**SpringSecurity第三天**

上海千锋 文威

* 理解什么是分布式认证
* Jwt介绍
* SpringSecurity+JWT+RSA实现分布式认证

# 1 分布式认证

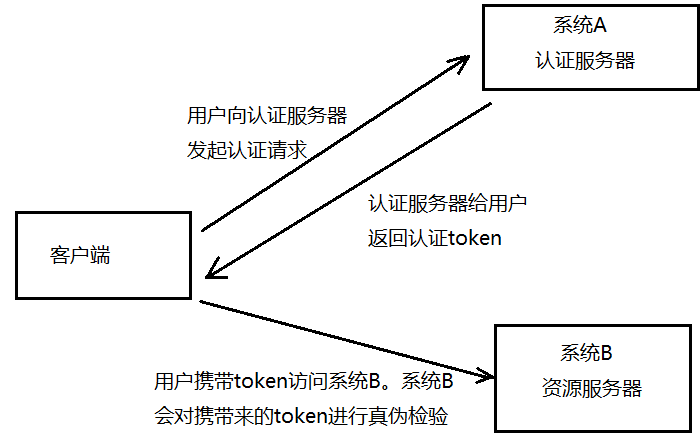
## 1.1 分布式认证的概念

分布式认证，即我们常说的单点登录，简称SSO，指的是在多应用系统的项目中，用户只需要登录一次，就可以访 问所有互相信任的应用系统。

## 1.2 分布式认证流程图

首先，我们要明确，在分布式项目中，每台服务器都有各自独立的session，而这些session之间是无法直接共享资 源的，所以，session通常不能被作为单点登录的技术方案。

**最合理的单点登录方案流程如下图所示：**



总结一下，单点登录的实现分两大环节：

**用户认证：**这一环节主要是用户向认证服务器发起认证请求，认证服务器给用户返回一个成功的令牌token， 主要在认证服务器中完成，即图中的A系统，注意A系统只能有一个。

**身份校验：**这一环节是用户携带token去访问其他服务器时，在其他服务器中要对token的真伪进行检验，主 要在资源服务器中完成，即图中的B系统，这里B系统可以有很多个。

# 2 JWT介绍

## 2.1 JWT概念

从分布式认证流程中，我们不难发现，这中间起最关键作用的就是token，token的安全与否，直接关系到系统的健壮性，这里我们选择使用JWT来实现token的生成和校验。 JWT，全称JSON Web Token，官网地址<https://jwt.io>，是一款出色的分布式身份校验方案。可以生成token，也可以解析检验token。

JWT生成的token由三部分组成：

**头部：**主要设置一些规范信息，签名部分的编码格式就在头部中声明。

**载荷：**token中存放有效信息的部分，比如用户名，用户角色，过期时间等，但是不要放密码，会泄露！

**签名：**将头部与载荷分别采用base64编码后，用“.”相连，再加入盐，最后使用头部声明的编码类型进行编码，就得到了签名。

## 2.2 JWT生成token的安全性分析

从JWT生成的token组成上来看，要想避免token被伪造，主要就得看签名部分了，而签名部分又有三部分组成，其中头部和载荷的base64编码，几乎是透明的，毫无安全性可言，那么最终守护token安全的重担就落在了加入的盐上面了！

试想：如果生成token所用的盐与解析token时加入的盐是一样的。岂不是类似于中国人民银行把人民币防伪技术公开了？大家可以用这个盐来解析token，就能用来伪造token。

这时，我们就需要对盐采用非对称加密的方式进行加密，以达到生成token与校验token方所用的盐不一致的安全效果！

## 2.3 非对称加密RSA介绍

* 基本原理：同时生成两把密钥：私钥和公钥，私钥隐秘保存，公钥可以下发给信任客户端
* 私钥加密
* 公钥解密
* 优点：安全，难以破解
* 缺点：算法比较耗时，为了安全，可以接受
* 历史：三位数学家Rivest、Shamir 和 Adleman 设计了一种算法，可以实现非对称加密。这种算法用他们三 个人的名字缩写：RSA。

