**[java实现简单的单点登录](http://www.blogjava.net/xcp/archive/2010/04/13/318125.html)**

Posted on 2010-04-13 10:28 [xcp](http://www.blogjava.net/xcp/) 阅读(46501) [评论(0)](http://www.blogjava.net/xcp/archive/2010/04/13/318125.html#Post)  [编辑](http://www.blogjava.net/xcp/admin/EditPosts.aspx?postid=318125)  [收藏](http://www.blogjava.net/xcp/AddToFavorite.aspx?id=318125) 所属分类: [JAVA](http://www.blogjava.net/xcp/category/33476.html) http://www.blogjava.net/xcp/aggbug/318125.html?webview=1

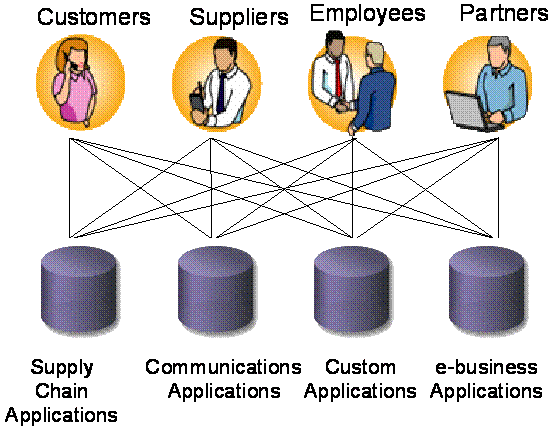
**摘要**：单点登录（SSO）的技术被越来越广泛地运用到各个领域的软件系统当中。本文从业务的角度分析了单点登录的需求和应用领域；从技术本身的角度分析了单点登录技术的内部机制和实现手段，并且给出Web-SSO和桌面SSO的实现、源代码和详细讲解；还从安全和性能的角度对现有的实现技术进行进一步分析，指出相应的风险和需要改进的方面。本文除了从多个方面和角度给出了对单点登录（SSO）的全面分析，还并且讨论了如何将现有的应用和SSO服务结合起来，能够帮助应用架构师和系统分析人员从本质上认识单点登录，从而更好地设计出符合需要的安全架构。

**关键字**：SSO, Java, J2EE, JAAS

***1 什么是单点登陆***

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。

较大的企业内部，一般都有很多的业务支持系统为其提供相应的管理和IT服 务。例如财务系统为财务人员提供财务的管理、计算和报表服务；人事系统为人事部门提供全公司人员的维护服务；各种业务系统为公司内部不同的业务提供不同的 服务等等。这些系统的目的都是让计算机来进行复杂繁琐的计算工作，来替代人力的手工劳动，提高工作效率和质量。这些不同的系统往往是在不同的时期建设起来 的，运行在不同的平台上；也许是由不同厂商开发，使用了各种不同的技术和标准。如果举例说国内一著名的IT公司（名字隐去），内部共有60多个业务系统，这些系统包括两个不同版本的SAP的ERP系统，12个不同类型和版本的数据库系统，8个不同类型和版本的操作系统，以及使用了3种不同的防火墙技术，还有数十种互相不能兼容的协议和标准，你相信吗？不要怀疑，这种情况其实非常普遍。每一个应用系统在运行了数年以后，都会成为不可替换的企业IT架构的一部分，如下图所示。



随 着企业的发展，业务系统的数量在不断的增加，老的系统却不能轻易的替换，这会带来很多的开销。其一是管理上的开销，需要维护的系统越来越多。很多系统的数 据是相互冗余和重复的，数据的不一致性会给管理工作带来很大的压力。业务和业务之间的相关性也越来越大，例如公司的计费系统和财务系统，财务系统和人事系 统之间都不可避免的有着密切的关系。

为了降低管理的消耗，最大限度的重用已有投资的系统，很多企业都在进行着企业应用集成（EAI）。 企业应用集成可以在不同层面上进行：例如在数据存储层面上的“数据大集中”，在传输层面上的“通用数据交换平台”，在应用层面上的“业务流程整合”，和用 户界面上的“通用企业门户”等等。事实上，还用一个层面上的集成变得越来越重要，那就是“身份认证”的整合，也就是“单点登录”。

通常来说，每个单独的系统都会有自己的安全体系和身份认证系统。整合以前，进入每个系统都需要进行登录，这样的局面不仅给管理上带来了很大的困难，在安全方面也埋下了重大的隐患。下面是一些著名的调查公司显示的统计数据：

* 用户每天平均 **16**分钟花在身份验证任务上 **-***资料来源： IDS*
* 频繁的 **IT**用户平均有 **21**个密码 **-***资料来源： NTA Monitor Password Survey*
* **49%**的人写下了其密码，而 **67%**的人很少改变它们
* 每 **79**秒出现一起身份被窃事件 **-***资料来源：National Small Business Travel Assoc*
* 全球欺骗损失每年约 **12B -***资料来源：Comm Fraud Control Assoc*
* 到 **2007**年，身份管理市场将成倍增长至 **$4.5B -***资料来源：IDS*

使用“单点登录”整合后，只需要登录一次就可以进入多个系统，而不需要重新登录，这不仅仅带来了更好的用户体验，更重要的是降低了安全的风险和管理的消耗。请看下面的统计数据：

* 提高 **IT**效率：对于每 **1000**个受管用户，每用户可节省**$70K**
* 帮助台呼叫减少至少**1/3**，对于 **10K**员工的公司，每年可以节省每用户 **$75**，或者合计 **$648K**
* 生产力提高：每个新员工可节省 **$1K**，每个老员工可节省 **$350 �***资料来源：Giga*
* **ROI**回报：**7.5**到 **13**个月 �*资料来源：Gartner*

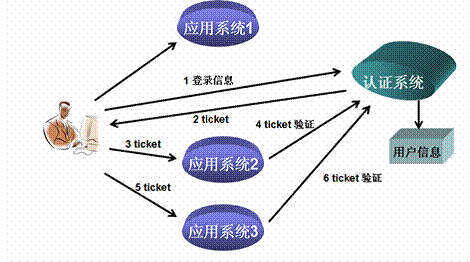
另外，使用“单点登录”还是SOA时代的需求之一。在面向服务的架构中，服务和服务之间，程序和程序之间的通讯大量存在，服务之间的安全认证是SOA应用的难点之一，应此建立“单点登录”的系统体系能够大大简化SOA的安全问题，提高服务之间的合作效率。

