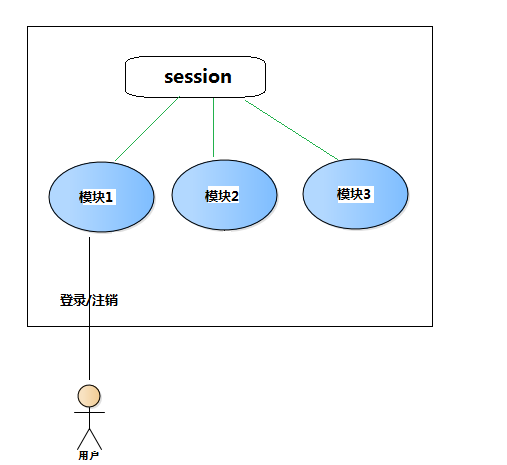
# SSO单点登录

## 为什么需要单点登录

### 单系统登录机制

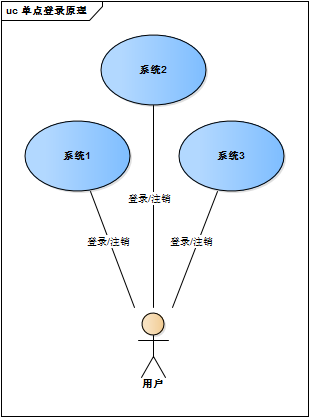
我们假设浏览器第一次请求服务器需要输入用户名与密码验证身份，服务器拿到用户名密码去数据库比对，正确的话说明当前持有这个会话的用户是合法用户，应该将这个会话标记为“已授权”或者“已登录”等等之类的状态，既然是会话的状态，自然要保存在会话对象session中，用户再次访问时，tomcat在会话对象中查看登录状态



