前端鉴权与后端鉴权方案

前言

什么是认证？

`认证(Identification)`是指根据声明者所特有的识别信息，确认声明者的身份。

白话文的意思就是：`你需要用身份证证明你自己是你自己`。

比如我们常见的认证技术：

- 身份证
- 用户名和密码

- 用户手机：手机短信、手机二维码扫描、手势密码
- 用户的电子邮箱
- 用户的生物学特征：指纹、语音、眼睛虹膜
- 用户的大数据识别
- 等等

什么是授权？

授权(Authorization)：在信息安全领域是指资源所有者委派执行者，赋予执行者指定范围的资源操作权限，以便对资源的相关操作。

在现实生活领域例如：银行卡（由银行派发）、门禁卡（由物业管理处派发）、钥匙（由房东派发），这些都是现实生活中授权的实现方式。

在互联网领域例如：web 服务器的 session 机制、web 浏览器的 cookie 机制、颁发授权令牌（token）等都是一个授权的机制。

什么是鉴权？

鉴权(Authentication)在信息安全领域是指**对于一个声明者所声明的身份权利，对其所声明的真实性进行鉴别确认的过程**。

若从授权出发，则会更加容易理解鉴权。授权和鉴权是两个上下游相匹配的关系，**先授权，后鉴权**。

在现实生活领域：门禁卡需要通过门禁卡识别器，银行卡需要通过银行卡识别器；

在互联网领域：校验 session/cookie/token 的合法性和有效性

鉴权是一个承上启下的一个环节，上游它接受授权的输出，校验其真实性后，然后获取权限（permission），这个将会为下一步的权限控制做好准备。

什么是权限控制？

权限控制(Access/Permission Control)将可执行的操作定义为权限列表，然后判断操作是否允许/禁止

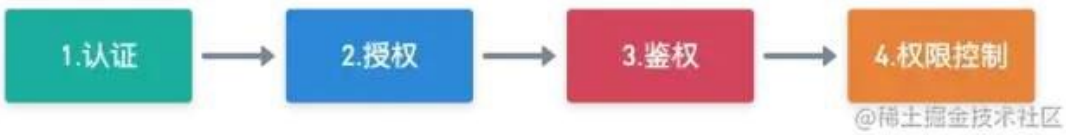
对于权限控制，可以分为两部分进行理解：一个是权限，另一个是控制。权限是抽象的逻辑概念，而控制是具体的实现方式。

在现实生活领域中：以门禁卡的权限实现为例，一个门禁卡，拥有开公司所有的门的权限；一个门禁卡，拥有管理员角色的权限，因而可以开公司所有的门。

在互联网领域：通过 web 后端服务，来控制接口访问，允许或拒绝访问请求。

认证、授权、鉴权和权限控制的关系？

看到这里，我们应该明白了认证、授权、鉴权和权限控制这四个环节是一个前后依次发生、上下游的关系;



需要说明的是，这四个环节在有些时候会同时发生。例如在下面的几个场景：

- **使用门禁卡开门：**认证、授权、鉴权、权限控制四个环节一气呵成，在瞬间同时发生
- **用户的网站登录：**用户在使用用户名和密码进行登录时，认证和授权两个环节一同完成，而鉴权和权限控制则发生在后续的请求访问中，比如在选购物品或支付时。

这里提个小问题，供大家思考：**认证和鉴权之间的关系？**

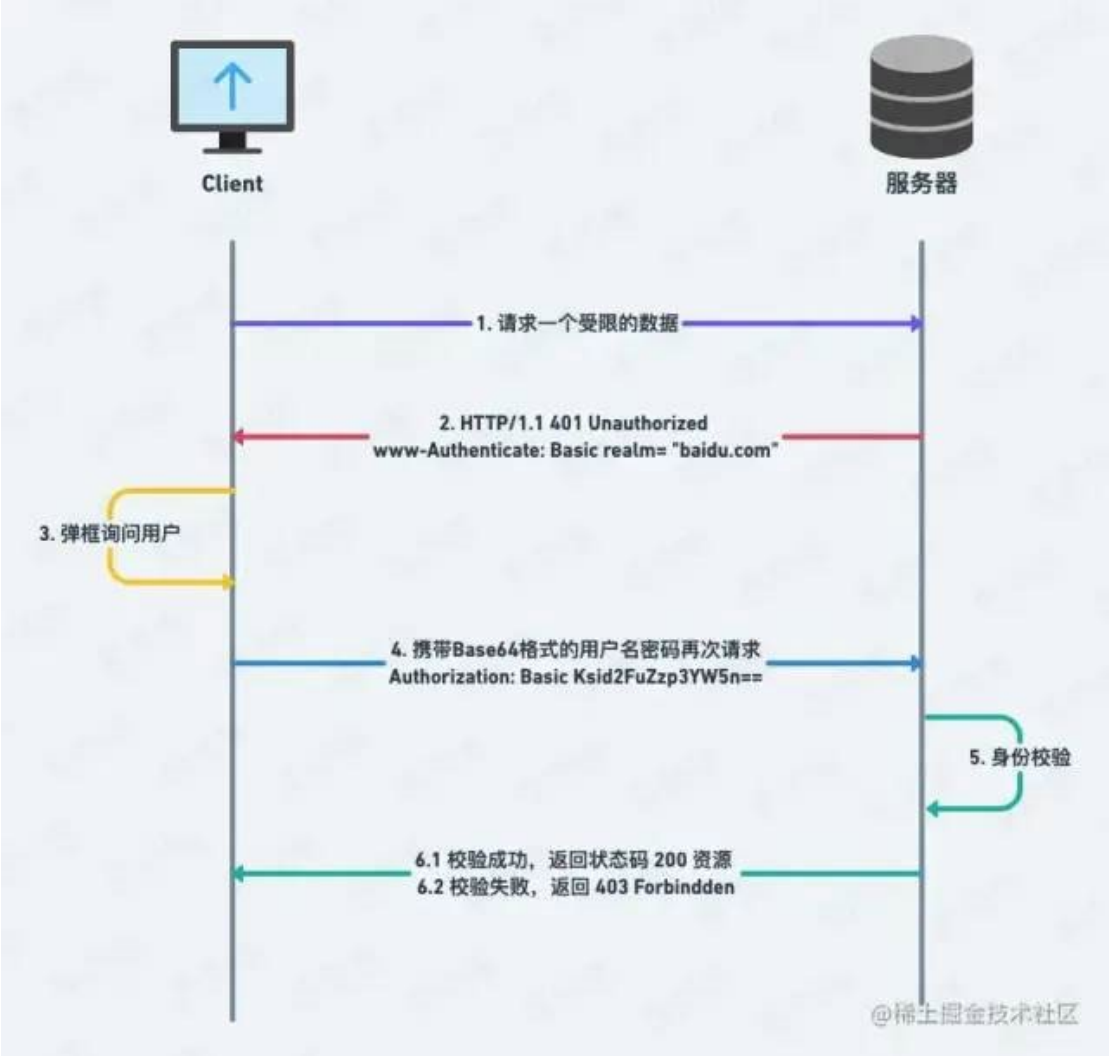
鉴权方案：

1. HTTP 基本鉴权

在 HTTP 中，`基本认证方案（Basic Access Authentication)`是允许客户端（通常指的就是网页浏览器）在请求时，通过用户提供用户名和密码的方式，实现对用户身份的验证。

因为几乎所有的线上网站都不会走该认证方案，所以该方案大家了解即可

1.1 认证流程图



1.2 认证步骤解析

1. **客户端(如浏览器)**：向服务器请求一个受限的列表数据或资源，例如字段如下

```
GET /list/ HTTP/1.1
Host: www.baidu.com
Authorization: Basic aHR0cHdhdGNoOmY=
```

2. **服务器**：客户端你好，这个资源在安全区 baidu.com 里，是受限资源，需要基本认证；

   并且向客户端返回 401 状态码（Unauthorized 未被授权的）以及附带提供了一个认证域 `www-Authenticate: Basic realm="baidu.com"`要求进行身份验证；

其中 `Basic` 就是验证的模式，而 `realm="baidu.com"` 说明客户端需要输入这个安全域的用户名和密码，而不是其他域的

```
HTTP/1.1 401 Unauthorized
www-Authenticate: Basic realm= "baidu.com"
```

