JWT

1.什么是JWT

```
JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA
---[摘自官网]


# 1.翻译
  官网地址: https://jwt.io/introduction/
  翻译: jsonwebtoken（JWT）是一个开放标准（rfc7519），它定义了一种紧凑的、自包含的方式，用于在各方之间以JSON对象安全地传输信息。此信息可以验证和信任，因为它是数字签名的。jwt可以使用秘密（使用HMAC算法）或使用RSA或ECDSA的公钥/私钥对进行签名

# 2.通俗解释
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。

```

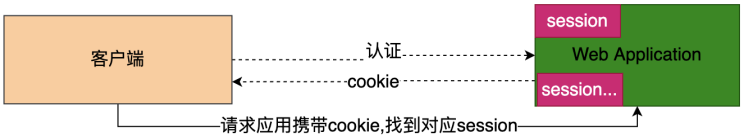
2.JWT能做什么

```
# 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，因此您还可以验证内容是否遭到篡改。


注意：jwt跟session不一样，jwt存储在客户端，session存储在服务器端，服务器断电后session就没了，而jwt因为存储在客户端，所以就不会被影响，只要jwt不过期，就可以继续使用。

```

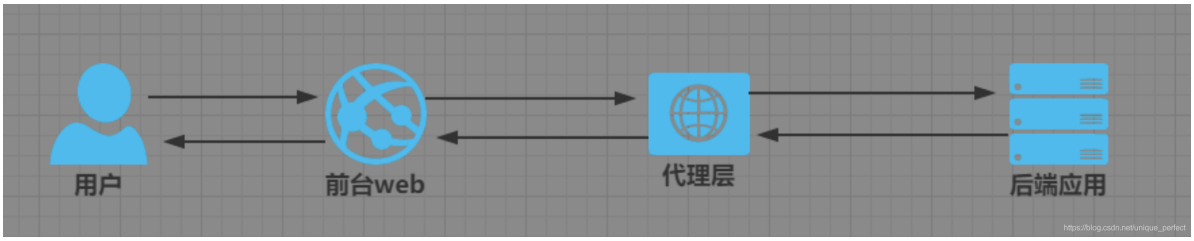
3.为什么是JWT

3.1 基于传统的Session认证

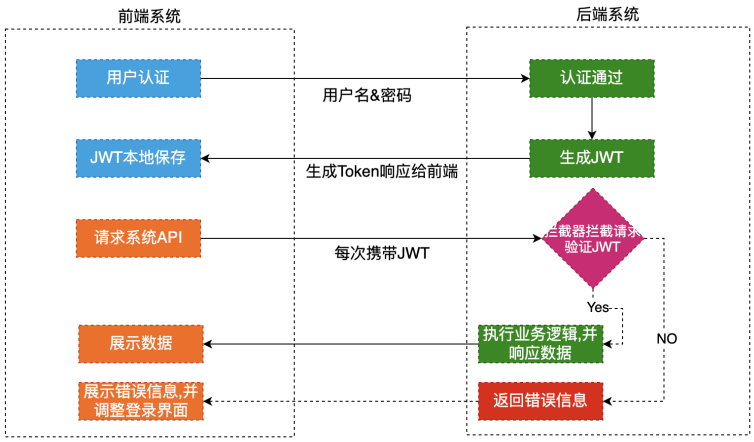
```
# 1.认证方式
我们知道，http协议本身是一种无状态的协议，而这就意味着如果用户向我们的应用提供了用户名和密码来进行用户认证，那么下一次请求时，用户还要再一次进行用户认证才行，因为根据http协议，我们并不能知道是哪个用户发出的请求，所以为了让我们的应用能识别是哪个用户发出的请求，我们只能在服务器存储一份用户登录的信息，这份登录信息会在响应时传递给浏览器，告诉其保存为cookie,以便下次请求时发送给我们的应用，这样我们的应用就能识别请求来自哪个用户了,这就是传统的基于session认证。
t
# 2.认证流程

```




```
# 3.暴露问题
1.每个用户经过我们的应用认证之后，我们的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，这样在分布式的应用上，相应的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.因为是基于cookie来进行用户识别的， cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击。

4.在前后端分离系统中就更加痛苦:如下图所示
也就是说前后端分离在应用解耦后增加了部署的复杂性。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。还有就是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。

```



3.2 基于JWT认证



3.2.1.认证流程

首先，前端通过Web表单将自己的用户名和密码发送到后端的接口。这一过程一般是一个HTTP POST请求。建议的方式是通过SSL加密的传输（https协议），从而避免敏感信息被嗅探。

后端核对用户名和密码成功后，将用户的id等其他信息作为JWT Payload（负载），将其与头部分别进行Base64编码拼接后签名，形成一个JWT(Token)。形成的JWT就是一个形同lll.zzz.xxx的字符串。 token head.payload.singurater

后端将JWT字符串作为登录成功的返回结果返回给前端。前端可以将返回的结果保存在localStorage或sessionStorage上，退出登录时前端删除保存的JWT即可。

前端在每次请求时将JWT放入HTTP Header中的Authorization位。(解决XSS和XSRF问题) HEADER

后端检查是否存在，如存在验证JWT的有效性。例如，检查签名是否正确；检查Token是否过期；检查Token的接收方是否是自己（可选）。

验证通过后后端使用JWT中包含的用户信息进行其他逻辑操作，返回相应结果。

3.2.2.jwt优势

简洁(Compact): 可以通过URL，POST参数或者在HTTP header发送，因为数据量小，传输速度也很快

自包含(Self-contained)：负载中包含了所有用户所需要的信息，避免了多次查询数据库

因为Token是以JSON加密的形式保存在客户端的，所以JWT是跨语言的，原则上任何web形式都支持。

不需要在服务端保存会话信息，特别适用于分布式微服务。
    我的总结：1.节省服务端空间 2.防止跨站请求攻击3.避免第三方缓存组件来实现session共享

4.JWT的结构是什么?

```
token   string  ====>  header.payload.singnature   token

# 1.令牌组成
- 1.标头(Header)
- 2.有效载荷(Payload)
- 3.签名(Signature)
- 因此，JWT通常如下所示:xxxxx.yyyyy.zzzzz   Header.Payload.Signature
```

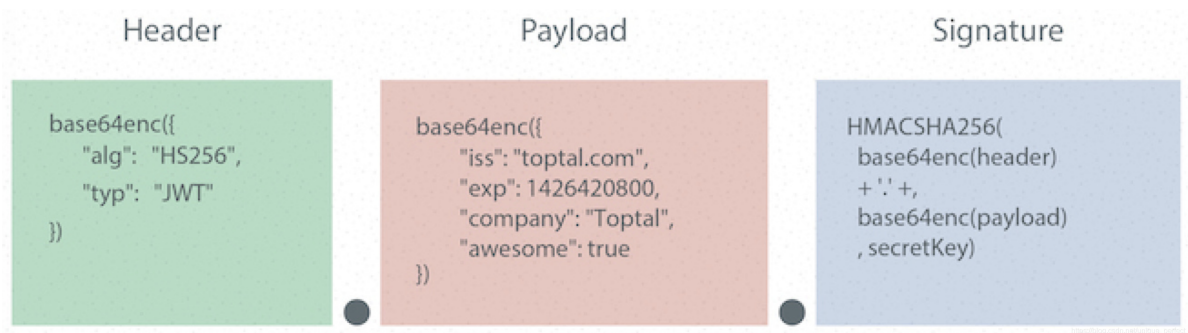
```
# 2.Header
- 标头通常由两部分组成：令牌的类型（即JWT）和所使用的签名算法，例如HMAC SHA256或RSA。它会使用 Base64 编码组成 JWT 结构的第一部分。

- 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" }
```

```
# 3.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
}
```

```
# 4.Signature
- 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);

# 签名目的
- 最后一步签名的过程，实际上是对头部以及负载内容进行签名，防止内容被窜改。如果有人对头部以及负载的内容解码之后进行修改，再进行编码，最后加上之前的签名组合形成新的JWT的话，那么服务器端会判断出新的头部和负载形成的签名和JWT附带上的签名是不一样的。如果要对新的头部和负载进行签名，在不知道服务器加密时用的密钥的话，得出来的签名也是不一样的。

# 信息安全问题
- 在这里大家一定会问一个问题：Base64是一种编码，是可逆的，那么我的信息不就被暴露了吗？

- 是的。所以，在JWT中，不应该在负载里面加入任何敏感的数据。在上面的例子中，我们传输的是用户的User ID。这个值实际上不是什么敏　　感内容，一般情况下被知道也是安全的。但是像密码这样的内容就不能被放在JWT中了。如果将用户的密码放在了JWT中，那么怀有恶意的第　　三方通过Base64解码就能很快地知道你的密码了。因此JWT适合用于向Web应用传递一些非敏感信息。JWT还经常用于设计用户认证和授权系　　统，甚至实现Web应用的单点登录。
```



```
# 5.放在一起
- 输出是三个由点分隔的Base64-URL字符串，可以在HTML和HTTP环境中轻松传递这些字符串，与基于XML的标准（例如SAML）相比，它更紧凑。
- 简洁(Compact)
    可以通过URL， POST 参数或者在 HTTP header 发送，因为数据量小，传输速度快
- 自包含(Self-contained)
    负载中包含了所有用户所需要的信息，避免了多次查询数据库

