Json Web Token

使用手册

V2.0

**目 录**

[1 简介 1](#_Toc488785706)

[1.1 传统的session认证 1](#_Toc488785707)

[1.2 基于Token的鉴权机制 2](#_Toc488785708)

[1.3 和Session方式存储id的差异 2](#_Toc488785709)

[2 JWT的构成 2](#_Toc488785710)

[2.1 header 3](#_Toc488785711)

[2.2 playload 3](#_Toc488785712)

[2.3 signature 4](#_Toc488785713)

[3 释疑 5](#_Toc488785714)

[3.1 签名的目的 5](#_Toc488785715)

[3.2 信息会暴露？ 5](#_Toc488785716)

[3.3 Token过期后怎么办？ 6](#_Toc488785717)

[4 应用 6](#_Toc488785718)

[4.1 工作时序 6](#_Toc488785719)

[4.2 信任链接 7](#_Toc488785720)

[4.3 单点登录 7](#_Toc488785721)

[5 开源工具 8](#_Toc488785722)

[5.1 在maven中添加jar包依赖 8](#_Toc488785723)

[5.2 JWT加密和解密的工具类 8](#_Toc488785724)

[5.3 Token的发送与获取 11](#_Toc488785725)

[5.4 Token验证机制 11](#_Toc488785726)

[5.5 Token过滤器 11](#_Toc488785727)

[5.6 资源注册管理器（Jersey） 13](#_Toc488785728)

[6 总结 14](#_Toc488785729)

[6.1 优点 14](#_Toc488785730)

[6.2 安全相关 14](#_Toc488785731)

# 简介

Json web token (JWT),官方说明：是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准（[(RFC 7519](https://tools.ietf.org/html/rfc7519))。该token被设计为紧凑且安全的，特别适用于分布式站点的单点登录（SSO）场景。JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

为了支撑大量用户、海量数据的应用，现在的系统复杂度越来越高，很多系统采用了SOA架构或微服务架构，将系统拆分为多个子系统，分别部署在不同的应用服务中。用户在使用系统的过程中，经常会在服务之间跳转或请求数据，这就要求使用一个可以满足多服务间认证的体系，并且要对系统处理压力小、速度快。这就是Token机制的优势。

## 传统的session认证

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。

但是这种基于session的认证使应用本身很难得到扩展，随着不同客户端用户的增加，独立的服务器已无法承载更多的用户，而这时候基于session认证应用的问题就会暴露出来.

**基于session认证所显露的问题**

* **Session**: 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。
* **扩展性**: 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。
* **CSRF**: 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。

## 基于Token的鉴权机制

基于token的鉴权机制类似于http协议，也是无状态的，它不需要在服务端去保留用户的认证信息或者会话信息。这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

流程上是这样的：

* 用户使用用户名密码来请求服务器
* 服务器进行验证用户的信息
* 服务器通过验证发送给用户一个token
* 客户端存储token，并在每次请求时附送上这个token值
* 服务端验证token值，并返回数据

这个token必须要在每次请求时传递给服务端，它应该保存在请求头里， 另外，服务端要支持CORS(跨来源资源共享)策略，一般我们在服务端这么做就可以了Access-Control-Allow-Origin: \*。

## 和Session方式存储id的差异

Session方式存储用户id的最大弊病在于要占用大量服务器内存，对于较大型应用而言可能还要保存许多的状态。一般而言，大型应用还需要借助一些KV数据库和一系列缓存机制来实现Session的存储。

而JWT方式将用户状态分散到了客户端中，可以明显减轻服务端的内存压力。除了用户id之外，还可以存储其他的和用户相关的信息，例如该用户是否是管理员等。

虽说JWT方式让服务器有一些计算压力（例如加密、编码和解码），但是这些压力相比磁盘I/O而言或许是半斤八两。具体是否采用，需要在不同场景下用数据说话。

# JWT的构成

JWT是由三段信息构成的：头部（header)、载荷（payload, 类似于飞机上承载的物品)、签名（signature)。将这三段信息文本用.链接一起就构成了JWT字符串。就像这样:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

## header

jwt的头部承载两部分信息：

* 声明类型，这里是jwt
* 声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：

{

'typ': 'JWT',

'alg': 'HS256'

}

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.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

## playload

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
* 公共的声明
* 私有的声明

**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
* 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（UNIX时间戳）
* exp: jwt的过期时间（UNIX时间戳），这个过期时间必须要大于签发时间
* 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:

{

"iss": "John Wu JWT",

"iat": 1441593502,

"exp": 1441594722,

"aud": "www.example.com",

"sub": "jrocket@example.com",

"from\_user": "B",

"target\_user": "A"

}

然后将其进行base64加密，得到Jwt的第二部分。

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9

## 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

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。

// javascript

var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);

var signature = HMACSHA256(encodedString, 'secret'); // TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

将这三部分用.连接成一个完整的字符串,构成了最终的jwt:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

**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。**

# 释疑

且慢，我们一定会有一些问题：

* 签名的目的是什么？
* Base64是一种编码，是可逆的，那么我的信息不就被暴露了吗？

## 签名的目的

最后一步签名的过程，实际上是对头部以及载荷内容进行签名。一般而言，加密算法对于不同的输入产生的输出总是不一样的。对于两个不同的输入，产生同样的输出的概率极其地小（有可能比我成世界首富的概率还小）。所以，我们就把“不一样的输入产生不一样的输出”当做必然事件来看待吧。

所以，如果有人对头部以及载荷的内容解码之后进行修改，再进行编码的话，那么新的头部和载荷的签名和之前的签名就将是不一样的。而且，如果不知道服务器加密的时候用的密钥的话，得出来的签名也一定会是不一样的。

服务器应用在接受到JWT后，会首先对头部和载荷的内容用同一算法再次签名。那么服务器应用是怎么知道我们用的是哪一种算法呢？别忘了，我们在JWT的头部中已经用alg字段指明了我们的加密算法了。

如果服务器应用对头部和载荷再次以同样方法签名之后发现，自己计算出来的签名和接受到的签名不一样，那么就说明这个Token的内容被别人动过的，我们应该拒绝这个Token，返回一个HTTP 401 Unauthorized响应。

所以，像颁发时间、过期时间等信息完全可以放在Token中，而不用在服务端通过session控制，不用担心被篡改。服务端只需要判断Token是否过期即可进行后续的操作（续期或者提示登录）。

## 信息会暴露？

是的！！所以，在JWT中，不应该在载荷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。

## Token过期后怎么办？

JWT实现的时候，一般会有两个过期时间：

* 第一个是Token本身的过期时间，这个时间一般1到2个小时，不能太长。
* 第二个是Token过期后，再次刷新的有效期，也就是Token过期后，你还有一段时间可以重新刷新，把过期的Token发给服务端，如果没有过刷新截止期，则服务端返回一个新的Token，不再需要通过用户名密码重新登录获取Token了。

所以为了减少过期后重新获取Token所带来的麻烦，我们一般在每次Http请求成功后，将目前的Token刷新，然后可以在Http响应中返回新的Token。

JWT由于过期数据(exp claim)是封装在Payload中的，所以必须返回一个新Token，而不是在旧Token的基础上刷新。

但是在并发的时候也会出现问题，如果前一个请求刷新了Token(为了安全，刷新后一般会把旧Token加入黑名单)，后面的请求使用了一个旧的Token像服务请求数据，这个时候请求会被拒绝。

可以说这真的是JWT的一个缺陷，目前没有特别好的办法来解决并发刷新的问题。不过可以通过设置一个宽限时间，在Token刷新后，如果旧Token仍处于刷新宽限时间内，就放行。

