2、JWT(json web token)的详解

（0）参考

- https://blog.csdn.net/weixin_44736637/article/details/124005231

（1）概念

- JWT，json web token，又叫JSON格式的互联网令牌。

（2）出现原因？

- 是为了给多种终端设备（桌面应用、移动端应用、平板应用、智能家居）提供统一的、安全的JWT令牌，仅仅是一个令牌而已。

（3）传输流程

- 服务端通过jwt工具生成JWT令牌，将JWT令牌响应到客户端的cookie、localstorage、手机文件、PC文件中，其实可以响应到任何位置，这个没有限制。同样的，客户端的传输，可以使用任何方式来传输JWT令牌给服务端，一般的会使用消息头来传输JWT令牌给服务端（只要客户端和服务器自行约定好即可，尽管这会增加额外的传输量）
- 客户端请求时，只需要将令牌作为请求的一部分发送到服务器，服务器通过jwt工具对JWT令牌验证，验证是一个合法并且没被篡改的JWT令牌才算是有效令牌。

（4）JWT令牌

（4-1）组成

复制代码

```
//一个完整的JWT令牌（TOKEN），分成三部分，都是编码后的结果
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJpYXQiOjE1ODc1NDgyMTV9.BCwUy3jnUQ_E6TqCayc7rCHkx-vxxdagUwPOWqwYCFc

```

笔记目录

```
//编码后的header
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
//编码后的payload
eyJmb28iOiJiYXIiLCJpYXQiOjE1ODc1NDgyMTV9
//编码后的signature
BCwUy3jnUQ_E6TqCayc7rCHkx-vxxdagUwPOWqwYCFc
```

java

（4-2）JWT令牌每部分编码前的讲解

复制代码

```
//编码前的header
header：令牌头部，记录了整个令牌的类型和签名算法
//编码前的payload
payload：令牌负荷，记录了保存的主体信息，比如你要保存的用户信息就可以放到这里.
//编码前的signature
signature：令牌签名，按照头部固定的签名算法对整个令牌进行签名，该签名的作用是：保证令牌不被伪造和篡改
```

java

- 编码前的header
  - 令牌的头部，记录了整个令牌的类型和签名算法，它的格式是一个json格式

复制代码

```
{
  "alg":"HS256",
 "typ":"JWT"
}
//解释
alg：signature部分使用的签名算法，通常可以取两个值
    HS256：一种对称加密算法，使用同一个秘钥对signature加密解密
```

```
    RS256：一种非对称加密算法，使用私钥加密，公钥解密
    通过密钥secret，生成key
typ：整个令牌的类型，固定写JWT即可
```

java

- 设置好令牌头部后，就可以进行编码，生成编码后的header值了，生成方式很简单，就是把header部分使用base64 url编码即可得到生成的header信息，base64 url不是一个加密算法，而是一种编码方式，它是在base64算法的基础上对+、=、/三个字符做出特殊处理的算法，而base64是使用64个可打印字符来表示一个二进制数据，具体的做法参考百度百科。

复制代码

```java
//①浏览器提供了btoa函数，进行编码
window.btoa(JSON.stringify({
  "alg":"HS256",
  "typ":"JWT"
}))
// 得到字符串：eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9


//②浏览器也提供了atob函数，可以对其进行解码
window.atob("eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9")
// 得到字符串：{"alg":"HS256","typ":"JWT"}

```

java

- 编码前的payload
  - 令牌主体，它仍然是一个JSON对象，以下属性可以全写，也可以一个都不写，它只是一个规范，就算写了，也需要你在将来验证这个jwt令牌时手动处理才能发挥作用

