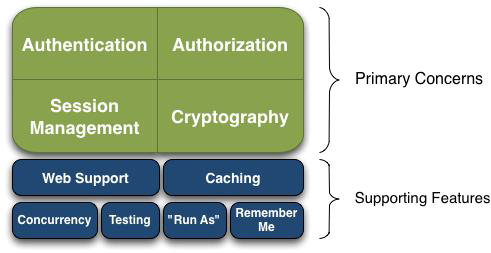
# 1.shiro简介



Authentication:身份验证/登录，验证用户有没有相应的身份。

Authorization:授权，验证已认证的用户有没有相应的权限。即判断用户有没有权限做某个事情，常见：验证某个用户有没有某个角色，或者细粒度验证某个用户对于某个资源是否具有某个权限。

Session Management:会话管理。用户登录后就是一次会话，会话结束前，相关的信息都在会话中。

Cryptography:加密，保证数据的安全性，如密码加密存储到数据库，而不是名文。

Web Support:shiro有良好的Web支持，能很容易的集成到web环境。

Caching:缓存。如用户登录后，用户信息/角色信息/权限等存入缓存，提高效率。

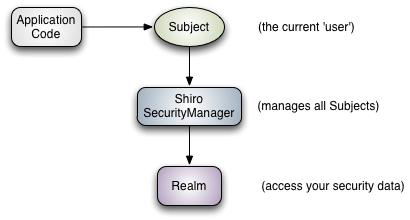
Concurrency:shiro 支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能

把权限自动传播过去。

Testing: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:登录时记住我的功能。



Subject:相当于当前的“用户”，这个用户不一定是一个具体的人，与当前应用交互

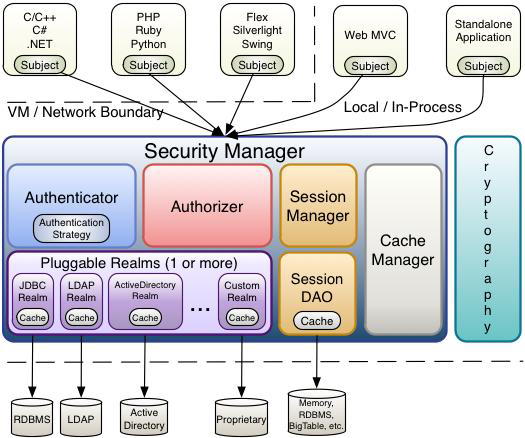
的任何东西都是Subject，如网络爬虫，机器人等；即一个抽象概念；所有Subject 都绑定

到SecurityManager，与Subject的所有交互都会委托给SecurityManager；可以把Subject认

为是一个门面；SecurityManager才是实际的执行者。

SecurityManager:安全管理器，管理所有的subject，所有与安全相关的操作都要与SecurityManager交互。

Realm:域。shiro从realm中获取安全数据(如用户、角色、权限等)，SecurityManager验证用户时需要从realm中获取用户的数据来确定用户身份是否合法。Realm可以看作DataSource，即安全数据源。



**Subject：**主体，可以看到主体可以是任何可以与应用交互的“用户”；

**SecurityManager：**相当于SpringMVC 中的DispatcherServlet 或者Struts2 中的

FilterDispatcher；是Shiro的心脏；所有具体的交互都通过SecurityManager进行控制；它管

理着所有Subject、且负责进行认证和授权、及会话、缓存的管理。

**Authenticator：**认证器，负责主体认证的，这是一个扩展点，如果用户觉得Shiro 默认的

不好，可以自定义实现；其需要认证策略（Authentication Strategy），即什么情况下算用户

认证通过了。

**Authrizer：**授权器，或者访问控制器，用来决定主体是否有权限进行相应的操作；即控制

着用户能访问应用中的哪些功能。

**Realm：**可以有1个或多个Realm，可以认为是安全实体数据源，即用于获取安全实体的；

可以是JDBC 实现，也可以是LDAP 实现，或者内存实现等等；由用户提供；注意：Shiro不知道你的用户/权限存储在哪及以何种格式存储；所以我们一般在应用中都需要实现自己