### 多系统(分布式)登录

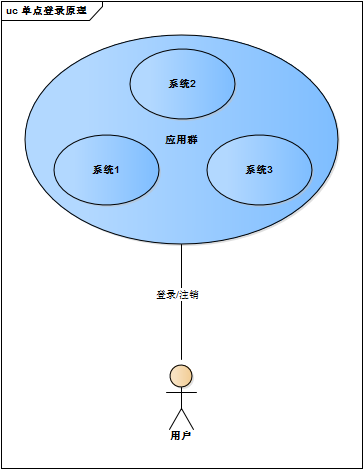
#### 解决方案一：

在每个应用系统是设置独立的登录注销模块



#### 解决方案二：

使用SSO单点登录解决方案 服务器（登录认证） <--🡪 客户端（应用）

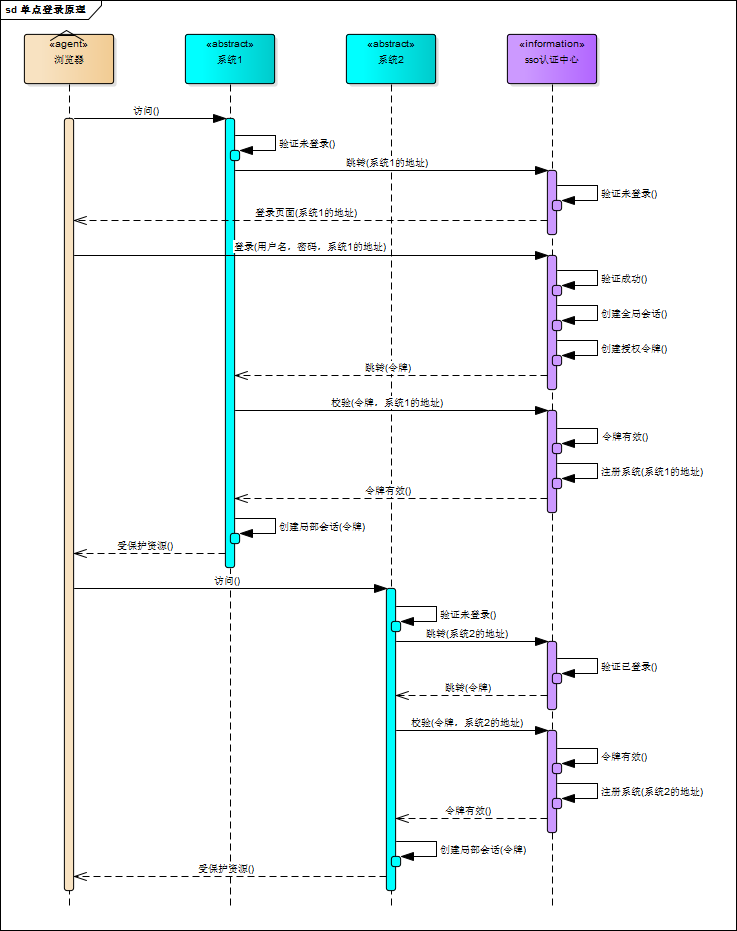


#### 解决方案三、

Session共享 Redis保存session来实现session共享！！

## 单点登录原理（通用）

#### 登录



下面对上图简要描述

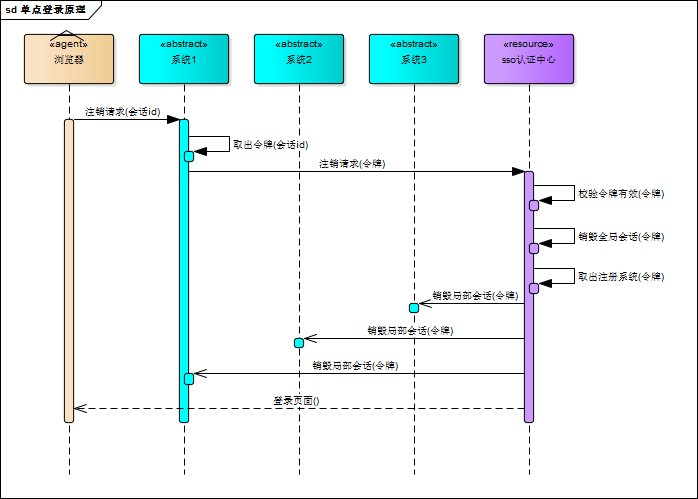
1. 用户访问系统1的受保护资源，系统1发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
2. sso认证中心发现用户未登录，将用户引导至登录页面
3. 用户输入用户名密码提交登录申请
4. sso认证中心校验用户信息，创建用户与sso认证中心之间的会话，称为全局会话，同时创建授权令牌
5. sso认证中心带着令牌跳转会最初的请求地址（系统1）
6. 系统1拿到令牌，去sso认证中心校验令牌是否有效
7. sso认证中心校验令牌，返回有效，注册系统1
8. 系统1使用该令牌创建与用户的会话，称为局部会话，返回受保护资源
9. 用户访问系统2的受保护资源
10. 系统2发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
11. 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
12. 系统2拿到令牌，去sso认证中心校验令牌是否有效
13. sso认证中心校验令牌，返回有效，注册系统2
14. 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

1. 局部会话存在，全局会话一定存在
2. 全局会话存在，局部会话不一定存在
3. 全局会话销毁，局部会话必须销毁

#### 登出

单点登录自然也要单点注销，在一个子系统中注销，所有子系统的会话都将被销毁，用下面的图来说明

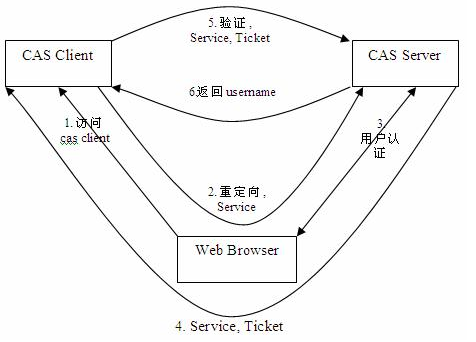
[](http://images2015.cnblogs.com/blog/797930/201611/797930-20161129155242271-222889796.png)

