简单代码实现JWT(json web token)完成SSO单点登录

Java3y 5月5日

本文作者：加耀
投稿来源：
https://zhuanlan.zhihu.com/p/64377462?utm_source=wechat_session&utm_medium=social&utm_oi=775841244587773952

使用JWT完成SSO单点登录

前两个月在公司面试过程中，发现很多求职者在简历中都写有实现过单点登录，并且使用的技术种类繁多，刚好公司项目中实现单点登录的是使用一款叫做JWT(json web token)的框架，其实现原理也挺简单的，遂想，是否自己可以用简单代码实现一个简易版的JWT来完成单点登录认证(SSO)，所谓SSO单点登录，其实是指的一类解决方案，有很多种方式都可以实现，这里描述的JWT就是其中一种；

首先，我们先来JWT官方看一下JWT的简单介绍吧；

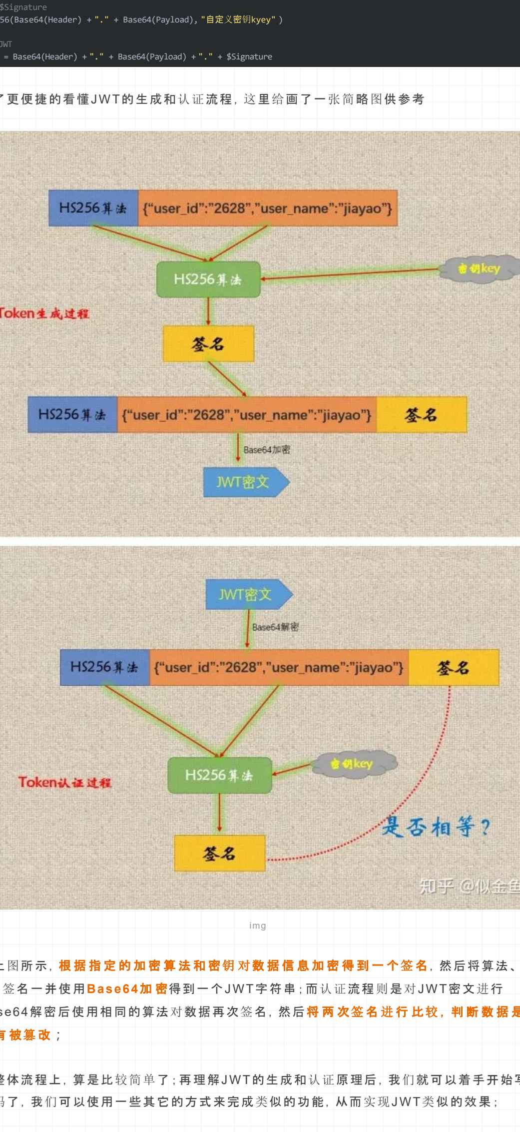
JWT的官网地址是：https://jwt.io/，我们在JWT的官网可以看到一个完整的JWT是由三个部分组成，分别是Header头部、Payload数据部分、Signature签名三部分组成；如图：

img

如果将上图进行简化，JWT数据结构大抵如下

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

// Payload
{
// reserved claims
"sub": "1234567890",
"name": "John Doe",
"iat": "1516239022"
}

// $Signature
HS256(Base64(Header) + "." + Base64(Payload), "自定义密钥kyey" )

// JWT
JWT = Base64(Header) + "." + Base64(Payload) + "." + $Signature
```

为了更便捷的看懂JWT的生成和认证流程，这里给画了一张简略图供参考



img

如上图所示，**根据指定的加密算法和密钥对数据信息加密得到一个签名**，然后将算法、数据、签名一并使用**Base64加密**得到一个JWT字符串；而认证流程则是对JWT密文进行Base64解密后使用相同的算法对数据再次签名，然后**将两次签名进行比较，判断数据是否有被篡改**；

在整体流程上，算是比较简单了；再理解JWT的生成和认证原理后，我们就可以着手开始写代码了，我们可以使用一些其它的方式来完成类似的功能，从而实现JWT类似的效果；

首先，我们创建一个SpringBoot工程(方便调试不用自己写请求映射)，创建好工程后，首先我们需要配置JWT的相关信息，比如：加密方式(当做是Header部分)、数据信息及token有效时间、JWT生成和认证算法等

在这里，我们先定义一个枚举FailureTime，用来定义支持的过期时间策略

```
public enum FailureTime {
  /**
   *秒
   */
   SECOND,
  /**
   *分
   */
   MINUTE,
  /**
   *时
   */
   HOUR,
  /**
   *天
   */
   DAY