复制代码

```java
{
  "ss": "发行者",
```

```
    "iat": "发布时间",
    "exp": "到期时间",
    "sub": "主题",
    "aud": "听众",
    "nbf": "在此之前不可用",
  "jti": "JWT ID"
}

//解释
//ss: 发行该jwt的是谁，可以写公司名字，也可以写服务名称
//iat: 该jwt的发放时间，通常写当前时间的时间戳
//exp: 该jwt的到期时间，通常写时间戳
//sub: 该jwt是用于干嘛的
//aud: 该jwt是发放给哪个终端的，可以是终端类型，也可以是用户名称，随意一点
//nbf: 一个时间点，在该时间点到达之前，这个令牌是不可用的
//jti: jwt的唯一编号，设置此项的目的，主要是为了防止重放攻击（重放攻击是在某些场景下，用户使用之前的令牌发送到服务器，被服务器正确的识别，从而导致不可预期的行为发生）
```

java

- 设置好令牌主体后，然后就将令牌主体进行编码生成编码后的payload值了
  - 其实就是当用户登陆成功之后，我可能需要把用户的一些信息写入到令牌主体中，比如用户id、用户对象等等，密码就算了。其实你可以向对象中加入任何想要加入的信息，然后进行编码生成编码后的payload值。

复制代码

```
//这也是一个有效的令牌主体
//foo: bar是我们自定义的信息
//iat: 1587548215是jwt规范中的信息

window.btoa(JSON.stringify({
  "foo":"bar",
```

```
  "iat":1587548215
}))
// 得到字符串：eyJmb28iOiJiYXIiLCJpYXQiOjE1ODc1NDgyMTV9
```

java

- 编码前的signature
  - 令牌签名，正是它的存在，保证了整个JWT令牌不会被篡改
  - 令牌签名的格式，是header和payload编码后的结果和header部分指定的加密方式组成的，当然你还得指定一个secret秘钥，比如固定的一个字符串shhhhh

复制代码

```
HS256(`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJpYXQiOjE1ODc1NDgyMTV9`, "shhhhh")

// 得到：BCwUy3jnUQ_E6TqCayc7rCHkx-vxxdagUwPOWqwYCFc
```

java

  - 按照header中的加密算法，对header编码后的结果、payload编码后的结果、密钥一块进行加密，得到的值就是signature签名。
- 最终，将编码后header.编码后的payload.编码后signature三部分组合在一起，就得到了完整的JWT令牌（TOKEN）

（5）JWT令牌如何保证不会被伪造（相同的一份）？

- 由于编码前的signature使用的秘钥保存在服务器，这样一来，客户端就无法伪造出签名，因为它拿不到秘钥，换句话说，之所以说无法伪造jwt，就是因为第三部分的存在，而前面两部分并没有加密，只是一个编码结果而已，可以认为几乎是明文传输，这不会造成太大的问题，因为既然用户登陆成功了，它当然有权力查看自己的用户信息，甚至在某些网站，用户的基本信息可以被任何人查看，你要保证的，是不要把敏感的信息存放到jwt中，比如密码。

（6）JWT令牌如何保证不会被篡改（是一个合法的JWT令牌，但不是当时生成的那一个令牌）？

- 比如，某个用户登陆成功了，生成了jwt返回给客户端，但是下一次客户端请求之前，他自己人为的篡改了payload，比如把自己的账户余额修改为原来的两倍，然后重新编码出payload再发送到服务器，服务器如何得知这些信息被篡改过了呢？
- 这就要说到令牌的验证了，服务器要验证这个令牌是否被篡改过，验证方式非常简单，就是对header+payload用同样的秘钥和加密算法进行重新加密，然后把加密的结果和客户端请求时jwt令牌中的signature进行对比，如果完全相同，则表示前面两部分没有动过，就是自己颁发的，如果不同，肯定是被篡改过了。

复制代码

```java
//客户端JWT令牌
请求中传入的header.请求中传入的payload.请求中传入的signature

//服务端根据请求中的header和payload重新进行加密计算，得到新的签名
val 新的signature = header中的加密算法(请求中传入的header.请求中传入的payload,秘钥)

//验证：这个新的签名和请求中的签名一致的话，代表JWT令牌没有被篡改
Assertions.assertEquals(请求中传入的signature  == 新的signature)
```

java

- 当令牌验证为没有被篡改后，服务器可以进行其他验证：比如令牌是否过期等等，这些就视情况而定了，注意：这些验证都需要在服务器中开发人员自己编写进行验证，没有哪个服务器会给你进行自动验证，当然，你可以借助第三方库来完成这些操作。