　　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

　　下面对上图简要说明

1. 用户向系统1发起注销请求
2. 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
3. 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
4. sso认证中心向所有注册系统发起注销请求
5. 各注册系统接收sso认证中心的注销请求，销毁局部会话
6. sso认证中心引导用户至登录页面

# CAS实现



## CAS单点登录访问流程：

### 第一次访问CAS客户端1：http://localhost:9001/

CAS Client 就是我们开发的web应用，CAS Client 以Filter的方式保护 Web 应用的受保护资源，过滤从客户端过来的每一个 Web 请求。当请求到达CAS Client时，首先会被CAS的认证过滤器AuthenticationFilter拦截

*<!-- 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:9100/cas/login</**param-value**>   
 *<!--这里的server是服务端的IP -->* </**init-param**>   
 <**init-param**>   
 <**param-name**>serverName</**param-name**>   
 <**param-value**>http://localhost:9001</**param-value**>  
 </**init-param**>   
</**filter**>   
<**filter-mapping**>   
 <**filter-name**>CASFilter</**filter-name**>   
 <**url-pattern**>/\*</**url-pattern**>   
</**filter-mapping**>

第一步：该过滤器会做如下处理

1. 分析 HTTP 请求中是否包含有效session如果有则直接放行。
2. 如果没有有效session，再判断是否有票据Service Ticket(ST) ，如果有则直接放行。
3. 如果没有则重定向到CAS服务器<http://localhost:9100/cas/login?service=http://localhost:9001>

我们第一次访问CAS客户端时，没有登录，走的是情形3。

重定向到<http://localhost:9100/cas/login?service=http://localhost:9001 cas>服务器

当请求到达CAS服务器时，CAS服务器会做如下判断：

request的cookies中是否存在CASTGC(票据授权cookie) ，如果没有（肯定没有），则给客户端浏览器返回一个登录页面。并携带一个LT(Login Ticket 保证当前登录页是服务器给你的)。

用户输入用户名和密码，点击登录，表单提交到CAS服务器端：

http://localhost:9100/cas/login?service=http://localhost:9001

服务器端接受用户名和密码，完成认证工作：

如果用户名和密码正确，CAS服务器端会生成一个Ticket Granted Ticket（TGT）,他可以签发ST。**并为客户端浏览器设置一个 Ticket Granted Cookie （ CASTGC ）根据它可以找到TGT，**客户端浏览器与CAS服务器端建立了会话session，并重定向到CAS Client端：

<http://localhost:9001/?ticket=ST-1-SmNrnzkswweMvSAjeksW-cas01.example.org>

此时访问CAS客户端时，走的是情形2，即直接放行，然后被票据验证过滤器拦截：

*<!-- 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:9100/cas</**param-value**>   
 </**init-param**>   
 <**init-param**>   
 <**param-name**>serverName</**param-name**>   
 <**param-value**>http://localhost:9001</**param-value**>  
 </**init-param**>   
</**filter**>   
<**filter-mapping**>   
 <**filter-name**>CAS Validation Filter</**filter-name**>   
 <**url-pattern**>/\*</**url-pattern**>   
</**filter-mapping**>

票据验证过滤器中会做如下操作：

1. 判断票据是否为空，如果为空则直接放行
2. 如果不为空，则向CAS服务器发送票据认证请求，认证通过，此时CAS客户端与浏览器之间建立session会话，并返回相关资源给浏览器

### 登录成功后再次访问CAS客户端1：http://localhost:9001/

首先会被CAS的认证过滤器AuthenticationFilter拦截，此时客户端已经与浏览器建立了会话，走的是情形1

1. 分析 HTTP 请求中是否包含有效session如果有则直接放行。
2. 如果没有有效session，再判断是否有票据Service Ticket(ST) ，如果有则直接放行。
3. 如果没有则重定向到CAS服务器<http://localhost:9100/cas/login?service=http://localhost:9001>