3. **客户端：**服务器，我已经携带了用户名和密码给你了，你看一下；

   （注：如客户端是浏览器，那么此时会自动弹出一个弹窗，让用户输入用户名和密码）；

   输入完用户名和密码后，则客户端将用户名及密码以 Base64 加密方式发送给服务器

   传送的格式如下（其中 Basic 内容为：**用户名:密码 的 ase64 形式**）：

   ```
   GET /list/ HTTP/1.1
   Authorization: Basic Ksid2FuZzp3YW5n==
   ```

4. **服务器：**客户端你好，我已经校验了 `Authorization` 字段你的用户名和密码，是正确的，这是你要的资源。

   ```
   HTTP/1.1 200 OK
   ...
   ```

1.3 优点

简单，基本所有流行的浏览器都支持

1.4 缺点

1. **不安全：**
   - 由于是基于 HTTP 传输，所以它在网络上几乎是裸奔的，虽然它使用了 Base64 来编码，但这个编码很容易就可以解码出来。

- 即使认证内容无法被解码为原始的用户名和密码也是不安全的，恶意用户可以再获取了认证内容后使用其不断的享服务器发起请求，这就是所谓的重放攻击。

2. **无法主动注销：**
   - 由于 HTTP 协议没有提供机制清除浏览器中的 Basic 认证信息，除非标签页或浏览器关闭、或用户清除历史记录。

1.5 使用场景

内部网络，或者对安全要求不是很高的网络。

2. Session-Cookie 鉴权

`Session-Cookie` 认证是利用服务端的 **Session（会话）**和**浏览器（客户端）**的 Cookie 来实现的前后端通信认证模式。

在理解这句话之前我们先简单了解下`什么是 Cookie`以及`什么是 Session`？

2.1 什么是 Cookie

众所周知，`HTTP 是无状态的协议`（对于事务处理没有记忆能力，每次客户端和服务端会话完成时，服务端不会保存任何会话信息）；

所以为了让服务器区分不同的客户端，就必须主动的去维护一个状态，这个状态用于告知服务端前后两个请求是否来自同一浏览器。而这个状态可以通过 `Cookie` 去实现。

特点：

- Cookie 存储在客户端，可随意篡改，不安全
- 有大小限制，最大为 4kb

- 有数量限制，一般一个浏览器对于一个网站只能存不超过 20 个 Cookie，浏览器一般只允许存放 300 个 Cookie
- Android 和 IOS 对 Cookie 支持性不好
- Cookie 是不可跨域的，但是一级域名和二级域名是允许共享使用的（靠的是 domain）

2.2 什么是 Session

Session 的抽象概念是会话，是无状态协议通信过程中，为了实现中断/继续操作，将用户和服务器之间的交互进行的一种抽象；

具体来说，是服务器生成的一种 Session 结构，可以通过多种方式保存，如内存、数据库、文件等，大型网站一般有专门的 Session 服务器集群来保存用户会话；

原理流程：

1. **客户端：**用户向服务器首次发送请求；
2. **服务器：**接收到数据并自动为该用户创建特定的 Session / Session ID，来标识用户并跟踪用户当前的会话过程；
3. **客户端：**浏览器收到响应获取会话信息，并且会在下一次请求时带上 Session / Session ID；
4. **服务器：**服务器提取后会与本地保存的 Session ID 进行对比找到该特定用户的会话，进而获取会话状态；
5. 至此客户端与服务器的通信变成有状态的通信；

特点：

- Session 保存在服务器上；

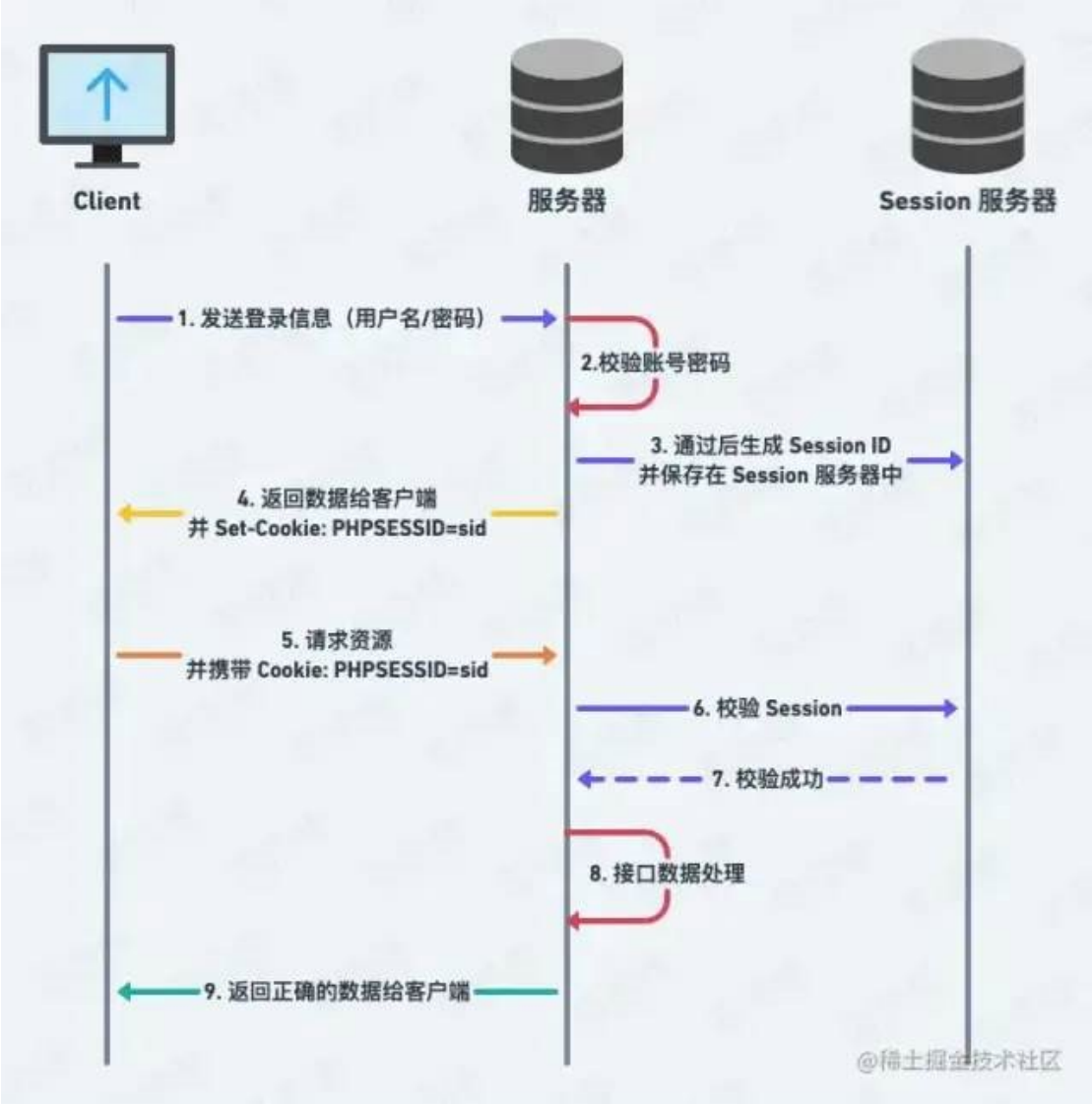
- 通过服务器自带的加密协议进行；

与 Cookie 的差异：

- **安全性：**Cookie 由于保存在客户端，可随意篡改，Session 则不同存储在服务器端，无法伪造，所以 Session 的安全性更高；
- **存取值的类型不同：**Cookie 只支持字符串数据，Session 可以存任意数据类型；
- **有效期不同：**Cookie 可设置为长时间保持，Session 一般失效时间较短；
- **存储大小不同：**Cookie 保存的数据不能超过 4K；

看到这里可能就有同学想到了，`Session-Cookie` 是不是就是把 `Session` 存储在了客户端的 `Cookie` 中呢？
Bingo，的确是这样的，我们接着往下看

2.3 Session-Cookie 的认证流程图



2.4 Session-Cookie 认证步骤解析

1. **客户端**：向服务器发送登录信息用户名/密码来请求登录校验；
2. **服务器**：验证登录的信息，验证通过后自动创建 Session（将 Session 保存在内存中，也可以保存在 Redis 中），然后给这个 Session 生成一个唯一的标识字符串会话身份凭证 `session_id`(通常称为 `sid`)，并在响应头 `Set-Cookie` 中设置这个唯一标识符；

   注：可以使用签名对 `sid` 进行加密处理，服务端会根据对应的 `secret` 密钥进行解密 （非必须步骤）

3. **客户端**：收到服务器的响应后会解析响应头，并自动将 `sid` 保存在本地 Cookie 中，浏览器在下次 HTTP 请求时请求头会自动附带上该域名下的 Cookie 信息；
4. **服务器**：接收客户端请求时会去解析请求头 Cookie 中的 `sid`，然后根据这个 `sid` 去找服务端保存的该客户端的 `sid`，然后判断该请求是否合法；

2.5 Session-Cookie 的优点

- Cookie 简单易用
- Session 数据存储在服务端，相较于 JWT 方便进行管理，也就是当用户登录和主动注销，只需要添加删除对应的 Session 就可以了，方便管理
- 只需要后端操作即可，前端可以无感等进行操作；

