# [[认证授权] 1.OAuth2授权](http://www.cnblogs.com/linianhui/p/oauth2-authorization.html)

*分类:*[[认证授权]](http://www.cnblogs.com/linianhui/category/929878.html)

*标签:*[OAuth2](http://www.cnblogs.com/linianhui/tag/OAuth2/)

# 1 OAuth2解决什么问题的？

举个栗子先。**小明**在**QQ空间**积攒了多年的照片，想挑选一些照片来打印出来。然后小明在找到一家提供在线打印并且包邮的网站（我们叫它**PP**吧（Print Photo缩写 😂））。

那么现在问题来了，小明有两个方案来得到打印的服务。

1. 在自己的QQ空间把想要打印的照片下载下来，然后提供给PP（直接发邮件给PP或者网盘共享给PP等等）。
2. 把自己的QQ账号密码给PP，然后告诉PP我要打印哪些照片。

针对方案（1）：小明要去下载这些照片，然后给PP，小明累觉不爱，，，

针对方案（2）：小明交出去自己的QQ账号密码，还要告诉PP哪些需要打印，哪些不需要，小明觉得自己有些小秘密不想给PP看，，，

小明觉得很痛苦，，，那么有没有不给PP账号密码，不下载照片，自己选哪些要打印直接扔给PP去打印的办法呢？OAuth走了过来扔给小明一块肥皂...

# 2 OAuth2简介

总结来说，OAuth2 是一个**开放授权标准，**它允许用户（小明）让第三方应用（PP）访问该用户在某服务的**特定私有资源**（QQ空间中小明的照片，可以不包含小明的小视频哦）但是不提供账号密码信息给第三方应用（PP）。

*有个小问题，为啥是OAuth2呢？1在哪？嗯，这个嘛，其实是有1和1.1版本的，只是因为1和1.1版本流程比较复杂，应用不是很广范，这里就不介绍了。据笔者以前做过的项目，Twitter是使用的OAuth1.1的版本，感兴趣的可以去了解下*[*https://dev.twitter.com/oauth*](https://dev.twitter.com/oauth)*。*

## 2.1 OAuth2的四个重要角色

进入正题，在OAuth2的完整授权流程中有4个重要的角色参与进来：

1. **Resource Owner**：资源拥有者，上面栗子中的**小明**；
2. **Resource Server**：资源服务器，上面栗子中的**QQ空间**，它是小明想要分享照片给PP的照片的提供方；
3. **Client**：第三方应用客户端，上面栗子中的**PP**，代指任何可以消费资源服务器的第三方应用；
4. **Authorization Server** ：授权服务器，管理**Resource Owner，Client和Resource Server的三角关系的中间层**。

其中Authorization server和Resource server可以是独立的服务提供商，也可以是在一起的，比如腾讯提供QQ空间作为资源服务器的同时也提供授权服务。

从这里可以看出，OAuth2在解决小明遇到的问题的过程中增加了一个**Authorization server**的角色。又印证了那句话，在计算机领域的所有问题都可以添加一个中间层来解决。

OAuth2解决问题的关键在于使用Authorization server提供一个访问凭据给Client，使得Client可以在不知道Resource owner在Resource server上的用户名和密码的情况下消费Resource owner的受保护资源。

# 3 部署OAuth2需要的完成的工作

由于OAuth2引入了Authorization server来管理Resource Owner，Client和Resource Server的三角关系，那么想要用上OAuth2，是实现以下功能的。

1. 增加一个Authorization server，提供授权的实现，一般由Resource server 来提供。
2. Resource server 为第三方应用程序提供注册接口。
3. Resource server 开放相应的受保护资源的API。
4. Client 注册成为Resource server的第三方应用。
5. Client 消费这些API。

作为资源服务提供商来说，1，2，3这三件事情是需要完成的。

作为第三方应用程序，要完成的工作是在4和5这两个步骤中。

其中作为Resource owner来说，是不用做什么的，是OAuth2受益的千千万万的最终人类用户。

## 3.1 作为Resource server

在一般情况下，Resource server提供Authorization server服务，主要提供两类接口：

1. 授权接口：接受Client的授权请求，引导用户到Resource server完成登陆授权的过程。
2. 获取访问令牌接口：使用授权接口提供的许可凭据来颁发Resource owner的访问令牌给Client，或者由Client更新过期的访问令牌。

除此之外，还需要提供一个第三方应用程序注册管理的服务。通常情况下会为注册完成的第三方应用程序分配两个成对出现的重要参数：

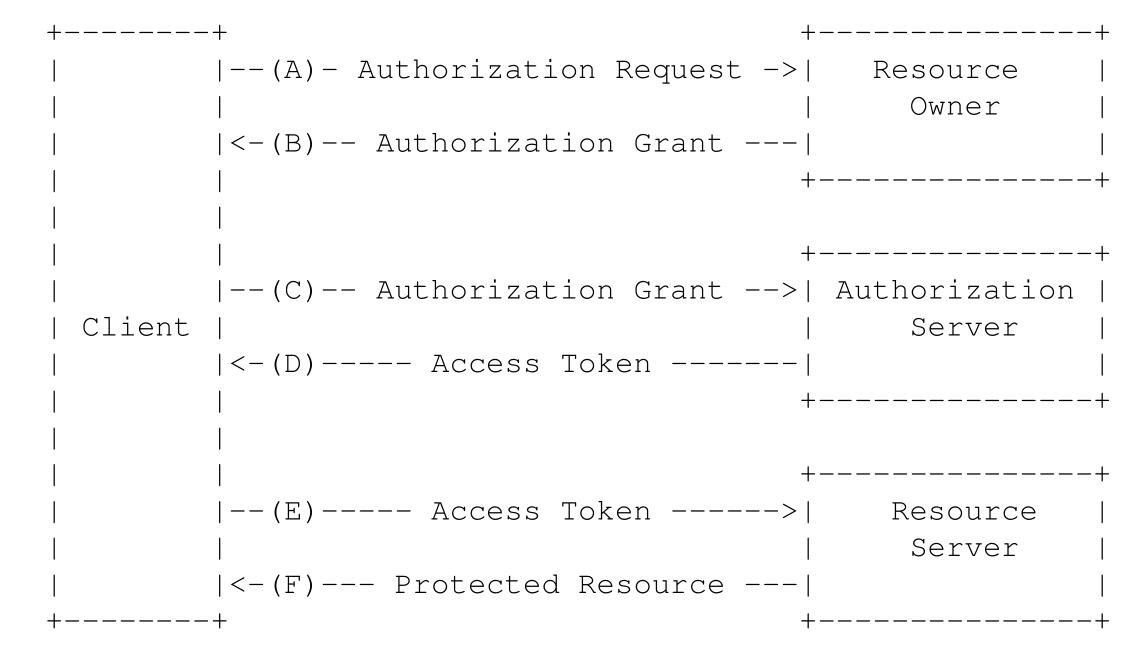
1. client\_id：第三方应用程序的一个标识id，这个信息通常是公开的信息，用来区分哪一个第三方应用程序。
2. client\_secret：第三方应用程序的私钥信息，这个信息是私密的信息，不允许在OAuth2流程中传递的，用于安全方面的检测和加密。

## 3.2 作为Client

在Client取得client\_id和client\_secret之后。使用这些信息来发起授权请求、获取access\_token请求和消费受保护的资源。

# 4 OAuth2的授权流程

贴个图瞅瞅OAuth2的工作流程：



在上述的OAuth完整流程中，（A）->（B）->（C）->（D）是授权的过程（参与者有**小明**，**PP**，**QQ空间**，**Authorization server**）；（E）->（F）是消费资源的过程（参与者有**PP**和**QQ空间**）。

* （A）**小明**访问**PP**，**PP**向**QQ空间**发起授权请求；
* （B）**QQ空间**接受**PP**的授权请求，并返回授权许可给**PP**；
* （C）**PP**使用授权许可向**Authorization server**发起请求**；**
* （D）**Authorization server**验证**PP**的身份和授权许可，发送访问令牌给**PP**；
* （E）**PP**用访问令牌请求**小明**存储在**QQ空间**的照片；
* （F）**QQ空间**根据访问令牌，返回**小明**的照片信息给**PP**。

这其中比较重要的一个概念是***访问令牌***，它代表的信息是整个OAuth2的核心，也是ABCD这些步骤最终要得到的信息。

访问令牌是对**PP可以在QQ空间访问小明的哪些信息**这个完整权限的一个抽象，比如PP要访问小李在QQ空间的照片，那么就是另外一个访问令牌了。

访问令牌背后抽象的信息有哪些呢?如下3类信息。

1. 客户端标识（比如**PP**）；
2. 用户标识（比如**小明**）；
3. 客户端能访问资源所有者的哪些资源以及其相应的权限。

有了这三类信息，那么资源服务器（Resouce Server）就可以区分出来是哪个第三方应用（Client）要访问哪个用户（Resource Owner）的哪些资源（以及有没有权限）。

# 5 OAuth2的4种授权许可

上一小节介绍了OAuth2的授权流程，除了访问令牌之外，还有一个重要的概念**授权许可（Authorization Grant）。**

书面化的方式解释就是**授权许可是一个代表资源所有者授权（访问受保护资源）的凭据，客户端用它来获取访问令牌**。读起来比较抽象，翻一下就是**授权许可是小明授予PP获得QQ空间访问令牌的一个凭据**。

