

主讲老师: Fox

学习本节课需要基础知识:

1. 理解oauth2协议, 熟悉授权码模式和密码模式
2. 掌握spring security使用
3. 掌握spring security oauth2基本使用

- 1 文档: [28 微服务安全Spring Security OAuth2?..](#)
- 2 链接: <http://note.youdao.com/noteshare?id=91c79470710860cce7e3c8dcf444d0e3&sub=2927370A4DEF4312AAD6207FD751E39D>

1. Spring Security OAuth2实现单点登录

1.1 什么是单点登录

1.2 单点登录常见实现方式

同域单点登录

跨域单点登录

基于OAuth2跨域单点登录

1.3 Spring Security OAuth2单点登录实战

创建客户端

创建授权服务器

1.4 OAuth2整合网关实现微服务单点登录

2. JWT

2.1 什么是JWT

2.2 JWT组成

头部 (header)

载荷 (payload)

签名 (signature)

如何应用

2.3 JJWT

快速开始

创建token

token的验证解析

token过期校验

自定义claims

2.4 Spring Security Oauth2整合JWT

整合JWT

扩展JWT中的存储内容

刷新令牌

1. Spring Security Oauth2实现单点登录

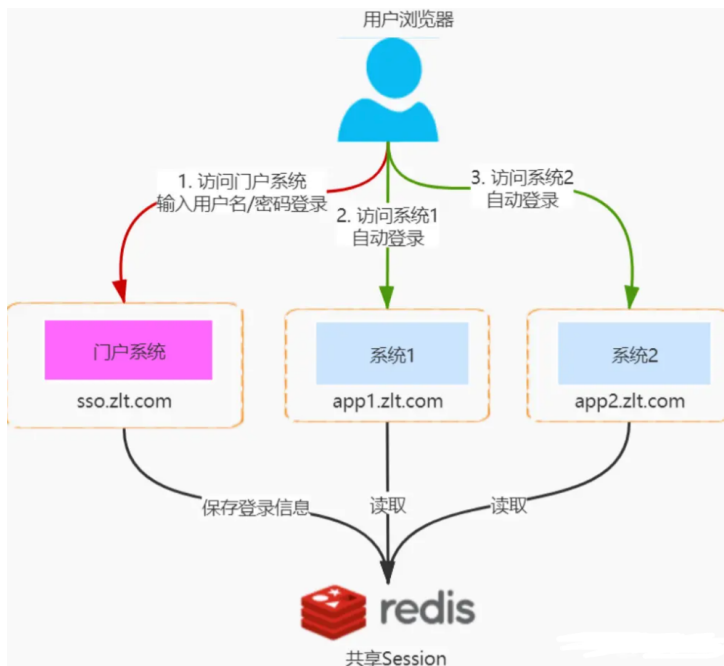
1.1 什么是单点登录

单点登录（Single Sign On），简称为 SSO，是比较流行的企业业务整合的解决方案之一。它的用途在于，不管多么复杂的应用群，只要在用户权限范围内，那么就可以做到，**用户只需要登录一次就可以访问权限范围内的所有应用子系统。**

1.2 单点登录常见实现方式

同域单点登录

适用场景：都是企业自己的系统，所有系统都使用同一个一级域名通过不同的二级域名来区分

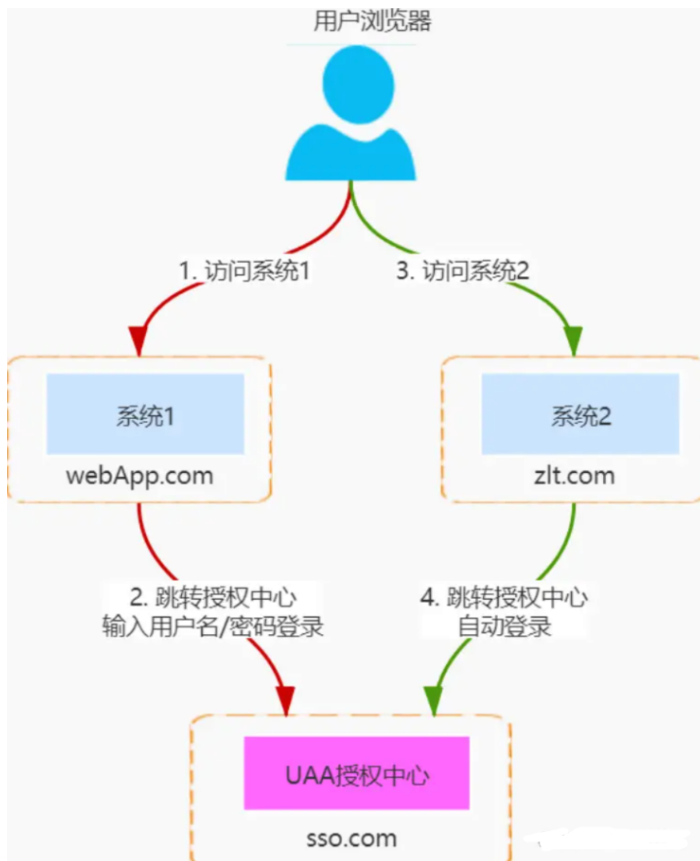


核心原理：

1. 门户系统设置 Cookie 的 domain 为一级域名也就是 zlt.com，这样就可以共享门户的 Cookie 给所有的使用该域名(xxx.zlt.com)的系统
2. 使用 Spring Session 等技术让所有系统共享 Session
3. 这样只要门户系统登录之后无论跳转应用1或者应用2，都能通过门户 Cookie 中的 sessionId 读取到 Session 中的登录信息实现单点登录