2.6 Session-Cookie 的缺点

- 依赖 Cookie，一旦用户在浏览器端禁用 Cookie，那么就 GG 思密达了；
- 非常不安全，Cookie 将数据暴露在浏览器中，增加了数据被盗的风险（容易被 CSRF 等攻击）；
- Session 存储在服务端，增大了服务端的开销，用户量大的时候会大大降低服务器性能；
- 对移动端的支持性不友好；

2.7 使用场景

- 一般中大型的网站都适用（除了 APP 移动端）；

- 由于一般的 Session 需集中存储在内存服务器上（如 Redis），这样就会增加服务器的预算，所以预算不够请谨慎选择；

2.8 前端常用的 Session 库推荐

- 使用 express：express-session[1]
- 使用 koa：koa-session[2]

3. Token 鉴权

现在我们已经得知，`Session-Cookie`的一些缺点，以及 Session 的维护给服务端造成很大困扰，我们必须找地方存放它，又要考虑分布式的问题，甚至要单独为了它启用一套 Redis 集群。那有没有更好的办法？那`Token`就应运而生了

3.1 什么是 Token（令牌）

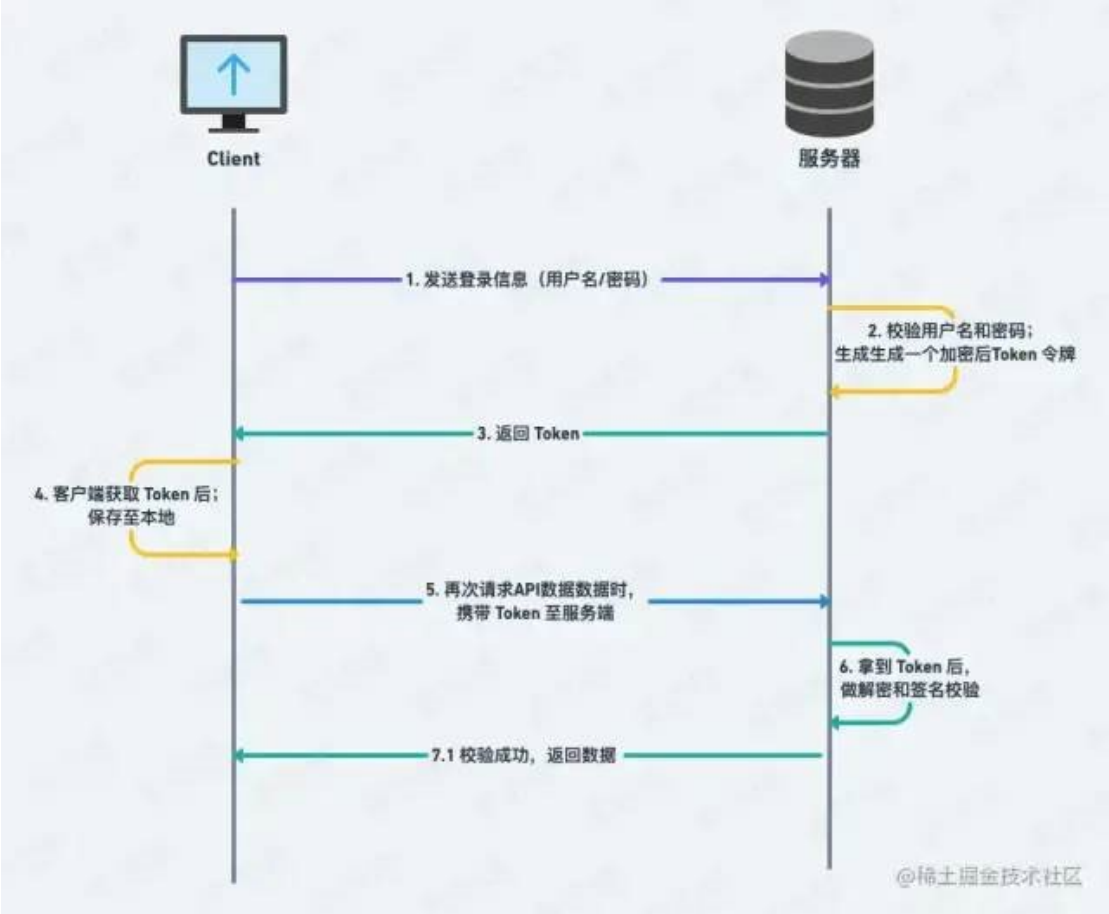
`Token`是一个令牌，客户端访问服务器时，验证通过后服务端会为其签发一张令牌，之后，客户端就可以携带令牌访问服务器，服务端只需要验证令牌的有效性即可。

一句话概括：**访问资源接口（API）时所需要的资源凭证**

一般 Token 的组成：

uid(用户唯一的身份标识) +**time**(当前时间的时间戳) +**sign**(签名，Token 的前几位以哈希算法压缩成的一定长度的十六进制字符串)

Token 的认证流程图：



Token 认证步骤解析：

1. **客户端：**输入用户名和密码请求登录校验；
2. **服务器：**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 并把这个 Token 发送给客户端；
3. **客户端：**收到 Token 以后需要把它存储起来，web 端一般会放在 localStorage 或 Cookie 中，移动端原生 APP 一般存储在本地缓存中；
4. **客户端发送请求：**向服务端请求 API 资源的时候，将 Token 通过 HTTP 请求头 Authorization 字段或者其它方式发送给服务端；

5. **服务器：**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，否则拒绝返还（401）；

Token 的优点：

- **服务端无状态化、可扩展性好：**Token 机制在服务端不需要存储会话（Session）信息，因为 Token 自身包含了其所标识用户的相关信息，这有利于在多个服务间共享用户状态
- **支持 APP 移动端设备；**
- **安全性好：**有效避免 CSRF 攻击（因为不需要 Cookie）
- **支持跨程序调用：**因为 Cookie 是不允许跨域访问的，而 Token 则不存在这个问题

Token 的缺点：

- **配合：**需要前后端配合处理；
- **占带宽：**正常情况下比 `sid` 更大，消耗更多流量，挤占更多宽带
- **性能问题：**虽说验证 Token 时不用再去访问数据库或远程服务进行权限校验，但是需要对 Token 加解密等操作，所以会更耗性能；
- **有效期短：**为了避免 Token 被盗用，一般 Token 的有效期会设置的较短，所以就有了 `Refresh Token`；

3.2 什么是 Refresh Token（刷新 Token）

业务接口用来鉴权的 Token，我们称之为 `Access Token`。

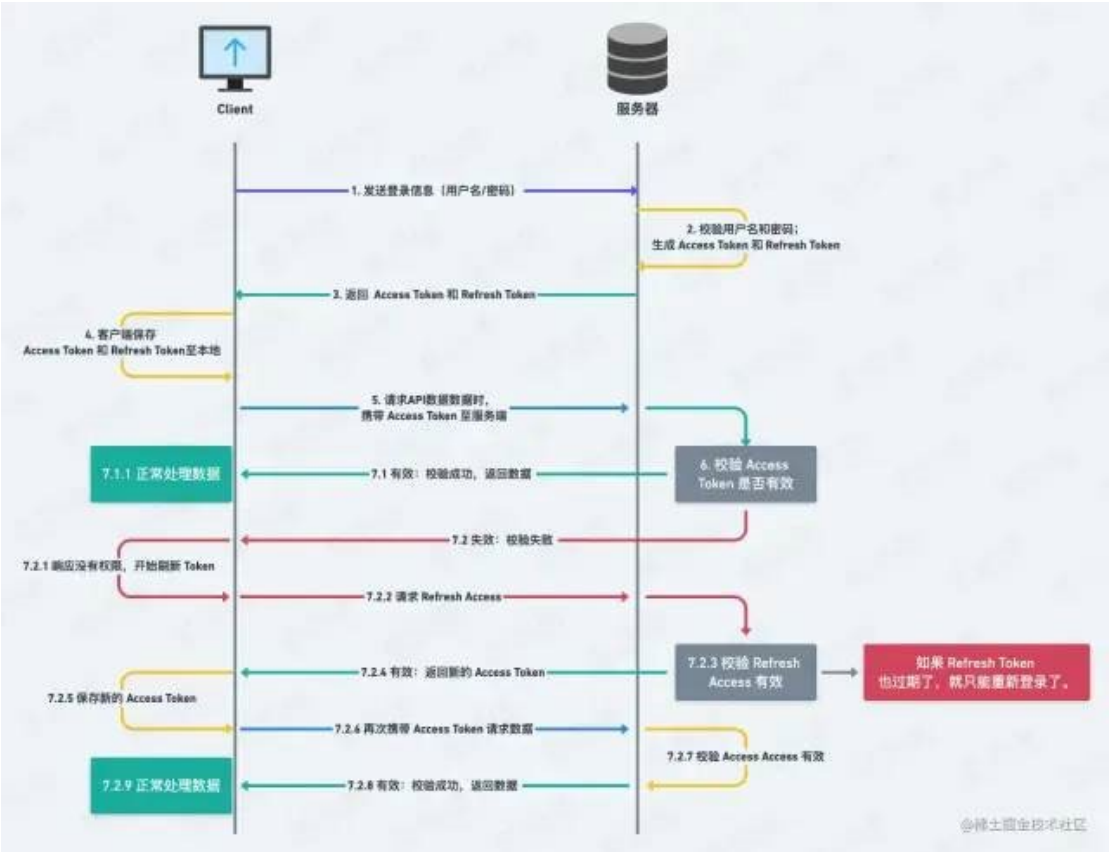
为了安全，我们的 `Access Token` 有效期一般设置较短，以避免被盗用。但过短的有效期会造成 `Access Token` 经常过期，过期后怎么办呢？

