Apache Shiro

# Shrio简介

## Apache Shiro是什么

Apache Shiro 是功能强大并且容易集成的开源权限框架，它能够完成认证、授权、加密、会话管理、与Web集成、缓存等。

认证和授权为权限控制的核心，简单来说：

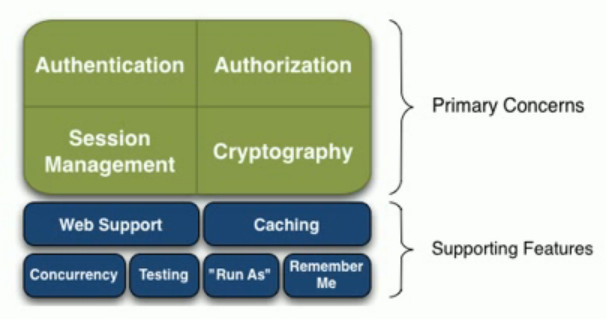
* “认证”就是证明你是谁？Web应用程序一般做法通过表单提交用户名及密码达到认证目的。
* “授权”即是否允许已认证用户访问受保护资源。

## Spring Security 和 Shiro

下面对两者略微比较：

1. 简单性，Shiro在使用上较Spring Security更简单，更容易理解。适合于入门。
2. 灵活性，Shiro可运行在Web、EJB、IoC、Google APP Engine等任何应用环境，却不依赖这些环境。而Spring Security只能与Spring一起集成使用。
3. 可插拔，Shiro干净的API和设计模式使它可以方便地与许多其它的框架和应用进行集成。Shiro可以与诸如Spring、Grails、Wicket、Tapestry、Mule、Apache Camel、Vaadin这类第三方框架无缝集成。而Spring Security在这方面就显得有些捉襟见肘。

# 基本功能



**Authentication**：身份认证/登录，验证用户是不是拥有相应的身份；

**Authorization**：授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限；

**Session Manager**：会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中；会话可以是普通JavaSE环境的，也可以是如Web环境的；

**Cryptography**：加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储；

**Web Support**：Web支持，可以非常容易的集成到Web环境；

**Caching**：缓存，比如用户登录后，其用户信息、拥有的角色/权限不必每次去查，这样可以提高效率；

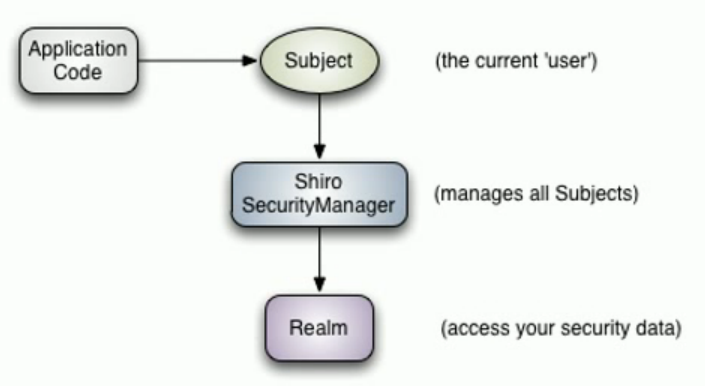
Concurrency：shiro支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去；

Testing：提供测试支持；

Run As：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；

**Remember Me**：记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用重新输入用户名密码进行登录了。

注意：shiro不会去维护用户、维护权限；这些需要我们自己去设计/提供；然后通过相应的接口诸如给shiro即可。



**从应用程序角度来观察如何使用Shiro完成工作**

1. 应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；
2. 我们需要给Shiro的SecurityManager注入Realm，从而让SecurityManager能得到合法的用户及其权限进行判断。