}
```

在上面的代码中，我们定义好这个jwt支持的过期时间策略有秒、分、时、天四种四种类型；定义好规则后，我们再来写一个类，用来根据规则生成token相应的过期时间的工具类

```
public class FailureTimeUtils {

  /**
   *@demand: 根据指定的时间规则和时间生成有效时间
   *@parameters:
   *@creationDate:
   *@email: huangjy19940202@gmail.com
   */
  public static Date creatValidTime(FailureTime failureTime, int jwtValidTime) {
      Date date = new Date();
    if (failureTime.name().equals(FailureTime.SECOND)) {
      return createBySecond(date, jwtValidTime);
    }
    if (failureTime.name().equals(FailureTime.MINUTE)) {
      return createBySecond(date, jwtValidTime * 60);
    }
    if (failureTime.name().equals(FailureTime.HOUR)) {
      return createBySecond(date, jwtValidTime * 60 * 60);
    }
    if (failureTime.name().equals(FailureTime.DAY)) {
      return getDateAfter(date, jwtValidTime);
    }
    return null;
  }

  /**
   *得到几天后的时间
   *
   *@param day
   *@return
   */
  public static Date getDateAfter(Date date, int day) {
       Calendar now = Calendar.getInstance();
      now.setTime(date);
       now.set(Calendar.DATE, now.get(Calendar.DATE) + day);
    return now.getTime();
  }

  /**
   *得到几天前的时间
   *
   *@param date
   *@param day
   *@return
   */
  public static Date getDateBefore(Date date, int day) {
       Calendar now = Calendar.getInstance();
      now.setTime(date);
       now.set(Calendar.DATE, now.get(Calendar.DATE) - day);
    return now.getTime();
  }

  /**
   *得到多少秒之后的时间
   *
   *@param date
   *@param jwtValidTime
   *@return
   */
  public static Date createBySecond(Date date, int jwtValidTime) {
       Calendar calendar = Calendar.getInstance();
       calendar.setTime(date);
       calendar.add(Calendar.SECOND, jwtValidTime);
    return calendar.getTime();
  }

}
```

上面的代码中，我们定义了几个方法，分别是计算几天后的当前时间和多少秒后的当前时间；然后我们再来定义一个枚举用来定义所支持的加密算法；

```
public enum Header {

  SM3("sm3","国密3加密算法,其算法不可逆，类似于MD5"),
  SM4("sm4","国密4加密算法，对称加密"),
  AES("aes","AES加密算法，对称加密");

  private String code;

  private String details;

   Header(String code, String details) {
    this.code = code;
    this.details = details;
  }
}
```

在上面代码中，我们定义我们这个JWT支持的加密方式有三种，分别是SM3、SM4、AES，都是属于对称加密算法；SM2是非对称加密算法(此处不做讲解)；

下面再定义记录用户数据的部分，我们创建一个JwtClaims 用来存储我们需要保存到JWT中的个性数据，代码如下

```
import java.util.HashMap;
import java.util.Objects;

public class JwtClaims extends HashMap {
  public JwtClaims() {
    this.put(ID, null);
    this.put(NAME, null);
    this.put(PHONE, null);
    this.put(FAILURETIME, null);
  }
   String ID = "id";
   String NAME = "name";
   String PHONE = "phone";
  /**
   *有效期
   */
   String FAILURETIME = "failureTime";
  public JwtClaims put(String key, Object value) {
    super.put(key, value);
    return this;
  }
  /**
   *重写hashCode方法
   *
   *@return
   */
  @Override
  public int hashCode() {
    return Objects.hash(super.hashCode(), this);
  }
}
```

在上面我们将JWT中需要用到的数据都定义好了后，下面我们就可以开始写JWT相关的算法了，代码如下所示：

```
@Slf4j
public class Jwts extends ConcurrentHashMap {

  private static Jwts jwts;

  static {
      jwts = new Jwts();
  }

  /**
   *默认加密密钥
   */
  private final String jwtSafetySecret = "0dcac1b6ec8843488fbe90e166617e34";

  /**
   *指定加密算法和密钥
   *
   *@param header
   *@param jwtSafetySecret
   *@return
   */
  public static Jwts header(Header header, String jwtSafetySecret) {
      HashMap<String, Object> map = new HashMap<>();
      map.put("code", header);
      map.put("jwtSafetySecret", jwtSafetySecret);
      jwts.put("header", map);
    return jwts;
  }

