SpringSecurity整合SpringBoot集中式版

技术选型

SpringBoot2.1.3，SpringSecurity，MySQL，mybatis，jsp

初步整合认证第一版

创建工程并导入jar包

先只导入SpringBoot

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

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

提供处理器

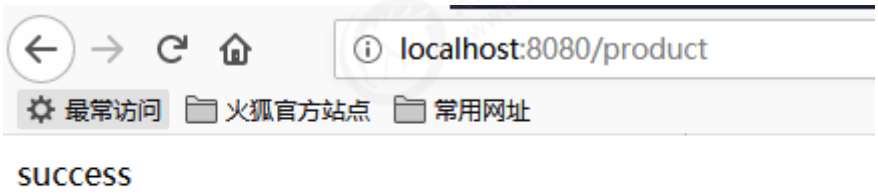
```
@Controller
@RequestMapping("/product")
public class ProductController {

    @RequestMapping
    @ResponseBody
    public String hello(){
        return "success";
    }

}
```

编写启动类

```
@SpringBootApplication
public class SecurityApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecurityApplication.class, args);
    }
}
```

测试效果

使用SpringBoot内置tomcat启动项目，即可访问处理器。

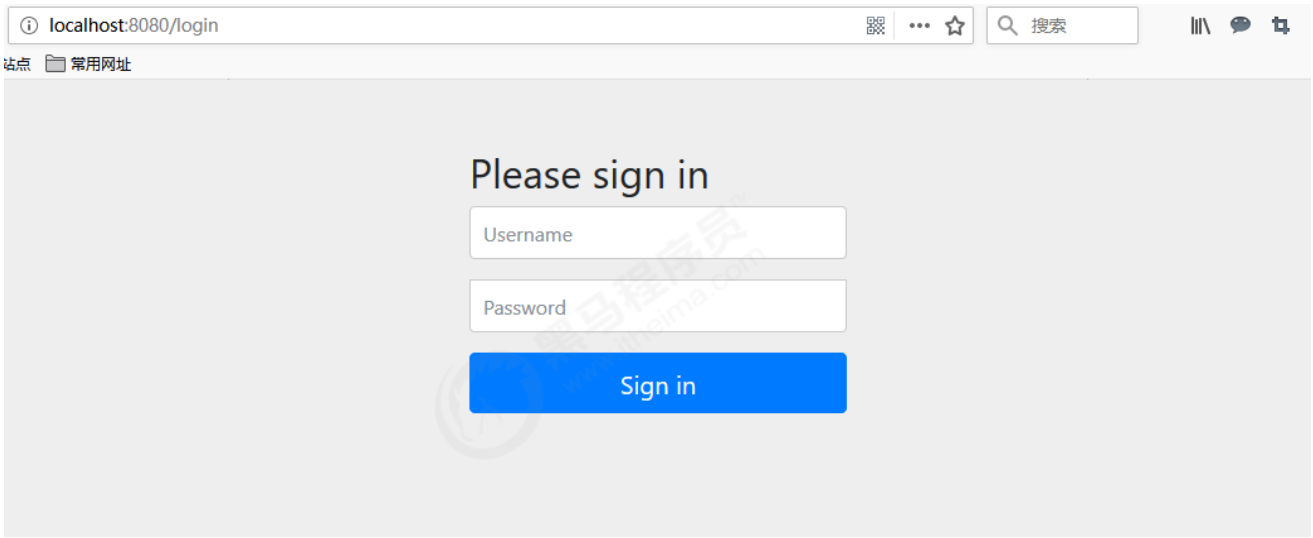


加入SpringSecurity的jar包

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

重启再次测试

SpringBoot已经为SpringSecurity提供了默认配置，默认所有资源都必须认证通过才能访问。

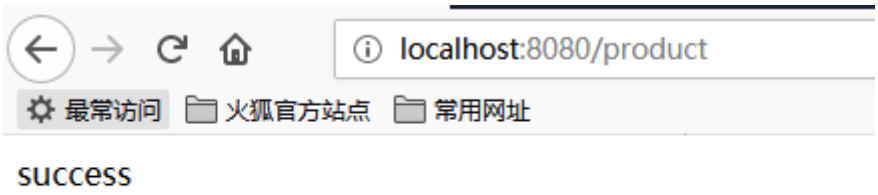


那么问题来了！此刻并没有连接数据库，也并未在内存中指定认证用户，如何认证呢？

其实SpringBoot已经提供了默认用户名user，密码在项目启动时随机生成，如图：

```
Console  Endpoints
  Initializing ExecutorService 'applicationTaskExecutor'
2019-09-22 10:19:42.649  INFO 9068 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 7244ff88-a9db-423a-8489-0714ace5640c

2019-09-22 10:19:42.773  INFO 9068 --- [           main] o.s.s.web.DefaultSecurityFilterChain     :
 Creating filter chain: any request, [org.springframework.security.web.context.request.async
 .WebAsyncManagerIntegrationFilter@30feffc, org.springframework.security.web.context
 .SecurityContextPersistenceFilter@3e6f3bae, org.springframework.security.web.header
 .HeaderWriterFilter@7e97551f, org.springframework.security.web.csrf.CsrfFilter@35e478f, org
 .springframework.security.web.authentication.logout.LogoutFilter@29f0802c, org.springframework.security
 .web.authentication.UsernamePasswordAuthenticationFilter@74fef3f7, org.springframework.security.web
```

认证通过后可以继续访问处理器资源：



整合认证第二版【加入jsp使用自定义认证页面】

说明

SpringBoot官方是不推荐在SpringBoot中使用jsp的，那么到底可以使用吗？答案是肯定的！

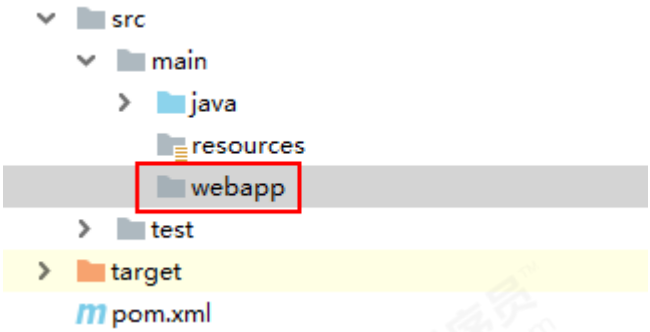
不过需要导入tomcat插件启动项目，不能再用SpringBoot默认tomcat了。