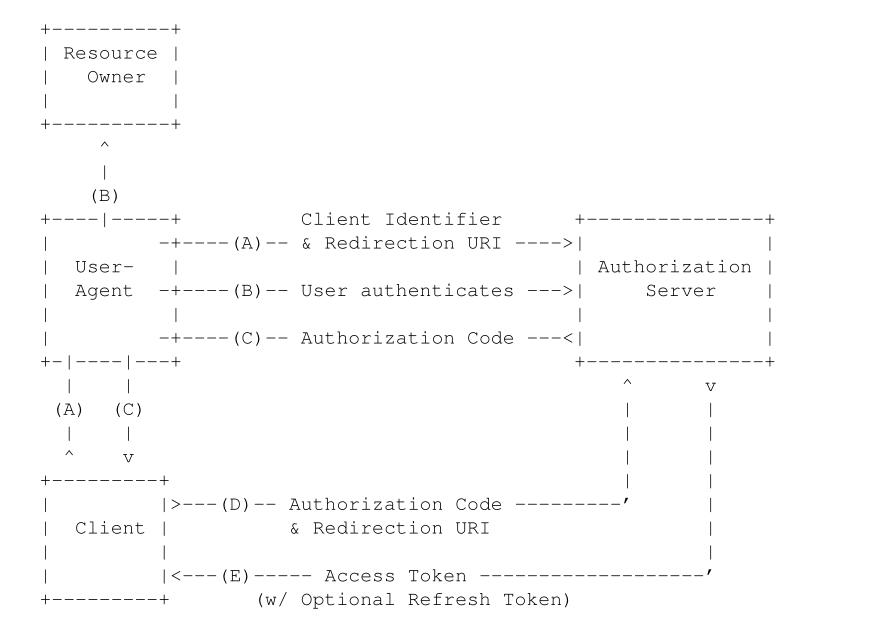
那么如何获得这个凭据呐，OAuth2定义了四种许可类型以及用于定义其他类型的可扩展性机制：

1. Authorization Code：授权码；
2. Implicit：隐式许可；
3. Resource Owner Password Credentials：资源所有者密码凭据；
4. Client Credentials ：客户端凭据。

**注意：以下4种授权许可是对上述(4. OAuth2的授权流程)中的ABDE四个阶段的展开。**

## ****5.1 Authorization Code****

这是OAuth2最常用的一种授权许可类型，比如QQ，微博，Facebook和豆瓣等等。要求Client具有可公开访问的Server服务器来接受**Authorization Code**，具体的流程如下：



上图ABCDE这5个步骤，既是完整的获取访问令牌的一个过程，其中引入了一些其他的概念，比如客户端标识，刷新令牌等和重定向URL等概念，后续会在[6. OAuth2附属概念和流程](http://www.cnblogs.com/linianhui/p/oauth2-authorization.html#auto_id_6)介绍。

* （A）**Client**使用浏览器（用户代理）访问**Authorization server。**也就是用浏览器访问一个URL，这个URL是**Authorization server**提供的，访问的收Client需要提供（客户端标识，请求范围，本地状态和重定向URL）这些参数。
* （B）**Authorization server**验证**Client**在（A）中传递的参数信息，如果无误则提供一个页面供**Resource owner**登陆，登陆成功后选择**Client**可以访问**Resource server**的哪些资源以及读写权限。
* （C）在（B）无误后返回一个**授权码（Authorization Code）**给Client**。**
* （D）**Client**拿着（C）中获得的**授权码（Authorization Code）**和（客户端标识、重定向URL等信息）作为参数，请求**Authorization server**提供的获取访问令牌的URL**。**
* （E）**Authorization server**返回**访问令牌**和可选的**刷新令牌**以及**令牌有效时间**等信息给**Client**。

### 5.1.1 Authorization Request

对应步骤（A），客户端提供以下参数请求Authorization Server：

1. response\_type：必选。值固定为“code”。
2. client\_id：必选。第三方应用的标识ID。
3. state：**推荐**。Client提供的一个字符串，服务器会原样返回给Client。
4. redirect\_uri：必选。授权成功后的重定向地址。
5. scope：可选。表示授权范围。

比如以下示例：

*GET /authorize?response\_type=code&client\_id=1&state=xyz&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Foauth2&scope=user,photo HTTP/1.1*

*Host: server.example.com*

### 5.1.2 Authorization Response

对应步骤（C），Authorization Server会返回如下信息：

1. code：授权码。
2. state：步骤（A）中客户端提供的state参数原样返回。

比如示例如下：

*HTTP/1.1 302 Found*

*Location: https://client.example.com/oauth2?code=SplxlOBeZQQYbYS6WxSbIA&state=xyz*

Location头部信息指向步骤（A）提供的redirect\_uri地址，同时携带code信息和state信息给client，这样浏览器在重定向的时候就会已GET的方式访问Client提供的redirect\_uri，同时Client接收到code信息和state信息。下一步就可以请求access\_token了。

### 5.1.3 Access Token Request

对应步骤（D），客户端提供以下参数请求Authorization Server：

1. grant\_type：必选。固定值“authorization\_code”。
2. code : 必选。Authorization Response 中响应的code。
3. redirect\_uri：必选。必须和Authorization Request中提供的redirect\_uri相同。
4. client\_id：必选。必须和Authorization Request中提供的client\_id相同。

比如以下示例：

*POST /token HTTP/1.1*

*Host: server.example.com*

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

*grant\_type=authorization\_code&code=123&client\_id=1&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Foauth2*

### 5.1.4 Access Token Response

对应步骤（E），Authorization Server会返回如下典型的信息：

1. access\_token：访问令牌。
2. refresh\_token：刷新令牌。
3. expires\_in：过期时间。

比如以下示例：

*HTTP/1.1 200 OK*

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

*{*

*"access\_token":"2YotnFZFEjr1zCsicMWpAA",*

*"token\_type":"example",*

*"expires\_in":3600,*

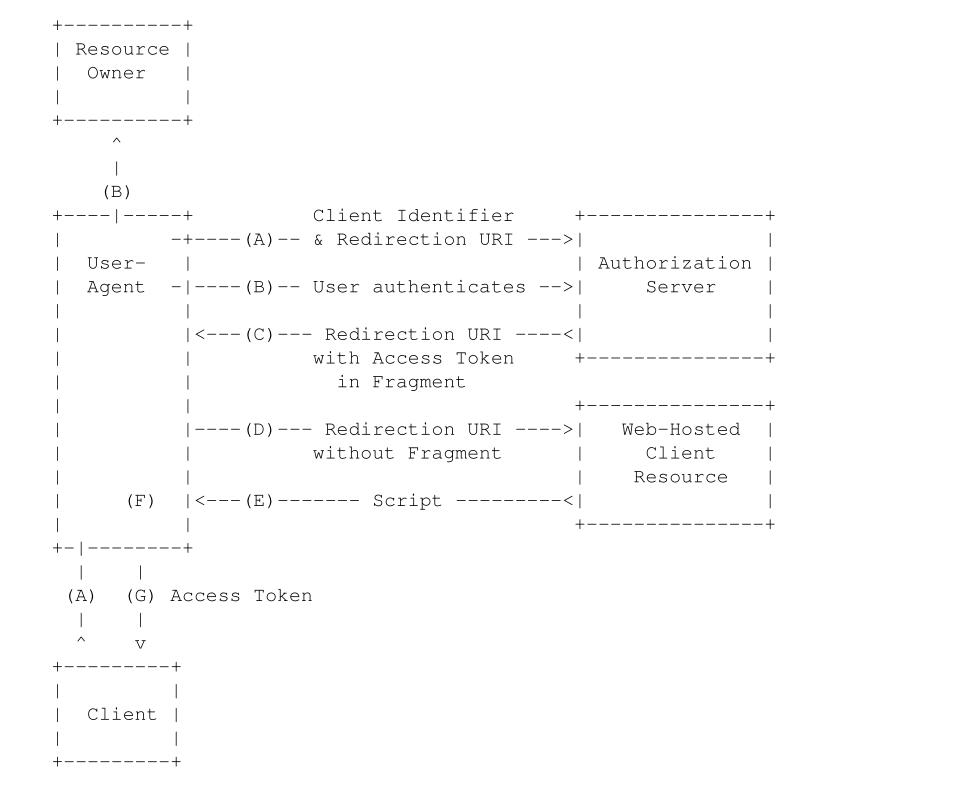
*"refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",*

*"example\_parameter":"example\_value"*

*}*

## 5.2 Implicit

这个是Authorization Code的简化版本。其中省略掉了颁发授权码（Authorization Code）给客户端的过程，而是直接返回访问令牌和可选的刷新令牌。其适用于没有Server服务器来接受处理Authorization Code的第三方应用，其流程如下：



其步骤就不做详细介绍了，相信大家都能理解。和Authorzation Code类型下重要的区分就是省略了Authorization Response和Access Token Request。而是直接由Authorization Request返回Access Token Response信息，具体如下。

