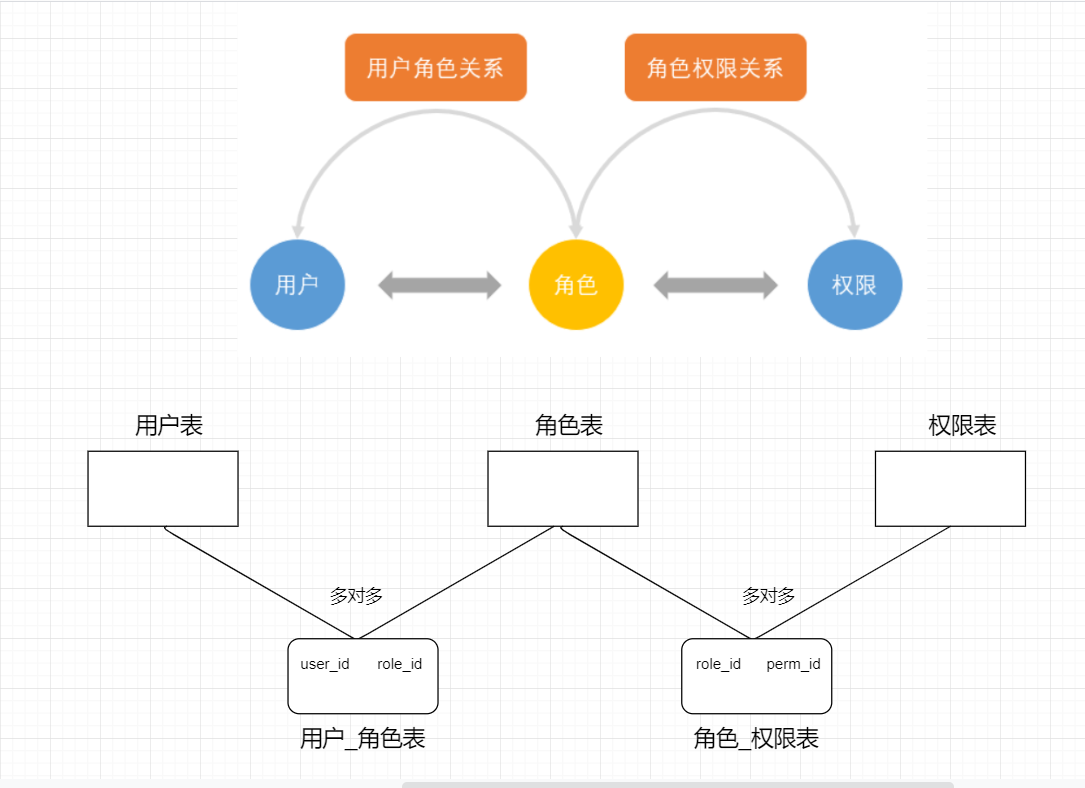
## 5.2 实现UserDetailService 接口

|  |
| --- |
| */\*\*  \* 登录的实现  \*  \* @param username  \* @return  \* @throws UsernameNotFoundException  \*/* @Override public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  ServletRequestAttributes requestAttributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  String loginType = requestAttributes.getRequest().getParameter("login\_type");  if (StringUtils.*isEmpty*(loginType)) {  throw new AuthenticationServiceException("请添加login\_type参数");  }  UserDetails userDetails = null;  switch (loginType) {  case LoginConstant.*ADMIN\_TYPE*: *// 管理员登录* userDetails = loadAdminUserByUsername(username);  break;  case LoginConstant.*MEMBER\_TYPE*: *// 会员登录* userDetails = loadMemberUserByUsername(username);  break;  default:  throw new AuthenticationServiceException("暂不支持的登录方式" + loginType);  }  return userDetails; } |

## 5.3 管理员用户的登录

### 5.3.1 RBAC模型

RBAC 是基于角色的访问控制（Role-Based Access Control ）在 RBAC 中，权限与角色相关联，用户通过成为适当角色的成员而得到这些角色的权限。这就极大地简化了权限的管理。这样管理都是层级相互依赖的，权限赋予给角色，而把角色又赋予用户，这样的权限设计很清楚，管理起来很方便。



### 5.3.2 依赖导入

我们选择简单的jdbcTemplate 来做权限的查询操作

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId> </dependency> |

### 5.3.3 配置数据源

|  |
| --- |
| server:  port: 9999 spring:  application:  name: authorization-server  cloud:  nacos:  discovery:  server-addr: nacos-server:8848  datasource:  url: jdbc:mysql://mysql-server:3307/coin-exchange?useSSL=false&serverTimezone=GMT%2B8  driver-class-name: com.mysql.cj.jdbc.Driver  username: root  password: Ltd3411?? |

### 5.3.4 准备SQL语句

以下sql 语句都位于LoginConstant 常量里面

#### 一、用于登录

使用用户名查询用户的SQL:

|  |
| --- |
| public static final String *QUERY\_ADMIN\_SQL* =  "SELECT `id` ,`username`, `password`, `status` FROM sys\_user WHERE username = ? "; |

#### 二、查询用户的权限

0 判断用户是否为管理员

|  |
| --- |
| public static final String *QUERY\_ROLE\_CODE\_SQL* =  "SELECT `code` FROM sys\_role LEFT JOIN sys\_user\_role ON sys\_role.id = sys\_user\_role.role\_id WHERE sys\_user\_role.user\_id= ?"; |

1 用户为管理员时：（拥有全部的权限）

|  |
| --- |
| public static final String *QUERY\_ALL\_PERMISSIONS* =  "SELECT `name` FROM sys\_privilege"; |

2 普通用户时（通过用户的角色查询用户的权限）

|  |
| --- |
| public static final String *QUERY\_PERMISSION\_SQL* =  "SELECT *\** FROM sys\_privilege LEFT JOIN sys\_role\_privilege ON sys\_role\_privilege.privilege\_id = sys\_privilege.id LEFT JOIN sys\_user\_role ON sys\_role\_privilege.role\_id = sys\_user\_role.role\_id WHERE sys\_user\_role.user\_id = ?"; |

