# Shrio简介

# Shrio主要API

## 认证

### Authenticator

|  |
| --- |
| **public** **interface** Authenticator {  **public** AuthenticationInfo authenticate (AuthenticationToken authenticationToken)  **throws** AuthenticationException;  } |

### Authenticator 真正实现体系结构

|  |
| --- |
|  |

注：像SecurityManager接口继承自Authenticator接口，但SecurityManager的实现类完成认证还是通过以上实现来完成的，故这里言“真正实现”。

#### AbstractAuthenticator

|  |
| --- |
| **public** **abstract** **class** AbstractAuthenticator **implements** Authenticator, LogoutAware {  **public** **final** AuthenticationInfo authenticate(AuthenticationToken token) **throws** AuthenticationException {  **if** (token == **null**) {  **throw** **new** IllegalArgumentException("Method argumet (authentication token) cannot be null.");  }  ***log***.trace("Authentication attempt received for token [{}]", token);  AuthenticationInfo info;  **try** {  **// 重要**  info = doAuthenticate(token);  **if** (info == **null**) {  String msg = "No account information found for authentication token [" + token + "] by this " +  "Authenticator instance. Please check that it is configured correctly.";  **throw** **new** AuthenticationException(msg);  }  } **catch** (Throwable t) {  AuthenticationException ae = **null**;  **if** (t **instanceof** AuthenticationException) {  ae = (AuthenticationException) t;  }  **if** (ae == **null**) {  //Exception thrown was not an expected AuthenticationException. Therefore it is probably a  little more  //severe or unexpected. So, wrap in an AuthenticationException, log to warn, and propagate:  String msg = "Authentication failed for token submission [" + token + "]. Possible unexpected "  +"error? (Typical or expected login exceptions should extend from AuthenticationException).";  ae = **new** AuthenticationException(msg, t);  }  **try** {  notifyFailure(token, ae);  } **catch** (Throwable t2) {  **if** (***log***.isWarnEnabled()) {  String msg = "Unable to send notification for failed authentication attempt - listener  error?. " +  "Please check your AuthenticationListener implementation(s). Logging sending  exception " +  "and propagating original AuthenticationException instead...";  ***log***.warn(msg, t2);  }  }  **throw** ae;  }  ***log***.debug("Authentication successful for token [{}]. Returned account [{}]", token, info);  notifySuccess(token, info);  **return** info;  }  **//让子类来实现,shiro 目前仅有一个真正实现🡪 ModularRealmAuthenticator**  **protected** **abstract** AuthenticationInfo doAuthenticate(AuthenticationToken token)  **throws** AuthenticationException;  …  } |

#### ModularRealmAuthenticator

|  |
| --- |
| **public** **class** ModularRealmAuthenticator **extends** AbstractAuthenticator {  **private** Collection<Realm> realms;  **private** AuthenticationStrategy authenticationStrategy;  **public** ModularRealmAuthenticator() {  **this**.authenticationStrategy = **new** AtLeastOneSuccessfulStrategy();  }  **protected** AuthenticationInfo doAuthenticate(AuthenticationToken authenticationToken) **throws** AuthenticationException {  assertRealmsConfigured();  Collection<Realm> realms = getRealms();  **if** (realms.size() == 1) {  **return** doSingleRealmAuthentication(realms.iterator().next(), authenticationToken);  } **else** {  **return** doMultiRealmAuthentication(realms, authenticationToken);  }  }  **protected** AuthenticationInfo doSingleRealmAuthentication(Realm realm, AuthenticationToken token) {  **if** (!realm.supports(token)) {  String msg = "Realm [" + realm + "] does not support authentication token [" +  token + "]. Please ensure that the appropriate Realm implementation is " +  "configured correctly or that the realm accepts AuthenticationTokens of this type.";  **throw** **new** UnsupportedTokenException(msg);  }  AuthenticationInfo info = realm.getAuthenticationInfo(token);  **if** (info == **null**) {  String msg = "Realm [" + realm + "] was unable to find account data for the " +  "submitted AuthenticationToken [" + token + "].";  **throw** **new** UnknownAccountException(msg);  }  **return** info;  }  **protected** AuthenticationInfo doMultiRealmAuthentication(Collection<Realm> realms, AuthenticationToken token) {  AuthenticationStrategy strategy = getAuthenticationStrategy();  AuthenticationInfo aggregate = strategy.beforeAllAttempts(realms, token);  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Iterating through {} realms for PAM authentication", realms.size());  }  **for** (Realm realm : realms) {  aggregate = strategy.beforeAttempt(realm, token, aggregate);  **if** (realm.supports(token)) {  ***log***.trace("Attempting to authenticate token [{}] using realm [{}]", token, realm);  AuthenticationInfo info = **null**;  Throwable t = **null**;  **try** {  info = realm.getAuthenticationInfo(token);  } **catch** (Throwable throwable) {  t = throwable;  **if** (***log***.isDebugEnabled()) {  String msg = "Realm [" + realm + "] threw an exception during a multi-realm authentication attempt:";  ***log***.debug(msg, t);  }  }  aggregate = strategy.afterAttempt(realm, token, info, aggregate, t);  } **else** {  ***log***.debug("Realm [{}] does not support token {}. Skipping realm.", realm, token);  }  }  aggregate = strategy.afterAllAttempts(token, aggregate);  **return** aggregate;  } |