### 5.2.1 Authorization Request

客户端提供以下参数请求Authorization Server：

1. response\_type：必选。值固定为“token”。
2. client\_id：必选。第三方应用的标识ID。
3. state：**推荐**。Client提供的一个字符串，服务器会原样返回给Client。
4. redirect\_uri：可选。授权成功后的重定向地址。
5. scope：可选。表示授权范围。

重点区别在于**response\_type为“token”**，而不再是“code”，redirect\_uri也变为了可选参数。

比如以下示例：

*GET /authorize?response\_type=token&client\_id=1&state=xyz&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Foauth2&scope=user,photo HTTP/1.1*

*Host: server.example.com*

### 5.2.2 Access Token Response

Authorization Server会返回如下典型的信息：

1. access\_token：访问令牌。
2. refresh\_token：刷新令牌。
3. expires\_in：过期时间。

比如以下示：

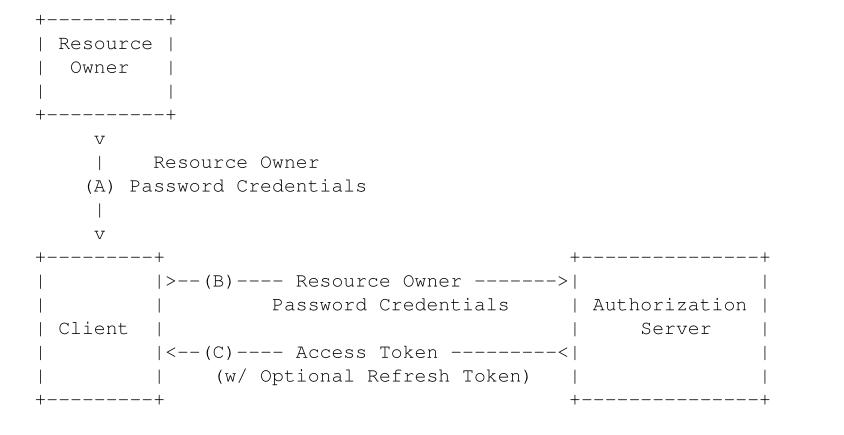
*HTTP/1.1 302 Found*

*Location: http://client.example.com/oauth2#access\_token=2YotnFZFEjr1zCsicMWpAA&state=xyz&expires\_in=3600*

注意其和Authorization Code的最大区别在于它是把token信息放在了url的hash部分（#后面），而不是作为参数(?后面)。这样浏览器在访问重定向的Location指定的url时，就不会把这些数据发送到服务器。而Client可以通过读取Location头信息中获取到access\_token信息。

## 5.3 Resource Owner Password Credentials Grant

看看流程图：



这种模式再一步简化，和Authorzation Code类型下重要的区分就是省略了Authorization Request和Authorization Response。而是Client直接使用Resource owner提供的username和password来直接请求access\_token（直接发起Access Token Request然后返回Access Token Response信息）。这种模式一般适用于Resource server高度信任第三方Client的情况下。

客户端提供以下参数请求Authorization Server：

1. grant\_type：必选。值固定为“password”。
2. username：必选。用户登陆名。
3. passward：必选**。**用户登陆密码。
4. scope：可选。表示授权范围。

比如以下示例：

*POST /token HTTP/1.1*

*Host: server.example.com*

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

*grant\_type=password&username=blackheart&password=1234*

Access Token Response和Authorization Code一致，就不列出来了。

## 5.4 Client Credentials Grant

这种类型就更简化了，Client直接已自己的名义而不是Resource owner的名义去要求访问Resource server的一些受保护资源。



客户端提供以下参数请求Authorization Server：

1. grant\_type：必选。值固定为“client\_credentials”。
2. scope：可选。表示授权范围。

比如以下示例：

*POST /token HTTP/1.1*

*Host: server.example.com*

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

*grant\_type=client\_credentials*

Access Token Response和Authorization Code一致，就不列出来了。

以笔者以前做公共号开发的经验，它提供由这类的OAuth2许可类型，这个场景下得到的access\_token的所属人是公众号的，可以用此access\_token来获取所有已关注的用户的信息，而不局限于特定的某一个用户，正是Client Credentials Grant这种类型的许可的用武之地，案例文档地址在文章最后面。

# 6 OAuth2刷新令牌

在上述得到访问令牌（access\_token）时，一般会提供一个过期时间和刷新令牌。以便在访问令牌过期失效的时候可以由客户端自动获取新的访问令牌，而不是让用户再次登陆授权。那么问题来了，是否可以把过期时间设置的无限大呢，答案是可以的，笔者记得Pocket的OAuth2拿到的访问令牌就是无限期的，好像豆瓣的也是。如下是刷新令牌的收客户端需要提供给Authorization Server的参数：

1. grant\_type：必选。固定值“refresh\_token”。
2. refresh\_token：必选。客户端得到access\_token的同时拿到的刷新令牌。

比如以下示例：

*POST /token HTTP/1.1*

*Host: server.example.com*

*grant\_type=refresh\_token&refresh\_token=tGzv3JOkF0XG5Qx2TlKWIA*

响应信息和5.1.4 Access Token Response保持一致。

# 7 Token传递方式

在第三方Client拿到access\_token后，如何发送给Resouce Server这个问题并没有在RFC6729种定义，而是作为一个单独的RFC6750来独立定义了。这里做以下简单的介绍，主要有三种方式如下：

1. URI Query Parameter.
2. Authorization Request Header Field.
3. Form-Encoded Body Parameter.

## 7.1 URI Query Parameter

这种使用途径应该是最常见的一种方式，非常简单，比如：

GET /resource?access\_token=mF\_9.B5f-4.1JqM HTTP/1.1

Host: server.example.com

在我们请求受保护的资源的Url后面追加一个access\_token的参数即可。另外还有一点要求，就是Client需要设置以下Request Header的**Cache-Control:no-store**，用来阻止access\_token不会被Web中间件给log下来，属于安全防护方面的一个考虑。

## 7.2 Authorization Request Header Field

因为在HTTP应用层协议中，专门有定义一个授权使用的Request Header，所以也可以使用这种方式：

GET /resource HTTP/1.1Host: server.example.comAuthorization: Bearer mF\_9.B5f-4.1JqM

其中"Bearer "是固定的在access\_token前面的头部信息。

## 7.3 Form-Encoded Body Parameter

使用Request Body这种方式，有一个额外的要求，就是Request Header的"Content-Type"必须是固定的“application/x-www-form-urlencoded”，此外还有一个限制就是不可以使用GET访问，这个好理解，毕竟GET请求是不能携带Request Body的。

POST /resource HTTP/1.1Host: server.example.comContent-Type: application/x-www-form-urlencodedaccess\_token=mF\_9.B5f-4.1JqM

# 8 OAuth2的安全问题

在OAuth2早期的时候爆发过不少相关的安全方面的漏洞，其实仔细分析后会发现大都都是没有严格遵循OAuth2的安全相关的指导造成的，相关的漏洞事件百度以下就有了。

其实OAuth2在设计之初是已经做了很多安全方面的考虑，并且在RFC6749中加入了一些安全方面的规范指导。比如

1. 要求Authorization server进行有效的Client验证；
2. client\_serect,access\_token,refresh\_token,code等敏感信息的安全存储（不得泄露给第三方）、传输通道的安全性（TSL的要求）；
3. 维持refresh\_token和第三方应用的绑定，刷新失效机制；
4. 维持Authorization Code和第三方应用的绑定，这也是state参数为什么是推荐的一点，以防止CSRF；
5. 保证上述各种令牌信息的不可猜测行，以防止被猜测得到；

安全无小事，这方面是要靠各方面（开放平台，第三方开发者）共同防范的。如QQ互联的OAuth2 API中，state参数是强制必选的参数，授权接口是基于HTTPS的加密通道等；同时作为第三方开发者在使用消费这些服务的时候也应该遵循其相关的安全规范。

# 9 总结 & 参考 & 案例

OAuth2是一种**授权**标准框架，用来解决的是第三方服务在无需用户提供账号密度的情况下访问用户的私有资源的一套流程规范。与其配套的还有其他相关的规范，都可以到<https://oauth.net/2/>去延伸阅读和了解。

相关参考：

<https://oauth.net/2/>

<https://www.oauth.com/>

<https://aaronparecki.com/oauth-2-simplified/>

[RFC6749 : The OAuth 2.0 Authorization Framework](https://tools.ietf.org/html/rfc6749)

[RFC6749中文版（https://github.com/jeansfish/RFC6749.zh-cn）](https://github.com/jeansfish/RFC6749.zh-cn)

[RFC6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage](https://tools.ietf.org/html/rfc6750).

[RFC6819 - OAuth 2.0 Threat Model and Security Considerations](https://tools.ietf.org/html/rfc6819).

