- JWT介绍
  - JWT能做什么
  - JWT的优势
    - 基于传统的Session认证
    - 基于JWT认证
- JWT结构
  - JWT的组成
  - Header
  - Payload
  - Sinature
  - 小结
- JWT使用
  - 引入依赖
  - 生成token
  - 根据令牌解析数据
  - 常见异常
  - 封装工具类
  - 整合SpringBoot

JWT介绍

JWT简称JSON Web Token,也就是通过JSON形式作为Web应用中的令牌,用于在各方之间安全地将信息作为JSON对象传输。在数据传输过程中还可以完成数据加密、签名等相关处理。

JWT能做什么

1.授权
这是使用JWT的最常见方案。一旦用户登录，每个后续请求将包括JWT，从而允许用户访问该令牌允许的路由，服务和资源。单点登录是当今广泛使用JWT的一项功能，因为它的开销很小并且可以在不同的域中轻松使用。

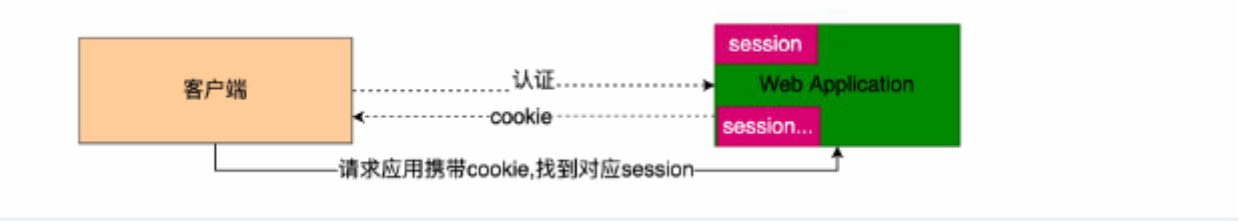
2.信息交换
JSON Web Token是在各方之间安全地传输信息的好方法。因为可以对JWT进行签名（例如，使用公钥/私钥对），所以您可以确保发件人是他们所说的人。此外，由于签名是使用标头和有效负载计算的，因此还可以验证内容是否遭到篡改。

JWT的优势

基于传统的Session认证

http协议本身是一种无状态的协议，而这就意味着如果用户向应用提供了用户名和密码来进行用户认证，那么下一次请求时，用户还要再一次进行用户认证才行。因为根据http协议，服务器并不能知道是哪个用户发出的请求，所以为使得应用能识别是哪个用户发出的请求，需要在服务器存储一份用户登录的信息。登录信息会在响应时传递给浏览器，告诉其保存为cookie,以便下次请求时发送给我们的应用，这样应用就能识别请求来自哪个用户了,这就是传统的基于session认证。

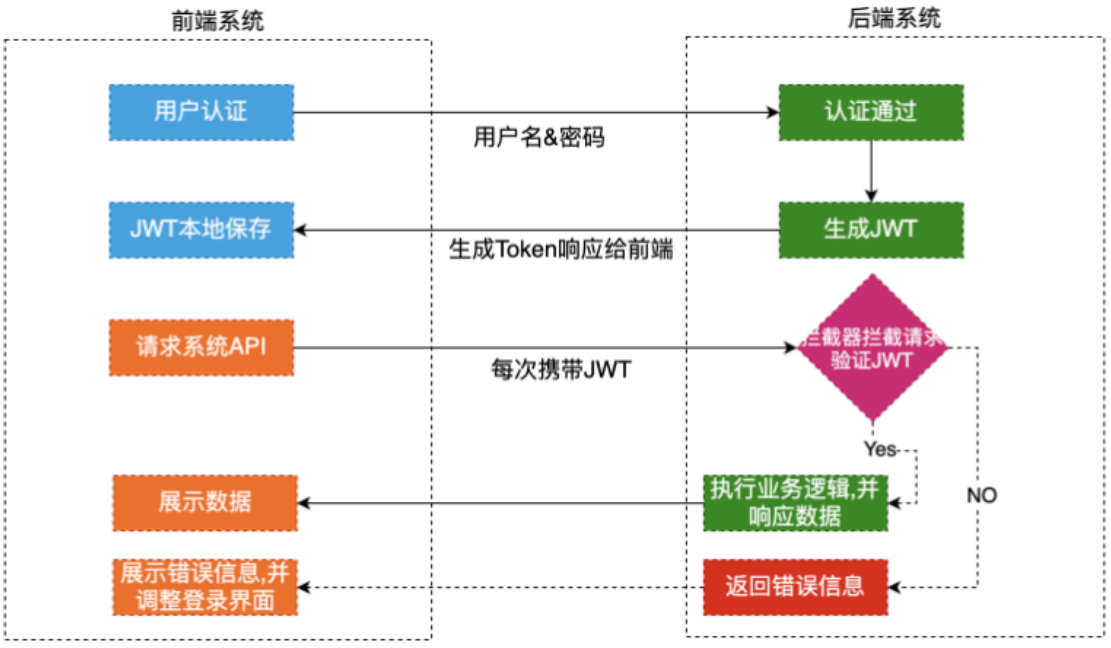
认证流程



暴露的问题

1. 每个用户经过应用认证之后，应用都要在服务端做一次记录，以方便用户下次请求的鉴别，通常而言session都是保存在内存中，而**随着认证用户的增多，服务端的开销会明显增大**
2. 用户认证之后，服务端做认证记录，如果认证的记录被保存在内存中的话，这意味着用户下次请求还必须要请求在这台服务器上,这样才能拿到授权的资源。如果是在分布式的应用上，**限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。**
3. 基于cookie来进行用户识别, cookie如果被截获，用户就会**很容易受到跨站请求伪造的攻击。**
4. 在前后端分离系统中：通常用户一次请求就要转发多次。如果用session 每次携带sessionid 到服务 器，服务器还要查询用户信息。同时如果用户很多。这些信息存储在服务器内存中，给服务器增加负担。还有CSRF（跨站伪造请求攻 击）攻击，session是基于cookie进行用户识别的, cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击。还有就是 sessionid就是一个特征值，表达的信息不够丰富。不容易扩展。而且如果你后端应用是多节点部署。那么就需要实现session共享机制。 不方便集群应用。

基于JWT认证



认证流程

1. 前端通过web将自己的用户名和密码发送到后端，一般是HttpPost请求。
2. 后端核对用户名和密码后，将用户ID及其他信息作为JWT payload，生成一个token返回给前端。
3. 前端保存token，可以将结果保存在localStorage或sessionStorage中，退出登录时删除token即可。
4. 前端在此后的每次请求中的header中放入token
5. 后端检查token的有效性，包括签名是否正确，token是否过期等等
6. 后端验证token通过后，进行业务处理，返回相应结果。

JWT优势

1. 简洁：可以通过URL、Post参数或者Header中发送，**数据量小，传输速度快。**
2. 自包含：负载中**包含了用户的部分不敏感信息**，避免多次查询数据库。
3. token是以加密形式放在客户端的，所以**JWT是跨语言**的，原则上任何WEB形式都支持。
4. 不需要在服务端保存会话信息，特别**适用于分布式微服务。**验证token是否有效时，会根据签名利用算法生成一份新的token，将新的token和传过来的token进行比对，比对结果决定token的验证是否通过。

JWT结构

JWT的组成

JWT有三部分组成：

1. 标头Header
2. 负载Payload
3. Signature

三者之间由.隔开，形式如：

> xxxxx.yyyyy.zzzzz header.payload.signature

Header

- 标头通常由两部分组成：令牌的类型（即JWT）和所使用的签名算法，例如HMAC SHA256或RSA。它会使用 Base64 编码组成 JWT 结构的第一部分。
- 注意:Base64是一种编码，也就是说，它是可以被翻译回原来的样子来的。它并不是一种加密过程。