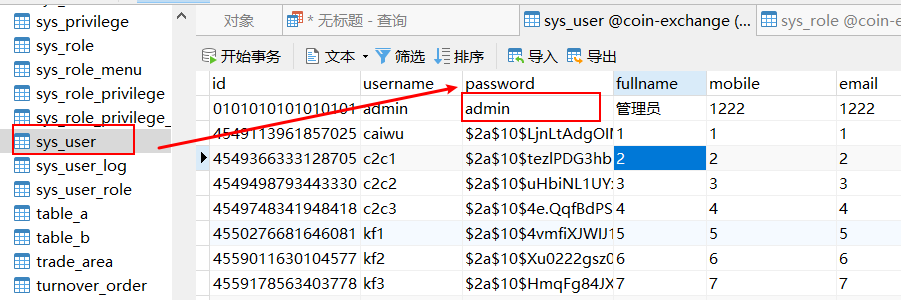
### 5.3.5 代码实现

|  |
| --- |
| */\*\*  \* 登录的实现  \*  \* @param username  \* @return  \* @throws UsernameNotFoundException  \*/* @Override public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  ServletRequestAttributes requestAttributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  String loginType = requestAttributes.getRequest().getParameter("login\_type");  if (StringUtils.*isEmpty*(loginType)) {  throw new AuthenticationServiceException("请添加login\_type参数");  }  UserDetails userDetails = null;  try {  switch (loginType) {  case LoginConstant.*ADMIN\_TYPE*: *// 管理员登录* userDetails = loadAdminUserByUsername(username);  break;  case LoginConstant.*MEMBER\_TYPE*: *// 会员登录* userDetails = loadMemberUserByUsername(username);  break;  default:  throw new AuthenticationServiceException("暂不支持的登录方式" + loginType);  }  } catch (IncorrectResultSizeDataAccessException e) {  throw new UsernameNotFoundException("会员：" + username + "不存在");  }  return userDetails; } |

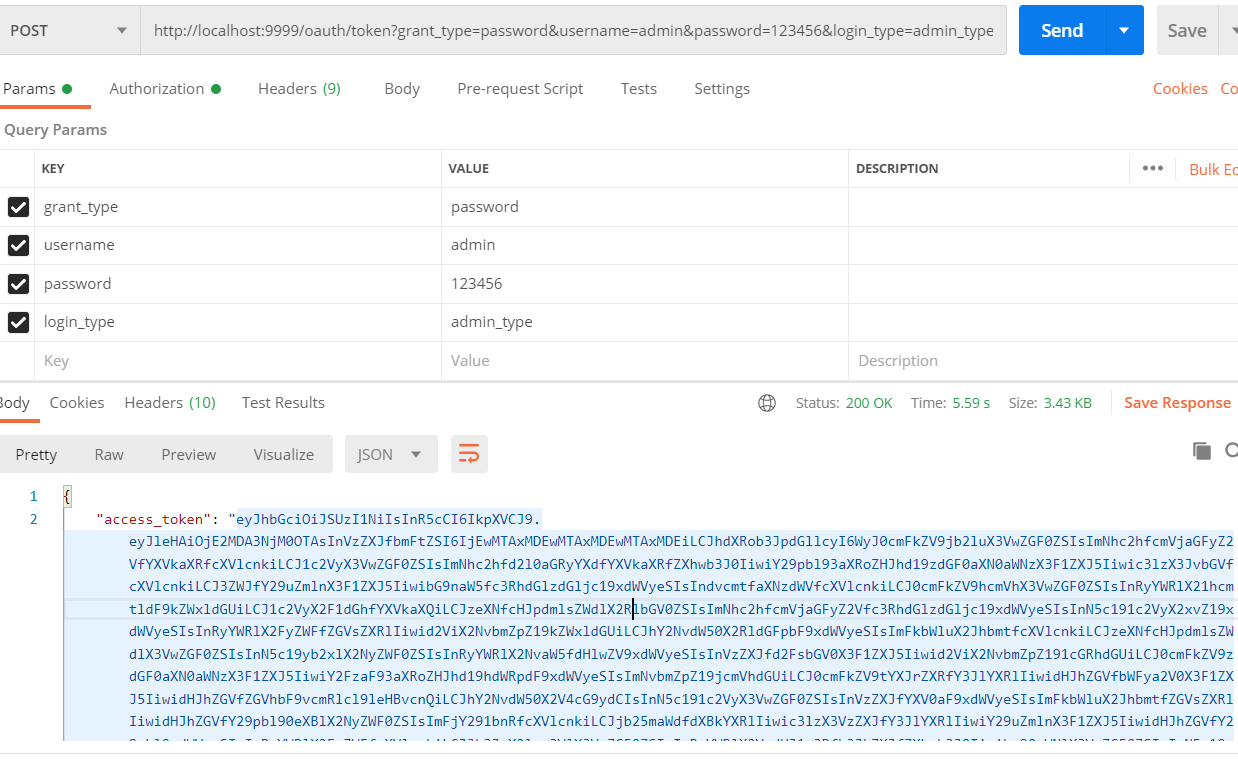
|  |
| --- |
| */\*\*  \* 对接管理员的登录  \*  \* @param username  \* @return  \*/* private UserDetails loadAdminUserByUsername(String username) {  return jdbcTemplate.queryForObject(*QUERY\_ADMIN\_SQL*, new RowMapper<User>() {  @Override  public User mapRow(ResultSet rs, int rowNum) throws SQLException {  if (rs.wasNull()) {  throw new UsernameNotFoundException("用户：" + username + "不存在");  }  Long id = rs.getLong("id");  String password = rs.getString("password");  int status = rs.getInt("status");  User user = new User(  String.*valueOf*(id), *// 使用用户的id 代替用户的名称，这样会使得后面的很多情况得以处理* password,  status == 1,  true,  true,  true,  getUserPermissions(id));  return user;  }  }, username); }  */\*\*  \* 通过用户的id 获取用户的权限  \*  \* @param id  \* @return  \*/* private Set<SimpleGrantedAuthority> getUserPermissions(Long id) {  *// 查询用户是否为管理员* String code = jdbcTemplate.queryForObject(*QUERY\_ROLE\_CODE\_SQL*, String.class, id);  List<String> permissions = null;  if (*ADMIN\_CODE*.equals(code)) { *// 管理员* permissions = jdbcTemplate.queryForList(*QUERY\_ALL\_PERMISSIONS*, String.class);  } else {  permissions = jdbcTemplate.queryForList(*QUERY\_PERMISSION\_SQL*, String.class, id);  }  if (permissions == null || permissions.isEmpty()) {  return Collections.*EMPTY\_SET*;  }  return permissions  .stream()  .distinct() *// 去重* .map(  perm -> new SimpleGrantedAuthority(perm) *// perm - >security可以识别的权限* )  .collect(Collectors.*toSet*()); } |