## 2.4 JWT相关工具类

* 相关jar包

|  |
| --- |
| <**dependency**>  <**groupId**>io.jsonwebtoken</**groupId**>  <**artifactId**>jjwt-api</**artifactId**>  <**version**>0.10.7</**version**> </**dependency**> <**dependency**>  <**groupId**>io.jsonwebtoken</**groupId**>  <**artifactId**>jjwt-impl</**artifactId**>  <**version**>0.10.7</**version**>  <**scope**>runtime</**scope**> </**dependency**> <**dependency**>  <**groupId**>io.jsonwebtoken</**groupId**>  <**artifactId**>jjwt-jackson</**artifactId**>  <**version**>0.10.7</**version**>  <**scope**>runtime</**scope**> </**dependency**> |

* 载荷对象

|  |
| --- |
| */\*\*  \** ***@author*** *krisswen  \* 为了方便后期获取token中的用户信息，将token中载荷部分单独封装成一个对象  \*/* @Data **public class** Payload<T> {  **private** String **id**;  **private** T **userInfo**;  **private** Date **expiration**; } |

* JWT工具类

|  |
| --- |
| **package** com.qf.utils;  **import** com.qf.pojo.Payload; **import** io.jsonwebtoken.Claims; **import** io.jsonwebtoken.Jws; **import** io.jsonwebtoken.Jwts; **import** io.jsonwebtoken.SignatureAlgorithm; **import** org.joda.time.DateTime;  **import** java.security.PrivateKey; **import** java.security.PublicKey; **import** java.util.Base64; **import** java.util.UUID;  */\*\*  \** ***@author:*** *krisswen  \* 生成token以及校验token相关方法  \*/* **public class** JwtUtils {   **private static final** String ***JWT\_PAYLOAD\_USER\_KEY*** = **"user"**;   */\*\*  \* 私钥加密token  \*  \** ***@param userInfo*** *载荷中的数据  \** ***@param privateKey*** *私钥  \** ***@param expire*** *过期时间，单位分钟  \** ***@return*** *JWT  \*/* **public static** String generateTokenExpireInMinutes(Object userInfo, PrivateKey privateKey, **int** expire) {  **return** Jwts.*builder*()  .claim(***JWT\_PAYLOAD\_USER\_KEY***, JsonUtils.*toString*(userInfo))  .setId(*createJTI*())  .setExpiration(DateTime.*now*().plusMinutes(expire).toDate())  .signWith(privateKey, SignatureAlgorithm.***RS256***)  .compact();  }   */\*\*  \* 私钥加密token  \*  \** ***@param userInfo*** *载荷中的数据  \** ***@param privateKey*** *私钥  \** ***@param expire*** *过期时间，单位秒  \** ***@return*** *JWT  \*/* **public static** String generateTokenExpireInSeconds(Object userInfo, PrivateKey privateKey, **int** expire) {  **return** Jwts.*builder*()  .claim(***JWT\_PAYLOAD\_USER\_KEY***, JsonUtils.*toString*(userInfo))  .setId(*createJTI*())  .setExpiration(DateTime.*now*().plusSeconds(expire).toDate())  .signWith(privateKey, SignatureAlgorithm.***RS256***)  .compact();  }   */\*\*  \* 公钥解析token  \*  \** ***@param token*** *用户请求中的token  \** ***@param publicKey*** *公钥  \** ***@return*** *Jws*<*Claims*>  *\*/* **private static** Jws<Claims> parserToken(String token, PublicKey publicKey) {  **return** Jwts.*parser*().setSigningKey(publicKey).parseClaimsJws(token);  }   **private static** String createJTI() {  **return new** String(Base64.*getEncoder*().encode(UUID.*randomUUID*().toString().getBytes()));  }   */\*\*  \* 获取token中的用户信息  \*  \** ***@param token*** *用户请求中的令牌  \** ***@param publicKey*** *公钥  \** ***@return*** *用户信息  \*/* **public static** <T> Payload<T> getInfoFromToken(String token, PublicKey publicKey, Class<T> userType) {  Jws<Claims> claimsJws = *parserToken*(token, publicKey);  Claims body = claimsJws.getBody();  Payload<T> claims = **new** Payload<>();  claims.setId(body.getId());  claims.setUserInfo(JsonUtils.*toBean*(body.get(***JWT\_PAYLOAD\_USER\_KEY***).toString(), userType));  claims.setExpiration(body.getExpiration());  **return** claims;  }   */\*\*  \* 获取token中的载荷信息  \*  \** ***@param token*** *用户请求中的令牌  \** ***@param publicKey*** *公钥  \** ***@return*** *用户信息  \*/* **public static** <T> Payload<T> getInfoFromToken(String token, PublicKey publicKey) {  Jws<Claims> claimsJws = *parserToken*(token, publicKey);  Claims body = claimsJws.getBody();  Payload<T> claims = **new** Payload<>();  claims.setId(body.getId());  claims.setExpiration(body.getExpiration());  **return** claims;  } } |

