关于鉴权，看懂这篇就够了

刘妮萍: 微医前端技术部前端工程师。养鱼养花养狗，熬夜蹦迪喝酒。出走半生，归来仍是三年前端工作经验。

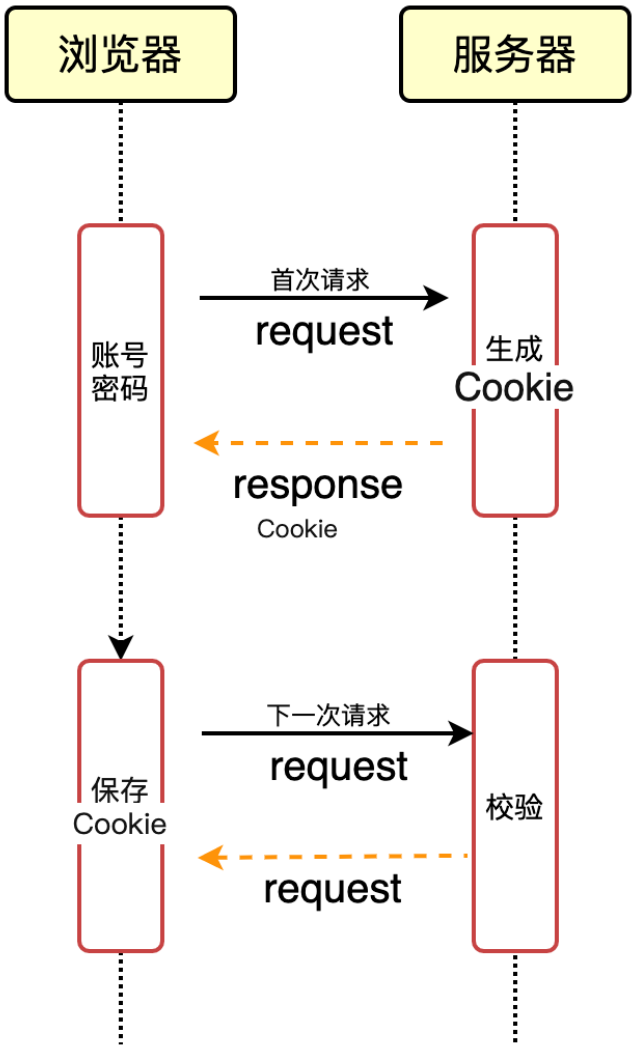
第一篇章

Cookie 的诞生及其特点

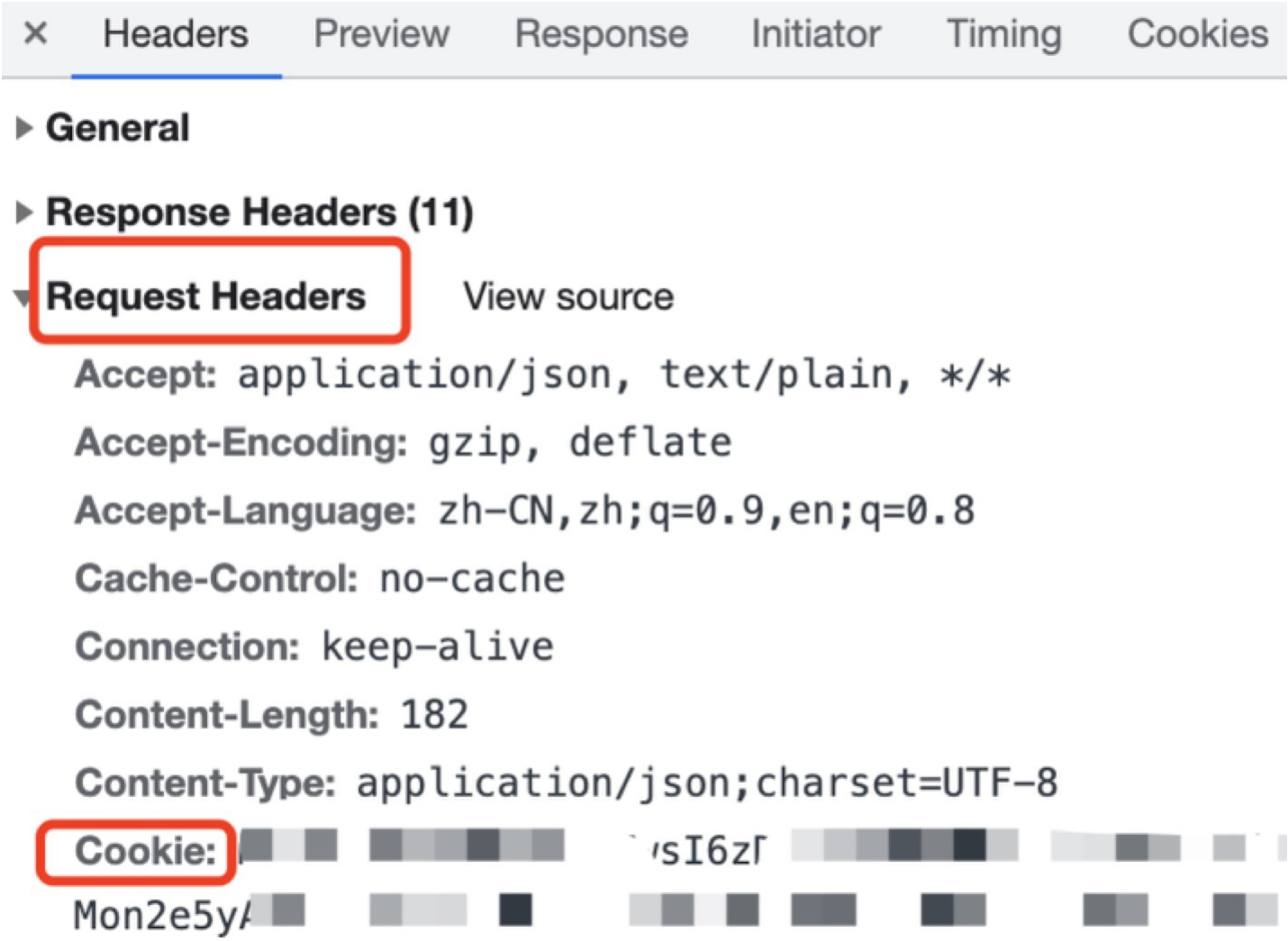
众所周知，web 服务器是**无状态**的，无状态的意思就是服务器不知道用户上一次请求做了什么，各请求之间是相互独立的，客户信息仅来自于**每次请求**时携带的，或是服务器自身保存的且可以被所有请求使用的公共信息。所以为了跟踪用户请求的状态信息，比如记录用户网上购物的购物车历史记录，Cookie 应运而生。
服务端在响应客户端请求的时候，会向客户端推送一个 Cookie，这个 Cookie 记录服务端上面的一些信息，客户端在后续的请求中携带这个 Cookie，服务端可以根据这个 Cookie 判断该请求的上下文关系。

Cookie 的出现，是无状态化向状态化过渡的一种手段。