# 应用

## 工作时序

一般是在请求头里加入Authorization，并加上Bearer标注：

fetch('api/user/1', {

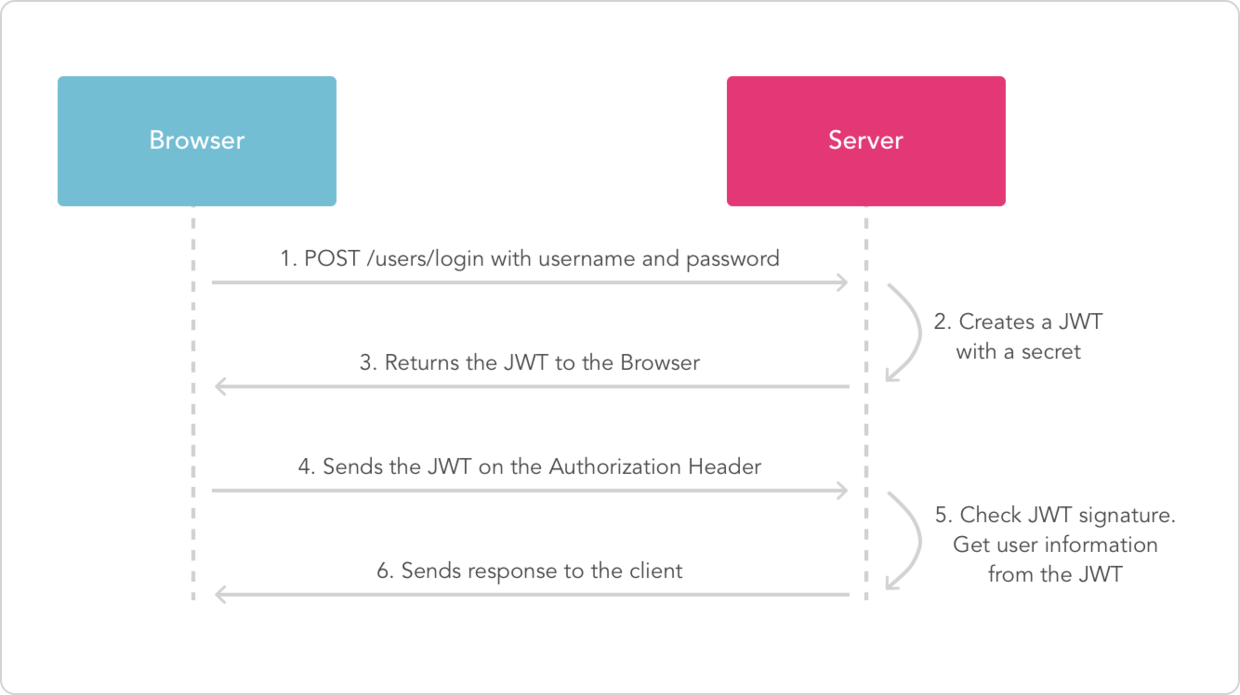
headers: {

'Authorization': 'Bearer ' + token

}

})

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:



## 信任链接

让我们来假想一下一个场景。在A用户关注了B用户的时候，系统发邮件给B用户，并且附有一个链接“点此关注A用户”。链接的地址可以是这样的

|  |  |
| --- | --- |
| 1 | https://your.awesome-app.com/make-friend/?from\_user=B&target\_user=A |

上面的URL主要通过URL来描述这个当然这样做有一个弊端，那就是要求用户B用户是一定要先登录的。可不可以简化这个流程，让B用户不用登录就可以完成这个操作。JWT就允许我们做到这点。

我们可以看到，JWT适合用于向Web应用传递一些非敏感信息。例如在上面提到的完成加好友的操作，还有诸如下订单的操作等等。

其实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。

## 单点登录

Session方式来存储用户id，一开始用户的Session只会存储在一台服务器上。对于有多个子域名的站点，每个子域名至少会对应一台不同的服务器，例如：

* www.taobao.com
* nv.taobao.com
* nz.taobao.com
* login.taobao.com

所以如果要实现在login.taobao.com登录后，在其他的子域名下依然可以取到Session，这要求我们在多台服务器上同步Session。

使用JWT的方式则没有这个问题的存在，因为用户的状态已经被传送到了客户端。因此，我们只需要将含有JWT的Cookie的domain设置为顶级域名即可，例如

|  |  |
| --- | --- |
| 1 | Set-Cookie: jwt=lll.zzz.xxx; HttpOnly; max-age=980000; domain=.taobao.com |

注意domain必须设置为一个点加顶级域名，即.taobao.com。这样，taobao.com和\*.taobao.com就都可以接受到这个Cookie，并获取JWT了。