* RSA工具类

|  |
| --- |
| **package** com.qf.utils;  **import** java.io.File; **import** java.io.IOException; **import** java.nio.file.Files; **import** java.security.\*; **import** java.security.spec.InvalidKeySpecException; **import** java.security.spec.PKCS8EncodedKeySpec; **import** java.security.spec.X509EncodedKeySpec; **import** java.util.Base64;  */\*\*  \** ***@author*** *krisswen  \*/* **public class** RsaUtils {   **private static final int *DEFAULT\_KEY\_SIZE*** = 2048;  */\*\*  \* 从文件中读取公钥  \*  \** ***@param filename*** *公钥保存路径，相对于classpath  \** ***@return*** *公钥对象  \** ***@throws*** *Exception  \*/* **public static** PublicKey getPublicKey(String filename) **throws** Exception {  **byte**[] bytes = *readFile*(filename);  **return** *getPublicKey*(bytes);  }   */\*\*  \* 从文件中读取密钥  \*  \** ***@param filename*** *私钥保存路径，相对于classpath  \** ***@return*** *私钥对象  \** ***@throws*** *Exception  \*/* **public static** PrivateKey getPrivateKey(String filename) **throws** Exception {  **byte**[] bytes = *readFile*(filename);  **return** *getPrivateKey*(bytes);  }   */\*\*  \* 获取公钥  \*  \** ***@param bytes*** *公钥的字节形式  \** ***@return*** *\** ***@throws*** *Exception  \*/* **private static** PublicKey getPublicKey(**byte**[] bytes) **throws** Exception {  bytes = Base64.*getDecoder*().decode(bytes);  X509EncodedKeySpec spec = **new** X509EncodedKeySpec(bytes);  KeyFactory factory = KeyFactory.*getInstance*(**"RSA"**);  **return** factory.generatePublic(spec);  }   */\*\*  \* 获取密钥  \*  \** ***@param bytes*** *私钥的字节形式  \** ***@return*** *\** ***@throws*** *Exception  \*/* **private static** PrivateKey getPrivateKey(**byte**[] bytes) **throws** NoSuchAlgorithmException, InvalidKeySpecException {  bytes = Base64.*getDecoder*().decode(bytes);  PKCS8EncodedKeySpec spec = **new** PKCS8EncodedKeySpec(bytes);  KeyFactory factory = KeyFactory.*getInstance*(**"RSA"**);  **return** factory.generatePrivate(spec);  }   */\*\*  \* 根据密文，生存rsa公钥和私钥,并写入指定文件  \*  \** ***@param publicKeyFilename*** *公钥文件路径  \** ***@param privateKeyFilename*** *私钥文件路径  \** ***@param secret*** *盐 生成密钥的密文  \*/* **public static void** generateKey(String publicKeyFilename, String privateKeyFilename, String secret, **int** keySize) **throws** Exception {  KeyPairGenerator keyPairGenerator = KeyPairGenerator.*getInstance*(**"RSA"**);  SecureRandom secureRandom = **new** SecureRandom(secret.getBytes());  keyPairGenerator.initialize(Math.*max*(keySize, ***DEFAULT\_KEY\_SIZE***), secureRandom);  KeyPair keyPair = keyPairGenerator.genKeyPair();  *// 获取公钥并写出* **byte**[] publicKeyBytes = keyPair.getPublic().getEncoded();  publicKeyBytes = Base64.*getEncoder*().encode(publicKeyBytes);  *writeFile*(publicKeyFilename, publicKeyBytes);  *// 获取私钥并写出* **byte**[] privateKeyBytes = keyPair.getPrivate().getEncoded();  privateKeyBytes = Base64.*getEncoder*().encode(privateKeyBytes);  *writeFile*(privateKeyFilename, privateKeyBytes);  }   **private static byte**[] readFile(String fileName) **throws** Exception {  **return** Files.*readAllBytes*(**new** File(fileName).toPath());  }   **private static void** writeFile(String destPath, **byte**[] bytes) **throws** IOException {  File dest = **new** File(destPath);  **if** (!dest.exists()) {  dest.createNewFile();  }  Files.*write*(dest.toPath(), bytes);  } } |