一种办法是：`刷新 Access Token`，让用户重新登录获取新 Token，会很麻烦；

另外一种办法是：再来一个 Token，一个专门生成 Access Token 的 Token，我们称为 `Refresh Token`；

- **Access Token**：用来访问业务接口，由于有效期足够短，盗用风险小，也可以使请求方式更宽松灵活；
- **Refresh Token**：用来获取 Access Token，有效期可以长一些，通过独立服务和严格的请求方式增加安全性；由于不常验证，也可以如前面的 Session 一样处理；

Refresh Token 的认证流程图：



Refresh Token 认证步骤解析：

1. **客户端**：输入用户名和密码请求登录校验；
2. **服务端**：收到请求，验证用户名与密码；验证成功后，服务端会签发一个 `Access Token` 和 `Refresh Token` 并返回给客户端；
3. **客户端**：把 `Access Token` 和 `Refresh Token` 存储在本地；
4. **客户端发送请求**：请求数据时，携带 `Access Token` 传输给服务端；
5. **服务端**：
    - 验证 Access Token 有效：正常返回数据
    - 验证 Access Token 过期：拒绝请求
6. **客户端**(Access Token 已过期)：则重新传输 Refresh Token 给服务端；
7. **服务端**(Access Token 已过期)：验证 Refresh Token ，验证成功后返回新的 Access Token 给客户端；
8. **客户端**：重新携带新的 Access Token 请求接口；

3.3 Token 和 Session-Cookie 的区别

`Session-Cookie` 和 `Token` 有很多类似的地方，但是 `Token` 更像是 `Session-Cookie` 的升级改良版。

- **存储地不同**：Session 一般是存储在服务端；Token 是无状态的，一般由前端存储；
- **安全性不同**：Session 和 Token 并不矛盾，作为身份认证 Token 安全性比 Session 好，因为每一个请求都有签名还能防止监听以及重放攻击；

- **支持性不同：**Session-Cookie 认证需要靠浏览器的 Cookie 机制实现，如果遇到原生 NativeAPP 时这种机制就不起作用了，或是浏览器的 Cookie 存储功能被禁用，也是无法使用该认证机制实现鉴权的；而 Token 验证机制丰富了客户端类型。

如果你的用户数据可能需要和第三方共享，或者允许第三方调用 API 接口，用 Token 。如果永远只是自己的网站，自己的 App，用什么就无所谓了。

4. JWT（JSON Web Token）鉴权

通过第三节，我们知道了 `Token` 的使用方式以及组成，我们不难发现，服务端验证客户端发送过来的 Token 时，还需要查询数据库获取用户基本信息，然后验证 Token 是否有效；

这样每次请求验证都要查询数据库，增加了查库带来的延迟等性能消耗；

那么这时候业界常用的 `JWT` 就应运而生了！！！

4.1 什么是 JWT

`JWT` 是 `Auth0` 提出的通过`对 JSON 进行加密签名`来实现授权验证的方案；

就是登录成功后将相关用户信息组成 JSON 对象，然后对这个对象进行某种方式的`加密`，返回给客户端；客户端在下次请求时带上这个 Token；服务端再收到请求时`校验 token 合法性`，其实也就是在校验请求的合法性。

4.2 JWT 的组成

JWT 由三部分组成：`Header 头部`、`Payload 负载`和 `Signature 签名`

它是一个很长的字符串，中间用点（`.`）分隔成三个部分。列如 ：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Header 头部：

在 Header 中通常包含了两部分：

- typ：代表 Token 的类型，这里使用的是 JWT 类型；
- alg：使用的 Hash 算法，例如 HMAC SHA256 或 RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload 负载：

它包含一些声明 Claim (实体的描述，通常是一个 User 信息，还包括一些其他的元数据)，用来存放实际需要传递的数据，JWT 规定了 7 个官方字段：

- iss (issuer)：签发人
- exp (expiration time)：过期时间
- sub (subject)：主题
- aud (audience)：受众
- nbf (Not Before)：生效时间
- iat (Issued At)：签发时间
- jti (JWT ID)：编号

除了官方字段，你还可以在这个部分定义私有字段，下面就是一个例子。

```
{
  "sub": "1234567890",
  "name": "John Doe",
```

```
  "admin": true
}
```

Signature 签名

Signature 部分是对前两部分的签名，防止数据篡改。

首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

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

4.3JWT 的使用方式

客户端收到服务器返回的 JWT，可以储存在 Cookie 里面，也可以储存在 localStorage。

此后，客户端每次与服务器通信，都要带上这个 JWT。你可以把它放在 Cookie 里面自动发送，但是这样不能跨域，所以更好的做法是放在 HTTP 请求的头信息 `Authorization` 字段里面。

```
Authorization: Bearer <token>
```

4.4 JWT 的认证流程图

其实 JWT 的认证流程与 Token 的认证流程差不多，只是不需要再单独去查询数据库查找用户用户；简要概括如下：

4.5 JWT 的优点

- 不需要在服务端保存会话信息（RESTful API 的原则之一就是无状态），所以易于应用的扩展，即信息不保存在服务端，不会存在 Session 扩展不方便的情况；

- JWT 中的 Payload 负载可以存储常用信息，用于信息交换，有效地使用 JWT，可以降低服务端查询数据库的次数

4.6 JWT 的缺点

- **加密问题：**JWT 默认是不加密，但也是可以加密的。生成原始 Token 以后，可以用密钥再加密一次。
- **到期问题：**由于服务器不保存 Session 状态，因此无法在使用过程中废止某个 Token，或者更改 Token 的权限。也就是说，一旦 JWT 签发了，在到期之前就会始终有效，除非服务器部署额外的逻辑。

4.7 前端常用的 JWT 库推荐

- 使用 express：express-jwt[3]
- 使用 koa：koa-jwt[4]

5. 单点登录（Single Sign On）

前面我们已经知道了，在同域下的客户端/服务端认证系统中，通过客户端携带凭证，可以维持一段时间内的登录状态。

但随着企业的发展，一个大型系统里可能包含 n 多子系统，用户在操作不同的系统时，需要多次登录，很麻烦，那么单点登录（SSO）就可以很好的解决这个问题的，在多个应用系统中，只需要登录一次，就可以访问其他相互信任的应用系统。

- 例如登录天猫，淘宝也会自动登录；
- 登录百度贴吧，百度网盘也会自动登录；

5.1 同域下的 SSO（主域名相同）

当百度网站存在两个相同主域名下的贴吧子系统 `tieba.baidu.com` 和网盘子系统 `pan.baidu.com` 时，以下为他们实现 SSO 的步骤：

1. **客户端：**用户访问某个子系统时（例如 `tieba.baidu.com`），如果没有登录，则跳转至 SSO 认证中心提供的登录页面进行登录；
2. **服务端：**登录认证后，服务端把登录用户的信息存储于 Session 中，并且附加在响应头的 `Set-Cookie` 字段中，设置 Cookie 的 Domain 为 `.baidu.com`；
3. 客户端：再次发送请求时，携带主域名 Domain 下的 Cookie 给服务器，此时服务端就可以通过该 Cookie 来验证登录状态了；

其实我们不难发现，这就是我们上面讲的 `Session-Cookie 认证`登录方式；但如果是不同域呢？毕竟不同域之间 Cookie 是不共享的，那怎么办？

5.2 跨域下的 SSO（主域名不同）

在我们常见的购物网站天猫 (tmall.com) 和 淘宝 (taobao.com) 中，我们只需要登录其中某一个系统，另外一个系统打开后就会默认登录，那么这是怎么做的呢？

那么就有了 `CAS（Central Authentication Service）中央授权服务`，那么我们先主要说下 `CAS` 的流程；

单点登录下的 CAS 认证流程图：

单点登录下的 CAS 认证步骤详解：

1. **客户端：**开始访问系统 A；
2. **系统 A：**发现用户未登录，重定向至 CAS 认证服务（sso.com），同时 URL 地址参数携带登录成功后回跳到系统 A 的页面链接

（ https://sso.com/login?redirectURL=https%3A%2F%2Fwww.taobao.com%EF%BC%89%EF%BC%9B）

3. **CAS 认证服务：**发现请求 Cookie 中没有携带登录的票据凭证（TGC），所以 CAS 认证服务判定用户处于未登录状态，重定向用户页面至 CAS 的登录界面，用户在 CAS 的登录页面上进行登录操作。
4. **客户端：**输入用户名密码进行 CAS 系统认证；
5. **CAS 认证服务：**校验用户信息，并且生成 TGC 放入自己的 Session 中，同时以 Set-Cookie 形式写入 Domain 为 sso.com 的域下 ；同时生成一个授权令牌 ST (Service Ticket)，然后重定向至系统 A 的地址，重定向的地址中包含生成的 ST（重定向地址：https://www.taobao.com?token=ST-345678%EF%BC%89）
6. **系统 A：**拿着 ST 向 CAS 认证服务发送请求，CAS 认证服务验证票据 (ST) 的有效性。验证成功后，系统 A 知道用户已经在 CAS 登录了（其中的 ST 可以保存到 Cookie 或者本地中），系统 A 服务器使用该票据 (ST) 创建与用户的会话，称为局部会话，返回受保护资源；

到这里客户端就可以跟系统 A 愉快的交往啦 ~

7.客户端：开始访问系统 B；

8.系统 B：发现用户未登录，重定向至 SSO 认证服务，并将自己的地址作为参数传递，并附上在 sso.com 域下的 cookie 值是第五步生成的 TGC；

9. CAS 认证服务：CAS 认证服务中心发现用户已登录，跳转回系统 B 的地址，并附上票据 (ST)；

10.系统 B：拿到票据 (ST)，去 CAS 认证服务验证票据 (ST) 的有效性。验证成功后，客户端也可以跟系统 B 交往了 ~

(PS：脚踏两只船，感觉有点渣呀 ~)

单点登录下需要注意的点：

- 如图中流程所示，我们发现`CAS 认证服务`在签发的`授权令牌 ST`后，直接重定向，这样其实是比较容易容易被窃取，那么我们需要在系统 A 或者系统 B 在向 CAS 验证成功 (如图中的第 14 步和第 11 步) 后，再生成另一个新的验证 Token 返回给客户端保存；
- CAS 一般提供四个接口：
  - `/login`：登录接口，用于登录到中央授权服务
  - `/logout`：登出接口，用于从中央授权服务中登出
  - `/validate`：用于验证用户是否登录中央授权服务
  - `/serviceValidate`：用于让各个 Service 验证用户是否登录中央授权服务
- CAS 生成的票据：
  - **TGT（Ticket Grangting Ticket）：**TGT 是 CAS 为用户签发的`登录票据`，拥有了 TGT，用户就可以证明自己在 CAS 成功登录过。

- **TGC：Ticket Granting Cookie：**CAS Server 生成 TGT 放入自己的 Session 中，而 TGC 就是这个 Session 的唯一标识（SessionId），以 Cookie 形式放到浏览器端，是 CAS Server 用来明确用户身份的凭证。
- **ST（Service Ticket）：**ST 是 CAS 为用户签发的访问某个 Service 的票据。

6. OAuth 2.0

在我们实际浏览网站的时候，当我们登录的时候除了输入当前网站的账号密码外，我们还发现可以通过第三方的 QQ 或者 微信登录，那么这又是如何做到了呢，这就要谈到 **OAuth** 了。

OAuth 协议又有 1.0 和 2.0 两个版本，2.0 版整个授权验证流程更简单更安全，也是目前最主要的用户身份验证和授权方式。

6.1 什么是 OAuth 2.0?

OAuth 是一个开放标准，允许用户授权第三方网站 (CSDN、思否等) 访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方网站；

常见的提供 OAuth 认证服务的厂商：`支付宝、QQ、微信、微博`

简单说，OAuth 就是一种授权机制。数据的所有者告诉系统，同意授权第三方应用进入系统，获取这些数据。系统从而产生一个短期的进入令牌（Token），用来代替密码，供第三方应用使用。

令牌与密码的差异：

`令牌（Token）`与`密码（Password）`的作用是一样的，都可以进入系统，但是有三点差异。

1. **令牌是短期的，到期会自动失效**：用户自己无法修改。密码一般长期有效，用户不修改，就不会发生变化。
2. **令牌可以被数据所有者撤销，会立即失效。**
3. **令牌有权限范围（scope）**：对于网络服务来说，只读令牌就比读写令牌更安全。密码一般是完整权限。

OAuth 2.0 对于如何颁发令牌的细节，规定得非常详细。具体来说，一共分成**四种授权**模式（**Authorization Grant**），适用于不同的互联网场景。

无论哪个模式都拥有三个必要角色：客户端、授权服务器、资源服务器，有的还有用户（资源拥有者），下面简单介绍下四种授权模式。

6.2 授权码模式

授权码（Authorization Code Grant)方式，指的是第三方应用先申请一个授权码，然后再用该码获取令牌。

这种方式是最常用的流程，安全性也最高，它适用于那些**有后端服务**的 Web 应用。授权码通过前端传送，令牌则是储存在后端，而且所有与资源服务器的通信都在后端完成。这样的前后端分离，可以避免令牌泄漏。

一句话概括：客户端换取授权码，客户端使用授权码换 token，客户端使用 token 访问资源

授权码模式的步骤详解

1. **客户端：**

   打开网站 A，点击登录按钮，请求 A 服务，A 服务重定向 (重定向地址如下) 至授权服务器 (如 QQ、微信授权服务)。

```
https://qq.com/oauth/authorize?
  response_type=code&
  client_id=CLIENT_ID&
  redirect_uri=CALLBACK_URL&
  scope=read
```

上面 URL 中，`response_type` 参数表示要求返回授权码（`code`），`client_id` 参数让 B 知道是谁在请求，`redirect_uri` 参数是 B 接受或拒绝请求后的跳转网址，`scope` 参数表示要求的授权范围（这里是只读）

2. **授权服务器：**

   授权服务网站会要求用户登录，然后询问是否同意给予 A 网站授权。用户表示同意，这时授权服务网站就会跳回 `redirect_uri` 参数指定的网址。跳转时，会传回一个授权码，就像下面这样。

   ```
   https://a.com/callback?code=AUTHORIZATION_CODE
   ```

   上面 URL 中，`code` 参数就是授权码。

3. **网站 A 服务器：**

   拿到授权码以后，就可以向授权服务器 (qq.com)请求令牌，请求地址如下：

   ```
   https://qq.com/oauth/token?
    client_id=CLIENT_ID&
    client_secret=CLIENT_SECRET&
    grant_type=authorization_code&
    code=AUTHORIZATION_CODE&
    redirect_uri=CALLBACK_URL
   ```

   上面 URL 中，`client_id` 参数和 `client_secret` 参数用来让授权服务器 确认 A 的身份（`client_secret` 参数是保密的，因此只能在后端发请求），`grant_type` 参数的值是 `AUTHORIZATION_CODE`，

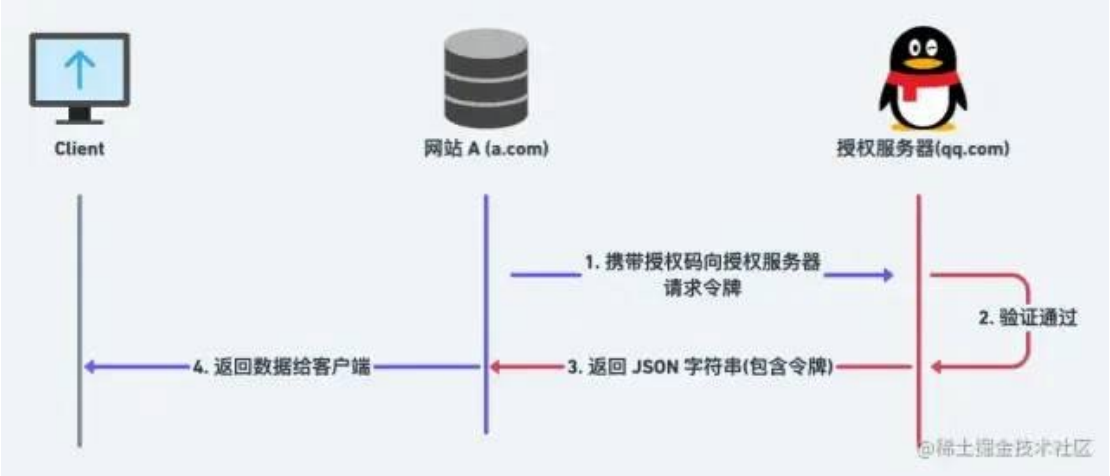
表示采用的授权方式是授权码，`code` 参数是上一步拿到的授权码，`redirect_uri` 参数是令牌颁发后的回调网址。