### 总结

**从ModularRealmAuthenticator上可看出认证最终将会移交给realm来完成。**

## 授权

### Authorizer

|  |
| --- |
| **public** **interface** Authorizer {  **boolean** isPermitted(PrincipalCollection principals, String permission);  **boolean** isPermitted(PrincipalCollection subjectPrincipal, Permission permission);  **boolean**[] isPermitted(PrincipalCollection subjectPrincipal, String... permissions);  **boolean**[] isPermitted(PrincipalCollection subjectPrincipal, List<Permission> permissions);  **boolean** isPermittedAll(PrincipalCollection subjectPrincipal, String... permissions);  **boolean** isPermittedAll(PrincipalCollection subjectPrincipal, Collection<Permission> permissions);  **void** checkPermission(PrincipalCollection subjectPrincipal, String permission) **throws** AuthorizationException;  **void** checkPermission(PrincipalCollection subjectPrincipal, Permission permission) **throws** AuthorizationException;  **void** checkPermissions(PrincipalCollection subjectPrincipal, String... permissions) **throws** AuthorizationException;  **void** checkPermissions(PrincipalCollection subjectPrincipal, Collection<Permission> permissions) **throws** AuthorizationException;  **boolean** hasRole(PrincipalCollection subjectPrincipal, String roleIdentifier);  **boolean**[] hasRoles(PrincipalCollection subjectPrincipal, List<String> roleIdentifiers);  **boolean** hasAllRoles(PrincipalCollection subjectPrincipal, Collection<String> roleIdentifiers);  **void** checkRole(PrincipalCollection subjectPrincipal, String roleIdentifier) **throws** AuthorizationException;  **void** checkRoles(PrincipalCollection subjectPrincipal, Collection<String> roleIdentifiers) **throws** AuthorizationException;  **void** checkRoles(PrincipalCollection subjectPrincipal, String... roleIdentifiers) **throws** AuthorizationException; |

Authorizer

|  |
| --- |
|  |

### Authorizer真正实现体系

|  |
| --- |
|  |

#### ModularRealmAuthorizer

|  |
| --- |
| **public** **class** ModularRealmAuthorizer **implements** Authorizer, PermissionResolverAware, RolePermissionResolverAware {  **protected** Collection<Realm> realms;  **protected** PermissionResolver permissionResolver;  **protected** RolePermissionResolver rolePermissionResolver;  **//由于Authorizer接口方法很多，这里只展示其中之一**  **//由代码可看出isPermitted的真正实现是由realm来完成。**  **public** **boolean** isPermitted(PrincipalCollection principals, String permission) {  assertRealmsConfigured();  **for** (Realm realm : getRealms()) {  **if** (!(realm **instanceof** Authorizer)) **continue**;  **if** (((Authorizer) realm).isPermitted(principals, permission)) {  **return** **true**;  }  }  **return** **false**;  }  …  } |

### 总结

**从ModularRealmAuthorizer可看出授权最终将会移交给realm来完成。**

## SecurityManager

### SecurityManager

|  |
| --- |
| **public** **interface** SecurityManager **extends** Authenticator, Authorizer, SessionManager {  Subject login(Subject subject, AuthenticationToken authenticationToken) **throws** AuthenticationException;  **void** logout(Subject subject);  Subject createSubject(SubjectContext context);  } |

### SecurityManager概览

|  |
| --- |
|  |

#### CachingSecurityManager

|  |
| --- |
| **public** **abstract** **class** CachingSecurityManager **implements** SecurityManager, Destroyable, CacheManagerAware {  **private** CacheManager cacheManager;  **public** **void** setCacheManager(CacheManager cacheManager) {  **this**.cacheManager = cacheManager;  afterCacheManagerSet();  }  **protected** **void** afterCacheManagerSet() {  }  …  } |

