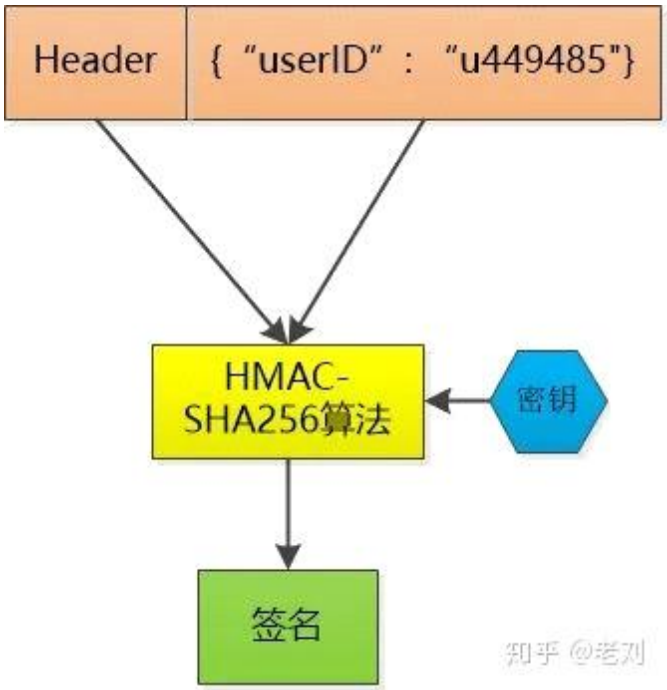
cookie session token

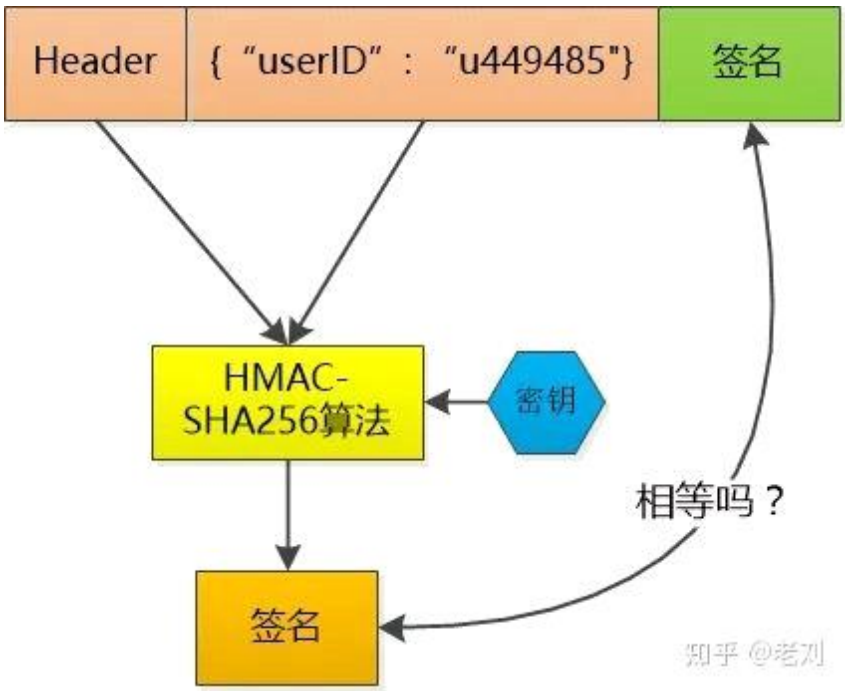
2020年2月12日 星期三　　下午7:14

发展史

1. 很久很久以前，Web 基本上就是文档的浏览而已， 既然是浏览，作为服务器， 不需要记录谁在某一段时间里都浏览了什么文档，每次请求都是一个新的HTTP协议， 就是请求加响应， 尤其是我不用记住是谁刚刚发了HTTP请求， 每个请求对我来说都是全新的。
2. 但是随着交互式Web应用的兴起，像在线购物网站，需要登录的网站等等，马上就面临一个问题，那就是要管理会话，必须记住哪些人登录系统， 哪些人往自己的购物车中放商品， 也就是说我必须把每个人区分开，这就是一个不小的挑战，因为HTTP请求是无状态的，所以想出的办法就是给大家发一个会话标识(session id), 说白了就是一个随机的字串，每个人收到的都不一样， 每次大家向我发起HTTP请求的时候，把这个字符串给一并捎过来， 这样我就能区分开谁是谁了
3. 这样大家很嗨皮了，可是服务器就不嗨皮了，每个人只需要保存自己的session id，而服务器要保存所有人的session id！ 如果访问服务器多了， 就得由成千上万，甚至几十万个
   这对服务器说是一个巨大的开销， 严重的限制了服务器扩展能力， 比如说我用两个机器组成了一个集群， 小F通过机器A登录了系统， 那session id会保存在机器A上， 假设小F的下一次请求被转发到机器B怎么办？ 机器B可没有小F的 session id啊。
4. 于是有人就一直在思考， 我为什么要保存这可恶的session呢， 只让每个客户端去保存该多好?
   可是如果不保存这些session id，怎么验证客户端发给我的session id 的确是我生成的呢？ 如果不去验证，我们都不知道他们是不是合法登录的用户， 那些不怀好意的家伙们就可以伪造session id，为所欲为了。
   关键点就是验证！
   比如说， 小F已经登录了系统， 我给他发一个令牌(token)， 里边包含了小F的 user id， 下一次小F再次通过Http 请求访问我的时候， 把这个token 通过Http header 带过来不就可以了。
   不过这和session id没有本质区别啊， 任何人都可以可以伪造， 所以我得想点儿办法， 让别人伪造不了
   那就对数据做一个签名吧， 比如说我用HMAC-SHA256 算法，加上一个只有我才知道的密钥， 对数据做一个签名， 把这个签名和数据一起作为token， 由于密钥别人不知道， 就无法伪造token了。



这个token 我不保存， 当小F把这个token 给我发过来的时候，我再用同样的HMAC-SHA256 算法和同样的密钥，对数据再计算一次签名， 和token 中的签名做个比较， 如果相同， 我就知道小F已经登录过了，并且可以直接取到小F的user id , 如果不相同， 数据部分肯定被人篡改过， 我就告诉发送者： 对不起，没有认证。



Token 中的数据是明文保存的（虽然我会用Base64做下编码， 但那不是加密）， 还是可以被别人看到的， 所以我不能在其中保存像密码这样的敏感信息。

