在线oj项目-身份认证01

身份认证方式

- **基于Session的身份认证**：这是最常见的身份认证方式。当用户首次登录时，服务器会将用户信息存入session并生产一个唯一的Session ID，然后返回给客户端。此后的请求中客户端会要携带这个Session ID，服务器通过验证Session ID的有效性来判断用户的身份。
- **基于OAuth的身份认证**：OAuth认证机制是一种安全、开放且简易的授权标准，它允许用户授权第三方应用访问其账户资源，而无需向这些应用提供用户名和密码。如使用微信、QQ等账号登录其他网站或应用。
- **基于Token的身份认证**：这种方式中，服务器在用户登录成功后，会返回一个Token给客户端。客户端每次请求资源时，都需要在请求头中携带这个Token。服务器通过验证Token的有效性来判断用户的身份。这种方式常见于前后端分离的架构中，如使用JWT（JSON Web Token）进行身份认证。

我们将使用JWT认证机制，完成用户身份的认证。

Jwt（JSON Web Token）

简介：

官网地址：https://jwt.io/

JWT的全称是Json Web Token，是一种基于JSON的安全跨平台信息传输格式。JWT定义了一种紧凑且自包含的方式，用于在各方之间安全地传输信息。此信息可以用作验证和相互信任。

JWT组成：

它由三部分组成：头部（header）、载荷（payload）和签名（signature）。

- **头部（header）**：包含令牌的类型和使用的算法。

  > 使用base64编码

- **载荷（payload）**：包含用户信息和其他元数据。（使用base64编码）

  > 使用base64编码

- **签名（signature）**：用于验证令牌的完整性和真实性。

  > Header中定义的签名算法(
  >
  > base64编码(header) + "." + base64编码(payload),
  >
  > secret

```
)
```

示例：

下面是一个jwt串，上述三部分以 . 进行分割

```
eyJhbGciOiJIUzUxMiJ9.eyJ1c2VyX2lkIjoxLCJuaWNrTmFtZSI6InRlc3QiLCJ1c2VyX2tleSI6Ij
lmYTQ5NDdiLTZiMzYtNDk5My04MjA5LTgzZDNiOGViOGMwMiJ9.8VX9V2vDB-
X8ZgHtWITDDPF6UE413e4bs5GTWnM9cZkv_tUYloobIb991tzA_z0m3lse0ssD_AzIikWFbTc5HA
```

身份认证流程：

- 客户端使用用户名跟密码请求登录 。
- 服务端收到请求，去验证用户名与密码 。
- 验证成功后，服务端会签发一个Token，再把这个Token发送给客户端 。（token上述的jwt串）
- 客户端收到Token以后可以把它存储起来，比如放在Cookie里或者Local Storage里 。
- 客户端每次向服务端请求资源的时候需要带着服务端签发的Token 。
- 服务端收到请求，然后去验证客户端请求里面带着的Token，如果验证成功，就向客户端返回请求的数据 。

为什么选择JWT

- **简单方便**：JWT认证机制不需要像传统的Session认证那样在服务器端存储任何会话信息，所有的认证和授权信息都包含在JWT中。这种方式简化了认证流程，减少了服务器的负担。
- **安全可靠**：JWT使用数字签名来验证其完整性和真实性，确保数据在传输过程中不被篡改。
- **易于扩展**：JWT是无状态的，服务器不需要存储用户的会话信息，这使得应用程序更容易进行水平扩展，这一点很适用于我们采用的微服务架构。当系统需要处理大量用户请求时，无状态的认证方式更加适合。
- **支持跨域**：JWT认证机制中在客户端与服务器进行通信时，客户端会将JWT作为请求的一部分发送给服务器，不依赖于浏览器的cookie或session，因此不会受到同源策略的限制。这使得它非常适合处理跨域请求。如果你的Web项目需要与前端应用或其他服务进行跨域通信，JWT认证机制会是一个好选择。