以登录为例，用户输入账户名密码，发送请求到服务端，服务器生成 Cookie 后发送给浏览器，浏览器把 Cookie 以 k-v 的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。服务器校验该接收的 Cookie 与服务端的 Cookie 是否一致，不一致则验证失败。这是最初的设想。



在浏览器中存储的 Cookie 在下图所示位置



Cookie的原理决定了他有以下特点：
1，存储在客户端，可随意篡改，不安全
2，它的内容会随着 http 交互传接，影响性能，所以 Cookie 可存储的数据不能过大，最大为 4kb
3，一个浏览器对于一个网站只能存不超过 20 个 Cookie，而浏览器一般只允许存放 300 个 Cookie
4，移动端对 Cookie 支持不友好
5,一般情况下存储的是纯文本，对象需要序列化之后才可以存储，解析需要反序列化

二级域名之间的 Cookie 共享

还是以登录 Cookie 为例，比如现在有两个二级域名，http://a.xxx.com(域名 A)和http://b.xxx.com(域名 B)。那么域名 A 的登录 Cookie 在域名 B 下可以使用吗?
默认情况下，域名 A 服务主机中生成的 Cookie，只有域名 A 的服务器能拿到，其他域名是拿不到这个 Cookie 的，这就是**仅限主机Cookie。**