的Realm。

**SessionManager：**如果写过Servlet就应该知道Session的概念，Session呢需要有人去管理

它的生命周期，这个组件就是SessionManager；而Shiro 并不仅仅可以用在Web 环境，也

可以用在如普通的JavaSE 环境、EJB 等环境；所有呢，Shiro 就抽象了一个自己的Session

来管理主体与应用之间交互的数据；这样的话，比如我们在Web 环境用，刚开始是一台

Web 服务器；接着又上了台EJB 服务器；这时想把两台服务器的会话数据放到一个地方，

这个时候就可以实现自己的分布式会话（如把数据放到Memcached服务器）；

**SessionDAO：**DAO 大家都用过，数据访问对象，用于会话的CRUD，比如我们想把Session

保存到数据库，那么可以实现自己的SessionDAO，通过如JDBC 写到数据库；比如想把

Session 放到Memcached 中，可以实现自己的Memcached SessionDAO；另外SessionDAO

中可以使用Cache进行缓存，以提高性能；

**CacheManager：**缓存控制器，来管理如用户、角色、权限等的缓存的；因为这些数据基本

上很少去改变，放到缓存中后可以提高访问的性能

**Cryptography：**密码模块，Shiro 提高了一些常见的加密组件用于如密码加密/解密的。

# 身份验证

**身份验证**，即在应用中谁能证明他就是他本人。一般提供如他们的身份ID 一些标识信息来

表明他就是他本人，如提供身份证，用户名/密码来证明。

在 shiro 中，用户需要提供principals （身份）和credentials（证明）给shiro，从而应用能

验证用户身份：

**principals**：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。

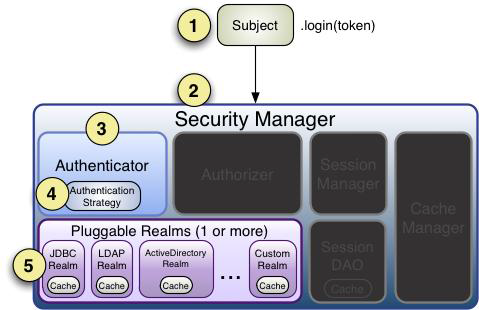
一个主体可以有多个principals，但只有一个Primary principals，一般是用户名/密码/手机号。

**credentials**：证明/凭证，即只有主体知道的安全值，如密码/数字证书等。

最常见的principals和credentials组合就是用户名/密码了。接下来先进行一个基本的身份认

证。

另外两个相关的概念是之前提到的**Subject**及**Realm**，分别是主体及验证主体的数据源。



流程如下：

1、首先调用Subject.login(token)进行登录，其会自动委托给Security Manager，调用之前必

须通过SecurityUtils. setSecurityManager()设置；

2、SecurityManager负责真正的身份验证逻辑；它会委托给Authenticator进行身份验证；

3、Authenticator才是真正的身份验证者，Shiro API中核心的身份认证入口点，此处可以自

定义插入自己的实现；

4、Authenticator可能会委托给相应的AuthenticationStrategy进行多Realm身份验证，默认

ModularRealmAuthenticator会调用AuthenticationStrategy进行多Realm身份验证；

5、Authenticator 会把相应的token 传入Realm，从Realm 获取身份验证信息，如果没有返

回/抛出异常表示身份验证失败了。此处可以配置多个Realm，将按照相应的顺序及策略进

行访问。

# 授权

授权，也叫访问控制，即在应用中控制谁能访问哪些资源（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

权力。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（增查改删）式权限控制）

打印文档等等。。。

如上可以看出，权限代表了用户有没有操作某个资源的权利，即反映在某个资源上的操作

允不允许，不反映谁去执行这个操作。所以后续还需要把权限赋予给用户，即定义哪个用

户允许在某个资源上做什么操作（权限），Shiro 不会去做这件事情，而是由实现人员提供。

Shiro 支持粗粒度权限（如用户模块的所有权限）和细粒度权限（操作某个用户的权限，即

实例级别的），后续部分介绍。

**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、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。

**隐式角色**：即直接通过角色来验证用户有没有操作权限，如在应用中CTO、技术总监、开

发工程师可以使用打印机，假设某天不允许开发工程师使用打印机，此时需要从应用中删

除相应代码；再如在应用中CTO、技术总监可以查看用户、查看权限；突然有一天不允许

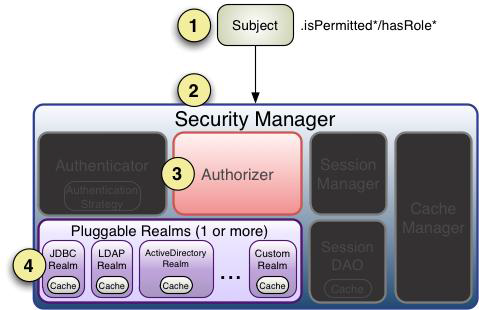
技术总监查看用户、查看权限了，需要在相关代码中把技术总监角色从判断逻辑中删除掉；

即粒度是以角色为单位进行访问控制的，粒度较粗；如果进行修改可能造成多处代码修改。

**显示角色**：在程序中通过权限控制谁能访问某个资源，角色聚合一组权限集合；这样假设

哪个角色不能访问某个资源，只需要从角色代表的权限集合中移除即可；无须修改多处代

码；即粒度是以资源/实例为单位的；粒度较细。



流程如下：

1、首先调用Subject.isPermitted\*/hasRole\*接口，其会委托给SecurityManager，而

SecurityManager接着会委托给Authorizer；

2、Authorizer是真正的授权者，如果我们调用如isPermitted(“user:view”)，其首先会通过

PermissionResolver把字符串转换成相应的Permission实例；

3、在进行授权之前，其会调用相应的Realm获取Subject相应的角色/权限用于匹配传入的

角色/权限；

4、Authorizer会判断Realm的角色/权限是否和传入的匹配，如果有多个Realm，会委托给

ModularRealmAuthorizer 进行循环判断，如果匹配如isPermitted\*/hasRole\*会返回true，否

则返回false表示授权失败。

# 编码/加密

PasswordService/CredentialsMatcher

Shiro 提供了PasswordService及CredentialsMatcher用于提供加密密码及验证密码服务。

public interface PasswordService {

//输入明文密码得到密文密码

String encryptPassword(Object plaintextPassword) throws IllegalArgumentException;

}

public interface CredentialsMatcher {

//匹配用户输入的token 的凭证（未加密）与系统提供的凭证（已加密）

boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info);