当然， 如果一个人的token 被别人偷走了， 那我也没办法， 我也会认为小偷就是合法用户， 这其实和一个人的session id 被别人偷走是一样的。

这样一来， 我就不保存session id 了， 我只是生成token , 然后验证token ， 我用我的CPU计算时间获取了我的session 存储空间！

Cookie

cookie 是一个非常具体的东西，指的就是浏览器里面能永久存储的一种数据，仅仅是浏览器实现的一种数据存储功能。

cookie由服务器生成，发送给浏览器，浏览器把cookie以键值对形式保存到某个目录下的文本文件内，下一次请求同一网站时会把该cookie发送给服务器。由于cookie是存在客户端上的，所以浏览器加入了一些限制确保cookie不会被恶意使用，同时不会占据太多磁盘空间，所以每个域的cookie数量是有限的。

在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。cookie的出现就是为了解决这个问题，第一次登录后服务器返回一些数据（cookie）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会自动的把上次请求存储的cookie数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前用户是哪个了。cookie存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4KB。因此使用cookie只能存储一些小量的数据。

Session

session 从字面上讲，就是会话。这个就类似于你和一个人交谈，你怎么知道当前和你交谈的是张三而不是李四呢？对方肯定有某种特征（长相等）表明他就是张三。

session 也是类似的道理，服务器要知道当前发请求给自己的是谁。为了做这种区分，服务器就要给每个客户端分配不同的“身份标识”，然后客户端每次向服务器发请求的时候，都带上这个“身份标识”，服务器就知道这个请求来自于谁了。至于客户端怎么保存这个“身份标识”，可以有很多种方式，对于浏览器客户端，大家都默认采用 cookie 的方式

sessio和cookie的作用有点类似，都是为了存储用户相关的信息。不同的是，cookie是存储在本地浏览器，而session存储在服务器

服务器使用session把用户的信息临时保存在了服务器上，用户离开网站后session会被销毁。这种用户信息存储方式相对cookie来说更安全，可是session有一个缺陷：如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。

Cookie和Session结合使用

web开发发展至今，cookie和session的使用已经出现了一些非常成熟的方案。在如今的市场或者企业里，一般有两种存储方式：

1、存储在服务端：通过cookie存储一个session_id，然后具体的数据则是保存在session中。如果用户已经登录，则服务器会在cookie中保存一个session_id，下次再次请求的时候，会把该session_id携带上来，服务器根据session_id在session库中获取用户的session数据。就能知道该用户到底是谁，以及之前保存的一些状态信息。这种专业术语叫做server side session。

