主讲老师： Fox

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

1. Spring Security介绍

1.1 Spring Security定义

Spring Security是一个能够为基于Spring的企业应用系统提供声明式的安全访问控制解决方案的安全框架。Spring Security 主要实现了**Authentication**（认证，解决who are you? ） 和 **AccessControl**（访问控制，也就是what are you allowed to do？，也称为**Authorization**）。SpringSecurity在架构上将认证与授权分离，并提供了扩展点。

> Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications. Spring Security 是一个功能强大且高度可定制的身份验证和访问控制框架。它是用于保护基于 Spring 的应用程序。
> Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements
> Spring Security 是一个框架，侧重于为 Java 应用程序提供身份验证和授权。与所有 Spring 项目一样，Spring 安全性
> 的真正强大之处，在于它很容易扩展以满足定制需求。

认证：用户认证就是判断一个用户的身份是否合法的过程，用户去访问系统资源时系统要求验证用户的身份信息，身份合法方可继续访问，不合法则拒绝访问。常见的用户身份认证方式有：用户名密码登录，二维码登录，手机短信登录，指纹认证等方式。

授权： 授权是用户认证通过根据用户的权限来控制用户访问资源的过程，拥有资源的访问权限则正常访问，没有权限则拒绝访问。

1.2 Spring Security和Shiro比较

在 Java 生态中，目前有 Spring Security 和 Apache Shiro 两个安全框架，可以完成认证和授权的功能。

- **Spring Security**
- Apache Shiro：一个功能强大且易于使用的Java安全框架,提供了认证,授权,加密,和会话管理。

相同点:

1. 认证功能
2. 授权功能
3. 加密功能
4. 会话管理
5. 缓存支持
6. rememberMe功能

不同点：

优点：

1. Spring Security基于Spring开发，项目中如果使用Spring作为基础，配合Spring Security做权限更加方便，而Shiro需要和Spring进行整合开发
2. Spring Security功能比Shiro更加丰富些，例如安全防护
3. Spring Security社区资源比Shiro丰富

缺点：

1. Shiro的配置和使用比较简单，Spring Security上手复杂
2. Shiro依赖性低，不需要任何框架和容器，可以独立运行，而Spring Security依赖于Spring容器

一般来说，常见的安全管理技术栈的组合是这样的：

- SSM + Shiro
- Spring Boot/Spring Cloud +Spring Security

2. Spring Security使用

2.1 用户身份认证

快速开始

创建一个SpringBoot项目

1）引入依赖

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

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

2）编写测试的Controller

```
@RestController
@RequestMapping("/admin")
```

```
public class AdminController {
    @GetMapping("/demo")
    public String demo() {
    return "spring security demo";
    }
}
```

3) 启动项目后测试接口调用

引入Spring Security之后，访问 API 接口时，需要首先进行登录，才能进行访问。

测试 http://localhost:8080/admin/demo ,会跳转到登录界面

> 页面生成源码：DefaultLoginPageGeneratingFilter#generateLoginPageHtml
> 用户名密码认证Filter: UsernamePasswordAuthenticationFilter

需要登录，默认用户名：user，密码可以查看控制台日志获取

登录之后跳转回请求接口

4）退出登录

Spring security默认实现了logout退出，用户只需要向 Spring Security 项目中发送 http://localhost:8080/logout 退出请求即可。

设置用户名密码

基于application.yml方式

可以在application.yml中自定义用户名密码：

```
spring:
  # Spring Security 配置项，对应 SecurityProperties 配置类
  security:
    user:
      name: user   # 用户名
      password: 123456  # 密码
      roles:   # 拥有角色
        - admin
```

思考： 为什么可以这样配置?

原理：

默认情况下，UserDetailsServiceAutoConfiguration自动化配置类，会创建一个内存级别的InMemoryUserDetailsManager对象，提供认证的用户信息。

- 添加 spring.security.user 配置项，UserDetailsServiceAutoConfiguration 会基于配置的信息在内存中创建一个用户User。
- 未添加 spring.security.user 配置项，UserDetailsServiceAutoConfiguration 会自动在内存中创建一个用户名为 user，密码为 UUID 随机的用户 User