}

Shiro 默认提供了PasswordService 实现DefaultPasswordService；CredentialsMatcher 实现

PasswordMatcher及HashedCredentialsMatcher（更强大）。

**HashedCredentialsMatcher实现密码验证服务**

Shiro 提供了CredentialsMatcher 的散列实现HashedCredentialsMatcher，和之前的

PasswordMatcher不同的是，它只用于密码验证，且可以提供自己的盐，而不是随机生成盐，

且生成密码散列值的算法需要自己写，因为能提供自己的盐。

可通过继承HashedCredentialsMatcher实现密码重试次数限制

**密码重试次数限制**

如在1 个小时内密码最多重试5 次，如果尝试次数超过5 次就锁定1 小时，1 小时后可再

次重试，如果还是重试失败，可以锁定如1 天，以此类推，防止密码被暴力破解。我们通

过继承HashedCredentialsMatcher，且使用Ehcache记录重试次数和超时时间。

public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

String username = (String)token.getPrincipal();

//retry count + 1

Element element = passwordRetryCache.get(username);

if(element == null) {

element = new Element(username , new AtomicInteger(0));

passwordRetryCache.put(element);

}

AtomicInteger retryCount = (AtomicInteger)element.getObjectValue();

if(retryCount.incrementAndGet() > 5) {

//if retry count > 5 throw

throw new ExcessiveAttemptsException();

}

boolean matches = super.doCredentialsMatch(token, info);

if(matches) {

//clear retry count

passwordRetryCache.remove(username);

}

return matches;

}

如果密码输入正确清除cache中的记录；否则cache中的重试次

数+1，如果超出5次那么抛出异常表示超出重试次数了。

# Realm及相关对象

**看一个示例UserRealm**

public class UserRealm extends AuthorizingRealm {

private UserService userService = new UserServiceImpl();

protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {

String username = (String)principals.getPrimaryPrincipal();

SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();

authorizationInfo.setRoles(userService.findRoles(username));

authorizationInfo.setStringPermissions(userService.findPermissions(username));

return authorizationInfo;

}

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws

AuthenticationException {

String username = (String)token.getPrincipal();

User user = userService.findByUsername(username);

if(user == null) {

throw new UnknownAccountException();//没找到帐号

}

if(Boolean.TRUE.equals(user.getLocked())) {

throw new LockedAccountException(); //帐号锁定

}

//交给AuthenticatingRealm使用CredentialsMatcher进行密码匹配，如果觉得人家

的不好可以在此判断或自定义实现

SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(

user.getUsername(), //用户名

user.getPassword(), //密码

ByteSource.Util.bytes(user.getCredentialsSalt()),//salt=username+salt

getName() //realm name

);

return authenticationInfo;

}

}

1. **UserRealm 父类AuthorizingRealm 将获取Subject 相关信息分成两步**：获取身份验证

信息（doGetAuthenticationInfo）及授权信息（doGetAuthorizationInfo）；

**2、doGetAuthenticationInfo 获取身份验证相关信息**：首先根据传入的用户名获取User 信

息；然后如果user 为空，那么抛出没找到帐号异常UnknownAccountException；如果user

找到但锁定了抛出锁定异常LockedAccountException；最后生成AuthenticationInfo 信息，

交给间接父类AuthenticatingRealm使用CredentialsMatcher进行判断密码是否匹配，如果不

匹配将抛出密码错误异常IncorrectCredentialsException；另外如果密码重试此处太多将抛出

超出重试次数异常ExcessiveAttemptsException；在组装SimpleAuthenticationInfo 信息时，

需要传入：身份信息（用户名）、凭据（密文密码）、盐（username+salt），CredentialsMatcher

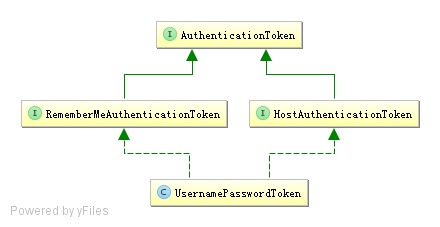
使用盐加密传入的明文密码和此处的密文密码进行匹配。

**3、doGetAuthorizationInfo获取授权信息**：PrincipalCollection是一个身份集合，因为我们

现在就一个Realm，所以直接调用getPrimaryPrincipal得到之前传入的用户名即可；然后根

据用户名调用UserService接口获取角色及权限信息。

AuthenticationToken



AuthenticationToken 用于收集用户提交的身份（如用户名）及凭据（如密码）：

public interface AuthenticationToken extends Serializable {

Object getPrincipal(); //身份

Object getCredentials(); //凭据

}

扩展接口RememberMeAuthenticationToken：提供了“boolean isRememberMe()”现“记住

我”的功能；

扩展接口是HostAuthenticationToken：提供了“String getHost()”方法用于获取用户“主机”

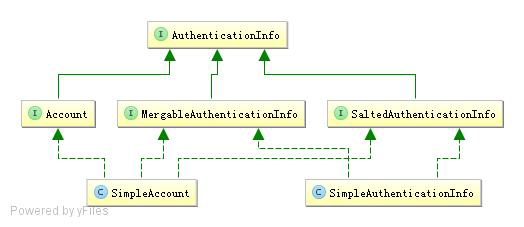
的功能。

Shiro 提供了一个直接拿来用的UsernamePasswordToken，用于实现用户名/密码Token组，

另外其实现了RememberMeAuthenticationToken和HostAuthenticationToken，可以实现记住

我及主机验证的支持。

AuthenticationInfo



AuthenticationInfo有两个作用：

1、如果Realm 是AuthenticatingRealm 子类，则提供给AuthenticatingRealm 内部使用的

CredentialsMatcher进行凭据验证；（如果没有继承它需要在自己的Realm中自己实现验证）；

1. 提供给SecurityManager来创建Subject（提供身份信息）；

MergableAuthenticationInfo 用于提供在多Realm 时合并AuthenticationInfo 的功能，主要合

并Principal、如果是其他的如credentialsSalt，会用后边的信息覆盖前边的。

比 如HashedCredentialsMatcher ， 在验证时会判断AuthenticationInfo 是否是

SaltedAuthenticationInfo子类，来获取盐信息。

Account相当于我们之前的User，SimpleAccount是其一个实现；在IniRealm、PropertiesRealm

这种静态创建帐号信息的场景中使用，这些Realm 直接继承了SimpleAccountRealm，而

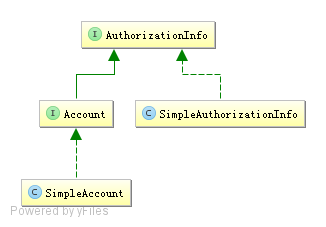
SimpleAccountRealm 提供了相关的API 来动态维护SimpleAccount；即可以通过这些API

来动态增删改查SimpleAccount；动态增删改查角色/权限信息。及如果您的帐号不是特别

多，可以使用这种方式，具体请参考SimpleAccountRealm Javadoc。

其他情况一般返回SimpleAuthenticationInfo即可。

AuthorizationInfo



AuthorizationInfo用于聚合授权信息的：

public interface AuthorizationInfo extends Serializable {

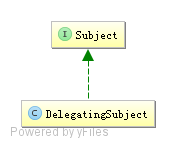
Collection<String> getRoles(); //获取角色字符串信息

Collection<String> getStringPermissions(); //获取权限字符串信息

Collection<Permission> getObjectPermissions(); //获取Permission对象信息

}

Subject



Subject是Shiro 的核心对象，基本所有身份验证、授权都是通过Subject完成。

1. **身份信息获取**

Object getPrincipal(); //Primary Principal

PrincipalCollection getPrincipals(); // PrincipalCollection

1. **身份验证**

void login(AuthenticationToken token) throws AuthenticationException;

boolean isAuthenticated();

boolean isRemembered();

通过login登录，如果登录失败将抛出相应的AuthenticationException，如果登录成功调用

isAuthenticated就会返回true，即已经通过身份验证；如果isRemembered返回true，表示是

通过记住我功能登录的而不是调用login方法登录的。isAuthenticated/isRemembered是互斥

的，即如果其中一个返回true，另一个返回false。

1. **角色授权验证**

boolean hasRole(String roleIdentifier);

boolean[] hasRoles(List<String> roleIdentifiers);

boolean hasAllRoles(Collection<String> roleIdentifiers);

void checkRole(String roleIdentifier) throws AuthorizationException;

void checkRoles(Collection<String> roleIdentifiers) throws AuthorizationException;

void checkRoles(String... roleIdentifiers) throws AuthorizationException;

hasRole\*进行角色验证，验证后返回true/false；而checkRole\*验证失败时抛出

AuthorizationException异常。

1. **权限授权验证**

boolean isPermitted(String permission);

boolean isPermitted(Permission permission);

boolean[] isPermitted(String... permissions);

boolean[] isPermitted(List<Permission> permissions);

boolean isPermittedAll(String... permissions);

boolean isPermittedAll(Collection<Permission> permissions);

void checkPermission(String permission) throws AuthorizationException;

void checkPermission(Permission permission) throws AuthorizationException;

void checkPermissions(String... permissions) throws AuthorizationException;

void checkPermissions(Collection<Permission> permissions) throws AuthorizationException;

isPermitted\*进行权限验证，验证后返回true/false；而checkPermission\*验证失败时抛出

AuthorizationException。

1. **会话**

Session getSession(); //相当于getSession(true)

Session getSession(boolean create);

类似于Web 中的会话。如果登录成功就相当于建立了会话，接着可以使用getSession 获取；

如果create=true如果没有会话将返回null，而create=true 如果没有会话会强制创建一个。

1. **退出**

void logout();

1. **RunAs**

void runAs(PrincipalCollection principals) throws NullPointerException, IllegalStateException;

boolean isRunAs();

PrincipalCollection getPreviousPrincipals();

PrincipalCollection releaseRunAs();

RunAs即实现“允许A 假设为B 身份进行访问”；通过调用subject.runAs(b)进行访问；接

着调用subject.getPrincipals将获取到B 的身份；此时调用isRunAs将返回true；而a 的身

份需要通过subject. getPreviousPrincipals获取；如果不需要RunAs了调用subject.

releaseRunAs即可。

1. **多线程**

<V> V execute(Callable<V> callable) throws ExecutionException;

void execute(Runnable runnable);

<V> Callable<V> associateWith(Callable<V> callable);

Runnable associateWith(Runnable runnable);

实现线程之间的Subject传播，因为Subject是线程绑定的；因此在多线程执行中需要传播

到相应的线程才能获取到相应的Subject。最简单的办法就是通过execute(runnable/callable

实例)直接调用；或者通过associateWith(runnable/callable实例)得到一个包装后的实例；它

们都是通过：1、把当前线程的Subject绑定过去；2、在线程执行结束后自动释放。

Subject自己不会实现相应的身份验证/授权逻辑，而是通过DelegatingSubject委托给

SecurityManager实现；及可以理解为Subject是一个面门。

对于Subject的构建一般没必要我们去创建；一般通过SecurityUtils.getSubject()获取：

public static Subject getSubject() {

Subject subject = ThreadContext.getSubject();

if (subject == null) {

subject = (new Subject.Builder()).buildSubject();

ThreadContext.bind(subject);

}

return subject;

}

即首先查看当前线程是否绑定了Subject，如果没有通过Subject.Builder构建一个然后绑定

到现场返回。

如果想自定义创建，可以通过：

new Subject.Builder().principals(身份).authenticated(true/false).buildSubject()

这种可以创建相应的Subject实例了，然后自己绑定到线程即可。在new Builder()时如果没

有传入SecurityManager，自动调用SecurityUtils.getSecurityManager获取；也可以自己传入

一个实例。

对于Subject我们一般这么使用：

**1、身份验证（login）**

**2、授权（hasRole\*/isPermitted\*或checkRole\*/checkPermission\*）**

**3、将相应的数据存储到会话（Session）**

**4、切换身份（RunAs）/多线程身份传播**

**5、退出**

# 与Web 集成

Shiro 提供了与Web 集成的支持，其通过一个ShiroFilter入口来拦截需要安全控制的URL，

然后进行相应的控制，ShiroFilter类似于如Strut2/SpringMVC这种web 框架的前端控制器，

其是安全控制的入口点，其负责读取配置（如ini 配置文件），然后判断URL 是否需要登

录/权限等工作。

**与Spring集成**

**以下添加到web.xml中**

<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：

以下添加到spring配置文件中

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">

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

<!—忽略其他，详见与Spring 集成部分-->

</bean>

最后不要忘了使用org.springframework.web.context.ContextLoaderListener 加载这个spring

配置文件即可

**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 或/admin12；

**\*：匹配零个或多个字符串**，如/admin\*将匹配/admin、/admin123，但不匹配/admin/1；

**\*\*：匹配路径中的零个或多个路径**，如/admin/\*\*将匹配/admin/a或/admin/a/b。

**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

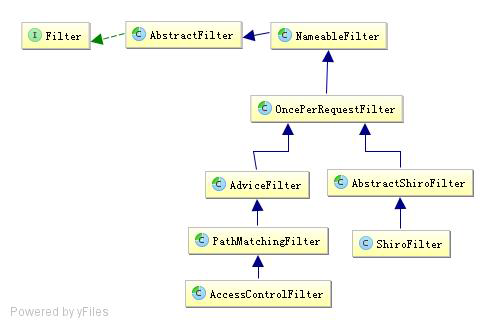
/bb/aa=filter2

/\*\*=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。

# 拦截器机制

首先下图是Shiro拦截器的基础类图：



**1、NameableFilter**

NameableFilter给Filter起个名字，如果没有设置默认就是FilterName；还记得之前的如authc

吗？当我们组装拦截器链时会根据这个名字找到相应的拦截器实例；

**2、OncePerRequestFilter**

OncePerRequestFilter用于防止多次执行Filter的；也就是说一次请求只会走一次拦截器链；

另外提供enabled属性，表示是否开启该拦截器实例，默认enabled=true表示开启，如果不

想让某个拦截器工作，可以设置为false即可。

**3、ShiroFilter**

ShiroFilter是整个Shiro的入口点，用于拦截需要安全控制的请求进行处理，这个之前已经

用过了。

**4、AdviceFilter**

AdviceFilter提供了AOP风格的支持，类似于SpringMVC 中的Interceptor：

boolean preHandle(ServletRequest request, ServletResponse response) throws Exception

void postHandle(ServletRequest request, ServletResponse response) throws Exception

void afterCompletion(ServletRequest request, ServletResponse response, Exception exception)

throws Exception;

preHandler：类似于AOP 中的前置增强；在拦截器链执行之前执行；如果返回true 则继续

拦截器链；否则中断后续的拦截器链的执行直接返回；进行预处理（如基于表单的身份验

证、授权）

postHandle：类似于AOP 中的后置返回增强；在拦截器链执行完成后执行；进行后处理（如

记录执行时间之类的）；

afterCompletion：类似于AOP 中的后置最终增强；即不管有没有异常都会执行；可以进行

清理资源（如接触Subject与线程的绑定之类的）；

**5、PathMatchingFilter**

PathMatchingFilter 提供了基于Ant 风格的请求路径匹配功能及拦截器参数解析的功能，如

“roles[admin,user]”自动根据“，”分割解析到一个路径参数配置并绑定到相应的路径：

pathsMatch：该方法用于path 与请求路径进行匹配的方法；如果匹配返回true；

onPreHandle：在preHandle 中，当pathsMatch 匹配一个路径后，会调用opPreHandler 方法

并将路径绑定参数配置传给mappedValue；然后可以在这个方法中进行一些验证（如角色

授权），如果验证失败可以返回false 中断流程；默认返回true；也就是说子类可以只实现

onPreHandle即可，无须实现preHandle。如果没有path 与请求路径匹配，默认是通过的（即

preHandle 返回true）。

**6、AccessControlFilter**

AccessControlFilter提供了访问控制的基础功能；比如是否允许访问/当访问拒绝时如何处理

等：

abstract boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object

mappedValue) throws Exception;

boolean onAccessDenied(ServletRequest request, ServletResponse response, Object

mappedValue) throws Exception;

abstract boolean onAccessDenied(ServletRequest request, ServletResponse response) throws

Exception;

isAccessAllowed：表示是否允许访问；mappedValue 就是[urls]配置中拦截器参数部分，如

果允许访问返回true，否则false；

onAccessDenied：表示当访问拒绝时是否已经处理了；如果返回true 表示需要继续处理；

如果返回false表示该拦截器实例已经处理了，将直接返回即可。

onPreHandle会自动调用这两个方法决定是否继续处理：

boolean onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue)

throws Exception {

return isAccessAllowed(request, response, mappedValue) || onAccessDenied(request,

response, mappedValue);

}

另外AccessControlFilter还提供了如下方法用于处理如登录成功后/重定向到上一个请求：

void setLoginUrl(String loginUrl) //身份验证时使用，默认/login.jsp

String getLoginUrl()

Subject getSubject(ServletRequest request, ServletResponse response) //获取Subject实例

boolean isLoginRequest(ServletRequest request, ServletResponse response)//当前请求是否是

登录请求

void saveRequestAndRedirectToLogin(ServletRequest request, ServletResponse response)

throws IOException //将当前请求保存起来并重定向到登录页面

void saveRequest(ServletRequest request) //将请求保存起来，如登录成功后再重定向回该请

求

void redirectToLogin(ServletRequest request, ServletResponse response) //重定向到登录页面

拦截器链

Shiro 对Servlet 容器的FilterChain 进行了代理，即ShiroFilter 在继续Servlet 容器的Filter

链的执行之前，通过ProxiedFilterChain 对Servlet 容器的FilterChain 进行了代理；即先走

Shiro 自己的Filter 体系，然后才会委托给Servlet 容器的FilterChain 进行Servlet 容器级别

的Filter链执行；Shiro的ProxiedFilterChain执行流程：1、先执行Shiro自己的Filter链；2、

再执行Servlet容器的Filter链（即原始的Filter）。

而 ProxiedFilterChain 是通过FilterChainResolver 根据配置文件中[urls]部分是否与请求的

URL是否匹配解析得到的。

FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain

originalChain);

即传入原始的chain得到一个代理的chain。

Shiro 内部提供了一个路径匹配的FilterChainResolver 实现：

PathMatchingFilterChainResolver，其根据[urls]中配置的url 模式（默认Ant 风格）=拦截器

链和请求的url是否匹配来解析得到配置的拦截器链的；而PathMatchingFilterChainResolver

内部通过FilterChainManager维护着拦截器链，比如DefaultFilterChainManager实现维护着

url 模式与拦截器链的关系。因此我们可以通过FilterChainManager 进行动态动态增加url

模式与拦截器链的关系。

DefaultFilterChainManager 会默认添加org.apache.shiro.web.filter.mgt.DefaultFilter 中声明的

拦截器：

public enum DefaultFilter {

anon(AnonymousFilter.class),

authc(FormAuthenticationFilter.class),

authcBasic(BasicHttpAuthenticationFilter.class),

logout(LogoutFilter.class),

noSessionCreation(NoSessionCreationFilter.class),

perms(PermissionsAuthorizationFilter.class),

port(PortFilter.class),

rest(HttpMethodPermissionFilter.class),

roles(RolesAuthorizationFilter.class),

ssl(SslFilter.class),

user(UserFilter.class);

}

自定义拦截器

通过自定义自己的拦截器可以扩展一些功能，诸如动态url-角色/权限访问控制的实现、根

据Subject 身份信息获取用户信息绑定到Request（即设置通用数据）、验证码验证、在线

用户信息的保存等等，因为其本质就是一个Filter；所以Filter能做的它就能做。

**任意角色授权拦截器**

Shiro 提供roles 拦截器，其验证用户拥有所有角色，没有提供验证用户拥有任意角色的拦

截器。

public class AnyRolesFilter extends AccessControlFilter {

private String unauthorizedUrl = "/unauthorized.jsp";

private String loginUrl = "/login.jsp";

protected boolean isAccessAllowed(ServletRequest request, ServletResponse response,

Object mappedValue) throws Exception {

String[] roles = (String[])mappedValue;

if(roles == null) {

return true;//如果没有设置角色参数，默认成功

}

for(String role : roles) {

if(getSubject(request, response).hasRole(role)) {

onAccessDenied(ServletRequest request, ServletResponse response)

throws Exception {

Subject subject = getSubject(request, response);

if (subject.getPrincipal() == null) {//表示没有登录，重定向到登录页面

saveRequest(request);

WebUtils.issueRedirect(request, response, loginUrl);

} else {

if (StringUtils.hasText(unauthorizedUrl)) {//如果有未授权页面跳转过去

WebUtils.issueRedirect(request, response, unauthorizedUrl);

} else {//否则返回401 未授权状态码

WebUtils.toHttp(response).sendError(HttpServletResponse.SC\_UNAUTHORIZED);

}

}

return false;

}

}

流程：

1、首先判断用户有没有任意角色，如果没有返回false，将到onAccessDenied 进行处理；

2、如果用户没有角色，接着判断用户有没有登录，如果没有登录先重定向到登录；

3、如果用户没有角色且设置了未授权页面（unauthorizedUrl），那么重定向到未授权页面；

否则直接返回401 未授权错误码。

**PathMatchingFilter**

PathMatchingFilter 继承了AdviceFilter，提供了url 模式过滤的功能，如果需要对指定的请

求进行处理，可以扩展PathMatchingFilter：

public class MyPathMatchingFilter extends PathMatchingFilter {

@Override

protected boolean onPreHandle(ServletRequest request, ServletResponse response, Object

mappedValue) throws Exception {

System.out.println("url matches,config is " + Arrays.toString((String[])mappedValue));

return true;

}

}

preHandle：会进行url 模式与请求url 进行匹配，如果匹配会调用onPreHandle；如果没有

配置url模式/没有url模式匹配，默认直接返回true；

onPreHandle：如果url 模式与请求url 匹配，那么会执行onPreHandle，并把该拦截器配置

的参数传入。默认什么不处理直接返回true。

[filters]

myFilter3=com.github.zhangkaitao.shiro.chapter8.web.filter.MyPathMatchingFilter

[urls]

/\*\*= myFilter3[config]

/\*\*就是注册给PathMatchingFilter 的url 模式，config 就是拦截器的配置参数，多个之间逗

号分隔，onPreHandle使用mappedValue接收参数值。

**此处，“config”就是mappedValue的值**

默认拦截器

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：

|  |  |  |
| --- | --- | --- |
| 默认拦截器名 | 拦截器类 | 说明（括号里的表示默认值） |
| ****身份验证相关的**** |  |  |
| authc | org.apache.shiro.web.filter.authc  .FormAuthenticationFilter | 基于表单的拦截器；如“/\*\*=authc”，如果没有登录会跳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：表单提交的密码参数名（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” |
| ****授权相关的**** |  |  |
| roles | org.apache.shiro.web.filter.authz  .RolesAuthorizationFilter | 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  .PermissionsAuthorizationFilter | 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，将自动将请求端口改为80并重定向到该80端口，其他路径/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；否则自动跳转会https端口（443）；其他和port拦截器一样； |
| ****其他**** |  |  |
| noSessionCreation | org.apache.shiro.web.filter.session  .NoSessionCreationFilter | 不创建会话拦截器，调用 subject.getSession(false)不会有什么问题，但是如果 subject.getSession(true)将抛出 DisabledSessionException异常； |

这些默认的拦截器会自动注册，可以直接在ini 配置文件中通过“拦截器名.属性”设置其

属性：

perms.unauthorizedUrl=/unauthorized

另外如果某个拦截器不想使用了可以直接通过如下配置直接禁用：

perms.enabled=false

# 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(Session)

所谓会话，即用户访问应用时保持的连接关系，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 的会话支持不仅可以在普通的JavaSE 应用中使用，也可以在JavaEE应用中使用，如

web应用。且使用方式是一致的。

//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）。

login("classpath:shiro.ini", "zhang", "123");

Subject subject = SecurityUtils.getSubject();

Session session = subject.getSession();

//获取当前会话的唯一标识。

session.getId();

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

session.getHost();

//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.getTimeout();

session.setTimeout(毫秒);

//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.getStartTimestamp();

session.getLastAccessTime();

//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.touch();

session.stop();

//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。

session.setAttribute("key", "123");

Assert.assertEquals("123", session.getAttribute("key"));

session.removeAttribute("key");

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 都继承了SessionsSecurityManager。

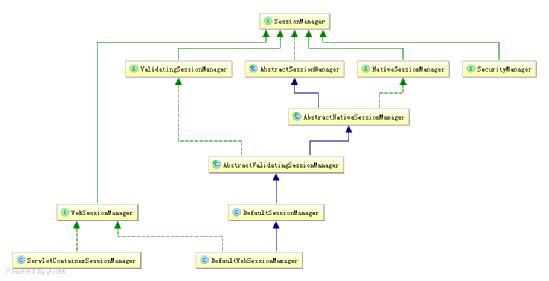
SecurityManager提供了如下接口：

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

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

另外用于Web环境的WebSessionManager又提供了如下接口：

boolean isServletContainerSessions();//是否使用Servlet容器的会话



Shiro 提供了三个默认实现：

**DefaultSessionManager**：DefaultSecurityManager使用的默认实现，用于JavaSE 环境；

**ServletContainerSessionManager**：DefaultWebSecurityManager使用的默认实现，用于Web

环境，其直接使用Servlet 容器的会话；

**DefaultWebSessionManager** ： 用于Web 环境的实现， 可以替代

ServletContainerSessionManager，自己维护着会话，直接废弃了Servlet容器的会话管理。

在 Servlet容器中，默认使用JSESSIONID Cookie 维护会话，且会话默认是跟容器绑定的；

在某些情况下可能需要使用自己的会话机制，此时我们可以使用DefaultWebSessionManager

来维护会话：

sessionIdCookie=org.apache.shiro.web.servlet.SimpleCookie

sessionManager=org.apache.shiro.web.session.mgt.DefaultWebSessionManager

sessionIdCookie.name=sid

#sessionIdCookie.domain=sishuok.com

#sessionIdCookie.path=

sessionIdCookie.maxAge=1800

sessionIdCookie.httpOnly=true

sessionManager.sessionIdCookie=$sessionIdCookie

sessionManager.sessionIdCookieEnabled=true

securityManager.sessionManager=$sessionManager

sessionIdCookie是sessionManager创建会话Cookie的模板：

sessionIdCookie.name：设置Cookie 名字，默认为JSESSIONID；

sessionIdCookie.domain：设置Cookie的域名，默认空，即当前访问的域名；

sessionIdCookie.path：设置Cookie 的路径，默认空，即存储在域名根下；

sessionIdCookie.maxAge：设置Cookie 的过期时间，秒为单位，默认-1 表示关闭浏览器时

过期Cookie；

sessionIdCookie.httpOnly：如果设置为true，则客户端不会暴露给客户端脚本代码，使用

HttpOnly cookie有助于减少某些类型的跨站点脚本攻击；此特性需要实现了Servlet 2.5 MR6

及以上版本的规范的Servlet容器支持；

sessionManager.sessionIdCookieEnabled：是否启用/禁用Session Id Cookie，默认是启用的；

如果禁用后将不会设置Session Id Cookie，即默认使用了Servlet容器的JSESSIONID，且通

过URL重写（URL中的“;JSESSIONID=id”部分）保存Session Id。

另外我们可以如“sessionManager. sessionIdCookie.name=sid”这种方式操作Cookie 模板。

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 {

@Override

public void onStart(Session session) {//会话创建时触发

System.out.println("会话创建：" + session.getId());

}

@Override

public void onExpiration(Session session) {//会话过期时触发

System.out.println("会话过期：" + session.getId());

}

@Override

public void onStop(Session session) {//退出/会话过期时触发

System.out.println("会话停止：" + session.getId());

}

}

如果只想监听某一个事件，可以继承SessionListenerAdapter实现：

public class MySessionListener2 extends SessionListenerAdapter {

@Override

public void onStart(Session session) {

System.out.println("会话创建：" + session.getId());

}

}

把会话监听器配置到相应的SessionManager下即可

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 后会调用该方法；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；

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;

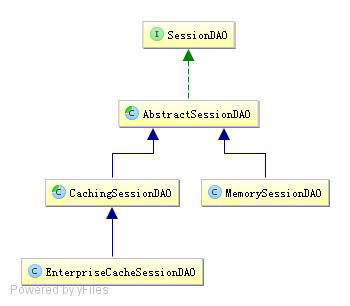
//删除会话；当会话过期/会话停止（如用户退出时）会调用

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

Collection<Session> getActiveSessions();

Shiro 内嵌了如下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。

会话验证

Shiro 提供了会话验证调度器，用于定期的验证会话是否已过期，如果过期将停止会话；出

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是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。

可以通过如下ini配置开启会话验证：

sessionValidationScheduler=org.apache.shiro.session.mgt.ExecutorServiceSessionValidationSc

heduler

sessionValidationScheduler.interval = 3600000

sessionValidationScheduler.sessionManager=$sessionManager

sessionManager.globalSessionTimeout=1800000

sessionManager.sessionValidationSchedulerEnabled=true

sessionManager.sessionValidationScheduler=$sessionValidationScheduler

sessionValidationScheduler ： 会话验证调度器， sessionManager 默认就是使用

ExecutorServiceSessionValidationScheduler，其使用JDK的ScheduledExecutorService进行定

期调度并验证会话是否过期；

sessionValidationScheduler.interval：设置调度时间间隔，单位毫秒，默认就是1 小时；

sessionValidationScheduler.sessionManager：设置会话验证调度器进行会话验证时的会话管

理器；

sessionManager.globalSessionTimeout：设置全局会话超时时间，默认30 分钟，即如果30

分钟内没有访问会话将过期；

sessionManager.sessionValidationSchedulerEnabled：是否开启会话验证器，默认是开启的；

sessionManager.sessionValidationScheduler： 设置会话验证调度器，默认就是使用

ExecutorServiceSessionValidationScheduler。

SessionFactory

SessionFactory 是创建会话的工厂，根据相应的Subject 上下文信息来创建会话；默认提供

了SimpleSessionFactory用来创建SimpleSession会话。

首先自定义一个Session：

public class OnlineSession extends SimpleSession {

public static enum OnlineStatus {

on\_line("在线"), hidden("隐身"), force\_logout("强制退出");

private final String info;

private OnlineStatus(String info) {

this.info = info;

}

public String getInfo() {

return info;

}

}

private String userAgent; //用户浏览器类型

private OnlineStatus status = OnlineStatus.on\_line; //在线状态

private String systemHost; //用户登录时系统IP

//省略其他

}

OnlineSession 用于保存当前登录用户的在线状态，支持如离线等状态的控制。

接着自定义SessionFactory：

public class OnlineSessionFactory implements SessionFactory {

@Override

public Session createSession(SessionContext initData) {

OnlineSession session = new OnlineSession();

if (initData != null && initData instanceof WebSessionContext) {

WebSessionContext sessionContext = (WebSessionContext) initData;

HttpServletRequest request = (HttpServletRequest)

sessionContext.getServletRequest();

if (request != null) {

session.setHost(IpUtils.getIpAddr(request));

session.setUserAgent(request.getHeader("User-Agent"));

session.setSystemHost(request.getLocalAddr() + ":" +

request.getLocalPort());

}

}

return session;

}

}

根据会话上下文创建相应的OnlineSession。

最后在shiro-web.ini配置文件中配置：

sessionFactory=org.apache.shiro.session.mgt.OnlineSessionFactory

sessionManager.sessionFactory=$sessionFactory