4. **授权服务器：**

   收到请求以后，验证通过，就会颁发令牌。具体做法是向 `redirect_uri` 指定的网址，发送一段 JSON 数据。

```
{
  "access_token":"ACCESS_TOKEN",
  "token_type":"bearer",
  "expires_in":2592000,
  "refresh_token":"REFRESH_TOKEN",
  "scope":"read",
  "uid":100101,
  "info":{...}
}
```

   上面 JSON 数据中，`access_token` 字段就是令牌，A 网站在后端拿到了，然后返回给客户端即可。



6.3 隐藏式模式（Implicit Grant）

有些 Web 应用是纯前端应用，没有后端。这时就不能用上面的方式了，必须将令牌储存在前端。**OAuth2.0 就规定了第二种方式，允许直接向前端颁发令牌。这种方式没有授权码这个中间步骤，所以称为（授权码）"隐藏式"（implicit）。**

一句话概括：客户端让用户登录授权服务器换 token，客户端使用 token 访问资源。

隐藏式模式的步骤详解

1. **客户端：**

   打开网站 A，A 网站提供一个链接，要求用户跳转到授权服务器，授权用户数据给 A 网站使用。如下链接：

   ```
   https://qq.com/oauth/authorize?
     response_type=token&
     client_id=CLIENT_ID&
     redirect_uri=CALLBACK_URL&
     scope=read
   ```

   上面 URL 中，response_type 参数为 token，表示要求直接返回令牌。

2. **授权服务器：**

   用户跳转到授权服务器，登录后同意给予 A 网站授权。这时，授权服务器就会跳回 redirect_uri 参数指定的跳转网址，并且把令牌作为 URL 参数，传给 A 网站。

   ```
   https://a.com/callback#token=ACCESS_TOKEN
   ```

   上面 URL 中，token 参数就是令牌，A 网站因此直接在前端拿到令牌。

注意：

1. 令牌的位置是 URL 锚点（fragment），而不是查询字符串（querystring），这是因为 OAuth 2.0 允许跳转网址是 HTTP 协议，因此存在"中间人攻击"的风险，而浏览器跳转时，锚点不会发到服务器，就减少了泄漏令牌的风险。
2. 这种方式把令牌直接传给前端，是很不安全的。因此，只能用于一些安全要求不高的场景，并且令牌的有效期必须非常短，通常就是会话期间（session）有效，浏览器关掉，令牌就失效了。

6.4 用户名密码式模式（Password Credentials Grant）

如果你高度信任某个应用，OAuth 2.0 也允许用户把用户名和密码，直接告诉该应用。该应用就使用你的密码，申请令牌，这种方式称为"密码式"（password）。

一句话概括：用户在客户端提交账号密码换 token，客户端使用 token 访问资源。

密码式模式的步骤详解

1. **客户端：**

   A 网站要求用户提供`授权服务器(qq.com)`的用户名和密码。拿到以后，A 就直接向`授权服务器`请求令牌。

   ```
   https://oauth.b.com/token?
     grant_type=password&
     username=USERNAME&
     password=PASSWORD&
     client_id=CLIENT_ID
   ```

   上面 URL 中，`grant_type`参数是授权方式，这里的`password`表示"密码式"，`username`和`password`是`授权服务器`的用户名和密码。

2. **授权服务器：**

   `授权服务器`验证身份通过后，直接给出令牌。

   注意，这时不需要跳转，而是把令牌放在 JSON 数据里面，作为 HTTP 回应，A 网站因此拿到令牌。

这种方式需要用户给出自己的用户名/密码，显然风险很大，因此只适用于其他授权方式都无法采用的情况，而且必须是用户高度信任的应用。

6.5 客户端模式（Client Credentials Grant）

客户端模式指客户端以自己的名义，而不是以用户的名义，向`授权服务器`进行认证。

主要适用于没有前端的命令行应用。

一句话概括：客户端使用自己的标识换 token，客户端使用 token 访问资源。

客户端模式的步骤详解

1. 客户端：

   客户端向授权服务器进行身份认证，并要求一个访问令牌。请求链接地址：

   ```
   https://oauth.b.com/token?
     grant_type=client_credentials&
     client_id=CLIENT_ID&
     client_secret=CLIENT_SECRET
   ```

   上面 URL 中，`grant_type` 参数等于 `client_credentials` 表示采用凭证式，`client_id` 和 `client_secret` 用来让授权服务器确认 A 的身份。

2. 授权服务器：

   授权服务器验证通过以后，直接返回令牌。

注意：这种方式给出的令牌，是针对第三方应用的，而不是针对用户的，即有可能多个用户共享同一个令牌。

6.6 授权模式选型

按授权需要的多端情况：

模式	需要前端	需要后端	需要用户响应	需要客户端密钥
授权码模式 Authorization Code	✅	✅	✅	✅
隐式授权模式 Implicit Grant	✅	❌	✅	❌
密码授权模式 Password Grant	✅	✅	✅	✅
客户端授权模式 Client Credentials	❌	✅	❌	✅

按照客户端类型与访问令牌所有者分类：

上述主要比较浅显的讲解了 OAuth2.0 的基本逻辑，如若想详细深入的了解，可查看官方文档 OAuth[5]或 RFC 6749[6]；亦可查看 OAuth 2.0 概念及授权流程梳理[7]做对比

7. 联合登录和信任登录

7.1 什么是联合登陆

联合登录指同时包含多种凭证校验的登录服务，同时，也可以理解为使用第三方凭证进行校验的登录服务。

通俗点讲：对于两个网站 A 和 B，在登录 A 网站的时候用 B 网站的帐号密码，就是联合登录，或者登录 B 网站的时候使用 A 网站的帐号密码，也是联合登录。

这样的概念其实与上面所讲的 OAuth2.0 的用户名密码式模式认证方式类似。

最经典的莫过于 APP 内嵌 H5 的使用场景，当用户从 APP 进入内嵌的 H5 时，我们希望 APP 内已登录的用户能够访问到 H5 内受限的资源，而未登录的用户则需要登录后访问。

这里思路主要有两种，一种是原生跳转内嵌 H5 页面时，将登录态 Token 附加在 URL 参数上，另一种则是内嵌 H5 主动通过与原生客户端制定的协议获取应用内的登录状态。

7.2 什么是信任登录

信任登录是指所有不需要用户主动参与的登录，例如建立在私有设备与用户之间的绑定关系，凭证就是私有设备的信息，此时不需要用户再提供额外的凭证。信任登录又指用第三方比较成熟的用户库来校验凭证，并登录当前访问的网站。