```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

5.使用JWT

1.引入依赖

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

2.生成token

```
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);
```

```
生成结果
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOlsicGhvbmUiLCIxNDMyMzIzNDEzNCJd
LCJleHAiOjE1OTU3MzkONDIsInVzZXJuYW1lIjoi5byg5LiJIn0.aHmE3RNqvAjFr_dvyn_sD2VJ4
6P7EGiS5OBMO_TI5jg
```

3.根据令牌和签名解析数据

```
JWTVerifier jwtVerifier =
JWT.require(Algorithm.HMAC256("token!Q2W#E$RW")).build();
DecodedJWT decodedJWT = jwtVerifier.verify(token);
System.out.println("用户名: " + decodedJWT.getClaim("username").asString());
 、// 存的是时候是什么类型，取得时候就是什么类型，否则取不到值。
System.out.println("过期时间: "+decodedJWT.getExpiresAt());
```

4.常见异常信息

```
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
```

6.封装工具类

```java
public class JWTUtils {
    private static String TOKEN = "token!Q@w3e4r";
    /**
     * 生成token
     * @param map  //传入payload
     * @return 返回token
     */
    public static String getToken(Map<String,String> map){
        JWTCreator.Builder builder = JWT.create();
        map.forEach((k,v)->{
            builder.withClaim(k,v);
        });
        Calendar instance = Calendar.getInstance();
        instance.add(Calendar.SECOND,7);
        builder.withExpiresAt(instance.getTime());
        return builder.sign(Algorithm.HMAC256(TOKEN));
    }
    /**
     * 验证token
     * @param token
     * @return
     */
    public static void verify(String token){
        JWT.require(Algorithm.HMAC256(TOKEN)).build().verify(token);  // 如果验证通过，则不会把报错，否则会报错
    }
    /**
     * 获取token中payload
     * @param token
     * @return
     */
    public static DecodedJWT getToken(String token){
        return JWT.require(Algorithm.HMAC256(TOKEN)).build().verify(token);
    }
}
```

7.整合springboot

```
# 0.搭建springboot+mybatis+jwt环境
- 引入依赖
- 编写配置
```

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

<!--引入mybatis-->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.1.3</version>
</dependency>

<!--引入lombok-->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.12</version>
</dependency>

<!--引入druid-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.19</version>
</dependency>

<!--引入mysql-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.38</version>
</dependency>
```

```properties
server.port=8989
spring.application.name=jwt

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/jwt?characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root

mybatis.type-aliases-package=com.baizhi.entity
mybatis.mapper-locations=classpath:com/baizhi/mapper/*.xml

logging.level.com.baizhi.dao=debug
```

\1. 开发数据库 - 这里采用最简单的表结构验证JWT使用

```sql
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `name` varchar(80) DEFAULT NULL COMMENT '用户名',
  `password` varchar(40) DEFAULT NULL COMMENT '用户密码',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

```

\2. 开发entity

```java
@Data
@Accessors(chain=true)
public class User {
    private String id;
    private String name;
    private String password;
}
```

```java
@Data
@Accessors(chain=true)
public class User {
    private String id;
    private String name;
    private String password;
}
```

3.开发DAO接口和mapper.xml

```java
@Mapper
public interface UserDAO {
    User login(User user);
}
```

```java
@Mapper
public interface UserDAO {
    //直接根据用户名密码登录
    User login(User user);
}
```

```xml
<mapper namespace="com.baizhi.dao.UserDAO">
    <!--这里就写的简单点了毕竟不是重点-->
    <select id="login" parameterType="User" resultType="User">
        select * from user where name=#{name} and password = #{password}
    </select>
</mapper>

```

```xml
<mapper namespace="com.baizhi.dao.UserDAO">

    <!--这里就写的简单点了毕竟不是重点-->
    <select id="login" parameterType="User" resultType="User">
        select * from user where name=#{name} and password = #{password}
    </select>