总结：使用JWT简单方便、安全可靠。可以减少服务器存储带来的开销和复杂性，实现跨域支持和水平扩展，并且更适应无状态和微服务架构。

问题分析

提出问题：身份认证仅仅使用jwt机制就可以吗？

- jwt中payload中存储用户相关信息，采用base64编码，没有加密因此jwt中不能存储敏感数据。

> 但是我们的一些业务逻辑中需要获取当前登录用户一些敏感信息参与到业务逻辑中。

- jwt是无状态的，因此如果想要修改里面的内容就必须重新签发一次新的jwt。

  > 用户修改自己个人信息之后就需要重新登录

- 无法延长jwt的过期时间

  > 用户正在操作突然身份认证失败

问题处理思路：

- **payload中不能存放敏感信息。**
  - payload中仅仅存储用户唯一标识信息。
  - 需要第三方存储机制，作为辅助存储用于用户身份认证的信息。存储需要保证安全、可靠。
  - 不需要长期存储，仅在用户处于登录状态时使用。
- **想办法用户修改了个人信息之后，jwt不变。**
  - payload中仅仅存储用户唯一标识信息，那么修改用户信息就不会引起jwt改变。
- **能够控制jwt的过期时间**
  - jwt的过期时间，不通过jwt机制提供的方法设置。
  - 通过第三方组件设置并记录jwt的过期时间。

具体处理方案：

我们将使用redis + jwt的结构完成身份认证。jwt中仅存储用户的唯一标识信息，使用redis作为第三方存储机制，存储用于用户身份认证的信息，并通过redis控制jwt的过期时间。

redis缓存设计：

存储信息	**redis中数据结构**	**key**	**value**	**缓存有效时间**	**缓存刷新时机**
登录用户信息	string类型	login_tokens:用户token	JSON结构存储登录用户信息 token *用户唯一标识* userId *用户名id* nickName *用户昵称* identity *用户身份*	720分钟 **（用户长时间不操作自动下线，防止被他人盗用）**	用户缓存有效期内访问页面，如果缓存即将失效更新缓存有效期。**（防止使用过程中突然下线）** 处理方案：使用拦截器在业务处理之前 用户重新登录时重新将用户信息录入缓存。

			~~loginTime 登录时间~~ ~~permissions 用户权限~~ ~~roles 用户角色~~ ~~ip 用户ip~~		

功能实现

- **导入相关依赖：**

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>

<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.4.0-b180830.0359</version>
</dependency>

<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>5.8.22</version>
</dependency>
```

- **jwt工具类：**

```
package com.bite.common.core.utils;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

import java.util.HashMap;
import java.util.Map;

public class JwtUtils {

```

```
    /**
     * 生成令牌
     *
     * @param claims 数据
     * @param secret 密钥
     * @return 令牌
     */
    public static String createToken(Map<String, Object> claims, String secret)
 {
        String token =
Jwts.builder().setClaims(claims).signWith(SignatureAlgorithm.HS512,
secret).compact();
        return token;
    }

    /**
     * 从令牌中获取数据
     *
     * @param token 令牌
     * @param secret 密钥
     * @return 数据
     */
    public static Claims parseToken(String token, String secret) {
        return
Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();
    }
}
```

- **定义CacheConstants**

```
public class CacheConstants {
    /**
     * 缓存有效期，默认720（分钟）
     */
    public final static long EXPIRATION = 720;

    /**
     * 用户身份认证缓存前缀
     */
    public final static String LOGIN_TOKEN_KEY = "login_tokens:";
}
```

- **提供TokenService**

```java
@Component
public class TokenService {

    @Autowired
    private RedisService redisService;

    /**
     * 创建令牌
     */
    public String createToken(LoginUser loginUser) {
        String token = UUID.fastUUID().toString();
        loginUser.setToken(token);
        refreshToken(loginUser);

        // Jwt存储信息
        Map<String, Object> claimsMap = new HashMap<String, Object>();
        claimsMap.put(SecurityConstants.USER_KEY, token);
        claimsMap.put(SecurityConstants.DETAILS_USER_ID, loginUser.getUserId());
        claimsMap.put(SecurityConstants.DETAILS_NICKNAME, loginUser.getNickName());

        return JwtUtils.createToken(claimsMap);;
    }

    /**
     * 刷新令牌有效期
     *
     * @param loginUser 登录信息
     */
    public void refreshToken(LoginUser loginUser) {
        // 根据uuid将loginUser缓存
        String userKey = getTokenKey(loginUser.getToken());
        redisService.setCacheObject(userKey, loginUser, CacheConstants.EXPIRATION, TimeUnit.MINUTES);
    }

    private String getTokenKey(String token) {
        return CacheConstants.LOGIN_TOKEN_KEY + token;
    }
}
```

- **定义loginUser**