跨域单点登录

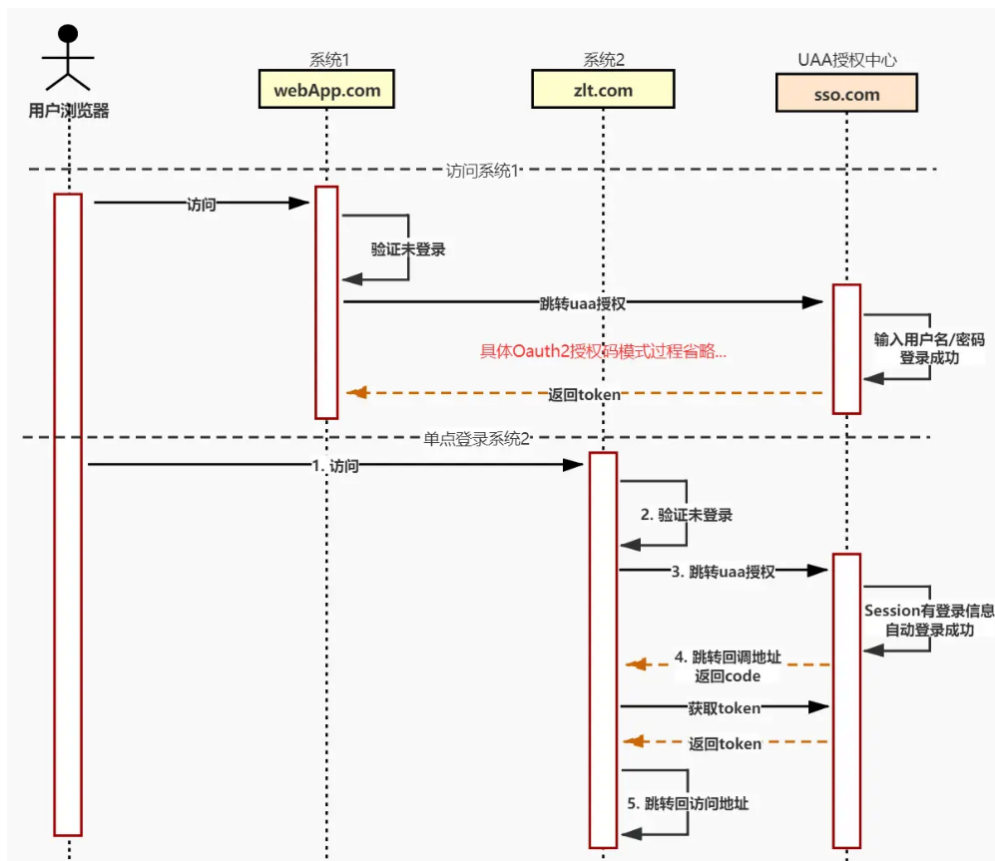
单点登录之间的系统域名不一样，例如第三方系统。由于域名不一样不能共享 Cookie 了，这样就需要通过一个单独的授权服务(UAA)来做统一登录，并基于共享UAA的 Cookie 来实现单点登录。



核心原理:

1. 访问系统1判断未登录，则跳转到UAA系统请求授权
2. 在UAA系统域名 sso.com 下的登录地址中输入用户名/密码完成登录
3. 登录成功后UAA系统把登录信息保存到 Session 中，并在浏览器写入域为 sso.com 的 Cookie
4. 访问系统2判断未登录，则跳转到UAA系统请求授权
5. 由于是跳转到UAA系统的域名 sso.com 下，所以能通过浏览器中UAA的 Cookie 读取到 Session 中之前的登录信息完成单点登录

基于Oauth2跨域单点登录



1.3 Spring Security OAuth2单点登录实战

OAuth2单点登录除了需要授权中心完成统一登录/授权逻辑之外, 各个系统本身(sso客户端)也需要实现以下逻辑:

1. 拦截请求判断登录状态
2. 与 UAA授权中心 通过 OAuth2授权码模式 交互完成登录/单点登录
3. 保存用户登录信息

以上逻辑只需使用一个 @EnableOAuth2Sso 注解即可实现

创建客户端

1) 引入依赖

```

1 <dependencies>
2 <dependency>
3 <groupId>org.springframework.boot</groupId>
4 <artifactId>spring-boot-starter-web</artifactId>
5 </dependency>
6
7 <dependency>
8 <groupId>org.springframework.cloud</groupId>
9 <artifactId>spring-cloud-starter-oauth2</artifactId>

```

```

10 </dependency>
11
12
13 <dependency>
14 <groupId>org.springframework.boot</groupId>
15 <artifactId>spring-boot-starter-test</artifactId>
16 <scope>test</scope>
17 </dependency>
18
19 </dependencies>

```

2) 修改application.properties

```

1 server:
2   port: 8081
3   #防止Cookie冲突，冲突会导致登录验证不通过
4   servlet:
5     session:
6     cookie:
7       name: OAUTH2-CLIENT-SESSIONID${server.port}
8
9   #与授权服务器对应的配置
10  security:
11    oauth2:
12      client:
13        client-id: client
14        client-secret: 123123
15        user-authorization-uri: http://localhost:8080/oauth/authorize
16        access-token-uri: http://localhost:8080/oauth/token
17        resource:
18        token-info-uri: http://localhost:8080/oauth/check_token