</mapper>
```

4.开发Service 接口以及实现类

```
public interface UserService {
    User login(User user);//登录接口
}
```

```
public interface UserService {
    User login(User user);//登录接口
}
```

```
@Service
@Transactional
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDAO userDAO;
    @Override
    @Transactional(propagation = Propagation.SUPPORTS)
    public User login(User user) {
        User userDB = userDAO.login(user);
        if(userDB!=null){
            return userDB;
        }
        throw  new RuntimeException("登录失败~~");
    }
}
```

```
@Service
@Transactional
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDAO userDAO;

    @Override
    @Transactional(propagation = Propagation.SUPPORTS)
    public User login(User user) {
        User userDB = userDAO.login(user);
        if(userDB!=null){
            return userDB;
        }
        throw  new RuntimeException("登录失败~~");
    }
}
```

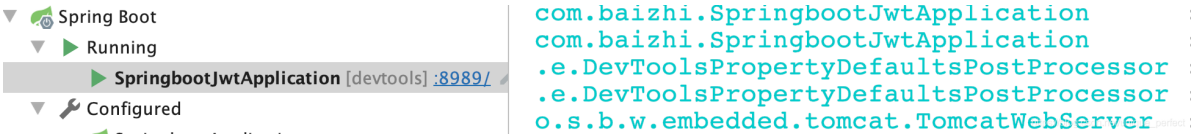
5.开发controller

```
@RestController
@Slf4j
public class UserController {
    @Autowired
    private UserService userService;
    @GetMapping("/user/login")
    public Map<String,Object> login(User user) {
        Map<String,Object> result = new HashMap<>();
        log.info("用户名: [{}]", user.getName());
        log.info("密码: [{}]", user.getPassword());
        try {
            User userDB = userService.login(user);
            Map<String, String> map = new HashMap<>();//用来存放payload
            map.put("id",userDB.getId());
            map.put("username", userDB.getName());
            String token = JWTUtils.getToken(map);
            result.put("state",true);
            result.put("msg","登录成功!!!");
```

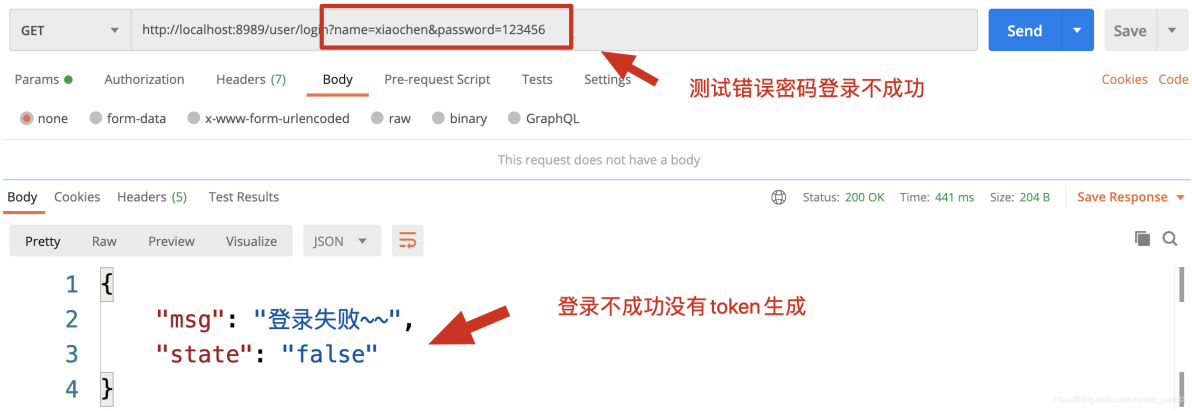
```
            result.put("token",token); //成功返回token信息
        } catch (Exception e) {
            e.printStackTrace();
            result.put("state","false");
            result.put("msg",e.getMessage());
        }
        return result;
    }
}
```

```
@RestController
@Slf4j
public class UserController {

    @Autowired
    private UserService userService;