（7）JWT令牌的总结

- jwt本质上是一种令牌格式。它和终端设备无关，同样和服务器无关，甚至与如何传输无关，它只是规范了令牌的格式而已。

- jwt由三部分组成：header、payload、signature。主体信息在payload
- jwt难以被篡改和伪造。这是因为有第三部分的签名存在。

(8) JWT认证流程

(8-1) 登录

- 用户登录时，将用户名和密码通过 POST 提交到服务端
- 服务端接收到登录数据之后，与数据库中的用户信息进行校验
- 校验通过
- 服务端通过用户id，jwtid，secret 等相关数据 (组装三部分) 生成 JWT令牌
- 将 JWT 字符串和其他信息一起返回给浏览器，浏览器拿到 jwt 字符串后，缓存到本地，等待下一次请求

(8-2) 下一次请求

- 用户登录之后的每次请求，都会将 token (JWT令牌) 携带在请求头的 Authorization /Set-Cookie 中
- 过滤器中对 token 验证成功之后，我们可以拿到保存在 token 中的用户数据，放到上下文中，然后放行，才进入业务代码处理，否则就抛出认证错误异常。

(9) 实际案例

(9-1) 引入依赖

复制代码

```java
implementation("io.jsonwebtoken:jjwt-api:0.11.5")
implementation("io.jsonwebtoken:jjwt-impl:0.11.5")
implementation("io.jsonwebtoken:jjwt-jackson:0.11.5")
```

java

（9-2）源码解释

- 工具类Jwts的进行编码和解码

复制代码

```java
public final class Jwts {

    private static final Class[] MAP_ARG = new Class[]
{Map.class};

    private Jwts() {
    }

    //拿到构建jwt串的构建器
    public static JwtBuilder builder() {
        return
Classes.newInstance("io.jsonwebtoken.impl.DefaultJwtBuilder");
    }

    //拿到解析jwt串的解析器
    public static JwtParserBuilder parserBuilder() {
        return
Classes.newInstance("io.jsonwebtoken.impl.DefaultJwtParserBuilder
");
    }

}
```

java

复制代码

```
//构建器构建一个jwt串的相关方法
```

```java
public interface JwtBuilder extends ClaimsMutator<JwtBuilder> {
    JwtBuilder setHeader(Header header);
    JwtBuilder setHeader(Map<String, Object> header);
    JwtBuilder claim(String name, Object value);
    //内部使用的是claim方法
    JwtBuilder addClaims(Map<String, Object> claims);
    //内部使用的是claim方法，只不过键是sub，值是参数sub的值
    JwtBuilder setSubject(String sub);
    //内部使用的是claim方法，只不过键是aud，值是参数aud的值
    Claims setAudience(String aud);
    //内部使用的是claim方法，只不过键是exp，值是参数exp的值
    Claims setExpiration(Date exp);
    //内部使用的是claim方法，只不过键是nbf，值是参数nbf的值
    Claims setNotBefore(Date nbf);
    //内部使用的是claim方法，只不过键是iat，值是参数iat的值
    Claims setIssuedAt(Date iat);
    Claims setId(String jti);//jwtid的值
    //指定加密时的秘钥（固定的secret密钥来生成key，通过key拿到加密方式，根据加密方式和密钥对前两部分加密后的结果来计算签名）
    JwtBuilder signWith(Key key);
    JwtBuilder signWith(SignatureAlgorithm alg, byte[] secretKey)
    JwtBuilder signWith(SignatureAlgorithm alg, String base64EncodedSecretKey)
    JwtBuilder signWith(SignatureAlgorithm alg, Key key)
    JwtBuilder signWith(Key key, SignatureAlgorithm alg)
    String compact();
}
```

java

复制代码

```
//解析器解析一个jwt串的相关方法
public interface JwtParserBuilder {

    JwtParserBuilder require(String claimName, Object value);

    JwtParserBuilder requireId(String id);

    JwtParserBuilder requireSubject(String subject);

    JwtParserBuilder requireAudience(String audience);


    JwtParserBuilder requireIssuer(String issuer);


    JwtParserBuilder requireIssuedAt(Date issuedAt);


    JwtParserBuilder requireExpiration(Date expiration);


    JwtParserBuilder requireNotBefore(Date notBefore);

    JwtParserBuilder setAllowedClockSkewSeconds(long seconds)
     //指定解密时的秘钥
    JwtParserBuilder setSigningKey(Key key);

    JwtParserBuilder setSigningKey(byte[] key);

    JwtParserBuilder setSigningKey(String base64EncodedSecretKey)