从以上也可以看出，Shiro不提供维护用户/权限，而是通过Realm让开发人员自己注入。

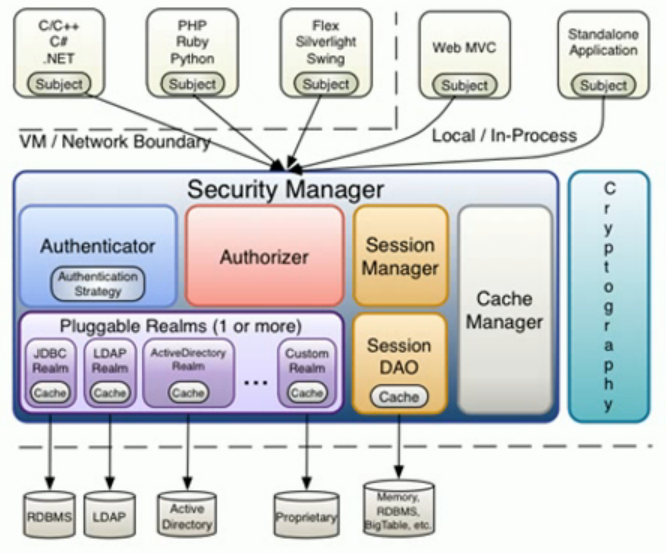
可以看到：应用代码直接交互的对象是Subject，也就是说Shiro的对外API核心就是Subject；其每个API的含义：

* Subject：主体，代表了当前“用户”，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等；即一个抽象概念；所有Subject都绑定到SecurityManager，与Subject的所有交互都会委托给SecurityManager；可以把Subject认为一个门面；SecurityManager才是实际的执行者；
* SecurityManager：安全管理器；即所有与安全有关的操作都会与SecurityManager交互；且它管理着所有Subject；可以看出它是Shiro的核心，它负责与后边介绍的其他组件进行交互，如果学习过Spring MVC，你可以把它看成DispatcherServlet前端控制器；
* Realm：域，Shiro从Realm获取安全数据（如用户、角色、权限），就是说SecurityManager要验证用户身份，那么它需要从Realm获取相应的用户进行比较以确定用户身份是否合法；也需要从Realm得到用户相应的角色/权限进行验证用户是否能进行操作；可以把Realm看成DataSource，即安全数据源。

也就是说对于我们而言，最简单的一个Shiro应用：

1. 应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；
2. 我们需要给Shiro的SecurityManager注入Realm，从让SecurityManager能得到合法的用户即其权限进行判断。

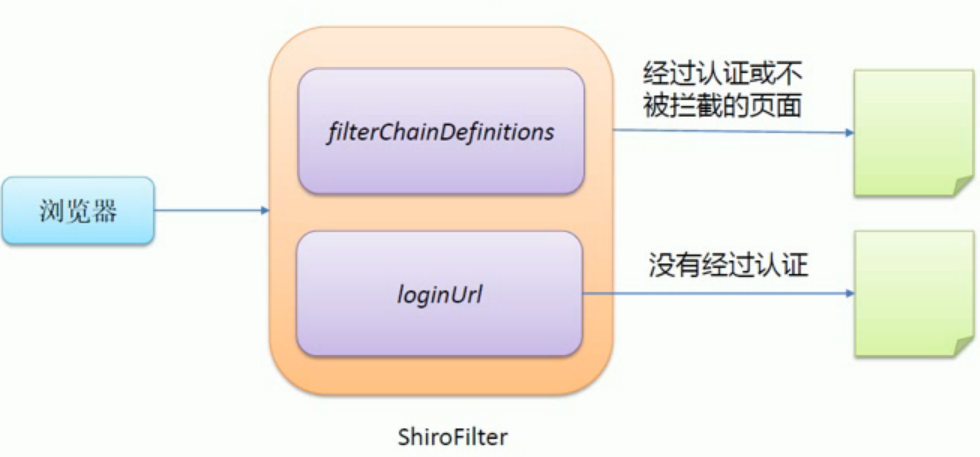
## Shiro的内部架构



## Shiro工作流程和注意事项

|  |
| --- |
| <filter>  <filter-name>shiroFilter</filter-name>  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <init-param>  <param-name>targetFilterLifecycle</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>shiroFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  DelegatingFilterProxy作用是自动到spring容器查找名字为shiroFilter(filter-name)的bean并把所有Filter的操作委托给它。然后将ShiroFilter配置到spring容器即可  <bean id=”shiroFilter” class=”org.apache.shiro.spring.web.ShiroFilterFactoryBean”>  <property name=”securityManager” ref=”securityManager”/>  </bean> |

