**SSO**

一、 SSO摘要

1 SSO这一概念由来已久，网络上对应不同场景的成熟SSO解决方案比比皆是，从简单到复杂，各式各样应有尽有！开源的有OpenSSO、CAS ，微软的AD SSO，及基于kerberos 的SSO等等……这些优秀的解决方案尽显开发及使用者的逼格，当然需求所致无谓好坏高低，满足实际之需才是王道！

2 本文并不讨论上述提到的方案的整合使用、或者复杂场景如：安全、防火墙、N 多个系统层叠调用这种"巨型项目"里SSO的实现与使用，也并不涉及 C/S 、C/S+B/S 的SSO解决方案，仅关注B/S 上的SSO实现。虽是如此，然而万变不离其宗，这里我们将从一个简而小的登录场景去接触SSO的本质，描述如何原生态地自实现一个轻量、微核的SSO。

3 文章将由浅入深地探讨SSO（单点登录），涉及SSO的定义、表现、原理、实现细节等方面的阐述，借助大家熟知的淘宝、天猫登录场景，通过对阿里登录的模仿实现，建立一个简单模型，然后不断由该模型进行迭代并对每一个迭代版本进行详细描述，最终得到一个支持跨域的SSO（ 力求条理清晰，层层递进，简单但有深度！！！！开始部分本着让即使从未听过SSO的同学也能够从抽象文字定义的概念印象过渡到具象的视觉认知这一宏。

## 二、. SSO简介

**1 SSO定义**

SSO（ Single Sign-On ），中文意即单点登录，翻译得比较精简，个人觉得 Wiki 上的解释更细腻点—— SSO, is a property of access control of multiple related, but independent software systems. With this property a user logs with a single ID and password to gain access to connected system or systems without using different usernames or passwords, or in some configurations seamlessly sign on at each system. ( 单点登录是一种控制多个相关但彼此独立的系统的访问权限, 拥有这一权限的用户可以使用单一的ID和密码访问某个或多个系统从而避免使用不同的用户名或密码，或者通过某种配置无缝地登录每个系统 ).  注：系统，在本文特指WEB 应用或者WEB 服务；用户，下文也会称之为User；ID，用户标识；密码，本文也称其为口令，Password, Passcode 或者 Pin。

**2 与 SSO 交互的2个元素**

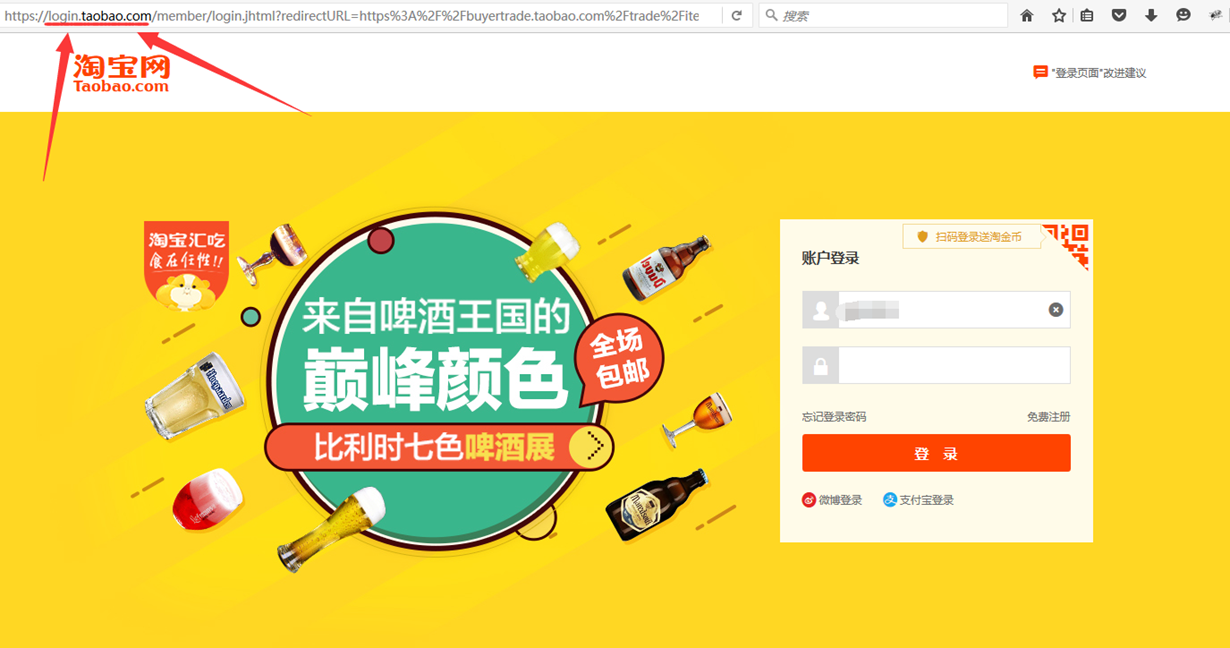
1.  用户，2. 系统，它的特点是：一次登录，全部访问。上面提到SSO是访问控制的一种，控制用户能否登录，即验证用户身份，而且是所有其它系统的身份验证都在它这里进行，那么我们是不是可以认为SSO还是一个验证中心。那么从整个系统层面来看SSO，它的核心就是这3个元素了：1. 用户，2. 系统，3. 验证中心。可能扯了那么多还是不足以形象地描述我们萌萌的SSO。



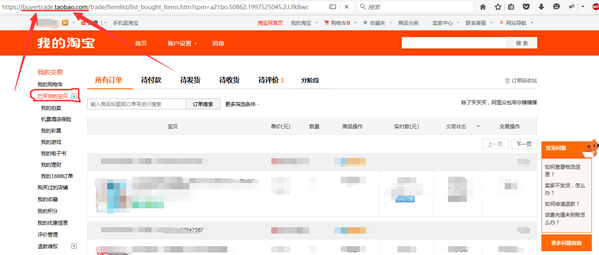
**3 SSO示例——淘宝、天猫的登录场景**

我们暂不考虑细节，先从SSO需要解决的问题入手：使用一个账户通过一次登录，即可在多个相关的系统之间来回访问，为了更加形像我们还是上图：(多图预警)

**登录页面，网址：login.taobao.com ….. 我将在 login.taobao.com 所指的系统进行登录**



访问网站，第一张网址：buyertrade.taobao.com…. 访问 buyertrade.taobao.com所指的系统了；然后访问另一张网页网址为：favoriate.taobao.com， 访问favoriate.taobao.com 所指系统，两个系统的 Domain 是相同的，请注意这点；

接下来我再分别访问淘宝([www.taobao.com](http://www.taobao.com))和天猫([www.tmall.com](http://www.tmall.com))的首页 ，图中显示我仍旧是登录的( 注意：这里是不同的Domain下，系统之间的来回访问)

可以看到，我除了在第一张网页图那里需要输入用户名（ID）和口令（password）进行登录，再访问其它相关系统时，从图2-5 中所有的访问操作，无论域名相同还是不同我都不需要再登录了，它们都知道我叫"望向明天"！对，没错，这就是SSO的作用：一次登录，全部访问，读者也可以尝试下看看是不是如此；

**3. SSO实现描述**

好，经过我上面一大段废话，基本上对SSO要解决什么问题有一个清晰的认识。现在我们自行脑（yi）补（yin）下SSO 的原理是什么样的。

一个账户：嗯，规定所有系统统一使用相同账户，就能保证一个账户了；

一次登录全部访问：通过SSO登录后，让其告知其它各个系统保存该用户的信息，用户就不用重复多次的登录了；

**3.1 方案1**

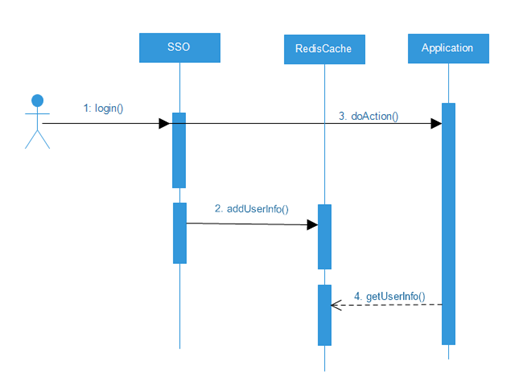
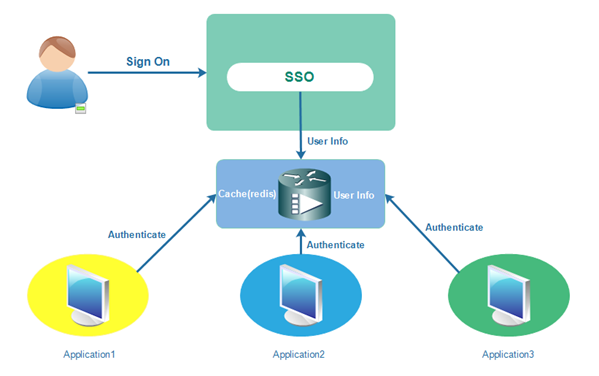
由上面的猜想可以得到第1个解决方案，记为方案1。这里对这个猜想做一点小小的优化，猜想中第2点 "各个系统保存" 好让人闹心，同一份数据保存多份，太浪费，这里我们把每个已登录的用户信息保存到公共缓存中。好，我们再来描述下这个方案：

 User 发送登录请求给SSO，附上自己的 ID 和 password；

SSO验证成功将用户信息保存在公共缓存 Cache 中；

User每次发送请求给系统 Ai 时，将 ID 作为请求参数；

系统 Aj 通过 请求中传过来的 User ID从公共缓存 Cache 中验证 User 是否登录，完成后续动作；



嗯，图文并茂的样子，难道就这么大功告成了？ 我们先把方案1中完成的第一版 SSO 记为SSO\_V1，接下来我们来好好地捋一捋。

**3.2 方案2**

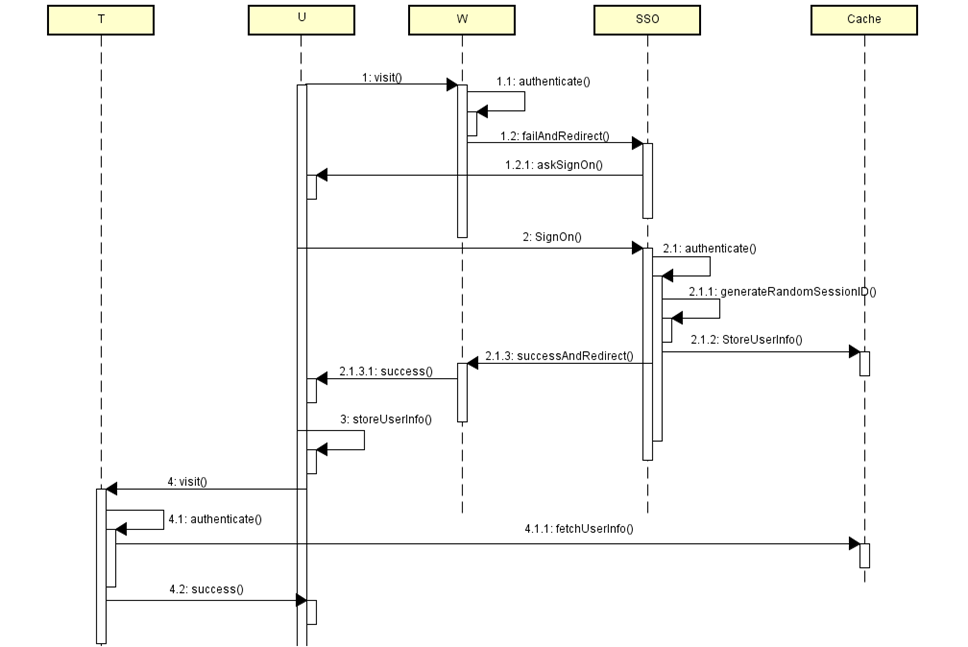
SSO\_V1 貌似解决了问题，但是深入思考，细思极恐！因为这个设计有Bug：每次传 ID 给服务Ai，但是这个ID 每次怎么获取来呢？登录SSO的时候，这倒没有问题，可以让用户填！但第2次请求是发给Ai中的某一个 Aj 时，ID 要怎么来（ 假设百度和新浪是相关但彼此独立的系统，登录百度后，再访问新浪时怎么让新浪取到与登录百度时一样的ID吧）？总不至于每次发请求时都要求用户填一遍ID 吧？

其实我们把 猜想 中最值得思考的问题之一忽略掉了：

如何让SSO"告知"系统Ai，当前登录的User 的ID和password？

这问题可以这样来描述：假设有W ( [www.weidai.com](http://www.weidai.com) )和 T( trade.weidai.com ) 两个系统，W和T 都通过S (login.weidai.com) 系统登录，当由U访问W再转向S 完成登录后，怎样做才能使 U 访问T 时不需要再一次通过 S 进行登录验证？

对，如果你是WEB 开发的老司机，很自然你会想到用cookie ，即把用户信息（ 本文也会称之为UserInfo ）保存在cookie 当中，因为 无论W 、T 或者 S 它们的Domain是一样的——都是 weidai.com ——同一Domain，这有何用？用处就在于 W 、T 以及 S 可以共享此路径下的 cookie。这里，让我们优化的心再一次燃烧起来——直接保存用户的 ID 和 口令 对于我们这么有逼格，有追求的猿来说有点太不讲究——为什么呢？不太安全，cookie 中 最好保存一个 公共Session ID（ 请和WEB 自己生成的Seesion ID进行区分 ） ，而我们的公共缓存 Cache 中保存的 UserInfo 是一个由 公共Session ID为Key ，以包含用户标识和口令的数据结构为Value的Map。最后附上这一流程的时序图及简要说明：



U访问W ，W进行验证，验证失败，跳转至SSO，要求U登录；

U通过SSO登录，SSO进行验证，成功并生成SessionID，随后将UserInfo（ SessionID、ID和口令）存储到公共缓存C 中，跳转至W（携带SessionID），并允许U访问W；

U保存UserInfo （ SessionID ） 至 cookie ;（这里请将 U 看成一个浏览器，当下文有提到 U 保存XXX至Cookie时，读者请自行切换）

U 再访问 T ( 并携带 在3 中保存至cookie 中的 UserInfo ) ，T从公共缓存中拉取UserInfo 进行验证，成功则允许访问；

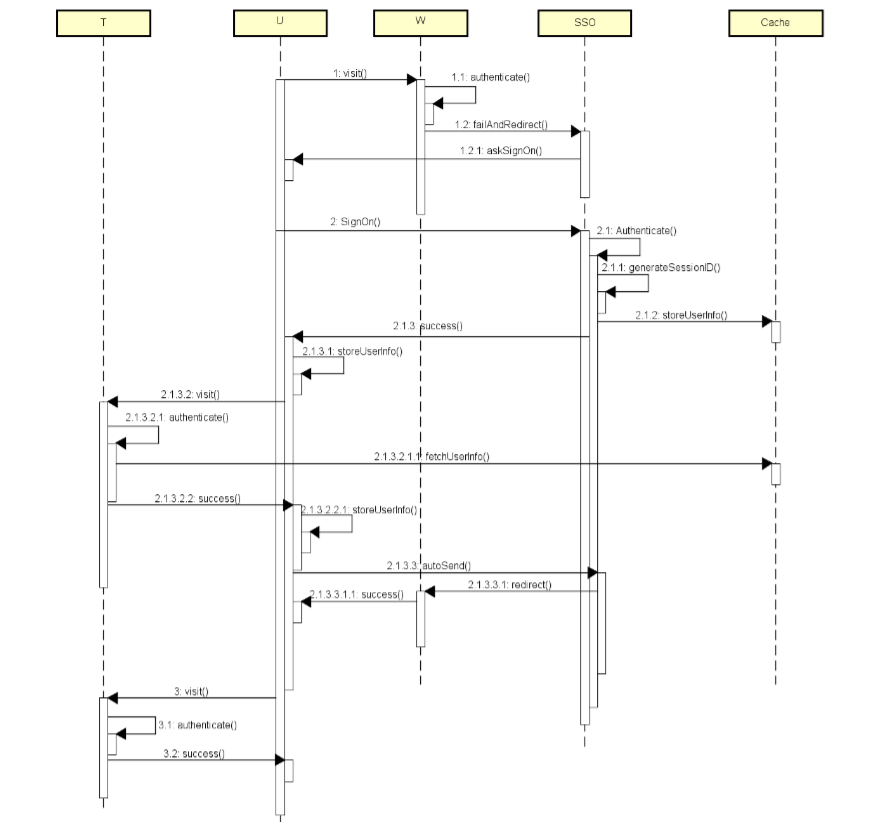
嗯，又是图文并茂的样子，难道再一次大功告成？ 我们暂时把刚才的方案记为方案2，并把方案2中完成的升级版SSO记为SSO\_V2，接下来我们再来好好地捋一捋！

**3.3 方案3**

SSO\_V2 能够在 Domain 相同的情况下"完美"解决问题，但是在Domain不同的情况下怎么做到免登呢？如上面图示淘宝（ [www.taobao.com](http://www.taobao.com) ）和天猫（ [www.tmall.com](http://www.tmall.com) ）若采用SSO\_V2 肯定无法做到免登的，因为我们知道当访问天猫时（Domain 为tmall.com ），淘宝（ Domain 为 taobao.com ）下的 cookie 是无法随访问请求一并传给与天猫相关的系统的。所以问题变成，怎么让不同Domain下的系统也"知晓"用户已经登录的实事？

在我们提出SSO\_V3前，我们先看看SSO 本质是什么？通过这么多的文字描述、样图解释，我们可以看到，要让用户"一次登录，全部访问"无非就是让所有的系统共享"一份"（相同）已验证的、安全可靠的验证信息。所以问题就可以转化为：不同Domain下的系统如何共享一份的验证信息？既然Domain无法做到交叉访问，那我们可以让不同Domain下的WEB应用持有相同的验证信息，这在效果上不就是一份吗！所以最终要解决的问题就是：SSO系统如何使不同的 Domain 拥有一份相同的cookie？ —— 让SSO在用户进行登录时再去访问其它域下的系统，并让各个系统保存一样的验证信息，这样不同域下就会有同一份cookie。

以下是SSO\_V3的时序图和文字说明，这里我们假设 SSO 的Domain 为 SD，T 的 Domain 为 TD：



U第一次访问W，W验证失败，跳转至SSO要求U进行登录验证；

登录并使各不同Domain下：

U 给SSO发送登录请求，SSO验证成功，生成SessionID 并保存UserInfo；

返回给U的Response 将 UserInfo 存放至cookie中，Domain为SD；

将 2 中 cookie 内容作为query parameter 重定向至T，T验证后成功返回给U，也在Response 中设置 cookie；Domain为TD；

U自动访问SSO，SSO将请求重定向至W，完成U对W 的访问；

U 再访问 T，验证成功并允许U进行访问；

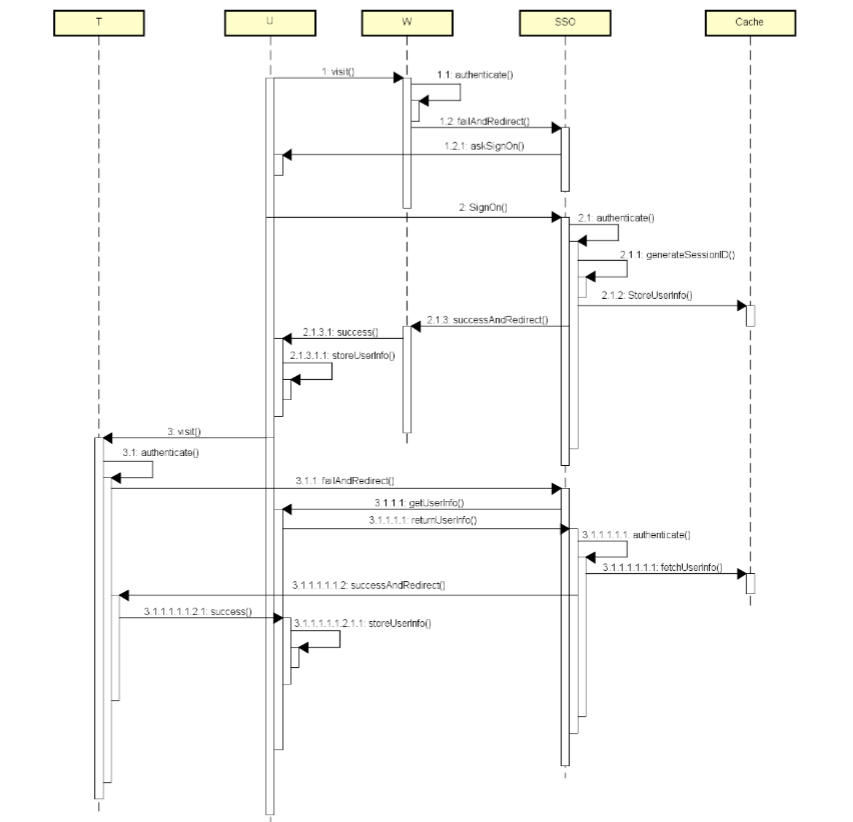
嗯，还是图文并茂的样子，这下是不是可以完事了呢？我们还是把刚才的方案记为方案3，并把方案3中完成的升级版SSO记为SSO\_V3，然后还是来好好地捋一捋！

**3.4 方案4**

再细细的考虑下SSO\_V3的实现方式，有没有感觉它哪里有点不对劲（ 思维一直跟着我来走，是不是被绕晕了，想发现不对劲，怎么可能）？ SSO\_V3 使不同 Domain 获取相同的cookie 拷贝时，表面是在U处主动发出向Ｔ的请求（其实是被动）， 但实际上是 SSO 返回给 U 的页面自动完成的（通过 JS、通过页面自动跳转、iframe都可以实现）。所以方案SSO\_V3要求SSO 预先知道有哪些系统是跨域的！！！而且它还有一个很严重的问题：假如与SSO相关但相互独立的系统中，有 20+ 需要跨域才能访问，而SSO要在用户登录时完成20+跳转……现在你是不是要呵呵了？貌似完美解决跨域的SSO\_V3 竟然如此有问题，有没有心好塞！

SSO\_V3 解决的核心问题是：针对跨域的系统，各系统间如何保证获取到的 验证信息是一致的，解决方法即是在用户第一次登录时把验证信息复制给所有跨域的系统。这种方案在跨域系统少的情况下倒是不需要有太多担心，但是当跨域系统多、且验证步骤比较复杂时用户将会卡在登录界面，最后不得不怒关页面！所以当理清这些逻辑，很自然就会想到接下来要如何对SSO\_V3进行优化。核心思想就是：既然一次性解决会有问题，那就分多次解决！简单的描述下我们将要看到的SSO\_V4，用户登录后，当第一次访问跨域系统W 时，跳到SSO复制一份至W的cookie中，过程结束；当访问T时，重复该处理动作。

以下为SSO\_V4的时序图及简要说明：

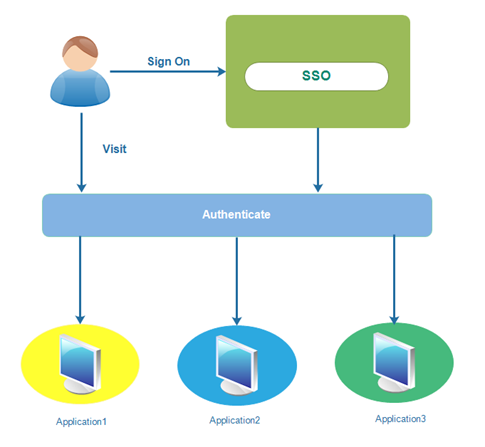
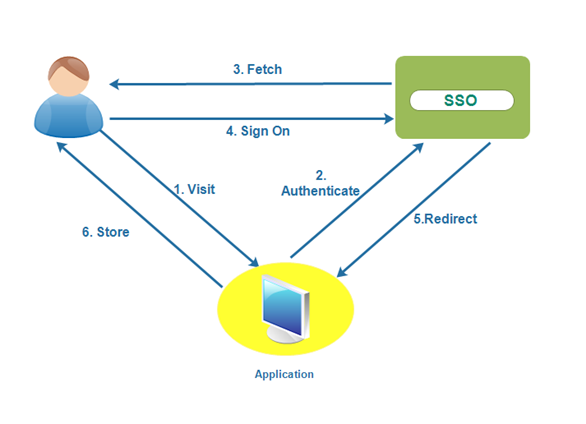


用户U访问W ，W进行验证，验证失败，跳转至SSO，要求U登录；U通过SSO登录，SSO进行验证，成功并生成SessionID，随后将UserInfo（ SessionID、ID和口令）存储到公共缓存C 中，跳转至W（携带SessionID），并允许U访问W；U保存UserInfo （ SessionID ） 至 cookie；U访问T，T 进行验证，失败跳转至SSO，SSO将触发U请求SSO将验证信息随请求一并发给SSO，经SSO验证成功跳转至Ｔ，允许U对T 的访问；使U保存UserInfo（ SessionID）至cookie；

**3.5 小结**

其实我们通过上面的实用版（SSO\_V2，SSO\_V3，SSO\_V4）SSO，可以看到除了用户的第一次登录某个应用相对来说比较特殊，其它处理都是一致的。所以当我们抛去细节之后，不仿这样联想SSO的实现：完成登录逻辑并使各系统共享验证信息和验证逻辑，从这个层次去看SSO，我们发现它其实只负责用户登录和身份验证这2、3个点。

下面是用户第一次登录及SSO与其它系统交互的简图：



**CAS**

## 一、CAS概述

单点登录（Single Sign On , 简称 SSO ）是目前比较流行的服务于企业业务整合的解决方案之一， SSO 使得在多个应用系统中，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。CAS(Central Authentication Service)是一款不错的针对 Web 应用的单点登录框架，本文介绍了 CAS 的原理、协议、在 Tomcat 中的配置和使用，对于采用 CAS 实现轻量级单点登录解决方案的入门读者具有一定指导作用。

## 二、CAS介绍

CAS 是 Yale 大学发起的一个开源项目，旨在为 Web 应用系统提供一种可靠的单点登录方法，CAS 在 2004 年 12 月正式成为 JA-SIG 的一个项目（http://www.jasig.org）。

CAS 具有以下特点：

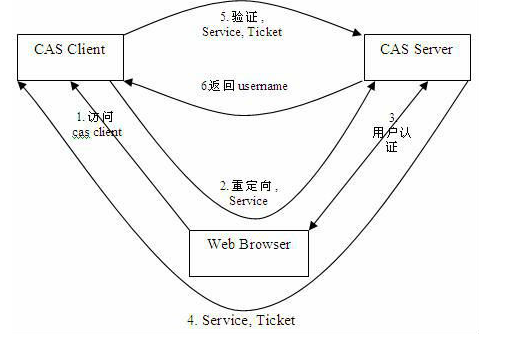
1)开源的企业级单点登录解决方案

2)CAS Server 为需要独立部署的 Web 应用

3)CAS Client 支持非常多的客户端(指Web 应用)，包括Java,.Net,PHP,Perl,Ruby 等

## 三、CAS原理及协议

从结构上看，CAS 包含两个部分： CAS Server 和 CAS Client。CAS Server 需要独立部署，主要负责对用户的认证工作；CAS Client 负责处理对客户端受保护资源的访问请求，需要登录时，重定向到 CAS Server。AS 最基本的协议过程：



CAS Client 与受保护的客户端应用部署在一起，以 Filter 方式保护受保护的资源。对于访问受保护资源的每个 Web 请求，CAS Client 会分析该请求的 Http 请求中是否包含 Service Ticket，如果没有，则说明当前用户尚未登录，于是将请求重定向到指定好的 CAS Server 登录地址，并传递 Service （也就是要访问的目的资源地址），以便登录成功过后转回该地址。用户在第 3 步中输入认证信息，如果登录成功，CAS Server 随机产生一个相当长度、唯一、不可伪造的 Service Ticket，并缓存以待将来验证，之后系统自动重定向到 Service 所在地址，并为客户端浏览器设置一个 Ticket Granted Cookie（TGC），CAS Client 在拿到 Service 和新产生的 Ticket 过后，在第 5，6 步中与 CAS Server 进行身份合适，以确保 Service Ticket 的合法性。

在该协议中，所有与 CAS 的交互均采用 SSL 协议，确保，ST 和 TGC 的安全性。协议工作过程中会有 2 次重定向的过程，但是 CAS Client 与 CAS Server 之间进行 Ticket 验证的过程对于用户是透明的。另外，CAS 协议中还提供了 Proxy （代理）模式，以适应更加高级、复杂的应用场景，具体介绍可以参考 CAS 官方网站上的相关文档。

## 四、CAS SERVER配置

**1.准备工作**

安装配置JDK、安装Tomcat7，此处不做详解。

到CAS官网下载CAS Server和Client，地址如下：

http://downloads.jasig.org/cas/cas-server-4.0.0-release.zip

http://downloads.jasig.org/cas-clients/cas-client-3.2.1-release.zip

**2.部署**

（1）将下载的cas-server-4.0.0-release.zip解开，

把cas-server-4.0.0/modules/cas-server-webapp-4.0.0.war拷贝到 tomcat的webapps目录，并更名为cas.war。

（2）修改cas\WEB-INF\spring-configuration\ticketGrantingTicketCookieGenerator.xml文件，将属性p:cookieSecure="true" 变成 p:cookieSecure="false"（这个设置主要是让CAS不走SSL协议，详见：让CAS不用SSL也可实现跨域）

1. **启动tomcat，**然后访问：http://localhost:8888/cas，如果能出现正常的CAS登录页面，则说明CAS Server 已经部署成功。如下图：



虽然 CAS Server 已经部署成功，但这只是一个缺省的实现，在实际使用的时候，还需要根据实际概况做扩展和定制，最主要的是扩展认证 (Authentication) 接口和 CAS Server 的界面。

## 五、CAS Client配置

**1 将下载的cas-client-3.2.1-release.zip解开**，将cas-client-3.2.1/modules/

cas-client-core-3.2.1.jar，放入你的web项目lib目录中,修改web.xml,添加如下web.xml中单点登录块配置信息：

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://java.sun.com/xml/ns/javaee"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

id="WebApp\_ID" version="2.5">

<display-name>CasClientOne</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<!-- ======================== 单点登录结束 ======================== -->

<!-- 用于单点退出，该过滤器用于实现单点登出功能，可选配置 -->

<listener>

<listener-class>org.jasig.cas.client.session.SingleSignOutHttpSessionListener</listener-class>

</listener>

<!-- 该过滤器用于实现单点登出功能，可选配置。 -->

<filter>

<filter-name>CAS Single Sign Out Filter</filter-name>

<filter-class>org.jasig.cas.client.session.SingleSignOutFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>CAS Single Sign Out Filter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- 该过滤器负责用户的认证工作，必须启用它 -->

<filter>

<filter-name>CASFilter</filter-name>

<filter-class>org.jasig.cas.client.authentication.AuthenticationFilter</filter-class>

<init-param>

<param-name>casServerLoginUrl</param-name>

<param-value>http://localhost:8888/cas/login</param-value><!-- cas 服务器登录 地址 http://IP:PORT/CasWebProName/login -->

</init-param>

<init-param>

<!-- 这里的server是服务端的IP -->

<param-name>serverName</param-name>

<param-value>http://localhost:8080</param-value><!-- 客户端服务器地址 http://IP:PORT -->

</init-param>

</filter>

<filter-mapping>

<filter-name>CASFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- 该过滤器负责对Ticket的校验工作，必须启用它 -->

<filter>

<filter-name>CAS Validation Filter</filter-name>

<filter-class>org.jasig.cas.client.validation.Cas20ProxyReceivingTicketValidationFilter</filter-class>

<init-param>

<param-name>casServerUrlPrefix</param-name>

<param-value>http://localhost:8888/cas</param-value><!-- cas 服务器地址 http://IP:PORT/CasWebProName -->

</init-param>

<init-param>

<param-name>serverName</param-name>

<param-value>http://localhost:8080</param-value><!-- 客户端服务器地址 http://IP:PORT -->

</init-param>

</filter>

<filter-mapping>

<filter-name>CAS Validation Filter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- 该过滤器负责实现HttpServletRequest请求的包裹， 比如允许开发者通过HttpServletRequest的getRemoteUser()方法获得SSO登录用户的登录名，可选配置。 -->

<filter>

<filter-name>CAS HttpServletRequest Wrapper Filter</filter-name>

<filter-class>org.jasig.cas.client.util.HttpServletRequestWrapperFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>CAS HttpServletRequest Wrapper Filter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- 该过滤器使得开发者可以通过org.jasig.cas.client.util.AssertionHolder来获取用户的登录名。 比如AssertionHolder.getAssertion().getPrincipal().getName()。 -->

<filter>

<filter-name>CAS Assertion Thread Local Filter</filter-name>

<filter-class>org.jasig.cas.client.util.AssertionThreadLocalFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>CAS Assertion Thread Local Filter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- ======================== 单点登录结束 ======================== -->

</web-app>

**2 配置完毕后，启动tomcat**，然后访问：http://IP:PORT/WebProName/index.jsp，如果系统跳转到CAS登录页面，输入用户名/密码（正确的）后，会跳转到http://IP:PORT/WebProName/index.jsp，则说明CAS Client已经部署成功。

**3 注：到此CAS框架已搭建完毕**，此种方法未采用SSL协议(需要修改Cas)，如果想采用SSL协议方式，则在上述配置中需要以下调整：修改Cas Server所处Web容器（本文为Tomcat），使其支持SSL协议Cas Server部署时，不需要修改cas\WEB-INF\spring-configuration\ticketGrantingTicketCookieGenerator.xm

Cas Client web.xml中配置的Cas Server相关地址需要将http变更成https

## 六、JDBC认证方式

**1.相关JAR包准备**

cas-server-4.0.0-release.zip\cas-server-4.0.0\modules\cas-server-support-jdbc-4.0.0.jar及相应的数据库驱动包（这里是MySQL驱动mysql-connector-java-5.1.7-bin.jar）复制到cas/WEB-INF/lib目录下。

**2.修改deployerConfigContext.xml1.**

<!-- 注释掉原本固定登录用户 -->

<!-- <bean id="primaryAuthenticationHandler" -->

<!-- class="org.jasig.cas.authentication.AcceptUsersAuthenticationHandler"> -->

<!-- <property name="users"> -->

<!-- <map> -->

<!-- <entry key="casuser" value="Mellon"/> -->

<!-- </map> -->

<!-- </property> -->

<!-- </bean> -->

<!-- 变更为JDBC验证方式 -->

<bean id="primaryAuthenticationHandler" class="org.jasig.cas.adaptors.jdbc.QueryDatabaseAuthenticationHandler">

<property name="dataSource" ref="dataSource"></property>

<property name="sql" value="select password from user where user\_name=?"></property>

<property name="passwordEncoder" ref="MD5PasswordEncoder"></property>

</bean>

<!-- 数据源配置 -->

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>

<property name="url"><value>jdbc:mysql://localhost:3309/ossm?characterEncoding=utf8</value></property>

<property name="username"><value>root</value></property>

<property name="password"><value>root</value></property>

</bean>

<!-- 添加MD5密码加密功能 -->

<bean id="MD5PasswordEncoder" class="org.jasig.cas.authentication.handler.DefaultPasswordEncoder">

<constructor-arg index="0">

<value>MD5</value>

</constructor-arg>

</bean>

## 七、客户端获取用户登录信息

CAS登录成功默认返回的只有用户名，JAVA客户端获取

AttributePrincipal principal = (AttributePrincipal) request.getUserPrincipal();

String username = principal.getName();

而在实际应用中，客户端需要知道更多的用户信息，比如用户的性别，年龄，爱好，地址，用户的分组，角色信息等等，下面介绍如何给客户端返回更多的用户资料和信息：

1.修改cas/WEB-INF/deployerConfigContext.xml，注释掉原有的attributeRepository及attributeRepository的引用信息，添加如下信息

<bean id="attributeRepository" class="org.jasig.services.persondir.support.jdbc.SingleRowJdbcPersonAttributeDao">

<constructor-arg index="0" ref="dataSource"/>

<constructor-arg index="1" value="select \* from user where {0}"/>

<!-- <constructor-arg index="1" value="select \* from user where {0} {1}"/> -->

<!-- 组装sql用的查询条件属性 -->

<property name="queryAttributeMapping">

<map>

<!-- 这里的key需写username，value对应数据库用户名字段 -->

<entry key="username" value="user\_name"/>

<!-- <entry key="id" value="id"/> -->

</map>

</property>

<!-- 如果要组装多个查询条件，需要加上下面这个，默认为AND -->

<property name="queryType">

<value>OR</value>

</property>

<!-- 要获取的属性在这里配置 -->

<property name="resultAttributeMapping">

<map>

<!--key为对应的数据库字段名称，value为提供给客户端获取的属性名字，系统会自动填充值-->

<entry key="id" value="id"/>

<entry key="user\_name" value="username"/>

<entry key="phone" value="phone"/>

<entry key="email" value="email"/>

</map>

</property>

</bean>

2.修改cas/WEB-INF/view/jsp/protocol/2.0/casServiceValidationSuccess.jsp,添加下方标红部分

<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>

<cas:authenticationSuccess>

<cas:user>${fn:escapeXml(assertion.primaryAuthentication.principal.id)}</cas:user>

<%-- 返回更多用户信息配置 By Goma --%>

<c:if test="${fn:length(assertion.chainedAuthentications[fn:length(assertion.chainedAuthentications)-1].principal.attributes) > 0}">

<cas:attributes>

<c:forEach var="attr" items="${assertion.chainedAuthentications[fn:length(assertion.chainedAuthentications)-1].principal.attributes}">

<cas:${fn:escapeXml(attr.key)}>${fn:escapeXml(attr.value)}</cas:${fn:escapeXml(attr.key)}>

</c:forEach>

</cas:attributes>

</c:if>

<c:if test="${not empty pgtIou}">

<cas:proxyGrantingTicket>${pgtIou}</cas:proxyGrantingTicket>

</c:if>

<c:if test="${fn:length(assertion.chainedAuthentications) > 1}">

<cas:proxies>

<c:forEach var="proxy" items="${assertion.chainedAuthentications}" varStatus="loopStatus" begin="0" end="${fn:length(assertion.chainedAuthentications)-2}" step="1">

<cas:proxy>${fn:escapeXml(proxy.principal.id)}</cas:proxy>

</c:forEach>

</cas:proxies>

</c:if>

</cas:authenticationSuccess>

</cas:serviceResponse>

3.客户端取值3.

AttributePrincipal principal = (AttributePrincipal) request.getUserPrincipal();

String username = principal.getName();

Map<String,Object> attributes = principal.getAttributes();

String email = attributes .get("email") + "";

## 八、界面定制

CAS 提供了一套默认的页面，在目录cas/WEB-INF/view/jsp/default下。在部署 CAS 之前，我们可能需要定制一套新的CAS Server页面，添加一些个性化的内容。最简单的方法就是拷贝一份 default文件到“ cas/WEB-INF/view/jsp ”目录下，比如命名为newUI，接下来是实现和修改必要的页面，有 4 个页面是必须的：

casConfirmView.jsp: 当用户选择了“warn”时会看到的确认界面

casGenericSuccess.jsp: 在用户成功通过认证而没有目的Service时会看到的界面

casLoginView.jsp: 当需要用户提供认证信息时会出现的界面

casLogoutView.jsp: 当用户结束 CAS 单点登录系统会话时出现的界面

CAS 的页面采用 Spring框架编写，对于不熟悉 Spring 的使用者，在修改之前需要熟悉该框架。页面定制完过后，还需要做一些配置从而让 CAS 找到新的页面，拷贝cas/WEB-INF/classes/default\_views.properties，重命名为as/WEB-INF/classes/ newUI\_views.properties，并修改其中所有的值到相应新页面。最后是更新cas/WEB-INF/cas-servlet.xml文件中的 viewResolver，将protocol\_views更改为newUI\_views。

## 九、其他功能扩展

增加验证码、密码有效期、限制用户登录之类的功能这里不做描述，如果需要请自行查找。

**共享Session**

一、http协议

**1 http协议是无状态的**，即你连续访问某个网页100次和访问1次对服务器来说是没有区别对待的，因为它记不住你。那么，在一些场合，确实需要服务器记住当前用户怎么办？比如用户登录邮箱后，接下来要收邮件、写邮件，总不能每次操作都让用户输入用户名和密码吧，为了解决这个问题，session的方案就被提了出来，事实上它并不是什么新技术，而且也不能脱离http协议以及任何现有的web技术。

**2 原理很简单，假设你访问网页时就像逛澡堂，第一次进去你是没有钥匙的**，这个时候你交了钱服务台就分配一把钥匙给你，你走到哪里都要带上，因为这是你身份的唯一标识，接下来你用这把钥匙可以去打开一个专有的储物柜存储你的衣物，游完泳，你再用钥匙去打开柜子拿出衣物，最后离开游泳池时，把钥匙归还，你的这次游泳的过程就是一次session，或者叫做会话，在这个例子中，钥匙就是session的key，而储物柜可以理解为存储用户会话信息的介质。

**3 那么在web server中如何实现session呢**？想必看了上面的例子你会很容易理解，主要是解决两个问题，一个是钥匙的问题，一个是存储用户信息的问题。对于第一个问题，即什么东西可以让你每次请求都会自动带到服务器呢？如果你比较了解http协议，那么答案一目了然，就是cookie，如果你想为用户建立一次会话，可以在用户授权成功时给他一个cookie，叫做会话id，它当然是唯一的，比如php就会为建立会话的用户默认set一个名为phpsessid，值看起来为一个随机字符串的cookie，如果下次发现用户带了这个cookie，服务器就知道，哎呀，刚刚这位顾客来了。

**4 剩下的是解决第二个问题，即如何存储用户的信息，**服务器知道会话id为abc的用户来了，那abc想存储自己的私人信息，比如购物车信息，如何处理？这个时候可以用内存、也可以用文件，也可以用数据库了，但有个要求是，数据需要用用户的会话id即可取到，比如php就默认会把会话id为abc的用户会话数据存储到/tmp/phpsess\_abc的文件里面，每次读取都要反序列化程序可以理解的数据，写的时候又需要序列化为持久的数据格式。

**5 较好理解的描述**

session被用于表示一个持续的连接状态，在网站访问中一般指代客户端浏览器的进程从开启到结束的过程。session其实就是网站分析的访问（visits）度量，表示一个访问的过程。

session的常见实现形式是会话cookie（session cookie），即未设置过期时间的cookie，这个cookie的默认生命周期为浏览器会话期间，只要关闭浏览器窗口，cookie就消失了。实现机制是当用户发起一个请求的时候，服务器会检查该请求中是否包含sessionid，如果未包含，则系统会创造一个名为JSESSIONID的输出 cookie返回给浏览器(只放入内存，并不存在硬盘中)，并将其以HashTable的形式写到服务器的内存里面；当已经包含sessionid是，服务端会检查找到与该session相匹配的信息，如果存在则直接使用该sessionid，若不存在则重新生成新的 session。这里需要注意的是session始终是有服务端创建的，并非浏览器自己生成的。　但是浏览器的cookie被禁止后session就需要用get方法的URL重写的机制或使用POST方法提交隐藏表单的形式来实现。

## 二、如何实现session的共享

首先我们应该明白，为什么要实现共享，如果你的网站是存放在一个机器上，那么是不存在这个问题的，因为会话数据就在这台机器，但是如果你使用了负载均衡把请求分发到不同的机器呢？这个时候会话id在客户端是没有问题的，但是如果用户的两次请求到了两台不同的机器，而它的session数据可能存在其中一台机器，这个时候就会出现取不到session数据的情况，于是session的共享就成了一个问题。

1 各种web框架早已考虑到这个问题，比如asp.net，是支持通过配置文件修改session的存储介质为sql server的，所有机器的会话数据都从同一个数据库读，就不会存在不一致的问题；

2 以cookie加密的方式保存在客户端.优点是减轻服务器端的压力，缺点是受到cookie的大小限制，可能占用一定带宽，因为每次请求会在头部附带一定大小的cookie信息,另外这种方式在用户禁止使用cookie的情况下无效.

3 服务器间同步。定时同步各个服务器的session信息，此方法可能有一定延时，用户体验也不是很好。

4 php支持把会话数据存储到某台memcache服务器，你也可以手工把session文件存放的目录改为nfs网络文件系统，从而实现文件的跨机器共享。

5 还有一个简单的办法可以用于会话信息不会频繁变更的情况，在机器a设置用户会话的时候，把会话数据post到机器b的一个cgi，机器b的cgi把会话数据存下来，这样机器a和b都会有同一份session数据的拷贝。

## 三、网站跨服务器之间的Session共享方案

**1. 基于NFS的Session共享**

NFS是Net FileSystem的简称，最早由Sun公司为解决Unix网络主机间的目录共享而研发。这个方案实现最为简单，无需做过多的二次开发，仅需将共享目录服务器mount到各频道服务器的本地session目录即可，缺点是NFS依托于复杂的安全机制和文件系统，因此并发效率不高，尤其对于session这类高并发读写的小文件，会由于共享目录服务器的io-wait过高，最终拖累前端WEB应用程序的执行效率。

**2. 基于数据库的Session共享**

首选当然是大名鼎鼎的MySQL数据库，并且建议使用内存表Heap，提高session操作的读写效率。这个方案的实用性比较强，相信大家普遍在使用，它的缺点在于session的并发读写能力取决于Mysql数据库的性能，同时需要自己实现session淘汰逻辑，以便定时从数据表中更新、删除 session记录，当并发过高时容易出现表锁，虽然我们可以选择行级锁的表引擎，但不得不否认使用数据库存储Session还是有些杀鸡用牛刀的架势。

**3. 基于Cookie的Session共享**

这个方案我们可能比较陌生，但它在大型网站中还是比较普遍被使用。原理是将全站用户的Session信息加密、序列化后以Cookie的方式，统一种植在根域名下（如：.host.com），利用浏览器访问该根域名下的所有二级域名站点时，会传递与之域名对应的所有Cookie内容的特性，从而实现用户的Cookie化Session 在多服务间的共享访问。

这个方案的优点无需额外的服务器资源；缺点是由于受http协议头信心长度的限制，仅能够存储小部分的用户信息，同时Cookie化的 Session内容需要进行安全加解密（如：采用DES、RSA等进行明文加解密；再由MD5、SHA-1等算法进行防伪认证），另外它也会占用一定的带宽资源，因为浏览器会在请求当前域名下任何资源时将本地Cookie附加在http头中传递到服务器。

**4. 基于Memcache的Session共享**

Memcache由于是一款基于Libevent多路异步I/O技术的内存共享系统，简单的Key + Value数据存储模式使得代码逻辑小巧高效，因此在并发处理能力上占据了绝对优势，目前本人所经历的项目达到2000/秒 平均查询，并且服务器CPU消耗依然不到10%。

另外值得一提的是Memcache的内存hash表所特有的Expires数据过期淘汰机制，正好和Session的过期机制不谋而合，降低了过期Session数据删除的代码复杂度，对比“基于数据库的存储方案”，仅这块逻辑就给数据表产生巨大的查询压力。

**5. 一些安全框架的自带解决方案**

**（1）Shiro ：**public class JedisSessionDAO extends AbstractSessionDAO implements SessionDAO

**（2）Spring Security：** spring session 和 spring security整合

## 四、JSESSIONID

**1 所谓session可以这样理解**：当与服务端进行会话时，比如说登陆成功后，服务端会为用户开壁一块内存区间，用以存放用户这次会话的一些内容，比如说用户名之类的。那么就需要一个东西来标志这个内存区间是你的而不是别人的，这个东西就是session id,而这个内存区间你可以理解为session。 **然后，服务器会将这个sessionid发回给你的浏览器，放入你的浏览器的cookies中，它会随着浏览器的关闭而消失。** 之后，只有你浏览器没有关闭，你每向服务器发请求，服务器就会从你发送过来的cookies中拿出这个session id,然后根据这个session id到相应的内存中取你之前存放的数据。 但是，如果你退出登陆了，服务器会清掉属于你的内存区域，所以你再登的话，会产生一个新的session了。

**2 做web开发的同学都知道，http是无状态的会话协议**，也就是说无法保存用户的信息。那如果有一些信息需要在用户的浏览活动中一直保持，该怎么做呢？我们可以把这些信息在每次请求的时候作为参数传递给服务器，但这样做既麻烦又耗费资源，这时候就体现出了session的重要性。session是web开发中不可或缺的一个特性。它是对于一个特定的用户请求，在web服务器上保存的一个全局变量。有了它我们就可以把用户的一些信息保存在服务器上，而不用在服务器和客户端之间来回传递。知道了session的作用，那session是怎么实现的呢？服务器上为每个用户都保存了一个session，那当用户请求过来的时候是怎么知道某一个用户应该对应哪个session呢？这时jsessionid就派上用场了。每一个session都有一个id来作为标识，这个id会传到客户端，每次客户端请求都会把这个id传到服务器，服务器根据id来匹配这次请求应该使用哪个session。

**jsessionid就是客户端用来保存sessionid的变量，主要是针对j2ee实现的web容器**，没有研究过其他语言是用什么变量来保存的。一般对于web应用来说，客户端变量都会保存在cookie中，jsessionid也不例外。不过与一般的cookie变量不同，jsessionid是保存在内存cookie中（设置了过期时间cookie就会存储在硬盘里面。当会话结束时失效，即关闭浏览器窗口，如果没有设置Expires，cookie就会存储在内存里面），在一般的cookie文件中是看不到它的影子的。内存cookie在打开一个浏览器窗口的时候会创建，在关闭这个浏览器窗口的时候也同时销毁。这也就解释了为什么session变量不能跨窗口使用，要跨窗口使用就需要手动把jsessionid保存到cookie里面。

**3 http://www.xxx.com/xxx\_app;jsessionid=xxxxxxxxxx?a=x&b=x**

jsession是什么：jsessionid是session的标识。这就好比每个人都有身份证一样。jsessionid只是tomcat中对session id的叫法，在其它容器里面，不一定就是叫jsessionid了。这是一个保险措施 因为Session默认是需要Cookie支持的，但有些客户浏览器是关闭Cookie的。而jsessionid是存储在Cookie中的，如果禁用Cookie的话，也就是说服务器那边得不到jsessionid，这样也就没法根据jsessionid获得对应的session了，获得不了session就得不到session中存储的数据了。这个时候就需要在URL中指定服务器上的session标识,也就是类似于“jsessionid=5F4771183629C9834F8382E23BE13C4C” 这种格式。

在用户禁用了cookie时候，我们可以通过url重写来实现jsessionid的传递。这就是我上面指出的那样的url:http://www.xxx.com/xxx\_app;jsessionid=xxxxxxxxxx?a=x&b=x..。jessionid通过这样的方式来从客户端传递到服务器端，从而来标识session。注意一点，**jsessionid跟一般的url参数传递方式是不同的，不是作为参数跟在？后面，而是紧跟在url后面用；来分隔。**

**4 response.encodeURL()和response.encodeRedirectURL()。**

这2个方法会判断cookie是否可用，如果禁用了会解析出url中的jsessionid，并连接到指定的url后面，如果没有找到jessionid会自动帮我们生成一个。

**5 当你第一次与服务器通信时，浏览器会保存服务器返回的 Set-Cookie 这个健的值( JSESSIONID=64D21B4D69DFB3041B6375C1932BD6CB )，只要你不关闭浏览器(彻底关闭，关闭选项卡不算)，浏览器会从第二次向服务器发出请求开始，一直带上这个键值对，发给服务器。服务器就会知道，这是同一个人(同一个会话)发起的请求。**

**6 指定JSESSIONID名称**

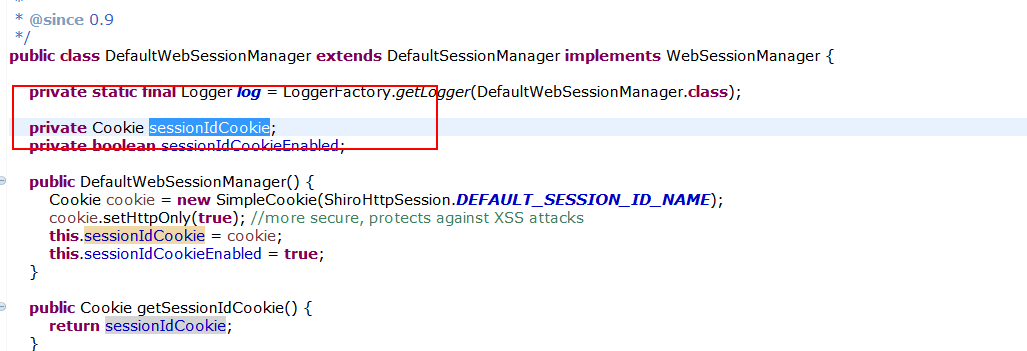
<!-- 指定本系统SESSIONID, 默认为: JSESSIONID 问题: 与SERVLET容器名冲突, 如JETTY, TOMCAT 等默认JSESSIONID,

当跳出SHIRO SERVLET时如ERROR-PAGE容器会为JSESSIONID重新分配值导致登录会话丢失! -->

<bean id=*"sessionIdCookie"* class=*"org.apache.shiro.web.servlet.SimpleCookie"*>

<constructor-arg name=*"name"* value=*"jeesite.session.id"*/>

</bean>





## 五、集群共享session和 SSO，cas的理解

**1 共享session**

集群共享session，因为页面刷新之后cookie里面的JSESSIONID不变的，由于负载均衡可能把请求分发到1,2,3其中一条机器上，如果session存储在本地内存，机器之间读取不到session的；需要把他存储在redis里面才行，还有一种就是session同步和复制，但是这样太复杂了。

**2 SSO/CAS**

表示相互信任的网站之间 其中一个登陆了，就可以自动登录到其他网站。

CAS 包含两个部分CAS Server 和 CAS Client。CAS Server 需要独立部署，主要负责对用户的认证工作；CAS Client 负责处理对客户端受保护资源的访问请求，需要登录时，重定向到 CAS Server。

CAS Client 与受保护的客户端应用部署在一起，以 Filter 方式保护受保护的资源。对于访问受保护资源的每个 Web 请求，CAS Client 会分析该请求的 Http 请求中是否包含 Service Ticket，如果没有，则说明当前用户尚未登录，于是将请求重定向到指定好的 CAS Server 登录地址，并传递 Service （也就是要访问的目的资源地址），以便登录成功过后转回该地址。

用户在第 3 步中输入认证信息，如果登录成功，CAS Server 随机产生一个相当长度、唯一、不可伪造的 Service Ticket，并缓存以待将来验证，之后系统自动重定向到 Service 所在地址，并为客户端浏览器设置一个 Ticket Granted Cookie（TGC），CAS Client 在拿到 Service 和新产生的 Ticket 过后，在第 5，6 步中与 CAS Server 进行身份核实，以确保 Service Ticket 的合法性。在该协议中，所有与 CAS 的交互均采用 SSL 协议，确保，ST 和 TGC 的安全性。协议工作过程中会有 2 次重定向的过程，但是 CAS Client 与 CAS Server 之间进行 Ticket 验证的过程对于用户是透明的。基本结构是Ticket对应一个Token（sessionid，用户名，密码，有效期，最近登录时间）