```

```
    JwtParserBuilder deserializeJsonWith(Deserializer<Map<String,
?>> deserializer);

    JwtParser build();
}
```

java

（9-3）实际使用

复制代码

```
/**
 * Jwt主体信息
 */
data class JwtSubject(
    @JsonProperty("jwtId")
    val jwtId: BigInteger,
    @JsonProperty("userId")
    val userId: BigInteger
)
```

java

复制代码

```
/**
 * JWT_TOKEN的转换实现
 */
class JwtAuthenticationConverter : JwtConvert {

    const val DATA_KEY: String = "data" //代表主体数据
    const val TYPE_KEY: String = "REQUEST_TYPE" //代表是客户端请求的
服务端，不是服务端之间的请求。
```

```java
    /**
     * JwtSubject转jwtStr（创建一个jwt串）
     */
    fun jwtSubjectToJwtStr(jwtSubject: JwtSubject): String {
      return Jwts.builder()
        .setId(jwtSubject.jwtId.toString())
        .claim(DATA_KEY, jwtSubject)
        .setHeader(mapOf(TYPE_KEY to true))
        .compact()
      }
}
```

java

复制代码

```java
/**
 * JWT_TOKEN的转换实现
 */
class JwtAuthenticationConverter(
    private val appSecurityProperties: AppSecurityProperties
) : JwtConvert {

    private val deserializer = JacksonDeserializer<Map<String,
*>>(
        mapOf(
            DATA_KEY to JwtSubject::class.java,
        )
    )

    /**
     *jwtStr转JwtSubject（解析一个jwt串）
```

```
     */
    fun jwtStrToJwtSubject(jwtStr: String): JwtSubject {
        val claimsJwt = Jwts.parserBuilder()
            .setAllowedClockSkewSeconds(
appSecurityProperties.jwt.allowedClockSkewSeconds.seconds
            )
            .deserializeJsonWith(deserializer)
            .build()
            .parseClaimsJwt(jwtStr).body
        return claimsJwt.get(DATA_KEY, JwtSubject::class.java)
      }
}
```

java

复制代码

```
package com.bjknrt.dtx.auth.security.common

import
org.springframework.boot.context.properties.ConfigurationProperti
es
import
org.springframework.cloud.context.config.annotation.RefreshScope
import org.springframework.stereotype.Component
import java.time.Duration

@RefreshScope
@Component
@ConfigurationProperties("app.security")
```

```kotlin
data class AppSecurityProperties(
    var jwt: Jwt = Jwt(),
    var rpcJwt: Jwt = Jwt(),
    var cookie: Cookie = Cookie(),
    /** 不会被过滤器拦截白名单 */
    var whiteList: MutableSet<String> = mutableSetOf("/auth/addIdentity")
) {

    data class Jwt(
        /**
         * 字符长度必须大于等于32;  32:HS256 48:HS384 64:HS512
         */
        var secret: String = "bjknrt-default-secret-must-update",//可以配置到yml文件中读取
        /**
         * jwt 有效时常 默认 3600s
         */
        var maxAge: Duration = Duration.ofSeconds(3600),
        /**
         * 允许的始终偏差（服务器之间有时钟偏差）
         */
        var allowedClockSkewSeconds: Duration = Duration.ofMinutes(5)
    )

    data class Cookie(
        var secure: Boolean = false,
        var isHttpOnly: Boolean = true,
        var maxAge: Duration = Duration.ofSeconds(3600),
        var domain: String? = null
    )
```

```
40 }
```

java