**ShiroFilter工作原理：**



## URL匹配规则

url模式使用Ant风格模式

Ant路径通配符支持?、\*、\*\*，注意通配符匹配不包括目录分隔符”/”：

* ?：匹配一个字符：如“/admin?”将匹配/admin1，但不匹配/admin；
* \*：匹配零个或多个字符串，如/admin将匹配/admin、/admin123，但不匹配/admin/1；
* \*\*：匹配路径中零个或多个路径，如“/admin/\*\*”将匹配/admin/a或/admin/a/b。

Shiro的内部架构

Shiro内置了很多默认的拦截器，比如身份验证，授权等相关的。默认拦截器可以参考org.apache.shiro.web.filter.mgt.DefaultFilter中的枚举拦截器：

|  |  |  |
| --- | --- | --- |
| 默认烂机器名称 | 拦截器类 | 说明（括号里的表示默认值） |
| 身份验证相关 |  |  |
| authc | org.apache.shiro.web.filter.authc.FormAuthenticationFilter | 基于表单的拦截器：如“/\*\*=authc”，如果没有登录会调到相应的登录页面登录；主要属性：usernameParam：表单提交的用户名参数名（username）；passwordParam：表单提交的密码参数名（password）：rememberMeParam：表示提交的密码参数名（rememberMe）；loginUrl：登录页面地址（/login.jsp）；successUrl：登录成功后的默认重定向地址；failureKeyAttribute：登录失败后错误信息存储key（shiroLoginFailure）； |
| authcBasic | org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter | Basic HTTP身份验证拦截器，主要属性：applicationName：弹出登录框显示的信息（application）； |
| logout | org.apache.shiro.web.filter.authc.LogoutFilter | 退出拦截器，主要属性：redirectUrl：退出成功后重定向的地址（/）；示例“/logout=logout” |
| user | org.apache.shiro.web.filter.authc.UserFilter | 用户拦截器，用户已经身份验证/记住我登录的都可：示例“/\*\*=user” |
| anon | org.apache.shiro.web.filter.authc.AnonymousFilter | 匿名拦截器，即不需要登录即可访问；一般用于静态资源过滤：示例（/static/\*\*=anon） |
| 授权相关 |  |  |
| roles | org.apache.shiro.web.filter.authz.AuthorizationFilter | 角色授权拦截器，验证用户是否拥有所有角色；主要属性：loginUrl：登录页面地址（/login.jsp）；unauthorizedUrl：未授权后重定向的地址：示例：“/admin/\*\*=roles[admin]” |
| perms | org.apache.shiro.web.filter.authz.PermisionsAuthorizationFilter | 权限授权拦截器，验证用户是否拥有所有权限：属性和roles一样：示例“/user/\*\*=perms[“user.create”]” |
| port | org.apache.shiro.web.filter.authz.PortFilter | 端口拦截器，主要属性：port(80):可以通过的端口；示例“/test=port[80]”，如果用户访问该页面是非80端口，将自动将请求端口改为80并重定向到该80端口，其它路径/参数等都一样 |
| rest | org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter | Rest风格拦截器，自动根据请求方法构建权限字符串（GET=read,POST=create,PUT=update,DELETE=delete,HEAD=read,TRACE=read,OPTIONS=read,MKCOL=create）构建权限字符串；示例“/users=rest[user]”，会自动拼出“user:read,user:create,user:update,user:delete”权限字符串进行权限匹配（所有都得匹配，isPermittedAll） |
| ssl | org.apache.shiro.web.filter.authz.SslFilter | SSL拦截器，只有请求协议是https才能通过；否则自动跳转回https端口(443)；其它和port拦截器一样； |
| 其他 |  |  |
| noSeesionCreation | org.apache.shiro.web.filter.session.NoSessionCreationFilter | 不创建会话拦截器，条用subject.getSession(false)不会有什么问题，但是如果subject.getSession(true)将会抛出DisabledSeesionException异常； |