导入SpringBoot的tomcat启动插件jar包

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

加入jsp页面等静态资源

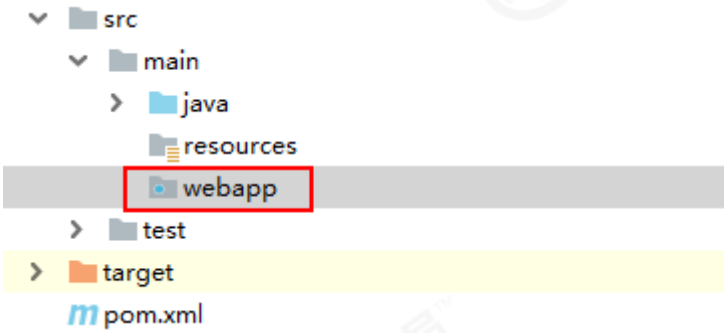
在src/main目录下创建webapp目录



这时webapp目录并不能正常使用，因为只有web工程才有webapp目录，在pom文件中修改项目为web工程

```
<groupId>com.itheima</groupId>
<artifactId>springboot_security_01</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

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

这时webapp目录，可以正常使用了！



导入第一天案例中静态资源，注意WEB-INF就不用了哈！

修改login.jsp中认证的url地址

```
<!-- /.login-logo -->
<div class="login-box-body">
    <p class="login-box-msg">登录系统</p>
    <form action="${pageContext.request.contextPath}/login" method="post">
        <div class="form-group has-feedback">
            <input type="text" name="username" class="form-control"
                placeholder="用户名"> <span
                class="glyphicon glyphicon-envelope form-control-feedback"></span>
        </div>
```

修改header.jsp中退出登录的url地址

```
<!-- Menu Footer-->
<li class="user-footer">
    <div class="pull-left">
        <a href="#" class="btn btn-default btn-flat">修改密码</a>
    </div>
    <div class="pull-right">
        <a href="${pageContext.request.contextPath}/logout"
            class="btn btn-default btn-flat">注销</a>
    </div>
</li>
```

提供SpringSecurity配置类

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    /**
     * 这里先不连接数据库了
     */
```

```
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user")
            .password("{noop}123")
            .roles("USER");
    }

    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/login.jsp", "/failer.jsp", "/css/**", "/img/**",
"/plugins/**").permitAll()
                .antMatchers("/**").hasAnyRole("USER")
                .anyRequest()
                .authenticated()
                .and()
                .formLogin()
                .loginPage("/login.jsp")
                .loginProcessingUrl("/login")
                .successForwardUrl("/index.jsp")
                .failureForwardUrl("/failer.jsp")
                .permitAll()
                .and()
                .logout()
                .logoutUrl("/logout")
                .invalidateHttpSession(true)
                .logoutSuccessUrl("/login.jsp")
                .permitAll()
                .and()
                .csrf()
                .disable();
    }
}
```

修改产品处理器

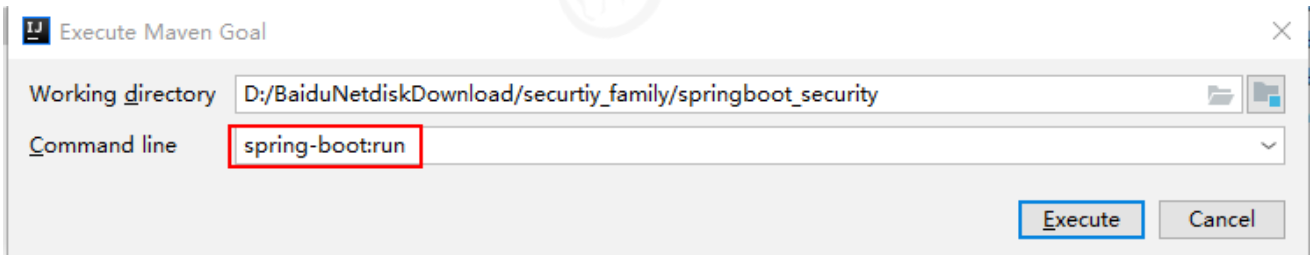
有页面了，就跳转一个真的吧！

```
@Controller
@RequestMapping("/product")
public class ProductController {

    @RequestMapping("/findAll")
    public String findAll(){
        return "product-list";
    }

}
```

配置视图解析器

```yaml
spring:
  mvc:
    view:
      prefix: /pages/
      suffix: .jsp
```