基于Java Bean配置方式

```
@Configuration
@EnableWebSecurity  //开启spring sercurity支持
public class SecurityConfig {


    /**
     * 配置用户信息
     * @return
     */
    @Bean
    public UserDetailsService userDetailsService() {
        //使用默认加密方式bcrypt对密码进行加密，添加用户信息
        UserDetails user = User.withDefaultPasswordEncoder()
                .username("fox")
                .password("123456")
                .roles("user")
                .build();

        UserDetails admin = User.withUsername("admin")
                .password("{noop}123456") //对密码不加密
                .roles("admin", "user")
                .build();
        return new InMemoryUserDetailsManager(user, admin);
    }

}
```

设置加密方式

方式1：{id}encodedPassword

Spring Security密码加密格式为：{id}encodedPassword

```
UserDetails user = User.withUsername("user")
      .password("{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG")
      .roles("USER")
      .build();
UserDetails admin = User.withUsername("admin")
        .password("{noop}123456") //noop表示对密码不加密
        .roles("admin", "user")
        .build();
```

如果密码不指定{id}会抛异常：

Spring Security支持的加密方式可以通过PasswordEncoderFactories查看

方式2： passwordEncoder().encode("123456")

也可以通过增加PasswordEncoder配置指定加密方式

```
UserDetails admin = User.withUsername("admin")
        //指定加密算法对密码加密
        .password(passwordEncoder().encode("123456"))
        .roles("admin", "user")
        .build();


@Bean
public PasswordEncoder passwordEncoder(){
    return NoOpPasswordEncoder.getInstance();  //不加密
    //return new BCryptPasswordEncoder();  //加密方式bcrypt
}
```

自定义用户信息加载方式：实现UserDetailsService接口

需要自定义从数据库获取用户信息，可以实现UserDetailsService接口

```
@Service
public class TulingUserDetailService implements UserDetailsService {

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        //TODO  根据用户名可以从数据库获取用户信息，角色以及权限信息
        // 模拟从数据库获取了用户信息，并封装成UserDetails对象
        UserDetails user = User
                .withUsername("fox")
                .password(passwordEncoder.encode("123456"))
                .roles("user")
                .build();

        return user;
    }
}
```

自定义登录页面

Spring Security默认登录页面通过DefaultLoginPageGeneratingFilter#generateLoginPageHtml生成

1) 编写登录页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
```

```
</head>
<body>
    <form action="/user/login" method="post">
        用户名:<input type="text" name="username"/><br/>
        密码：<input type="password" name="password"/><br/>
        <input type="submit" value="提交"/>
    </form>
</body>
</html>
```

2) 配置Spring Security的过滤器链

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    //表单提交
    http.formLogin((formLogin) -> formLogin
            .loginPage("/login.html") //指定自定义登录页面地址
            .loginProcessingUrl("/user/login")//登录访问路径：前台界面提交表单之后跳转到这个路径进行UserDetailsService的验证，必须和表单提交接口一样
            .defaultSuccessUrl("/admin/demo")//认证成功之后跳转的路径
    );
    //对请求进行访问控制设置
    http.authorizeHttpRequests((authorizeHttpRequests) -> authorizeHttpRequests
            //设置哪些路径可以直接访问，不需要认证
            .requestMatchers("/login.html","/user/login").permitAll()
            .anyRequest().authenticated() //其他路径的请求都需要认证
    );
    //关闭跨站点请求伪造csrf防护
    http.csrf((csrf) -> csrf.disable());

    return http.build();
}
```

测试 http://localhost:8080/admin/demo ,会跳转到自定义登录界面

前后端分离认证

表单登录配置模块提供了successHandler（）和failureHandler（）两个方法，分别处理登录成功和登录失败的逻辑。其中，successHandler()方法带有一个Authentication参数，携带当前登录用户名及其

角色等信息；而failureHandler()方法携带一个AuthenticationException异常参数。

```
//前后端分离认证逻辑
http.formLogin((formLogin) -> formLogin
        .loginProcessingUrl("/login") //登录访问接口
        .successHandler(new LoginSuccessHandler()) //登录成功处理逻辑
        .failureHandler(new LoginFailureHandler()) //登录失败处理逻辑
);


//
/**
 * 认证成功处理逻辑
 */
public class LoginSuccessHandler implements AuthenticationSuccessHandler {
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {
        response.setContentType("text/html;charset=utf-8");
        response.getWriter().write("登录成功");
    }
}

//
/**
 * 认证失败处理逻辑
 */
public class LoginFailureHandler implements AuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response, AuthenticationException exception) throws IOException, ServletException {
        // TODO
        response.setContentType("text/html;charset=utf-8");
        response.getWriter().write("登录失败");
        exception.printStackTrace();
    }
}
```