通俗点讲：在 A 网站有登录状态的时候，可以直接跳转到 B 网站而不用登录，就是信任登录。

目前比较常见的第三方信任登录帐号如：QQ 号淘宝帐号、支付宝帐号、微博帐号等。

我们不难发现 OAtuth 2.0 其实就是信任登录的缩影，因为正是有了 OAuth，我们的信任登录才得以实现。

8. 唯一登录

— 假设现在产品经理提一个需求：**我想要实现用户只能在一个设备上登录，禁止用户重复登录；**

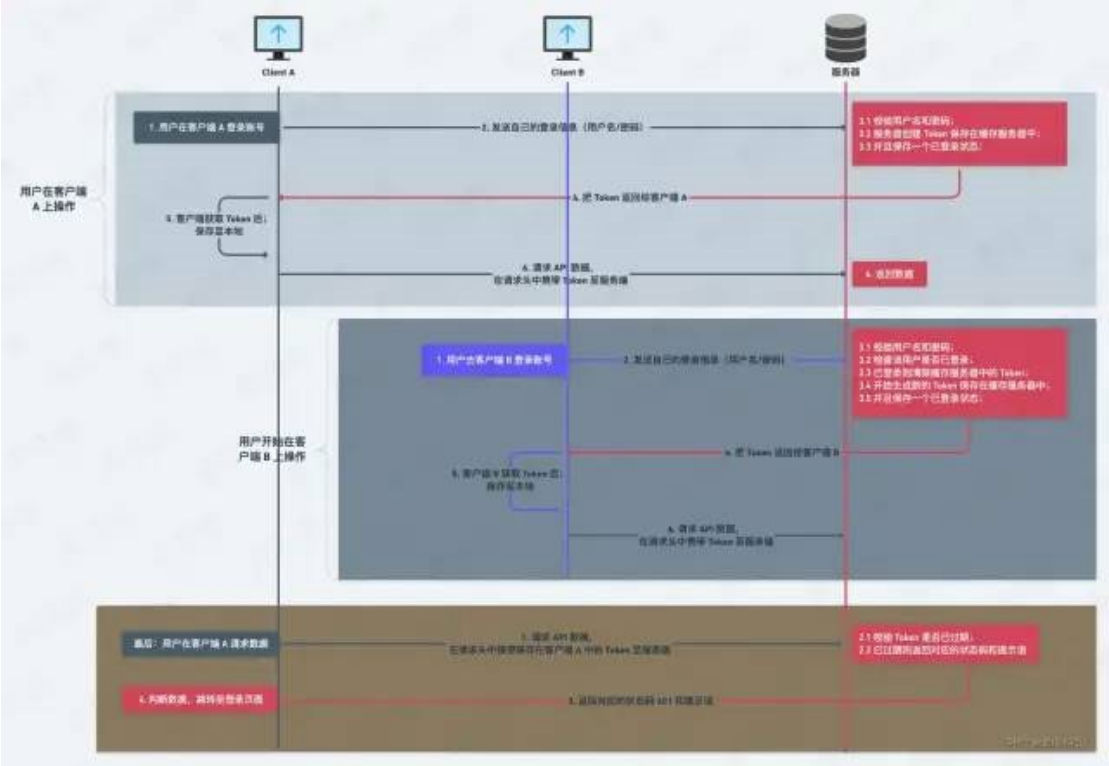
— 身为优秀的程序员的我们当然是满足他啦 ！！

8.1 什么是唯一登录

唯一登录，指的是**禁止多人同时登录同一账号，后者的登录行为，会导致前者掉线。**

通俗点讲就是：A 账号在 A 电脑上登录后，A 账号此时又用 B 电脑再次登录，则 A 电脑请求页面时，提示“重新登录”的信息，并跳转到登录页面

8.2 唯一登录流程图



8.3 唯一登录步骤详解

用户在客户端 A 操作：

1. 输入账号请求登录接口；
2. 后端生成对应 Token 并且返回给客户端 A，并且在服务端保存一个登录状态；
3. 客户端 A 保存 Token，并且每次请求都在 header 头中携带对应的 Token；

用户在客户端 B 操作：

突然用户在客户端 B 上开始登录操作，我们会发现，步骤和在客户端 A 上面的操作几乎是一致的；

只是后端在生成新的 Token 时，要先验证登录状态，然后再生成对应新的 Token；

9. 扫码登录

9.1 什么是扫码登录

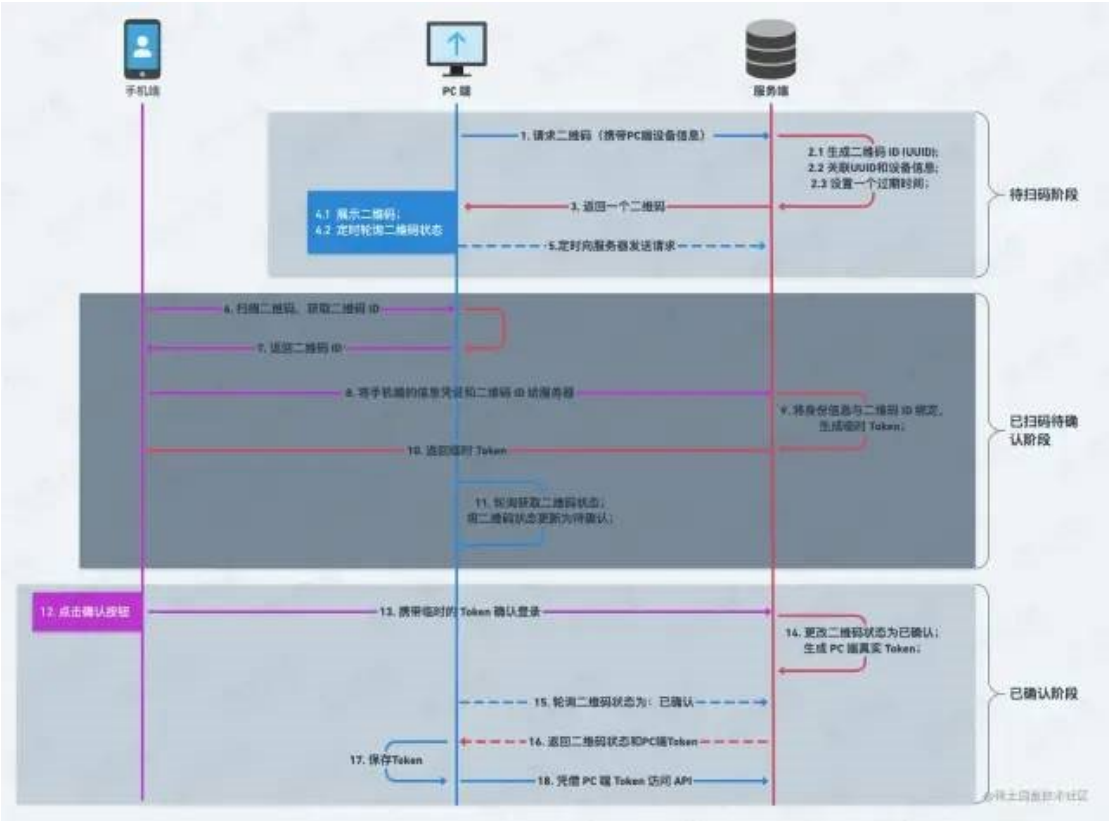
扫码登录通常见于移动端 APP 中，很多 PC 端的网站都提供了扫码登录的功能，无需在网页上输入任何账号和密码，只需要让移动端 APP (如微信、淘宝、QQ 等等) 中已登录用户主动扫描二维码，再确认登录，以使 PC 端的同款应用得以快速登录的方式就是扫码登录。

9.2 什么是二维码

二维码又称二维条码，常见的二维码为 QR Code，QR 全称 Quick Response，是一个近几年来移动设备上超流行的一种编码方式，它比传统的 Bar Code 条形码能存更多的信息，也能表示更多的数据类型。

通过上面所述，我们不难发现，扫码登录需要三端 (PC 端、手机端、服务端) 来进行配合才能达到登录成功的效果；

9.3 扫码登录的认证流程图



9.4 扫码登录的步骤详解 (待扫码阶段、待确认阶段、已确认阶段)

待扫码阶段：

1. **PC 端：**

   打开某个网站 (如 taobao.com) 或者某个 APP (如微信) 的扫码登录入口；就会携带 PC 端的设备信息向服务端发送一个获取二维码的请求；

2. **服务端：**

   服务器收到请求后，随机生成一个 UUID 作为二维码 ID，并将 UUID 与 PC 端的设备信息关联起来存储在 Redis 服务器中，然后返回给 PC 端；同时设置一个过期时间，在过期后，用户登录二维码需要进行刷新重新获取。

3. **PC 端：**

   收到二维码 ID 之后，将二维码 ID 以二维码的形式展示，等待移动端扫码。并且此时的 PC 端开始轮询查询二维码状态，直到登录成功。

   如果移动端未扫描，那么一段时间后二维码会自动失效。

已扫码待确认阶段：

1. **手机端：**

   打开手机端对应**已登录**的 APP (微信或淘宝等)，开始扫描识别 PC 端展示的二维码；

   移动端扫描二维码后，会自动获取到二维码 ID，并将移动端登录的信息凭证（Token）和二维码 ID 作为参数发送给服务端，

此时手机必须是已登录（使用扫描登录的前提是移动端的应用为已登录状态，这样才可以共享登录态）。

2. **服务端：**

   收到手机端发来的请求后，会将 `Token 与二维码 ID` 关联，为什么需要关联呢？因为，当我们在使用微信时，移动端退出时，PC 端也应该随之退出登录，这个关联就起到这个作用。然后会生成一个临时 Token，这个 Token 会返回给移动端，一次性 Token 用作确认时的凭证。

已确认阶段：

1. **手机端：**

   收到确认信息后，点击确认按钮，移动端携带上一步中获取的`临时 Token` 发送给服务端校验；

2. **服务端：**

   服务端校验完成后，会更新二维码状态，并且给 PC 端生成一个`正式的 Token`，后续 PC 端就是持有这个 Token 访问服务端。

3. **PC 端：**

   轮询到二维码状态为已登录状态，并且会获取到了生成的 Token，完成登录，后续访问都基于 Token 完成。

10. 一键登录（适用于原生 APP）

10.1 账号密码登录

大家都知道，最传统的登录方式就是使用**账号加密码登录**，简单粗暴，一般也不会出现什么问题；

缺点：

1. 但这种方式要求用户要记住自己的账号和密码，也就是有一个记忆成本。用户为了降低记忆成本，很可能会在不同平台使用同一套账号密码。从安全角度考虑，一旦某个平台的账号密码泄露了，会连累到该用户使用的其他平台。
2. 另外，由于账号和个人身份无关，意味着同一个用户可以注册多个不同的账号，也就是可能会有恶意注册的情况发生。

直到手机卡的强制实名制才得以解决！

10.2 手机号验证码登录

随着无线互联的发展以及手机卡实名制的推广，手机号俨然已成为特别的身份证明，与账号密码相比，手机号可以更好地验证用户的身份，防止恶意注册。

但是手机号注册还是需要一系列繁琐的操作：输入手机号、等待短信验证码、输入验证码、点击登录。整个流程少说二十秒，而且如果收不到短信，也就登录补了，这类问题有可能导致潜在的用户流失。

从安全角度考虑，还存在验证码泄漏的风险。如果有人知道了你的手机号，并且窃取到了验证码，那他也能登录你的账号了。

所以就有了一键登录操作！

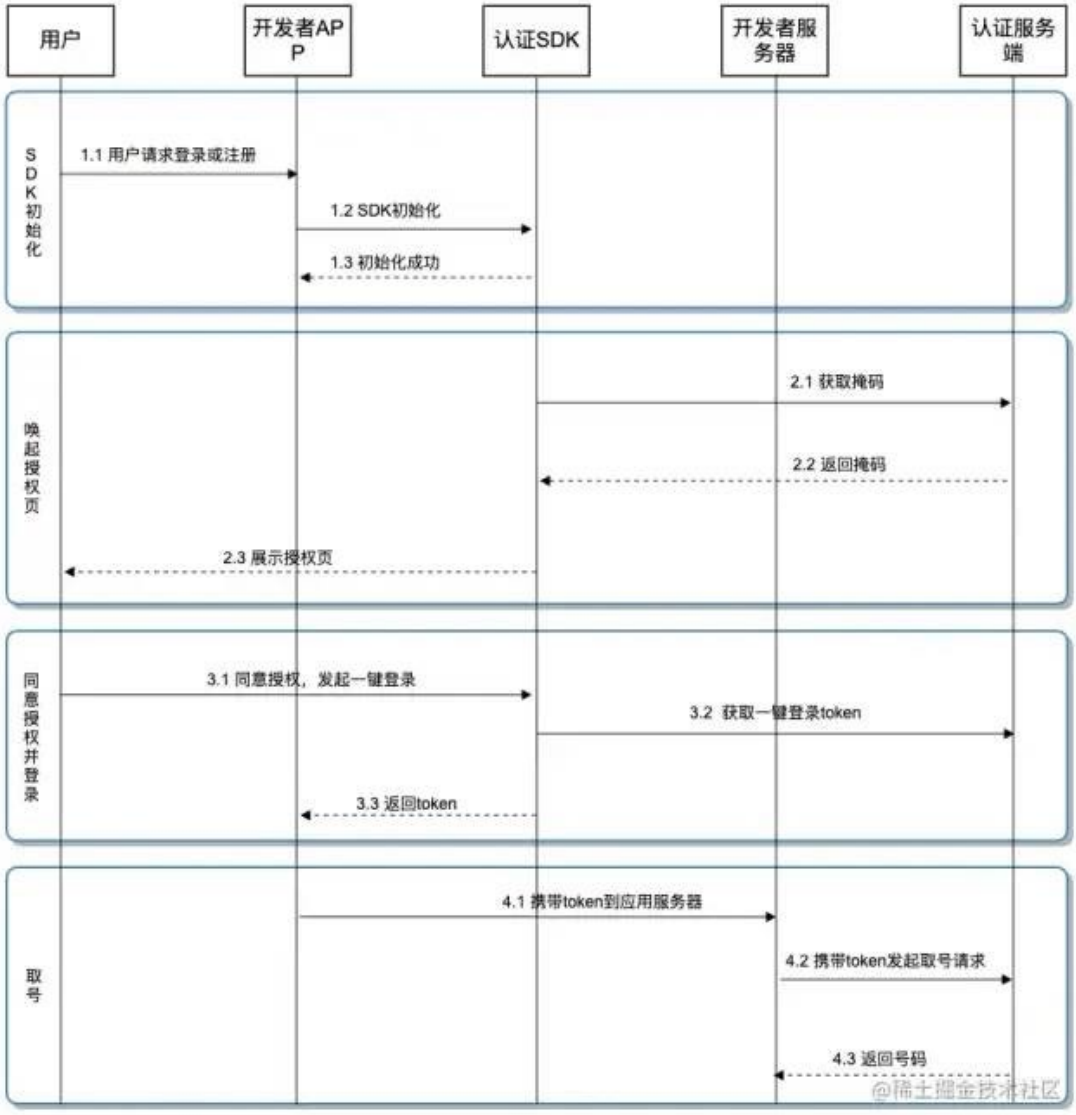
10.3 什么是一键登录

我们想一下，为什么我们需要验证码？验证码的作用就是确定这个手机号是你的，那除了使用短信，是否还有别的方式对手机号进行认证？于是，就有了咱们的主角一键登录。

短信验证码的作用就是证明当前操作页面的用户与输入手机号的用户为相同的人，那么实际上只要我们能够获取到当前手机使用的手机卡号，直接使用这个号码进行登录，不需要额外的操作，这就是**一键登录**。

一键登录能不能做，取决于运营商是否开放相关服务；随着运营商开放了相关的服务，我们现在已经能够接入运营商提供的 SDK 并付费使用相关的服务。

一键登录流程图：



一键登录步骤详解：

1. **SDK 初始化：**调用 SDK 方法，传入平台配置的 AppKey 和 AppSecret
2. **唤起授权页：**调用 SDK 唤起授权接口，SDK 会先向运营商发起获取手机号掩码的请求，请求成功后跳到授权页。授权页会显示手机号掩码以及运营商协议给用户确认。
3. **同意授权并登录：**用户同意相关协议，点击授权页面的登录按钮，SDK 会请求本次取号的 Token，请求成功后将 Token 返回给客户端
4. **取号：**将获取到的 Token 发送到自己的服务器，由服务端携带 Token 调用运营商一键登录的接口，调用成功就返回手机号码。服务端用手机号进行登录或注册操作，返回操作结果给客户端，完成一键登录。

三大运营商开放平台：

- 移动 - 互联网能力开放平台[8]
- 电信 - 天翼账号开放平台[9]
- 联通 - WO+ 开放平台[10]

  由于国内三大运营商各自有独立的 SDK，所以会导致兼容方面的工作会特别繁琐。如果要采用一键登录的方案，不妨采用第三方提供了号码认证服务，下列几家供应商都拥有手机号码认证能力：

  - 阿里 - 号码认证服务[11]

- 创蓝 - 闪验[12]
- 极光 - 极光认证[13]
- mob - 秒验[14]

注意：

在认证过程中，需要用户打开蜂窝网络，如果手机设备没有插入 SIM 卡、或者关闭蜂窝网络的情况下，是无法完成认证的。所以就算接入一键登录，还是要兼容传统的登录方式，允许用户在失败的情况下，仍能正常完成登录流程。

总结

在学习了解上面的 10 种鉴权方法后，我们简单概括一下

- `HTTP 基本认证`适用于内部网络，或者对安全要求不是很高的网络；
- `Session-Cookie`适用于一般中大型的网站（移动端 APP 除外）；
- `Token`和`JWT`都适用于市面上大部分的企业型网站，JWT 效能会优于 Token；
- `单点登录`适用于子系统较多的大型企业网站；
- `OAuth 2.0`适用于需要快速注册用户型的网站；
- `扫码登录`适用于已完成部署了三端的企业；
- `一键登录`适用于原生 APP；

参考资料

https://juejin.cn/post/7129298214959710244

https://juejin.cn/post/6898630134530752520

[1] express-session: https://github.com/expressjs/session
[2] koa-session: https://github.com/koajs/session
[3] express-jwt: https://github.com/auth0/express-jwt
[4] koa-jwt: https://github.com/koajs/jwt
[5] OAuth:https://en.wikipedia.org/wiki/OAuth
[6] RFC 6749:http://www.rfcreader.com/#rfc6749
[7] OAuth 2.0 概念及授权流程梳理:https://www.cnblogs.com/hellxz/p/oauth2_process.html
[8] 移动 - 互联网能力开放平台:http://dev.10086.cn/
[9] 电信 - 天翼账号开放平台: https://id.189.cn/
[10] 联通 - WO+ 开放平台:http://open.wo.com.cn/
[11] 阿里 - 号码认证服务:https://help.aliyun.com/product/75010.html
[12] 创蓝 - 闪验:http://shanyan.253.com/
[13] 极光 - 极光认证:https://www.jiguang.cn/identify
[14] mob - 秒验:https://www.mob.com/mobService/secverify