但是服务端可以通过显式地声明 Cookie 的 domian 来定义它的域，如上例子通过Set-Cookie将域名 A 的登录 Cookie 的 domain（域）设置成http://xxx.com（他们**共同的顶级域名**），path 设置成’/’，Set-Cookie： name=value;domain=xxx.com;path=’/’，那么域名 B 便可以读到。
在新的规范rfc6265 中，domain 的值**会忽略任何前导点**，也就是**xxx.com**和**.xxx.com**都可以在子域中使用。SSO(单点登录)也是依据这个原理实现的。
那比如现在又有两个域名，a.b.e.f.com.cn (域名 1)和c.d.e.f.com.cn (域名 2)，域名 2 想要读到域名 1 的 Cookie,域名 1 可以声明哪些 domain 呢？答案是.e.f.com.cn或.f.com.cn，浏览器不能接受 domian 为.com.cn 的 Cookie，因为 Cookie 域如果可以设置成后缀，那可就是峡谷大乱斗了。

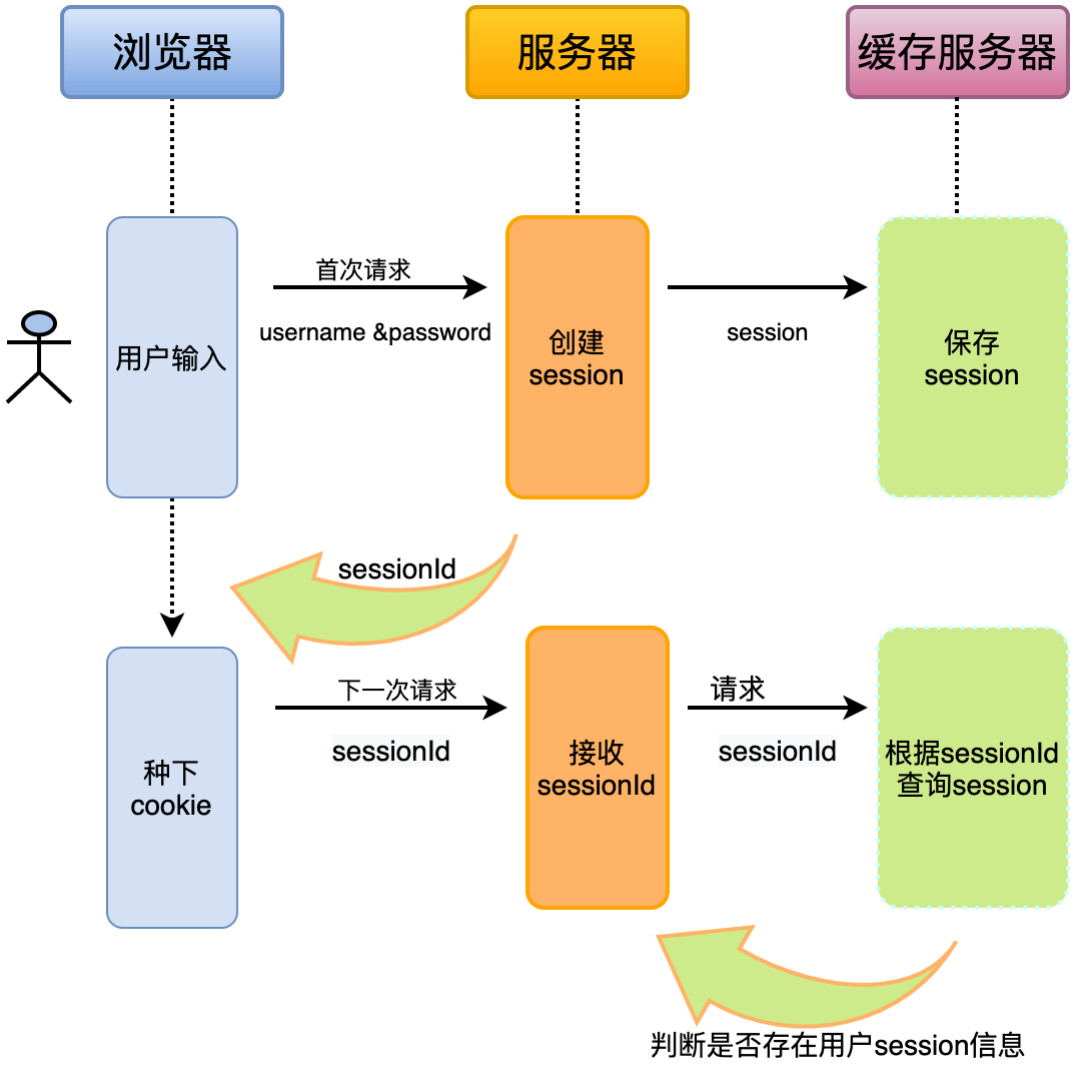
那如果域名 1 设置Set-Cookie： mykey=myvalue1;domain=e.f.com.cn;path=’/’
域名 2 设置Set-Cookie： mykey=myvalue2;domain=e.f.com.cn;path=’/’
那该域下 mykey 的值会被覆盖为 myvalue2，很好理解，同一个域下，Cookie 的 mykey 是唯一的。**通常，我们要通过设置正确的 domain 和 path，减少不必要的数据传输,节省带宽。**