URL模式匹配顺序

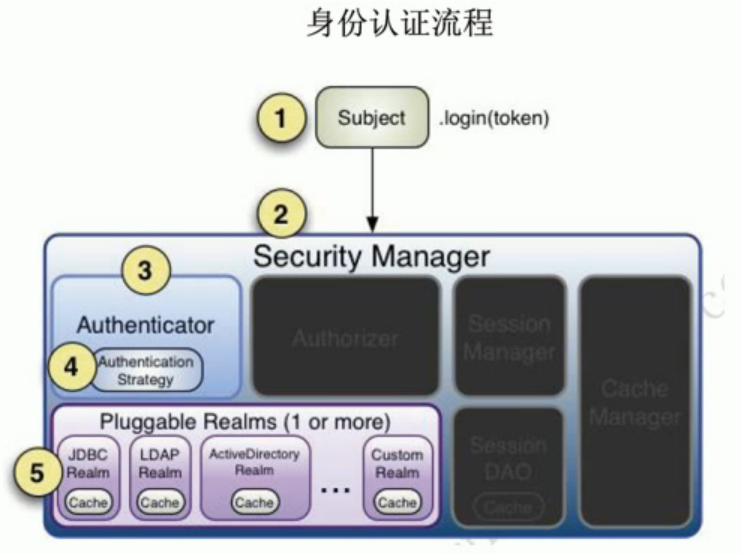
url模式匹配顺序是按照在配置中的声明顺序匹配， 即从头开始使用第一个匹配的url模式对应的拦截器链。如：

/bb/\*\*=filter1

/bb/aa=filter2

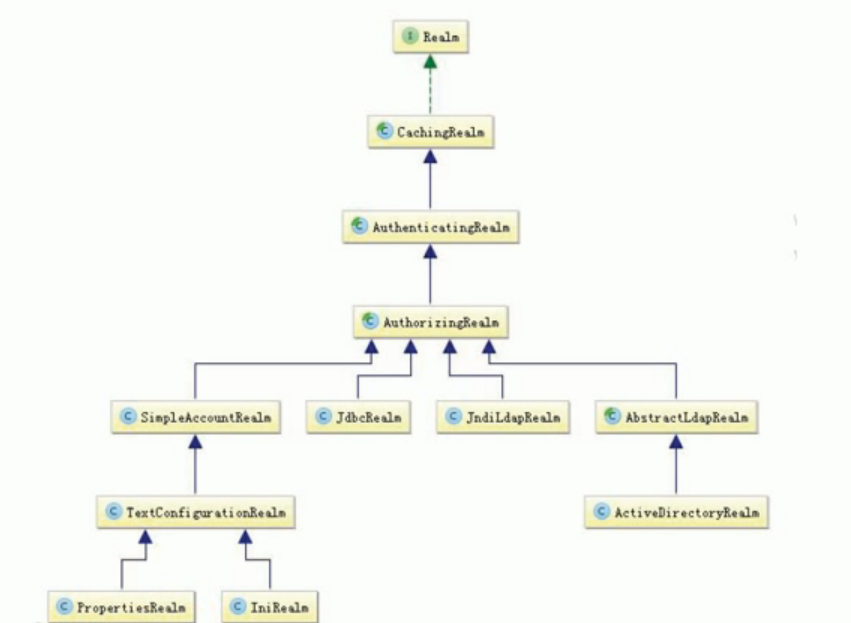
/\*\*=filter3

如果请求的url是“/bb/aa”，因为按照声明顺序进行匹配，那么将使用filter1进行拦截。



|  |
| --- |
| 1、获取当前subject.SecurityUtils.getSubject() 2、判断当前用户是否已经认证，subject.isAuthenticated() 3、如果没有被认证，则需要把用户名和密码封装为UsernamePasswordToken  3.1、 创建一个提交页面  3.2、 将用户名密码输入后提交到controller  3.3、 在controller中获取用户名和密码 4、执行登录操作subject.login(token) 5、最后登录信息会跳转到realm中，在realm中进行获取数据的操作，并返回realm  5.1、 如果只进行认证，则继承AutheticatingRealm  5.2、 实现getAuthenticationInfo回调方法 6、由shiro完成密码的对比工作 |