# 3 SpringSecurity+JWT+RSA分布式认证

SpringSecurity主要是通过过滤器来实现功能的！我们要找到SpringSecurity实现认证和校验身份的过滤器！

## 3.1 思路分析

* **用户认证：**

由于分布式项目，多数是前后端分离的架构设计，我们要满足可以接受异步post的认证请求参数，需要修改UsernamePasswordAuthenticationFilter过滤器中attemptAuthentication方法，让其能够接收请求体。另外，默认successfulAuthentication方法在认证通过后，是把用户信息直接放入session就完事了，现在我 们需要修改这个方法，在认证通过后生成token并返回给用户。

* **身份校验：**

原来BasicAuthenticationFilter过滤器中doFilterInternal方法校验用户是否登录，就是看session中是否有用 户信息，我们要修改为，验证用户携带的token是否合法，并解析出用户信息，交给SpringSecurity，以便于后续的授权功能可以正常使用。

## 3.2 代码实现

### 3.2.1 创建父工程

使用maven构建父工程: springboot-springSecurity

在父工程中导入依赖。

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.1.3.RELEASE</**version**>  <**relativePath**/> </**parent**> |

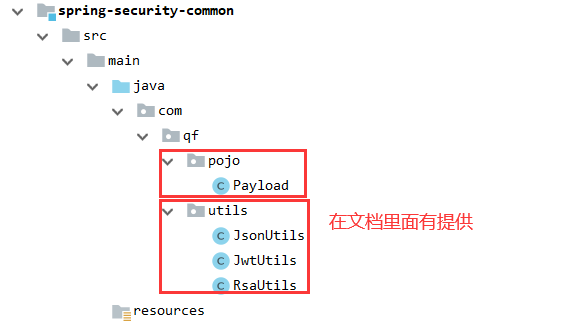
### 3.2.2 创建通用模块

右键父工程，创建子模块: spring-security-common

* 并导入JWT相关jar包。

|  |
| --- |
| <**dependencies**>  *<!--jwt所需jar包-->* <**dependency**>  <**groupId**>io.jsonwebtoken</**groupId**>  <**artifactId**>jjwt-api</**artifactId**>  <**version**>0.10.7</**version**>  </**dependency**>  <**dependency**>  <**groupId**>io.jsonwebtoken</**groupId**>  <**artifactId**>jjwt-impl</**artifactId**>  <**version**>0.10.7</**version**>  <**scope**>runtime</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>io.jsonwebtoken</**groupId**>  <**artifactId**>jjwt-jackson</**artifactId**>  <**version**>0.10.7</**version**>  <**scope**>runtime</**scope**>  </**dependency**>  *<!--lombok插件-->* <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  </**dependency**>  *<!--处理日期工具包-->* <**dependency**>  <**groupId**>joda-time</**groupId**>  <**artifactId**>joda-time</**artifactId**>  </**dependency**>  *<!--处理json工具包-->* <**dependency**>  <**groupId**>com.fasterxml.jackson.core</**groupId**>  <**artifactId**>jackson-databind</**artifactId**>  <**version**>2.9.9</**version**>  </**dependency**>  *<!--日志包-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  </**dependency**>  *<!--测试包-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  </**dependency**> </**dependencies**> |

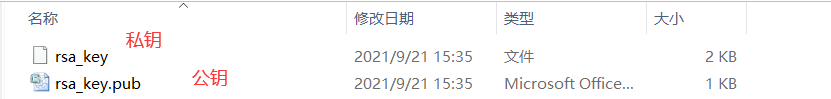
* 导入工具类



* 在通用模块的测试类中编写测试类生成rsa公钥和私钥。

|  |
| --- |