Cookie-session 模式原理

随着交互式 Web 应用的兴起，Cookie 大小的限制以及浏览器对存储 Cookie 的数量限制，我们一定需要更强大的空间来储存大量的用户信息，比如我们这个网站是谁登录了，谁的购物车里加入了商品等等，服务器要保存千万甚至更多的用户的信息，Cookie 显然是不行的。那怎么办呢?
试想，我们在服务器端寻找一个空间存储所有用户会话的状态信息，并给每个用户分配不同的“身份标识”，也就是sessionId ，再将这个 sessionId 推送给浏览器客户端存储在 Cookie 中记录当前的状态，下次请求的时候只需要携带这个 sessionId，服务端就可以去那个空间搜索到该标识对应的用户。**这样做既能解决 Cookie 限制问题，又不用暴露用户信息到客户端，大大增加了实用性和安全性。

那将用户信息存储在哪呢？能否直接存在服务器中?
如果存在服务器中，1、这对服务器说是一个巨大的开销，严重的限制了服务器的扩展能力。2、假设 web 服务器做了负载均衡，用户 user1 通过机器 A 登入该系统，那么下一个请求如果被转发到另一台机器 B 上，机器 B 上是没有存该用户信息的，所以也找不到 sessionId，因此 sessionId 不应该存储在服务器上。这个时候**redis/Memcached**便出来解决该问题了,可以简单的理解它们为一个**缓存数据库。**
当我们把 sessionId 集中存储到一个独立的缓存服务器上，所有的机器根据 sessionId 到这个缓存系统里去获取用户信息和认证。那么问题就迎刃而解了。

Cookie-session 工作原理流程图



根据其工作原理，你有没有发现这个方式会存在一个什么样的问题？
那就是增加了单点登录失败的可能性，如果负责 session 的机器挂了， 那整个登录也就挂了。但是一般在项目里，负责 session 的机器也是有多台机器的集群进行负载均衡，增加可靠性。

思考：
假如服务器重启的话，用户信息会丢失吗？
Redis 等缓存服务器也是有个集群的，假设某一台服务重启了，会从其他运行的服务器中查找用户信息，那假设真的某一次所有服务器全都崩溃了，怎么办呢？大概的应对策略就是，存储在内存中的用户信息会定期刷到主机硬盘中以持久化数据，即便丢失，也只会丢失重启的那几分钟内的用户数据。