```java
public class LoginUser {
```

```
    /**
     * 用户身份 (0: 普通用户 1: 管理员)
     */
    private Integer identity;
}
```

- **修改登录接口：**

```
    public R<String> login(String userAccount, String password) {
        LambdaQueryWrapper<SysUser> queryWrapper = new LambdaQueryWrapper<>();
        SysUser sysUser = sysUserMapper.selectOne(queryWrapper
                .select(SysUser::getUserId,
SysUser::getPassword).eq(SysUser::getUserAccount, userAccount));
        if (sysUser == null) {
            return R.fail(ResultCode.FAILED_USER_NOT_EXISTS);
        }
        if (BCryptUtils.matchesPassword(password, sysUser.getPassword())) {
            return R.ok(tokenService.createToken(sysUser.getUserId(), secret,
UserIdentity.ADMIN.getValue()));
        }
        return R.fail(ResultCode.FAILED_LOGIN);
    }
```

网关层代码改造：

- **引入依赖**

```
<dependency>
    <groupId>com.bite</groupId>
    <artifactId>oj-common-redis</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>com.bite</groupId>
    <artifactId>oj-common-core</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

- **修改nacos配置**

```
server:
```

```yaml
  port: 19090
spring:
  data:
    redis:
      host: localhost
      password: 123456
  cloud:
    gateway:
      routes:
        # 管理模块
        - id: oj-system
          uri: lb://oj-system
          predicates:
            - Path=/system/**
          filters:
            - StripPrefix=1

jwt:
  secret: zxcvbnmasdfghjuiyreqtuiwq

security:
  # 不校验白名单
  ignore:
    whites:
      - /system/**/login
```

- **RedisConfig修改**

  需要增加@AutoConfigureBefore(RedisAutoConfiguration.class)注解否则会报错如下：


```java
import org.springframework.boot.autoconfigure.AutoConfigureBefore;
import
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration;
import org.springframework.cache.annotation.CachingConfigurerSupport;

import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
@AutoConfigureBefore(RedisAutoConfiguration.class)
public class RedisConfig extends CachingConfigurerSupport {
    @Bean
    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);
        JsonRedisSerializer serializer = new JsonRedisSerializer(Object.class);
// 使用StringRedisSerializer来序列化和反序列化redis的key值
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(serializer);
// Hash的key也采用StringRedisSerializer的序列化方式
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(serializer);
        template.afterPropertiesSet();
        return template;
    }
}
```