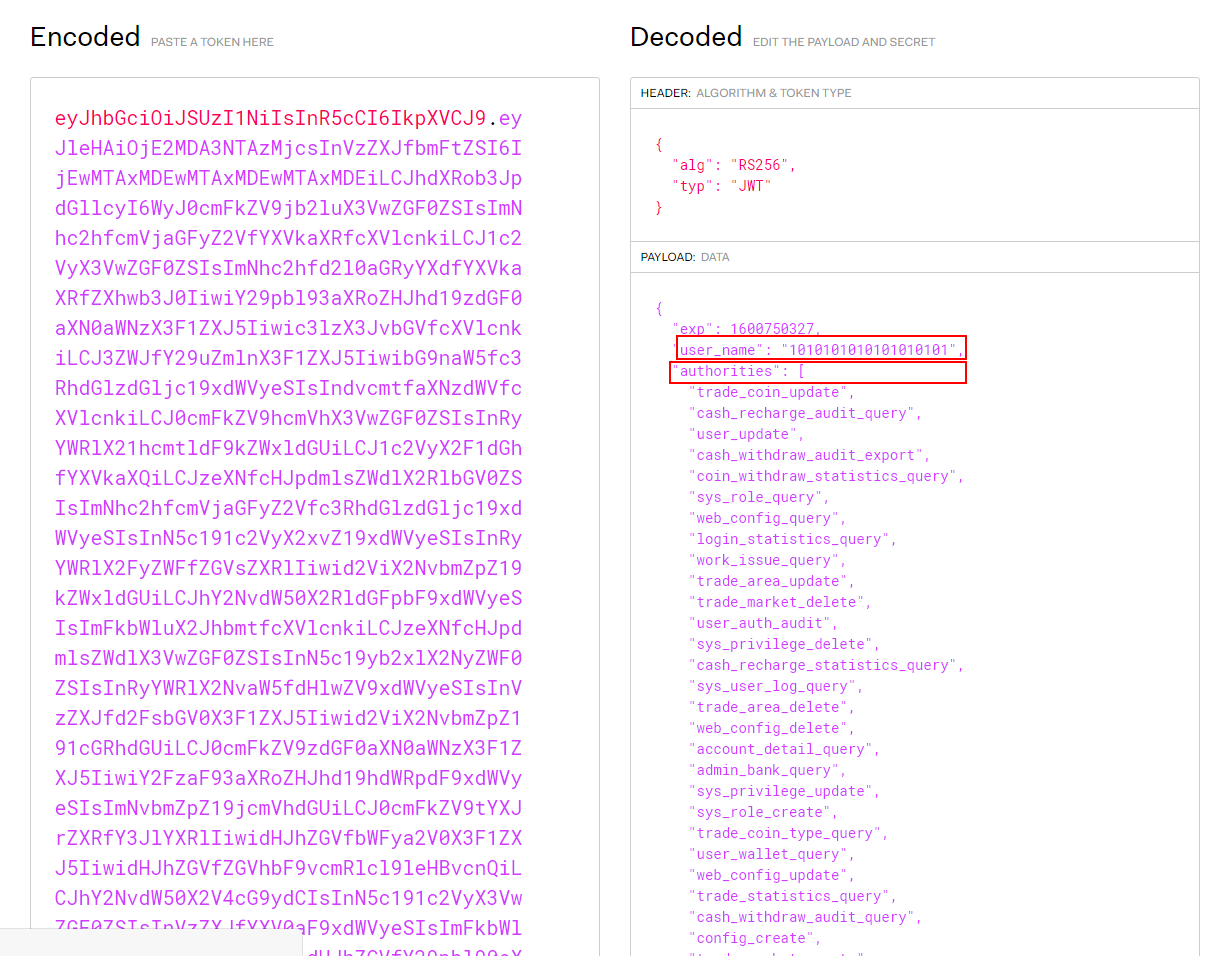
### 5.3.6 测试效果

注意，先将数据库里面sys\_user表里面，admin用户的密码修改为admin



这样测试起来简单一点。



我们发现，已经获取到了Token,看看token 里面都藏了什么：  


登录已经完成了

### 5.3.7 密码加密器



修改：WebSecurityConfig：将之前的PasswordEncoder 修改为以下的代码：

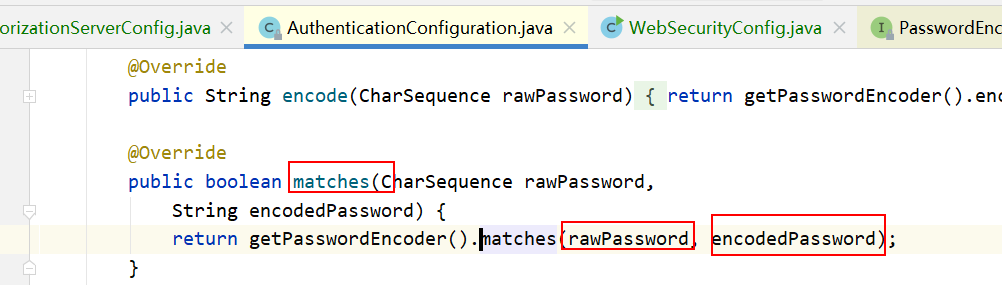
|  |
| --- |
| */\*\*  \* 密码加密  \*  \* @return  \*/* @Bean public PasswordEncoder passwordEncoder() {  return new BCryptPasswordEncoder() ; } |