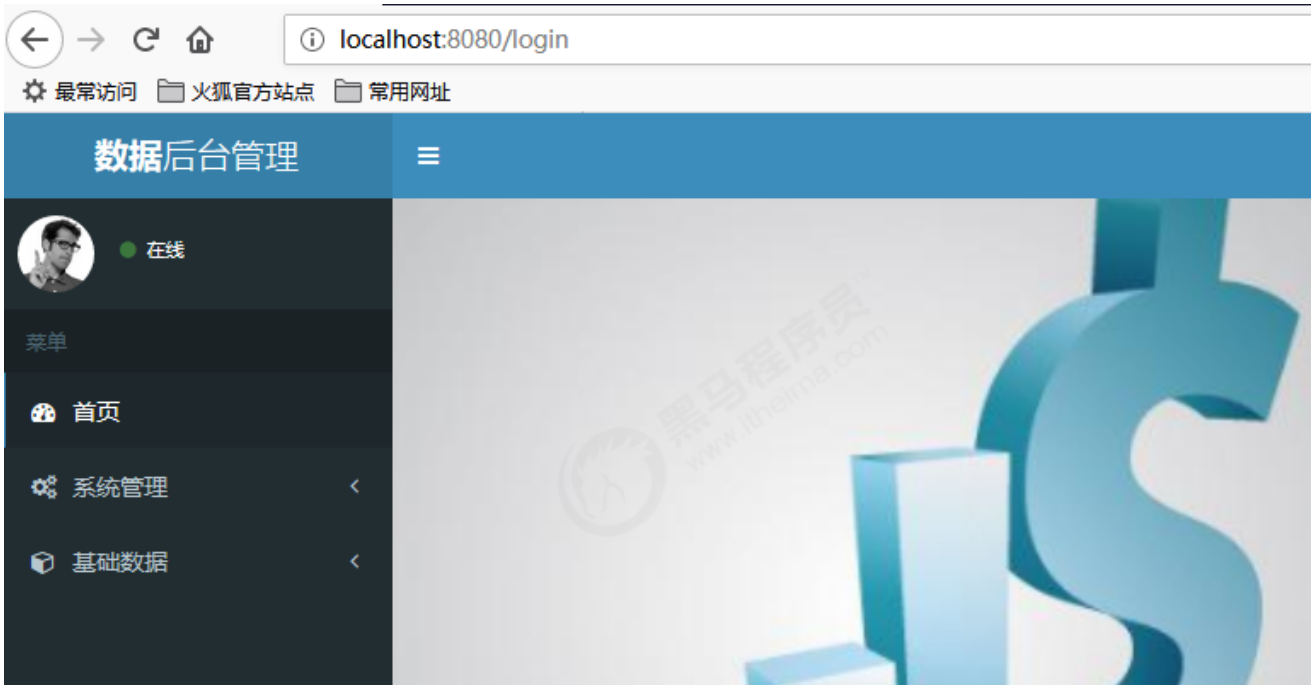
使用tomcat插件启动项目



测试效果

自定义的认证页面

认证成功页面



整合认证第三版【数据库认证】

数据库环境准备

依然使用security_authority数据库，sql语句在第一天资料里。

导入数据库操作相关jar包

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper-spring-boot-starter</artifactId>
    <version>2.1.5</version>
</dependency>
```

在配置文件中添加数据库操作相关配置

application.yml

```yaml
spring:
  mvc:
    view:
      prefix: /pages/
      suffix: .jsp
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///security_authority
    username: root
    password: root
mybatis:
  type-aliases-package: com.itheima.domain
  configuration:
    map-underscore-to-camel-case: true
logging:
  level:
    com.itheima: debug
```

在启动类上添加扫描dao接口包注解

```java
@SpringBootApplication
@MapperScan("com.itheima.mapper")
public class SecurityApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecurityApplication.class, args);
    }
}
```

创建角色pojo对象

这里直接使用SpringSecurity的角色规范

```java
public class SysRole implements GrantedAuthority {
    private Integer id;
    private String roleName;
    private String roleDesc;

    public Integer getId() {

        return id;
```

```
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getRoleName() {
        return roleName;
    }

    public void setRoleName(String roleName) {
        this.roleName = roleName;
    }

    public String getRoleDesc() {
        return roleDesc;
    }

    public void setRoleDesc(String roleDesc) {
        this.roleDesc = roleDesc;
    }

    //标记此属性不做json处理
    @JsonIgnore
    @Override
    public String getAuthority() {
        return roleName;
    }

}
```

创建用户pojo对象

这里直接实现SpringSecurity的用户对象接口，并添加角色集合私有属性。

注意接口属性都要标记不参与json的处理。

```
public class SysUser implements UserDetails {

    private Integer id;
    private String username;
    private String password;
    private Integer status;
    private List<SysRole> roles = new ArrayList<>();

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {

        this.id = id;
```

```
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Integer getStatus() {
        return status;
    }

    public void setStatus(Integer status) {
        this.status = status;
    }

    public List<SysRole> getRoles() {
        return roles;
    }

    public void setRoles(List<SysRole> roles) {
        this.roles = roles;
    }

    @JsonIgnore
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return roles;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @JsonIgnore
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @JsonIgnore
    @Override
    public boolean isAccountNonLocked() {
        return true;

    }
```

```
    @JsonIgnore
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @JsonIgnore
    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

提供角色mapper接口

```
public interface RoleMapper extends Mapper<SysRole> {

    @Select("SELECT r.id, r.role_name roleName, r.role_desc roleDesc " +
            "FROM sys_role r, sys_user_role ur " +
            "WHERE r.id=ur.rid AND ur.uid=#{uid}")
    public List<SysRole> findByUid(Integer uid);

}
```

提供用户mapper接口

```
public interface UserMapper extends Mapper<SysUser> {

    @Select("select * from sys_user where username=#{username}")
    @Results({
            @Result(id = true, property = "id", column = "id"),
            @Result(property = "roles", column = "id", javaType = List.class,
                many = @Many(select = "com.itheima.mapper.RoleMapper.findByUid"))
    })
    public SysUser findByUsername(String username);

}
```

提供认证service接口

```
package com.itheima.service;

import org.springframework.security.core.userdetails.UserDetailsService;

public interface UserService extends UserDetailsService {
}
```

提供认证service实现类

```
@Service
@Transactional
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper userMapper;

    @Override
    public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {
        return userMapper.findByUsername(s);
    }

}
```

在启动类中把加密对象放入IOC容器

```
@SpringBootApplication
@MapperScan("com.itheima.mapper")
public class SecurityApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecurityApplication.class, args);
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}
```

修改配置类

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserService userService;

    @Autowired
    private BCryptPasswordEncoder passwordEncoder;

    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder);
    }

    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
```

```
                .antMatchers("/login.jsp", "/failer.jsp", "/css/**", "/img/**",
"/plugins/**").permitAll()
                .antMatchers("/**").hasAnyRole("USER")
                .anyRequest()
                .authenticated()
                .and()
                .formLogin()
                .loginPage("/login.jsp")
                .loginProcessingUrl("/login")
                .successForwardUrl("/index.jsp")
                .failureForwardUrl("/failer.jsp")
                .permitAll()
                .and()
                .logout()
                .logoutUrl("/logout")
                .invalidateHttpSession(true)
                .logoutSuccessUrl("/login.jsp")
                .permitAll()
                .and()
                .csrf()
                .disable();
    }
}
```

大功告成尽管测试

注意还是用插件启动项目，使用数据库表中的用户名和密码。

整合实现授权功能

在启动类上添加开启方法级的授权注解

```
@SpringBootApplication
@MapperScan("com.itheima.mapper")
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecurityApplication.class, args);
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}
```

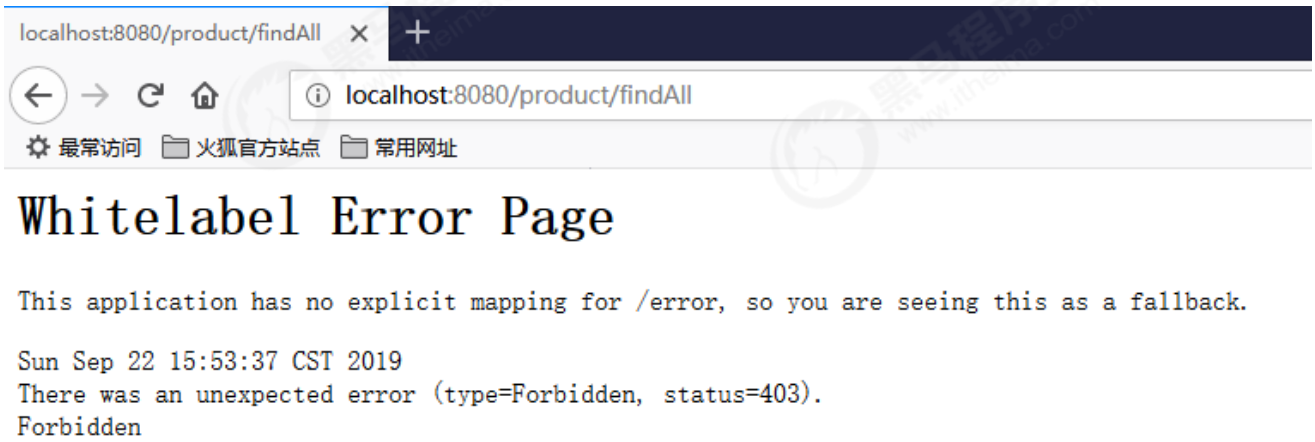
在产品处理器类上添加注解

要求产品列表功能必须具有ROLE_ADMIN角色才能访问!

```
@Controller
@RequestMapping("/product")
public class ProductController {

    @Secured("ROLE_ADMIN")
    @RequestMapping("/findAll")
    public String findAll(){
        return "product-list";
    }

}
```

重启项目测试

再次访问产品列表发现权限不足



指定自定义异常页面

编写异常处理器拦截403异常

```
@ControllerAdvice
public class HandleControllerException {

    @ExceptionHandler(RuntimeException.class)
    public String exceptionHandler(RuntimeException e){
        if(e instanceof AccessDeniedException){
            //如果是权限不足异常，则跳转到权限不足页面!
```

```
            return "redirect:/403.jsp";
        }
        //其余的异常都到500页面!
        return "redirect:/500.jsp";
    }

}
```

再次测试产品列表就可以到自定义异常页面了

SpringSecurity整合SpringBoot分布式版

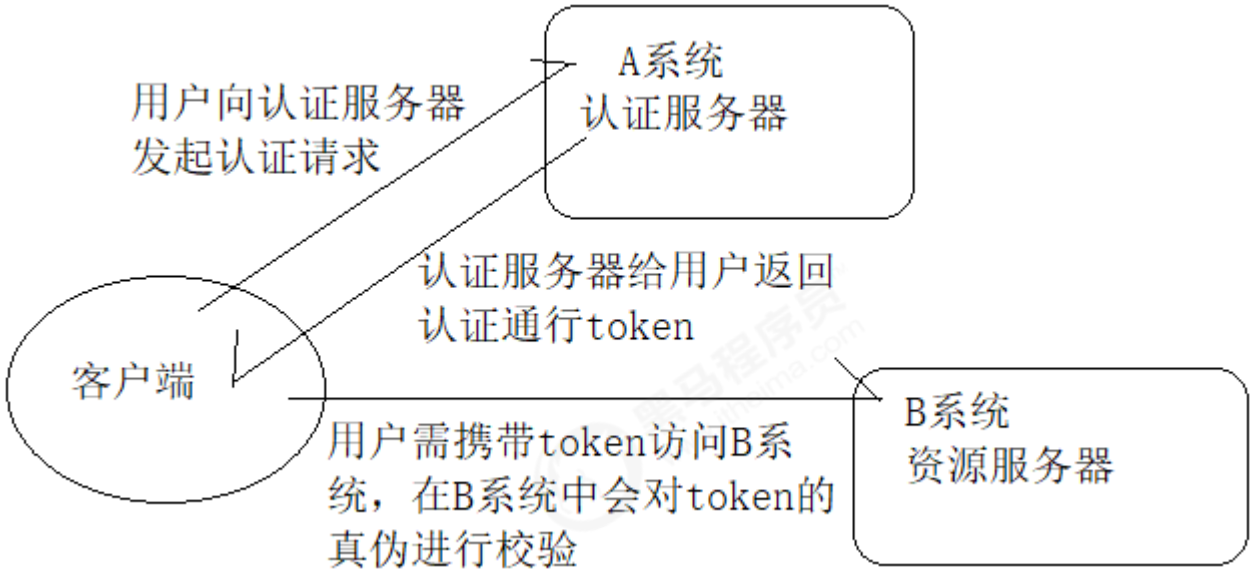
分布式认证概念说明

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。

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。

JWT介绍

概念说明

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。

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！

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，持有私钥或公钥才可以解密
  - 公钥加密，持有私钥才可解密
- 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。

JWT相关工具类

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 黑马程序员
 * 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

```
/**
```

```
 * @author: 黑马程序员
 * 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

```
/**
 * @author 黑马程序员
 */
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             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);
    }
}
```

SpringSecurity+JWT+RSA分布式认证思路分析

> 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！

回顾集中式认证流程

- 用户认证：

  使用UsernamePasswordAuthenticationFilter过滤器中attemptAuthentication方法实现认证功能，该过滤器父类中successfulAuthentication方法实现认证成功后的操作。

- 身份校验：

  使用BasicAuthenticationFilter过滤器中doFilterInternal方法验证是否登录，以决定能否进入后续过滤器。

分析分布式认证流程

- 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。

SpringSecurity+JWT+RSA分布式认证实现

创建父工程并导入jar包

springboot_security_jwt_rsa

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima</groupId>
    <artifactId>springboot_security_jwt_rsa</artifactId>
    <version>1.0-SNAPSHOT</version>