| **public class** RsaUtilsTest {   **private** String **publicFile** = **"D:\\auth\_key\\rsa\_key.pub"**;  **private** String **privateFile** = **"D:\\auth\_key\\rsa\_key"**;   */\*\*  \* 生成公钥和私钥  \** ***@throws*** *Exception  \*/* @Test  **public void** generateKey() **throws** Exception{  RsaUtils.*generateKey*(**publicFile**, **privateFile**, **"eric"**, 2048);  }   */\*\*  \* 获取公钥  \** ***@throws*** *Exception  \*/* @Test  **public void** getPublicKey() **throws** Exception{  PublicKey publicKey = RsaUtils.*getPublicKey*(**publicFile**);  System.***out***.println(publicKey);  }    @Test  **public void** getPrivateKey() **throws** Exception{  PrivateKey privateKey = RsaUtils.*getPrivateKey*(**privateFile**);  System.***out***.println(privateKey);  } } |

查看公钥和私钥:



### 3.2.4 创建认证模块

* 导入相关依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-security</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>com.qf.common</**groupId**>  <**artifactId**>spring-security-common</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.47</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>2.1.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**>  </**dependency**>  <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper-spring-boot-starter</**artifactId**>  <**version**>1.2.3</**version**>  </**dependency**> </**dependencies**> |

* 创建认证服务配置文件

|  |
| --- |
| **server**:  **port**: 9001 **spring**:  **datasource**:  **driver-class-name**: com.mysql.jdbc.Driver  **url**: jdbc:mysql://192.168.10.137:3306/hotel  **username**: root  **password**: Admin123! **mybatis**:  **type-aliases-package**: com.qf.pojo  **configuration**:  **map-underscore-to-camel-case**: **true logging**:  **level**:  **com.qf**: debug **eric**:  **key**:  **pubKeyPath**: D:\\auth\_key\\rsa\_key.pub  **priKeyPath**: D:\\auth\_key\\rsa\_key |

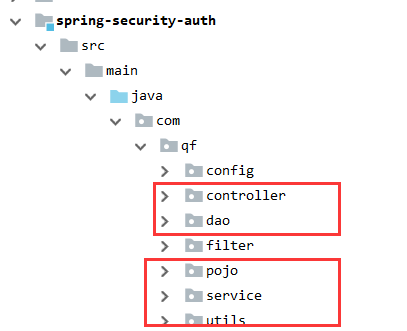
* 提供解析公钥和私钥的配置类

|  |
| --- |
| @Data @ConfigurationProperties(prefix = **"eric.key"**) **public class** RsaKeyProperties {   **private** String **pubKeyPath**;  **private** String **priKeyPath**;   **private** PublicKey **publicKey**;  **private** PrivateKey **privateKey**;   */\*\*  \* 在RsaKeyProperties的bean的实例创建之后，并且属性注入完毕之后。  \* 执行一个方法。  \** ***@throws*** *Exception  \*/* @PostConstruct  **public void** loadKey() **throws** Exception {  **publicKey** = RsaUtils.*getPublicKey*(**pubKeyPath**);  **privateKey** = RsaUtils.*getPrivateKey*(**priKeyPath**);  }  } |

* 创建认证服务启动类

|  |
| --- |
| @MapperScan(**"com.qf.dao"**) @EnableConfigurationProperties(RsaKeyProperties.**class**) @SpringBootApplication **public class** AuthApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(AuthApplication.**class**,args);  } } |

* 将之前案例的相关代码拷贝到工程中



处理器类上换成@RestController，这里前后端绝对分离，不能再跳转页面了，要返回数据。

注意:UserServiceImpl里面的认证方法需要修改:

|  |
| --- |
| @Override **public** UserDetails loadUserByUsername(String s) **throws** UsernameNotFoundException {  **return userDao**.findUserByName(s); } |

因为我们现在使用的是JWT方式的认证。

### 3.2.5 编写认证过滤器

|  |
| --- |
| **public class** TokenLoginFilter **extends** UsernamePasswordAuthenticationFilter {   **private** AuthenticationManager **authenticationManager**;   **private** RsaKeyProperties **prop**;   **public** TokenLoginFilter(AuthenticationManager authenticationManager, RsaKeyProperties prop) {  **this**.**authenticationManager** = authenticationManager;  **this**.**prop** = prop;  }   */\*\*  \* 接收并解析用户凭证，出現错误时，返回json数据前端  \** ***@param request*** *\** ***@param response*** *\** ***@return*** *\** ***@throws*** *AuthenticationException  \*/* @Override  **public** Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) **throws** AuthenticationException {  **try** {  UserInfo userInfo = **new** ObjectMapper().readValue(request.getInputStream(), UserInfo.**class**);  *//生成token* UsernamePasswordAuthenticationToken token = **new** UsernamePasswordAuthenticationToken(userInfo.getUsername(),userInfo.getPassword());  **return authenticationManager**.authenticate(token);  } **catch** (IOException e) {  *//如果认证失败，提供自定义json格式异常* **try** {  response.setContentType(**"application/json;charset=utf-8"**);  response.setStatus(HttpServletResponse.***SC\_UNAUTHORIZED***);  PrintWriter out = response.getWriter();  Map<String, Object> map = **new** HashMap<String, Object>();  map.put(**"code"**, HttpServletResponse.***SC\_UNAUTHORIZED***);  map.put(**"message"**, **"账号或密码错误！"**);  out.write(**new** ObjectMapper().writeValueAsString(map));  out.flush();  out.close();  } **catch** (IOException e1) {  e1.printStackTrace();  }  **throw new** RuntimeException(e);  }  }   */\*\*  \* 用户登录成功后，生成token,并且返回json数据给前端  \** ***@param request*** *\** ***@param response*** *\** ***@param chain*** *\** ***@param auth*** *\** ***@throws*** *IOException  \** ***@throws*** *ServletException  \*/* @Override  **protected void** successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication auth) **throws** IOException, ServletException {  UserInfo userInfo = **new** UserInfo();  userInfo.setUsername(auth.getName());  userInfo.setRoles((List<Role>) auth.getAuthorities());  String token = JwtUtils.*generateTokenExpireInMinutes*(userInfo, **prop**.getPrivateKey(), 24 \* 60);  response.addHeader(**"Authorization"**, **"Bearer "** + token);  **try** {  *//登录成功時，返回json格式进行提示* response.setContentType(**"application/json;charset=utf-8"**);  response.setStatus(HttpServletResponse.***SC\_OK***);  PrintWriter out = response.getWriter();  Map<String, Object> map = **new** HashMap<String, Object>();  map.put(**"code"**, HttpServletResponse.***SC\_OK***);  map.put(**"message"**, **"登陆成功！"**);  out.write(**new** ObjectMapper().writeValueAsString(map));  out.flush();  out.close();  } **catch** (Exception e) {  e.printStackTrace();  }   } } |