认证流程

2.2 用户授权（访问控制）

授权的方式包括 web授权和方法授权，web授权是通过url拦截进行授权，方法授权是通过方法拦截进行授权。

web授权： 基于url的访问控制

Spring Security可以通过 http.authorizeRequests() 对web请求进行授权保护， Spring Security使用标准Filter建立了对web请求的拦截，最终实现对资源的授权访问。配置顺序会影响之后授权的效果，越是具体的应该放在前面，越是笼统的应该放到后面。

```
//对请求进行访问控制设置
http.authorizeHttpRequests((authorizeHttpRequests) -> authorizeHttpRequests
        //设置哪些路径可以直接访问，不需要认证
        .requestMatchers("/login").permitAll()  //不需要认证
        .requestMatchers("/index").hasRole("user")  //需要user角色,底层会判断是否有ROLE_admin权限
        .requestMatchers("/index2").hasRole("admin")
        .requestMatchers("/user/**").hasAuthority("user:api") //需要user:api权限
        .requestMatchers("/order/**").hasAuthority("order:api")
        .anyRequest().authenticated()  //其他路径的请求都需要认证
);

@Bean
public UserDetailsService userDetailsService() {
    UserDetails user = User.withDefaultPasswordEncoder()
            .username("fox")
            .password("123456")
            .roles("user")
            .build();

    UserDetails admin = User.withDefaultPasswordEncoder()
            .username("admin")
```

```
            .password("123456")
            // 注意： roles和authorities不能同时配置，同时配置后者会覆盖前者的权限
            .authorities("ROLE_admin","ROLE_user","user:api","order:api")
            .build();

    return new InMemoryUserDetailsManager(user,admin);
}
```

自定义授权失败异常处理

使用 Spring Security 时经常会看见 403（无权限）。Spring Security 支持自定义权限受限处理，需要实现 AccessDeniedHandler接口

```
public class BussinessAccessDeniedHandler implements org.springframework.security.web.access.AccessDeniedHandler {
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response, AccessDeniedException accessDeniedException) throws IOException, ServletException {
        response.setContentType("text/html;charset=utf-8");
        response.getWriter().write("没有访问权限");
        accessDeniedException.printStackTrace();
    }
}
```

在配置类中设置访问受限后交给BussinessAccessDeniedHandler处理

```
//访问受限后的异常处理
http.exceptionHandling((exceptionHandling) ->
        exceptionHandling.accessDeniedHandler(new BussinessAccessDeniedHandler())
);
```

方法授权：基于注解的访问控制

Spring Security在方法的权限控制上支持三种类型的注解，JSR-250注解、@Secured注解和支持表达式的注解。这三种注解默认都是没有启用的，需要通过@EnableGlobalMethodSecurity来进行启用。

```java
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true, securedEnabled = true, prePostEnabled = true)
public class SecurityConfig {
}


//Controller
@RolesAllowed({"ROLE_user","ROLE_admin"})  //配置访问此方法时应该具有的角色
@GetMapping("/index5")
public String index5(){
    return "index5";
}

@Secured("ROLE_admin")    //配置访问此方法时应该具有的角色
@GetMapping("/index6")
public String index6(){
    return "index6";
}

```

Spring Security中定义了四个支持使用表达式的注解，分别是@PreAuthorize、@PostAuthorize、@PreFilter和@PostFilter。其中前两者可以用来在方法调用前或者调用后进行权限检查，后两者可以用来对集合类型的参数或者返回值进行过滤。

```java
@PreAuthorize("hasRole('ROLE_admin') and #id<10 ") //访问此方法需要具有admin角色，同时限制只能查询id小于10的用户
@GetMapping("/findUserById")
public String findById(long id) {
    //TODO 查询数据库获取用户信息
    return "success";
}
```

利用过滤器实现动态权限控制

Spring Security从5.5之后动态权限控制方式已经改变。

5.5之前需要实现接口：

- FilterInvocationSecurityMetadataSource： 获取访问URL所需要的角色信息
- AccessDecisionManager： 用于权限校验，失败抛出AccessDeniedException 异常

5.5之后，利用过滤器动态控制权限，在AuthorizationFilter中，只需要实现接口AuthorizationManager，如果没有权限，抛出AccessDeniedException异常