- **增加AuthFilter**

```
package com.bite.gateway;

import cn.hutool.core.util.StrUtil;
import com.alibaba.fastjson2.JSON;
import com.bite.common.core.JwtUtils;
import com.bite.common.core.constants.CacheConstants;
import com.bite.common.core.constants.HttpConstants;
import com.bite.common.core.domain.LoginUser;
import com.bite.common.core.domain.R;
import com.bite.common.core.enums.ResultCode;
import com.bite.common.core.enums.UserIdentity;
import com.bite.common.redis.service.RedisService;
import io.jsonwebtoken.Claims;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
```

```
18 import org.springframework.cloud.gateway.filter.GlobalFilter;
19 import org.springframework.core.Ordered;
20 import org.springframework.core.io.buffer.DataBuffer;
21 import org.springframework.http.HttpHeaders;
22 import org.springframework.http.HttpStatus;
23 import org.springframework.http.MediaType;
24 import org.springframework.http.server.reactive.ServerHttpRequest;
25 import org.springframework.http.server.reactive.ServerHttpResponse;
26 import org.springframework.stereotype.Component;
27 import org.springframework.util.AntPathMatcher;
28 import org.springframework.util.CollectionUtils;
29 import org.springframework.web.server.ServerWebExchange;
30 import reactor.core.publisher.Mono;
31
32 import java.util.List;
33
34 /**
35  * 网关鉴权
36  *
37  */
38 @Slf4j
39 @Component
40 public class AuthFilter implements GlobalFilter, Ordered {
41
42     // 排除过滤的 uri 白名单地址，在nacos自行添加
43     @Autowired
44     private IgnoreWhiteProperties ignoreWhite;
45
46     @Value("${jwt.secret}")
47     private String secret;
48
49     @Autowired
50     private RedisService redisService;
51
52     @Override
53     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
   chain) {
54         ServerHttpRequest request = exchange.getRequest();
55
56         String url = request.getURI().getPath();
57         // 跳过不需要验证的路径
58         if (matches(url, ignoreWhite.getWhites())) {
59             return chain.filter(exchange);
60         }
61         //从http请求头中获取token
62         String token = getToken(request);
63         if (StrUtil.isEmpty(token)) {
```

```
64              return unauthorizedResponse(exchange, "令牌不能为空");
65          }
66          Claims claims;
67          try {
68              claims = JwtUtils.parseToken(token, secret); //获取令牌中信息  解析
    payload中信息
69              if (claims == null) {
70                  return unauthorizedResponse(exchange, "令牌已过期或验证不正确！");
71              }
72          } catch (Exception e) {
73              return unauthorizedResponse(exchange, "令牌已过期或验证不正确！");
74          }
75
76          String userKey = JwtUtils.getUserKey(claims);  //获取jwt中的key
77          boolean isLogin = redisService.hasKey(getTokenKey(userKey));
78          if (!isLogin) {
79              return unauthorizedResponse(exchange, "登录状态已过期");
80          }
81          String userid = JwtUtils.getUserId(claims);  //判断jwt中的信息是否完整
82          if (StrUtil.isEmpty(userid)) {
83              return unauthorizedResponse(exchange, "令牌验证失败");
84          }
85
86          LoginUser user = redisService.getCacheObject(getTokenKey(userKey),
    LoginUser.class);
87          if (url.contains(HttpConstants.SYSTEM_URL_PREFIX) &&
    !UserIdentity.ADMIN.getValue().equals(user.getIdentity())) {
88              return unauthorizedResponse(exchange, "令牌验证失败");
89          }
90          if (url.contains(HttpConstants.FRIEND_URL_PREFIX) &&
    !UserIdentity.ORDINARY.getValue().equals(user.getIdentity())) {
91              return unauthorizedResponse(exchange, "令牌验证失败");
92          }
93
94          return chain.filter(exchange);
95      }
96
97      /**
98       * 查找指定url是否匹配指定匹配规则链表中的任意一个字符串
99       *
100      * @param url         指定url
101      * @param patternList 需要检查的匹配规则链表
102      * @return 是否匹配
103      */
104     private boolean matches(String url, List<String> patternList) {
105         if (StrUtil.isEmpty(url) || CollectionUtils.isEmpty(patternList)) {
106             return false;
```

```
107         }
108         for (String pattern : patternList) {
109             if (isMatch(pattern, url)) {
110                 return true;
111             }
112         }
113         return false;
114     }
115 
116     /**
117      * 判断url是否与规则匹配
118      * 匹配规则中:
119      * ? 表示单个字符;
120      * * 表示一层路径内的任意字符串，不可跨层级;
121      * ** 表示任意层路径;
122      *
123      * @param pattern 匹配规则
124      * @param url     需要匹配的url
125      * @return 是否匹配
126      */
127     private boolean isMatch(String pattern, String url) {
128         AntPathMatcher matcher = new AntPathMatcher();
129         return matcher.match(pattern, url);
130     }
131 
132     /**
133      * 获取缓存key
134      */
135     private String getTokenKey(String token) {
136         return CacheConstants.LOGIN_TOKEN_KEY + token;
137     }
138 
139     /**
140      * 从请求头中获取请求token
141      */
142     private String getToken(ServerHttpRequest request) {
143         String token =
    request.getHeaders().getFirst(HttpConstants.AUTHENTICATION);
144         // 如果前端设置了令牌前缀，则裁剪掉前缀
145         if (StrUtil.isNotEmpty(token) &&
    token.startsWith(HttpConstants.PREFIX)) {
146             token = token.replaceFirst(HttpConstants.PREFIX, StrUtil.EMPTY);
147         }
148         return token;
149     }
150 
```

```
151     private Mono<Void> unauthorizedResponse(ServerWebExchange exchange, String
    msg) {
152         log.error("[鉴权异常处理]请求路径:{}", exchange.getRequest().getPath());
153         return webFluxResponseWriter(exchange.getResponse(), msg,
    ResultCode.FAILED_UNAUTHORIZED.getCode());
154     }
155
156     //拼装webflux模型响应
157     private Mono<Void> webFluxResponseWriter(ServerHttpResponse response,
    String msg, int code) {
158         response.setStatusCode(HttpStatus.OK);
159         response.getHeaders().add(HttpHeaders.CONTENT_TYPE,
    MediaType.APPLICATION_JSON_VALUE);
160         R<?> result = R.fail(code, msg);
161         DataBuffer dataBuffer =
    response.bufferFactory().wrap(JSON.toJSONString(result).getBytes());
162         return response.writeWith(Mono.just(dataBuffer));
163     }
164
165     @Override
166     public int getOrder() {
167         return -200;
168     }
169
170     public static void main(String[] args) {
171         AuthFilter authFilter = new AuthFilter();
172 //         测试 ?
173         String pattern = "/sys/?bc";
174         System.out.println(authFilter.isMatch(pattern,"/sys/abc"));
175         System.out.println(authFilter.isMatch(pattern,"/sys/cbc"));
176         System.out.println(authFilter.isMatch(pattern,"/sys/acbc"));
177         System.out.println(authFilter.isMatch(pattern,"/sdsa/abc"));
178         System.out.println(authFilter.isMatch(pattern,"/sys/abcw"));
179
180 //         测试*
181 //         String pattern = "/sys/*/bc";
182 //         System.out.println(authFilter.isMatch(pattern,"/sys/a/bc"));
183 //
    System.out.println(authFilter.isMatch(pattern,"/sys/sdasdsadsad/bc"));
184 //         System.out.println(authFilter.isMatch(pattern,"/sys/a/b/bc"));
185 //         System.out.println(authFilter.isMatch(pattern,"/a/b/bc"));
186 //         System.out.println(authFilter.isMatch(pattern,"/sys/a/b/"));
187
188 //         测试**
189 //         String pattern = "/sys/**/bc";
190 //         System.out.println(authFilter.isMatch(pattern, "/sys/a/bc"));
```

```
//        System.out.println(authFilter.isMatch(pattern, "/sys/sdasdsadsad/bc"));
//        System.out.println(authFilter.isMatch(pattern, "/sys/a/b/bc"));
//        System.out.println(authFilter.isMatch(pattern, "/sys/a/b/s/23/432/fdsf///bc"));
//        System.out.println(authFilter.isMatch(pattern, "/a/b/s/23/432/fdsf///bc"));
//        System.out.println(authFilter.isMatch(pattern, "/sys/a/b/s/23/432/fdsf///"));
    }
}
```