### 3.2.6 编写检验token过滤器

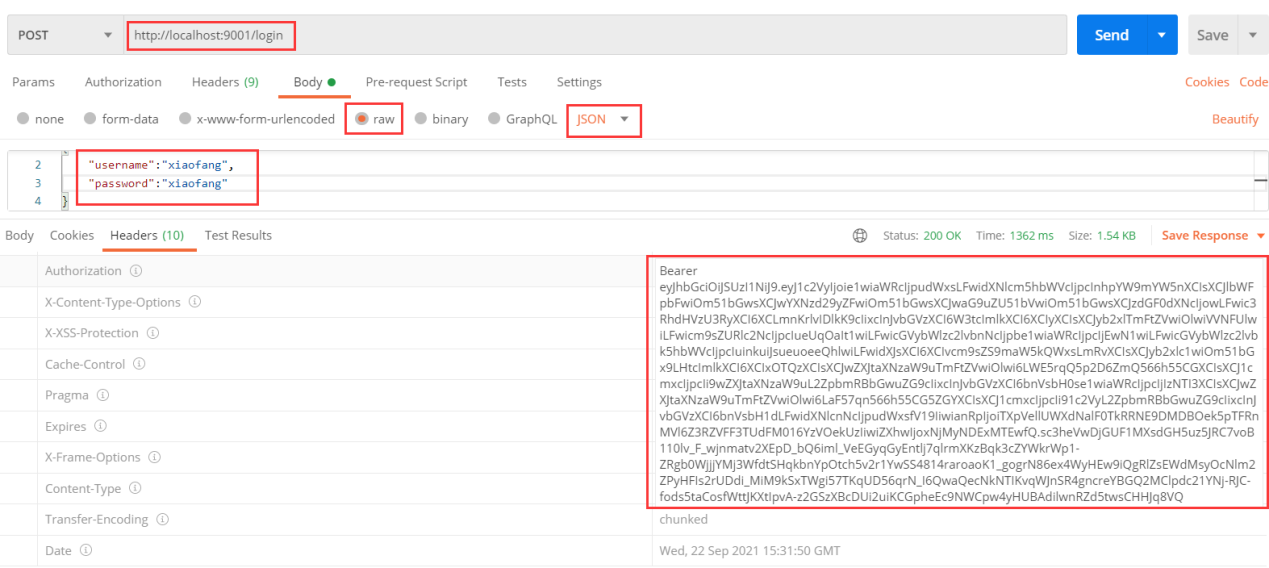
|  |
| --- |
| **public class** TokenVerifyFilter **extends** BasicAuthenticationFilter {   **private** RsaKeyProperties **prop**;   **public** TokenVerifyFilter(AuthenticationManager authenticationManager, RsaKeyProperties prop){  **super**(authenticationManager);  **this**.**prop** = prop;  }   @Override  **protected void** doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) **throws** IOException, ServletException {  String token = request.getHeader(**"Authorization"**);  **if**(token == **null** || !token.startsWith(**"Bearer "**)){  chain.doFilter(request, response);  **try** {  *//未登录提示* response.setContentType(**"application/json;charset=utf-8"**);  response.setStatus(HttpServletResponse.***SC\_FORBIDDEN***);  PrintWriter out = response.getWriter();  Map<String, Object> map = **new** HashMap<String, Object>();  map.put(**"code"**, HttpServletResponse.***SC\_FORBIDDEN***);  map.put(**"message"**, **"请登录！"**);  out.write(**new** ObjectMapper().writeValueAsString(map));  out.flush();  out.close();  } **catch** (Exception e1) {  e1.printStackTrace();  }  }**else**{  *//通过token解析出载荷信息* Payload<UserInfo> payload = JwtUtils.*getInfoFromToken*(token.replace(**"Bearer "**, **""**),  **prop**.getPublicKey(), UserInfo.**class**);  UserInfo userInfo = payload.getUserInfo();  UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = **new** UsernamePasswordAuthenticationToken(userInfo, **null**, (Collection<? **extends** GrantedAuthority>) userInfo.getAuthorities());  *//获取后，将Authentication写入SecurityContextHolder中供后序使用* SecurityContextHolder.*getContext*().setAuthentication(usernamePasswordAuthenticationToken);  chain.doFilter(request, response);   }  } } |