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

Payload

- 令牌的第二部分是有效负载，其中包含声明。声明是有关实体（通常是用户）和其他数据的声明。同样的，它会使用 Base64 编码组成 JWT 结构的第二部分

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Sinature

- 前面两部分都是使用 Base64 进行编码的，即前端可以解开知道里面的信息。Signature 需要使用编码后的 header 和 payload 以及我们提供的一个密钥，然后使用 header 中指定的签名算法（HS256）进行签名。签名的作用是保证 JWT 没有被篡改过
- 如: HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),secret);

密钥很重要，不要泄露！！！不要太简单！！

小结

JWT由头、负载、签名三部分组成。头信息一般是固定的，有令牌类型和加密类型；负载中有用户的部分不敏感信息；签名是头+负载+密钥组成。三个部分都会进行base64编码，由点隔开，最后生成JWT字符串。

> eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOlsicGhvbmUiLCIxNDMyMzIzNDEzNCJdLCJleHAiOjE1OTU3Mzk0NDIsInVzZXJuYW1lIjoi5byg5LiJIn0.aHmE3RNqvAjFr_dvyn_sD2VJ46P7EGiS5OBMO_TI5jg

> 由于base64编码是可逆的，所以负载中一定不能存放用户的敏感信息，存放一些用户ID等安全数据，可以减少后端查询数据库次数。

JWT使用

引入依赖

```xml
<!--引入jwt-->
<dependency>
  <groupId>com.auth0</groupId>
  <artifactId>java-jwt</artifactId>
  <version>3.4.0</version>
</dependency>
```

生成token

```java
Calendar instance = Calendar.getInstance();
instance.add(Calendar.SECOND, 90);
//生成令牌
String token = JWT.create()
  .withClaim("username", "张三")//设置自定义用户名
  .withExpiresAt(instance.getTime())//设置过期时间
  .sign(Algorithm.HMAC256("token!Q2W#E$RW"));//设置签名 保密 复杂
//输出令牌
System.out.println(token);
```

根据令牌解析数据

```java
// 后端根据同样的密钥+加密算法+base64生成的新的jwtVerifier
JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256("token!Q2W#E$RW")).build();
// 校验jwtVerifier和token
DecodedJWT decodedJWT = jwtVerifier.verify(token);
// 获取payload中存入的用户信息
System.out.println("用户名: " + decodedJWT.getClaim("username").asString());
System.out.println("过期时间: "+decodedJWT.getExpiresAt());
```

常见异常

验证jwt中常见的异常信息:

- SignatureVerificationException: 签名不一致异常
- TokenExpiredException: 令牌过期异常
- AlgorithmMismatchException: 算法不匹配异常
- InvalidClaimException: 失效的payload异常

封装工具类

这样后端在接收到请求的时候每次可以进行token验证，每一个请求都要做同样的验证。造成大量的代码冗余，将其封装为工具类:

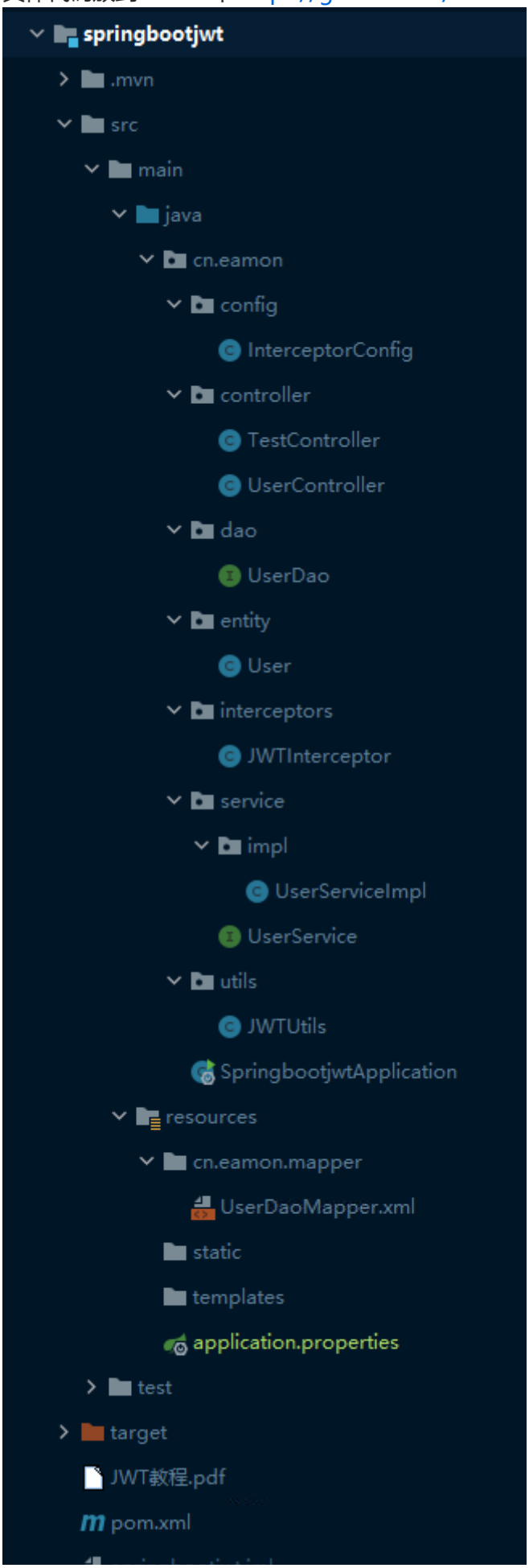
```
public class JWTUtils {

    private static final String SIGNATURE = "!@#$SGW^HDY*%G";

    /**
     * 生成token  header.payload.sing
     */
    public static String getToken(Map<String,String> map){
        //默认7天过期
        Calendar instance = Calendar.getInstance();
        instance.add(Calendar.DATE,7);
        //创建jwt builder
        JWTCreator.Builder builder = JWT.create();
        //设置payload
        map.forEach(builder::withClaim);
        //指定令牌过期时间
        return builder.withExpiresAt(instance.getTime())
                .sign(Algorithm.HMAC256(SIGNATURE));
    }

    /**
     * 验证token 合法性
     */
    public static DecodedJWT verify(String token){
        return JWT.require(Algorithm.HMAC256(SIGNATURE)).build().verify(token);
    }

}
```

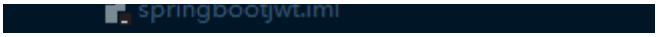
整合SpringBoot

了解JWT的认证流程和用法后就可以配置到项目中，搭建一个SpringBoot+Mybatis+JWT环境：

1. 引入依赖
2. 配置数据库，并添加测试数据
3. 开发entity
4. 开发dao接口和mapper.xml
5. 开发service接口和实现类
6. 开发controller
7. 使用PostMan测试

具体代码放到Github中:https://github.com/EamonHu/FrameWork/tree/master/springbootjwt





JWT的验证可以放到拦截器中，这样可以专心于完成业务需求，自定义拦截器：

```
public class JWTInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        Map<String, Object> map = new HashMap<>();
        //获取请求头中令牌
        String token = request.getHeader("token");
        try {
            //验证令牌
            JWTUtils.verify(token);
            return true;
        } catch (SignatureVerificationException e) {
            e.printStackTrace();
            map.put("msg","无效签名!");
        }catch (TokenExpiredException e){
            e.printStackTrace();
            map.put("msg","token过期!");
        }catch (AlgorithmMismatchException e){
            e.printStackTrace();
            map.put("msg","token算法不一致!");
        }catch (Exception e){
            e.printStackTrace();
            map.put("msg","token无效!!");
        }
        //未通过验证：设置状态
        map.put("state",false);
        //将map转为json  jackson
        String json = new ObjectMapper().writeValueAsString(map);
        response.setContentType("application/json;charset=UTF-8");
        response.getWriter().println(json);
        return false;
    }
}
```

添加一个拦截器：

```
@Configuration
public class InterceptorConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new JWTInterceptor())
                .addPathPatterns("/*")
                .excludePathPatterns("/user/*");
    }
}
```

这样除了用户类访问路径，其他的路径都需要进行token验证。