## 认证流程实现



一般集成AuthorizingRealm（授权）即可；其集成了AuthenticatingRealm（即身份验证），而且也间接集成了CachingRealm（带有缓存实现）。其中主要默认实现如下：

org.apache.shiro.realm.text.IniRealm：[users]部分指定用户名/密码及其角色；[roles]部分指定角色即权限信息；

org.apache.shiro.realm.text.PropertiesRealm：user.username=password,role1,role2之规定用户名/密码及其角色；role.role1=permission1,permission2指定角色其权限信息；

org.apache.shiro.realm.jdbc.JdbcRealm：通过sql查询相应的信息，如“select password from users where username = ?”获取用户密码，“select password, password\_salt from users where username = ?”获取用户密码及盐；“select role\_name from user\_roles where username = ?”获取用户角色；“select permission from roles\_permissions where role\_name = ?”获取角色对应的权限信息；也可以调用相应的api进行自定义sql；

## 多realm认证和策略

Authenticator

Authenticator的职责是验证用户账号，是Shiro API中身份验证核心的入口点：

public AuthenticationInfo authenticate(AuthenticationToken authenticationToken) throws AuthenticationException;

如果验证成功，将返回AuthenticationInfo验证信息；此信息中包含了身份及凭证；如果验证失败抛出相应的AuthenticationException实现。

SecurityManager接口继承了Authenticator，另外还有一个ModularRealmAuthenticator实现，其委托给多个Realm进行验证，验证规则通过AuthenticationStrategy接口指定，默认提供的实现：

FirstSuccessfulStrategy:只要有一个Realm验证成功即可，只返回第一个Realm身份验证成功的认证信息，其他的忽略；

AtLeastOneSuccessfulStrategy:只要有一个Realm验证成功即可，和FirstSuccessfulStrategy不同，返回所有Realm身份验证成功的认证信息；

AllSuccessfulStrategy:所有Realm验证成功才算成功，且返回所有Realm身份验证成功的认证信息，如果有一个失败就失败了。

ModularRealmAuthenticator默认使用AtLeastOneSuccessfulStrategy策略。

## 授权流程原理

授权

授权，也叫访问控制，即在应用中控制谁能访问那些资源（如访问页面/编辑数据/页面操作等）。在授权中需了解的几个关键对象：主体（Subject）、资源（Resource）、权限（Permission）、角色（Role）。

主体

主体，即访问应用的用户，在Shiro中使用Subject代表该用户。用户只有授权后才允许访问相应的资源。

资源

在应用中用户可以访问的任何东西，比如访问JSP页面、查看/编辑某些数据、访问某个业务方法、打印文本等等都是资源。用户只要授权后才能访问。

权限

安全策略中的原子授权单位，通过权限我们可以表示在应用中用户有没有操作某个资源的权利。即权限表示在应用中用户能不能访问某个资源，如：

访问用户列表页面

查看/新增/修改/删除用户数据（即很多时候都是CRUD（增删查改）式权限控制）

打印文档等等。。。

角色

角色代表了操作集合，可以理解为权限的集合，一般情况下我们会赋予用户角色而不是权限，即这样用户可以拥有一组权限，赋予权限时比较方便。典型的如：项目经理、技术总监、即这样用户可以拥有一组权限，赋予权限时比较方便。典型的如：项目经理、技术总监、CTO、开发工程师等都是角色，不同的角色拥有一组不同的权限。

### 授权方式

Shrio支持三种方式的授权：

编程式：通过写if/else授权代码块完成：

|  |
| --- |
| Subject subject = SecurityUtils.getSubject();  If(subject.hasRole(“admin”)){  //有权限  } else {  //无权限  } |

注解方式：通过在执行的Java方法上放置相应的注解完成：

|  |
| --- |
| @RequiresRoles(“admin”)  public void hello() {  //有权限  }  没有权限将抛出相应的异常； |

JSP/GSP 标签：在JSP/GSP页面通过相应的标签完成：

|  |
| --- |
| <shiro:hasRole name=”admin”>  </-有权限->  </shiro:hasRole> |

## 标签

JSP标签

Shiro提供了JSTL标签用于在JSP/GSP页面进行权限控制，如根据登录用户显示相应的页面按钮。

导入标签库

<%@taglib prefix=”shiro” uri=”http://shiro.apache.org/tags” %>

标签库定义在shiro-web.jar包下的META-INF/shiro.tld中定义。

guest标签

<shiro:guest>

欢迎游客访问，<a href=”${pageContext.request.contextPath}/login.jsp”>登录</a>

</shiro:guest>

用户没有身份验证时显示相应信息，即游客访问信息。

user标签

<shiro:user>

欢迎[<shiro:principal/>]登录，<a href=”${pageContext.request.contextPath}/logout”>退出</a>

</shiro:user>

用户已经身份验证/记住我登录后显示相应的信息。

authenticated标签

<shiro:authenticated>

用户[<shiro:principal/>]已身份验证通过

</shiro:authenticated>

用户已经身份验证通过，即Subject.login登录成功，不是记住我登录的。

notAuthenticated标签

<shrio: notAuthenticated>

未身份验证（包括记住我）

</shrio: notAuthenticated>

用户未进行身份验证，即没有调用Subject.login进行登录，包括记住我自动登录的也属于未进行身份验证。

principal标签

<shiro:principal/>

显示用户身份信息，默认调用Subject.getPrincipal()获取，即Primary Principal.

<shiro:principal property=”username”/>

相当于((User)Subject.getPrincipals()).getUsername()。

lacksPermission标签

<shiro:lacksPermission name=”org:create”>

用户[<shiro:principal/>]没有权限org:create<br/>

</shiro:lacksPermission>

如果当前Subject没有权限将显示body体内容。

hasRole标签

<shiro:hasRole name=”admin”>

用户[<shiro:principal/>]拥有角色admin<br/>

</shiro:hasRole>

如果当前Subject有角色将显示body内容。

hasAnyRoles标签

<shiro:hasAnyRoles name=”admin,user”>

用户[<shiro:principal/>]拥有角色admin或user<br/>

</shrio:hasAnyRoles>

如果当前Subject有任意一个角色（或的关系）将显示body体内容。

lacksRole标签

<shiro:lacksRole name=”abc”>

用户[<shiro:principal/>]没有角色abc<br/>

</shiro:lacksRole>

如果当前Subject没有角色将显示body体内容。

hasPermission标签

<shiro:hasPermission name=”user:create”>

用户[<shiro:principal/>]拥有权限user:create<br/>

</shiro:hasPermission>

如果当前Subject有权限将显示body体内容。

## 权限注解

Shiro权限注解

Shiro提供了相应的注解用于权限控制，如果使用这些注解就需要使用AOP的功能进行判断，如Spring AOP；Shiro提供了Spring AOP 集成用于权限注解的解析和验证。

@RequiresAuthentication

表示当前Subject已经通过login进行了身份验证；即Subject.isAuthenticated()返回true.

@RequiresUser

表示当前Subject已经身份验证或者通过记住我登录的。

@RequiresGuest

表示当前Subject没有身份验证或通过记住我登录过，即是游客身份。

@RequiresRoles(value={“admin”,”user”}，logical=Logical.AND)

表示当前Subject需要角色admin和user。

@RequiresPermissions(value={“user:a”,”user:b”}，logical=Logical.OR)

表示当前Subject需要权限user:a或user:b。

## 数据库中初始化资源及权限



## 会话管理

会话管理

Shiro提供了完整的企业级会话管理功能，不依赖于底层容器（如web容器tomcat），不管JavaSE还是JavaEE环境都可以使用，提供了会话管理、会话事件监听、会话存储/持久化、容器无关的集群、失效/过期支持、对Web的透明支持、SSO单点登录的支持等特性。即直接使用Shiro的会话管理可以直接替换如Web容器的会话管理。

会话

所谓会话，即用户访问应用时保持的链接关系，在多次交互中应用能够识别出当前访问的用户是谁，且可以在多次交互中保存一些数据。如访问一些网站时登录成功后，网站可以记住用户，且在退出之前都可以识别当前用户是谁。

Shiro提供的会话可以用于JavaSE/JavaEE环境，不依赖于任何底层容器，可以独立使用，是完整的会话模块。

API

登录成功后使用Subject.getSession()即可获取会话；其等价于Subject.getSession(true)，即如果当前没有创建Session对象会创建一个；另外Subject.getSession(false)，如果当前没有创建Session则返回null（不过默认情况下如果启用会话存储功能的话在创建Subject时会主动创建一个Session）。

session.getId();

获取当前会话的唯一标识。

session.getHost();