  /**
   *@param jwtClaims
   *@return
   */
  public Jwts payload(JwtClaims jwtClaims) {
      jwts.put("payload", jwtClaims);
    return jwts;
  }

  /**
   *签名并生成token
   *
   *@return
   */
  public String compact() throws Exception {
    // 头部
       HashMap<String, Object> headerObj = (HashMap<String, Object>) jwts.get("header");
    // 数据
       JwtClaims jwtClaims = (JwtClaims) jwts.get("payload");
       jwtClaims.put("uuid", UUID.randomUUID());
    // 生成签名
       Object jwtSafetySecretObj = headerObj.get("jwtSafetySecret");
    // 从头部信息中去除密钥信息
       headerObj.remove("jwtSafetySecret");
       String jwtSafetySecret = jwtSafetySecretObj == null ? this.jwtSafetySecret : jwtSafetySecretObj.toString();
       Object code = headerObj.get("code");
       String encryptionType = code == null ? "AES" : code.toString();
    // 开始签名
        String signature = dataSignature(headerObj, jwtClaims, encryptionType, jwtSafetySecret);
    // 生成token
        String token = Base64Utils.getBase64(JSONObject.toJSONString(headerObj)) + "."
              + Base64Utils.getBase64(JSONObject.toJSONString(jwtClaims)) + "."
           + signature;
       System.out.println("生成的token为:" + token);
    return token;
  }
  /**
   *生成摘要
   */
  private static String dataSignature(HashMap<String, Object> headerObj, JwtClaims jwtClaims, String encryptionType,
      String dataSignature = null;
    if (encryptionType.equals(Header.AES.name())) {
          dataSignature = AESUtils.encrypt(JSONObject.toJSONString(headerObj) + JSONObject.toJSONString(jwtClaims),
    }else if (encryptionType.equals(Header.SM3.name())) {
          dataSignature = SM3Cipher.sm3Digest(JSONObject.toJSONString(headerObj) + JSONObject.toJSONString(jwtClaims)
    }else if (encryptionType.equals(Header.SM4.name())) {
        dataSignature = new SM4Util().encode(JSONObject.toJSONString(headerObj) + JSONObject.toJSONString(jwtClaims)
    }
    return dataSignature;
  }
  /**
   *@demand: 校验token完整性和时效性
   */
  public static Boolean safetyVerification(String tokenString, String jwtSafetySecret) throws Exception {
    // 有坑，转义字符
      String[] split = tokenString.split("\\.");
    if (split.length != 3) {
      throw new RuntimeException("无效的token");
    }
    // 头部信息
      HashMap<String, Object> obj = JSON.parseObject(Base64Utils.getFromBase64(split[0]), HashMap.class);
    // 数据信息
      JwtClaims jwtClaims = JSON.parseObject(Base64Utils.getFromBase64(split[1]), JwtClaims.class);
    // 签名信息
      String signature = split[2];

    // 验证token是否在有效期内
    if (jwtClaims.get("failureTime") != null) {
        Date failureTime = (Date) jwtClaims.get("failureTime");
      int i = failureTime.compareTo(new Date());
      if (i > 0) {
        throw new RuntimeException("此token已过有效期");
      }
    }

    // 验证数据篡改
      Object code = obj.get("code");
      String encryptionType = code == null ? "AES" : code.toString();
    // 比较签名
      String signatureNew = dataSignature(obj, jwtClaims, encryptionType, jwtSafetySecret);
    return signature.equals(signatureNew.replaceAll("\r\n","")) ? true : false;
  }

}
```

在上述的代码中，我们定义了一个静态变量jwts，此处涉及线程安全，暂时先不调整，后期再做优化；在上述代码中，完成了对Header和payload签名操作，然后生成一个新的token，其原理和下图相似；

img

然后在代码中我们还完成了对Token认证的操作，其方法为：safetyVerification，在方法中，我们通过对token中的三部分进行签名和比对并且完成token时效性判断(当没有配置token时效性是则表示永久有效)；在这个步骤中可以有效防止数据被篡改，从而保证数据安全；

对JWT加密和解密方面的核心代码大抵如此，其它的引入了一些工具类类似国密加密算法、AES算法及Base64加密算法，这些在完整代码中都有，此处就不一一展示；GitLab地址：

- https://gitlab.com/qingsongxi/myjwt

代码结构如下图所示：

img

在这里，我们需要定义一个配置文件application.properties，在配置文件中加入相关参数，比如 对称加密密钥、token有效期、需要拦截的URL等等