- **增加百名带配置类**

```
@Configuration
@RefreshScope
@ConfigurationProperties(prefix = "security.ignore")
public class IgnoreWhiteProperties
{
    /**
     * 放行白名单配置，网关不校验此处的白名单
     */
    private List<String> whites = new ArrayList<>();

    public List<String> getWhites()
    {
        return whites;
    }

    public void setWhites(List<String> whites)
    {
        this.whites = whites;
    }
}
```

- **响应结构中增加方法**

```
public static <T> R<T> fail(int code, String msg) {
    return assembleResult(code, msg, null);
}

private static <T> R<T> assembleResult(int code, String msg, T data) {
    R<T> r = new R<>();
```

```
    r.setCode(code);
    r.setData(data);
    r.setMsg(msg);
    return r;
}
```

- **增加HttpConstants**

```
package com.bite.common.core.constants;

public class HttpConstants {
    /**
     * 服务端url标识
     */
    public static final String SYSTEM_URL_PREFIX = "system";

    /**
     * 用户端url标识
     */
    public static final String FRIEND_URL_PREFIX = "friend";

    /**
     * 令牌自定义标识
     */
    public static final String AUTHENTICATION = "Authorization";

    /**
     * 令牌前缀
     */
    public static final String PREFIX = "Bearer ";
}
```