获取当前Subject的主机地址，该地址是通过HostAuthenticationToken.getHost()提供的。

session.getTimeout();

session.setTimeout(毫秒);

获取/设置当前Session的过期时间；如果不设置默认是会话管理器的全局过期时间。

session.getStartTimestamp();

session.getLastAccessTime();

获取会话的启动时间及最后访问时间；如果是JavaSE应用需要自己定期调用session.touch()去更新最后访问时间；如果是Web应用，每次进入ShiroFilter都会自动调用session.touch()来更新最后访问时间。

session.touch();

session.stop();

更新会话最后访问时间及销毁会话；当Subject.logout()时会自动调用stop方法来销毁会话。如果在web中，调用javax.servlet.http.HttpSession.invalidate()也会自动调用Shiro Session.stop()方法进行销毁shiro的会话。

session.setAttribute(“key”, “value”);

Assert.assertEquals(“value”, session.getAttribute(“key”));

Session.removeAttribute(“key”);

设置/获取/删除会话属性；在整个会话范围内都可以对这些属性进行操作。

## 会话管理器

会话管理器

会话管理器管理着应用中所有Subject的会话的创建、维护、删除、失效、验证等工作。是Shiro的核心组件，顶层组件SecurityManager直接继承了SessionManager，且提供了SessionsSecurityManager实现直接把会话管理委托给相应的SessionManager，DefaultSecurityManager及DefaultWebSecurityManager默认SecurityManager都继承了SessionsSecurityManager。

SecurityManager提供了如下接口：

Session start(SessionContext context); // 启动会话

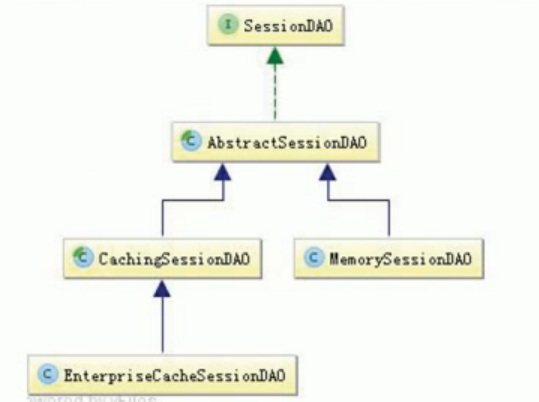
Session getSession(SessionKey key) throws SessionException; // 根据会话key获取会话。

会话监听器

会话监听器用于监听会话创建、过期及停止事件

|  |
| --- |
| public class MySessionListener1 implements SessionListener {  @Overriide  public void onStart(Session session) { // 会话创建时触发发  System.out.println(“会话创建：” + session.getId());  }  @Overriide  public void onExpiration(Session session) { // 会话过期时触发  System.out.println(“会话过期：” + session.getId());  }  @Overriide  public void onStop(Session session) { // 退出/会话过期时触发  System.out.println(“会话停止：” + session.getId());  }  } |

SessionDao



Shrio内嵌如上SessionDao实现：

AbstractSessionDAO提供了SessionDAO的基础实现，如生成会话ID等；CachingSessionDAO提供了对开发者透明的会话缓存的功能，只需要设置相应的CacheManager即可；

MemorySessionDAO直接在内存中进行会话维护；而EnterpriseCacheSessionDAO提供了缓存功能的会话维护，默认情况下使用MapCache实现，内部使用ConcurrentHashMap保存缓存的会话。

会话存储/持久化

Shiro提供SessionDAO用于会话的CRUD，即DAO（Data Access Object）模式实现：

如DefaultSessionManager在创建完session后悔调用该方法；如保存到关系数据库/文件系统/NoSQL数据库；即可以实现会话的持久化；返回会话ID；主要此处返回的ID.equals(session.getId());

Serializable create(Session session);

根据会话ID获取会话

Session readSession(Serializable sessionId) throws UnknownSessionException;

更新会话；如更新会话最后访问时间/停止会话/设置超时时间/设置移除属性等会调用

Void update(Session session) throws UnknownSessionException;

删除会话；当会话过期/会话停止（如用户退出时）会调用

Void delete(Session session);

获取当前所有活跃用户，如果用户量多此方法影响性能

会话验证

Shiro提供了会话验证调度器，用于定期的验证会话是否过期，如果过期将停止会话；处于性能考虑，一般情况下都是获取会话时来验证会话是否过期并停止会话的；但是如在Web环境中，如果用户不主动退出时不知道会话是否过期的，因此需要定期的检测会话是否过期，Shiro提供了会话验证调度器SessionValidationScheduler来做这件事情。

## 缓存机制

Shiro提供了类似于Spring的Cache抽象，即Shiro本身不实现Cache，但是对Cache进行了又抽象，方便更换不同的底层Cache实现。

Shrio提供的Cache接口：

Shrio提供的CacheManager接口：

Shiro还提供了CacheManagerAware用于注入CacheManager：

|  |
| --- |
| Public interface Cache<K, V> {  // 根据Key获取缓存中的值  Public V get(K key) throws CacheException;  // 往缓存中放入key-value，返回缓存中之前的值  Public V put(K key, V value) throws CacheException;  // 移除缓存中key对应的值，返回该值  Public V remove(K key) throws CacheException;  // 清空整个缓存  Public void clear() throws CacheException;  Public int size(); //返回缓存大小  Public Set<K> keys(); // 获取缓存中所有的key  // 获取缓存中所有的value  Public Collection<V> values();  } |

接口

Shiro提供的CacheManager接口

|  |
| --- |
| Public interface CacheManager {  // 根据缓存名字获取一个Cache  Public <K, V> Cache<K, V> getCache(String name) throws CacheException;  } |

Shrio还提供了CacheManagerAware用于注入CacheManager：

|  |
| --- |
| Public interface CacheManagerAware {  // 注入CacheManager  Void setCacheManager(CacheManager cacheManager);  } |

Shiro内部相应的组件（DefaultSecurityManager）会自动检测相应的对象（如Realm）是否实现了CacheManagerAware并自动注入相应的CacheManager。

Realm缓存

Shiro提供了CachingRealm，其实现了CacheManagerAware接口，提供了缓存的一些基础实现；另外AuthenticatingRealm及AuthorizingRealm分别提供了对AuthenticationInfo和AuthorizationInfo信息的缓存。

Session缓存

如securityManager实现了SessionSecurityManager，其会自动判断SessionManager是否实现了CacheManagerAware接口，如果实现了会把CacheManager设置给它。然后sessionManager会判断相应的sessionDAO（如集成自CachingSessionDAO）是否实现了CacheManagerAware，如果实现了会把CacheManager设置给它；

缓存的SessionDAO，其会先查缓存，如果找不到才查数据库。

## RememberMe

Shiro提供了记住我（RememberMe）的功能，比如访问如淘宝等一些网站时，关闭了浏览器下次再打开时还是能记住你是谁，下次访问时无需再等了即可访问，基本流程如下：

1. 首先在登录页面选中RememberMe然后登录成功；如果是浏览器登录，一般会把RememberMe的Cookie写到客户端并保存下来；
2. 关闭浏览器再重新打开；会发现浏览器还是记住你的；
3. 访问一般的网页服务器端还是知道你是谁，且能正常访问；
4. 但是比如我们访问淘宝时。如果要查看我的订单或进行支付时，此时还是需要再进行身份认证，以确保当前用户还是你。

判断方法

Subject.isAuthenticated()表示用户进行了身份验证登录的，即使有Subject.login进行了登录；

Subject.isRemembered()：表示用户是通过记住我登录的，此时可能并不是真正的你（如你的朋友使用你的电脑，或者你的cookie被窃取）在访问的；且两者二选一，即subject.isAuthenticated()==true，则subject.isRemembered()==false；反之一样。

建议，一般这样使用：

访问一般网页，如个人在主页之类的，我们使用user拦截器即可，user拦截器只要用户登录（isRemembered()==true or isAuthenticated()==true）过即可访问成功；

访问特殊网页，如我的订单，提交订单页面，我们使用authc拦截器即可，authc拦截器会判断用户是否通过subject.login(isAuthenticated()==true)登录的，如果是才放行，否则会跳转到登录页面叫你重新登录。

因此RememberMe使用过程中，需要配合相应的拦截器来实现相应的功能，用错了拦截器可能就不满足你的需求了。