### 登录成功后访问CAS客户端2：http://localhost:9002/

首先会被CAS的认证过滤器AuthenticationFilter拦截。

1. 分析 HTTP 请求中是否包含有效session如果有则直接放行。
2. 如果没有有效session，再判断是否有票据Service Ticket(ST) ，如果有则直接放行。
3. 如果没有则重定向到CAS服务器<http://localhost:9100/cas/login?service=http://localhost:9001>

此时走的是情形3

当请求到达CAS服务器时，CAS服务器会做如下判断：

request的cookies中是否存在CASTGC ，此时有，然后通过CASTGC获取TGT。通过TGT生成ST票据。携带票据重定向到CAS Client端

<http://localhost:9001/?ticket=ST-1-SmNrnzkswweMvSAjeksW-cas01.example.org>

此后逻辑略

## CAS总结

CAS 的 SSO 实现方式可简化理解为： 1 个 Cookie 和 N 个 Session 。 CAS Server 创建 cookie，在所有应用认证时使用，各应用通过创建各自的 Session 来标识用户是否已登录。

用户在一个应用验证通过后，以后用户在同一浏览器里访问此应用时，客户端应用中的过滤器会在 session 里读取到用户信息，所以就不会去 CAS Server 认证。如果在此浏览器里访问别的 web 应用时，客户端应用中的过滤器在 session 里读取不到用户信息，就会去 CAS Server 的 login 接口认证，但这时CAS Server 会读取到浏览器传的 cookie （ TGC ），所以 CAS Server 不会要求用户去登录页面登录，只是会根据 service 参数生成一个 Ticket（ST） ，然后再和 web 应用做一个验证 ticket 的交互而已。

## 登录状态判断

用户到认证中心登录后，用户和认证中心之间建立起了会话，我们把这个会话称为全局会话。当用户后续访问系统应用时，我们不可能每次应用请求都到认证中心去判定是否登录，这样效率非常低下，这也是单Web应用不需要考虑的。

我们可以在系统应用和用户浏览器之间建立起局部会话，局部会话保持了客户端与该系统应用的登录状态，局部会话依附于全局会话存在，全局会话消失，局部会话必须消失。

用户访问应用时，首先判断局部会话是否存在，如存在，即认为是登录状态，无需再到认证中心去判断。如不存在，就重定向到认证中心判断全局会话是否存在，如存在，按1提到的方式通知该应用，该应用与客户端就建立起它们之间局部会话，下次请求该应用，就不去认证中心验证了。

# CAS 安全性

CAS 的安全性依赖于 SSL (https)。使用的是 secure cookie 。



## CASTGC 安全性

对于一个 CAS 用户来说，最重要是要保护它的 TGC ，如果 TGC 不慎被 CAS Server 以外的实体获得， Hacker 能够找到该 TGC ，然后冒充 CAS 用户访问 **所有**授权资源。

从基础模式可以看出， TGC 是 CAS Server 通过 SSL 方式发送给终端用户，因此，要截取 TGC 难度非常大，从而确保 CAS 的安全性。

TGT (全局会话)的存活周期默认为 120 分钟。

## ST (票据)安全性

ST （ Service Ticket ）是通过 Http 传送的，因此网络中的其他人可以 Sniffer 到其他人的 Ticket 。 CAS 通过以下几方面来使 ST 变得更加安全（事实上都是可以配置的）：

1、   ST 只能使用一次

CAS 协议规定，无论 Service Ticket 验证是否成功， CAS Server 都会清除服务端缓存中的该Ticket ，从而可以确保一个 Service Ticket 不被使用两次。

2、   ST 在一段时间内失效

CAS 规定 ST 只能存活一定的时间，然后 CAS Server 会让它失效。默认有效时间为 5 分钟。

3、   ST 是基于随机数生成的

ST 必须足够随机，如果 ST 生成规则被猜出， Hacker 就等于绕过 CAS 认证，直接访问 **对应的**服务。

<https://blog.csdn.net/dovejing/article/details/44523545> cas服务器源码解析！！