# 前端鉴权的几种方式

## 常见鉴权方式

目前我们常用的鉴权有四种：

1. HTTP Basic Authentication (HTTP基本认证)
2. session-cookie
3. Token 验证(包括JWT,SSO)
4. OAuth(开放授权)

这种认证方式是浏览器遵守http协议实现的基本授权方式,HTTP协议进行通信的过程中，HTTP协议定义了基本认证认证允许HTTP服务器对客户端进行用户身份证的方法。

目前基本没有再使用这种认证方式的,一些老项目的内网认证可能还会有.

这里大概提一下验证过程,主要参考这篇文章

**认证过程：** 1． 客户端向服务器请求数据，请求的内容可能是一个网页或者是一个ajax异步请求，此时，假设客户端尚未被验证，则客户端提供如下请求至服务器:

```
1 bash
2 复制代码
3   Get /index.html HTTP/1.0  Host:www.google.com
```

2． **服务器向客户端发送验证请求代码401**，（`WWW-Authenticate: Basic realm="google.com"` 这句话是关键，如果没有客户端不会 **弹出用户名和密码输入界面**）服务器返回的数据大抵如下：

```
1 yaml
2 复制代码
3   HTTP/1.0 401 Unauthorised  Server: SokEvo/1.0  WWW-Authenticate: Basic
  realm="google.com"  Content-Type: text/html  Content-Length: xxx
```

3． 当符合http1.0或1.1规范的客户端（如IE，FIREFOX）收到401返回值时，将自动弹出一个登录窗口，要求用户输入用户名和密码。

4． 用户输入用户名和密码后，将 **用户名及密码以BASE64加密方式加密(base64不安全!)** ，并将密文放入前一条请求信息中，则客户端发送的第一条请求信息则变成如下内容：

```
sql
复制代码
  Get /index.html HTTP/1.0  Host:www.google.com  Authorization: Basic
d2FuZzp3YW5n
```

注：d2FuZzp3YW5n表示加密后的用户名及密码（用户名：密码 然后通过base64加密，加密过程是浏览器默认的行为，不需要我们人为加密，我们只需要输入用户名密码即可）

5． 服务器收到上述请求信息后，将Authorization字段后的用户信息取出、解密，将解密后的用户名及密码与用户数据库进行比较验证，如用户名及密码正确，服务器则根据请求，将所请求资源发送给客户端

效果： **客户端未未认证的时候，会弹出用户名密码输入框，这个时候请求时属于pending状态，** 这个时候其实服务当用户输入用户名密码的时候客户端会再次发送带Authentication头的请求。

## session-cookie

这个方式是利用服务器端的session（会话）和浏览器端的cookie来实现前后端的认证， **由于http请求时是无状态的**，服务器正常情况下是不知道当前请求之前有没有来过，这个时候我们如果要记录状态，就需要在服务器端创建一个会话(session),将同一个客户端的请求都维护在各自得会会话中，每当请求到达服务器端的时候，先去查一下该客户端有没有在服务器端创建session，如果有则已经认证成功了，否则就没有认证。

**认证过程：**

1，服务器在接受客户端首次访问时在服务器端创建session，然后保存session(我们可以将session保存在内存中，也可以保存在redis中，推荐使用后者)，然后给这个session生成一个唯一的标识字符串,然后在响应头中种下这个唯一标识字符串。

2.签名。这一步只是对sid进行加密处理，服务端会根据这个secret密钥进行解密。（非必需步骤）

3.浏览器中收到请求响应的时候会解析响应头，然后将sid保存在本地cookie中，浏览器在下次http请求的请求头中会带上该域名下的cookie信息，

4.服务器在接受客户端请求时会去解析请求头cookie中的sid，然后根据这个sid去找服务器端保存的该客户端的session，然后判断该请求是否合法。

**弊端:**

- 服务器内存消耗大: 用户每做一次应用认证,应用就会在服务端做一次记录,以方便用户下次请求时使用,通常来讲session保存在内存中,随着认证用户的增加,服务器的消耗就会很大.
- 易受到CSRF攻击: 基于cookie的一种跨站伪造攻击, 基于cookie来进行识别用户的话,用户本身就携带了值,cookie被截获,用户就很容易被伪造.