> 权限校验核心逻辑：
> org.springframework.security.web.access.intercept.AuthorizationFilter#doFilter
> 》org.springframework.security.authorization.AuthorityAuthorizationManager#check
> 》org.springframework.security.authorization.AuthoritiesAuthorizationManager#isAuthorized

3. Spring Security整合JWT实现自定义登录认证

3.1 自定义登录认证的业务需求

某企业要做前后端分离的项目，决定要用spring boot + spring security+JWT 框架实现登录认证授权功能，用户登录成功后，服务端利用JWT生成token，之后客户端每次访问接口，都需要在请求头上添加Authorization：Bearer token 的方式传值到服务器端，服务器端再从token中解析和校验token的合法性，如果合法，则取出用户数据，保存用户信息，不需要在校验登录，否则就需要重新登录

3.2 JWT详解

什么是JWT

JSON Web Token（JWT）是一个开放的行业标准（RFC 7519），它定义了一种简介的、自包含的协议格式，用于在通信双方传递json对象，传递的信息经过数字签名可以被验证和信任。JWT可以使用HMAC算法或使用RSA的公钥/私钥对来签名，防止被篡改。 官网： https://jwt.io/ 标准： https://tools.ietf.org/html/rfc7519

JWT令牌的优点：

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

缺点：

1. JWT令牌较长，占存储空间比较大。
2. 安全性取决于密钥管理

JWT 的安全性取决于密钥的管理。如果密钥被泄露或者被不当管理，那么 JWT 将会受到攻击。因此，在使用 JWT 时，一定要注意密钥的管理，包括生成、存储、更新、分发等等。

3. 无法撤销

由于 JWT 是无状态的，一旦 JWT 被签发，就无法撤销。如果用户在使用 JWT 认证期间被注销或禁用，那么服务端就无法阻止该用户继续使用之前签发的 JWT。因此，开发人员需要设计额外的机制来撤销 JWT，例如使用黑名单或者设置短期有效期等等。

使用 JWT 主要用来做下面两点：

- 认证(Authorization)：这是使用 JWT 最常见的一种情况，一旦用户登录，后面每个请求都会包含 JWT，从而允许用户访问该令牌所允许的路由、服务和资源。单点登录是当今广泛使用 JWT 的一项功能，因为它的开销很小。
- 信息交换(Information Exchange)：JWT 是能够安全传输信息的一种方式。通过使用公钥/私钥对 JWT 进行签名认证。此外，由于签名是使用 head 和 payload 计算的，因此你还可以验证内容是否遭到篡改。

JWT组成

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

头部（header）

头部用于描述关于该JWT的最基本的信息：类型（即JWT）以及签名所用的算法（如HMACSHA256或RSA）等。这也可以被表示成一个JSON对象：

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

然后将头部进行base64加密（该加密是可以对称解密的),构成了第一部分:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

载荷（payload）

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

- 标准中注册的声明（建议但不强制使用）

iss: jwt签发者

sub: jwt所面向的用户

aud: 接收jwt的一方

exp: jwt的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该jwt都是不可用的.

iat: jwt的签发时间

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

- 公共的声明 公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密.
- 私有的声明 私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个payload：

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

然后将其进行base64加密，得到Jwt的第二部分:

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
```

签名（signature）

jwt的第三部分是一个签证信息，这个签证信息由三部分组成：

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

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

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

var signature = HMACSHA256(encodedString, 'fox'); //
khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME
```

将这三部分用.连接成一个完整的字符串,构成了最终的jwt:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME
```

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

如何应用

一般是在请求头里加入Authorization，并加上Bearer标注：

```
fetch('api/user/1', {
  headers: {
    'Authorization': 'Bearer ' + token
  }
})
```

服务端会验证token，如果验证通过就会返回相应的资源。整个流程就是这样的:

3.3 自定义登录核心实现

结合课堂代码理解

1）实现校验token的过滤器

```
@Slf4j
@Component
public class JwtAuthenticationTokenFilter extends OncePerRequestFilter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

        //1.从请求头中取出token，进行判断，如果没有携带token，则继续往下走其他的其他的filter逻辑
        String tokenValue = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (!StringUtils.hasText(tokenValue)) {
            filterChain.doFilter(request, response);
            return;
```