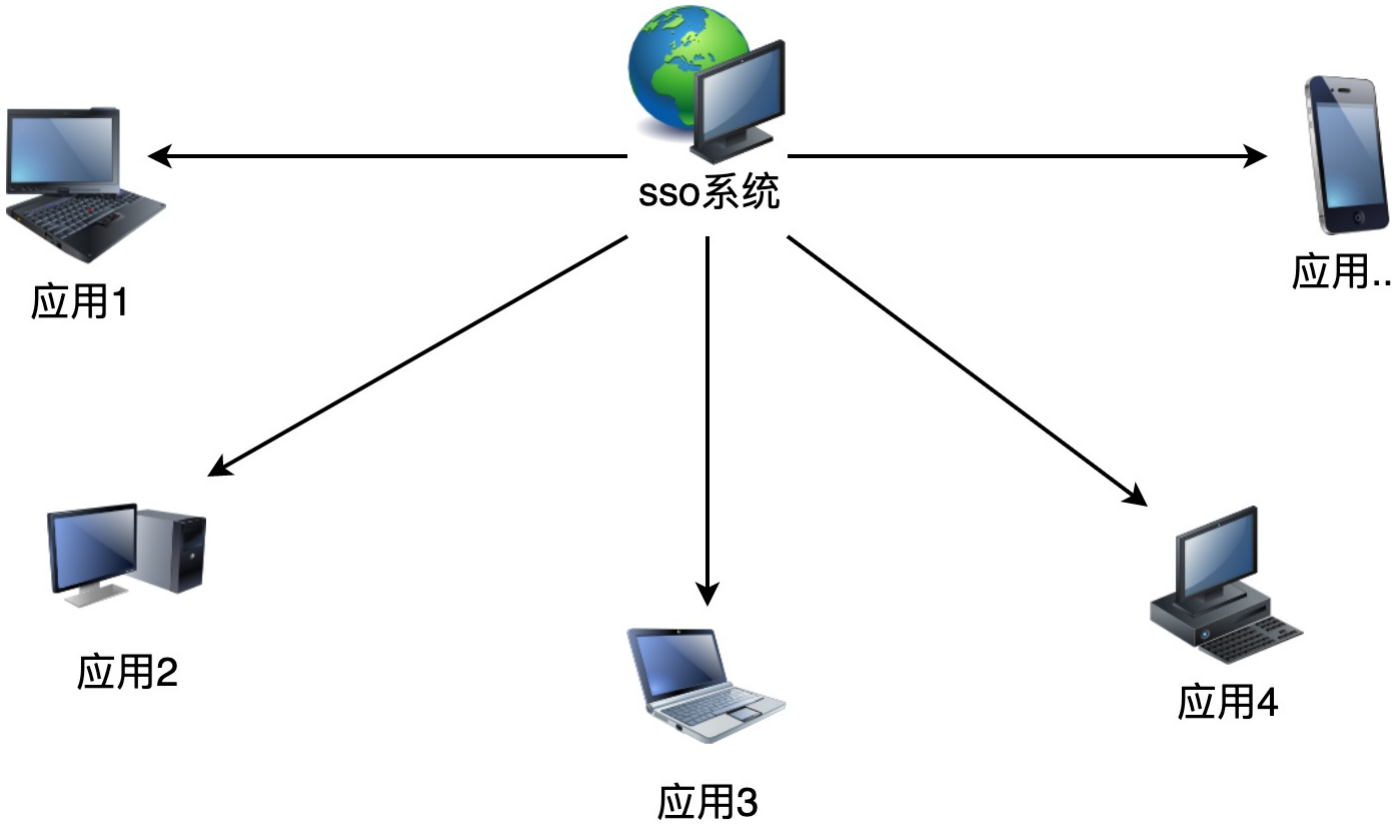
Cookie-session 局限性

1、依赖 Cookie，用户可以在浏览器端禁用 Cookie
2、不支持跨端兼容 app 等
3、业务系统不停的请求缓存服务器查找用户信息，使得内存开销增加，服务器压力过大
4、服务器是有状态的，如果是没有缓存服务器的方式，扩容就非常困难，需要在多台服务器中疯狂复制 sessionId
5、存在单点登录失败的可能性

第二篇章

SSO(单点登录)三种类型

单点登录（Single Sign On），简称为 SSO。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，单点登录就是用来解决这个问题的，**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。**