## Token验证

### 概念

token是用户身份的验证方式，我们通常叫它：令牌。当用户第一次登录后，服务器生成一个token并将此token返回给客户端，以后客户端只需带上这个token前来请求数据即可， **无需再次带上用户名和密码。**

最简单的token组成(用户唯一的身份标识)、time(当前时间的时间戳)、sign(签名，由token的前几位+盐以哈希算法压缩成一定长的十六进制字符串，可以防止恶意第三方拼接token请求服务器)。还可以把不变的参数也放进token，避免多次查库。

我们可以把Token想象成一个安全的护照。你在一个安全的前台验证你的身份（通过你的用户名和密码），如果你成功验证了自己，你就可以取得这个。当你走进大楼的时候（试图从调用API获取资源），你会被要求验证你的护照，而不是在前台重新验证。

## 验证流程

大概的流程是这样的：

- 1, 客户端使用用户名跟密码请求登录
- 2, 服务端收到请求，去验证用户名与密码
- 3, 验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端
- 4, 客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里
- 5, 客户端每次向服务端请求资源的时候需要带着服务端签发的 Token
- 6, 服务端收到请求，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

总的来说就是客户端在首次登陆以后，服务端再次接收http请求的时候，就只认token了，请求只要每次把token带上就行了，服务器端会拦截所有的请求，然后校验token的合法性，合法就放行， **不合法就返回401**（鉴权失败）。

## Token优点与缺点

**优点:**

- Token 完全由应用管理，所以它可以避开同源策略. (Cookie是不允许垮域访问的,token不存在)
- Token 可以避免 CSRF 攻击(也是因为不需要cookie了)
- Token 可以是无状态的，可以在多个服务间共享
- Token 支持手机端访问(Cookie不支持手机端访问)

服务器只需要对浏览器传来的token值进行解密，解密完成后进行用户数据的查询，如果查询成功，则通过认证.所以，即时有了多台服务器，服务器也只是做了token的解密和用户数据的查询， **它不需要在服务端去保留用户的认证信息或者会话信息，这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了**，这就为应用的扩展提供了便利，解决了session扩展性的弊端。

**缺点:**

- 占带宽: 正常情况下token要比 session_id更大，需要消耗更多流量，挤占更多带宽.(不过几乎可以忽略)
- 性能问题: 相比于session-cookie来说，token需要服务端花费更多的时间和性能来对token进行解密验证.其实Token相比于session-cookie来说就是一个"时间换空间"的方案.

## Token与session的区别

1. **使用Token,服务端不需要保存状态.** 在session中sessionid 是一个唯一标识的字符串，服务端是根据这个字符串，来查询在服务器端保持的session，这里面才保存着用户的登陆状态。但是token本身就是一种登陆成功凭证，他是在登陆成功后根据某种规则生成的一种信息凭证，他里面本身就保存着用户的登陆状态。服务器端只需要根据定义的规则校验这个token是否合法就行。
2. **Token不需要借助cookie的.** session-cookie是需要cookie配合的，那么在http代理客户端的选择上就只有浏览器了，因为只有浏览器才会去解析请求响应头里面的cookie,然后每次请求再默认带上该域名下的cookie。但是我们知道http代理客户端不只有浏览器，还有原生APP等等，这个时候cookie是不起作用的，或者浏览器端是可以禁止cookie的(虽然可以，但是这基本上是属于吃饱没事干的人干的事)，但是token 就不一样，他是登陆请求在登陆成功后再请求响应体中返回的信息，客户端在收到响应的时候，可以把他存在本地的cookie,storage，或者内存中，然后再下一次请求的请求头重带上这个token就行了。简单点来说cookie-session机制他限制了客户端的类型，而token验证机制丰富了客户端类型。
3. 时效性。session-cookie的sessionid实在登陆的时候生成的而且在登出事时一直不变的，在一定程度上安全就会低，而token是可以在一段时间内动态改变的。
4. 可扩展性。token验证本身是比较灵活的，一是token的解决方案有许多，常用的是JWT,二来我们可以基于token验证机制，专门做一个鉴权服务，用它向多个服务的请求进行统一鉴权。