我们的密码加密器将会影响2 个地方：

1 第三方客户端

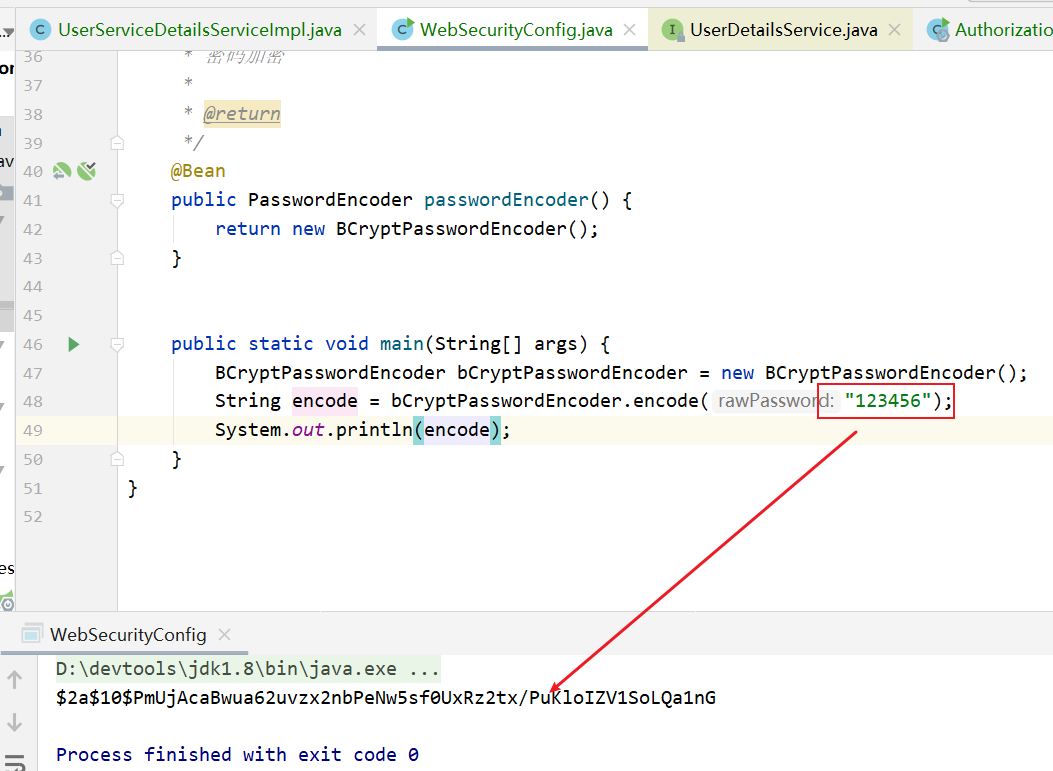


2 用户登录时的密码匹配



### 5.3.8 测试密码匹配器的效果

先获取测试数据：



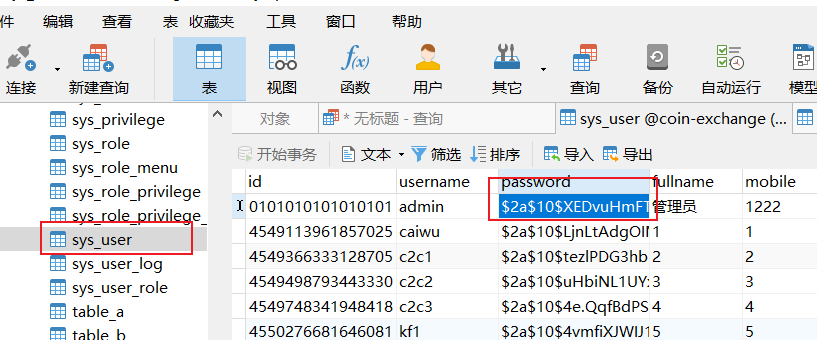
|  |
| --- |
| public static void main(String[] args) {  BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();  String encode = bCryptPasswordEncoder.encode("123456");  System.*out*.println(encode);  System.*out*.println(encode1); } |