    @GetMapping("/user/login")
    public Map<String,Object> login(User user) {
        Map<String,Object> result = new HashMap<>();
        log.info("用户名: [{}]", user.getName());
        log.info("密码: [{}]", user.getPassword());
        try {
            User userDB = userService.login(user);
            Map<String, String> map = new HashMap<>();//用来存放payload
            map.put("id",userDB.getId());
            map.put("username", userDB.getName());
            String token = JWTUtils.getToken(map);
            result.put("state",true);
            result.put("msg","登录成功!!!");
            result.put("token",token);
        } catch (Exception e) {
            e.printStackTrace();
            result.put("state","false");
            result.put("msg",e.getMessage());
        }
        return result;
    }
```

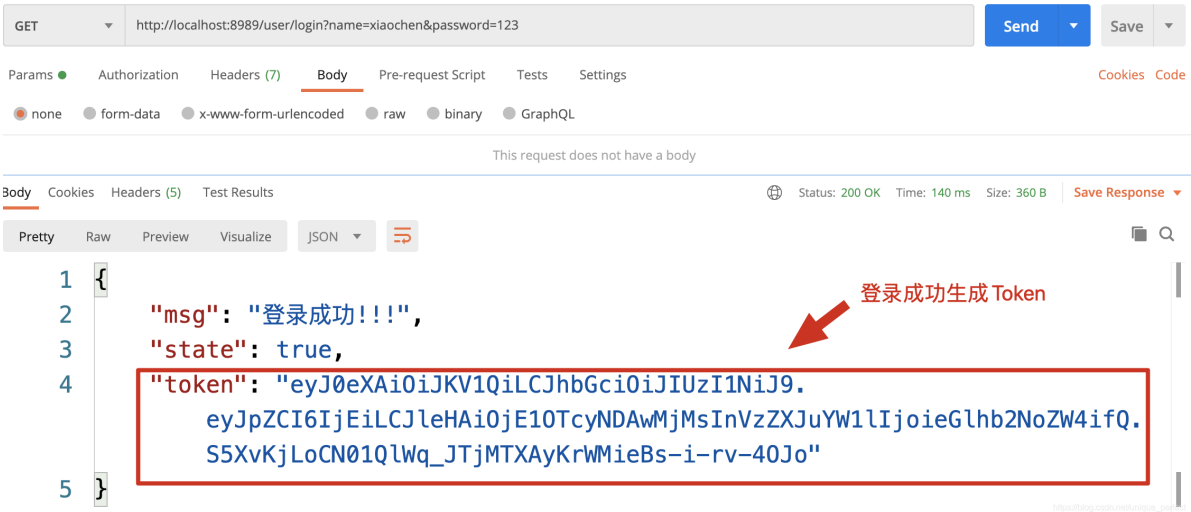
6.数据库添加测试数据启动项目



7.通过postman模拟登录失败



8.通过postman模拟登录成功



9.编写测试接口

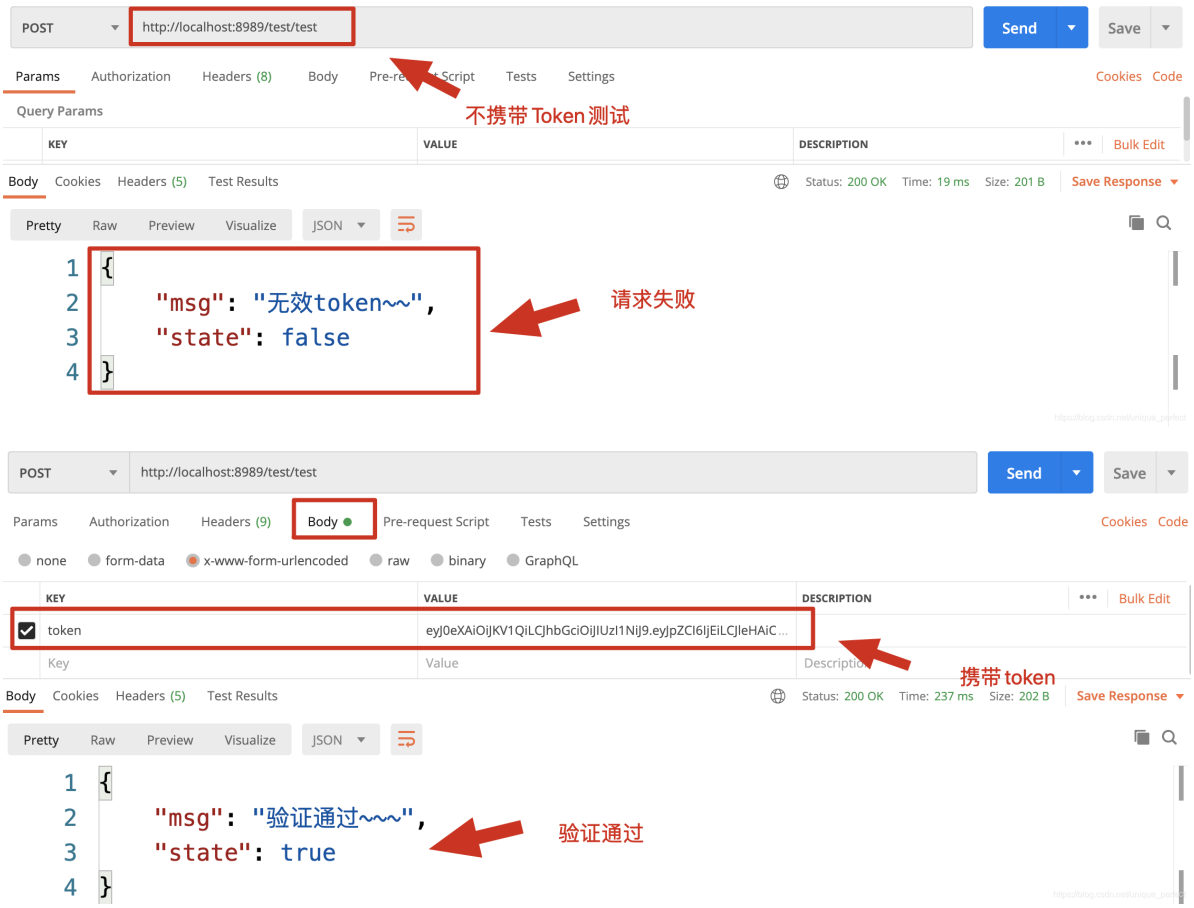
```
@PostMapping("/test/test")
public Map<String, Object> test(String token) {
  Map<String, Object> map = new HashMap<>();
  try {
    JWTUtils.verify(token);
    map.put("msg", "验证通过~~~");
    map.put("state", true);
  } catch (TokenExpiredException e) {
    map.put("state", false);
    map.put("msg", "Token已经过期!!!");
  } catch (SignatureVerificationException e){
    map.put("state", false);
    map.put("msg", "签名错误!!!");
  } catch (AlgorithmMismatchException e){
    map.put("state", false);
    map.put("msg", "加密算法不匹配!!!");
  } catch (Exception e) {
    e.printStackTrace();
    map.put("state", false);
    map.put("msg", "无效token~~");
  }
  return map;
}
```