2、将session数据加密，然后存储在cookie中。客户端发给服务器之后，服务器这里会解密。这种专业术

语叫做client side session。flask采用的就是这种方式，但是也可以替换成其他形式。

不要混淆 session 和 session 实现。

本来 session 是一个抽象概念，开发者为了实现中断和继续等操作，将 user agent 和 server 之间一对一的交互，抽象为“会话”，进而衍生出“会话状态”，也就是 session 的概念。
而 cookie 是一个实际存在的东西，http 协议中定义在 header 中的字段。可以认为是 session 的一种后端无状态实现。
而我们今天常说的 “session”，是为了绕开 cookie 的各种限制，通常借助 cookie 本身和后端存储实现的，一种更高级的会话状态实现。
所以 cookie 和 session，你可以认为是同一层次的概念，也可以认为是不同层次的概念。具体到实现，session 因为 session id 的存在，通常要借助 cookie 实现，但这并非必要，只能说是通用性较好的一种实现方案。

token 和session 的区别

session 和 oauth token并不矛盾，作为身份认证 token安全性比session好，因为每个请求都有签名还能防止监听以及重放攻击，而session就必须靠链路层来保障通讯安全了。如上所说，如果你需要实现有状态的会话，仍然可以增加session来在服务器端保存一些状态。

Token介绍

token和session的区别https://blog.csdn.net/mydistance/article/details/84545768

如果前端使用用户名/密码向服务端请求认证，服务端认证成功，那么在服务端会返回 Token 给前端。前端可以在每次请求的时候带上 Token 证明自己的合法地位

为什么要用Token

1. Token 完全由应用管理，所以它可以避开同源策略
2. Token 可以避免 CSRF 攻击(http://dwz.cn/7joLzx)
3. Token 可以是无状态的，可以在多个服务间共享

Token 是在服务端产生的。如果前端使用用户名/密码向服务端请求认证，服务端认证成功，那么在服务端会返回 Token 给前端。前端可以在每次请求的时候带上 Token 证明自己的合法地位。如果这个 Token 在服务端持久化（比如存入数据库），那它就是一个永久的身份令牌

Token需要设置日期吗

对于这个问题，我们不妨先看两个例子。一个例子是登录密码，一般要求定期改变密码，以防止泄漏，所以密码是有有效期的；另一个例子是安全证书。SSL 安全证书都有有效期，目的是为了解决吊销的问题。

然后新问题产生了，如果用户在正常操作的过程中，Token 过期失效了，要求用户重新登录……用户体验岂不是很糟糕。

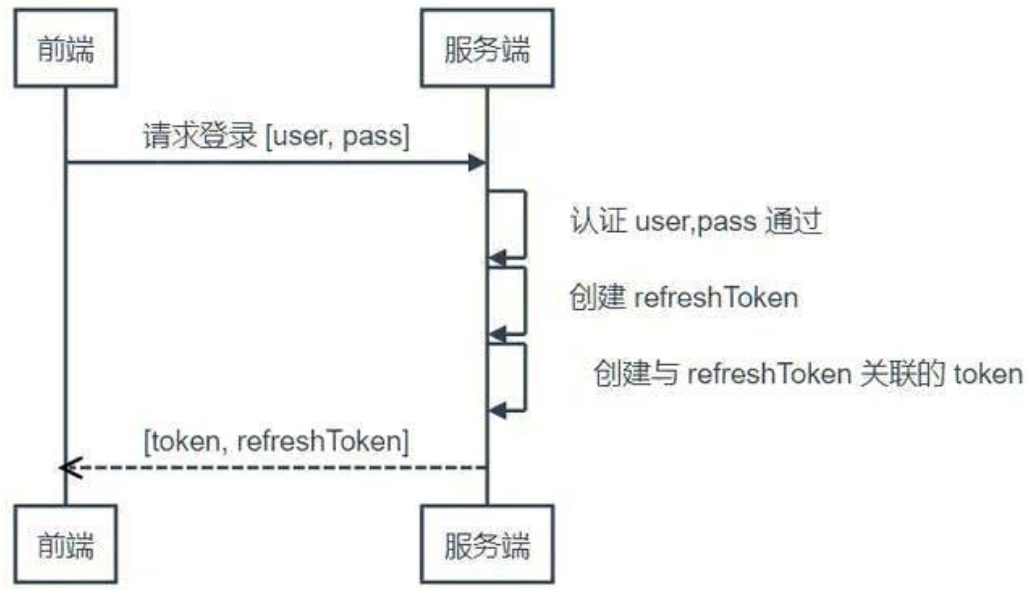
为了解决在操作过程不能让用户感到 Token 失效这个问题，有一种方案是在服务器端保存 Token 状态，用户每次操作都会自动刷新（推迟） Token 的过期时间——Session 就是采用这种策略来保持用户登录状态的。然而仍然存在这样一个问题，在前后端分离、单页 App 这些情况下，每秒种可能发起很多次请求，每次都去刷新过期时间会产生非常大的代价。如果 Token 的过期时间被持久化到数据库或文件，代价就更大了。所以通常为了提升效率，减少消耗，会把 Token 的过期时间保存在缓存或者内存中。