输出为：

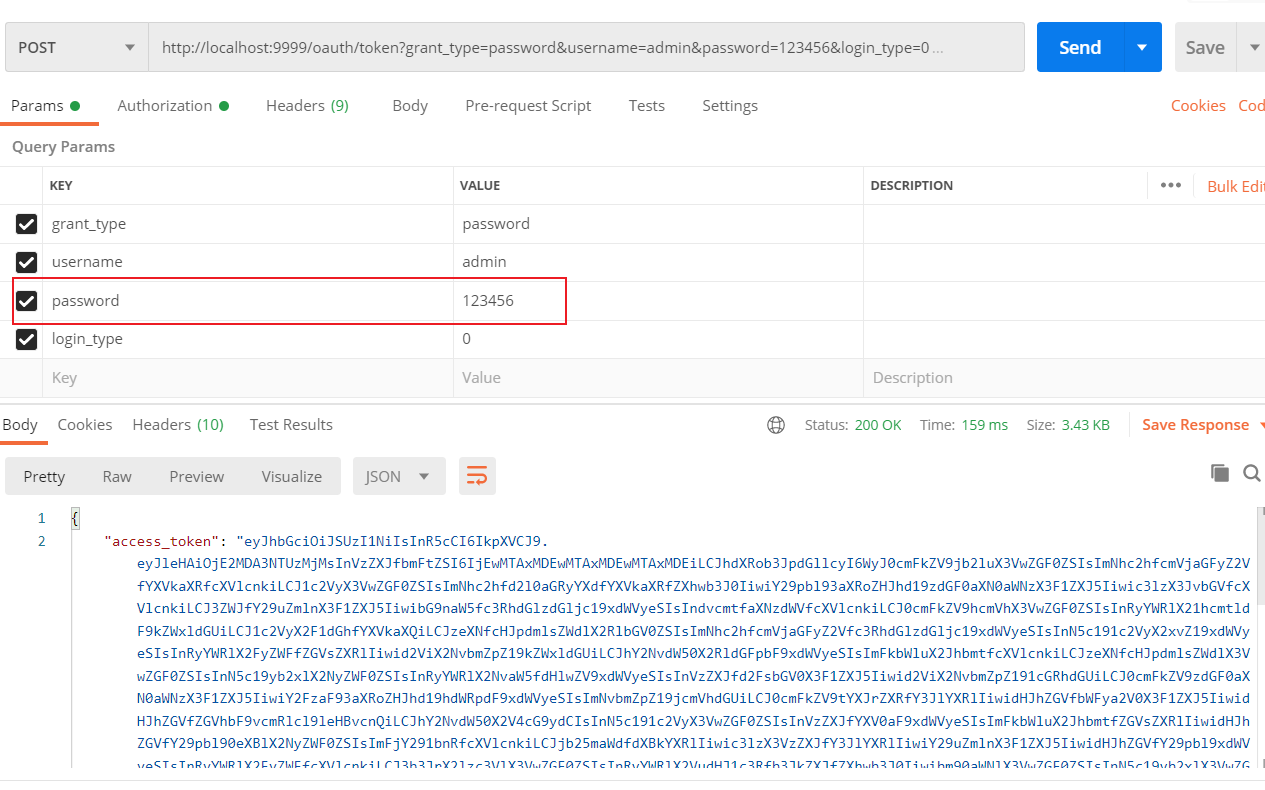
|  |
| --- |
| $2a$10$PmUjAcaBwua62uvzx2nbPeNw5sf0UxRz2tx/PuKloIZV1SoLQa1nG |

**先将数据库里面sys\_user表里面，admin用户的密码修改为：**

|  |
| --- |
| $2a$10$PmUjAcaBwua62uvzx2nbPeNw5sf0UxRz2tx/PuKloIZV1SoLQa1nG |



获取Token：



还是没有任何的错误。

## 5.4 会员登录的接入

会员没有复杂的RBAC模型处理，我们仅仅做简单的登录就可以了。

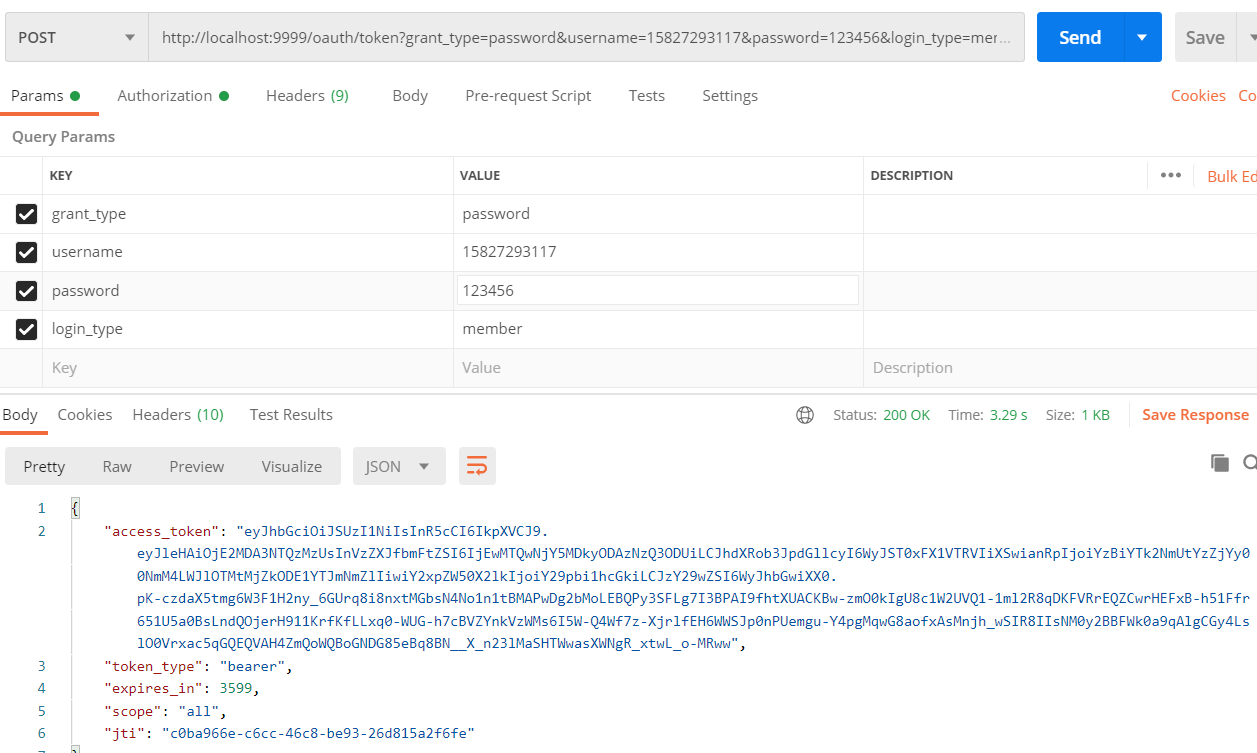
### 5.4.1 准备SQL 语句

|  |
| --- |
| public static final String *QUERY\_MEMBER\_SQL* =  "SELECT `id`,`password`, `status` FROM `user` WHERE mobile = ? or email = ? "; |