***2 单点登陆的技术实现机制***

随着SSO技术的流行，SSO的产品也是满天飞扬。所有著名的软件厂商都提供了相应的解决方案。在这里我并不想介绍自己公司（Sun Microsystems）的产品，而是对SSO技术本身进行解析，并且提供自己开发这一类产品的方法和简单演示。有关我写这篇文章的目的，请参考我的博客（[http://yuwang881.blog.sohu.com/3184816.html](http://yuwang881.blog.sohu.com/)）。

单 点登录的机制其实是比较简单的，用一个现实中的例子做比较。颐和园是北京著名的旅游景点，也是我常去的地方。在颐和园内部有许多独立的景点，例如“苏州 街”、“佛香阁”和“德和园”，都可以在各个景点门口单独买票。很多游客需要游玩所有德景点，这种买票方式很不方便，需要在每个景点门口排队买票，钱包拿 进拿出的，容易丢失，很不安全。于是绝大多数游客选择在大门口买一张通票（也叫套票），就可以玩遍所有的景点而不需要重新再买票。他们只需要在每个景点门 口出示一下刚才买的套票就能够被允许进入每个独立的景点。

单点登录的机制也一样，如下图所示，当用户第一次访问应用系统1的时候，因为还没有登录，会被引导到认证系统中进行登录（1）；根据用户提供的登录信息，认证系统进行身份效验，如果通过效验，应该返回给用户一个认证的凭据－－ticket（2）；用户再访问别的应用的时候（3，5）就会将这个ticket带上，作为自己认证的凭据，应用系统接受到请求之后会把ticket送到认证系统进行效验，检查ticket的合法性（4，6）。如果通过效验，用户就可以在不用再次登录的情况下访问应用系统2和应用系统3了。

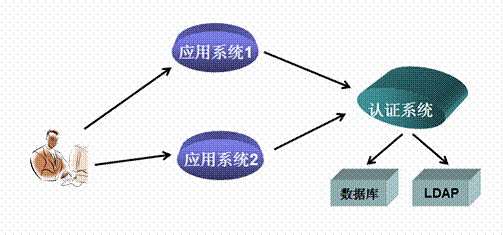


从上面的视图可以看出，要实现SSO，需要以下主要的功能：

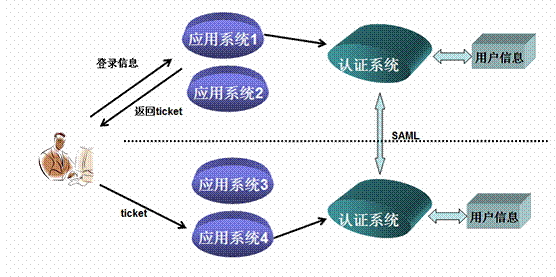
* 所有应用系统共享一个身份认证系统。  
  统一的认证系统是SSO的前提之一。认证系统的主要功能是将用户的登录信息和用户信息库相比较，对用户进行登录认证；认证成功后，认证系统应该生成统一的认证标志（ticket），返还给用户。另外，认证系统还应该对ticket进行效验，判断其有效性。
* 所有应用系统能够识别和提取ticket信息  
  要实现SSO的功能，让用户只登录一次，就必须让应用系统能够识别已经登录过的用户。应用系统应该能对ticket进行识别和提取，通过与认证系统的通讯，能自动判断当前用户是否登录过，从而完成单点登录的功能。

上面的功能只是一个非常简单的SSO架构，在现实情况下的SSO有着更加复杂的结构。有两点需要指出的是：

* 单一的用户信息数据库并不是必须的，有许多系统不能将所有的用户信息都集中存储，应该允许用户信息放置在不同的存储中，如下图所示。事实上，只要统一认证系统，统一ticket的产生和效验，无论用户信息存储在什么地方，都能实现单点登录。



* 统一的认证系统并不是说只有单个的认证服务器，如下图所示，整个系统可以存在两个以上的认证服务器，这些服务器甚至可以是不同的产品。认证服务器之间要通过标准的通讯协议，互相交换认证信息，就能完成更高级别的单点登录。如下图，当用户在访问应用系统1时，由第一个认证服务器进行认证后，得到由此服务器产生的ticket。当他访问应用系统4的时候，认证服务器2能够识别此ticket是由第一个服务器产生的，通过认证服务器之间标准的通讯协议（例如SAML）来交换认证信息，仍然能够完成SSO的功能。

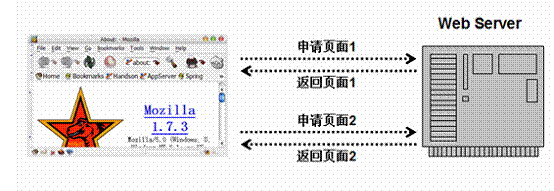


***3 WEB-SSO的实现***

随着互联网的高速发展，WEB应用几乎统治了绝大部分的软件应用系统，因此WEB-SSO是SSO应用当中最为流行。WEB-SSO有其自身的特点和优势，实现起来比较简单易用。很多商业软件和开源软件都有对WEB-SSO的实现。其中值得一提的是OpenSSO （[https://opensso.dev.java.net](https://opensso.dev.java.net/)），为用Java实现WEB-SSO提供架构指南和服务指南，为用户自己来实现WEB-SSO提供了理论的依据和实现的方法。

为什么说WEB-SSO比较容易实现呢？这是有WEB应用自身的特点决定的。

众所周知，Web协议（也就是HTTP）是一个无状态的协议。一个Web应用由很多个Web页面组成，每个页面都有唯一的URL来定义。用户在浏览器的地址栏输入页面的URL，浏览器就会向Web Server去发送请求。如下图，浏览器向Web服务器发送了两个请求，申请了两个页面。这两个页面的请求是分别使用了两个单独的HTTP连接。所谓无状态的协议也就是表现在这里，浏览器和Web服务器会在第一个请求完成以后关闭连接通道，在第二个请求的时候重新建立连接。Web服务器并不区分哪个请求来自哪个客户端，对所有的请求都一视同仁，都是单独的连接。这样的方式大大区别于传统的（Client/Server）C/S结构,在那样的应用中，客户端和服务器端会建立一个长时间的专用的连接通道。正是因为有了无状态的特性，每个连接资源能够很快被其他客户端所重用，一台Web服务器才能够同时服务于成千上万的客户端。