# 开源工具

JWT有多个开源实现，本例采用“io.jsonwebtoken”包，即jjwt。

## 在maven中添加jar包依赖

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.6.0</version>

</dependency>

## JWT加密和解密的工具类

**package** com.thinkdef.core.api.authorization.token;

**import** io.jsonwebtoken.Claims;

**import** io.jsonwebtoken.Jws;

**import** io.jsonwebtoken.JwtBuilder;

**import** io.jsonwebtoken.Jwts;

**import** io.jsonwebtoken.SignatureAlgorithm;

**import** io.jsonwebtoken.impl.crypto.MacProvider;

**import** java.security.Key;

**import** java.util.Date;

**import** java.util.Map;

**import** javax.crypto.spec.SecretKeySpec;

**import** org.apache.commons.codec.binary.Base64;

/\*\*

\* JWT工具类。可以创建JWT、解析JWT

\* **@author** 张镇

\*/

**public** **class** JwtUtils {

// 设置发行人

**public** **static** String *ISSUER* = "zhangzhen";

// 设置抽象主题

**public** **static** String *SUBJECT* = "subject";

// HS256 私钥

**public** **static** String *HS256KEY* = "zhangzhen-c8";

// 签名用算法

**public** **static** SignatureAlgorithm *SIGNATURE\_ALGORITHM* = SignatureAlgorithm.*HS256*;

// 根据私钥生成签名用Key

**public** **static** Key *SIGNIG\_KEY* = **new** SecretKeySpec(

Base64.*decodeBase64*(*HS256KEY*), *SIGNATURE\_ALGORITHM*.getJcaName());

**public** **static** **long** *OUT\_TIMES* = 10000;

/\*\*

\* 生成JWT串

\* **@param** strLoginID 用户登录名。此项必填

\* **@param** mapClaims “载荷”信息，即需要加入到JWT中的内容

\* **@return** 生成完毕的JWT字符串

\*/

**public** **static** String createJWT(String strLoginID, Map<String, Object> mapClaims) {

// 获取当前时间，UNIX格式

**long** longNowMillis = System.*currentTimeMillis*();

// 给载荷中加入系统标准值

mapClaims.put(Claims.*ISSUER*, *ISSUER*);

mapClaims.put(Claims.*SUBJECT*, *SUBJECT*);

mapClaims.put(Claims.*AUDIENCE*, strLoginID);

mapClaims.put(Claims.*EXPIRATION*, **new** Date(longNowMillis + *OUT\_TIMES*));

mapClaims.put(Claims.*ISSUED\_AT*, **new** Date(longNowMillis));

// 声明JwtBuilder类，将设置好的载荷信息加入到对象中

JwtBuilder jwtBuilder = Jwts.*builder*().setClaims(mapClaims);

// 使用指定的算法（HS256），指定的私钥的KEY，对内容进行签名

jwtBuilder.signWith(*SIGNATURE\_ALGORITHM*, *SIGNIG\_KEY*);

// 返回生成完毕的JWT字符串

**return** jwtBuilder.compact();

}

/\*\*

\* 根据有效期判断，TOKEN是否有效

\* **@param** strToken Token字符串

\* **@return** true/false 是否有效

\*/

**public** **static** **boolean** isValid(String strToken) {

**try** {

// 使用私钥验证TOKEN是否合法，合法则还原为JWS对象

Jws<Claims> jwsClaims = Jwts.*parser*().setSigningKey(*SIGNIG\_KEY*)

.parseClaimsJws(strToken.trim());

// 从JWS对象中，获取过期时间

Long longExp = (Long) jwsClaims.getBody().get(Claims.*EXPIRATION*);

// 如果过期时间减当前时间大于0，则还在有效期中，返回true；如果过期时间减当前时间小于0，则已经过期，返回false。

**return** longExp - System.*currentTimeMillis*() > 0;

} **catch** (Exception e) {

e.printStackTrace();

// 如果TOKEN被篡改，则无法通过签名验证，会有异常。此时应返回false，即TOKEN无效

**return** **false**;

}

}

/\*\*

\* 解码JWS字符串

\* **@param** strToken oken字符串

\* **@return** 解析后的Claims对象，类似Map。

\*/

**public** **static** Map<String, Object> parseJWT(String strToken) {

// 使用私钥验证TOKEN是否合法，合法则还原为Claims对象

Claims claims = Jwts.*parser*().setSigningKey(*SIGNIG\_KEY*)

.parseClaimsJws(strToken.trim()).getBody();

**return** claims;

}

/\*\*

\* 获取HS512加密算法的KEY

\* **@return** 返回的KEY字符串

\*/

**public** **static** String getHS512Key() {

Key key = MacProvider.*generateKey*(SignatureAlgorithm.*HS512*);

String keyStr = Base64.*encodeBase64String*(key.getEncoded());

**return** keyStr;

}

}

## Token的发送与获取

ajax为例子：

beforeSend:function(request) {

request.setRequestHeader("token",token);

}

后台获取：

request.getHeader("token");