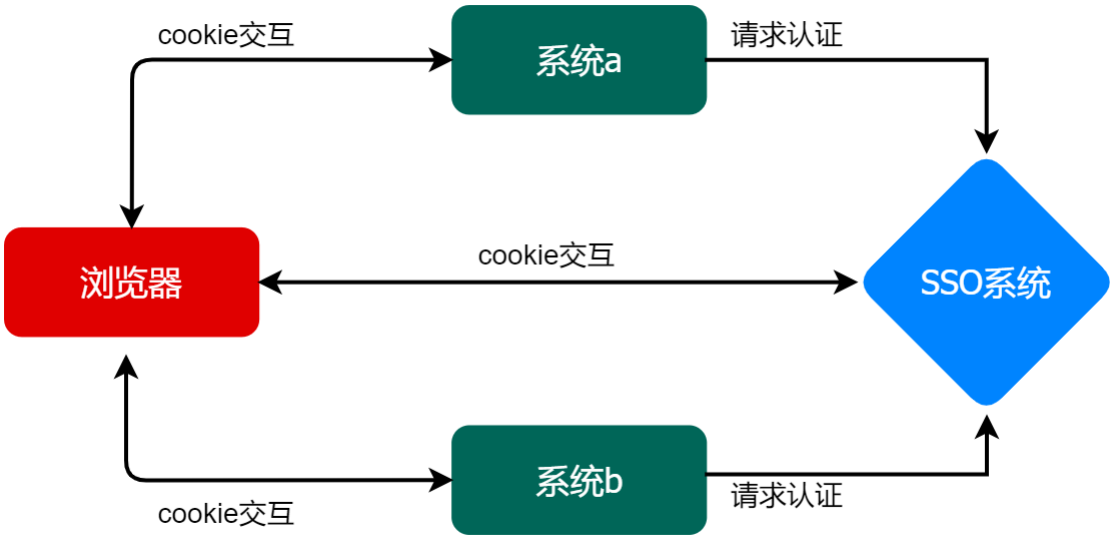


之前我们说过，单点登录是基于 cookie 同顶域共享的，那按照不同的情况可分为以下 3 种类型。
1、同一个站点下；
2、系统在相同的顶级域名下；
3、各子系统属于不同的顶级域名

一般情况下一个企业有一个顶级域名，前面讲过了，同一个站点和相同顶级域下的单点登录是利用了 Cookie 顶域共享的特性，相信大家已经明白这个流程，不再赘述。但如果是不同域呢？不同域之间 Cookie 是不共享的，怎么办？

CAS（中央认证服务）原理

这里我们就要说一说 CAS（中央认证服务）流程了，这个流程是单点登录的标准流程。它借助一个单独的系统专门做认证用，以下成为SSO系统。
它的流程其实跟 Cookie-session 模式是一样的，单点登录等于说是每个子系统都拥有一套完整的 Cookie-session 模式，再加上一套 Cookie-session 模式的 SSO 系统。



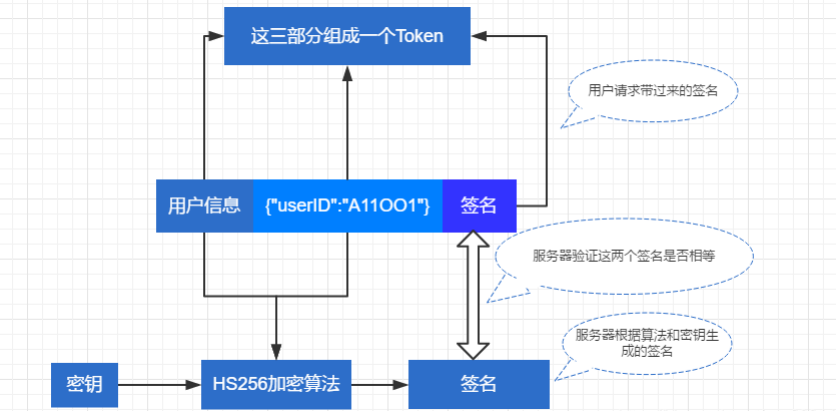
用户访问系统 a，需登录的时候跳到 SSO 系统，在 SSO 系统里通过账号密码认证之后，SSO 的服务器端保存 session，,并生成一个 sessionId 返回给 SSO 的浏览器端，浏览器端写入 SSO 域下的 Cookie，并生成一个生成一个 ST，携带该 ST 传入系统 a，系统 a 用这个 ST 请求 SSO 系统做校验，校验成功后，系统 a 的服务器端将登录状态写入 session 并种下系统 a 域下的 Cookie。之后系统 a 再做登录验证的时候，就是同域下的认证了。
这时，用户访问系统 b，当跳到 SSO 里准备登录的时候发现 SSO 已经登录了，那 SSO 生成一个 ST，携带该 ST 传入系统 b，系统 b 用这个 ST 请求 SSO 系统做校验，校验成功后，系统 b 的服务器端将登录状态写入 session 并设置系统 b 域下的 Cookie。可以看得出，在这个流程里系统 b 就不需要再走登录了。
关于“跳到 SSO 里准备登录的时候发现 SSO 已经登录了”，这个是怎么做的呢，这就涉及 Oauth2 授权机制了，在这里就不展开讲，简单提个思路，就是在系统 b 向 SSO 系统跳转的时候让它从系统 a 跳转，携带系统 a 的会话信息跳到 SSO，再通过重定向回系统 b。
关于 Oauth2，可移步阮一峰 的《OAuth 2.0 的四种方式》。