```
        }
        //2. 校验token
        //2.1 将token切割前缀“bearer ”，然后使用封装的JWT工具解析token，得到一个map对象
        String token = tokenValue.substring("bearer ".length());
        Map<String, Object> map = JWTUtils.parseToken(token);
        //2.2 取出token中的过期时间，调用JWT工具中封装的过期时间校验，如果token已经过期，则删除登录的用户，继续往下走其他filter逻辑
        if (JWTUtils.isExpiresIn((long) map.get("expiresIn"))) {
            //token 已经过期
            SecurityContextHolder.getContext().setAuthentication(null);
            filterChain.doFilter(request, response);
            return;
        }

        String username = (String) map.get("username");
        if (StringUtils.hasText(username) && SecurityContextHolder.getContext().getAuthentication() == null) {
            //获取用户信息
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);
            if (userDetails != null && userDetails.isEnabled()) {
                UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                //设置用户登录状态
                log.info("authenticated user {}, setting security context", username);
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
        filterChain.doFilter(request, response);

    }
}

```

2）SpringSecurity的过滤器链路中添加JWT登录过滤器

```

```

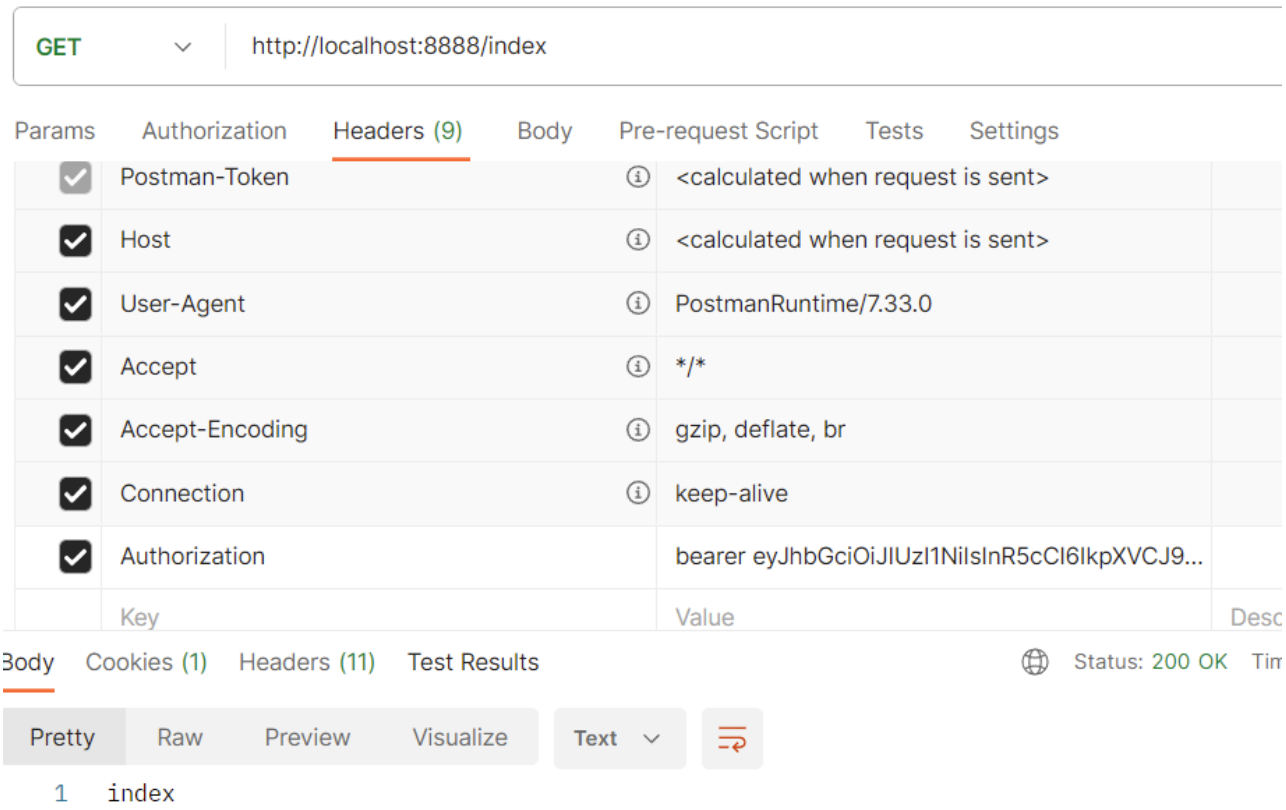
```
//添加JWT登录过滤器，在登录之前获取token并校验
http.addFilterBefore(jwtAuthenticationTokenFilter,
UsernamePasswordAuthenticationFilter.class);
```

3）测试效果

启动应用后调用登录接口返回token信息

不带token信息访问接口，返回401，没有权限

带token信息访问接口，返回正常



3.4 JWT续期问题

JWT（JSON Web Token）通常是在用户登录后签发的，用于验证用户身份和授权。JWT 的有效期限（或称“过期时间”）通常是一段时间（例如1小时），过期后用户需要重新登录以获取新的JWT。然而，在某些情况下，用户可能会在JWT到期之前使用应用程序，这可能会导致应用程序不可用或需要用户重新登录。为了避免这种情况，通常有两种解决方案来处理JWT续期问题：

刷新令牌（Refresh Token）

刷新令牌是一种机制，它允许应用程序获取一个新的JWT，而无需用户进行身份验证。当JWT过期时，应用程序使用刷新令牌向身份验证服务器请求一个新的JWT，而无需提示用户输入其凭据。这样，用户可以继续使用应用程序，而不必重新登录。

以下是一个示例Java代码，演示如何使用Refresh Token来更新JWT

```java
public String refreshAccessToken(String refreshToken) {

    // validate the refresh token (check expiration, signature, etc.)
    boolean isValid = validateRefreshToken(refreshToken);

    if (isValid) {
        // retrieve the user information associated with the refresh token (e.g. user ID)
        String userId = getUserIdFromRefreshToken(refreshToken);

        // generate a new JWT access token
        String newAccessToken = generateAccessToken(userId);

        return newAccessToken;
    } else {
        throw new RuntimeException("Invalid refresh token.");
    }
}
```

在这个示例中，refreshAccessToken方法接收一个刷新令牌作为参数，并使用validateRefreshToken方法验证该令牌是否有效。如果令牌有效，方法将使用getUserIdFromRefreshToken方法获取与令牌关联的用户信息，然后使用generateAccessToken方法生成一个新的JWT访问令牌，并将其返回。如果令牌无效，则抛出异常。

自动延长JWT有效期

在某些情况下，JWT可以自动延长其有效期。例如，当用户在JWT过期前继续使用应用程序时，应用重新设置token过期时间。

要自动延长JWT有效期，您可以在每次请求时检查JWT的过期时间，并在必要时更新JWT的过期时间。以下是一个示例Java代码，演示如何自动延长JWT有效期：

```java
public String getAccessToken(HttpServletRequest request) {

    String accessToken = extractAccessTokenFromRequest(request);

    if (isAccessTokenExpired(accessToken)) {
        String userId = extractUserIdFromAccessToken(accessToken);
        accessToken = generateNewAccessToken(userId);
```

```
    } else if (shouldRefreshAccessToken(accessToken)) {
        String userId = extractUserIdFromAccessToken(accessToken);
        accessToken = generateNewAccessToken(userId);
    }

    return accessToken;
}

private boolean isAccessTokenExpired(String accessToken) {
    // extract expiration time from the access token
    Date expirationTime = extractExpirationTimeFromAccessToken(accessToken);

    // check if the expiration time is in the past
    return expirationTime.before(new Date());
}

private boolean shouldRefreshAccessToken(String accessToken) {
    // extract expiration time and current time
    Date expirationTime = extractExpirationTimeFromAccessToken(accessToken);
    Date currentTime = new Date();

    // calculate the remaining time until expiration
    long remainingTime = expirationTime.getTime() - currentTime.getTime();

    // refresh the token if it expires within the next 5 minutes
    return remainingTime < 5 * 60 * 1000;
}

private String generateNewAccessToken(String userId) {
    // generate a new access token with a new expiration time
    Date expirationTime = new Date(System.currentTimeMillis() +
ACCESS_TOKEN_EXPIRATION_TIME);
    String accessToken = generateAccessToken(userId, expirationTime);

    return accessToken;
}

```

在这个示例中，getAccessToken方法接收HttpServletRequest对象作为参数，并使用extractAccessTokenFromRequest方法从请求中提取JWT访问令牌。然后，它使用isAccessTokenExpired方法检查JWT的过期时间是否已过期。如果过期，它使用extractUserIdFromAccessToken方法从JWT中提取用户ID，并使用generateNewAccessToken方法生成一个新的JWT访问令牌。如果JWT尚未过期，但即将到期，则使用shouldRefreshAccessToken方法检查JWT是否需要更新。如果是这样，它使用相同的流程生成一个新的JWT访问令牌。