但是我们通常的应用是有状态的。先不用提不同应用之间的SSO，在同一个应用中也需要保存用户的登录身份信息。例如用户在访问页面1的时候进行了登录，但是刚才也提到，客户端的每个请求都是单独的连接，当客户再次访问页面2的时候，如何才能告诉Web服务器，客户刚才已经登录过了呢？浏览器和服务器之间有约定：通过使用cookie技术来维护应用的状态。Cookie是可以被Web服务器设置的字符串，并且可以保存在浏览器中。如下图所示，当浏览器访问了页面1时，web服务器设置了一个cookie，并将这个cookie和页面1一起返回给浏览器，浏览器接到cookie之后，就会保存起来，在它访问页面2的时候会把这个cookie也带上，Web服务器接到请求时也能读出cookie的值，根据cookie值的内容就可以判断和恢复一些用户的信息状态。



Web-SSO完全可以利用Cookie结束来完成用户登录信息的保存，将浏览器中的Cookie和上文中的Ticket结合起来，完成SSO的功能。

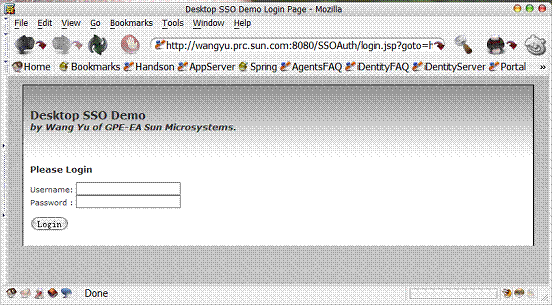
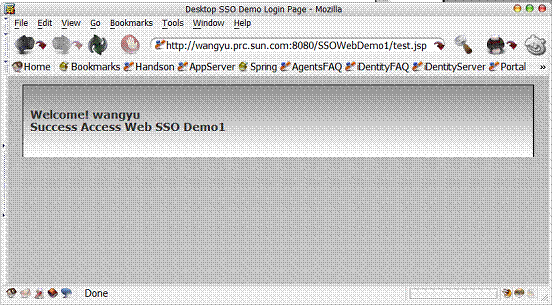
为了完成一个简单的SSO的功能，需要两个部分的合作：

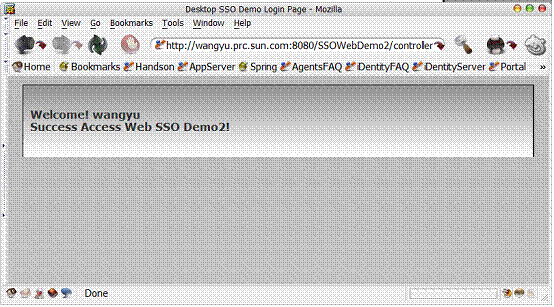
1. 统一的身份认证服务。
2. 修改Web应用，使得每个应用都通过这个统一的认证服务来进行身份效验。

**3.1 Web SSO 的样例**

根据上面的原理，我用J2EE的技术（JSP和Servlet）完成了一个具有Web-SSO的简单样例。样例包含一个身份认证的服务器和两个简单的Web应用，使得这两个 Web应用通过统一的身份认证服务来完成Web-SSO的功能。此样例所有的源代码和二进制代码都可以从网站地址<http://gceclub.sun.com.cn/wangyu/> 下载。

**样例下载、安装部署和运行指南：**

* Web-SSO的样例是由三个标准Web应用组成，压缩成三个zip文件，从[http://gceclub.sun.com.cn/wangyu/web-sso](http://gceclub.sun.com.cn/wangyu/)/中下载。其中SSOAuth（[http://gceclub.sun.com.cn/wangyu/web-sso/SSOAuth.zip](http://gceclub.sun.com.cn/wangyu/)）是身份认证服务；SSOWebDemo1（[http://gceclub.sun.com.cn/wangyu/web-sso/SSOWebDemo1.zip](http://gceclub.sun.com.cn/wangyu/)）和SSOWebDemo2（[http://gceclub.sun.com.cn/wangyu/web-sso/SSOWebDemo2.zip](http://gceclub.sun.com.cn/wangyu/)）是两个用来演示单点登录的Web应用。这三个Web应用之所以没有打成war包，是因为它们不能直接部署，根据读者的部署环境需要作出小小的修改。样例部署和运行的环境有一定的要求，需要符合Servlet2.3以上标准的J2EE容器才能运行（例如Tomcat5,Sun Application Server 8, Jboss 4等）。另外，身份认证服务需要JDK1.5的运行环境。之所以要用JDK1.5是因为笔者使用了一个线程安全的高性能的Java集合类“ConcurrentMap”，只有在JDK1.5中才有。
* 这三个Web应用完全可以单独部署，它们可以分别部署在不同的机器，不同的操作系统和不同的J2EE的产品上，它们完全是标准的和平台无关的应用。但是有一个限制，那两台部署应用（demo1、demo2）的机器的域名需要相同，这在后面的章节中会解释到cookie和domain的关系以及如何制作跨域的WEB-SSO
* 解压缩SSOAuth.zip文件，在/WEB-INF/下的web.xml中请修改“domainname”的属性以反映实际的应用部署情况，domainname需要设置为两个单点登录的应用（demo1和demo2）所属的域名。这个domainname和当前SSOAuth服务部署的机器的域名没有关系。我缺省设置的是“.sun.com”。如果你部署demo1和demo2的机器没有域名，请输入IP地址或主机名（如localhost），但是如果使用IP地址或主机名也就意味着demo1和demo2需要部署到一台机器上了。设置完后，根据你所选择的J2EE容器，可能需要将SSOAuth这个目录压缩打包成war文件。用“jar -cvf SSOAuth.war SSOAuth/”就可以完成这个功能。
* 解压缩SSOWebDemo1和SSOWebDemo2文件，分别在它们/WEB-INF/下找到web.xml文件，请修改其中的几个初始化参数  
  <init-param>  
  <param-name>SSOServiceURL</param-name>  
  <param-value><http://wangyu.prc.sun.com:8080/SSOAuth/SSOAuth></param-value>  
  </init-param>  
  <init-param>  
  <param-name>SSOLoginPage</param-name>  
  <param-value><http://wangyu.prc.sun.com:8080/SSOAuth/login.jsp></param-value>  
  </init-param>  
  将其中的SSOServiceURL和SSOLoginPage修改成部署SSOAuth应用的机器名、端口号以及根路径（缺省是SSOAuth）以反映实际的部署情况。设置完后，根据你所选择的J2EE容器，可能需要将SSOWebDemo1和SSOWebDemo2这两个目录压缩打包成两个war文件。用“jar -cvf SSOWebDemo1.war SSOWebDemo1/”就可以完成这个功能。
* 请输入第一个web应用的测试URL（test.jsp）,例如http://wangyu.prc.sun.com:8080/ SSOWebDemo1/test.jsp，如果是第一次访问，便会自动跳转到登录界面，如下图  
    