```

3) 在启动类上添加@EnableOAuth2Sso注解来启用单点登录功能

@EnableOAuth2Sso单点登录的原理简单来说就是：标注有@EnableOAuth2Sso的OAuth2 Client应用在通过某种OAuth2授权流程获取访问令牌后（一般是授权码流程），通过访问令牌访问userDetails用户明细这个受保护资源服务，获取用户信息后，将用户信息转换为Spring Security上下文中的认证后凭证Authentication，从而完成标注有@EnableOAuth2Sso的OAuth2 Client应用自身的登录认证的过程。整个过程是基于OAuth2的SSO单点登录

```

1 @SpringBootApplication
2 @EnableOAuth2Sso
3 public class OAuth2SsoClientDemoApplication {
4

```

```

5 public static void main(String[] args) {
6     SpringApplication.run(Oauth2SsoClientDemoApplication.class, args);
7 }
8
9 }

```

4) 添加接口用于获取当前登录用户信息

```

1 @RestController
2 @RequestMapping("/user")
3 public class UserController {
4
5     @RequestMapping("/getCurrentUser")
6     public Object getCurrentUser(Authentication authentication) {
7         return authentication;
8     }
9 }

```

创建授权服务器

1) 修改授权服务器中的AuthorizationServerConfig类

```

1 @Configuration
2 @EnableAuthorizationServer
3 public class AuthorizationServerConfig3 extends AuthorizationServerConfigur
erAdapter {
4
5     @Autowired
6     private PasswordEncoder passwordEncoder;
7
8     @Override
9     public void configure(AuthorizationServerSecurityConfigurer security) thro
ws Exception {
10         //允许表单认证
11         security.allowFormAuthenticationForClients();
12         //校验token
13         security.checkTokenAccess("permitAll()");
14
15     }
16
17     @Override
18     public void configure(ClientDetailsServiceConfigurer clients) throws Exce
ption {
19
20         /**

```

```
21  *授权码模式
22  */
23  clients.inMemory()
24  //配置client_id
25  .withClient("client")
26  //配置client-secret
27  .secret(passwordEncoder.encode("123123"))
28  //配置访问token的有效期
29  .accessTokenValiditySeconds(3600)
30  //配置刷新token的有效期
31  .refreshTokenValiditySeconds(86400)
32  //配置redirect_uri, 用于授权成功后跳转
33  .redirectUri("http://localhost:8081/login",
34  "http://localhost:8082/login")
35  //自动授权配置
36  .autoApprove(true)
37  //配置申请的权限范围
38  .scopes("all")
39  /**
40  * 配置grant_type, 表示授权类型
41  * authorization_code: 授权码模式
42  */
43  .authorizedGrantTypes("authorization_code");
44  }
45
46
47  }
```

2) 启动授权服务和客户端服务

测试：访问客户端需要授权的接口<http://localhost:8081/user/getCurrentUser>
会跳转到授权服务的登录界面


```
{
  - authorities: [
    - {
      authority: "admin"
    }
  ],
  - details: {
    remoteAddress: "0:0:0:0:0:0:1",
    sessionId: "352DB59C161F8A8E33D0FFDE8928D095",
    tokenValue:
      "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2VyX25hbWUiOiJmb3giLCJzY29wZSI6WyJhbGwiXSwiZXhwI
      k0NyuiZnxBMsP2-8MLF0",
    tokenType: "bearer",
    decodedDetails: null
  },
  authenticated: true,
  - userAuthentication: {
    - authorities: [
      - {
        authority: "admin"
      }
    ],
    details: null,
    authenticated: true,
    principal: "fox",
    credentials: "N/A",
    name: "fox"
  },
  clientOnly: false,
  principal: "fox",
  credentials: "",
  - oauth2Request: {
    clientId: "client",
    - scope: [
      "all"
    ]
  }
}
```

userAuthentication

授权后会跳转到原来需要权限的接口地址，展示登录用户信息

← → ↻ localhost:8081/user/getCurrentUser

```
{
  - authorities: [
    - {
      authority: "admin"
    }
  ],
  - details: {
    remoteAddress: "0:0:0:0:0:0:1",
    sessionId: "352DB59C161F8A8E33D0FFDE8928D095",
    tokenValue:
      "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJmb3giLCJzY29wZSI6WyJhbGwiXSwiZXhwI
      kONyuiZnxBMsP2-8MLF0",
    tokenType: "bearer",
    decodedDetails: null
  },
  authenticated: true,
  - userAuthentication: {
    - authorities: [
      - {
        authority: "admin"
      }
    ],
    details: null,
    authenticated: true,
    principal: "fox",
    credentials: "N/A",
    name: "fox"
  },
  clientOnly: false,
  principal: "fox",
  credentials: "",
  - oauth2Request: {
    clientId: "client",
    - scope: [
      "all"
    ]
  }
}
```

userAuthentication

3) 模拟两个客户端8081, 8082

修改application.properties配置

```
1 server.port=8082
2 #防止Cookie冲突，冲突会导致登录验证不通过
3 server.servlet.session.cookie.name=OAUTH2-CLIENT-SESSIONID${server.port}
```

