尚马教育 JAVA 高级课程

JJWT 组件与 token 使用

文档编号：C10_2

创建日期： 2021-02-26

最后修改日期：2021-02-26

版 本 号：V1.0

电子版文件名：尚马教育-第三阶段-10_2.jjwt 组件与 token 使用.docx

文档修改记录：

更新日期	更新作者	更新说明	版本号
2021-02-26	冯勇涛	认识 token，使用 jjwt，项目使用	V1.0

目录

知识点：

1. 掌握 token 概念
2. 掌握 jjwt 组件使用
3. 项目引入 token 进行客户端校验

1. Token 介绍

1.1. 概念

Token 是当用户第一次访问服务端，由服务端生成的一串加密字符串，以作后续客户端进行请求的一个通行令牌。当第一次登录后，服务器生成一个 Token 字符串，并将此字符串返回给客户端，以后客户端请求需要带上这个 Token 发送服务器，进行请求数据即可，无需再次带上用户名和密码。

类似于城主发放的路引，进城需要出示路引做身份证明。

1.2. token 作用

Token 的目的是为了减轻服务器的压力，减少频繁的查询数据库进行客户端校验，使服务器更加健壮。

在 tomcat 集群环境下，解决基于内存的 session 共享问题。

2. Jjwt 组件

生成解析 token 字符串的常用组件有 auth0，jjwt，这里选用 jjwt 进行学习。

JJWT 旨在成为最易于使用和理解的库，用于在 jvm 和 Android 上创建和验证 JSON Web Token（JWT）。对 JWT 进行加密签名后，称为 JWS

官网：https://github.com/jwtk/jjwt

jWT 表示形式是一个字符串，该字符串包含三个部分，每个部分之间都用.进行分隔，每个部分都是 Base64URL 编码的。如下：

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJKb2UifQ.ipevRNuRP6HflG8cFKnmUPtypruRC4fb1
DWtoLL62SY
```

第一部分：header 标头，需要指定用于签署 JWT 的算法

```
{
  "alg": "HS256"
}
```

第二部分：body 身体，jwt 中需要包含的 Claims 认证数据，claims 分为标准 cliams 与自定义。

```
{

  "sub": "Joe"

}
```

第三部分：签名，它是通过将标头和正文的组合通过标头中指定的算法加密来计算的。起到鉴伪作用。

2.1. 添加依赖

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is
preferred -->
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>
```

2.2. 生成秘钥

JWT 规范确定了 12 种标准签名算法：3 种对称密钥算法和 9 种非对称密钥算法。

- HS256：使用 SHA-256 的 HMAC
- HS384：使用 SHA-384 的 HMAC
- HS512：使用 SHA-512 的 HMAC

- ES256：使用 P-256 和 SHA-256 的 ECDSA
- ES384：使用 P-384 和 SHA-384 的 ECDSA
- ES512：使用 P-521 和 SHA-512 的 ECDSA
- RS256：使用 SHA-256 的 RSASSA-PKCS-v1_5
- RS384：使用 SHA-384 的 RSASSA-PKCS-v1_5
- RS512：使用 SHA-512 的 RSASSA-PKCS-v1_5
- PS256：使用 SHA-256 和 MGF1 与 SHA-256 的 RSASSA-PSS
- PS384：使用 SHA-384 和 MGF1 与 SHA-384 的 RSASSA-PSS
- PS512：使用 SHA-512 和 MGF1 与 SHA-512 的 RSASSA-PSS

前三者是对称秘钥算法，后 9 个是非对称算法。

以下代码采用 HS256 对称秘钥算法。

```
//生成符合 HS256 加密算法要求的秘钥，该秘钥要保存为常量，进行后续解密
    public static void generateKey() {
        Key key = Keys.secretKeyFor(SignatureAlgorithm.HS256);
        String secretString = Encoders.BASE64.encode(key.getEncoded());
        System.out.println(secretString);
    }
```

2.3. 生成 token 并加密

以下代码中成的 jwt 中的 body 部分包含如下 claims：

标准 claim：issuedAt 签发时间，expiration 过期时间

自定义 claim：p 手机号

{issuedAt:XXXX,expiration:XXXX,p:XXX}

```
//token 中含有用户信息，并使用秘钥加密
    public static String generate(String uphone) {
        Key key = Keys.hmacShaKeyFor(Decoders.BASE64.decode(keyStr));
        Date current = new Date();
        Date expirDate = new Date(current.getTime() + expirMills);
        Map<String, String> claims = new HashMap<>();
        claims.put("p", uphone);
        String jws =
Jwts.builder().setClaims(claims).setIssuedAt(current).setExpiration(expirDate).signWith(key).co
mpact();
        return jws;
    }
```

2.4. 解析 token

```
//解析 token 字符串，使用秘钥解密
    public static Claims parse(String token) {
        Jws<Claims> jws = Jwts.parserBuilder()
                .setSigningKey(keyStr).build().parseClaimsJws(token);
        return jws.getBody();
    }

  //获取解析后字符串中的手机号签名信息
    public static String getUphone(Claims claim){
        return (String)claim.get("p");
    }

  //获取解析后字符串中的签发时间信息
    public static Date getIssuedAt(Claims claim){
```

```
        return claim.getIssuedAt();
    }

  //获取解析后字符串中的过期时间信息
    public static Date getExpiation(Claims claim){
        return claim.getExpiration();
    }
```

