流云诸葛

前端，产品，管理 Fork me on GitHub

~ JWT实现token-based会话管理

阅读目录

上文《 3种web会话管理的方式 》介绍了3种会话管理的方式，其中token-based的方式有必要从实现层面了解一下。本文主要介绍这方面的内容。上文提到token-based的实现目前有一个开放的标准可用，这个标准就是JWT，从它的 官网 上也能看到，目前实现了JWT的技术非常多，基本上涵盖了所有的语言平台。本文选择 express 和 jsonwebtoken 基于nodejs来实现token-based会话管理。

相关代码： https://github.com/liuyunzhuge/blog/tree/master/node_jwt

demo的说明我会在本文第二部分介绍，下面先介绍一下JWT的相关知识。

认识JWT

JSON Web Token（JWT）是一个开放标准（RFC 7519），它定义了一种紧凑和自包含的方式，用于在各方之间作为JSON对象安全地传输信息。作为标准，它没有提供技术实现，但是大部分的语言平台都有按照它规定的内容提供了自己的技术实现，所以实际在用的时候，只要根据自己当前项目的技术平台，到官网上选用合适的实现库即可。

使用JWT来传输数据，实际上传输的是一个字符串，这个字符串就是所谓的json web token字符串。所以**广义上，JWT是一个标准的名称；狭义上，JWT指的就是用来传递的那个token字符串。**这个串有两个特点：
1）紧凑：指的是这个串很小，能通过url 参数，http 请求提交的数据以及http header的方式来传递；
2）自包含：这个串可以包含很多信息，比如用户的id、角色等，别人拿到这个串，就能拿到这些关键的业务信息，从而避免再通过数据库查询等方式才能得到它们。

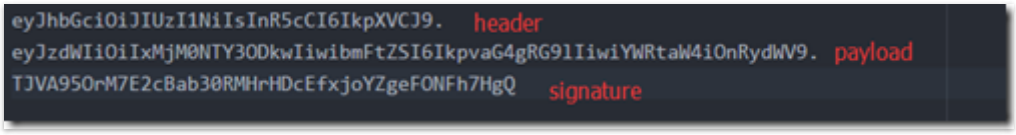
通常一个JWT是长这个样子的（这个串本来是不会换行的，为了让这个串看起来的样子跟后面要介绍的数据结构对应起来才手工加的换行）：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

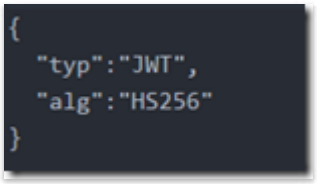
要知道一个JWT是怎么产生以及如何用于会话管理，只要弄清楚JWT的数据结构以及它签发和验证的过程即可。

1）JWT的数据结构以及签发过程

一个JWT实际上是由三个部分组成：header（头部）、payload（载荷）和signature（签名）。这三个部分在JWT里面分别对应**英文句号**分隔出来的三个串：



先来看header部分的结构以及它的生成方法。header部分是由下面格式的json结构生成出来：



这个json中的typ属性，用来标识整个token字符串是一个JWT字符串；它的alg属性，用来说明这个JWT签发的时候所使用的签名和摘要算法，常用的值以及对应的算法如下：

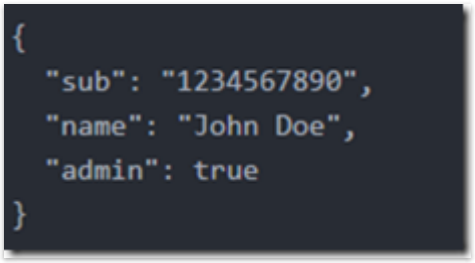
alg Parameter Value	Digital Signature or MAC Algorithm
HS256	HMAC using SHA-256 hash algorithm
HS384	HMAC using SHA-384 hash algorithm
HS512	HMAC using SHA-512 hash algorithm
RS256	RSASSA using SHA-256 hash algorithm
RS384	RSASSA using SHA-384 hash algorithm
RS512	RSASSA using SHA-512 hash algorithm
ES256	ECDSA using P-256 curve and SHA-256 hash algorithm
ES384	ECDSA using P-384 curve and SHA-384 hash algorithm
ES512	ECDSA using P-521 curve and SHA-512 hash algorithm
none	No digital signature or MAC value included

typ跟alg属性的全称其实是type跟algorithm，分别是类型跟算法的意思。之所以都用三个字母来表示，也是基于JWT最终字串大小的考虑，同时也是跟JWT这个名称保持一致，这样就都是三个字符了…**typ跟alg是JWT中标准中规定的属性名称**，虽然在签发JWT的时候，也可以把这两个名称换掉，但是如果随意更换了这个名称，就有可能在JWT验证的时候碰到问题，因为拿到JWT的人，默认会根据typ和alg去拿JWT中的header信息，当你改了名称之后，显然别人是拿不到header信息的，他又不知道你把这两个名字换成了什么。JWT作为标准的意义在于统一各方对同一个事情的处理方式，各个使用方都按它约定好的格式和方法来签发和验证token，这样即使运行的平台不一样，也能够保证token进行正确的传递。

一般签发JWT的时候，header对应的json结构只需要typ和alg属性就够了。JWT的header部分是把前面的json结构，经过Base64Url编码之后生成出来的：

（在线base64编码： http://www1.tc711.com/tool/BASE64.htm ）

再来看payload部分的结构和生成过程。payload部分是由下面**类似**格式的json结构生成出来：



payload的json结构并不像header那么简单，**payload用来承载要传递的数据，它的json结构实际上是对JWT要传递的数据的一组声明，这些声明被JWT标准称为claims，它的一个“属性值对”其实就是一个claim，每一个claim的都代表特定的含义和作用。**比如上面结构中的sub代表这个token的所有人，存储的是所有人的ID；name表示这个所有人的名字；admin表示所有人是否管理员的角色。**当后面对JWT进行验证的时候，这些claim都能发挥特定的作用。**

根据JWT的标准，这些claims可以分为以下三种类型：
a. **Reserved claims（保留）**，它的含义就像是编程语言的保留字一样，属于JWT标准里面规定的一些claim。JWT标准里面定好的claim有：

- iss(Issuser)：代表这个JWT的签发主体；
- sub(Subject)：代表这个JWT的主体，即它的所有人；
- aud(Audience)：代表这个JWT的接收对象；
- exp(Expiration time)：是一个时间戳，代表这个JWT的过期时间；
- nbf(Not Before)：是一个时间戳，代表这个JWT生效的开始时间，意味着在这个时间之前验证JWT是会失败的；
- iat(Issued at)：是一个时间戳，代表这个JWT的签发时间；
- jti(JWT ID)：是JWT的唯一标识。

b. **Public claims**，略（不重要）

c. **Private claims**，这个指的就是自定义的claim。比如前面那个结构举例中的admin和name都属于自定的claim。这些claim跟JWT标准规定的claim区别在于：JWT规定的claim，JWT的接收方在拿到JWT之后，都知道怎么对这些标准的claim进行验证；而private claims不会验证，除非明确告诉接收方要对这些claim进行验证以及规则才行。

按照JWT标准的说明：保留的claims都是可选的，在生成payload不强制用上面的那些claim，你可以完全按照自己的想法来定义payload的结构，不过这样搞根本没必要：第一是，如果把JWT用于认证， 那么JWT标准内规定的几个claim就足够用了，甚至只需要其中一两个就可以了，假如想往JWT里多存一些用户业务信息，比如角色和用户名等，这倒是用自定义的claim来添加；第二是，JWT标准里面针对它自己规定的claim都提供了有详细的验证规则描述，每个实现库都会参照这个描述来提供JWT的验证实现，所以如果是自定义的claim名称，那么你用到的实现库就不会主动去验证这些claim。

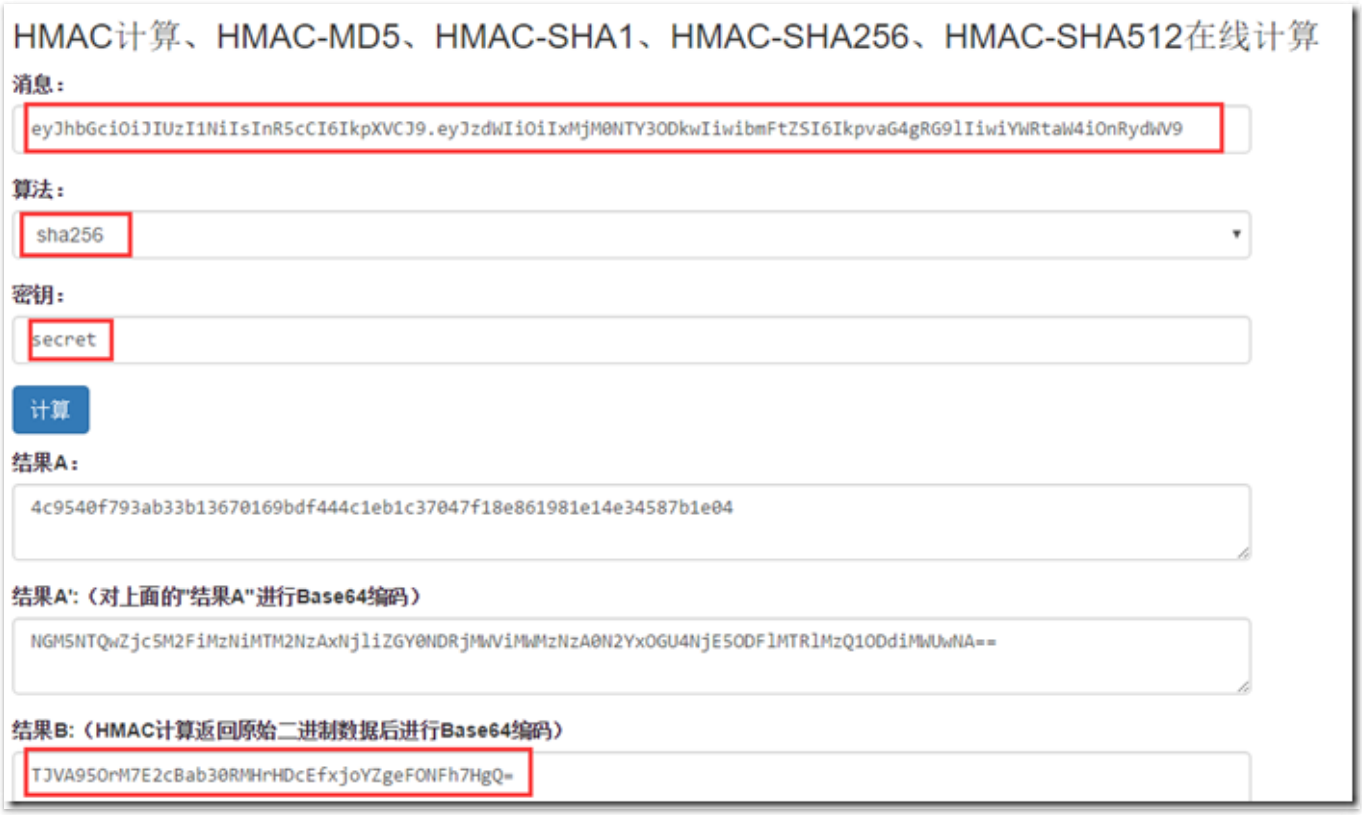
最后也是把这个json结构做base64url编码之后，就能生成payload部分的串：

（在线base64编码： http://www1.tc711.com/tool/BASE64.htm ）

最后看signature部分的生成过程。签名是把header和payload对应的json结构进行base64url编码之后得到的两个串用英文句点号拼接起来，然后根据header里面alg指定的签名算法生成出来的。算法不同，签名结果不同，但是不同的算法最终要解决的问题是一样的。以alg: HS256为例来说明前面的签名如何来得到。按照前面alg可用值的说明，HS256其实包含的是两种算法：HMAC算法和SHA256算法，前者用于生成摘要，后者用于对摘要进行数字签名。这两个算法也可以用HMACSHA256来统称。运用HMACSHA256实现signature的算法是：

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

正好找到一个在线工具能够测试这个签名算法的结果，比如我们拿前面的header和payload串来测试，并把“secret”这个字符串就当成密钥来测试：



（ https://1024tools.com/hmac ）

最后的结果B其实就是JWT需要的signature。不过对比我在介绍JWT的开始部分给出的JWT的举例：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

会发现通过在线工具生成的header与payload都与这个举例中的对应部分相同，但是通过在线工具生成的signature与上面图中的signature有细微区别，在于最后是否有“=”字符。这个区别产生的原因在于上图中的JWT是通过JWT的实现库签发的JWT，这些实现库最后编码的时候都用的是base64url编码，而前面那些在线工具都是bas64编码，这 两种编码方式 不完全相同，导致编码结果有区别。