  
* 使用系统自带的三个帐号之一登录（例如，用户名：wangyu,密码：wangyu），便能成功的看到test.jsp的内容：显示当前用户名和欢迎信息。  
  
* 请接着在同一个浏览器中输入第二个web应用的测试URL（test.jsp）,例如http://wangyu.prc.sun.com:8080/ SSOWebDemo2/test.jsp。你会发现，不需要再次登录就能看到test.jsp的内容，同样是显示当前用户名和欢迎信息，而且欢迎信息中明确的显示当前的应用名称（demo2）。



**3.2 WEB-SSO代码讲解**

**3.2.1身份认证服务代码解析**

Web-SSO的源代码可以从网站地址http://gceclub.sun.com.cn/wangyu/web-sso/websso\_src.zip下载。身份认证服务是一个标准的web应用，包括一个名为SSOAuth的Servlet，一个login.jsp文件和一个failed.html。身份认证的所有服务几乎都由SSOAuth的Servlet来实现了；login.jsp用来显示登录的页面（如果发现用户还没有登录过）；failed.html是用来显示登录失败的信息（如果用户的用户名和密码与信息数据库中的不一样）。

SSOAuth的代码如下面的列表显示，结构非常简单，先看看这个Servlet的主体部分**：**

|  |
| --- |
| package DesktopSSO;    import java.io.\*;  import java.net.\*;  import java.text.\*;  import java.util.\*;  import java.util.concurrent.\*;    import javax.servlet.\*;  import javax.servlet.http.\*;      public class SSOAuth extends HttpServlet {        static private ConcurrentMap accounts;      static private ConcurrentMap SSOIDs;      String cookiename="WangYuDesktopSSOID";      String domainname;        public void init(ServletConfig config) throws ServletException {          super.init(config);          domainname= config.getInitParameter("domainname");          cookiename = config.getInitParameter("cookiename");          SSOIDs = new ConcurrentHashMap();          accounts=new ConcurrentHashMap();          accounts.put("wangyu", "wangyu");          accounts.put("paul", "paul");          accounts.put("carol", "carol");      }        protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {          PrintWriter out = response.getWriter();          String action = request.getParameter("action");          String result="failed";          if (action==null) {              handlerFromLogin(request,response);          } else if (action.equals("authcookie")){              String myCookie = request.getParameter("cookiename");              if (myCookie != null) result = authCookie(myCookie);              out.print(result);              out.close();          } else if (action.equals("authuser")) {             result=authNameAndPasswd(request,response);              out.print(result);              out.close();          } else if (action.equals("logout")) {              String myCookie = request.getParameter("cookiename");              logout(myCookie);              out.close();          }      }    .....    } |

从代码很容易看出，SSOAuth就是一个简单的Servlet。其中有两个静态成员变量：accounts和SSOIDs，这两个成员变量都使用了JDK1.5中线程安全的MAP类： ConcurrentMap，所以这个样例一定要JDK1.5才能运行。Accounts用来存放用户的用户名和密码，在init()的方法中可以看到我给系统添加了三个合法的用户。在实际应用中，accounts应该是去数据库中或LDAP中获得，为了简单起见，在本样例中我使用了ConcurrentMap在内存中用程序创建了三个用户。而SSOIDs保存了在用户成功的登录后所产生的cookie和用户名的对应关系。它的功能显而易见：当用户成功登录以后，再次访问别的系统，为了鉴别这个用户请求所带的cookie的有效性，需要到SSOIDs中检查这样的映射关系是否存在。

在主要的请求处理方法processRequest()中，可以很清楚的看到SSOAuth的所有功能

1. 如果用户还没有登录过，是第一次登录本系统，会被跳转到login.jsp页面（在后面会解释如何跳转）。用户在提供了用户名和密码以后，就会用handlerFromLogin()这个方法来验证。
2. 如果用户已经登录过本系统，再访问别的应用的时候，是不需要再次登录的。因为浏览器会将第一次登录时产生的cookie和请求一起发送。效验cookie的有效性是SSOAuth的主要功能之一。
3. SSOAuth还能直接效验非login.jsp页面过来的用户名和密码的效验请求。这个功能是用于非web应用的SSO，这在后面的桌面SSO中会用到。
4. SSOAuth还提供logout服务。

下面看看几个主要的功能函数：

|  |
| --- |
| private void handlerFromLogin(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {          String username = request.getParameter("username");          String password = request.getParameter("password");          String pass = (String)accounts.get(username);          if ((pass==null)||(!pass.equals(password)))              getServletContext().getRequestDispatcher("/failed.html").forward(request, response);          else {              String gotoURL = request.getParameter("goto");              String newID = createUID();              SSOIDs.put(newID, username);              Cookie wangyu = new Cookie(cookiename, newID);              wangyu.setDomain(domainname);              wangyu.setMaxAge(60000);              wangyu.setValue(newID);              wangyu.setPath("/");              response.addCookie(wangyu);              System.out.println("login success, goto back url:" + gotoURL);              if (gotoURL != null) {                  PrintWriter out = response.getWriter();                  response.sendRedirect(gotoURL);                  out.close();              }          }      } |

handlerFromLogin()这个方法是用来处理来自login.jsp的登录请求。它的逻辑很简单：将用户输入的用户名和密码与预先设定好的用户集合（存放在accounts中）相比较，如果用户名或密码不匹配的话，则返回登录失败的页面（failed.html），如果登录成功的话，需要为用户当前的session创建一个新的ID，并将这个ID和用户名的映射关系存放到SSOIDs中，最后还要将这个ID设置为浏览器能够保存的cookie值。

登录成功后，浏览器会到哪个页面呢？那我们回顾一下我们是如何使用身份认证服务的。一般来说我们不会直接访问身份服务的任何URL，包括login.jsp。身份服务是用来保护其他应用服务的，用户一般在访问一个受SSOAuth保护的Web应用的某个URL时，当前这个应用会发现当前的用户还没有登录，便强制将也页面转向SSOAuth的login.jsp，让用户登录。如果登录成功后，应该自动的将用户的浏览器指向用户最初想访问的那个URL。在handlerFromLogin()这个方法中，我们通过接收“goto”这个参数来保存用户最初访问的URL，成功后便重新定向到这个页面中。