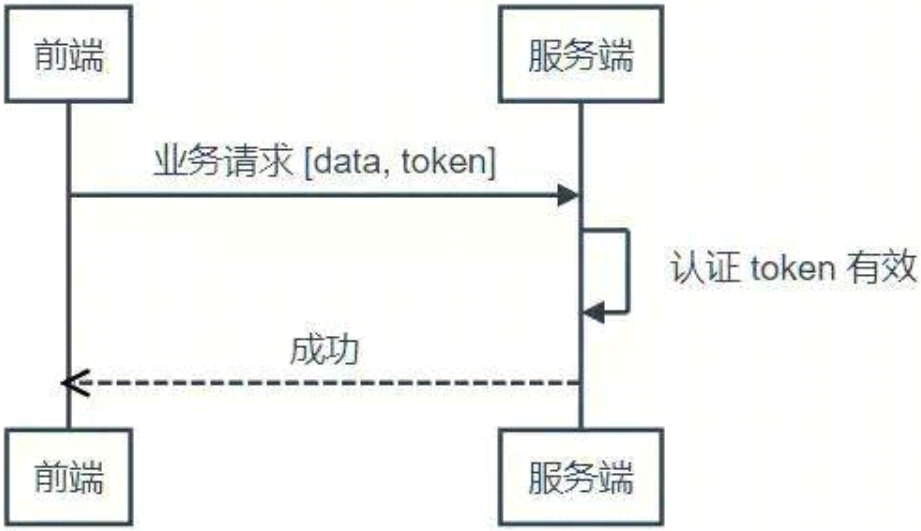
还有另一种方案，使用 Refresh Token，它可以避免频繁的读写操作。这种方案中，服务端不需要刷新 Token 的过期时间，一旦 Token 过期，就反馈给前端，前端使用 Refresh Token 申请一个全新 Token 继续使用。这种方案中，服务端只需要在客户端请求更新 Token 的时候对 Refresh Token 的有效性进行一次检查，大大减少了更新有效期的操作，也就避免了频繁读写。当然 Refresh Token 也是有有效期的，但是这个有效期就可以长一点了，比如，以天为单位的时间。