## Token验证机制

* 通过token解密是否成功可以判断token是否正确或者是否过期。
* 解密完成，可以对比用户属性或者用户的固定token（缓存中或者放入数据库）。

## Token过滤器

实现过滤器。我们这里实现了过滤器来判断是否有token,以及token是否过期或者token不正确(伪造的)。在这里如果正确我们也可以把信息，比如用户id,角色等信息写入上下文，在其他操作的时候就不需要在此再数据库中查询信息了。

**package** com.thinkdef.core.api.authorization.token;

**import** java.io.IOException;

**import** javax.ws.rs.WebApplicationException;

**import** javax.ws.rs.container.ContainerRequestContext;

**import** javax.ws.rs.container.ContainerRequestFilter;

**import** javax.ws.rs.core.Response;

**import** org.glassfish.jersey.server.ContainerRequest;

**import** com.thinkdef.core.api.authorization.token.JwtUtils;

**import** com.upsoft.common.utils.LogUtil;

**public** **class** TokenFilter **implements** ContainerRequestFilter {

**public** **void** filter(ContainerRequestContext containerRequestContext)

**throws** IOException {

LogUtil.*debugLog*("进入TokenFilter");

// 声明JWT工具类

JwtUtils ju=**new** JwtUtils();

// 对访问的路径和访问的方法进行过滤

// 得到访问的方法 例如GET,POST

String method = containerRequestContext.getMethod().toLowerCase();

// 得到访问路径

String path = ((ContainerRequest) containerRequestContext)

.getPath(**true**).toLowerCase();

// 去掉路径结尾的分隔符

**if**(path.endsWith("/")){

path=path.substring(0, path.length()-1);

}

// get方法访问login不需要验证，post验证不需验证,注册不需要验证。

**if** (

(

"get".equals(method) &&

("login".equals(path))

) ||

(

"post".equals(method) &&

("authentication".equals(path) || "regist".equals(path))

)

) {

// pass through the filter.

LogUtil.*debugLog*("不过滤部分内容。离开TokenFilter");

**return**;

}**else**{

// 获取头信息中的token

String strToken = ((ContainerRequest) containerRequestContext)

.getHeaderString("token");

// 如果token为空抛出

**if** (strToken == **null**) {

LogUtil.*debugLog*("Token为空，未通过认证。离开TokenFilter");

**throw** **new** WebApplicationException(Response.Status.*UNAUTHORIZED*);// 抛出未认证的错误

}

// 把Bear Token换成Token

**if** ((strToken != **null**) && (strToken.length() > 7))

{

String HeadStr = strToken.substring(0, 6).toLowerCase();

**if** (HeadStr.compareTo("bearer") == 0)

{

strToken = strToken.substring(7, strToken.length());

**if** (ju.isValid(strToken)) {

LogUtil.*debugLog*("认证通过。离开TokenFilter");

**return**;

} **else** {

LogUtil.*debugLog*("未通过认证。离开TokenFilter");

**throw** **new** WebApplicationException(Response.Status.*UNAUTHORIZED*);

}

}

}

}

}

}

## 资源注册管理器（Jersey）

如果使用Jersey，可以使用以下的资源注册管理器，将前面的Token过滤器注册到系统中，对所有针对接口的访问进行拦截过滤。

**package** com.thinkdef.core.api;

**import** org.glassfish.jersey.filter.LoggingFilter;

**import** org.glassfish.jersey.server.ResourceConfig;

**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;

**import** com.thinkdef.core.api.authorization.token.TokenFilter;

**import** com.thinkdef.core.api.resource.Login;

**public** **class** ApplicationAPI **extends** ResourceConfig{

**public** ApplicationAPI(){

// 注册过滤器

register(TokenFilter.**class**);

// 注册数据转换器

register(JacksonJsonProvider.**class**);

// 注册日志

register(LoggingFilter.**class**);

register(Login.**class**);

}

}

# 总结

## 优点

* 因为json的通用性，所以JWT是可以进行跨语言支持的，像JAVA,JavaScript,NodeJS,PHP等很多语言都可以使用。
* 因为有了payload部分，所以JWT可以在自身存储一些其他业务逻辑所必要的非敏感信息。
* 便于传输，jwt的构成非常简单，字节占用很小，所以它是非常便于传输的。
* 它不需要在服务端保存会话信息, 所以它易于应用的扩展

## 安全相关

* 不应该在jwt的payload部分存放敏感信息，因为该部分是客户端可解密的部分。
* 保护好secret私钥，该私钥非常重要。
* 如果可以，请使用https协议