第三篇章

我们已经分析过 Cookie-session 的局限性了，还有没有更彻底的解决办法呢？既然 SSO 认证系统的存在会增加单点失败的可能性，那我们是不是索性不要它？这就是去中心化的思路，即省去用来存储和校验用户信息的缓存服务器，以另外的方式在各自系统中进行校验。实现方式简单来说，就是把 session 的信息全部加密到 Cookie 里，发送给浏览器端，用 cpu 的计算能力来换取空间。

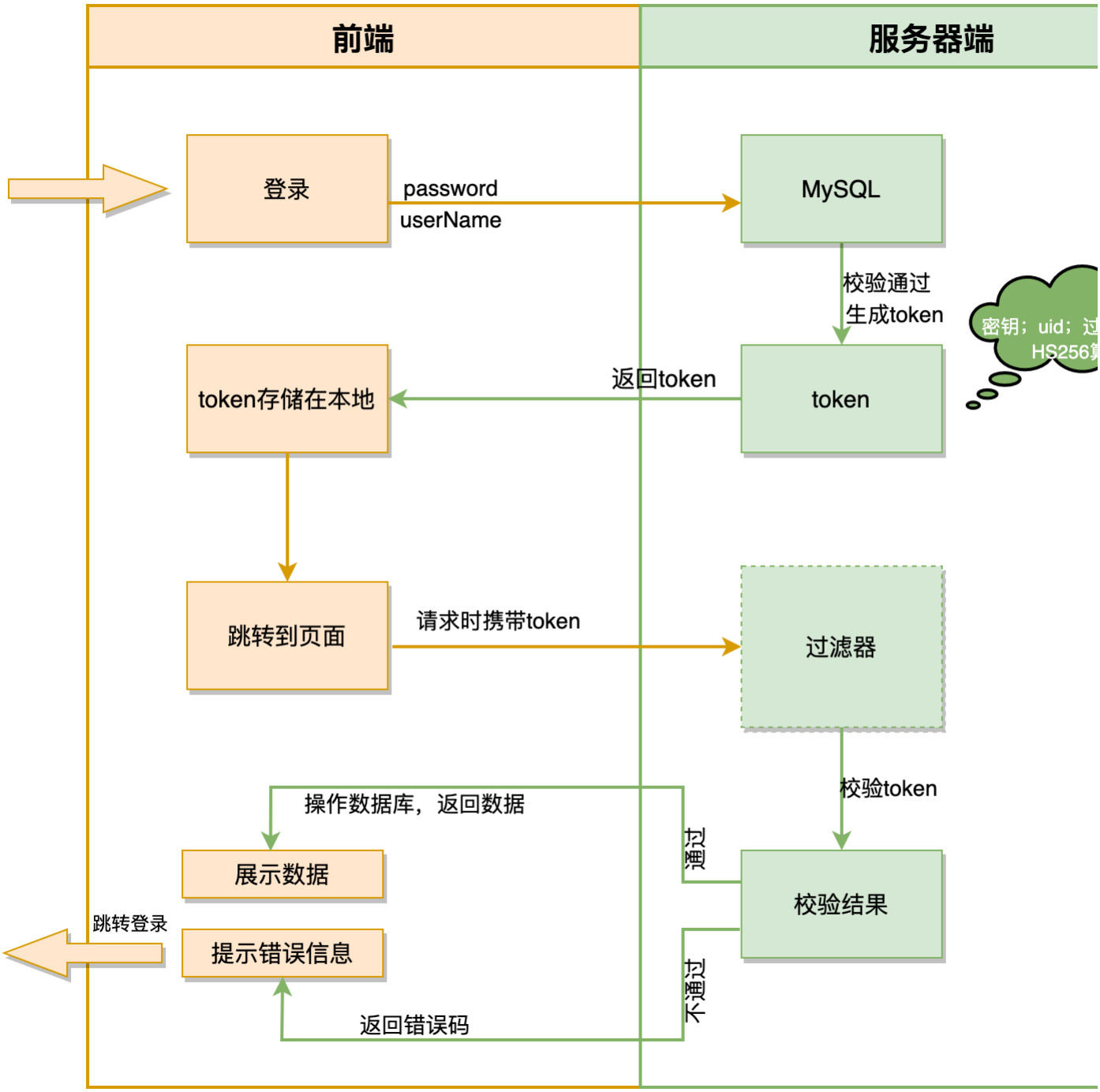
Json Web Token 模式

服务端不保存 sessionId，用户登录系统后，服务器给他下发一个令牌(token)，下一次用户再次通过 Http 请求访问服务器的时候， 把这个 token 通过 Http header 或者 url 带过来进行校验。为了防止别人伪造，我们可以把数据加上一个只有自己才知道的密钥，做一个签名，把数据和这个签名一起作为 token 发送过去。这样我们就不用保存 token 了，因为发送给用户的令牌里，已经包含了用户信息。当用户再次请求过来的时候我用同样的算法和密钥对这个 token 中的数据进行加密，如果加密后的结果和 token 中的签名一致，那我们就可以进行鉴权，并且也能从中取得用户信息。



对于服务端来说，这样只负责生成 token，然后验证 token ，不再需要额外的缓存服务器存储大量的 session，当面对访问量增加的情况，我们只需要针对访问需求大的服务器进行扩容就好了，比扩充整个用户中心的服务器更节省。
假如有人篡改了用户信息，但是由于密钥是不知道的，所以 token 中的签名和被篡改后客户端计算出来的签名肯定是不一致的，也会认证失败，所以不必担心安全问题。
关于 token 的时效性，是这样做的，首次登陆根据账号密码生成一个 token，之后的每次请求，服务端更新时间戳发送一个新的 token，客户端替换掉原来的 token。

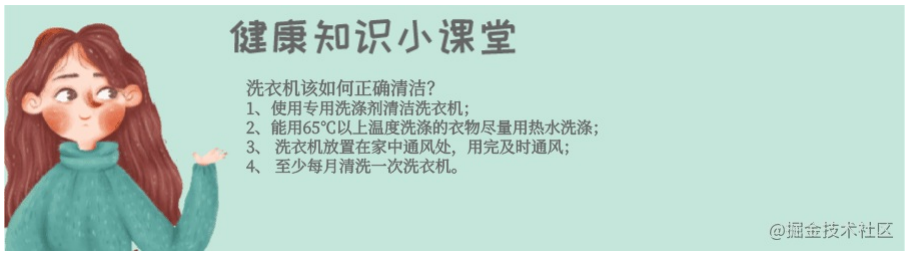
JWT 工作原理流程图



JWT 有什么优劣势

弊端
1.jwt 模式的退出登录实际上是假的登录失效，因为只是浏览器端清除 token 形成的假象，假如用之前的 token 只要没过期仍然能够登陆成功
2.安全性依赖密钥，一旦密钥暴露完蛋
3.加密生成的数据比较长，相对来说占用了更大的流量

优点
1.不依赖 Cookie，可跨端跨程序应用，支持移动设备
2.相对于没有单点登录的 cookie-session 模式来说非常好扩展
3.服务器保持了无状态特性，不需要将用户信息存在服务器或 Session 中
4.对于单点登录需要不停的向 SSO 站点发送验证请求的模式节省了大量请求


健康知识小课堂
洗衣机该如何正确清洁?
1、使用专用洗涤剂清洁洗衣机;
2、能用65℃以上温度洗涤的衣物尽量用热水洗涤;
3、 洗衣机放置在家中通风处，用完及时通风;
4、 至少每月清洗一次洗衣机。
@掘金技术社区