#### RealmSecurityManager

|  |
| --- |
| **public** **abstract** **class** RealmSecurityManager **extends** CachingSecurityManager {  **private** Collection<Realm> realms;  **public** RealmSecurityManager() {  **super**();  }  **public** **void** setRealm(Realm realm) {  **if** (realm == **null**) {  **throw** **new** IllegalArgumentException("Realm argument cannot be null");  }  Collection<Realm> realms = **new** ArrayList<Realm>(1);  realms.add(realm);  setRealms(realms);  }  **public** **void** setRealms(Collection<Realm> realms) {  **if** (realms == **null**) {  **throw** **new** IllegalArgumentException("Realms collection argument cannot be null.");  }  **if** (realms.isEmpty()) {  **throw** **new** IllegalArgumentException("Realms collection argument cannot be empty.");  }  **this**.realms = realms;  afterRealmsSet();  }  **protected** **void** afterRealmsSet() {  applyCacheManagerToRealms();  }  **protected** **void** applyCacheManagerToRealms() {  CacheManager cacheManager = getCacheManager();  Collection<Realm> realms = getRealms();  **if** (cacheManager != **null** && realms != **null** && !realms.isEmpty()) {  **for** (Realm realm : realms) {  **if** (realm **instanceof** CacheManagerAware) {  ((CacheManagerAware) realm).setCacheManager(cacheManager);  }  }  }  }  …  } |

#### AuthenticatingSecurityManager

|  |
| --- |
| **public** **abstract** **class** AuthenticatingSecurityManager **extends** RealmSecurityManager {  **private** Authenticator authenticator;  **public** AuthenticatingSecurityManager() {  **super**();  **//默认authenticator实现…**  **//ModularRealmAuthenticator本质是Realm接口实现的集合**  **this**.authenticator = **new** ModularRealmAuthenticator();  }  **public** **void** setAuthenticator(Authenticator authenticator) **throws** IllegalArgumentException {  **if** (authenticator == **null**) {**//自定义authenticator注入…**  String msg = "Authenticator argument cannot be null.";  **throw** **new** IllegalArgumentException(msg);  }  **this**.authenticator = authenticator;  }  **protected** **void** afterRealmsSet() {  **super**.afterRealmsSet();  **//如果是默认authenticator实现(即ModularRealmAuthenticator)，则把realm接口实现注入给它。**  **if** (**this**.authenticator **instanceof** ModularRealmAuthenticator) {  ((ModularRealmAuthenticator) **this**.authenticator).setRealms(getRealms());  }  }  **//重要：**  **//SecurityManger接口继承了Authenticator接口，此方法即为Authenticator#authenticate的实现**  **//很明显这个实现真正完成者是由Authenticator接口的实现authenticator来完成的，SecurityManger**  **//只是个摆设而已。**  **//另Authenticator默认实现是ModularRealmAuthenticator；**  **public** AuthenticationInfo authenticate(AuthenticationToken token) **throws** AuthenticationException {  **return** **this**.authenticator.authenticate(token);  }  …  } |

#### AuthorizingSecurityManager

|  |
| --- |
| **public** **abstract** **class** AuthorizingSecurityManager **extends** AuthenticatingSecurityManager {  **private** Authorizer authorizer;  **public** AuthorizingSecurityManager() {  **super**();  **//默认authorizer实现…**  **// ModularRealmAuthorizer本质是Realm接口实现的集合**  **this**.authorizer = **new** ModularRealmAuthorizer();  }  **public** Authorizer getAuthorizer() {  **return** authorizer;  }  **public** **void** setAuthorizer(Authorizer authorizer) {  **if** (authorizer == **null**) {**//自定义authorizer注入…**  String msg = "Authorizer argument cannot be null.";  **throw** **new** IllegalArgumentException(msg);  }  **this**.authorizer = authorizer;  }  **protected** **void** afterRealmsSet() {  **super**.afterRealmsSet();  **//如果是默认authorizer实现(即ModularRealmAuthorizer)，则把realm接口实现注入给它。**  **if** (**this**.authorizer **instanceof** ModularRealmAuthorizer) {  ((ModularRealmAuthorizer) **this**.authorizer).setRealms(getRealms());  }  }  **public** **void** destroy() {  LifecycleUtils.*destroy*(getAuthorizer());  **this**.authorizer = **null**;  **super**.destroy();  }    **//重要：**  **//SecurityManger接口继承了**Authorizer**接口，以下方法即为**Authorizer**接口的实现**  **//很明显这个实现真正完成者是由**Authorizer**接口的实现**authorizer**来完成的，SecurityManger**  **//只是个摆设而已。另Authenticator默认实现是ModularRealmAuthenticator；**  **public** **boolean** isPermitted(PrincipalCollection principals, String permissionString) {  **return** **this**.authorizer.isPermitted(principals, permissionString);  }  **public** **boolean** isPermitted(PrincipalCollection principals, Permission permission) {  **return** **this**.authorizer.isPermitted(principals, permission);  }  **public** **boolean**[] isPermitted(PrincipalCollection principals, String... permissions) {  **return** **this**.authorizer.isPermitted(principals, permissions);  }  **public** **boolean**[] isPermitted(PrincipalCollection principals, List<Permission> permissions) {  **return** **this**.authorizer.isPermitted(principals, permissions);  }  **public** **boolean** isPermittedAll(PrincipalCollection principals, String... permissions) {  **return** **this**.authorizer.isPermittedAll(principals, permissions);  }  **public** **boolean** isPermittedAll(PrincipalCollection principals, