# **[看图理解JWT如何用于单点登录](https://www.cnblogs.com/lyzg/p/6132801.html)**

**原文：<https://www.cnblogs.com/lyzg/p/6132801.html>**

**阅读目录**

* [前言](https://www.cnblogs.com/lyzg/p/6132801.html" \l "_label0)
* [方案介绍](https://www.cnblogs.com/lyzg/p/6132801.html" \l "_label1)
* [方案总结](https://www.cnblogs.com/lyzg/p/6132801.html" \l "_label2)
* [本文小结](https://www.cnblogs.com/lyzg/p/6132801.html" \l "_label3)

单点登录是我比较喜欢的一个技术解决方案，一方面他能够提高产品使用的便利性，另一方面他分离了各个应用都需要的登录服务，对性能以及工作量都有好处。自从上次研究过JWT如何应用于会话管理，加之以前的项目中也一直在使用[CAS](https://apereo.github.io/cas/4.1.x/)这个比较流行的单点登录框架，所以就一直在琢磨如何能够把JWT跟单点登录结合起来一起使用，尽量能把两种技术的优势都集成到项目中来。本文介绍我从CAS思考得出的SSO的实现方案。

## **前言**

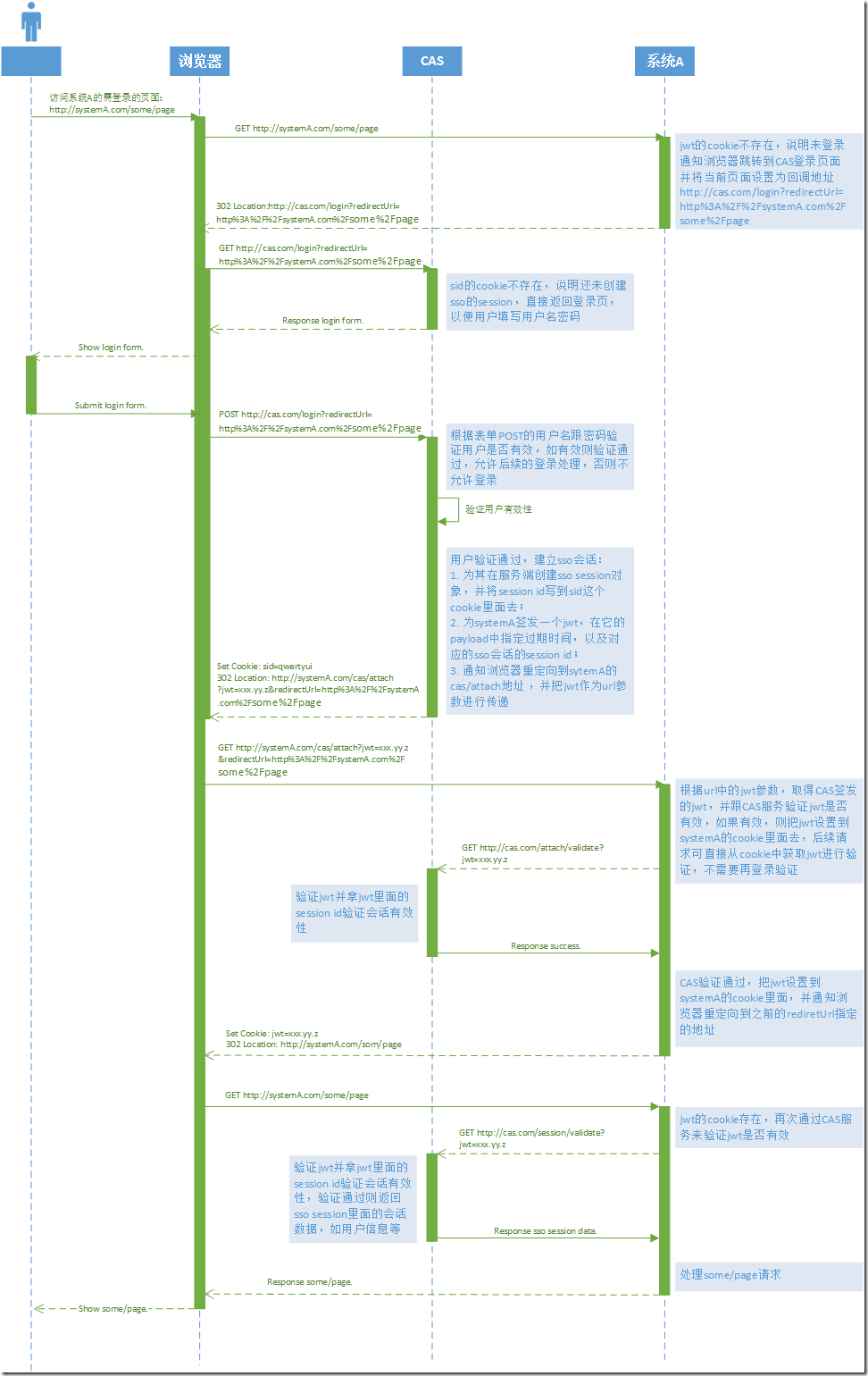
其实CAS这个方案很好，非常强大，它最新的版本已经集成JWT了，所以要是不想自己开发单点登录的服务的话，完全可以考虑使用CAS。但是我认为，我们在做项目的时候，也许一开始并不需要这么强大的产品，CAS提供的登录形式有很多，而我们只需要应用其中的一种；而且它这个框架正是因为强大，所以也会比较复杂，简单上手容易，但是遇到一些特殊的需求，比如我们想在CAS里面加入微信登录，那就需要对它的原理以及API有比较深入的了解才行。综合考虑，还是弄清楚CAS的原理，自己来实现一个基本的SSO服务比较放心。

本文的内容需要对JWT和SSO有一个基本的了解，你可以从这两篇文章来了解JWT的用途：[3种web会话管理的方式](http://www.cnblogs.com/lyzg/p/6067766.html)[JWT实现token-based会话管理](http://www.cnblogs.com/lyzg/p/6028341.html)，还可以从下面的资料来了解SSO的内容：[SSO\_百度百科](https://www.baidu.com/link?url=Br0IMdL-_nydPlcUqkj01OFBoHd_xFVsMdqa1zRrJqH8B12OpBtGjgKgenSNp5nShNiZZH5-wK0Obwg4SYIC8K&wd=&eqid=aead597e0000dd10000000025844d749)。

## **方案介绍**

本文主要是通过时序图的方式来介绍JWT SSO的实现原理，具体的技术实现暂时还没有，不过当你理解了这个方案的原理后，你会觉得最终的实现并不会特别复杂，你可以用任意的平台语言来实现它。下面的时序图，模拟了三个服务，分别是CAS、系统A、系统B，它们分别部署在cas.com，systemA.com和systemB.com；CAS这个服务用来管理SSO的会话；系统A和系统B代表着实际的业务系统。我从五个场景分别来说明这个SSO方案的实现细节。下面先来看第一个。

**场景一：用户发起对业务系统的第一次访问，假设他第一次访问的是系统A的some/page这个页面，它最终成功访问到这个页面的过程是：**

[](http://images2015.cnblogs.com/blog/459873/201612/459873-20161205095409866-1730460249.png)

**在这个过程里面，我认为理解的关键点在于：**

1. 它用到了两个cookie(jwt和sid)和三次重定向来完成会话的创建和会话的传递；

1. jwt的cookie是写在systemA.com这个域下的，所以每次重定向到systemA.com的时候，jwt这个cookie只要有就会带过去；

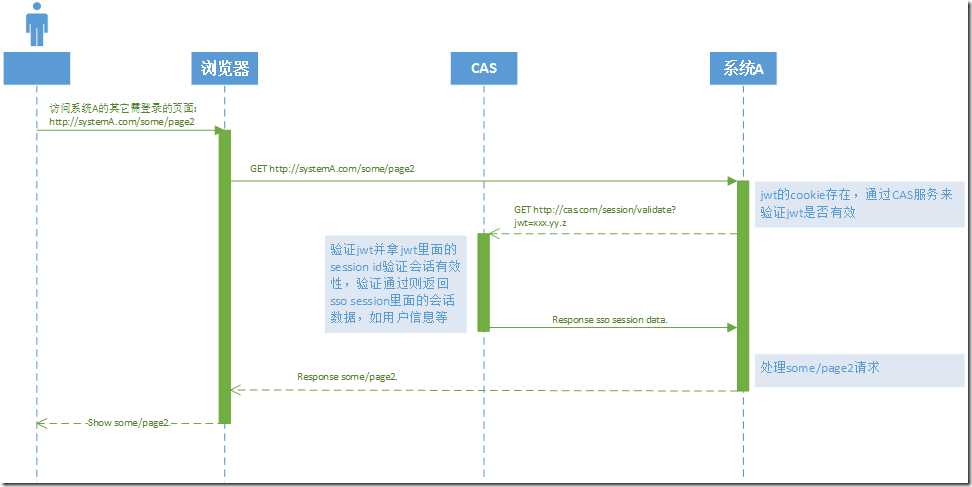
2. sid的cookie是写在cas.com这个域下的，所以每次重定向到cas.com的时候，sid这个cookie只要有就会带过去；

3. 在验证jwt的时候，如何知道当前用户已经创建了sso的会话？因为jwt的payload里面存储了之前创建的sso 会话的session id，所以当cas拿到jwt，就相当于拿到了session id，然后用这个session id去判断有没有的对应的session对象即可。

还要注意的是：CAS服务里面的session属于服务端创建的对象，所以要考虑session id唯一性以及session共享（假如CAS采用集群部署的话）的问题。session id的唯一性可以通过用户名密码加随机数然后用hash算法如md5简单处理；session共享，可以用memcached或者redis这种专门的支持集群部署的缓存服务器管理session来处理。

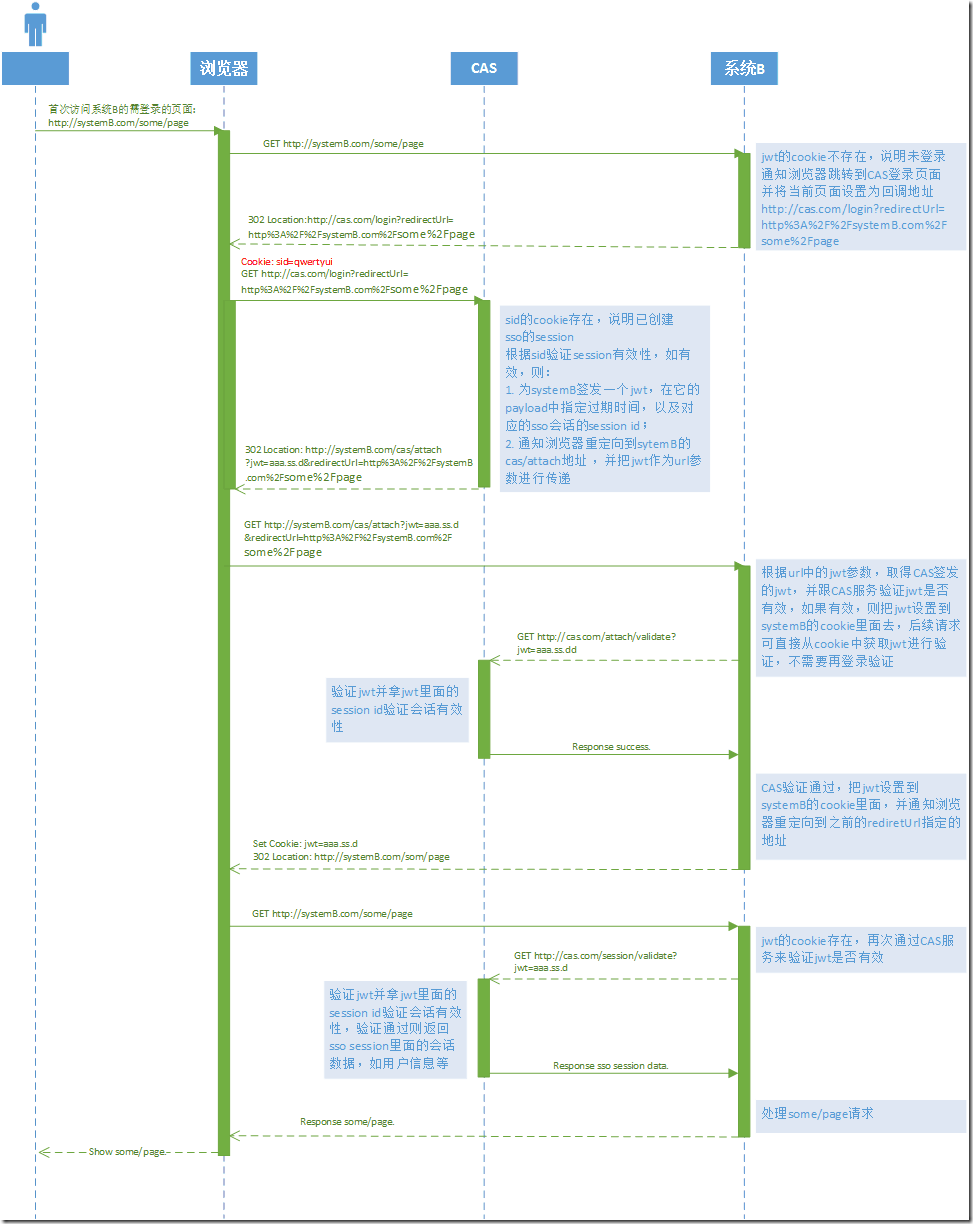
由于服务端session具有生命周期的特点，到期需自动销毁，所以不要自己去写session的管理，免得引发其它问题，到github里找开源的缓存管理中间件来处理即可。存储session对象的时候，只要用session id作为key，session对象本身作为value，存入缓存即可。session对象里面除了session id，还可以存放登录之后获取的用户信息等业务数据，方便业务系统调用的时候，从session里面返回会话数据。

**场景二：用户登录之后，继续访问系统A的其它页面，如some/page2，它的处理过程是：**

[](http://images2015.cnblogs.com/blog/459873/201612/459873-20161205095410976-351334861.png)

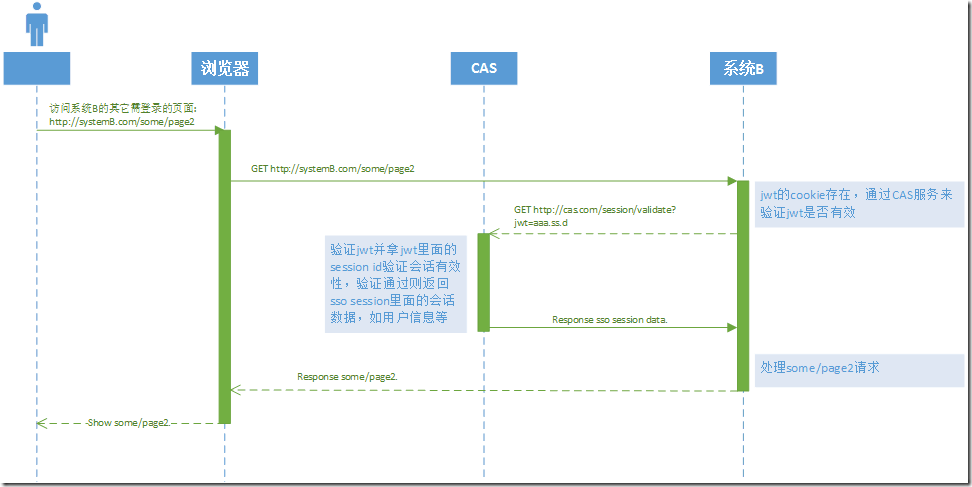
从这一步可以看出，即使登录之后，也要每次跟CAS校验jwt的有效性以及会话的有效性，其实jwt的有效性也可以放在业务系统里面处理的，但是会话的有效性就必须到CAS那边才能完成了。当CAS拿到jwt里面的session id之后，就能到session 缓存服务器里面去验证该session id对应的session对象是否存在，不存在，就说明会话已经销毁了（退出）。

**场景三：用户登录了系统A之后，再去访问其他系统如系统B的资源，比如系统B的some/page，它最终能访问到系统B的some/page的流程是：**

[](http://images2015.cnblogs.com/blog/459873/201612/459873-20161205095411929-1115573953.png)

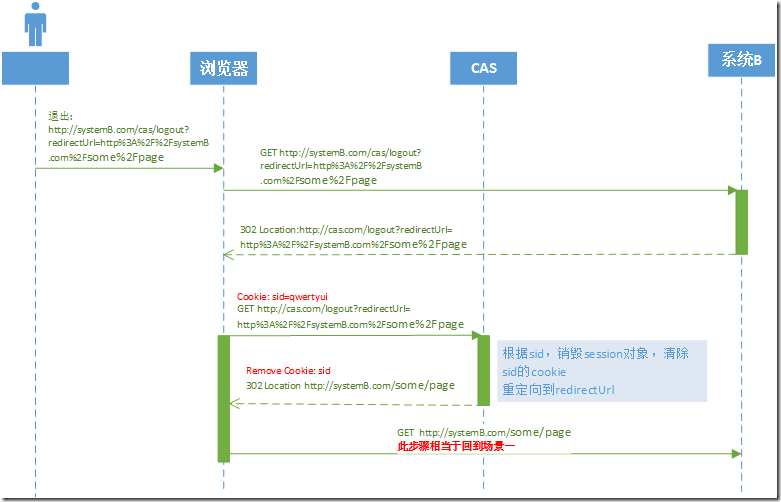
这个过程的关键在于第一次重定向的时候，它会把sid这个cookie带回给CAS服务器，所以CAS服务器能够判断出会话是否已经建立，如果已经建立就跳过登录页的逻辑。

**场景四：用户继续访问系统B的其它资源，如系统B的some/page2：**

[](http://images2015.cnblogs.com/blog/459873/201612/459873-20161205095413069-1717587714.png)

这个场景的逻辑跟场景二完全一致。

**场景五：退出登录，假如它从系统B发起退出，最终的流程是：**

[](http://images2015.cnblogs.com/blog/459873/201612/459873-20161205124912241-1697231898.png)

最重要的是要清除sid的cookie，jwt的cookie可能业务系统都有创建，所以不可能在退出的时候还挨个去清除那些系统的cookie，只要sid一清除，那么即使那些jwt的cookie在下次访问的时候还会被传递到业务系统的服务端，由于jwt里面的sid已经无效，所以最后还是会被重定向到CAS登录页进行处理。

## **方案总结**

**以上方案两个关键的前提：**

1. 整个会话管理其实还是基于服务端的session来做的，只不过这个session只存在于CAS服务里面；

2. CAS之所以信任业务系统的jwt，是因为这个jwt是CAS签发的，理论上只要认证通过，就可以认为这个jwt是合法的。

jwt本身是不可伪造，不可篡改的，但是不代表非法用户冒充正常用法发起请求，所以常规的几个安全策略在实际项目中都应该使用：

1. 使用https

2. 使用http-only的cookie，针对sid和jwt

3. 管理好密钥

4. 防范CSRF攻击。

尤其是CSRF攻击形式，很多都是钻代码的漏洞发生的，所以一旦出现CSRF漏洞，并且被人利用，那么别人就能用获得的jwt，冒充正常用户访问所有业务系统，这个安全问题的后果还是很严重的。考虑到这一点，为了在即使有漏洞的情况将损害减至最小，可以在jwt里面加入一个系统标识，添加一个验证，只有传过来的jwt内的系统标识与发起jwt验证请求的服务一致的情况下，才允许验证通过。这样的话，一个非法用户拿到某个系统的jwt，就不能用来访问其它业务系统了。

在业务系统跟CAS发起attach/validate请求的时候，也可以在CAS端做些处理，因为这个请求，在一次SSO过程中，一个系统只应该发一次，所以只要之前已经给这个系统签发过jwt了，那么后续 同一系统的attach/validate请求都可以忽略掉。

**总的来说，这个方案的好处有：**

1. 完全分布式，跨平台，CAS以及业务系统均可采用不同的语言来开发；

2. 业务系统如系统A和系统B，可实现服务端无状态

3. 假如是自己来实现，那么可以轻易的在CAS里面集成用户注册服务以及第三方登录服务，如微信登录等。

**它的缺陷是：**

1. 第一次登录某个系统，需要三次重定向（不过可以优化成两次）；

2. 登录后的后续请求，每次都需要跟CAS进行会话验证，所以CAS的性能负载会比较大

3. 登陆后的后续请求，每次都跟CAS交互，也会增加请求响应时间，影响用户体验。

## ****本文小结****

本文从理论层面介绍了结合jwt来实现SSO的方案原理，希望它能帮助一些朋友更好的理解SSO以及它的实现方法。本文方案参考自CAS的实现流程，你可以从下面这个资料了解CAS的单点登录实现过程：

[https://apereo.github.io/cas/4.1.x/protocol/CAS-Protocol.html](https://apereo.github.io/cas/4.1.x/protocol/CAS-Protocol.html" \o "https://apereo.github.io/cas/4.1.x/protocol/CAS-Protocol.html)

它的流程跟我这个差别不是特别大，但是从清晰层面来说，我写的还是要更明了一些，所以对比起来阅读，可能理解会更透彻些。