### 3.2.7 编写SpringSecurity配置类

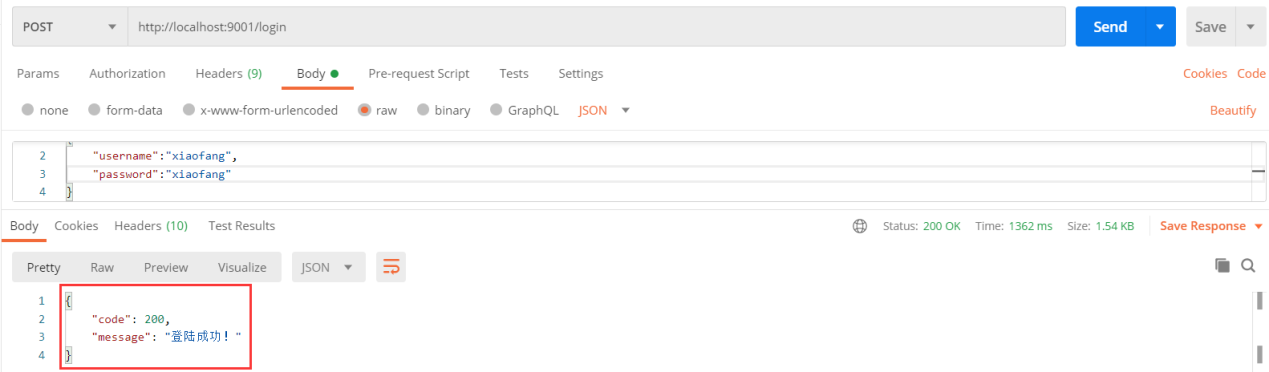
|  |
| --- |
| @Configuration @EnableWebSecurity @EnableGlobalMethodSecurity(securedEnabled = **true**) **public class** WebSecurityConfig **extends** WebSecurityConfigurerAdapter {   @Autowired  **private** UserDetailsService **myCustomUserService**;   @Autowired  **private** RsaKeyProperties **prop**;   @Bean  **public** BCryptPasswordEncoder myPasswordEncoder(){  **return new** BCryptPasswordEncoder();  }   *//释放拦截资源，指定资源拦截规则，指定自定义认证页面，指定退出认证配置，csrf配置* @Override  **public void** configure(HttpSecurity http) **throws** Exception {  http  *//关闭跨站请求防护* .cors().and().csrf().disable()  *//访问product 需要user角色* .authorizeRequests().antMatchers(**"/product"**).hasAnyRole(**"USER"**)  *//其他的需要授权后访问* .anyRequest().authenticated()  .and()  *//增加自定义认证过滤器* .addFilter(**new** TokenLoginFilter(authenticationManager(), **prop**))  *//增加自定义验证认证过滤器* .addFilter(**new** TokenVerifyFilter(authenticationManager(), **prop**))  *// 前后端分离是无状态的，不用session了，直接禁用。* .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);  }   *//认证用户的来源* @Override  **public void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  *//UserDetailsService类* auth.userDetailsService(**myCustomUserService**)  *//加密策略* .passwordEncoder(myPasswordEncoder());  } } |

### 3.2.8 启动测试认证服务

使用POSTMAN测试：



认证通过结果:



验证认证请求:

