OAuth 2.0 协议原理

OAuth 2.0协议是一种三方授权协议，目前大部分的第三方登录与授权都是基于该协议的标准或改进实现。OAuth 1.0的标准在2007年发布，OAuth 2.0的标准则在2011年发布，其中2.0的标准取消所有Token的加密过程，并简化了授权流程，但因强制使用HTTPS协议，被认为安全性高于1.0的标准。

一. 基础应用：第三方登录

对于OAuth2.0协议的第一次接触，我相信大部分开发者都是通过对接第三方登录才开始知道和了解该协议。的确，OAuth2.0协议被广泛应用于第三方授权登录中，借助第三方登录可以让用户免于再次注册之苦，支持第三方登录也对这些网站、APP起到了积极的作用，免去了复杂的注册过程，用户体验更佳，更愿意去登录。这样在提高留存率的同时，也更加易于收集用户的一些非敏感信息等，另外还可以借助一些社交类的第三方账号进行站点推广等。

账号服务对于一个公司来说是一个基础类服务，既简单也复杂。说它简单，是因为账号的主要业务就是注册和登录，相信很多人在初次接触WEB开发的时候，第一个作业就是实现一个用户注册和登录的流程；说它复杂，是因为账号服务往往是一个公司开展其它业务的基础，必须是公司业务中QPS最高的业务之一，需具备高可用、低延迟等特点，因为涉及到用户的敏感信息，还需要在安全方面下足功夫。所以对于一个规模不大的公司来说，将主要人力投入在建立自己的账号业务上是一件性价比很低的事情，这个时候接入大公司的第三方账户登录，应该是更加可取的一种选择。

二. OAuth2.0协议的基本定义与授权流程

作为第三方登录服务提供方，我们的核心矛盾点就是 **既要让用户在APP上登录，同时还不能让该APP拿到用户的登录凭证**。解决这一矛盾的利器就是token，而OAuth2.0协议的最终目的就是给第三方应用下发token，它记录了用户的登录或授权状态，通过将token传递给第三方应用，既能让第三方应用登录并拿到用户许可数据，也可以将用户的凭证牢牢拽在自己的手里。

说到用户登录状态的记录，我们可能最先想到的是session机制，想想你在做的第一个用户登录应用的时候，是不是拿服务器的session去记录用户是否登录。这一做法简单，但是也存在问题，session说到底也还是缓存，当用户量较大的时候，需要相当大容量的缓存才能够容纳所有用户的登录状态，并且我们的WEB服务器往往有多台，通过负载均衡机制来提升服务的可用性，这样的场景下，我们不能简单的通过本地session来记录用户的登录状态，必须有专门的session服务器，或者其它的一些session复制措施，还需要考虑宕机造成的session丢失等问题，总之用户量大了，许多最初不是问题的问题逐渐暴露出来，有的甚至可能是极其棘手的。实际上对于用户登录状态的保存，我们可以走token机制，让客户端自己去保存用户的登录状态，将服务器从繁重的压力中解脱出来，利用SSO来实现公司内各业务之间“一次登录，到处可用”。

回到OAuth2.0协议，上面的论述可能侧重了第三方登录，实际上登录只是一个授权的过程，对于一个应用，其最终目的还是希望能够拿到用户存储在资源服务器上的用户数据，所以登录授权还只是第一步，后续APP还需要携带token去资源服务器请求用户数据，这个时候是一个鉴权的过程，OAuth2.0协议的主要目的在于授权，至于鉴权，实现上主要还是对APP传递过来的token进行解析和验证，这一块相对要简单一些，所以下面主要讲解OAuth2.0 授权的流程。

2.1 OAuth2.0定义的5种角色

- **客户端（Client）**

客户端是OAuth服务的接入方，其目的是请求用户存储在资源服务器上的受保护资源，客户端可以移动应用、网页应用等等。

- **用户代理（User Agent）**

用户代理是用户参与互联网的工具，一般可以理解为浏览器。

- **资源所有者（Resource Owner）**

受保护资源所属的实体，比如资源的持有人等，下文的用户即资源所有者。

- **授权服务器（Authorization Server）**

授权服务器的主要职责是验证资源所有者的身份，并依据资源所有者的许可对第三方应用下发令牌。

- **资源服务器（Resource Server）**

托管资源的服务器，能够接收和响应持有令牌的资源访问请求，可以与授权服务器是同一台服务器，当然也是可以分开的。

2.2 基本概念

2.2.1 访问令牌（access token）

访问令牌是在用户授权许可下，授权服务器下发给客户端的一个授权凭证，该令牌所要表达的意思是“用户授予该APP在多长时间范围内允许访问哪些与自己相关的资源”，所以访问令牌主要在 **时间范围** 和 **权限范围** 两个维度进行控制，此外访问令牌对于客户端来说是非透明的，外在表现就是一个字符串，客户端无法知晓字符串背后所隐藏的用户信息，因此不用担心用户的登录凭证会因此而泄露。

OAuth2.0协议虽然最终以令牌的形式授权，却没有对令牌的具体生成策略和构成元素作过多的说明，这一块的实现给予OAuth2.0的实现者充分的自由。

2.2.2 刷新令牌（refresh token）

刷新令牌的作用在于更新访问令牌，访问令牌的有效期一般较短，这样可以保证在发生访问令牌泄露时，不至于造成太坏的影响，但是访问令牌有效期设置太短存在的副作用就是用户需要频繁授权，虽然可以通过一定的机制进行静默授权，但是频繁的调用授权接口，对于授权服务器也是一种压力，这种情况下就可以在下发访问令牌的同时下发一个刷新令牌，刷新令牌的有效期明显长于访问令牌，这样在访问令牌失效时，可以利用刷新令牌去授权服务器换取新的访问令牌，不过协议对于刷新令牌没有强制规定，是否需要该令牌是客户端可以自行选择。

2.2.3 回调地址（redirect uri）

OAuth2.0是一类基于回调的授权协议，在授权码模式中，整个授权需要分为两步进行，第一步下发授权码，第二步根据第一步拿到的授权码请求授权服务器下发访问令牌。OAuth2.0在第一步下发授权码时，是将授权码以参数的形式添加到回调地址后面，并以302跳转的形式进行下发，这样简化了客户端的操作，不需要再主动去触发一次请求，即可进入下一步流程。

回调请求的设计却存在一个很大的安全隐患，坏人如果在客户端请求过程中修改了对应的回调地址，并指向自己的服务器，那么坏人可以利用这种机制去拿到客户端的授权码，继而走后面的流程，最终拿到访问令牌，另外坏人可以利用该机制引导用户到一个恶意站点，继而对用户发起攻击。以上两点都是该机制对于用户所造成的安全威胁，对于授权服务器而言，也存在一定的危害，坏人可以利用该机制让授权服务器变成“请求发送器”，以授权服务器为代理请求目标地址，这样在消耗授权服务器性能的同时，也对目标地址服务器产生DDOS攻击。

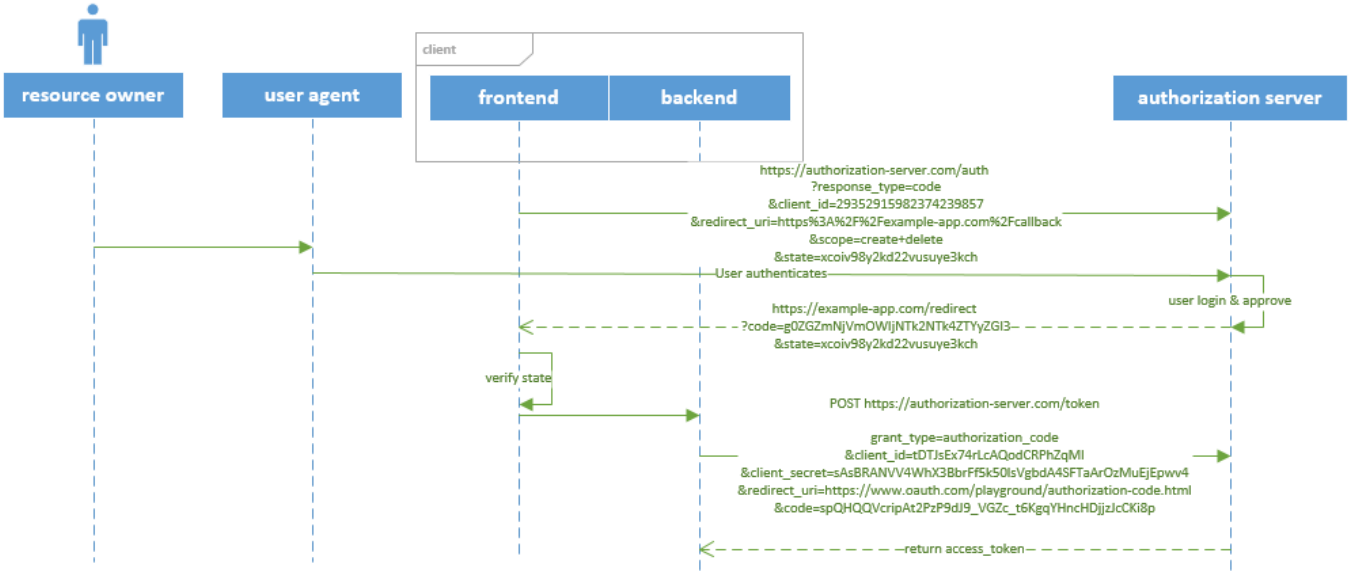
为了避免上述安全隐患，OAuth2.0协议强制要求客户端在注册时填写自己的回调地址，这个回调地址的目的是为了让回调请求能够到达客户端自己的服务器，从而可以走获取访问令牌的流程。客户端可以同时配置多个回调地址，并在请求授权时携带一个地址，服务器会验证客户端传递上来的回调地址是否与之前注册的回调地址相同，或者前者是后者集合的一个元素，只有在满足这一条件下才允许下发授权码，同时OAuth2.0协议还要求两步客户端请求携带的回调地址必须一致，通过这些措施来保证回调过程能够正常到达客户端自己的服务器，并继续后面拿授权码换取访问令牌的流程。