```
@PostMapping("/test/test")
public Map<String, Object> test(String token) {
    Map<String, Object> map = new HashMap<>();
    try {
        JWTUtils.verify(token);
        map.put("msg", "验证通过~~~");
        map.put("state", true);
    } catch (TokenExpiredException e) {
        map.put("state", false);
        map.put("msg", "Token已经过期!!!");
    } catch (SignatureVerificationException e){
        map.put("state", false);
        map.put("msg", "签名错误!!!");
    } catch (AlgorithmMismatchException e){
        map.put("state", false);
        map.put("msg", "加密算法不匹配!!!");
    } catch (Exception e) {
        e.printStackTrace();
        map.put("state", false);
        map.put("msg", "无效token~~");
    }
    return map;
}
```

所有接口定义接收 token 的参数

验证 token 有效性

10.通过postman请求接口



11.问题? - 使用上述方式每次都要传递token数据,每个方法都需要验证token代码冗余,不够灵活? 如何优化 - 使用拦截器进行优化

```
public class JWTInterceptor implements HandlerInterceptor{

    @Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
  String token = request.getHeader("token");
  Map<String,Object> map = new HashMap<>();
  try {
    JWTUtils.verify(token);
    return true;
  } catch (TokenExpiredException e) {
    map.put("state", false);
    map.put("msg", "Token已经过期!!!");
  } catch (SignatureVerificationException e){
    map.put("state", false);
    map.put("msg", "签名错误!!!");
  } catch (AlgorithmMismatchException e){
    map.put("state", false);
    map.put("msg", "加密算法不匹配!!!");
  } catch (Exception e) {
    e.printStackTrace();
    map.put("state", false);
    map.put("msg", "无效token~~");
  }
  String json = new ObjectMapper().writeValueAsString(map);
  response.setContentType("application/json;charset=UTF-8");
  response.getWriter().println(json);
  return false;
```

```
}

}
```

```
@Configuration
public class InterceptorConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new JwtTokenInterceptor()).
          excludePathPatterns("/user/**")  // 放行
          .addPathPatterns("/**"); // 拦截除了"/user/**的所有请求路径
    }
}

//https://blog.csdn.net/unique_perfect/article/details/109219826
```

8.实际应用

- 对于单体spring-boot应用，通过配置拦截器来简约代码。
- 对于分布式应用，配置在gateway处

9.demo地址

```
git@gitee.com:AugustT2/springboot-jwt-2020.git
```