另外一个要说明的是，在设置cookie的时候，我使用了一个setMaxAge(6000)的方法。这个方法是用来设置cookie的有效期，单位是秒。如果不使用这个方法或者参数为负数的话，当浏览器关闭的时候，这个cookie就失效了。在这里我给了很大的值（1000分钟），导致的行为是：当你关闭浏览器（或者关机），下次再打开浏览器访问刚才的应用，只要在1000分钟之内，就不需要再登录了。我这样做是下面要介绍的桌面SSO中所需要的功能。

其他的方法更加简单，这里就不多解释了。

**3.2.2具有SSO功能的web应用源代码解析**

要实现WEB-SSO的功能，只有身份认证服务是不够的。这点很显然，要想使多个应用具有单点登录的功能，还需要每个应用本身的配合：将自己的身份认证的服务交给一个统一的身份认证服务－SSOAuth。SSOAuth服务中提供的各个方法就是供每个加入SSO的Web应用来调用的。

一般来说，Web应用需要SSO的功能，应该通过以下的交互过程来调用身份认证服务的提供的认证服务：

* Web应用中每一个需要安全保护的URL在访问以前，都需要进行安全检查，如果发现没有登录（没有发现认证之后所带的cookie），就重新定向到SSOAuth中的login.jsp进行登录。
* 登录成功后，系统会自动给你的浏览器设置cookie，证明你已经登录过了。
* 当你再访问这个应用的需要保护的URL的时候，系统还是要进行安全检查的，但是这次系统能够发现相应的cookie。
* 有了这个cookie，还不能证明你就一定有权限访问。因为有可能你已经logout,或者cookie已经过期了，或者身份认证服务重起过，这些情况下，你的cookie都可能无效。应用系统拿到这个cookie，还需要调用身份认证的服务，来判断cookie时候真的有效，以及当前的cookie对应的用户是谁。
* 如果cookie效验成功，就允许用户访问当前请求的资源。

以上这些功能，可以用很多方法来实现：

* 在每个被访问的资源中（JSP或Servlet）中都加入身份认证的服务，来获得cookie，并且判断当前用户是否登录过。不过这个笨方法没有人会用:-)。
* 可以通过一个controller，将所有的功能都写到一个servlet中，然后在URL映射的时候，映射到所有需要保护的URL集合中（例如\*.jsp，/security/\*等）。这个方法可以使用，不过，它的缺点是不能重用。在每个应用中都要部署一个相同的servlet。
* Filter是比较好的方法。符合Servlet2.3以上的J2EE容器就具有部署filter的功能。（Filter的使用可以参考JavaWolrd的文章http://www.javaworld.com/javaworld/jw-06-2001/jw-0622-filters.html）Filter是一个具有很好的模块化，可重用的编程API，用在SSO正合适不过。本样例就是使用一个filter来完成以上的功能。

|  |
| --- |
| package SSO;    import java.io.\*;  import java.net.\*;  import java.util.\*;  import java.text.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  import javax.servlet.\*;  import org.apache.commons.httpclient.\*;  import org.apache.commons.httpclient.methods.GetMethod;    public class SSOFilter implements Filter {      private FilterConfig filterConfig = null;      private String cookieName="WangYuDesktopSSOID";      private String SSOServiceURL= "http://wangyu.prc.sun.com:8080/SSOAuth/SSOAuth";      private String SSOLoginPage= "http://wangyu.prc.sun.com:8080/SSOAuth/login.jsp";        public void init(FilterConfig filterConfig) {            this.filterConfig = filterConfig;          if (filterConfig != null) {              if (debug) {                  log("SSOFilter:Initializing filter");              }          }          cookieName = filterConfig.getInitParameter("cookieName");          SSOServiceURL = filterConfig.getInitParameter("SSOServiceURL");          SSOLoginPage = filterConfig.getInitParameter("SSOLoginPage");      }  .....  .....    } |

以上的初始化的源代码有两点需要说明：一是有两个需要配置的参数SSOServiceURL和SSOLoginPage。因为当前的Web应用很可能和身份认证服务（SSOAuth）不在同一台机器上，所以需要让这个filter知道身份认证服务部署的URL，这样才能去调用它的服务。另外一点就是由于身份认证的服务调用是要通过http协议来调用的（在本样例中是这样设计的，读者完全可以设计自己的身份服务，使用别的调用协议，如RMI或SOAP等等），所有笔者引用了apache的commons工具包（详细信息情访问apache 的网站http://jakarta.apache.org/commons/index.html），其中的“httpclient”可以大大简化http调用的编程。

下面看看filter的主体方法doFilter():

|  |
| --- |
| public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {          if (debug) log("SSOFilter:doFilter()");          HttpServletRequest request = (HttpServletRequest) req;          HttpServletResponse response = (HttpServletResponse) res;          String result="failed";          String url = request.getRequestURL().toString();          String qstring = request.getQueryString();          if (qstring == null) qstring ="";            //检查http请求的head是否有需要的cookie          String cookieValue ="";          javax.servlet.http.Cookie[] diskCookies = request.getCookies();          if (diskCookies != null) {              for (int i = 0; i < diskCookies.length; i++) {                  if(diskCookies[i].getName().equals(cookieName)){                      cookieValue = diskCookies[i].getValue();                        //如果找到了相应的cookie则效验其有效性                      result = SSOService(cookieValue);                      if (debug) log("found cookies!");                  }              }          }          if (result.equals("failed")) { //效验失败或没有找到cookie，则需要登录              response.sendRedirect(SSOLoginPage+"?goto="+url);          } else if (qstring.indexOf("logout") > 1) {//logout服务              if (debug) log("logout action!");              logoutService(cookieValue);              response.sendRedirect(SSOLoginPage+"?goto="+url);          } else {//效验成功              request.setAttribute("SSOUser",result);              Throwable problem = null;              try {                  chain.doFilter(req, res);              } catch(Throwable t) {                  problem = t;                  t.printStackTrace();              }              if (problem != null) {                  if (problem instanceof ServletException) throw (ServletException)problem;                  if (problem instanceof IOException) throw (IOException)problem;                  sendProcessingError(problem, res);              }          }      } |

doFilter()方法的逻辑也是非常简单的，在接收到请求的时候，先去查找是否存在期望的cookie值，如果找到了，就会调用SSOService(cookieValue)去效验这个cookie的有效性。如果cookie效验不成功或者cookie根本不存在，就会直接转到登录界面让用户登录；如果cookie效验成功，就不会做任何阻拦，让此请求进行下去。在配置文件中，有下面的一个节点表示了此filter的URL映射关系：只拦截所有的jsp请求。

<filter-mapping>  
<filter-name>SSOFilter</filter-name>  
<url-pattern>\*.jsp</url-pattern>  
</filter-mapping>