OAuth2案例：

[豆瓣OAuth2 API（Authorization Code）](https://developers.douban.com/wiki/?title=oauth2)

[QQ OAuth2 API（Authorization Code）](http://wiki.connect.qq.com/%e4%bd%bf%e7%94%a8authorization_code%e8%8e%b7%e5%8f%96access_token)

[豆瓣OAuth2 API（Implicit )](https://developers.douban.com/wiki/?title=browser)

[QQ OAuth2 API（Implicit）](http://wiki.connect.qq.com/%e4%bd%bf%e7%94%a8implicit_grant%e6%96%b9%e5%bc%8f%e8%8e%b7%e5%8f%96access_token)

[微信公众号获取access\_token（Client Credentials Grant）](https://mp.weixin.qq.com/wiki?id=mp1421140183&t=0.2731444596120334)。

至于Resource Owner Password Credentials Grant这种类型的许可方式，由于其适用常见，平时作为第三方开发者的开发工作中，没有遇到此类的案例。其适用场景在于第三方应用和Resoure server属于同一方这样高度可信的环境下。

作者：[Blackheart](http://linianhui.cnblogs.com/)

出处：[http://linianhui.cnblogs.com](http://linianhui.cnblogs.com/)

本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

# 10 OAuth2 扩展：Multiple Response Types

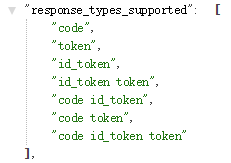
在本系列的第一篇博客[[认证授权] 1.OAuth2授权](http://www.cnblogs.com/linianhui/p/oauth2-authorization.html)中解释OAuth2的授权请求的时候，其请求参数中有一个 response\_type 的参数，其允许的值有 code 和 token 两个，在这两个的基础上，OIDC增加了一个新值 id\_token （详细信息定义在<http://openid.net/specs/oauth-v2-multiple-response-types-1_0.html>）：

code：oauth2定义的。用于获取authorization\_code。

token：oauth2定义的。用户获取access\_token。

id\_token：OIDC定义的。用户获取id\_token。

至此OIDC是支持三种类型的response\_type的，不但如此，OIDC还允许了可以组合这三种类型，即在一个response\_type中包含多个值（空格分隔）。比如当参数是这样的时候 response\_type=id\_token token ，OIDC服务就会把access\_token和id\_token一并给到调用方。OIDC对这些类型的支持情况体现在上面提到的Discovery服务中返回的**response\_types\_supported字段**中：



# 阮一峰理解OAuth 2.0 博客

作者： [阮一峰](http://www.ruanyifeng.com/)

日期： [2014年5月12日](http://www.ruanyifeng.com/blog/2014/05/)

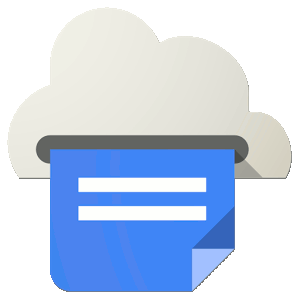
[OAuth](http://en.wikipedia.org/wiki/OAuth)是一个关于授权（authorization）的开放网络标准，在全世界得到广泛应用，目前的版本是2.0版。

本文对OAuth 2.0的设计思路和运行流程，做一个简明通俗的解释，主要参考材料为[RFC 6749](http://www.rfcreader.com/#rfc6749)。

## 一、应用场景

为了理解OAuth的适用场合，让我举一个假设的例子。

有一个"云冲印"的网站，可以将用户储存在Google的照片，冲印出来。用户为了使用该服务，必须让"云冲印"读取自己储存在Google上的照片。



问题是只有得到用户的授权，Google才会同意"云冲印"读取这些照片。那么，"云冲印"怎样获得用户的授权呢？

传统方法是，用户将自己的Google用户名和密码，告诉"云冲印"，后者就可以读取用户的照片了。这样的做法有以下几个严重的缺点。

（1）"云冲印"为了后续的服务，会保存用户的密码，这样很不安全。

（2）Google不得不部署密码登录，而我们知道，单纯的密码登录并不安全。

（3）"云冲印"拥有了获取用户储存在Google所有资料的权力，用户没法限制"云冲印"获得授权的范围和有效期。

（4）用户只有修改密码，才能收回赋予"云冲印"的权力。但是这样做，会使得其他所有获得用户授权的第三方应用程序全部失效。

（5）只要有一个第三方应用程序被破解，就会导致用户密码泄漏，以及所有被密码保护的数据泄漏。

OAuth就是为了解决上面这些问题而诞生的。

## 二、名词定义

在详细讲解OAuth 2.0之前，需要了解几个专用名词。它们对读懂后面的讲解，尤其是几张图，至关重要。

（1） **Third-party application**：第三方应用程序，本文中又称"客户端"（client），即上一节例子中的"云冲印"。

（2）**HTTP service**：HTTP服务提供商，本文中简称"服务提供商"，即上一节例子中的Google。

（3）**Resource Owner**：资源所有者，本文中又称"用户"（user）。

（4）**User Agent**：用户代理，本文中就是指浏览器。

（5）**Authorization server**：认证服务器，即服务提供商专门用来处理认证的服务器。

（6）**Resource server**：资源服务器，即服务提供商存放用户生成的资源的服务器。它与认证服务器，可以是同一台服务器，也可以是不同的服务器。

知道了上面这些名词，就不难理解，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。

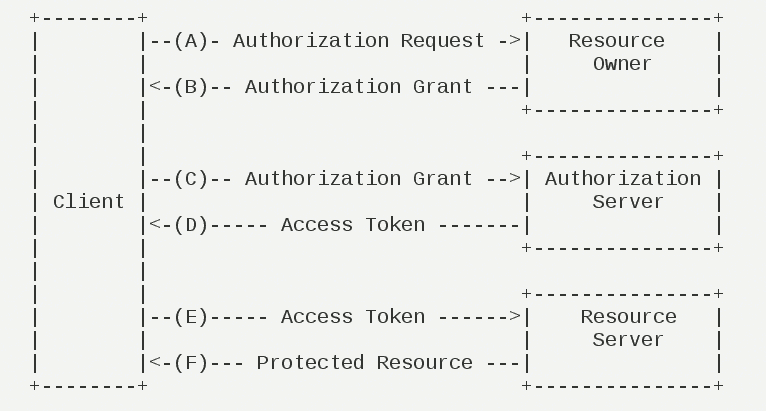
## 三、OAuth的思路

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。

## 四、运行流程

OAuth 2.0的运行流程如下图，摘自RFC 6749。



（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。

下面一一讲解客户端获取授权的四种模式。

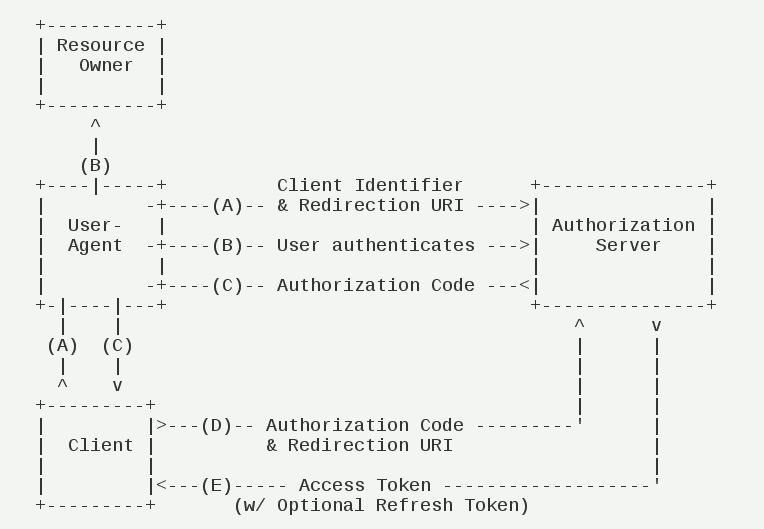
## 五、客户端的授权模式

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。

* 授权码模式（authorization code）
* 简化模式（implicit）
* 密码模式（resource owner password credentials）
* 客户端模式（client credentials）

## 六、授权码模式

授权码模式（authorization code）是功能最完整、流程最严密的授权模式。它的特点就是通过客户端的后台服务器，与"服务提供商"的认证服务器进行互动。



它的步骤如下：

（A）用户访问客户端，后者将前者导向认证服务器。

（B）用户选择是否给予客户端授权。

（C）假设用户给予授权，认证服务器将用户导向客户端事先指定的"重定向URI"（redirection URI），同时附上一个授权码。

（D）客户端收到授权码，附上早先的"重定向URI"，向认证服务器申请令牌。这一步是在客户端的后台的服务器上完成的，对用户不可见。

（E）认证服务器核对了授权码和重定向URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。

下面是上面这些步骤所需要的参数。

A步骤中，客户端申请认证的URI，包含以下参数：

* response\_type：表示授权类型，必选项，此处的值固定为"code"
* client\_id：表示客户端的ID，必选项
* redirect\_uri：表示重定向URI，可选项
* scope：表示申请的权限范围，可选项
* state：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

下面是一个例子。

GET /authorize?response\_type=code&client\_id=s6BhdRkqt3&state=xyz

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1

Host: server.example.com

C步骤中，服务器回应客户端的URI，包含以下参数：

* code：表示授权码，必选项。该码的有效期应该很短，通常设为10分钟，客户端只能使用该码一次，否则会被授权服务器拒绝。该码与客户端ID和重定向URI，是一一对应关系。
* state：如果客户端的请求中包含这个参数，认证服务器的回应也必须一模一样包含这个参数。

下面是一个例子。

HTTP/1.1 302 Found

Location: <https://client.example.com/cb>?code=SplxlOBeZQQYbYS6WxSbIA

&state=xyz

D步骤中，客户端向认证服务器申请令牌的HTTP请求，包含以下参数：

* grant\_type：表示使用的授权模式，必选项，此处的值固定为"authorization\_code"。
* code：表示上一步获得的授权码，必选项。
* redirect\_uri：表示重定向URI，必选项，且必须与A步骤中的该参数值保持一致。
* client\_id：表示客户端ID，必选项。

下面是一个例子。

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

E步骤中，认证服务器发送的HTTP回复，包含以下参数：

* access\_token：表示访问令牌，必选项。
* token\_type：表示令牌类型，该值大小写不敏感，必选项，可以是bearer类型或mac类型。
* expires\_in：表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。
* refresh\_token：表示更新令牌，用来获取下一次的访问令牌，可选项。
* scope：表示权限范围，如果与客户端申请的范围一致，此项可省略。

下面是一个例子。

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",

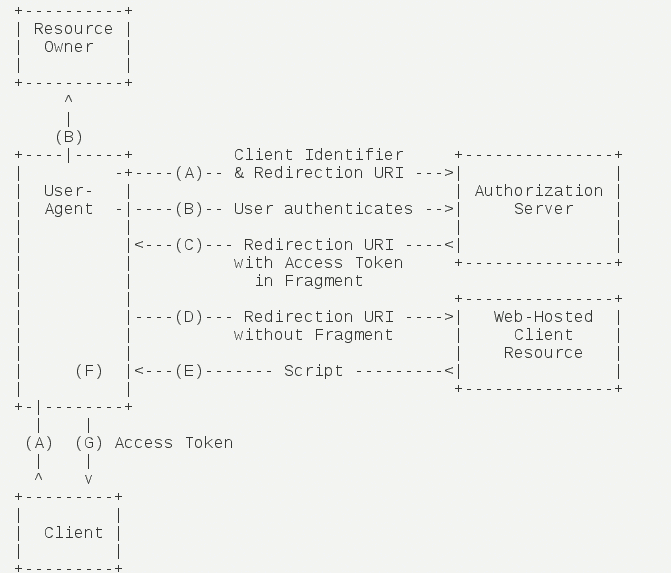
"example\_parameter":"example\_value"

}

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。

## 七、简化模式

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。



它的步骤如下：

（A）客户端将用户导向认证服务器。

（B）用户决定是否给于客户端授权。

（C）假设用户给予授权，认证服务器将用户导向客户端指定的"重定向URI"，并在URI的Hash部分包含了访问令牌。

（D）浏览器向资源服务器发出请求，其中不包括上一步收到的Hash值。

（E）资源服务器返回一个网页，其中包含的代码可以获取Hash值中的令牌。

（F）浏览器执行上一步获得的脚本，提取出令牌。

（G）浏览器将令牌发给客户端。

下面是上面这些步骤所需要的参数。

A步骤中，客户端发出的HTTP请求，包含以下参数：

* response\_type：表示授权类型，此处的值固定为"token"，必选项。
* client\_id：表示客户端的ID，必选项。
* redirect\_uri：表示重定向的URI，可选项。
* scope：表示权限范围，可选项。
* state：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

下面是一个例子。

GET /authorize?response\_type=token&client\_id=s6BhdRkqt3&state=xyz

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1

Host: server.example.com

C步骤中，认证服务器回应客户端的URI，包含以下参数：

* access\_token：表示访问令牌，必选项。
* token\_type：表示令牌类型，该值大小写不敏感，必选项。
* expires\_in：表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。
* scope：表示权限范围，如果与客户端申请的范围一致，此项可省略。
* state：如果客户端的请求中包含这个参数，认证服务器的回应也必须一模一样包含这个参数。

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

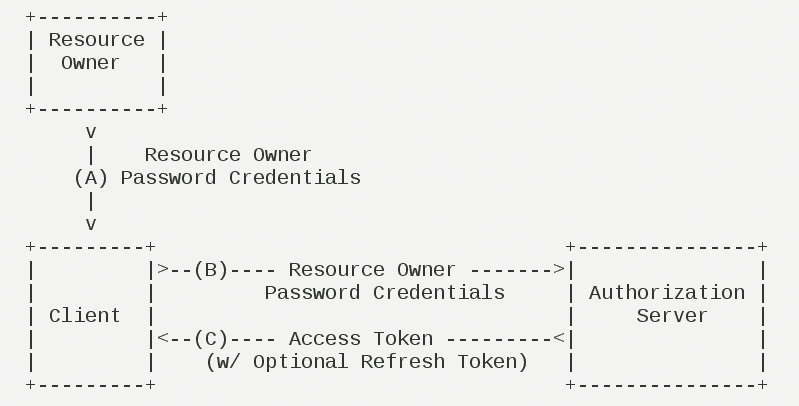
在上面的例子中，认证服务器用HTTP头信息的Location栏，指定浏览器重定向的网址。注意，在这个网址的Hash部分包含了令牌。

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。

## 八、密码模式

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。



它的步骤如下：

（A）用户向客户端提供用户名和密码。

（B）客户端将用户名和密码发给认证服务器，向后者请求令牌。

（C）认证服务器确认无误后，向客户端提供访问令牌。

B步骤中，客户端发出的HTTP请求，包含以下参数：

* grant\_type：表示授权类型，此处的值固定为"password"，必选项。
* username：表示用户名，必选项。
* password：表示用户的密码，必选项。
* scope：表示权限范围，可选项。

下面是一个例子。

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

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

grant\_type=password&username=johndoe&password=A3ddj3w

C步骤中，认证服务器向客户端发送访问令牌，下面是一个例子。

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"

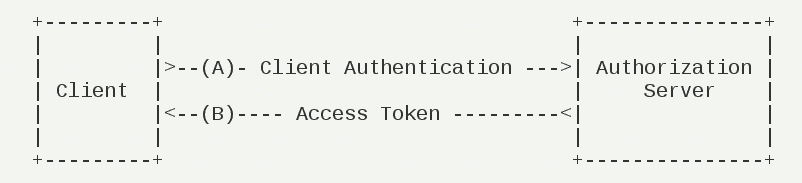
}

上面代码中，各个参数的含义参见《授权码模式》一节。

整个过程中，客户端不得保存用户的密码。

## 九、客户端模式

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。



它的步骤如下：

（A）客户端向认证服务器进行身份认证，并要求一个访问令牌。

（B）认证服务器确认无误后，向客户端提供访问令牌。

A步骤中，客户端发出的HTTP请求，包含以下参数：

* granttype：表示授权类型，此处的值固定为"clientcredentials"，必选项。
* scope：表示权限范围，可选项。

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

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

grant\_type=client\_credentials

认证服务器必须以某种方式，验证客户端身份。

B步骤中，认证服务器向客户端发送访问令牌，下面是一个例子。

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,

"example\_parameter":"example\_value"

}

上面代码中，各个参数的含义参见《授权码模式》一节。

## 十、更新令牌

如果用户访问的时候，客户端的"访问令牌"已经过期，则需要使用"更新令牌"申请一个新的访问令牌。

客户端发出更新令牌的HTTP请求，包含以下参数：

* granttype：表示使用的授权模式，此处的值固定为"refreshtoken"，必选项。
* refresh\_token：表示早前收到的更新令牌，必选项。
* scope：表示申请的授权范围，不可以超出上一次申请的范围，如果省略该参数，则表示与上一次一致。

