## 基于token的登陆

### ****token的意思是“令牌”，是服务端生成的一串字符串，作为客户端进行请求的一个标识****

## Token Auth的优点

Token机制相对于Cookie机制又有什么好处呢？

* **支持跨域访问**: Cookie是不允许垮域访问的，这一点对Token机制是不存在的，前提是传输的用户认证信息通过HTTP头传输.
* **无状态(也称：服务端可扩展行)**:Token机制在服务端不需要存储session信息，因为Token 自身包含了所有登录用户的信息，只需要在客户端的cookie或本地介质存储状态信息.
* **更适用CDN**: 可以通过内容分发网络请求你服务端的所有资料（如：javascript，HTML,图片等），而你的服务端只要提供API即可.
* **去耦**: 不需要绑定到一个特定的身份验证方案。Token可以在任何地方生成，只要在你的API被调用的时候，你可以进行Token生成调用即可.
* **更适用于移动应用**: 当你的客户端是一个原生平台（iOS, Android，Windows 8等）时，Cookie是不被支持的（你需要通过Cookie容器进行处理），这时采用Token认证机制就会简单得多。
* **CSRF**:因为不再依赖于Cookie，所以你就不需要考虑对CSRF（跨站请求伪造）的防范。
* **性能**: 一次网络往返时间（通过数据库查询session信息）总比做一次HMACSHA256计算 的Token验证和解析要费时得多.
* **不需要为登录页面做特殊处理**: 如果你使用Protractor 做功能测试的时候，不再需要为登录页面做特殊处理.
* **基于标准化**:你的API可以采用标准化的 JSON Web Token (JWT). 这个标准已经存在多个后端库（.NET, Ruby, Java,Python, PHP）和多家公司的支持（如：Firebase,Google, Microsoft）.

### token的运行原理

### token基本原理[#](https://www.cnblogs.com/loveer/p/11429034.html#3382584390)

Request指在一次请求的全过程中有效，即从http请求到服务器处理结束，返回响应的整个过程，存放在HttpServletRequest对象中。

Session是用户全局变量，在整个会话期间都有效。只要页面不关闭就一直有效（或者直到用户一直未活动导致会话过期，默认session过期时间为30分钟）

token 认证过程 \*

客户端使用用户名跟密码请求登录

服务端收到请求，去验证用户名与密码

验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

客户端收到 Token 以后可以把它存储起来，比如放在 session Storage 里或者 Local Storage 里

客户端每次向服务端请求资源的时候需要带着服务端签发的 Token

服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

因为token是被签名的，所以我们可以认为一个可以解码认证通过的token是由我们系统发放的，其中带的信息是合法有效的

## cookie 和session的区别

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗  
   考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能  
   考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：  
   将登陆信息等重要信息存放为SESSION  
   其他信息 如果需要保留，可以放在COOKIE中

### 基于JWT的Token认证[#](https://www.cnblogs.com/loveer/p/11429034.html#1352540416)

JSON Web Token（JWT）是一个非常轻巧的规范。这个规范允许我们使用JWT在用户和服务器之间传递安全可靠的信息。

JWT的组成：  
一个JWT实际上就是一个字符串，它由三部分组成，头部、载荷与签名

头部（Header）

用于描述关于该JWT的最基本的信息，例如其类型以及签名所用的算法等。这也可以被表示成一个JSON对象。

然后将其进行base64编码，得到第一部分

{"typ": "JWT",

"alg": "HS256"

}

载荷（Payload）

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

（base64是对称解密的，意味着该部分信息可以归类为明文信息）

然后将其进行base64编码，得到第二部分

{ "iss": "JWT Builder",

"iat": 1416797419,

"exp": 1448333419,

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

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

"Email": "aaa@example.com",

"Role": [ "admin", "user" ]

}

iss:

该JWT的签发者，是否使用是可选的；

sub:

该JWT所面向的用户，是否使用是可选的；

aud:

接收该JWT的一方，是否使用是可选的；

exp(expires):

什么时候过期，这里是一个Unix时间戳，是否使用是可选的；

iat(issued at):

在什么时候签发的(UNIX时间)，是否使用是可选的；

nbf (Not Before):

如果当前时间在nbf里的时间之前，则Token不被接受；是否使用是可选的；

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。

签名（Signature）

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（在加密的时候，我们还需要提供一个密钥（secret），加盐secret组合加密）

然后就构成了jwt的第三部分。

最后，将这一部分签名也拼接在被签名的字符串后面，我们就得到了完整的JWT

注意：secret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

## 起源

说起JWT，我们应该来谈一谈基于token的认证和传统的session认证的区别。

## 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: \*。