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

</project>
```

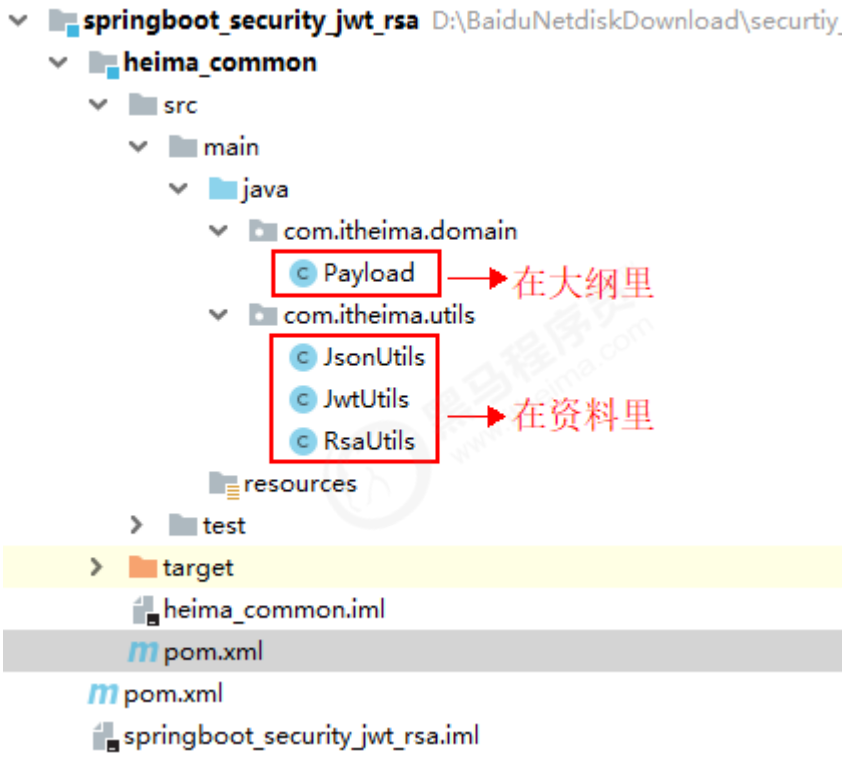
通用模块

创建通用子模块并导入JWT相关jar包

```xml
<parent>
    <artifactId>springboot_security_jwt_rsa</artifactId>
    <groupId>com.itheima</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>heima_common</artifactId>
```

```
<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>
```

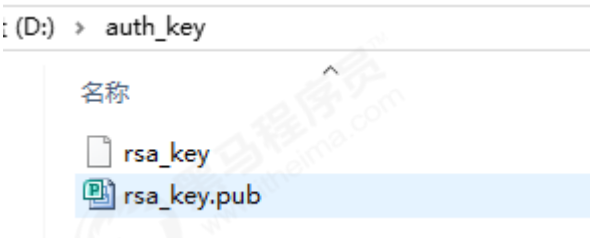
导入工具类



在通用子模块中编写测试类生成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";

    @Test
    public void generateKey() throws Exception {
        RsaUtils.generateKey(publicFile, privateFile, "heima", 2048);
    }
}
```

执行后查看D:\auth_key目录发现私钥和公钥文件生成成功



认证服务

创建认证服务工程并导入jar包

```
<parent>
    <artifactId>springboot_security_jwt_rsa</artifactId>

    <groupId>com.itheima</groupId>
```

```
    <version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>heima_auth_server</artifactId>

<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.itheima</groupId>
        <artifactId>heima_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>
</dependencies>
```

创建认证服务配置文件

```
server:
  port: 9001
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///security_authority
    username: root
    password: root
mybatis:
  type-aliases-package: com.itheima.domain
  configuration:
    map-underscore-to-camel-case: true
logging:
  level:
    com.itheima: debug
heima:
  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

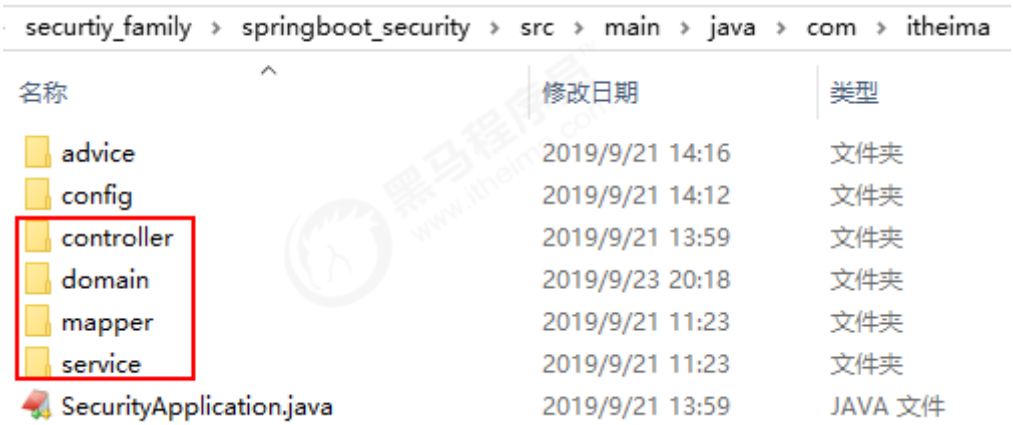
```
@Data
@ConfigurationProperties(prefix = "heima.key")
public class RsaKeyProperties {
    private String pubKeyPath;
    private String priKeyPath;

    private PublicKey publicKey;
    private PrivateKey privateKey;

    @PostConstruct
    public void loadKey() throws Exception {
        publicKey = RsaUtils.getPublicKey(pubKeyPath);
        privateKey = RsaUtils.getPrivateKey(priKeyPath);
    }

}
```

创建认证服务启动类

```
@SpringBootApplication
@MapperScan("com.itheima.mapper")
@EnableConfigurationProperties(RsaKeyProperties.class)
public class AuthApplication {
    public static void main(String[] args) {
        SpringApplication.run(AuthApplication.class, args);
    }
}
```

将上面集中式案例中数据库认证相关代码复制到认证服务中

需要复制的代码如果所示：



注意这里要去掉mapper中继承的通用mapper接口

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。

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，出现错误时，返回json数据前端
     */
    @Override
    public Authentication attemptAuthentication(HttpServletRequest req, HttpServletResponse res)
{
        try {
            //将json格式请求体转成JavaBean对象
            SysUser user = new ObjectMapper().readValue(req.getInputStream(), SysUser.class);
            return authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            user.getUsername(),
                            user.getPassword())
            );
        } catch (Exception e) {
            try {
                //如果认证失败，提供自定义json格式异常
                res.setContentType("application/json;charset=utf-8");
                res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                PrintWriter out = res.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 (Exception 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
     */
    @Override

    protected void successfulAuthentication(HttpServletRequest req, HttpServletResponse res,
```

```
FilterChain chain, Authentication auth) {
        //得到当前认证的用户对象
        SysUser user = new SysUser();
        user.setUsername(auth.getName());
        user.setRoles((List<SysRole>) auth.getAuthorities());
        //json web token构建
        String token = JwtUtils.generateTokenExpireInMinutes(user, prop.getPrivateKey(), 24*60);

        //返回token
        res.addHeader("Authorization", "Bearer " + token);

        try {
            //登录成功时，返回json格式进行提示
            res.setContentType("application/json;charset=utf-8");
            res.setStatus(HttpServletResponse.SC_OK);
            PrintWriter out = res.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 e1) {
            e1.printStackTrace();
        }
    }
}
```

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;
    }


    /**
     * 过滤请求
     */
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain chain) {
        try {
            //请求体的头中是否包含Authorization
            String header = request.getHeader("Authorization");
            //Authorization中是否包含Bearer，不包含直接返回

            if (header == null || !header.startsWith("Bearer ")) {
```

```
                chain.doFilter(request, response);
                responseJson(response);
                return;
            }
            //获取权限失败，会抛出异常
            UsernamePasswordAuthenticationToken authentication = getAuthentication(request);
            //获取后，将Authentication写入SecurityContextHolder中供后序使用
            SecurityContextHolder.getContext().setAuthentication(authentication);
            chain.doFilter(request, response);
        } catch (Exception e) {
            responseJson(response);
            e.printStackTrace();
        }
    }

    /**
     * 未登录提示
     * @param response
     */
    private void responseJson(HttpServletResponse 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();
        }
    }

    /**
     * 通过token，获取用户信息
     *
     * @param request
     * @return
     */
    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
        String token = request.getHeader("Authorization");
        if (token != null) {
            //通过token解析出载荷信息
            Payload<SysUser> payload = JwtUtils.getInfoFromToken(token.replace("Bearer ", ""),
prop.getPublicKey(), SysUser.class);
            SysUser user = payload.getUserInfo();
            //不为null，返回
            if (user != null) {
                return new UsernamePasswordAuthenticationToken(user, null, user.getRoles());

            }
```

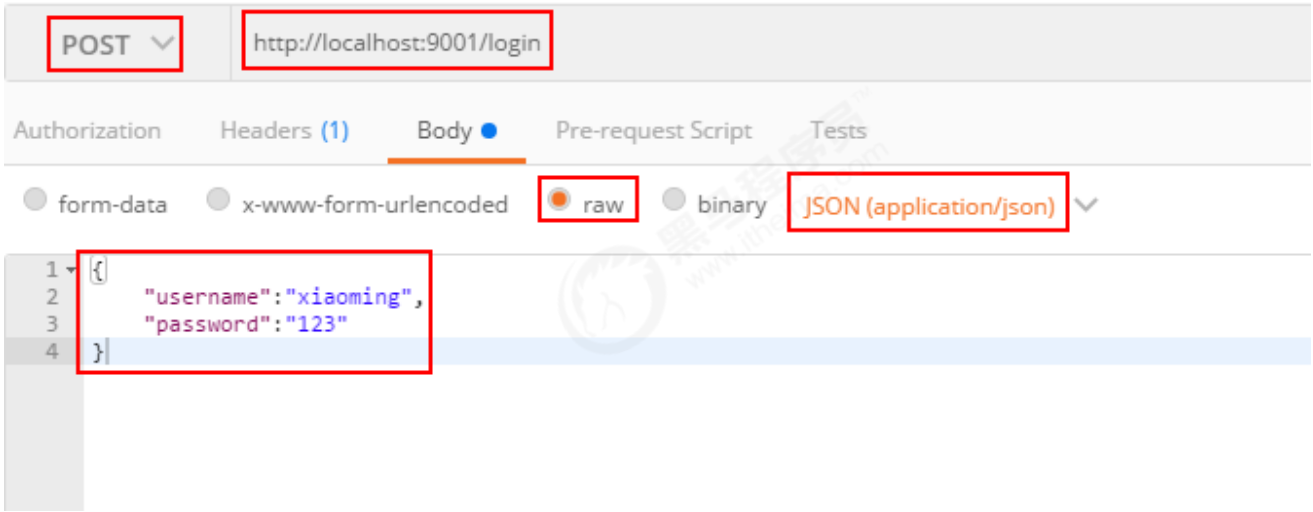
```
            return null;
        }
        return null;
    }
}
```

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();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                //关闭跨站请求防护
                .cors().and().csrf().disable()
                //允许不登陆就可以访问的方法，多个用逗号分隔
                .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);

    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        //UserDetailsService类
        auth.userDetailsService(myCustomUserService)
                //加密策略
                .passwordEncoder(myPasswordEncoder());
    }


}
```

