# OAuth2.0 简介

## 常用的登录认证的三种方式

### 1 OAuth 认证

维基百科对它的解释摘要如下：

OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的网站（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。这样，OAuth让用户可以授权第三方网站访问他们存储在另外服务提供者的某些特定信息，而非所有内容。

**认证过程大致如下：**

1. 我们电商平台的用户过来登录，常用场景是点击“微信登录”按钮；
2. 接下来，用户终端将用户引导到微信授权页面；
3. 用户同意授权，应用服务器重定向到之前设置好的 **redirect\_uri** （应用服务器所在的地址），并附带上授权码（code）;
4. 应用服务器用上一步获取的 code 向微信授权服务器发送请求，获取 **access\_token**，也就是上面说的令牌；
5. 之后应用服务器用上一步获取的 **access\_token** 去请求微信授权服务器获取用户的基本信息，例如头像、昵称等；

### 2 Cookie-Sesssion 认证

早期互联网以 web 为主，客户端是浏览器，所以 Cookie-Session 方式最那时候最常用的方式，直到现在，一些 web 网站依然用这种方式做认证。

**认证过程大致如下：**

1. 用户输入用户名、密码或者用短信验证码方式登录系统；
2. 服务端验证后，创建一个 Session 信息，并且将 SessionID 存到 cookie，发送回浏览器；
3. 下次客户端再发起请求，自动带上 cookie 信息，服务端通过 cookie 获取 Session 信息进行校验；

**弊端**

* 只能在 web 场景下使用，如果是 APP 中，不能使用 cookie 的情况下就不能用了；
* 即使能在 web 场景下使用，也要考虑跨域问题，因为 cookie 不能跨域；
* cookie 存在 CSRF（跨站请求伪造）的风险；
* 如果是分布式服务，需要考虑 Session 同步问题；

### 3 Cookie-Session 改造版

由于传统的 Cookie-Session 认证存在诸多问题，可以把上面的方案改造一下。改动的地方如下：

* 不用 cookie 做客户端存储，改用其他方式，web 下使用 local storage，APP 中使用客户端数据库，这样就实现了跨域，并且避免了 CSRF ;
* 服务端也不存 Session 了，把 Session 信息拿出来存到 Redis 等内存数据库中，这样即提高了速度，又避免了 Session 同步问题；

**经过改造之后变成了如下的认证过程：**

1. 用户输入用户名、密码或者用短信验证码方式登录系统；
2. 服务端经过验证，将认证信息构造好的数据结构存储到 Redis 中，并将 key 值返回给客户端；
3. 客户端拿到返回的 key，存储到 local storage 或本地数据库；
4. 下次客户端再次请求，把 key 值附加到 header 或者 请求体中；
5. 服务端根据获取的 key，到 Redis 中获取认证信息；

### 4 基于JWT的Token认证

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。

**认证过程**

1. 依然是用户登录系统；
2. 服务端验证，将认证信息通过指定的算法（例如HS256）进行加密，例如对用户名和用户所属角色进行加密，加密私钥是保存在服务器端的，将加密后的结果发送给客户端，加密的字符串格式为三个"." 分隔的字符串 Token，分别对应**头部**、**载荷**与**签名**，头部和载荷都可以通过 base64 解码出来，签名部分不可以；
3. 客户端拿到返回的 Token，存储到 local storage 或本地数据库；
4. 下次客户端再次发起请求，将 Token 附加到 header 中；
5. 服务端获取 header 中的 Token ，通过相同的算法对 Token 中的用户名和所属角色进行相同的加密验证，如果验证结果相同，则说明这个请求是正常的，没有被篡改。这个过程可以完全不涉及到查询 Redis 或其他存储；

**优点**

* 使用 json 作为数据传输，有广泛的通用型，并且体积小，便于传输；
* 不需要在服务器端保存相关信息；
* jwt 载荷部分可以存储业务相关的信息（非敏感的），例如用户信息、角色等；

## OAuth

#### 是什么？

[OAuth](http://en.wikipedia.org/wiki/OAuth)是一个关于授权（authorization）的开放网络标准。

简单来说，OAuth2.0就是客户端和认证服务器之间由于相互**不信任**而产生的一个**授权协议**。只要授权方和被授权方遵守这个协议去写代码提供服务，那双方就是实现了OAuth模式。

OAuth2.0中最经典最常用的一种授权模式：**授权码模式。**

OAuth的作用就是让"客户端"安全可控地获取"用户"的授权，与"服务商提供商"进行互动。

OAuth在"客户端"与"服务提供商"之间，设置了一个授权层（authorization layer）。"客户端"不能直接登录"服务提供商"，只能登录授权层，以此将用户与客户端区分开来。"客户端"登录授权层所用的令牌（token），与用户的密码不同。用户可以在登录的时候，指定授权层令牌的权限范围和有效期。

"客户端"登录授权层以后，"服务提供商"根据令牌的权限范围和有效期，向"客户端"开放用户储存的资料。

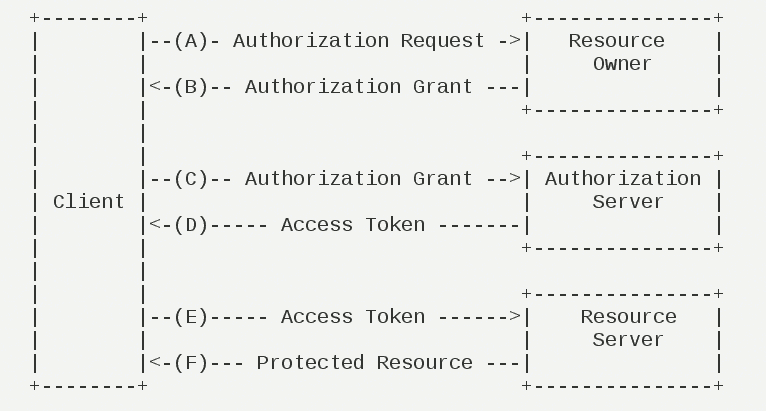
OAuth认证是为了做到第三方应用在未获取到用户敏感信息（如：账号密码、用户PIN等）的情况下，能让用户授权予他来访问开放平台（主要访问平台中的资源服务器Resource Server）中的资源接口。