以上就是一个JWT包含的全部内容以及它的签发过程。接下来看看该如何去验证一个JWT是否为一个有效的JWT。

2）JWT的验证过程

这个部分介绍JWT的验证规则，主要包括签名验证和payload里面各个标准claim的验证逻辑介绍。只有验证成功的JWT，才能当做有效的凭证来使用。

先说签名验证。当接收方接收到一个JWT的时候，首先要对这个JWT的完整性进行验证，这个就是签名认证。它验证的方法其实很简单，只要把header做base64url解码，就能知道JWT用的什么算法做的签名，然后用这个算法，再次用同样的逻辑对header和payload做一次签名，并比较这个签名是否与JWT本身包含的第三个部分的串是否完全相同，只要不同，就可以认为这个JWT是一个被篡改过的串，自然就属于验证失败了。接收方生成签名的时候必须使用跟JWT发送方相同的密钥，意味着要做好密钥的安全传递或共享。

再来看payload的claim验证，拿前面标准的claim来一一说明：

- iss(Issuser)：如果签发的时候这个claim的值是“a.com”，验证的时候如果这个claim的值不是“a.com”就属于验证失败；
- sub(Subject)：如果签发的时候这个claim的值是“liuyunzhuge”，验证的时候如果这个claim的值不是“liuyunzhuge”就属于验证失败；
- aud(Audience)：如果签发的时候这个claim的值是“['b.com','c.com']”，验证的时候这个claim的值至少要包含b.com，c.com的其中一个才能验证通过；
- exp(Expiration time)：如果验证的时候超过了这个claim指定的时间，就属于验证失败；
- nbf(Not Before)：如果验证的时候小于这个claim指定的时间，就属于验证失败；
- iat(Issued at)：它可以用来做一些maxAge之类的验证，假如验证时间与这个claim指定的时间相差的时间大于通过maxAge指定的一个值，就属于验证失败；
- jti(JWT ID)：如果签发的时候这个claim的值是“1”，验证的时候如果这个claim的值不是“1”就属于验证失败

需要注意的是，在验证一个JWT的时候，签名认证是每个实现库都会自动做的，但是payload的认证是由使用者来决定的。因为JWT里面可能不会包含任何一个标准的claim，所以它不会自动去验证这些claim。

以登录认证来说，在签发JWT的时候，完全可以只用sub跟exp两个claim，用sub存储用户的id，用exp存储它本次登录之后的过期时间，然后在验证的时候仅验证exp这个claim，以实现会话的有效期管理。

以上就是我觉得需要介绍的JWT的各方面的内容，希望大家能看的明白。主要参考的资料有：

https://jwt.io/introduction/

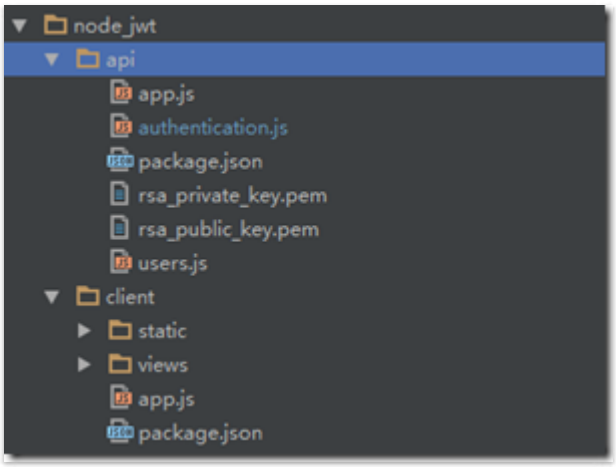
http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html

https://www.iana.org/assignments/jwt/jwt.xml#IESG

接下来看看本文相关demo的内容。

demo要点说明

这个demo分为两个文件夹，一个api，一个client，分别模拟一个需要登录认证的服务，以及一个发起登录认证请求的客户端：



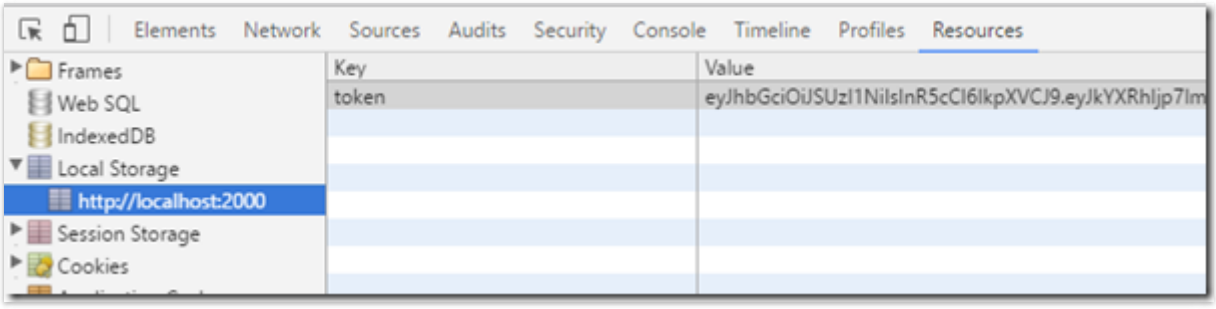
这两个文件下的内容都是用express框架简单搭建的，不了解express的话，可以去它官网上看看相关文档，这两个文件夹并没有用太多express的东西，主要满足demo的需要。

在这两个文件夹下分别运行node app命令，就能启动两个服务：

然后打开浏览输入 http://localhost:2000 ，就能看到客户端的服务了：

客户端的页面提供了三个接口调用的按钮，作用分别是发起登录验证（获取token），以及登录验证后获取用户信息（获取用户信息），模拟退出（销毁token）。只有登录验证之后，获取用户信息的接口才能拿到数据。

客户端在token-based的认证里面，主要是完成token的保存和发送工作。当token从服务器返回后，我把它直接存放到了localStorage里面：



然后当发送请求的时候，我会从localStorage里面拿出来，然后把token以Bearer token的形式加到http Authorization这个header里面：

```
function getUser() {
    var token = localStorage.getItem('token');
    if(!token) {
        alert('请先登录');
        return;
    }
    $.ajax({
        url: 'http://localhost:3000/api/getUser',
        type: 'GET',
        dataType: 'json',
        headers: {
            'Authorization': 'Bearer ' + token
        },
        success: function (res) {
            if (res.code == 200) {
                console.log(res.data);
                localStorage.setItem('token', res.token);
            } else if(res.code == 304) {
                alert('请先登录');
            }
        }
    });
}
```

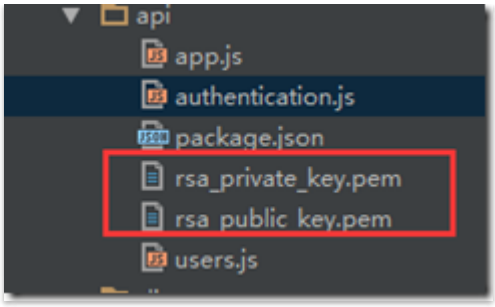
当ajax请求发送的时候，这个token就会跟着request header一起发送到服务端：

服务端在token-based认证里面主要的事情有：用户的验证、token的签发、从http中解析出token串、token的验证、token的刷新等。

由于这是个简单的demo，所以用户的验证，也没有用数据库查询这种级别的方式，直接用用户名密码写死的方式来处理，代码都在user.js这个模块里面。

token的签发和认证，我用的是 node-jsonwebtoken 这个JWT的实现，它基于nodejs，用起来相对比较简单，它的github主页都有详细的使用说明。

在前面介绍token的签发和签名认证的时候，我用的都是HS256的算法，这是考虑这个算法网上有在线工具可用。在demo里面，我用的是RS256的算法，这个算法由于用到RSA算法来加密解密，它是一个非对称加密的算法。需要一对密钥才能完成加密和解密。所以我用windows的openssl工具来生成rsa所需要的密钥对，也就是这两个文件：



这个工具可以从这个地址下载： https://indy.fulgan.com/SSL/
生成的方法可以参考： http://blog.csdn.net/yhl_jxy/article/details/51538332

在签发token的时候，我会读取这两个文件用于JWT的签发和验证：

```
//公钥
var pub_cert = fs.readFileSync(opts.pub_cert_file);
//私钥
var pri_cert = fs.readFileSync(opts.pri_cert_file);
```