修改授权服务器配置，配置多个跳转路径

```
1 //配置redirect_uri，用于授权成功后跳转
2 .redirectUri("http://localhost:8081/login",
3   "http://localhost:8082/login")
```

8081登录成功之后，8082无需再次登录就可以访问

<http://localhost:8082/user/getCurrentUser>

1.4 Oauth2整合网关实现微服务单点登录

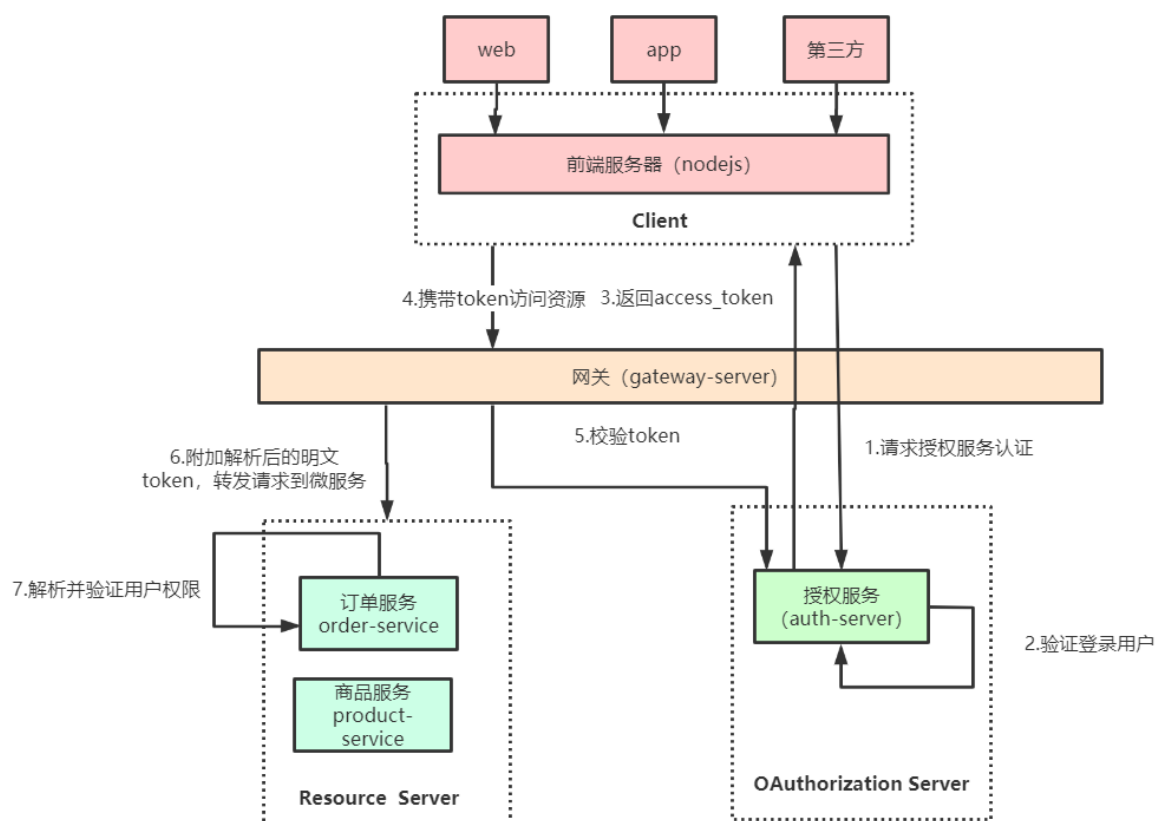
网关整合 OAuth2.0 有两种思路，一种是授权服务器生成令牌，所有请求统一在网关层验证，判断权限等操作；另一种是由各资源服务处理，网关只做请求转发。比较常用的是第一种，把API网关作为OAuth2.0的资源服务器角色，实现接入客户端权限拦截、令牌解析并转发当前登录用户信息给微服务，这样下游微服务就不需要关心令牌格式解析以及OAuth2.0相关机制了。