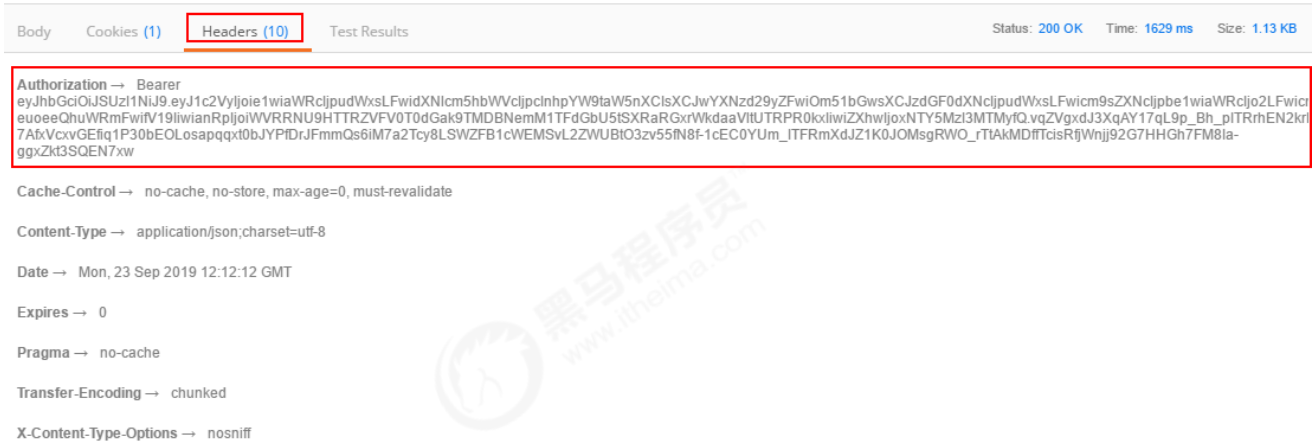
启动测试认证服务

认证请求

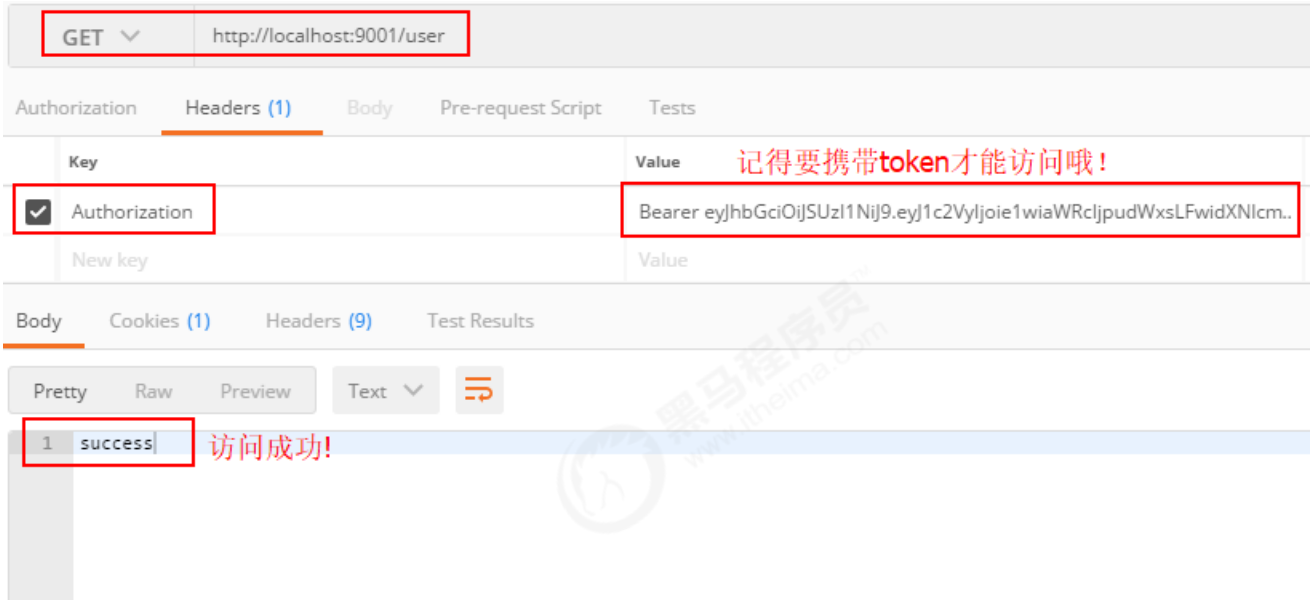


认证通过结果

token在Headers中：



验证认证请求



资源服务

说明

资源服务可以有很多个，这里只拿产品服务为例，记住，资源服务中只能通过公钥验证认证。不能签发token！

创建产品服务并导入jar包

根据实际业务导包即可，咱们就暂时和认证服务一样了。

```
<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.itheima</groupId>
        <artifactId>heima_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>

</dependencies>
```

编写产品服务配置文件

切记这里只能有公钥地址！

```yaml
server:
  port: 9002
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///security_authority
    username: root
    password: root
mybatis:
  type-aliases-package: com.itheima.domain
  configuration:
    map-underscore-to-camel-case: true
logging:
  level:
    com.itheima: debug
heima:
  key:
    pubKeyPath: D:\\auth_key\\rsa_key.pub
```

编写读取公钥的配置类

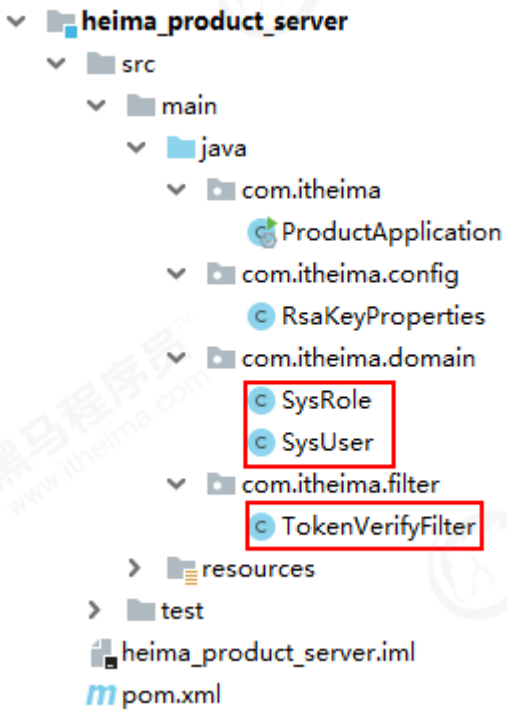
```java
@Data
@ConfigurationProperties(prefix = "heima.key")
public class RsaKeyProperties {
    private String pubKeyPath;

    private PublicKey publicKey;

    @PostConstruct
    public void loadKey() throws Exception {
        publicKey = RsaUtils.getPublicKey(pubKeyPath);
    }

}
```

编写启动类

```
@SpringBootApplication
@MapperScan("com.itheima.mapper")
@EnableConfigurationProperties(RsaKeyProperties.class)
public class ProductApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductApplication.class, args);
    }
}
```

复制认证服务中，用户对象，角色对象和校验认证的接口

这时目录结构如图：



复制认证服务中SpringSecurity配置类做修改

去掉"增加自定义认证过滤器"即可！

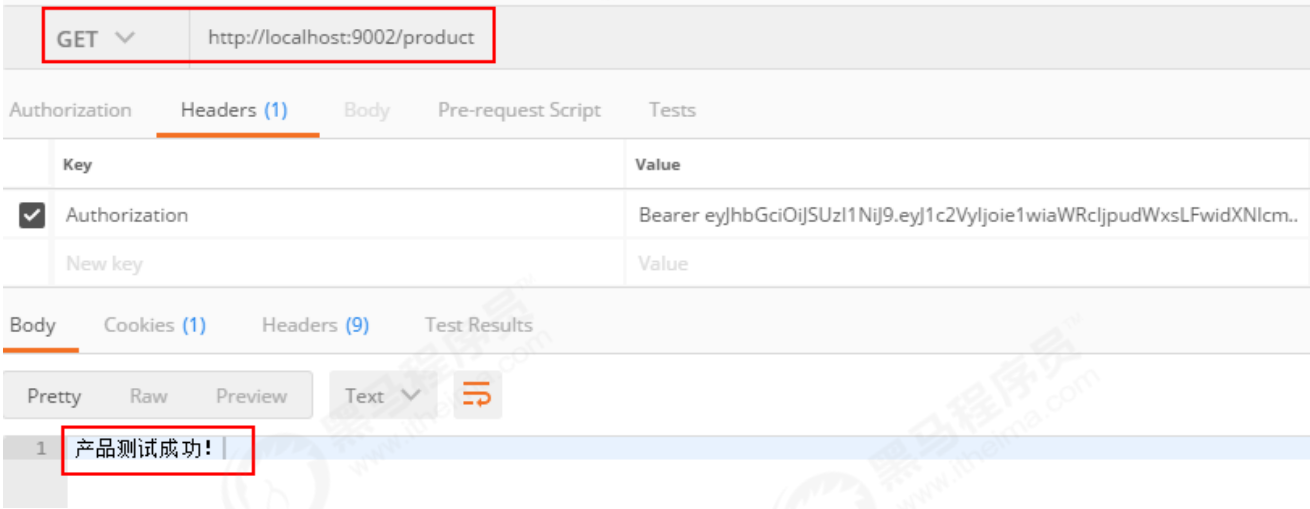
```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private UserDetailsService myCustomUserService;

    @Autowired
    private RsaKeyProperties prop;

    @Bean
    public BCryptPasswordEncoder myPasswordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                //关闭跨站请求防护
                .cors().and().csrf().disable()
                //允许不登陆就可以访问的方法，多个用逗号分隔
                .authorizeRequests().antMatchers("/product").hasAnyRole("USER")
                //其他的需要授权后访问
                .anyRequest().authenticated()

                .and()
                //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);

    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        //UserDetailsService类
        auth.userDetailsService(myCustomUserService)
                //加密策略
                .passwordEncoder(myPasswordEncoder());
    }

}
```

编写产品处理器

```
@RestController
@RequestMapping("/product")
public class ProductController {

    @GetMapping
    public String findAll(){
        return "产品测试成功！";
    }
}
```

启动产品服务做测试

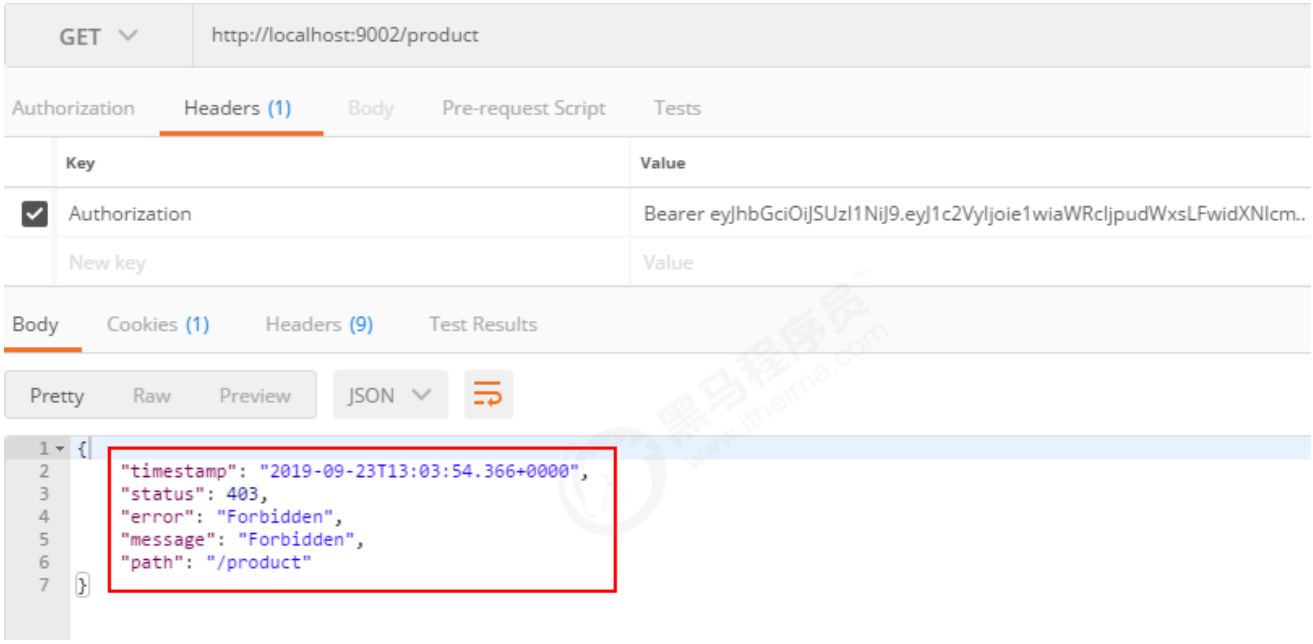
携带token



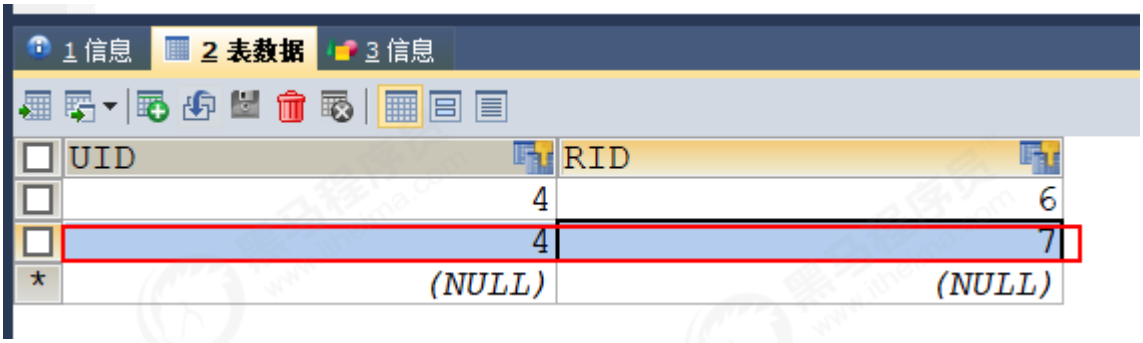
在产品处理器上添加访问需要ADMIN角色

```
@RestController
@RequestMapping("/product")
public class ProductController {

    @Secured("ROLE_ADMIN")
    @GetMapping
    public String findAll(){
        return "产品测试成功！";
    }
}
```

重启测试权限不足



在数据库中手动给用户添加ADMIN角色



重新认证获取新token再测试OK了！