### 5.4.2 代码实现

|  |
| --- |
| */\*\*  \* 对接会员的登录  \*  \* @param username  \* @return  \*/* private UserDetails loadMemberUserByUsername(String username) {  return jdbcTemplate.queryForObject(*QUERY\_MEMBER\_SQL*, new RowMapper<UserDetails>() {  @Override  public UserDetails mapRow(ResultSet rs, int rowNum) throws SQLException {  if(rs.wasNull()){  throw new UsernameNotFoundException("会员：" + username + "不存在");  }  long id = rs.getLong("id"); *// 获取用户的id* String password = rs.getString("password");  int status = rs.getInt("status");  return new User(  String.*valueOf*(id),  password,  status == 1 ,  true ,  true ,  true,  Arrays.*asList*(new SimpleGrantedAuthority("ROLE\_USER"))  );  }  }, username, username); } |

### 5.4.3 测试登录



# refresh\_token和过期时间

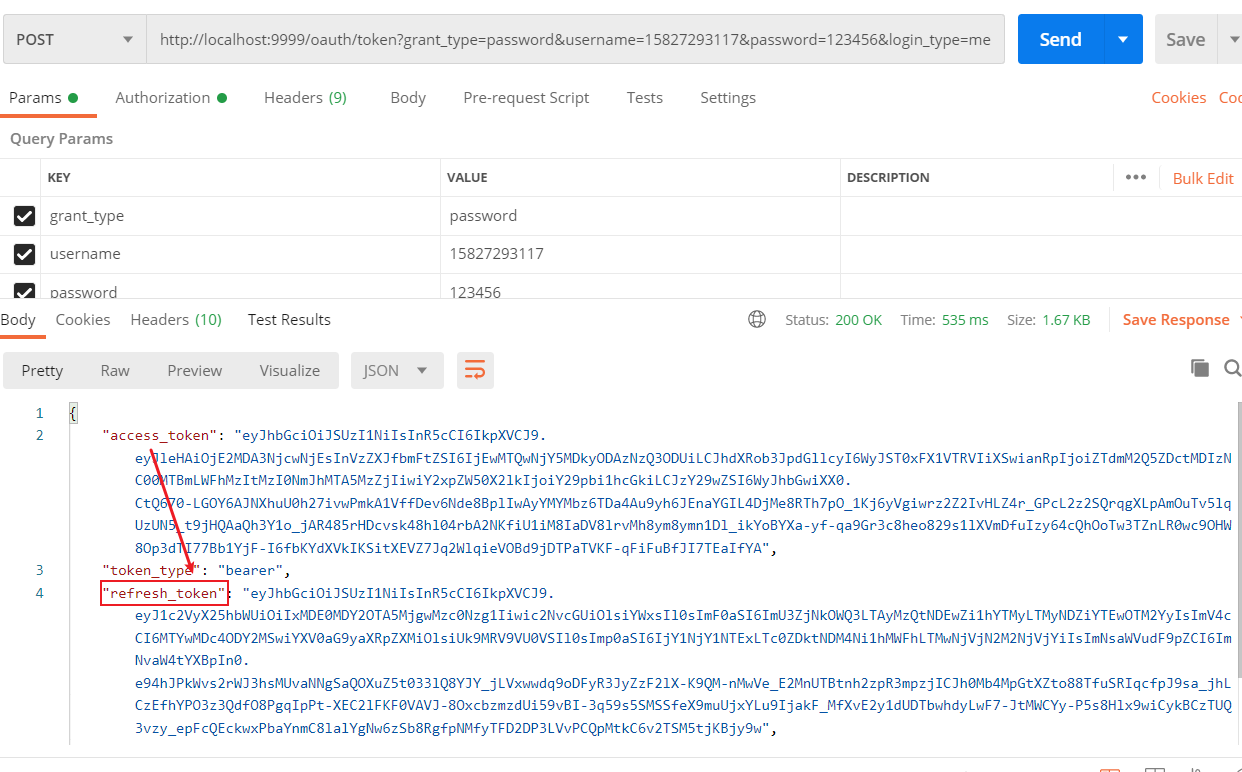
我们可以可以使用refresh\_token 来为过期的token 获取一个新的token数据

## 6.1 添加验证方式



## 6.2 获取信息的token 测试

重启后重新后期：



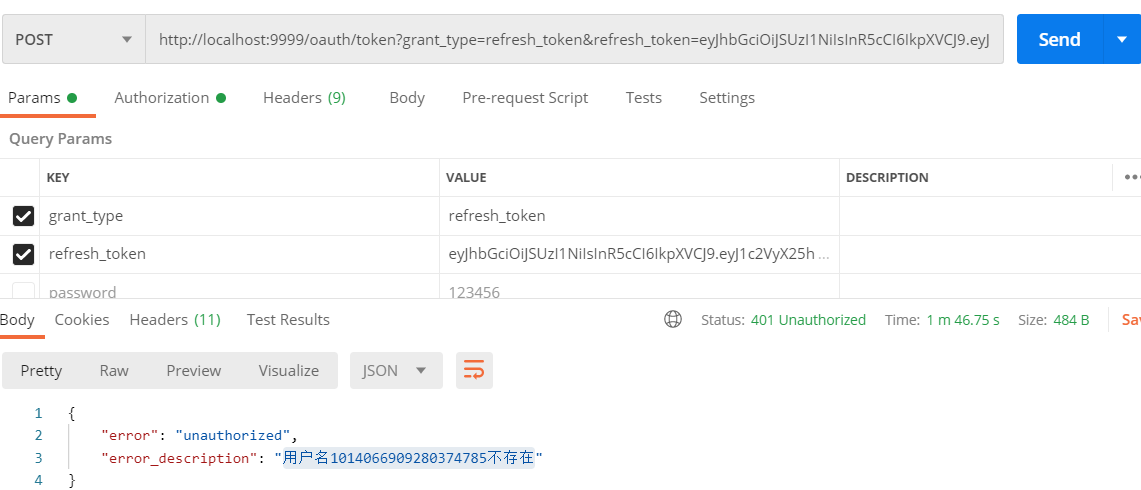
大家可以看见，我们获取到的数据新增了refresh\_token的一项，我们来看看它里面包含那些信息：



基本和之前时没有区别的。

## 6.3 使用Refresh\_token获取新的token

先看错误：



原因在于：我们把jwt 里面的username 换成了现在的 用户 id ，导致的。

现在，我们需要一个纠正的过程：

Refresh\_token的标识：（loginConstant）

|  |
| --- |
| */\*\*  \* token的刷新  \*/* public static final String *REFRESH\_TOKEN* = "REFRESH\_TOKEN" ; |

|  |
| --- |
| @Override public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  ServletRequestAttributes requestAttributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  String loginType = requestAttributes.getRequest().getParameter("login\_type"); *// 区分时后台人员还是我们的用户登录* if (StringUtils.*isEmpty*(loginType)) {  throw new AuthenticationServiceException("登录类型不能为null");  }  String grantType = requestAttributes.getRequest().getParameter("grant\_type");  UserDetails userDetails = null;  try {  if (LoginConstant.*REFRESH\_TOKEN*.equals(grantType.toUpperCase())) {  username = adjustUsername(username, loginType); *// 为refresh\_token 时，需要将id->username* }  switch (loginType) {  case LoginConstant.*ADMIN\_TYPE*:  userDetails = loadSysUserByUsername(username);  break;  case LoginConstant.*MEMBER\_TYPE*:  userDetails = loadMemberUserByUsername(username);  break;  default:  throw new AuthenticationServiceException("暂不支持的登录方式:" + loginType);  }  } catch (IncorrectResultSizeDataAccessException e) { *// 我们的用户不存在* throw new UsernameNotFoundException("用户名" + username + "不存在");  }   return userDetails; } |

纠正的实现：

添加SQL语句：

|  |
| --- |
| */\*\*  \* 使用用户的id 查询用户名称  \*/* public static final String *QUERY\_ADMIN\_USER\_WITH\_ID* = "SELECT `username` FROM sys\_user where id = ?" ;  */\*\*  \* 使用用户的id 查询用户名称  \*/* public static final String *QUERY\_MEMBER\_USER\_WITH\_ID* = "SELECT `mobile` FROM user where id = ?" ; |

实现纠正：

|  |
| --- |
| */\*\*  \* 纠正在refresh 场景下的登录问题  \*  \* @param username  \* @param loginType  \* @return  \*/* private String adjustUsername(String username, String loginType) {  if (LoginConstant.*ADMIN\_TYPE*.equals(loginType)) {  return jdbcTemplate.queryForObject(LoginConstant.*QUERY\_ADMIN\_USER\_WITH\_ID*, String.class, username);  }  if (LoginConstant.*MEMBER\_TYPE*.equals(loginType)) {  return jdbcTemplate.queryForObject(LoginConstant.*QUERY\_MEMBER\_USER\_WITH\_ID*, String.class, username);  }  return username; } |