## Token过期与Refresh Token

**Token过期:**

token是访问特定资源的凭证，出于安全考虑,肯定是要有过期时间的。要不然一次登录便可能一直使用，那token认证还有什么意义? token可定是有过期时间的,一般不会很长,不会超高一个小时.

**Refresh Token :**

为什么需要refresh token?

如果token过期了，就要重新获取。继续重复第一次获取token的过程(比如登录，扫描授权等)，每一小时就必须获取一次! 这样做是非常不好的用户体验。为了解决这个问题,于是就有了refresh token. **通过refresh token去重新获取新的 token.**

refresh token，也是加密字符串，并且和token是相关联的。与获取资源的token不同， **refresh token的作用仅仅是获取新的token**，因此其作用和安全性要求都较低，所以其过期时间也可以设置得长一些,可以以天为最小单位。当然如果refresh token过期了,还是需要重新登录验证的.

.

## JWT原理

JWT 的原理是，服务器认证以后，生成一个 JSON 对象，发回给用户.之后用户与服务器通信的时候.服务器完全只靠这个对象认定用户身份。为了防止用户篡改数据，服务器在生成这个对象的时候， **会加上签名**。

jwt最大的特点就是: **服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展。**

## JWT 的数据结构

它是一个很长的 **字符串**，中间用点（.）分隔成三个部分。

分别是

（头部）.Payload（负载）.Signature（签名）

- **Header :** 部分是一个 JSON 对象，描述 JWT 的元数据，例如: `{ "alg": "HS256","typ": "JWT"}` .alg属性表示签名的算法.默认是 HMAC SHA256（写成 HS256）；typ属性表示这个令牌（token）的类型（type），JWT 令牌统一写为JWT。

头部的 JSON 对象使用 Base64URL 算法转成字符串。

- **Payload:** 部分也是一个 JSON 对象，用来存放实际需要传递的数据。这个 JSON 对象也要使用 Base64URL 算法转成字符串。

**注意: JWT 默认是不加密的，任何人都可以读到，所以不要把秘密信息放在这个部分。**

- **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）.

## JWT 的几个特点

- （1）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。
- （2）JWT 不加密的情况下，不能将秘密数据写入 JWT。
- （3）JWT 不仅可以用于认证，也可以用于交换信息。有效使用 JWT，可以降低服务器查询数据库的次数。
- （4）JWT 的最大缺点是， **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。**
- （5）JWT 本身包含了认证信息，一旦泄露，任何人都可以获得该令牌的所有权限。为了减少盗用，J **WT的有效期应该设置得比较短。** 对于一些比较重要的权限，使用时应该再次对用户进行认证。
- （6）为了减少盗用，JWT 不应该使用 HTTP 协议明码传输，要使用 HTTPS 协议传输。

首次登录时，后端服务器判断用户账号密码正确之后，根据用户id、用户名、定义好的秘钥、过期时间生成 token ，返回给前端； 前端拿到后端返回的 token ,存储在 localStroage 和 Vuex 里； 前端每次路由跳转，判断 localStroage 有无 token ，没有则跳转到登录页，有则请求获取用户信息，改变登录状态； 每次请求接口，在 Axios 请求头里携带 token; 后端接口判断请求头有无 token，没有或者 token 过期，返回401； 前端得到 401 状态码，重定向到登录页面。

### 补充: 后端主动让JWT失效的方法

前面说过JWT一旦签发了,就不再收服务端控制了.因为它在服务端没有记录,是无状态的,是它最大的优点也是最大的缺点.这样就会造成一种不可控性.