```
# 密钥key
jwt.safety.secret=y2W89L6BkRAFljhN
# token有效期
jwt.valid.time=7
# 需要jwt拦截的url
jwt.secret-url=/findCustomerById
# 端口
server.port=80
```

在这里我们需要定义一个拦截器，用来拦截需要token才能访问的URL；

```
/**
 *@author: JiaYao
 *@demand: 自定义web拦截器
 */
@Slf4j
@Component
public class WebInterceptor implements HandlerInterceptor {

  /**
   * JWT密钥
   */
  @Value("${jwt.safety.secret}")
  private String jwtSafetySecret;

  @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object o) throws Exception {
    // 进入拦截器 WebInterceptor...
       String authorization = request.getHeader("Authorization");
    if (authorization == null || !authorization.startsWith("Bearer ")) {
      return noAccess403(response);
    }else {
      try {
            String token = authorization.substring(7).replaceAll(" ", "");
        // 验证token的完整性和有效性
        if (StringUtils.isNotEmpty(token) && Jwts.safetyVerification(token, jwtSafetySecret)) {
              JwtClaims jwtClaims = JSON.parseObject(Base64Utils.getFromBase64(token.split("\\.")[1]), JwtClaims.clas
            request.setAttribute("claims", jwtClaims);
          return true;
        }
      }catch (Exception e) {
          e.printStackTrace();
        return noAccess(response);
      }
    }
    return false;
  }

  /**
   *在未登录状态或登录状态失效时请求需要登录状态才能请求的URL
   *
   *@param httpServletResponse
   *@return
   *@throws Exception
   */
  public boolean noAccess(HttpServletResponse httpServletResponse) throws Exception {
       httpServletResponse.setContentType("text/json; charset=UTF-8");
       httpServletResponse.getWriter().write(JSON.toJSONString(Json.newInstance(Apistatus.CODE_401)));
    return false;
  }

  /**
   *在未登录状态或登录状态失效时请求需要登录状态才能请求的URL
   *
   *@param httpServletResponse
   *@return
   *@throws Exception
   */
  public boolean noAccess403(HttpServletResponse httpServletResponse) throws Exception {
       httpServletResponse.setContentType("text/json; charset=UTF-8");
       httpServletResponse.getWriter().write(JSON.toJSONString(Json.newInstance(Apistatus.CODE_403)));
    return false;
  }

  @Override
  public void postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object
        o, ModelAndView modelAndView) throws Exception {
  }

  @Override
  public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse
         httpServletResponse, Object o, Exception e) throws Exception {
  }
}


/**
 *@author: JiaYao
 *@demand: 将拦截器添加到列表中，即观察者与被观察者
 *@parameters:
 *@creationDate: 2018/12/19 0019 9:16
 */
@Configuration
public class WebRequestInterceptor extends WebMvcConfigurerAdapter {

  @Autowired
  private WebInterceptor webInterceptor;

  /**
   *需要JWT拦截的Url
   */
  @Value("${jwt.secret-url}")
  private String jwtSecretUrl;
  /**
   * JWT密钥
   */
  @Value("${jwt.safety.secret}")
  private String jwtSafetySecret;

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
       jwtSecretUrl = jwtSecretUrl.replaceAll(" ", "");
       registry.addInterceptor(webInterceptor).addPathPatterns(jwtSecretUrl.split(","));
  }

}
```

到这里，我们的JWT小工具基本上就算是已经写完了，只需要整合到具体的业务中就可以开始投入使用，下面编写一个访问控制层，在里面定义两个方法，一个是请求登录获取token，另一个是请求需要登录下才能请求的资源；

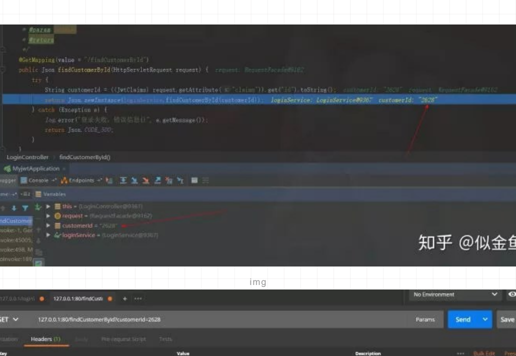
```
/**
 *类名: LoginController
 */
@Slf4j
@RestController
public class LoginController {

  @Autowired
  private LoginService loginService;

  /**
   *登录
   *
   *@param customerId
   *@return
   */
  @GetMapping(value = "/login")
  public Json login(String customerId) {
    try {
      return Json.newInstance(loginService.login(customerId));
    }catch (Exception e) {
        log.error("登录失败，错误信息{}", e.getMessage());
      return Json.CODE_500;
    }
  }