## 6.4 token 过期时间的设置



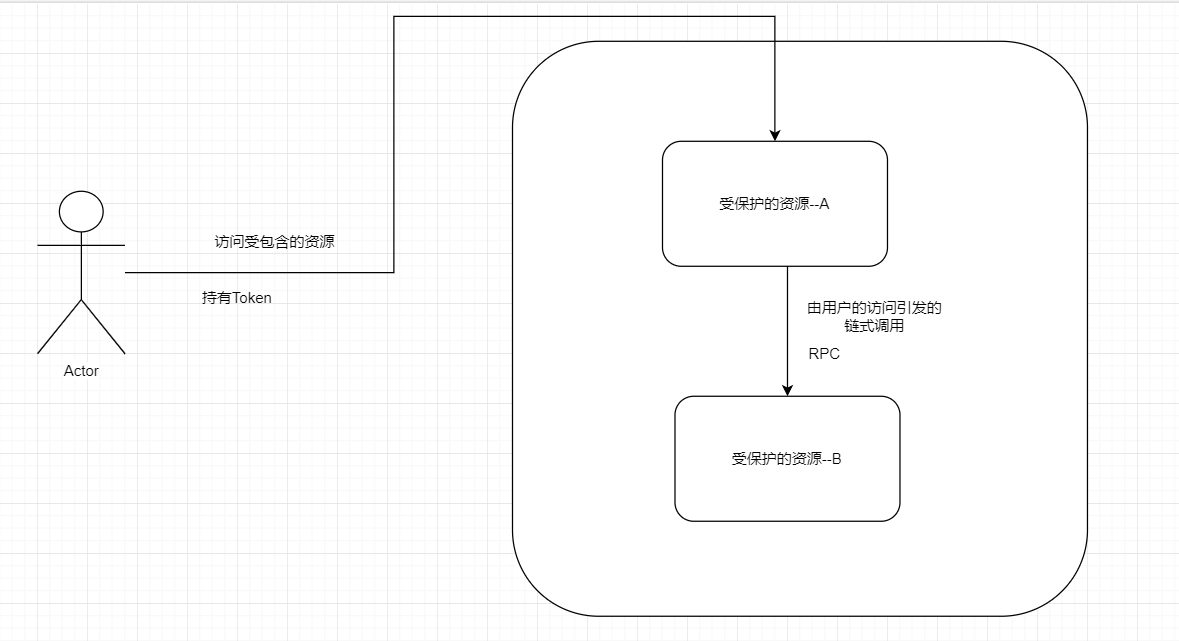
Token的有效期为一周，

Refresh\_token的有效期为一个月。

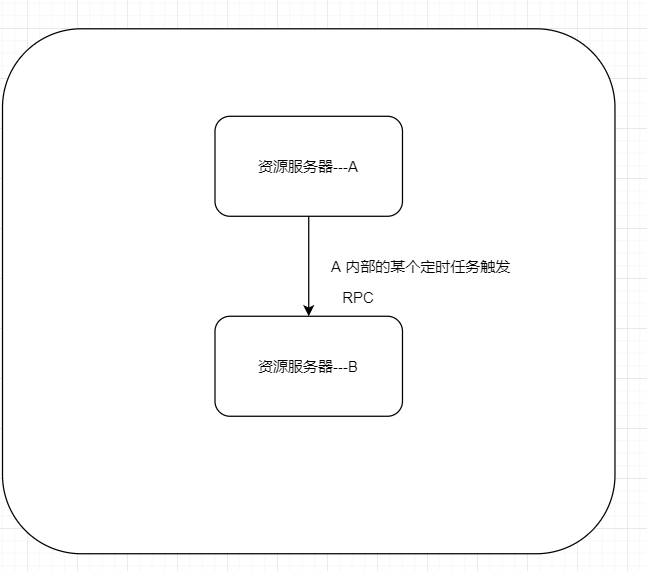
# Token传递和获取

## 7.1 受保护资源之前Token的传递

Case1：



Case2：



在第一种Case 里面，我们可以从本次**请求的上下文里面获取用户的token** ，进行一个Token的传递。

在第二种Case 里面，我们没有一个用户请求的上下文，因此我们需要应用自己去获取一个临时的token。

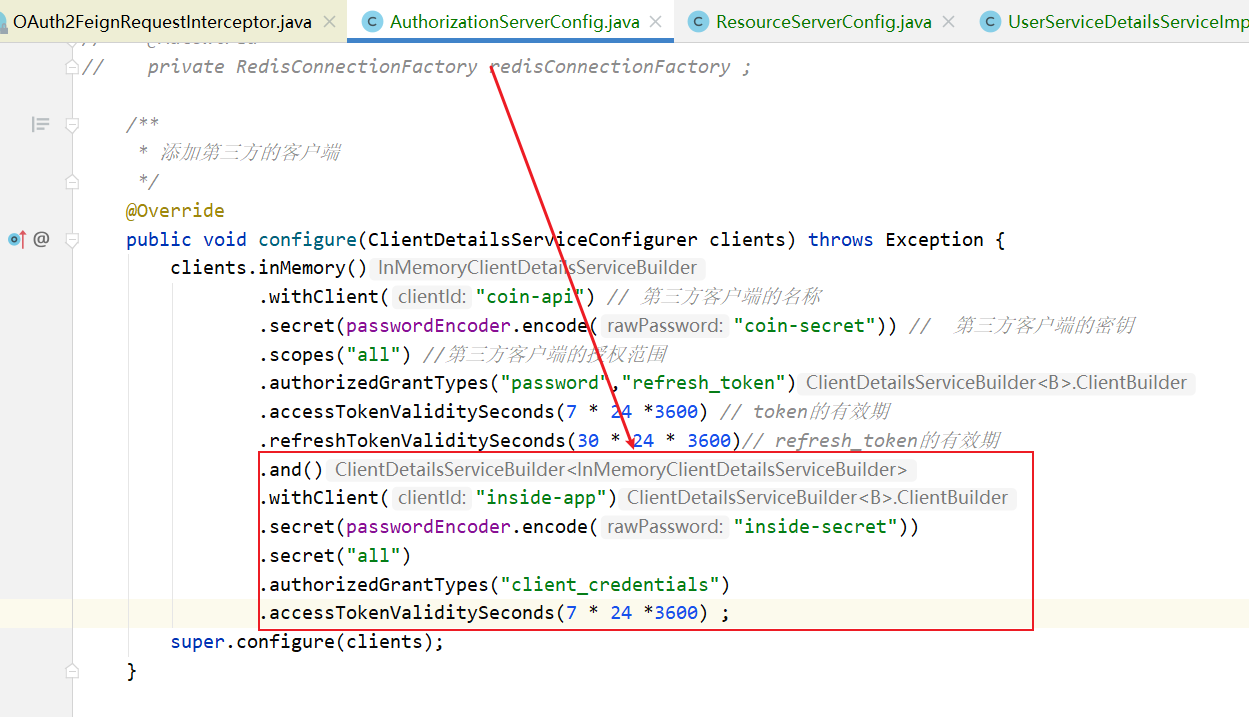
这2种请求的源码实现，OAuth2.0 已经帮我们写好了，在：



获取的方式非常的简单：使用client\_credentials 授权方式来进行的

## 7.2 在authorization-server 里面添加客户端授权的方式

|  |
| --- |
| */\*\*  \* 添加第三方的客户端  \*/* @Override public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  clients.inMemory()  .withClient("coin-api") *// 第三方客户端的名称* .secret(passwordEncoder.encode("coin-secret")) *// 第三方客户端的密钥* .scopes("all") *//第三方客户端的授权范围* .authorizedGrantTypes("password","refresh\_token")  .accessTokenValiditySeconds(7 \* 24 \*3600) *// token的有效期* .refreshTokenValiditySeconds(30 \* 24 \* 3600)*// refresh\_token的有效期* .and()  .withClient("inside-app")  .secret(passwordEncoder.encode("inside-secret"))  .secret("all")  .authorizedGrantTypes("client\_credentials")  .accessTokenValiditySeconds(7 \* 24 \*3600) ;  super.configure(clients); } |



## 7.3 获取token 测试