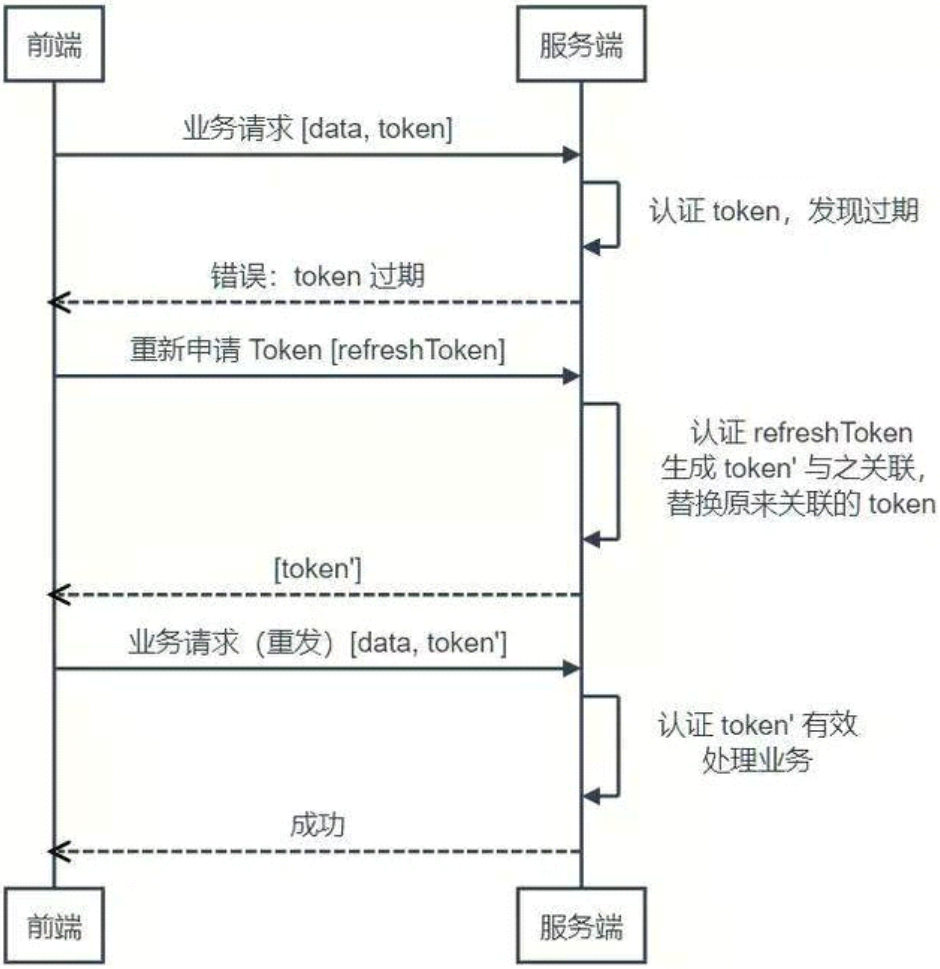
时序图：

使用 Token 和 Refresh Token 的时序图如下

登陆：







上面的时序图中并未提到 Refresh Token 过期怎么办。不过很显然，Refresh Token 既然已经过期，就该要求用户重新登录了

到目前为止，Token 都是有状态的，即在服务端需要保存并记录相关属性。

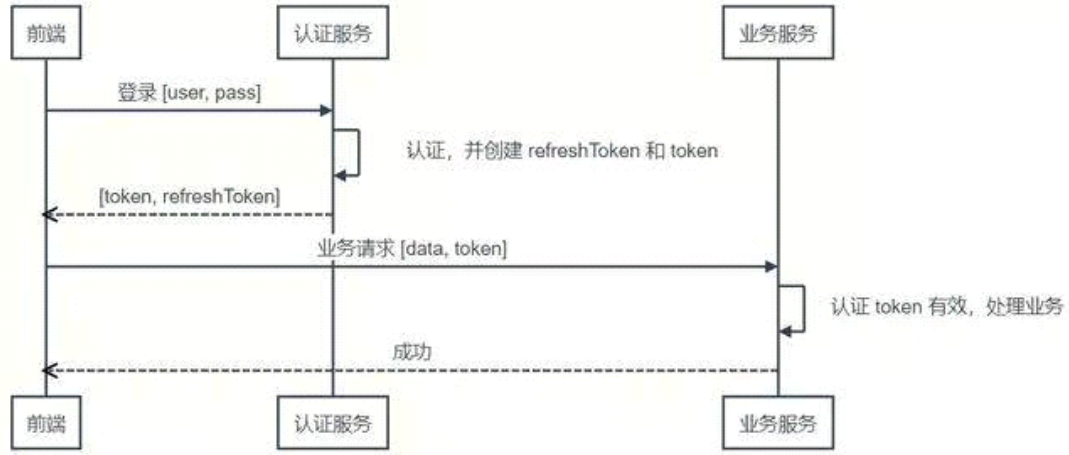
无状态Token

如果我们把所有状态信息都附加在 Token 上，服务器就可以不保存。但是服务端仍然需要认证 Token 有效。不过只要服务端能确认是自己签发的 Token，而且其信息未被改动过，那就可以认为 Token 有效——

“签名”可以作此保证。平时常说的签名都存在一方签发，另一方验证的情况，所以要使用非对称加密算法。但是在这里，签发和验证都是同一方，所以对称加密算法就能达到要求，而对称算法比非对称算法要快得多（可达数十倍差距）

更进一步思考，对称加密算法除了加密，还带有还原加密内容的功能，而这一功能在对 Token 签名时并无必要——既然不需要解密，摘要（散列）算法就会更快。可以指定密码的散列算法，自然是 HMAC

当 Token 无状态之后，单点登录就变得容易了。前端拿到一个有效的 Token，它就可以在任何同一体系的服务上认证通过——只要它们使用同样的密钥和算法来认证 Token 的有效性



当然，如果 Token 过期了，前端仍然需要去认证服务更新 Token