  /**
   *根据用户id查询用户信息
   *
   *@param request
   *@return
   */
  @GetMapping(value = "/findCustomerById")
  public Json findCustomerById(HttpServletRequest request) {
    try {
          String customerId = ((JwtClaims) request.getAttribute("claims")).get("id").toString();
      return Json.newInstance(loginService.findCustomerById(customerId));
    }catch (Exception e) {
        log.error("登录失败，错误信息{}", e.getMessage());
      return Json.CODE_500;
    }
  }

}
```

然后再来编写一个业务层代码

```
@Service
public class LoginService {

  @Value("${jwt.safety.secret}")
  private String jwtSafetySecret;

  @Value("${jwt.valid.time}")
  private int jwtValidTime;

  /**
   *登录
   *
   *@param customerId
   *@return
   */
  public String login(String customerId) {
      Customer customer = new Customer();
       customer.setId(customerId);
       customer.setName("jiayao");
       customer.setPhone("1234567890");
    return createTokenString(customer);
  }

  /**
   *根据id查用户
   *
   *@param customerId
   *@return
   */
  public Customer findCustomerById(String customerId) {
      Customer customer = new Customer();
       customer.setId(customerId);
       customer.setName("jiayao");
       customer.setPhone("1234567890");
    return customer;
  }

  /**
   *生成token
   *
   *@param customer
   *@return
   */
  public String createTokenString(Customer customer) {
      String jwtToken = null;
    try {
         jwtToken = Jwts.header(Header.SM4, jwtSafetySecret)
            .payload(new JwtClaims()
              .put("id", customer.getId())
              .put("name", customer.getName())
              .put("phone", customer.getPhone())
              .put("failureTime", FailureTimeUtils.creatValidTime(FailureTime.DAY, jwtValidTime))
              .put("mytest", "我的个性属性"))
            .compact();
    }catch (Exception e) {
          e.printStackTrace();
    }
    return jwtToken.replaceAll("\r\n","");
  }

}
```

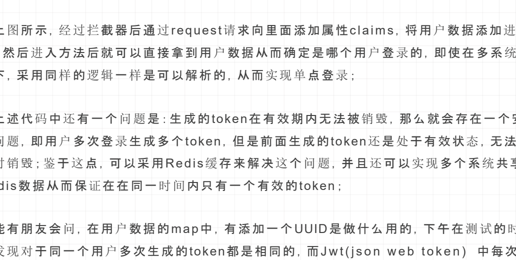
在上述代码中，有一个createTokenString的方法，此方法可进一步抽取为一个静态的工具类，在里面我们指定加密方式和密钥信息、指定token有效策略；

启动项目后我们通过Postman请求登录接口获取token信息，如下：

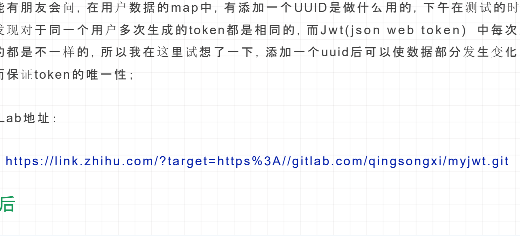


img

如上图所示，通过请求登录接口我们成功获取到了token，我们使用这个token去请求一个需要登录才能请求的资源试试；



img



img

如上图所示，经过拦截器后通过request请求向里面添加属性claims，将用户数据添加进来，然后进入方法后就可以直接拿到用户数据从而确定是哪个用户登录的，即使在多系统情况下，采用同样的逻辑一样是可以解析的，从而实现单点登录；

在上述代码中还有一个问题是：生成的token在有效期内无法被销毁，那么就会存在一个安全问题，即用户多次登录生成多个token，但是前面生成的token还是处于有效状态，无法被及时销毁；鉴于这点，可以采用Redis缓存来解决这个问题，并且还可以实现多个系统共享Redis数据从而保证在在同一时间内只有一个有效的token；

可能有朋友会问，在用户数据的map中，有添加一个UUID是做什么用的，下午在测试的时候我发现对于同一个用户多次生成的token都是相同的，而Jwt(json web token) 中每次生成的都是不一样的，所以我在这里试想了一下，添加一个uuid后可以使数据部分发生变化，从而保证token的唯一性；

GitLab地址：

- https://link.zhihu.com/?target=https%3A//gitlab.com/qingsongxi/myjwt.git

最后

阅读原文