#### 为什么？

传统的方式可能存在的问题：

1. 会泄露已信任网站的账号和密码泄露给第三方
2. 信任网站得部署专门的登录服务，并不安全
3. 没法设授权有效期，没法解除

#### 怎么用？



（A）用户打开客户端以后，客户端要求用户给予授权。

（B）用户同意给予客户端授权。

（C）客户端使用上一步获得的授权，向认证服务器申请令牌。

（D）认证服务器对客户端进行认证以后，确认无误，同意发放令牌。

（E）客户端使用令牌，向资源服务器申请获取资源。

（F）资源服务器确认令牌无误，同意向客户端开放资源。

不难看出来，上面六个步骤之中，B是关键，即用户怎样才能给于客户端授权。有了这个授权以后，客户端就可以获取令牌，进而凭令牌获取资源。

##### 客户端的授权模式：

客户端必须得到用户的授权（authorization grant），才能获得令牌（access token）。OAuth 2.0定义了四种授权方式。

###### A授权码模式（authorization code）

例：获取Token 接口

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

Content-Type: application/x-www-form-urlencoded

grant\_type=authorization\_code&code=SplxlOBeZQQYbYS6WxSbIA

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb

返回：

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token":"2YotnFZFEjr1zCsicMWpAA",

"token\_type":"example",

"expires\_in":3600,

"refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",

"example\_parameter":"example\_value"

}

从上面代码可以看到，相关参数使用JSON格式发送（Content-Type: application/json）。此外，HTTP头信息中明确指定不得缓存。

###### B简化模式（implicit）

简化模式（implicit grant type）不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了"授权码"这个步骤，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且客户端不需要认证。

下面是一个例子：

HTTP/1.1 302 Found

Location: <http://example.com/cb>#access\_token=2YotnFZFEjr1zCsicMWpAA

&state=xyz&token\_type=example&expires\_in=3600

在上面的例子中，认证服务器用HTTP头信息的Location栏，指定浏览器重定向的网址。注意，在这个网址的Hash部分包含了令牌。

根据上面的D步骤，下一步浏览器会访问Location指定的网址，但是Hash部分不会发送。接下来的E步骤，服务提供商的资源服务器发送过来的代码，会提取出Hash中的令牌。

###### C密码模式（resource owner password credentials）

密码模式（Resource Owner Password Credentials Grant）中，用户向客户端提供自己的用户名和密码。客户端使用这些信息，向"服务商提供商"索要授权。

在这种模式中，用户必须把自己的密码给客户端，但是客户端不得储存密码。这通常用在用户对客户端高度信任的情况下，比如客户端是操作系统的一部分，或者由一个著名公司出品。而认证服务器只有在其他授权模式无法执行的情况下，才能考虑使用这种模式。

示例：

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token":"2YotnFZFEjr1zCsicMWpAA",

"token\_type":"example",

"expires\_in":3600,

"refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",

"example\_parameter":"example\_value"

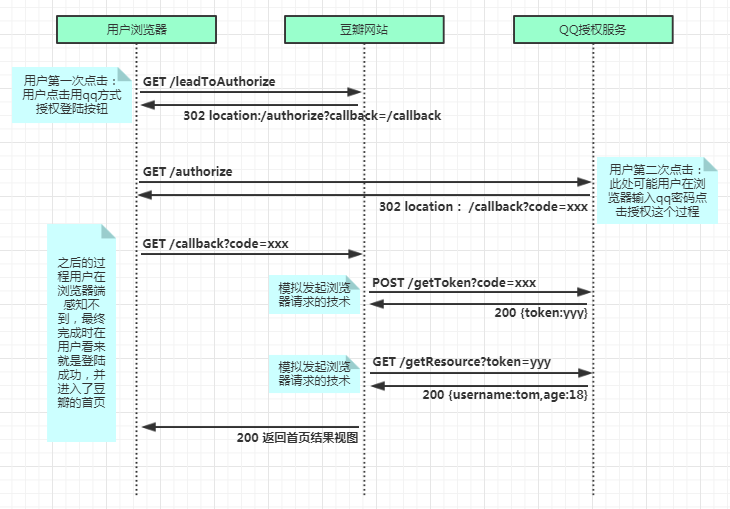
}

整个过程中，客户端不得保存用户的密码。

###### D客户端模式（client credentials）

客户端模式（Client Credentials Grant）指客户端以自己的名义，而不是以用户的名义，向"服务提供商"进行认证。严格地说，客户端模式并不属于OAuth框架所要解决的问题。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求"服务提供商"提供服务，其实不存在授权问题。

一张图看懂OAuth



## 单点登录

单点登录是指用户在某个应用系统上登录之后，进入其子应用或相关应用系统可以免去登录步骤，从而实现一次登录，到处操作。

SSO 登录服务器

多个业务服务器

参考文献：

1. 一张图搞定OAuth2.0

<https://www.cnblogs.com/flashsun/p/7424071.html>

1. 理解OAuth 2.0

<http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html>

1. 前端需了解的 SSO 和 CAS

<http://mp.weixin.qq.com/s/KQuMk6B-nxIC_ph6l0R97g>