整个token的管理我都封装在authentication.js这个模块里面。它的逻辑并不复杂，关键在于理解node-jsonwebtoken的用法，所以需要花点时间去它主页上看它使用说明才行。唯一需要补充一点的就是这个模块内如何从http里面解析出token串：

```
//从请求中获取token的字符串
_getReqToken: function (req) {
    var token = '';

    //获取token
    if (req.headers.authorization && req.headers.authorization.split(' ')[0] === 'Bearer') {
        token = req.headers.authorization.split(' ')[1];
    } else if (req.query && req.query[opts.query_name]) {
        token = req.query[opts.query_name];
    }

    return token;
},
```

其实也就是拿authorization这个header，然后按照Bearer token的格式进行解析就行了。考虑到token可能通过url传递，所以这里面也多加了一个直接从url解析token的处理。

客户端的主模块文件app.js没有要介绍的，服务端的主模块app.js内容较多，可以把一些要点再说明一下。首先因为token的管理都统一封装起来了，所以我在服务启动的时候就初始化了一个Authentication的实例：

```
//认证管理的组件实例
var auth = new Authentication({
    getCredentials: function (req) {
        return {
            username: req.body.username || req.query.username,
            password: req.body.password || req.query.password
        }
    },
    verifyIdentity: function (formData) {
        return users.getUser(formData.username, formData.password);
    }
});
```

它提供两个回调，分别用来从请求中获取用户密码，以及根据用户密码完成用户信息的验证。

然后我通过CORS（跨域资源共享）的设置来使得客户端的ajax请求能够顺利地从服务端拿到数据，而不会引发跨域的拦截：

```
//设置跨域访问
app.all('*', function (req, res, next) {
    res.header("Access-Control-Allow-Origin", "http://localhost:2000");
    res.header("Access-Control-Allow-Headers", "X-Requested-With,Authorization");
    res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE");
    res.header("X-Powered-By", '3.2.1');
    res.header("Content-Type", "application/json;charset=utf-8");
    next();
});
```

细心的话，在客户端里面，发起获取用户信息的请求时，会从network里面看到两个http请求，其中第一个请求是OPTIONS请求，这个是CORS导致的，如果想了解这个请求产生的具体原因，可以从以下两篇文章详细了解CORS的相关介绍：

https://developer.mozilla.org/en-US/docs/Web/HTTP/Server-Side_Access_Control

http://www.ruanyifeng.com/blog/2016/04/cors.html

最后在客户端对应的请求路由里面，我会继续用到authorization的实例来完成一些token相关的工作。比如这个登录的路由：

```
//获取token
app.get('/api/auth', function (req, res) {
    res.json({
        code: 200,
        token: auth.generateToken(req)
    });
});
```

最终通过authorization实例的generateToken方法来完成用户的登录信息验证和token的签发工作。

这个demo的代码其实很好理解，我也是从中抽取一些我认为比较关键的点拿到博客里来单独介绍，实际上你要是没看明白上面的某些内容，完全可以自己把demo弄到本地进行研究，相信那样会有更好的效果。如果遇到问题或者发现错误，欢迎随时跟我反馈交流。

小结

以上就是整个使用JWT来完成token-based会话管理的方案介绍。它跟我在上文介绍的内容其实有一个差别，就是JWT在传递的过程中其实仅仅只做了base64url编码，而不是加密处理，所以当别人拦截到正常用户的JWT的时候是很容易解码看到其中的信息的，尤其是一些重要的业务信息。所以在真正使用的时候，是值得对JWT做一次整体的加密和解密处理的。

分类: 架构设计

流云诸葛
关注 - 17
粉丝 - 956
荣誉： 推荐博客
+加关注

« 上一篇： 3种web会话管理的方式
» 下一篇： 看图理解JWT如何用于单点登录

posted @ 2016-11-24 08:39 流云诸葛 阅读(15082) 评论(3) 收藏

评论列表

#1楼 2017-04-09 13:55 zhujinhu

怎么退出销毁token的?

支持(1) 反对(0)

#2楼 2017-07-11 20:26 天下

感觉有一个问题，A 获取了一个token ， B拿到也能用，这样就不能限制每个用户的有效性

支持(0) 反对(0)

#3楼 2017-12-17 10:49 大~熊

非常感谢您的分享，还有个问题希望您能帮忙解答下，您将token存到了localstroage，像二楼说的那样，要是其他人从localstroage中获取了token不也可以认证通过了吗

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。