2.5. 测试工具类

首先需要使用生成秘钥方法生成秘钥字符串，保存到静态常量；

```
//保存生成的秘钥
    private static String keyStr = "Y35yyCm16xZ7TwcNIAZRjDtiNc/loypN0GN331O/3r4=";
  //设置超期时间为 1 周
    private static Long expirMills = 7*24*60*60*1000L;


public static void main(String[] args) throws InterruptedException {
  //1.生成 token
    String t = generate("13663830111");
    System.out.println(t);
  //2.解析 token，解析失败抛出异常
    Claims claims = parse(t);
  //3.成功解析后，获取签名信息
    String uphone = getUphone(claims);
    Date expiation = getExpiation(claims);
    System.out.println(uphone+"--"+expiation);
    }
```

3. 登陆接口

3.1. 首次登录服务端生成 token 发送客户端

```
//用户名密码登录
@PostMapping("login")
public ResponseEntity dologin(@RequestBody Sysuser user){
    //根据用户名查询用户对象
    Sysuser loginuser = us.selectUserByUname(user);
    //用户名正确与否
    if(loginuser==null){
        return ResponseEntity.ok(new RB(SE.LOGIN_UNAME_ERROR));
    }
    //判断密码有否有误
    boolean isok = us.comparseUpwd(loginuser,user);
    if(!isok){
        return ResponseEntity.ok(new RB(SE.LOGIN_UPWD_ERROR));
    }
    //登录成功，生成 token
    String token = JwtUtil.generate(loginuser.getUname());

    loginuser.setUpwd("");
    //把 token 放在响应头中返回客户端
    HttpHeaders headers = new HttpHeaders();
    headers.add("token",token);
    return new ResponseEntity(new RB(SE.SUC,loginuser),headers,HttpStatus.OK);
}
```

3.2. Springmvc 配置自定义返回响应头

3.1 代码中添加的自定义响应头 token，前端默认收不到该信息，需要在跨域过滤器中配置自定义响应头。

```
<bean id="corsFilter" class="org.springframework.web.filter.CorsFilter">
        <constructor-arg name="configSource">
        <bean class="org.springframework.web.cors.UrlBasedCorsConfigurationSource">
            <property name="corsConfigurations">
                <map>
                    <entry key="/**">
                        <bean class="org.springframework.web.cors.CorsConfiguration">
                            <property name="allowCredentials" value="true"/>
                            <property name="allowedMethods">
                                <list>
                                    <value>GET</value>
                                    <value>POST</value>
                                    <value>HEAD</value>
                                    <value>PUT</value>
                                    <value>DELETE</value>
                                    <value>OPTIONS</value>
                                </list>
                            </property>
                            <property name="allowedHeaders" value="*"/>
                            <property name="allowedOrigins" value="*"/>
                        <!--允许客户端收到 token 响应头-->
                            <property name="exposedHeaders"
value="token"></property><!--暴露的响应头-->
                        </bean>
                    </entry>
```

```
                    </map>
                </property>
            </bean>
        </constructor-arg>
        </bean>
```

3.3. 客户端接收 token 并保存

在 axios 的后拦截器中获取 token 字符串，并保存起来。

```
//axios 后拦截器
_axios.interceptors.response.use(function(resp) {
    let code = resp.data.code
    if(code==50004){
        router.push("/")
    }

    if('token' in resp.headers){
        //获取响应头中的 token 信息
        let token = resp.headers['token']
        if(token!=null && token!=undefined){
            //把 token 保存 locatlStorage 域
            localStorage.setItem("token",token);
        }
    }
        return resp.data;
    },
    function(error) {
        return Promise.reject(error);
    }
```

```
);
```

3.4. 客户端后续请求携带 token 发送服务端

前端 axios 请求，获取 localStorage 中的 token 加入请求头

```
//axios 前拦截器
_axios.interceptors.request.use(function(config) {
    let url= config.url;
    if(url=='/user/login'){
        return config;
    }else{
        let loginuser = localStorage.getItem("loginuser")
        if(loginuser==null||loginuser==undefined){
                router.push("/");
        }else{
                let token = localStorage.getItem("token")
                if(token==null || token==undefined){
                        router.push("/");
                }else{
                        config.headers['token']=token;
                        return config;
                }
        }
    }
    },
    function(error) {
        return Promise.reject(error);
    }
);
```

3.5. 服务端拦截器进行 token 校验

```
@Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String token = request.getHeader("token");
        System.out.println(token);
        if(token==null)
            throw new WebException(SE.NO_LOGIN);
    //解析 token
        Claims claims = null;
        try{
            claims = JwtUtil.parse(token);
        }catch (Exception e){
            throw new WebException(SE.NO_LOGIN);
        }
        //从 redis 中获取登录用户
        String uname = JwtUtil.getCustomClaim(claims);
        String userStr = rs.get(RedisKey.users + uname);
        if(userStr==null)
            throw new WebException(SE.NO_LOGIN);

    //刷新 token
        String newToken = JwtUtil.generate(uname);
        response.addHeader("token",newToken);

        Sysuser u    = JSON.parseObject(userStr,Sysuser.class);
        //使用线程变量把 u 登录用户对象保存，确保当前线程中都可以取得登录用户
        CurrentLoginUser.setLoginuser(u);
```

```
            return true;
        }
```

```
public class CurrentLoginUser {
    private static ThreadLocal<Sysuser> loginuser = new ThreadLocal<>();
    public static void setLoginuser(Sysuser u){
        loginuser.set(u);
    }
    public static Sysuser getLoginuser(){
        return loginuser.get();
    }
}
```

4. 注销接口

4.1. 用户注销登录

注销操作，需要前端删除 token

```
localStorage.removeItem('token');
router.push("/");
```