下面还有几个主要的函数需要说明：

|  |
| --- |
| private String SSOService(String cookievalue) throws IOException {          String authAction = "?action=authcookie&cookiename=";          HttpClient httpclient = new HttpClient();          GetMethod httpget = new GetMethod(SSOServiceURL+authAction+cookievalue);          try {              httpclient.executeMethod(httpget);              String result = httpget.getResponseBodyAsString();              return result;          } finally {              httpget.releaseConnection();          }      }        private void logoutService(String cookievalue) throws IOException {          String authAction = "?action=logout&cookiename=";          HttpClient httpclient = new HttpClient();          GetMethod httpget = new GetMethod(SSOServiceURL+authAction+cookievalue);          try {              httpclient.executeMethod(httpget);              httpget.getResponseBodyAsString();          } finally {              httpget.releaseConnection();          }      } |

这两个函数主要是利用apache中的httpclient访问SSOAuth提供的认证服务来完成效验cookie和logout的功能。

其他的函数都很简单，有很多都是我的IDE（NetBeans）替我自动生成的。

***4 当前方案的安全局限性***

当前这个WEB-SSO的方案是一个比较简单的雏形，主要是用来演示SSO的概念和说明SSO技术的实现方式。有很多方面还需要完善，其中安全性是非常重要的一个方面。

我们说过，采用SSO技术的主要目的之一就是加强安全性，降低安全风险。因为采用了SSO，在网络上传递密码的次数减少，风险降低是显然的，但是当前的方案却有其他的安全风险。由于cookie是一个用户登录的唯一凭据，对cookie的保护措施是系统安全的重要环节：

* cookie的长度和复杂度  
  在本方案中，cookie是有一个固定的字符串（我的姓名）加上当前的时间戳。这样的cookie很容易被伪造和猜测。怀有恶意的用户如果猜测到合法的cookie就可以被当作已经登录的用户，任意访问权限范围内的资源
* cookie的效验和保护  
  在本方案中，虽然密码只要传输一次就够了，可cookie在网络中是经常传来传去。一些网络探测工具（如sniff, snoop,tcpdump等）可以很容易捕获到cookie的数值。在本方案中，并没有考虑cookie在传输时候的保护。另外对cookie的效验也过于简单，并不去检查发送cookie的来源究竟是不是cookie最初的拥有者，也就是说无法区分正常的用户和仿造cookie的用户。
* 当其中一个应用的安全性不好，其他所有的应用都会受到安全威胁  
  因为有SSO，所以当某个处于 SSO的应用被黒客攻破，那么很容易攻破其他处于同一个SSO保护的应用。

这些安全漏洞在商业的SSO解决方案中都会有所考虑，提供相关的安全措施和保护手段，例如Sun公司的Access Manager，cookie的复杂读和对cookie的保护都做得非常好。另外在OpneSSO （[https://opensso.dev.java.net](https://opensso.dev.java.net/)）的架构指南中也给出了部分安全措施的解决方案。

***5 当前方案的功能和性能局限性***

除了安全性，当前方案在功能和性能上都需要很多的改进：

* 当前所提供的登录认证模式只有一种：用户名和密码，而且为了简单，将用户名和密码放在内存当中。事实上，用户身份信息的来源应该是多种多样的，可以是来自数据库中，LDAP中，甚至于来自操作系统自身的用户列表。还有很多其他的认证模式都是商务应用不可缺少的，因此SSO的解决方案应该包括各种认证的模式，包括数字证书，Radius， SafeWord ，MemberShip，SecurID等多种方式。最为灵活的方式应该允许可插入的JAAS框架来扩展身份认证的接口
* 我们编写的Filter只能用于J2EE的应用，而对于大量非Java的Web应用，却无法提供SSO服务。
* 在将Filter应用到Web应用的时候，需要对容器上的每一个应用都要做相应的修改，重新部署。而更加流行的做法是Agent机制：为每一个应用服务器安装一个agent，就可以将SSO功能应用到这个应用服务器中的所有应用。
* 当前的方案不能支持分别位于不同domain的Web应用进行SSO。这是因为浏览器在访问Web服务器的时候，仅仅会带上和当前web服务器具有相同domain名称的那些cookie。要提供跨域的SSO的解决方案有很多其他的方法，在这里就不多说了。Sun的Access Manager就具有跨域的SSO的功能。
* 另外，Filter的性能问题也是需要重视的方面。因为Filter会截获每一个符合URL映射规则的请求，获得cookie，验证其有效性。这一系列任务是比较消耗资源的，特别是验证cookie有效性是一个远程的http的调用，来访问SSOAuth的认证服务，有一定的延时。因此在性能上需要做进一步的提高。例如在本样例中，如果将URL映射从“.jsp”改成“/\*”，也就是说filter对所有的请求都起作用，整个应用会变得非常慢。这是因为，页面当中包含了各种静态元素如gif图片，css样式文件，和其他html静态页面，这些页面的访问都要通过filter去验证。而事实上，这些静态元素没有什么安全上的需求，应该在filter中进行判断，不去效验这些请求，性能会好很多。另外，如果在filter中加上一定的cache，而不需要每一个cookie效验请求都去远端的身份认证服务中执行，性能也能大幅度提高。
* 另外系统还需要很多其他的服务，如在内存中定时删除无用的cookie映射等等，都是一个严肃的解决方案需要考虑的问题。

***6 桌面SSO的实现***

从WEB-SSO的概念延伸开，我们可以把SSO的技术拓展到整个桌面的应用，不仅仅局限在浏览器。SSO的概念和原则都没有改变，只需要再做一点点的工作，就可以完成桌面 SSO 的应用。

桌面SSO和WEB-SSO一样，关键的技术也在于如何在用户登录过后保存登录的凭据。在WEB-SSO中，登录的凭据是靠浏览器的cookie机制来完成的；在桌面应用中，可以将登录的凭证保存到任何地方，只要所有SSO的桌面应用都共享这个凭证。

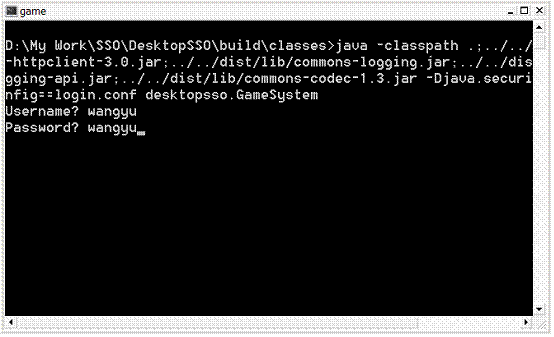
从网站可以下载一个简单的桌面SSO的样例([http://gceclub.sun.com.cn/wangyu/desktop-sso/desktopsso.zip)](http://gceclub.sun.com.cn/wangyu/)和全部源码（http://gceclub.sun.com.cn/wangyu/desktop-sso/desktopsso\_src.zip），虽然简单，但是它具有桌面SSO大多数的功能，稍微加以扩充就可以成为自己的解决方案。

**6.1桌面样例的部署**

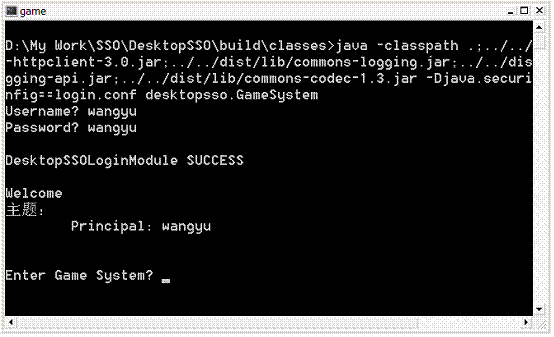
1. 运行此桌面SSO需要三个前提条件：  
   a) WEB-SSO的身份认证应用应该正在运行，因为我们在桌面SSO当中需要用到统一的认证服务  
   b) 当前桌面需要运行Mozilla或Netscape浏览器，因为我们将ticket保存到mozilla的cookie文件中  
   c) 必须在JDK1.4以上运行。（WEB-SSO需要JDK1.5以上）
2. 解开desktopsso.zip文件，里面有两个目录bin和lib。
3. bin目录下有一些脚本文件和配置文件，其中config.properties包含了三个需要配置的参数：  
   a) SSOServiceURL要指向WebSSO部署的身份认证的URL  
   b) SSOLoginPage要指向WebSSO部署的身份认证的登录页面URL  
   c) cookiefilepath要指向当前用户的mozilla所存放cookie的文件
4. 在bin目录下还有一个login.conf是用来配置JAAS登录模块，本样例提供了两个，读者可以任意选择其中一个（也可以都选），再重新运行程序，查看登录认证的变化
5. 在bin下的运行脚本可能需要作相应的修改  
   a) 如果是在unix下，各个jar文件需要用“:”来隔开，而不是“;”  
   b) java 运行程序需要放置在当前运行的路径下，否则需要加上java的路径全名。

**6.2桌面样例的运行**

样例程序包含三个简单的Java控制台程序，这三个程序单独运行都需要登录。如果运行第一个命叫“GameSystem”的程序，提示需要输入用户名和密码：



效验成功以后，便会显示当前登录的用户的基本信息等等。



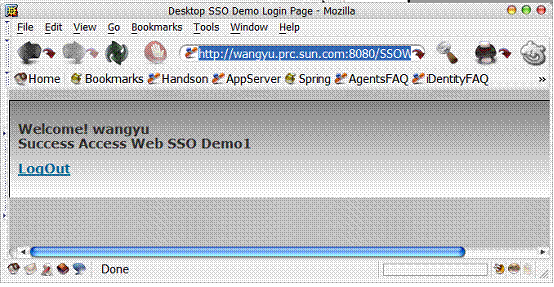
 这时候再运行第二个桌面Java应用（mailSystem）的时候，就不需要再登录了，直接就显示出来刚才登录的用户。



第三个应用是logout，运行它之后，用户便退出系统。再访问的时候，又需要重新登录了。请读者再制裁执行完logout之后，重新验证一下前两个应用的SSO：先运行第二个应用，再运行第一个，会看到相同的效果。

我们的样例并没有在这里停步，事实上，本样例不仅能够和在几个Java应用之间SSO，还能和浏览器进行SSO，也就是将浏览器也当成是桌面的一部分。这对一些行业有着不小的吸引力。

这时候再打开Mozilla浏览器，访问以前提到的那两个WEB应用，会发现只要桌面应用如果登录过，Web应用就不用再登录了，而且能显示刚才登录的用户的信息。读者可以在几个桌面和Web应用之间进行登录和logout的试验，看看它们之间的SSO。



**6.3桌面样例的源码分析**

桌面SSO的样例使用了JAAS（要了解JAAS的详细的信息请参考http://java.sun.com/products/jaas）。JAAS是对PAM（Pluggable Authentication Module）的Java实现，来完成 Java应用可插拔的安全认证模块。使用JAAS作为Java应用的安全认证模块有很多好处，最主要的是不需要修改源代码就可以更换认证方式。例如原有的Java应用如果使用JAAS的认证，如果需要应用SSO，只需要修改JAAS的配置文件就行了。现在在流行的J2EE和其他 Java的产品中，用户的身份认证都是通过JAAS来完成的。在样例中，我们就展示了这个功能。请看配置文件login.conf

|  |
| --- |
| DesktopSSO {     desktopsso.share.PasswordLoginModule required;     desktopsso.share.DesktopSSOLoginModule required;  }; |

当我们注解掉第二个模块的时候，只有第一个模块起作用。在这个模块的作用下，只有test用户（密码是12345）才能登录。当我们注解掉第一个模块的时候，只有第二个模块起作用，桌面SSO才会起作用。

所有的Java桌面样例程序都是标准JAAS应用，熟悉JAAS的程序员会很快了解。JAAS中主要的是登录模块（LoginModule）。下面是SSO登录模块的源码：

|  |
| --- |
| public class DesktopSSOLoginModule implements LoginModule {     ..........     private String SSOServiceURL = "";     private String SSOLoginPage = "";     private static String cookiefilepath = "";     ......... |

在config.properties的文件中，我们配置了它们的值：

|  |
| --- |
| SSOServiceURL=http://wangyu.prc.sun.com:8080/SSOAuth/SSOAuth  SSOLoginPage=http://wangyu.prc.sun.com:8080/SSOAuth/login.jsp  cookiefilepath=C:\\Documents and Settings\\yw137672\\Application Data\\Mozilla\\Profiles\\default\\hog6z1ji.slt\\cookies.txt |

SSOServiceURL和SSOLoginPage成员变量指向了在Web-SSO中用过的身份认证模块：SSOAuth，这就说明在桌面系统中我们试图和Web应用共用一个认证服务。而cookiefilepath成员变量则泄露了一个“天机”：我们使用了Mozilla浏览器的cookie文件来保存登录的凭证。换句话说，和Mozilla共用了一个保存登录凭证的机制。之所以用Mozilla是应为它的Cookie文件格式简单，很容易编程访问和修改任意的Cookie值。（我试图解析Internet Explorer的cookie文件但没有成功。）