网关在认证授权体系里主要负责两件事：

- (1) 作为OAuth2.0的资源服务器角色，实现接入方权限拦截。
- (2) 令牌解析并转发当前登录用户信息（明文token）给微服务

微服务拿到明文token(明文token中包含登录用户的身份和权限信息)后也需要做两件事：

- (1) 用户授权拦截（看当前用户是否有权访问该资源）
- (2) 将用户信息存储进当前线程上下文（有利于后续业务逻辑随时获取当前用户信息）



核心代码，网关自定义全局过滤器进行身份认证

```
1 @Component
2 @Order(0)
3 public class AuthenticationFilter implements GlobalFilter, InitializingBean
4 {
```

```
4
5 @Autowired
6 private RestTemplate restTemplate;
7
8 private static Set<String> shouldSkipUrl = new LinkedHashSet<>();
9 @Override
10 public void afterPropertiesSet() throws Exception {
11     // 不拦截认证的请求
12     shouldSkipUrl.add("/oauth/token");
13     shouldSkipUrl.add("/oauth/check_token");
14     shouldSkipUrl.add("/user/getCurrentUser");
15 }
16
17 @Override
18 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
19
20     String requestPath = exchange.getRequest().getURI().getPath();
21
22     //不需要认证的url
23     if(shouldSkip(requestPath)) {
24         return chain.filter(exchange);
25     }
26
27     //获取请求头
28     String authHeader = exchange.getRequest().getHeaders().getFirst("Authorization");
29
30     //请求头为空
31     if(StringUtils.isEmpty(authHeader)) {
32         throw new RuntimeException("请求头为空");
33     }
34
35     TokenInfo tokenInfo=null;
36     try {
37         //获取token信息
38         tokenInfo = getTokenInfo(authHeader);
39     }catch (Exception e) {
40         throw new RuntimeException("校验令牌异常");
41     }
42     exchange.getAttributes().put("tokenInfo",tokenInfo);
43     return chain.filter(exchange);
```

```

44     }
45
46     private boolean shouldSkip(String reqPath) {
47
48         for(String skipPath:shouldSkipUrl) {
49             if(reqPath.contains(skipPath)) {
50                 return true;
51             }
52         }
53         return false;
54     }
55
56     private TokenInfo getTokenInfo(String authHeader) {
57         // 获取token的值
58         String token = StringUtils.substringAfter(authHeader, "bearer ");
59
60         HttpHeaders headers = new HttpHeaders();
61         headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
62         headers.setBasicAuth(MDA.clientId, MDA.clientSecret);
63
64         MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
65         params.add("token", token);
66
67         HttpEntity<MultiValueMap<String, String>> entity = new HttpEntity<>(params, headers);
68
69         ResponseEntity<TokenInfo> response = restTemplate.exchange(MDA.checkTokenUrl, HttpMethod.POST, entity, TokenInfo.class);
70
71         return response.getBody();
72     }
73 }

```

2. JWT

OAuth 2.0是当前业界标准的授权协议，它的核心是若干个针对不同场景的令牌颁发和管理流程；而JWT是一种轻量级、自包含的令牌，可用于在微服务间安全地传递用户信息。

2.1 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:

JWT令牌较长, 占存储空间比较大。

2.2 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) 。

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

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

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  fox)
```

fox

☐ secret base64 encoded

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：

```
1 {  
2   "alg": "HS256",  
3   "typ": "JWT"  
4 }
```

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：

```
1 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：

```
1 {  
2   "sub": "1234567890",  
3   "name": "John Doe",  
4   "iat": 1516239022  
5 }
```

然后将其进行base64加密，得到Jwt的第二部分：

```
1 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：

```
1 var encodedString = base64UrlEncode(header) + '.' +  
  base64UrlEncode(payload);  
2  
3 var signature = HMACSHA256(encodedString, 'fox'); // khA7TNYc7_0iELcDyTc7gH  
  BZ_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:

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.khA7TNYc7_0iELcDyTc7gHBZ_xfIcgbfpzUNWwQtzME
```

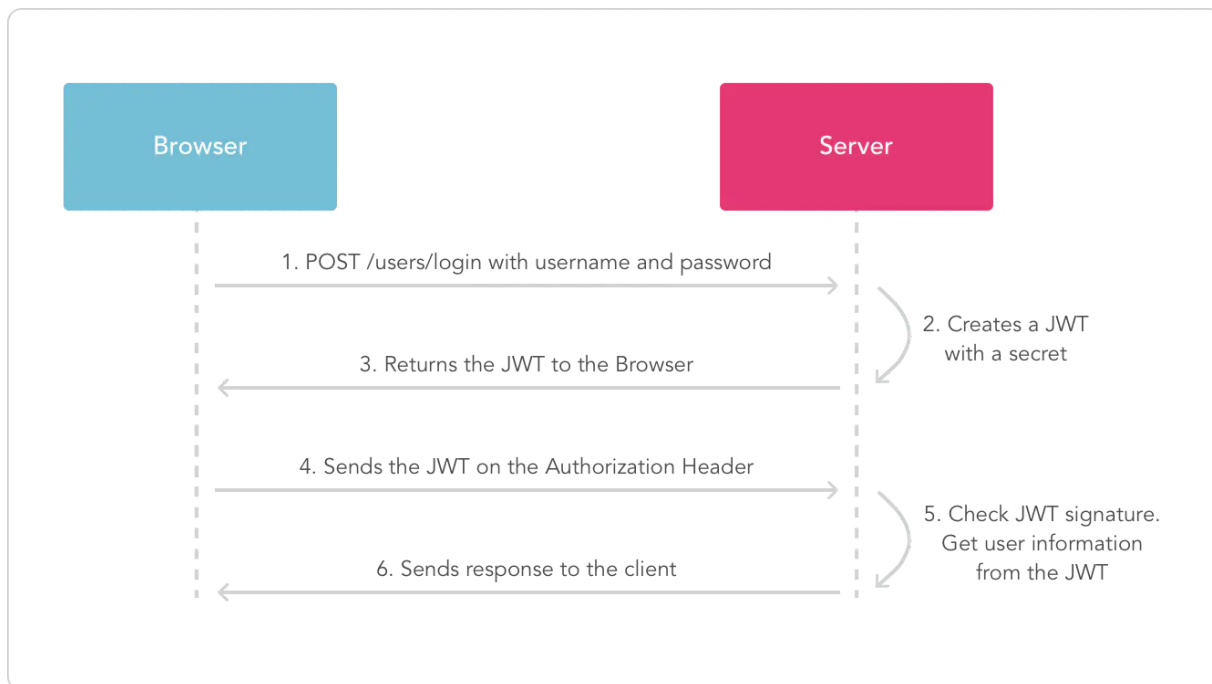
注意：secret是保存在服务器端的，jwt的签发生成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

如何应用

一般是在请求头里加入Authorization，并加上Bearer标注：