2.2.4 权限范围（scope）

访问令牌自带过期时间，可以在时间维度上对授权进行控制，而在范围维度上，OAuth2.0引入了一个scope的概念。scope可以看做是一个对象，包含一个权限的ID，名称，以及描述信息等，比如“获取您的基本资料（头像、昵称）”。应该在接入账号服务时必须向第三方登录服务提供方申请相应的scope，并在请求授权时指明该参数（否则表明获取该应用所允许的所有权限），这些权限在用户确认授权时，必须毫无保留的展示给用户，以让用户知道该APP需要获取用户的哪些数据或服务。

2.3 基本授权流程

OAuth2.0协议已定义了4种授权模式（简单通俗的理解就是：所谓授权模式，就是客户端获取access_token的方式），其中最具代表性的就是授权码模式，这个在3.1小节中详细介绍，这里先以该模式来简单感受一下OAuth2.0的授权流程，授权流程图如下：



假设整个流程开始之前，用户已经登录，那么整个授权流程如下：

1. 客户端请求授权服务器授权
2. 授权服务的授权端点重定向用户至授权交互页面，并询问用户是否授权
3. 如果用户许可，则授权端点验证客户端的身份，并发放授权码给客户端
4. 客户端拿到授权码之后，携带授权码请求授权服务器的令牌端点下发访问令牌
5. 令牌端点验证客户端的身份和授权码，通过则下发访问令牌和刷新令牌（可选）
6. 客户端拿到访问令牌后，携带访问令牌请求资源服务器上的受保护资源
7. 资源服务器验证客户端身份和访问令牌，通过则响应受保护资源访问请求

整个流程中，客户端都无法接触到用户的登录凭证信息，客户端通过访问令牌请求受保护资源，用户可以通过对授权操作的控制来间接控制客户端对于受保护资源的访问权限范围和时效。

三. 四种授权模式

OAuth2.0相对于1.0版本在授权模式上做了更多的细化，已定义的授权模式分为四种：

1）授权码模式（Authorization Code Grant）；

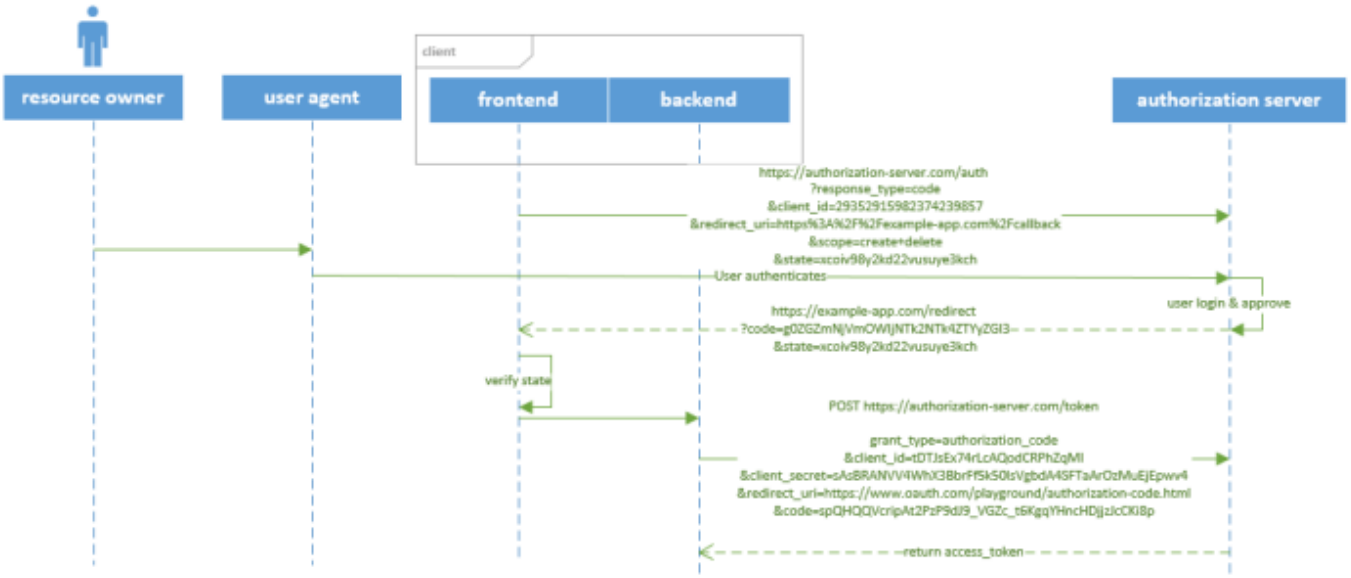
2）隐式授权模式（Implicit Grant）；

3）客户端凭证模式（Client Credentials Grant）。

4）资源所有者密码凭证模式（Resource Owner Password Credentials Grant）；

3.1 授权码授权模式（Authorization Code Grant）

授权码模式在整个授权流程上与1.0版本最贴近，但是整个流程还是要简化了许多，也是OAuth2.0中最标准，应用最广泛的授权模式。这类授权模式非常适合于具备服务端的应用，当然现在大多数APP都有自己的服务端，所以大部分APP的OAuth2.0授权都可以采取授权码模式，下图为授权码在各个角色之间的交互时序：



整个授权流程说明如下（具体参数释义见下文）：

1. 客户端携带client_id, scope, redirect_uri, state等信息引导用户请求授权服务器的授权端点下发code
2. 授权服务器验证客户端身份，验证通过则询问用户是否同意授权（此时会跳转到用户能够直观看到的授权页面，等待用户点击确认授权）
3. 假设用户同意授权，此时授权服务器会将code和state（如果客户端传递了该参数）拼接在redirect_uri后面，以302形式下发code
4. 客户端携带code, redirect_uri, 以及client_secret请求授权服务器的令牌端点下发access_token（这一步实际上中间经过了客户端的服务器，除了code，其它参数都是在应用服务器端添加）
5. 授权服务器验证客户端身份，同时验证code，以及redirect_uri是否与请求code时相同，验证通过后下发access_token，并选择性下发refresh_token

3.1.1 获取授权码

授权码是授权流程的一个中间临时凭证，是对用户确认授权这一操作的一个暂时性的证书，其生命周期一般较短，协议建议最大不要超过10分钟，在这一有效时间周期内，客户端可以凭借该暂时性证书去授权服务器换取访问令牌。

请求参数说明：

名称	是否必须	描述信息
response_type	必须	对于授权码模式response_type=code
client_id	必须	客户端ID，用于标识一个客户端，等同于appId，在注册应用时生成
redirect_uri	可选	授权回调地址，具体参见2.2.3

		小节
scope	可选	权限范围，用于对客户端的权限进行控制，如果客户端没有传递该参数，那么服务器则以该应用的所有权限代替
state	推荐	用于维持请求和回调过程中的状态，防止CSRF攻击，服务器不对该参数做任何处理，如果客户端携带了该参数，则服务器在响应时原封不动的返回

请求参数示例：

```
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
```

客户端携带上述参数请求授权服务器的令牌端点，授权服务器会验证客户端的身份以及相关参数，并在确认用户登录的前提下弹出确认授权页询问用户是否授权，如果用户同意授权，则会将授权码（code）和state信息（如果客户端传递了该参数）添加到回调地址后面，以302的形式下发。

成功响应参数说明：

名称	是否必须	描述信息
code	必须	授权码，授权码代表用户确认授权的暂时性凭证，只能使用一次，推荐最大生命周期不超过10分钟
state	可选	如果客户端传递了该参数，则必须原封不动返回

成功响应示例：

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA&state=xyz
```

如果请求参数错误，或者服务器端响应错误，那么需要将错误信息添加在回调地址后面，以302形式下发（回调地址错误，或客户端标识无效除外）。

错误响应参数说明：

名称	是否必须	描述信息
error	必须	错误代码
error_description	可选	具备可读性的错误描述信息
error_uri	可选	错误描述信息页面地址
state	可选	如果客户端传递了该参数，则必须原封不动返回

错误响应示例：

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?error=access_denied&state=xyz
```

3.1.2 下发访问令牌

授权服务器的授权端点在以302形式下发code之后，用户User-Agent，比如浏览器，将携带对应的code回调请求用户指定的redirect_url，这个地址应该能够保证请求打到应用服务器的对应接口，该接口可以由此拿到code，并附加相应参数请求授权服务器的令牌端点，授权端点验证code和相关参数，验证通过则下发access_token。

请求参数说明：

名称	是否必须	描述信息
grant_type	必须	对于授权码模式 grant_type=authorization_code
code	必须	上一步骤获取的授权码
redirect_uri	必须	授权回调地址，具体参见2.2.3小节，如果上一步有设置，则必须相同
client_id	必须	客户端ID，用于标识一个客户端，等同于appId，在注册应用时生成

如果在注册应用时有下发客户端凭证信息（client_secret），那么客户端必须携带该参数以让授权服务器验证客户端的有效性。针对客户端凭证需要多说的一点就是，不能将其传递到客户端，客户端无法保证凭证的安全，凭证应该始终留在应用的服务器端，当下发code回调请求到应用服务器时，在服务器端携带上凭证再次请求下发令牌。

请求参数示例：

```
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
```

授权服务器需要验证客户端的有效性，以及是否与之前请求授权码的客户端是同一个（请求授权时的信息可以记录在code，或以code为key建立缓存），授权服务器还要保证code处于生命周期内（推荐10分钟内有效），且只能被使用一次。授权服务器验证通过之后，生成access_token，并选择性下发refresh_token，OAuth2.0协议明确了token的下发策略，对于生成策略没有做太多说明，我们将在本系列的下一篇详细介绍两种token类型，即BEARER类型和MAC类型。

成功响应参数说明：

名称	是否必须	描述信息
access_token	必须	访问令牌
token_type	必须	访问令牌类型，比如bearer，mac等等
expires_in	推荐	访问令牌的生命周期，以秒为单位，表示令牌下发后多久时间过期，如果没有指定该项，则使用默认值
refresh_token	可选	刷新令牌，选择性下发，参见2.2.2
scope	可选	权限范围，如果最终下发的访问令牌对应的权限范围与实际应用指定的不一致，则必须在下发访问令牌时用该参数指定说明

最后访问令牌以JSON格式响应，并要求指定响应首部`Cache-Control: no-store`和`Pragma: no-cache`。

成功响应示例：

```
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",
```

```
 8 "expires_in":3600,
 9 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
10 "example_parameter":"example_value"
11 }
```

错误响应参数说明：

名称	是否必须	描述信息
error	必须	错误代码
error_description	可选	具备可读性的错误描述信息
error_uri	可选	错误描述信息页面地址

错误响应示例：

```
1 HTTP/1.1 400 Bad Request
2 Content-Type: application/json;charset=UTF-8
3 Cache-Control: no-store
4 Pragma: no-cache
5 {
6 "error":"invalid_request"
7 }
```

3.1.3 对于授权码模式的一点感悟

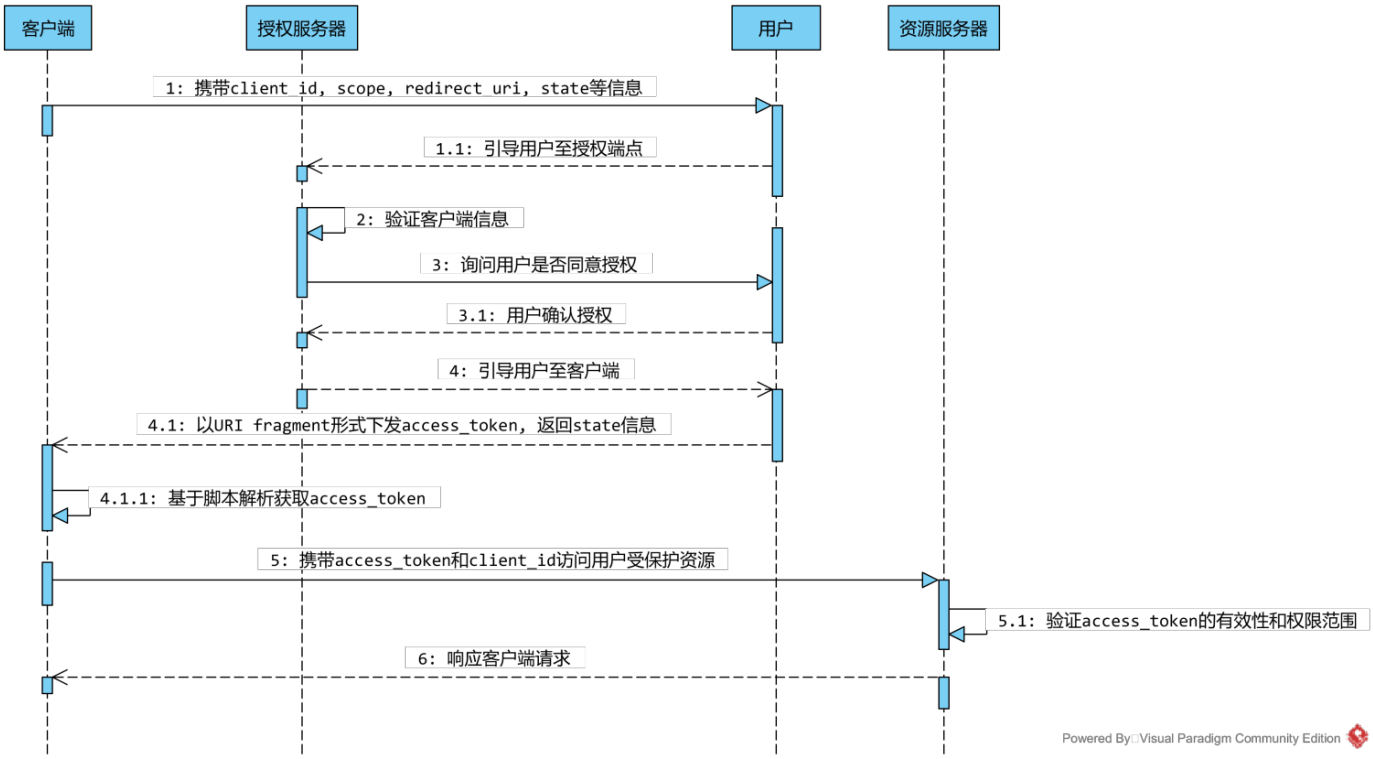
授权码授权模式是OAuth2.0协议已定义4种模式中最严谨的模式，剩余3中模式都是建立在一些特殊场景下，并对这些场景做了一些妥协和优化。授权码授权流程分为两步走，将用户授权与下发token分开，这给授权带来了更多的灵活性，正常授权过程中必须经过用户登录这一步骤，在用户已登录的前提下，可以直接询问用户是否同意授权，但是在一些场景下，比如内部走SSO登录的应用集成了基于OAuth登录的第三方应用，这个时候在OAuth授权登录第三方应用时用户体验较好的流程是不需要用户再一次输入用户名和密码登录的，这就需要将外围APP的登录态传递给该应用，但是这样是存在安全问题的，用户的登录态必须把握在走SSO登录流程的应用中，这样的场景下授权码授权模式的两步走流程就可以满足在不交出用户登录态的情况下，无需再次登录即可授权。

内部应用可以拿着第三方应用的client_id等信息代替第三方应用去请求获取code，因为自己持有用户的登录态，所以过程中无需用户再次输入用户名和密码，拿到code之后将其交给第三方应用，第三方应用利用code和自己的client_secret信息去请求授权服务器下发token，整个流程内部应用不需要交出自己持有的用户登录态，第三方应用也无需交出自己的client_secret信息，最终却能够实现在保护用户登录凭证的前提下无需再次登录即可完成整个授权流程。

3.2 隐式授权模式（Implicit Grant）

对于一些纯客户端应用，往往无法妥善的保管客户端的凭证，但是因为没有服务器端，所以无法向授权服务器传递客户端凭证，并且纯客户端应用在请求交互上要弱于有服务器的应用，这时候减少交互可以让应用的稳定性和用户体验更好，隐式授权模式是对这一应用场景的优化。

隐式授权模式在安全性上要弱于授权码模式，因为无法对当前客户端的真实性进行验证，同时对于下发的access_token存在被同设备上其它应用窃取的风险，为了降低这类风险，隐式授权模式强制要求不能下发refresh_token，这一强制要求的另外一个考量个人觉得是因为refresh_token的生命周期较长，而客户端无法安全的对其进行存储和保护。下图为授权码各个角色之间的交互时序（这里让用户直接参与其中，省略了用户代理）：



整个授权流程说明如下：

1. 客户端携带client_id, scope, redirect_uri, state等信息引导用户请求授权服务器下发access_token
2. 授权服务器验证客户端身份，验证通过则询问用户是否同意授权（此时会跳转到用户能够直观看到的授权页面，等待用户点击确认授权）
3. 假设用户同意授权，此时授权服务器会将access_token和state（如果客户端传递了该参数）等信息以URI Fragment形式拼接在redirect_uri后面，并以302形式下发
4. 客户端利用脚本解析获取access_token

3.2.1 请求获取访问令牌

不同于授权码模式的分两步走，隐式授权码模式一步即可拿到访问令牌。

请求参数说明：

名称	是否必须	描述信息
response_type	必须	对于授权码模式

		response_type=token
client_id	必须	客户端ID，用于标识一个客户端，等同于appId，在注册应用时生成
redirect_uri	可选	授权回调地址，具体参见2.2.3小节
scope	可选	权限范围，用于对客户端的权限进行控制，如果客户端没有传递该参数，那么服务器则以该应用的所有权限代替
state	推荐	用于维持请求和回调过程中的状态，防止CSRF攻击，服务器不对该参数做任何处理，如果客户端携带了该参数，则服务器在响应时原封不动的返回

请求参数示例：

```
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
```

成功响应参数说明：

名称	是否必须	描述信息
access_token	必须	访问令牌
token_type	必须	访问令牌类型，比如bearer，mac等等
expires_in	推荐	访问令牌的生命周期，以秒为单位，表示令牌下发后多久时间过期，如果没有指定该项，则使用默认值
scope	可选	权限范围，如果最终下发的访问令牌对应的权限范围与实际应用指定的不一致，则必须在下发访问令牌时用该参数指定说明
state	可选	如果客户端传递了该参数，则必

		须原封不动返回

隐式授权模式不下发刷新令牌，访问令牌以URI Fragment的形式拼接在授权回调地址后面以302形式下发，并要求指定响应首部`Cache-Control: no-store`和`Pragma: no-cache`。

成功响应示例：

```
HTTP/1.1 302 Found
Location: http://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpA
A&state=xyz&token_type=example&expires_in=3600
```

错误响应参数说明：

名称	是否必须	描述信息
error	必须	错误代码
error_description	可选	具备可读性的错误描述信息
error_uri	可选	错误描述信息页面地址
state	可选	如果客户端传递了该参数，则必须原封不动返回

授权服务器将上述元素以URI Fragment形式拼接在授权回调地址后面以302形式下发（redirect_uri或client_id错误除外）。

错误响应参数示例：

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb#error=access_denied&state=
xyz
```

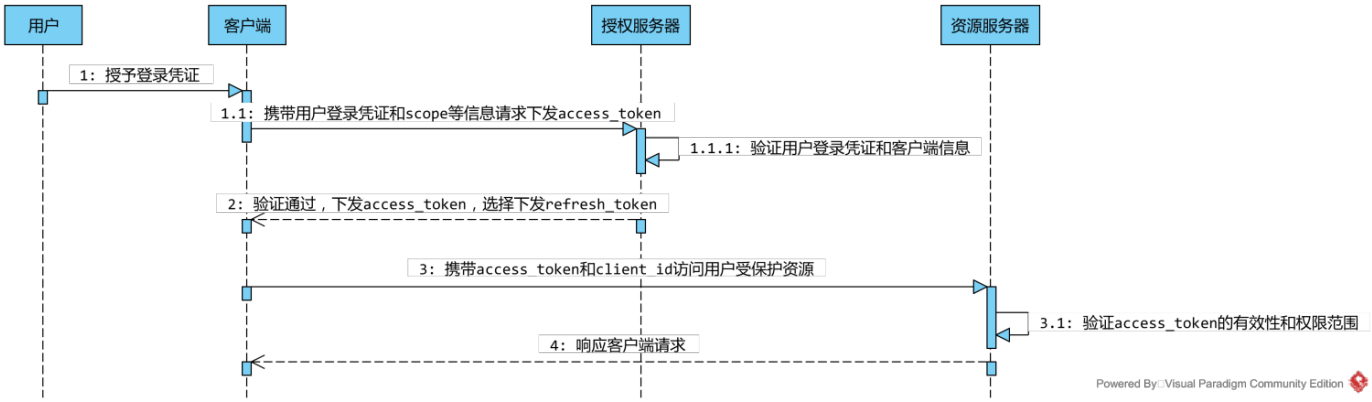
3.3 资源所有者密码凭证授权模式（Resource Owner Password Credentials Grant）

资源所有者密码凭证授权模式建立在资源所有者充分信任客户端的前提下，因为该模式客户端可以拿到用的登录凭证，从而在用户无感知的情况下完成整个授权流程，毕竟都有用户的登录凭证了，再弹窗让用户确认授权也是多此一举。

这里可能有一个比较疑惑的地方是既然已经拿到了用户的登录凭证，为什么还需要绕一大圈子走OAuth授权，拿到令牌再去请求用户的受保护资源呢？实际中事情可能并不会这么简单，拿到用户登录凭证的不一定是用户本身，而且这里协议指的用户登录凭证是用户的用户名和密码，实际中还可以是走SSO登录下发的token，token在持有权限上要小于等于用户的用户名和密码，这是从客户端角度出发，对于资源服务器来说，有些敏感数据需要在用户级别做权限控制，对于服务级别的控制粒度太粗，所以这些服务往往需要服务携带access_token来请求某一个用户的敏感数据。

举个例子来说，比如有一个服务是获取某个用户的通讯录，这是一个十分敏感的数据，且一般只能授予内部应用，如果是在服务级别进行控制，那么只要拿到服务权限，该应用可以请求获取任何一个用户的

通讯录数据，这是一件十分危险的事情。然而如果基于access_token来做鉴权，那么就可以将粒度控制在用户级别，前面讲的两种授权方式在这里应用时都有一个共同的缺点，需要弹出授权页让用户确认授权，要知道这样的场景往往是发生在内部应用里面，内部应用是可以持有用户登录态的，这里的确认授权对于一个用户体验好的APP来说就应该发生在用户登录时，通过用户协议等方式直接告诉用户，从而让用户在一次登录过程中可以让应用拿到用户的登录态和访问令牌。资源所有者密码凭证授权模式的交互时序如下：



整个授权流程说明如下：

1. 用于授予客户端登录凭证（比如用户名和密码信息）
2. 客户端携带用户的登录凭证和scope等信息请授权服务器的令牌端点下发refresh_token
3. 授权服务器验证用户的登录凭证和客户端信息的有效性，验证通过则下发access_token，并选择性下发refresh_token

3.3.1 用户授予登录凭证

用于登录凭证如何传递给客户端这一块协议未做说明，实际应用中该类授权一般应用在内部应用，这类应用的特点就是为用户提供登录功能，当用户登录之后，这类应用也就持有了用户的登录态，可以是用户登录的session标识，也可以是走SSO下发的token信息。

3.3.2 请求获取访问令牌

请求参数说明：

名称	是否必须	描述信息
grant_type	必须	对于本模式 grant_type=password
username	必须	用户名
password	必须	用户密码
scope	可选	权限范围，如果最终下发的访问令牌对应的权限范围与实际应用指定的不一致，则必须在下发访问令牌时用该参数指定说明

如果在注册应用时有下发客户端凭证信息（client_secret），那么客户端必须携带该参数以让授权服务器验证客户端的有效性。

请求参数示例：

```
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
```

成功响应参数说明：

名称	是否必须	描述信息
access_token	必须	访问令牌
token_type	必须	访问令牌类型，比如bearer，mac等等
expires_in	推荐	访问令牌的生命周期，以秒为单位，表示令牌下发后多久时间过期，如果没有指定该项，则使用默认值
refresh_token	可选	刷新令牌，选择性下发，参见2.2.2
scope	可选	权限范围，如果最终下发的访问令牌对应的权限范围与实际应用指定的不一致，则必须在下发访问令牌时用该参数指定说明

最后访问令牌以JSON格式响应，并要求指定响应首部`Cache-Control: no-store`和`Pragma: no-cache`。

成功响应参数示例：