下面是登录模块DesktopSSOLoginModule的主体：login()方法。逻辑也是非常简单：先用Cookie来登陆，如果成功，则直接就进入系统，否则需要用户输入用户名和密码来登录系统。

|  |
| --- |
| public boolean login() throws LoginException{          try {              if (Cookielogin()) return true;          } catch (IOException ex) {              ex.printStackTrace();          }        if (passwordlogin()) return true;        throw new FailedLoginException();   } |

下面是Cookielogin()方法的实体，它的逻辑是：先从Cookie文件中获得相应的Cookie值，通过身份效验服务效验Cookie的有效性。如果cookie有效就算登录成功；如果不成功或Cookie不存在，用cookie登录就算失败。

|  |
| --- |
| public boolean Cookielogin() throws LoginException,IOException {        String cookieValue="";        int cookieIndex =foundCookie();        if (cookieIndex<0)              return false;        else              cookieValue = getCookieValue(cookieIndex);       username = cookieAuth(cookieValue);       if (! username.equals("failed")) {           loginSuccess = true;           return true;       }       return false;   } |

用用户名和密码登录的方法要复杂一些，通过Callback的机制和屏幕输入输出进行信息交互，完成用户登录信息的获取；获取信息以后通过userAuth方法来调用远端SSOAuth的服务来判定当前登录的有效性。

|  |
| --- |
| public boolean passwordlogin() throws LoginException {      //      // Since we need input from a user, we need a callback handler      if (callbackHandler == null) {         throw new LoginException("No CallbackHandler defined");      }      Callback[] callbacks = new Callback[2];      callbacks[0] = new NameCallback("Username");      callbacks[1] = new PasswordCallback("Password", false);      //      // Call the callback handler to get the username and password      try {        callbackHandler.handle(callbacks);        username = ((NameCallback)callbacks[0]).getName();        char[] temp = ((PasswordCallback)callbacks[1]).getPassword();        password = new char[temp.length];        System.arraycopy(temp, 0, password, 0, temp.length);        ((PasswordCallback)callbacks[1]).clearPassword();      } catch (IOException ioe) {        throw new LoginException(ioe.toString());      } catch (UnsupportedCallbackException uce) {        throw new LoginException(uce.toString());      }        System.out.println();      String authresult ="";      try {          authresult = userAuth(username, password);      } catch (IOException ex) {          ex.printStackTrace();      }      if (! authresult.equals("failed")) {          loginSuccess= true;          clearPassword();          try {              updateCookie(authresult);          } catch (IOException ex) {              ex.printStackTrace();          }          return true;      }          loginSuccess = false;      username = null;      clearPassword();      System.out.println( "Login: PasswordLoginModule FAIL" );      throw new FailedLoginException();   } |

CookieAuth和userAuth方法都是利用apahce的httpclient工具包和远程的SSOAuth进行http连接，获取服务。

|  |
| --- |
| private String cookieAuth(String cookievalue) throws IOException{          String result = "failed";            HttpClient httpclient = new HttpClient();          GetMethod httpget = new GetMethod(SSOServiceURL+Action1+cookievalue);            try {              httpclient.executeMethod(httpget);              result = httpget.getResponseBodyAsString();          } finally {              httpget.releaseConnection();          }          return result;      }    private String userAuth(String username, char[] password) throws IOException{          String result = "failed";          String passwd= new String(password);          HttpClient httpclient = new HttpClient();          GetMethod httpget = new GetMethod(SSOServiceURL+Action2+username+"&password="+passwd);          passwd = null;            try {              httpclient.executeMethod(httpget);              result = httpget.getResponseBodyAsString();          } finally {              httpget.releaseConnection();          }          return result;        } |

还有一个地方需要补充说明的是，在本样例中，用户名和密码的输入都会在屏幕上显示明文。如果希望用掩码形式来显示密码，以提高安全性，请参考：http://java.sun.com/developer/technicalArticles/Security/pwordmask/

***7 真正安全的全方位SSO解决方案：Kerberos***

我们的样例程序（桌面SSO和WEB-SSO）都有一个共性：要想将一个应用集成到我们的SSO解决方案中，或多或少的需要修改应用程序。Web应用需要配置一个我们预制的filter；桌面应用需要加上我们桌面SSO的JAAS模块（至少要修改JAAS的配置文件）。可是有很多程序是没有源代码和无法修改的，例如常用的远程通讯程序telnet和ftp等等一些操作系统自己带的常用的应用程序。这些程序是很难修改加入到我们的SSO的解决方案中。

事实上有一种全方位的SSO解决方案能够解决这些问题，这就是Kerberos协议（RFC 1510）。Kerberos是网络安全应用标准(<http://web.mit.edu/kerberos/>)，由MIT学校发明，被主流的操作系统所采用。在采用kerberos的平台中，登录和认证是由操作系统本身来维护，认证的凭证也由操作系统来保存，这样整个桌面都可以处于同一个SSO的系统保护中。操作系统中的各个应用（如ftp,telnet）只需要通过配置就能加入到SSO中。另外使用Kerberos最大的好处在于它的安全性。通过密钥算法的保证和密钥中心的建立，可以做到用户的密码根本不需要在网络中传输，而传输的信息也会十分的安全。

目前支持Kerberos的操作系统包括Solaris, windows,Linux等等主流的平台。只不过要搭建一个Kerberos的环境比较复杂，KDC（密钥分发中心）的建立也需要相当的步骤。Kerberos拥有非常成熟的API，包括Java的API。使用Java Generic Security Services(GSS) API并且使用JAAS中对Kerberos的支持（详细信息请参见Sun的Java&Kerberos教程http://java.sun.com/ j2se/1.5.0/docs/guide/security/jgss/tutorials/index.html），要将我们这个样例改造成对Kerberos的支持也是不难的。 值得一提的是在JDK6.0 （http://www.java.net/download/jdk6）当中直接就包含了对GSS的支持，不需要单独下载GSS的包。

***8 总结***

本文的主要目的是阐述SSO的基本原理，并提供了一种实现的方式。通过对源代码的分析来掌握开发SSO服务的技术要点和充分理解SSO的应用范围。但是，本文仅仅说明了身份认证的服务，而另外一个和身份认证密不可分的服务----权限效验，却没有提到。要开发出真正的SSO的产品，在功能上、性能上和安全上都必须有更加完备的考虑。