下面是一个例子。

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

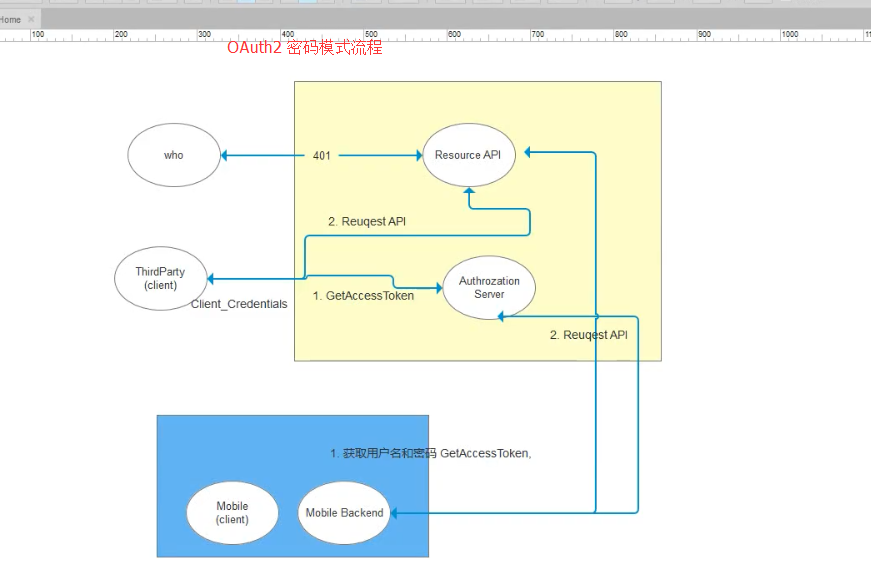
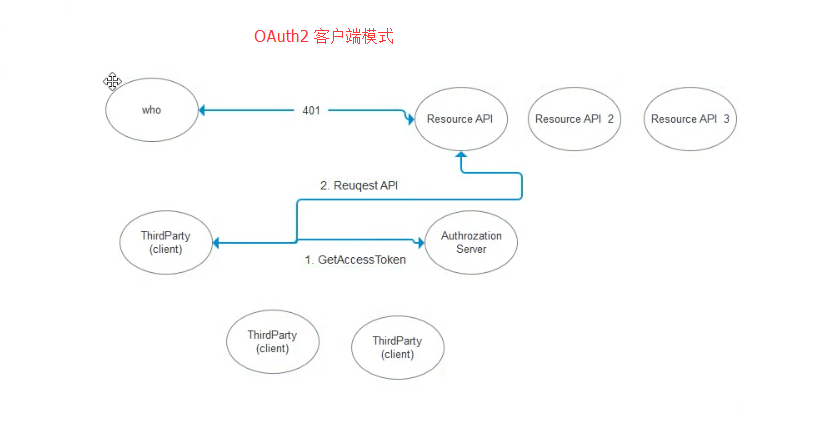
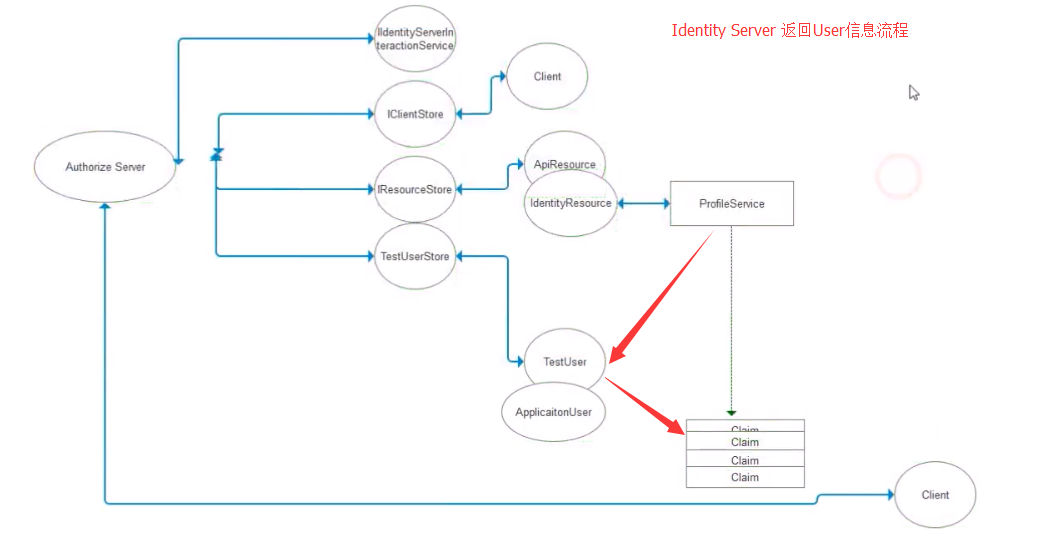
Content-Type: application/x-www-form-urlencoded

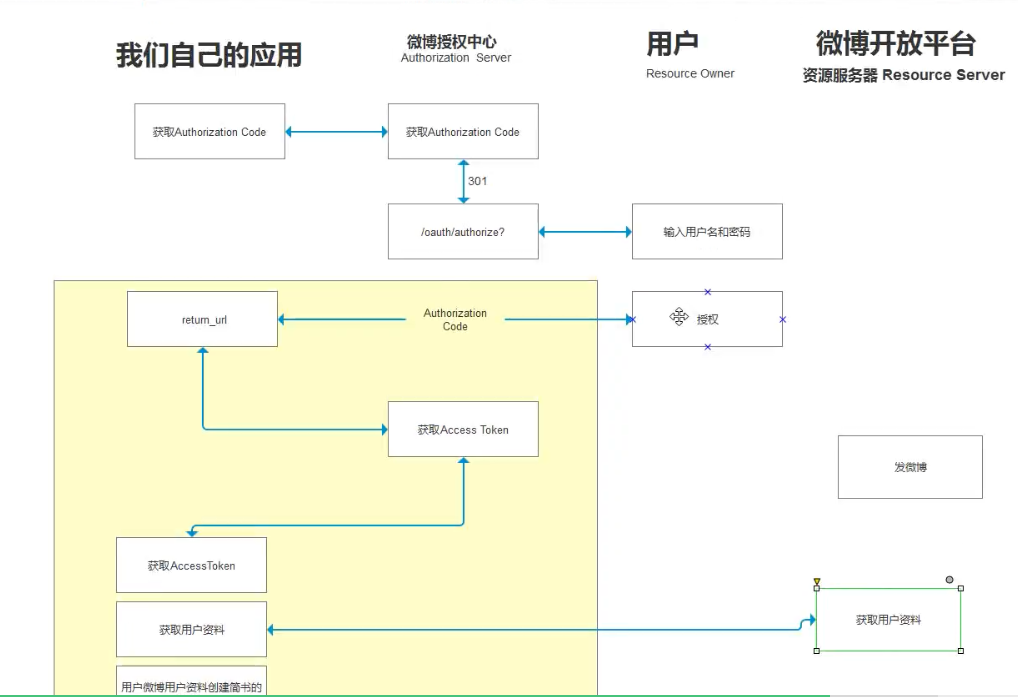
grant\_type=refresh\_token&refresh\_token=tGzv3JOkF0XG5Qx2TlKWIA

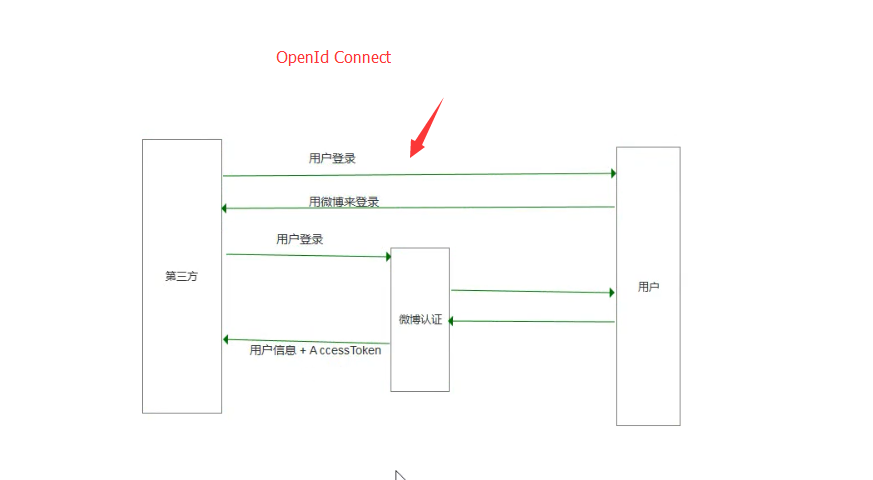
## 十一、博客地址

http://www.ruanyifeng.com/blog/2014/05/oauth\_2\_0.html

# jesse老师讲解视频截图





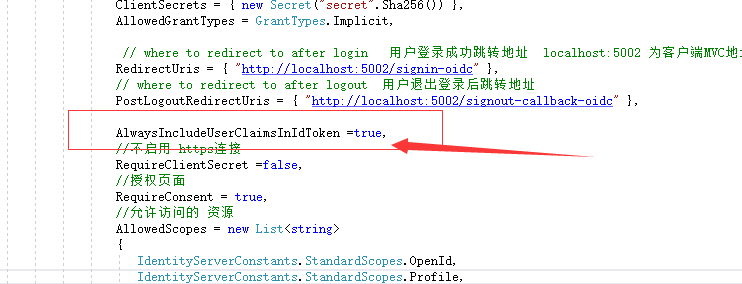


# Id token Access\_Token中附加UserClaims

## 1 第一种方式

在IdentityServer Client中配置 AlwaysIncludeUserClaimsInIdToken = rue

在Id\_token 中包含用户信息（profileService赋值的内容）



这种方式会在ID\_TOKEN ,ACCESS\_TOKEN中附加UserClaims

## 2 第二种方式

客户端通过配置GetClaimsFromUserInfoEndpoint =true ,返回的数据必须绑定，才能在id\_token 中拿到。该方法实际是调用使用Access\_Token调用 userinfo Endpoint ，获取到数据，并通过绑定返回到Access\_Token



## 3 手动通过 Access\_token 请求 Userinfo Endpoint 从profileService 获取UserClaims。

# jesse---我们在《ASP.NET Core项目实战的课程》第一章里面给identity server4做了一个全面的介绍和示例的练习 。

http://www.jessetalk.cn/2018/04/04/oidc-asp-net-core/

如果想完全理解本文所涉及到的话题，你需要了解的背景知识有：

* 什么是OpenId Connect (OIDC)
* OIDC 对oAuth进行了哪些扩展？
* Identity Server4提供的OIDC认证服务（服务端）
* ASP.NET Core的权限体系中的OIDC认证框架（客户端）

### 什么是 OIDC

在了解OIDC之前，我们先看一个很常见的场景。假使我们现在有一个网站要集成微信或者新浪微博的登录，两者现在依然采用的是oAuth 2.0的协议来实现 。 关于微信和新浪微博的登录大家可以去看看它们的开发文档。

在我们的网站集成微博或者新浪微博的过程大致是分为五步：

1. 准备工作：在微信/新浪微博开发平台注册一个应用，得到AppId和AppSecret
2. 发起 oAauth2.0 中的 Authorization Code流程请求Code
3. 根据Code再请求AccessToken（通常在我们应用的后端完成，用户不可见）
4. 根据 AccessToken 访问微信/新浪微博的某一个API，来获取用户的信息
5. 后置工作：根据用户信息来判断是否之前登录过？如果没有则创建一个用户并将这个用户作为当前用户登录（我们自己应用的登录逻辑，比如生成jwt），如果有了则用之前的用户登录。

中间第2到3的步骤为标准的oAuth2 授权码模式的流程，如果不理解的可以参考阮一峰所写的《[理解oAuth2.0](http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html)》一文。我们主要来看第4和5步，对于第三方应用要集成微博登录这个场景来说最重要的是我希望能快速拿到用户的一些基本信息（免去用户再次输入的麻烦）然后根据这些信息来生成一个我自己的用户跟微博的用户Id绑定（为的是下次你使用微博登录的时候我还能把你再找出来）。

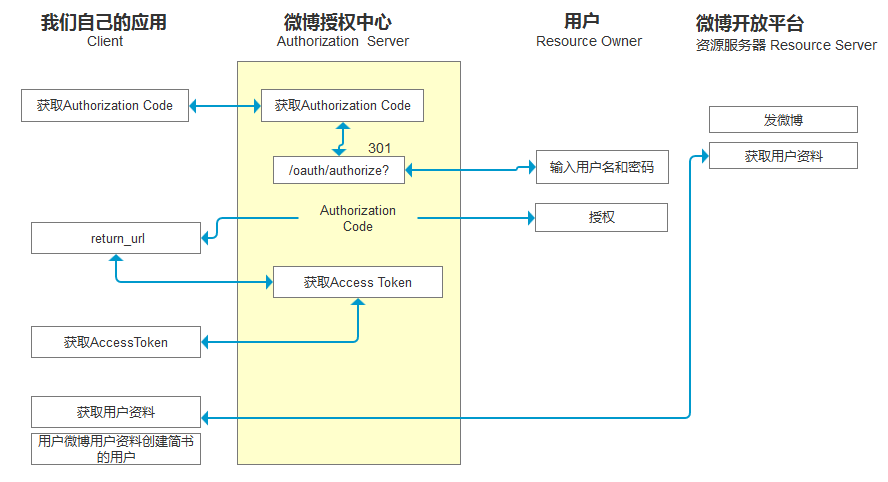
oAuth在这里麻烦的地方是我还需要再请求一次API去获取用户数据，注意这个API和登录流程是不相干的，其实是属于微博开放平台丛多API中的一个，包括微信开放平台也是这样来实现。这里有个问题是前面的 2和3是oAuth2的标准化流程，而第4步却不是，但是大家都这么干（它是一个大家都默许的标准）

于是大家干脆就建立了一套标准协议并进行了一些优化，它叫OIDC

*OIDC 建立在oAuth2.0协议之上，允许客户端(Clients)通过一个授权服务(Authorization Server)来完成对用户认证的过程，并且可以得到用户的一些基本信息包含在JWT中。*

# OIDC对oAuth进行了哪些扩展？

在oAuth2.0授权码模式的帮助下，我们拿到了用户信息。

[](https://github.com/jessetalk/aspnet-core-in-practise/blob/master/assets/authroization_code_flow)以上没有认证的过程，只是给我们的应用授权访问一个API的权限，我们通过这个API去获取当前用户的信息，这些都是通过oAuth2的授权码模式完成的。 我们来看看oAuth2 授权码模式的流程：

第一步，我们向authorize endpoint请求code的时候所传递的response\_type表示授权类型，原来只有固定值code

GET **/connect/**authorize?response\_type=code&client\_id=postman&state=xyz&scope=api1

&redirect\_uri=http://localhost:5001/oauth2/callback

第二步，上面的请求执行完成之后会返回301跳转至我们传过去的redirect\_uri并带上code

https://localhost:5001/oauth2/callback?code=835d584d4bc96d46ce49e27ebdbf272e40234d5f31097f63163f17da61fcd01c

&scope=api1

&state=111271607

第三步，用code换取access token

POST **/connect/**token?grant\_type=authorization\_code&code=835d584d4bc96d46ce49e27ebdbf272e40234d5f31097f63163f17da61fcd01c

&redirect\_uri=http://localhost:5001/oauth2/callback

&client\_id=postman

&client\_secret=secret

通过这个POST我们就可以得到access\_token

{

"access\_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjV",

"expires\_in": 3600,

"token\_type": "Bearer"

}

我们拿到access\_token之后，再把access\_token放到authorization头请求 api来获取用户的信息。在这里，这个api不是属于授权服务器提供的，而是属于资源服务器。

OIDC给oAuth2进行扩展之后就填补了这个空白，让我们可以授权它添加了以下两个内容：

* response\_type 添加IdToken
* 添加userinfo endpoint，用idToken可以获取用户信息

OIDC对它进行了扩展，现在你有三个选择：code, id\_token和 token，现在我们可以这样组合来使用。

| “response\_type” value | Flow |
| --- | --- |
| code | Authorization Code Flow |
| id\_token | Implicit Flow |
| id\_token token | Implicit Flow |
| code id\_token | Hybrid Flow |
| code token | Hybrid Flow |
| code id\_token token | Hybrid Flow |

我们简单的来理解一下这三种模式：

* Authorization Code Flow授权码模式：保留oAuth2下的授权模式不变response\_type=code
* Implicit Flow 隐式模式：在oAuth2下也有这个模式，主要用于客户端直接可以向授权服务器获取token，跳过中间获取code用code换accesstoken的这一步。在OIDC下，responsetype=token idtoken，也就是可以同时返回access\_token和id\_token。
* Hybrid Flow 混合模式： 比较有典型的地方是从authorize endpoint 获取 code idtoken，这个时候id\_token可以当成认证。而可以继续用code获取access\_token去做授权，比隐式模式更安全。再来详细看一下这三种模式的差异：

| Property | Authorization Code Flow | Implicit Flow | Hybrid Flow |
| --- | --- | --- | --- |
| access token和id token都通过Authorization endpoint返回 | no | yes | no |
| 两个token都通过token end point 返回 | yes | no | no |
| 用户使用的端(浏览器或者手机）无法查看token | yes | no | no |
| Client can be authenticated | yes | no | yes |
| 支持刷新token | yes | no | yes |
| 不需要后端参与 | no | yes | no |
|  |  |  |  |

我们来看一下通过Hybird如何获取 code、id\_token、\_以及access\_token，然后再用id\_token向userinfo endpoint请求用户信息。

第一步：获取code，

* response\_type=code id\_token
* scope=api1 openid profile 其中openid即为用户的唯一识别号

GET **/connect/**authorize?response\_type=code id\_token&client\_id=postman&state=xyz&scope=api1 openid profile

&nonce=7362CAEA-9CA5-4B43-9BA3-34D7C303EBA7

&redirect\_uri=http://localhost:5001/oauth2/callback

当我们使用OIDC的时候，我们请求里面多了一个nonce的参数，与state有异曲同工之妙。我们给它一个guid值即可。

第二步：我们的redirect\_uri在接收的时候即可以拿到code 和 id\_token

https://localhost:5001/oauth2/callback#

code=c5eaaaca8d4538f69f670a900d7a4fa1d1300b26ec67fba2f84129f0ab4ffa35

&id\_token=eyJhbGciOiJSUzI1NiIsImtpZCI6IjVjMzA5ZGIwYTE2OGEwOTgGtpbj0GVXNnkKhGdrzA

&scope=openid%20profile%20api1&state=111271607

第三步：用code换access\_token(这一步与oAuth2中的授权码模式一致）

第四步：用access\_token向userinfo endpoint获取用户资料

**Get** http://localhost:5000/connect/userinfo

Authorization Bearer access\_token

返回的用户信息

{

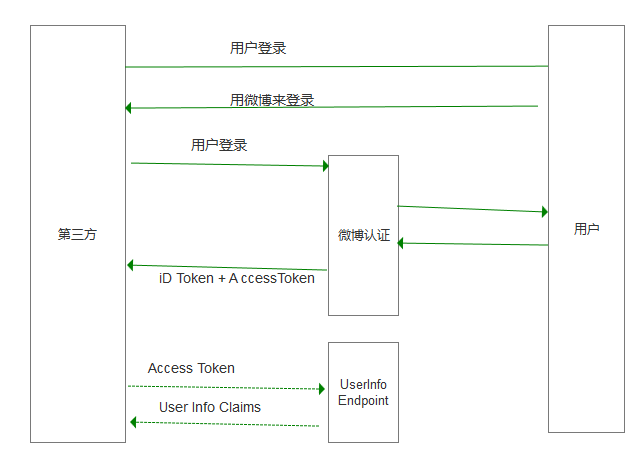
"name": "scott",

"family\_name": "liu",

"sub": "5BE86359-073C-434B-AD2D-A3932222DABE"

}

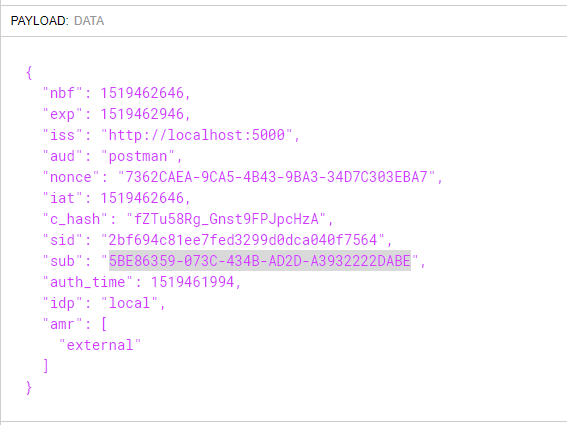
以下是我们的流程示意图。

[](https://github.com/jessetalk/aspnet-core-in-practise/blob/master/assets/oidc_userinfo_get)

有人可能会注意到，在这里我们拿到的idtoken没有派上用场，我们的用户资料还是通过access\_token从userinfo endpoint里拿的。这里有两个区别：

1. userinfo endpoint是属于认证服务器实现的，并非资源服务器，有归属的区别
2. id\_token 是一个jwt，里面带有用户的唯一标识，我们在判断该用户已经存在的时候不需要再请求userinfo endpoint

下图是对id\_token进行解析得到的信息：sub即subject\_id(用户唯一标识 )

[](https://github.com/jessetalk/aspnet-core-in-practise/blob/master/assets/id_token_jwt)

对jwt了解的同学知道它里面本身就可以存储用户的信息，那么id\_token可以吗？答案当然是可以的，我们将在介绍完identity server4的集成之后最后来实现。

# Identity Server4提供的OIDC认证服务

Identity Server4是asp.net core2.0实现的一套oAuth2 和OIDC框架，用它我们可以很快速的搭建一套自己的认证和授权服务。我们来看一下用它如何快速实现OIDC认证服务。

由于用户登录代码过多，完整代码可以加入ASP.NET Core QQ群 92436737获取。 此处仅展示配置核心代码。

过程

* 新建asp.net core web应用程序
* 添加identityserver4 nuget引用
* 依赖注入初始化

**services**.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddInMemoryIdentityResources(**Config**.GetIdentityResources())

.AddInMemoryApiResources(**Config**.GetApiResources())

.AddInMemoryClients(**Config**.GetClients())

.AddTestUsers(**Config**.GetTestUsers());

* 中间件添加

**app**.UseIdentityServer();

* 配置

在测试的时候我们新建一个Config.cs来放一些配置信息

api resources

**public** **static** IEnumerable<ApiResource> **GetApiResources**()

{

**return** **new** List<ApiResource>

{

**new** ApiResource("api1", "API Application"){

UserClaims = { "role", JwtClaimTypes.Role }

}

};

}

identity resources

**public** **static** IEnumerable<IdentityResource> **GetIdentityResources**()

{

**return** **new** List<IdentityResource> {

**new** IdentityResources.OpenId(),

**new** IdentityResources.Profile(),

**new** IdentityResources.Email(),

};

}

clients

我们要讲的关键信息在这里，client有一个AllowGrantTypes它是一个string的集合。我们要写进去的值就是我们在上一节讲三种模式: Code，Implict和Hybird。因为这三种模式决定了我们的response\_type可以请求哪几个值，所以这个地方一定不能写错。

IdentityServer4.Models.GrantTypes这个枚举给我们提供了一些选项，实际上是把oAuth的4种和OIDC的3种进行了组保。

**public** **static** IEnumerable<Client> **GetClients**()

{

**return** **new** List<Client>

{

**new** Client

{

ClientId = "postman",

AllowedGrantTypes = GrantTypes.Hybird,

RedirectUris = { "https://localhost:5001/oauth2/callback" },

ClientSecrets =

{

**new** Secret("secret".Sha256())

},

AllowedScopes = **new** List<**string**>

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile,

"api1"

},

AllowOfflineAccess=**true**,

},

};

}

users

**public** **static** **List**<TestUser> GetTestUsers()

{

**return** **new** **List**<TestUser> {

**new** TestUser {

SubjectId = "5BE86359-073C-434B-AD2D-A3932222DABE",

Username = "scott",

Password = "password",

Claims = **new** **List**<Claim> {

**new** Claim(JwtClaimTypes.Name, "scott"),

**new** Claim(JwtClaimTypes.FamilyName, "liu"),

**new** Claim(JwtClaimTypes.Email, "scott@scottbrady91.com"),

**new** Claim(JwtClaimTypes.Role, "user"),

}

}

};

}

# ASP.NET Core的权限体系中的OIDC认证框架

在Microsoft.AspNetCore.All nuget引用中包含了Microsoft.AspNetCore.Authentication.OpenIdConnect即asp.net core OIDC的客户端。我们需要在依赖注入中添加以下配置：

services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("Cookies")

.AddOpenIdConnect("oidc", options =>

{

options.SignInScheme = "Cookies";

options.Authority = "http://localhost:5000";

options.RequireHttpsMetadata = **false**;

options.ClientId = "postman";

options.ClientSecret = "secret";

options.ResponseType = "code id\_token";

options.GetClaimsFromUserInfoEndpoint = **true**;

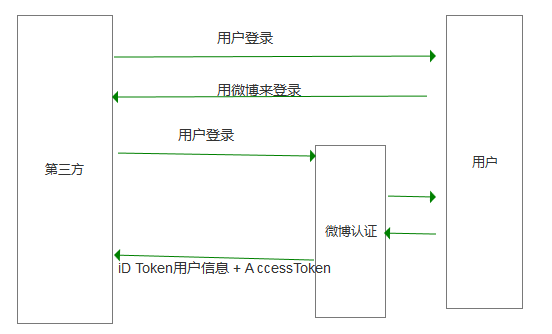
options.Scope.Add("api1");

options.Scope.Add("offline\_access");

});

Authority即我们的用identity server4搭建的认证授权服务器，而其中的GetClaimsFromUserInfoEndpoint则会在拿到id\_token之后自动向userinfo endpoint请求用户信息并放到asp.net core的User Identity下。

我们上面讲过，可以不需要请求userinfo endpoint, 直接将用户信息放到id\_token中。

[](https://github.com/jessetalk/aspnet-core-in-practise/blob/master/assets/userinfo_in_idtoken)

这样我们就不需要再向userinfo endpoint发起请求，从id\_token中即可以获取到用户的信息。而有了identity server4的帮助，完成这一步只需要一句简单的配置即可：

**new** Client

{

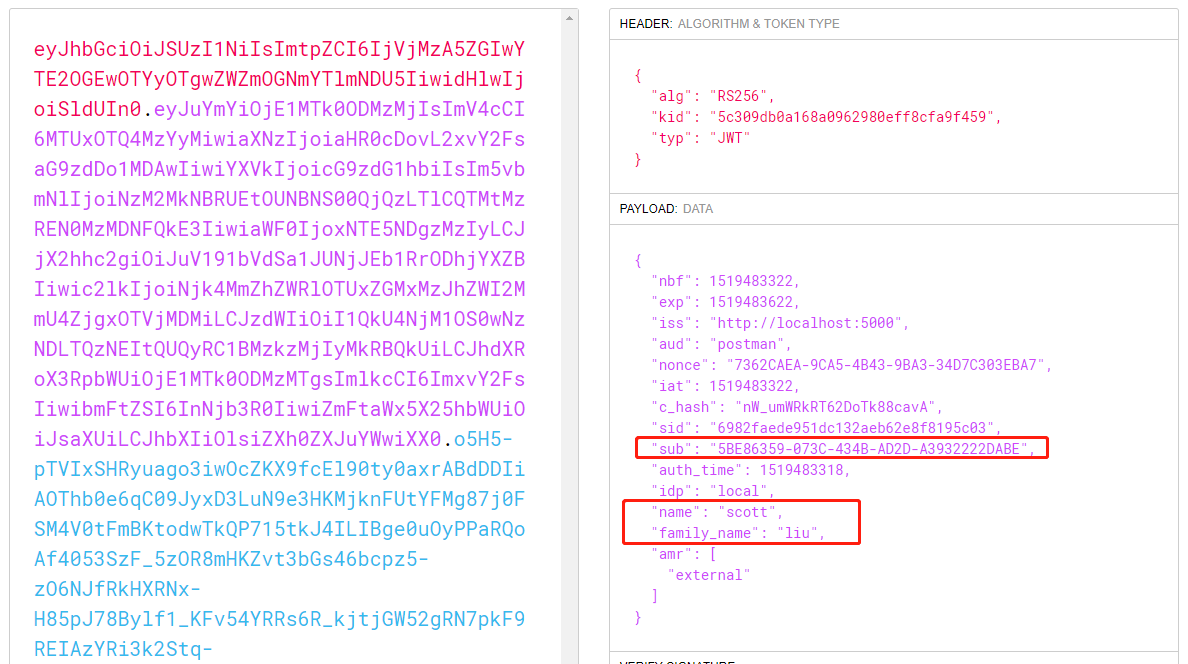
**ClientId** = "postman",

AlwaysIncludeUserClaimsInIdToken = true,

AllowOfflineAccess=true,

}

这样我们在拿到id\_token之后，里即包含了我们的用户信息。

[](https://github.com/jessetalk/aspnet-core-in-practise/blob/master/assets/user_claims_in_id_token)