```
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,
```

```
 9 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
10 "example_parameter":"example_value"
11 }
```

错误响应参数说明：

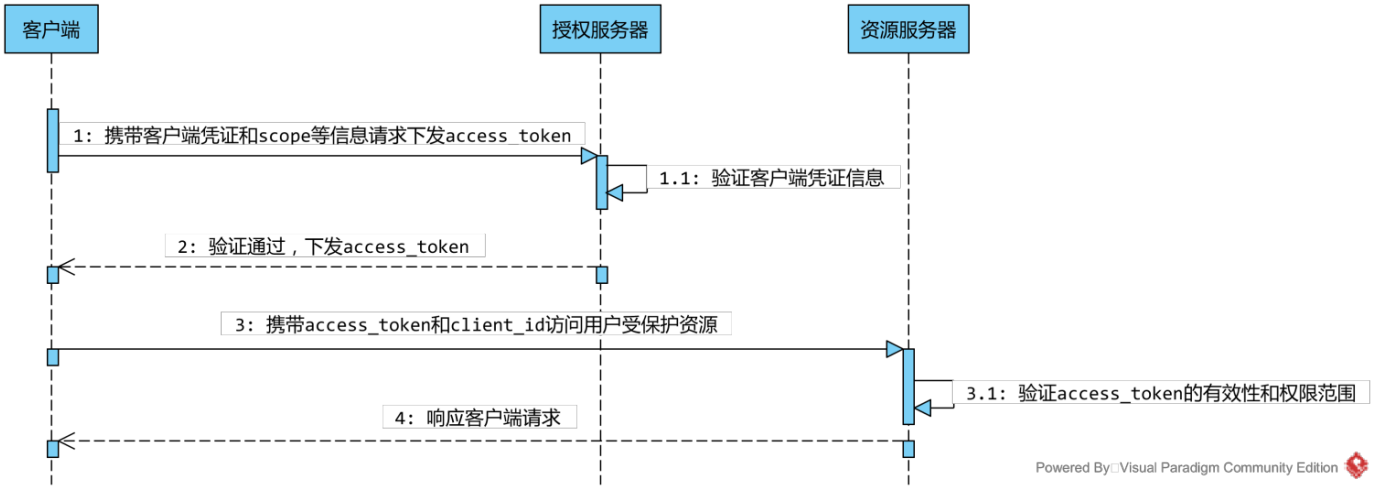
名称	是否必须	描述信息
error	必须	错误代码
error_description	可选	具备可读性的错误描述信息
error_uri	可选	错误描述信息页面地址

错误响应示例：

```
1 HTTP/1.1 400 Bad Request
2 Content-Type: application/json;charset=UTF-8
3 Cache-Control: no-store
4 Pragma: no-cache
5 {
6 "error":"invalid_request"
7 }
```

3.4 客户端凭证授权模式（Client Credentials Grant）

客户端凭证授权模式基于客户端持有的证书去请求用户的受保护资源，如果把这里的受保护资源定义得更加宽泛一点，比如说是对一个内网接口权限的调用，那么这类授权方式可以被改造为内网权限验证服务。客户端凭证授权模式的交互时序如下：



整个授权流程说明如下：

1. 客户端携带客户端凭证和scope等信息请求授权服务器的令牌端点
2. 授权服务器验证客户端凭证，验证通过下发access_token

3.4.1 请求获取访问令牌：

请求参数说明：

名称	是否必须	描述信息
grant_type	必须	对于本模式`grant_type=client_credentials`
scope	可选	权限范围，如果最终下发的访问令牌对应的权限范围与实际应用指定的不一致，则必须在下发访问令牌时用该参数指定说明

请求参数示例：

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
grant_type=client_credentials
```

成功响应参数说明：

名称	是否必须	描述信息
access_token	必须	访问令牌
token_type	必须	访问令牌类型，比如bearer，mac等等
expires_in	推荐	访问令牌的生命周期，以秒为单位，表示令牌下发后多久时间过期，如果没有指定该项，则使用默认值
scope	可选	权限范围，如果最终下发的访问令牌对应的权限范围与实际应用指定的不一致，则必须在下发访问令牌时用该参数指定说明

最后访问令牌以JSON格式响应，并要求指定响应首部`Cache-Control: no-store`和`Pragma: no-cache`。

成功响应参数示例：

```
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,
"example_parameter":"example_value"
}
```

错误响应参数说明：

名称	是否必须	描述信息
error	必须	错误代码
error_description	可选	具备可读性的错误描述信息
error_uri	可选	错误描述信息页面地址

错误响应示例：

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
"error":"invalid_request"
}
```

四. 本篇小结

本篇介绍了OAuth2.0授权协议的理论知识，OAuth2.0被广泛应用于第三方授权登录，很多其它的协议都是可以基于该协议进行改造的，比如前面多次提到的SSO，作为开发人员，还是建议对该协议或多或少有些了解。

OAuth2.0协议是一个介绍授权框架的协议，对于token的生成没有做太多说明，如果要自己实现一个OAuth授权和鉴权服务，OAuth2.0协议为我们的服务的框架建立绘制了蓝图，但是还有很多细节实现需要我们再去查阅各种资料和实践，本系列还有两篇文章，下一篇我将介绍token的生成策略。

参考文献

1. RFC5849 – The OAuth 1.0 Protocol
2. RFC6749 – The OAuth 2.0 Authorization Framework

3. RFC6750 – The OAuth 2.0 Authorization Framework: Bearer Token Usage
4. HTTP Authentication: MAC Authentication (draft–hammer–oauth–v2–mac–token–02)