```
1 fetch('api/user/1', {  
2   headers: {  
3     'Authorization': 'Bearer ' + token  
4   }  
5 })
```

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：



2.3 JJWT

JJWT是一个提供端到端的JWT创建和验证的Java库，永远免费和开源(Apache License，版本2.0)。JJW很容易使用和理解。它被设计成一个以建筑为中心的流畅界面，隐藏了它的大部分复杂性。

快速开始

引入依赖

```
1 <!-- JWT 依赖 -->
2 <dependency>
3   <groupId>io.jsonwebtoken</groupId>
4   <artifactId>jjwt</artifactId>
5   <version>0.9.1</version>
6 </dependency>
```

创建token

创建测试类，生成token

```
1 @Test
2 public void test() {
3   //创建一个JwtBuilder对象
4   JwtBuilder jwtBuilder = Jwts.builder()
5   //声明的标识{"jti":"666"}
6   .setId("666")
7   //主体，用户{"sub":"Fox"}
8   .setSubject("Fox")
9   //创建日期{"ita":"xxxxxx"}
```

```

10 .setIssuedAt(new Date())
11 //签名手段, 参数1: 算法, 参数2: 盐
12 .signWith(SignatureAlgorithm.HS256, "123123");
13 //获取token
14 String token = jwtBuilder.compact();
15 System.out.println(token);
16
17 //三部分的base64解密
18 System.out.println("=====");
19 String[] split = token.split("\\.");
20 System.out.println(Base64Codec.BASE64.decodeToString(split[0]));
21 System.out.println(Base64Codec.BASE64.decodeToString(split[1]));
22 //无法解密
23 System.out.println(Base64Codec.BASE64.decodeToString(split[2]));
24 }

```

运行结果

```

eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI2NjYiLCJzdWIiOiJGb3giLCJpYXQiOiJlNDh9.Hz7tk6pJaest_jxFrJ4BWiMg3HQxjwY9cGmJ4GQwfuU
=====
{"alg":"HS256"}
{"jti":"666","sub":"Fox","iat":1608272548}

```

token的验证解析

在web应用中由服务端创建了token然后发给客户端, 客户端在下次向服务端发送请求时需要携带这个token (这就好像是拿着一张门票一样), 那服务端接到这个token应该解析出token中的信息 (例如用户id), 根据这些信息查询数据库返回相应的结果。

```

1 @Test
2 public void testParseToken(){
3     //token
4     String token = "eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI2NjYiLCJzdWIiOiJGb3giLCJpYXQiOiJlNDh9" +
5         ".Hz7tk6pJaest_jxFrJ4BWiMg3HQxjwY9cGmJ4GQwfuU";
6     //解析token获取载荷中的声明对象
7     Claims claims = Jwts.parser()
8         .setSigningKey("123123")
9         .parseClaimsJws(token)
10        .getBody();
11
12     System.out.println("id:"+claims.getId());
13     System.out.println("subject:"+claims.getSubject());
14     System.out.println("issuedAt:"+claims.getIssuedAt());

```

```
id:666
subject:Fox
issuedAt:Fri Dec 18 14:22:28 GMT+08:00 2020
```

试着将token或签名密钥篡改一下，会发现运行时就会报错，所以解析token也就是验证token

```
io.jsonwebtoken.SignatureException: JWT signature does not match locally computed signature. JWT validity cannot be asserted and should not be trusted
    at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:354)
    at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:481)
    at io.jsonwebtoken.impl.DefaultJwtParser.parseClaimsJws(DefaultJwtParser.java:541)
    at com.luban.jwtdemo.JwtDemoApplicationTests.testParseToken(JwtDemoApplicationTests.java:49) <31 internal calls>
    at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>
    at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>
```

token过期校验

有很多时候，我们并不希望签发的token是永久生效的，所以我们可以为token添加一个过期时间。原因：从服务器发出的token，服务器自己并不做记录，就存在一个弊端：服务端无法主动控制某个token的立刻失效。

```
JwtBuilder jwtBuilder = Jwts.builder()
    //声明的标识{"jti":"666"}
    .setId("666")
    //主体，用户{"sub":"Fox"}
    .setSubject("Fox")
    //创建日期{"ita":"xxxxxx"}
    .setIssuedAt(new Date())
    //设置过期时间 1分钟
    .setExpiration(new Date(System.currentTimeMillis()+60*1000))
    //签名手段，参数1：算法，参数2：盐
    .signWith(SignatureAlgorithm.HS256, base64EncodedSecretKey: "123123");
```

当未过期时可以正常读取，当过期时会引发io.jsonwebtoken.ExpiredJwtException异常。

```
io.jsonwebtoken.ExpiredJwtException: JWT expired at 2020-12-18T15:18:49Z. Current time: 2020-12-18T15:19:09Z, a difference of 20908 milliseconds.
    at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:385)
    at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:481)
    at io.jsonwebtoken.impl.DefaultJwtParser.parseClaimsJws(DefaultJwtParser.java:541)
    at com.luban.jwtdemo.JwtDemoApplicationTests.testParseToken(JwtDemoApplicationTests.java:52) <31 internal calls>
    at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>
    at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>
```

自定义claims

我们刚才的例子只是存储了id和subject两个信息，如果你想存储更多的信息（例如角色）可以自定义claims。

```
1 @Test
2 public void test() {
3     //创建一个JwtBuilder对象
4     JwtBuilder jwtBuilder = Jwts.builder()
```

```
5 //声明的标识{"jti":"666"}
6 .setId("666")
7 //主体, 用户{"sub":"Fox"}
8 .setSubject("Fox")
9 //创建日期{"ita":"xxxxxx"}
10 .setIssuedAt(new Date())
11 //设置过期时间 1分钟
12 .setExpiration(new Date(System.currentTimeMillis()+60*1000))
13 //直接传入map
14 // .addClaims(map)
15 .claim("roles","admin")
16 .claim("logo","xxx.jpg")
17 //签名手段, 参数1: 算法, 参数2: 盐
18 .signWith(SignatureAlgorithm.HS256, "123123");
19 //获取token
20 String token = jwtBuilder.compact();
21 System.out.println(token);
22
23 //三部分的base64解密
24 System.out.println("=====");
25 String[] split = token.split("\\.");
26 System.out.println(Base64Codec.BASE64.decodeToString(split[0]));
27 System.out.println(Base64Codec.BASE64.decodeToString(split[1]));
28 //无法解密
29 System.out.println(Base64Codec.BASE64.decodeToString(split[2]));
30 }
31
32 @Test
33 public void testParseToken(){
34 //token
35 String token = "eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI2NjYiLCJzdWIiOiJGb3giLCJpYXQiOiJlMDgyNzYzMTUsImV4cCI6MTYwODI3NjM3NSwicm9sZXMiOiJhZG1pbiIsImxvZ28iOiJ4eHguanBnIn0.Geg2tmkmJ9iWCWdvZNE3jRSfRaXaR4P3kiPDG3Lb0z4";
36 //解析token获取载荷中的声明对象
37 Claims claims = Jwts.parser()
38 .setSigningKey("123123")
39 .parseClaimsJws(token)
40 .getBody();
41
42 System.out.println("id:"+claims.getId());
43 System.out.println("subject:"+claims.getSubject());
44 System.out.println("issuedAt:"+claims.getIssuedAt());
```

```

45
46   DateFormat sf =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
47   System.out.println("签发时间:"+sf.format(claims.getIssuedAt()));
48   System.out.println("过期时间:"+sf.format(claims.getExpiration()));
49   System.out.println("当前时间:"+sf.format(new Date()));
50
51   System.out.println("roles:"+claims.get("roles"));
52   System.out.println("logo:"+claims.get("logo"));
53 }

```

结果

```

id:666
subject:Fox
issuedAt:Fri Dec 18 15:25:15 GMT+08:00 2020
签发时间:2020-12-18 15:25:15
过期时间:2020-12-18 15:26:15
当前时间:2020-12-18 15:25:41
roles:admin
logo:xxx.jpg

```

2.4 Spring Security Oauth2整合JWT

整合JWT

在之前的spring security Oauth2的代码基础上修改

引入依赖

```

1 <dependency>
2   <groupId>org.springframework.security</groupId>
3   <artifactId>spring-security-jwt</artifactId>
4   <version>1.0.9.RELEASE</version>
5 </dependency>
6

```

添加配置文件JwtTokenStoreConfig.java

```

1 @Configuration
2 public class JwtTokenStoreConfig {
3
4   @Bean
5   public TokenStore jwtTokenStore(){
6     return new JwtTokenStore(jwtAccessTokenConverter());
7   }

```

```
8
9 @Bean
10 public JwtAccessTokenConverter jwtAccessTokenConverter(){
11     JwtAccessTokenConverter accessTokenConverter = new
12     JwtAccessTokenConverter();
13     //配置JWT使用的秘钥
14     accessTokenConverter.setSigningKey("123123");
15     return accessTokenConverter;
16 }
17 }
```

在授权服务器配置中指定令牌的存储策略为JWT

```

@Autowired
@Qualifier("jwtTokenStore")
private TokenStore tokenStore;

@Autowired
private JwtAccessTokenConverter jwtAccessTokenConverter;

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    endpoints.authenticationManager(authenticationManagerBean) //使用密码模式需要配置
        .tokenStore(tokenStore) //配置存储令牌策略
        .accessTokenConverter(jwtAccessTokenConverter)
        .reuseRefreshTokens(false) //refresh token 是否重复使用
        .userService(userService) //刷新令牌授权包含对用户信息的检查
        .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST); //支持GET, POST请求
}

```

用密码模式测试

[illegible]

发现获取到的令牌已经变成了JWT令牌，将access_token拿到<https://jwt.io/> 网站上去解析下可以获得其中内容。

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MDgyODMxMDQsInVzZXJfbmFtZSI6ImZveCIsImF1dGhvcm10aWVzIjpbImFkbWluIiwiaWF0IjE6IjE2MDY2YWI0ZmUxZDFkOSIsImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3BlIjpbImFsbCJdfQ.mB-ial8k21B0tyUeoWSWOSz4
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "exp": 1608283104,
  "user_name": "fox",
  "authorities": [
    "admin"
  ],
  "jti": "9361c40a-9e26-46f2-b57d-66ab4fe1d1d9",
  "client_id": "client",
  "scope": [
    "all"
  ]
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  fox
) ☐ secret base64 encoded
```

扩展JWT中的存储内容

有时候我们需要扩展JWT中存储的内容，这里我们在JWT中扩展一个 key 为 enhance，value 为 enhance info 的数据。

继承TokenEnhancer实现一个JWT内容增强器

```
1 public class JwtTokenEnhancer implements TokenEnhancer {
2
3     @Override
4     public OAuth2AccessToken enhance(OAuth2AccessToken accessToken,
5     OAuth2Authentication authentication) {
6         Map<String, Object> info = new HashMap<>();
7         info.put("enhance", "enhance info");
8         ((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(info);
9         return accessToken;
10    }
11 }
```

创建一个JwtTokenEnhancer实例

```
1 @Bean
2 public JwtTokenEnhancer jwtTokenEnhancer() {
3     return new JwtTokenEnhancer();
4 }
```


在授权服务器配置中配置JWT的内容增强器

```
1 @Autowired
2 private JwtTokenEnhancer jwtTokenEnhancer;
3
4 @Override
5 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
6     //配置JWT的内容增强器
7     TokenEnhancerChain enhancerChain = new TokenEnhancerChain();
8     List<TokenEnhancer> delegates = new ArrayList<>();
9     delegates.add(jwtTokenEnhancer);
10    delegates.add(jwtAccessTokenConverter);
11    enhancerChain.setTokenEnhancers(delegates);
12    endpoints.authenticationManager(authenticationManagerBean) //使用密码模式需要配置
13    .tokenStore(tokenStore) //配置存储令牌策略
14    .accessTokenConverter(jwtAccessTokenConverter)
15    .tokenEnhancer(enhancerChain) //配置tokenEnhancer
16    .reuseRefreshTokens(false) //refresh_token是否重复使用
17    .userDetailsService(userService) //刷新令牌授权包含对用户信息的检查
18    .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST); //支持GET, POST请求
19 }
```

运行项目后使用密码模式来获取令牌，之后对令牌进行解析，发现已经包含扩展的内容。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJmb3giLCJzY29wZSI6WyJhbGwiXSwiZXhwIjoxNjA4Mjg0NDQ4LCJhdXRob3JpdG1lcYI6WyJhZG1pbjJdLCJqdGkiOiIwOGEzNDkzMy1hMGRiLTQyNzYyMGEzNy0yNWE5ZmF1NTQwY2YiLCJjbG1bnRfaWQiOiJjbG1bnRfaWQiLCJlbmhhbmN1IjoiaW50YW5jZSBpbmZvIn0.kKV43kR_T8lyAUhWjnm1aA4aL4Mie8NdvzZTIu5GYhw
```

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "user_name": "fox",
  "scope": [
    "all"
  ],
  "exp": 1608284408,
  "authorities": [
    "admin"
  ],
  "jti": "09a34933-a0db-4273-8a37-25a9fae540cf",
  "client_id": "client",
  "enhance": "enhance info"
}
```

解析JWT

添加依赖

```
1 <!--JWT依赖-->
2 <dependency>
```



```
3 <groupId>io.jsonwebtoken</groupId>
4 <artifactId>jjwt</artifactId>
5 <version>0.9.1</version>
6 </dependency>
```

修改UserController类，使用jjwt工具类来解析Authorization头中存储的JWT内容

```
1 @GetMapping("/getCurrentUser")
2 public Object getCurrentUser(Authentication authentication,
3     HttpServletRequest request) {
4     String header = request.getHeader("Authorization");
5     String token = null;
6     if(header!=null){
7         token = header.substring(header.indexOf("bearer") + 7);
8     }else {
9         token = request.getParameter("access_token");
10    }
11    return Jwts.parser()
12        .setSigningKey("123123".getBytes(StandardCharsets.UTF_8))
13        .parseClaimsJws(token)
14        .getBody();
15 }
```

将令牌放入Authorization头中，访问如下地址获取信息：

<http://localhost:8080/user/getCurrentUser>

The screenshot shows a REST client interface with the following details:

- URL:** http://localhost:8080/user/getCurrentUser
- Method:** GET
- Headers:** Authorization: bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJm...
- Body:** A JSON object containing user information:

```
{  "user_name": "fox",  "scope": [    "all"  ],  "exp": 1608284859,  "authorities": [    "admin"  ],  "jti": "57a9a7fa-9e52-41eb-a9b7-b31a2f4d3f01",  "client_id": "client",  "enhance": "enhance info"}
```

刷新令牌

<http://localhost:8080/oauth/token?>

[grant_type=refresh_token&client_id=client&client_secret=123123&refresh_token=\[refresh_token值\]](http://localhost:8080/oauth/token?grant_type=refresh_token&client_id=client&client_secret=123123&refresh_token=[refresh_token值])

```
{  
  "access_token":  
    "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyZS25hbWUiOiJmb3giLCJzY29wZSI6WyJibGwiXSwiZXNvbjA4Mjg3NTQxLjEjLmVudCJqdGkiOiJlM2ZhOTUwZj00NDt-Zabvwow_4X7h_A0qnUGhBug",  
  "token_type": "bearer",  
  "refresh_token":  
    "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyZS25hbWUiOiJmb3giLCJzY29wZSI6WyJibGwiXSwiZXNvbjA4Mjg3NTQxLjEjLmVudCJqdGkiOiJlM2ZhOTUwZj00NDt-Zabvwow_4X7h_A0qnUGhBug",  
  "expires_in": 3599,  
  "scope": "all",  
  "enhance": "enhance info",  
  "jti": "e3fa950f-47e5-4961-9d93-2acefe90b15a"  
}
```