例如:如果用户修改了,那他之前未到期的token怎么废弃掉???此时服务端是没有记录的,它是不知道哪些未到期的token是被废弃了的.为了解决这个问题,其实是 **没有完美的方法的!** 都需要后端添加状态,只是那种方法开销最小.

目前常见的处理方法有:

- 1,将 token 存入 DB（如 Redis）中，失效则删除；但增加了一个每次校验时候都要先从 DB 中查询 token是否存在的步骤，而且违背了 JWT 的无状态原则（不推荐）。
- 2,维护一个 token 黑名单，失效则加入黑名单中(用的比较多)。
- 3,在 JWT 中增加一个版本号字段，失效则改变该版本号。
- 4,在服务端设置加密的 key 时，为每个用户生成唯一的 key，失效则改变该 key。

这里就简单说下第二种方法:黑名单

- 1,在签发的jwt中payload加入一个为随机字符的字段 `token_id` .
- 2, 在服务端的分布式缓存上保存一份"groupId"黑名单。如果用户的jwt重置密码等需要作废已经签发但未过期的jwt时，就将该之前用户的"token__id"存入到黑名单中。并分配给他一个新的"token__id"到token中.
- 3,存入到黑名单中的"token__id"会设置一个过期时间.过期后"token_id"自动从黑名单中删除。
- 4,所有需要做JWT有效性校验的服务器,启动时访问分布式缓存. 将黑名单下载到本地内存。并且订阅分布式缓存的消息推送功能，在黑名单发生增删的时候，接收推送消息同步修改内存中的黑名单列表。
- 5,服务器做JWT校验的时候，除了校验过期时间，还要查询内存中的黑名单列表。若在黑名单中，则判定该JWT为失效。

虽然黑名单还是做了分布式存储,但黑名单本身的体积和使用频率却很低,所以开销很小.

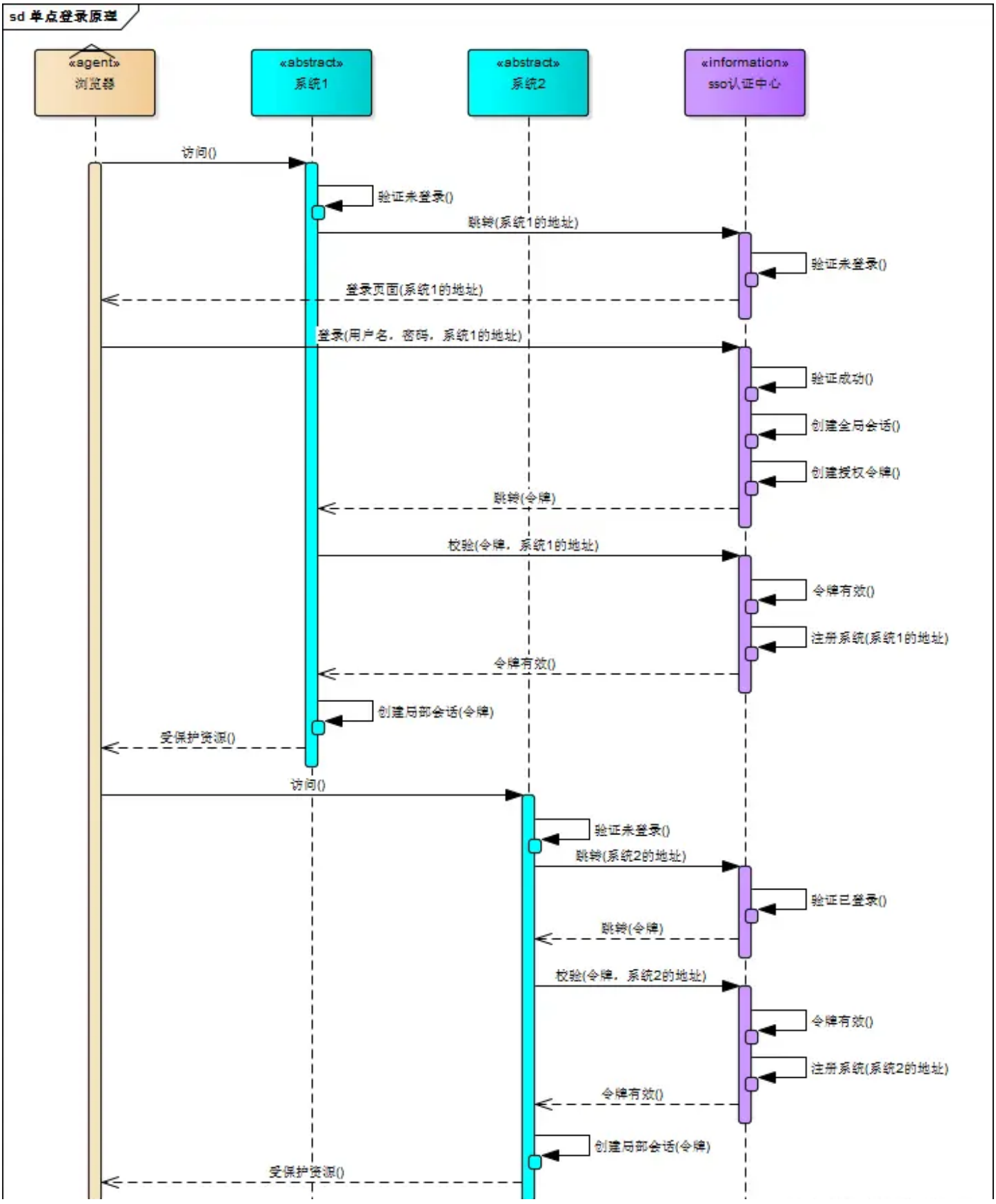
## 单点登录

### 概念

单点登录（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一。SSO的定义是在多个应用系统中，**用户只需要登录一次就可以访问所有相互信任的应用系统。**

SSO一般都需要一个独立的认证中心（passport），子系统的登录均得通过passport，子系统本身将不参与登录操作，当一个系统成功登录以后，passport将会颁发一个令牌给各个子系统，子系统可以拿着令牌会获取各自的受保护资源，为了减少频繁认证，各个子系统在被passport授权以后，会建立一个局部会话，在一定时间内可以无需再次向passport发起认证

## 单点登录流程



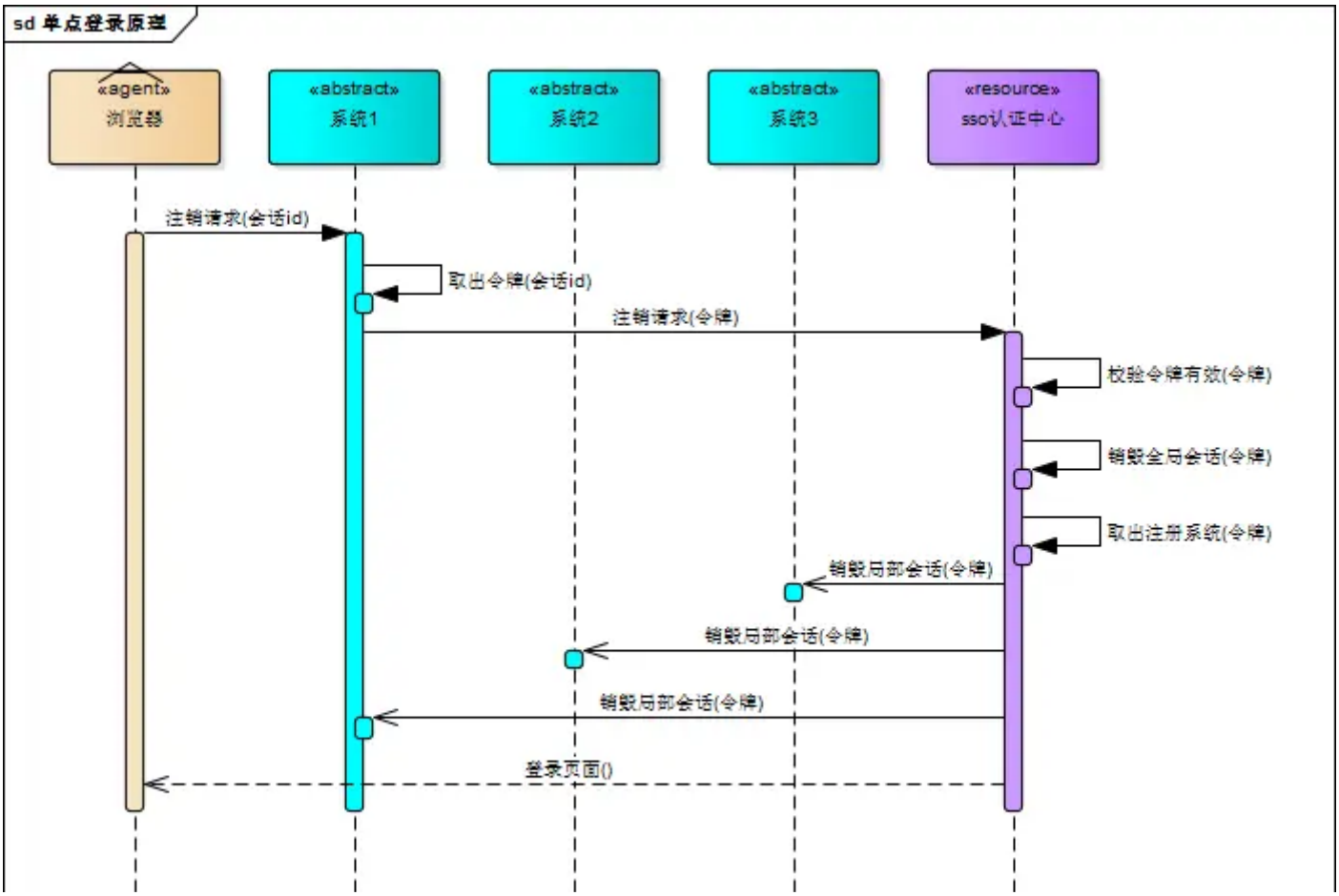

- 用户访问系统1的受保护资源，系统1发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
- sso认证中心发现用户未登录，将用户引导至登录页面
- 用户输入用户名密码提交登录申请

- sso认证中心校验用户信息，创建用户与sso认证中心之间的会话，称为全局会话，同时创建授权令牌
- sso认证中心带着令牌跳转会最初的请求地址（系统1）
- 系统1拿到令牌，去sso认证中心校验令牌是否有效
- sso认证中心校验令牌，返回有效，注册系统1
- 系统1使用该令牌创建与用户的会话，称为局部会话，返回受保护资源
- 用户访问系统2的受保护资源
- 系统2发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
- sso认证中心发现用户已登录，跳转回系统2的地址，并附上令牌
- 系统2拿到令牌，去sso认证中心校验令牌是否有效
- sso认证中心校验令牌，返回有效，注册系统2
- 系统2使用该令牌创建与用户的局部会话，返回受保护资源

用户登录成功之后，会与sso认证中心及各个子系统建立会话，用户与sso认证中心建立的会话称为全局会话，用户与各个子系统建立的会话称为局部会话，局部会话建立之后，用户访问子系统受保护资源将不再通过sso认证中心，全局会话与局部会话有如下约束关系

- 局部会话存在，全局会话一定存在
- 全局会话存在，局部会话不一定存在
- 全局会话销毁，局部会话必须销毁

**注销:**




sso认证中心一直监听全局会话的状态，一旦全局会话销毁，监听器将通知所有注册系统执行注销操作。

- 用户向系统1发起注销请求
- 系统1根据用户与系统1建立的会话id拿到令牌，向sso认证中心发起注销请求
- sso认证中心校验令牌有效，销毁全局会话，同时取出所有用此令牌注册的系统地址
- sso认证中心向所有注册系统发起注销请求
- 各注册系统接收sso认证中心的注销请求，销毁局部会话
- sso认证中心引导用户至登录页面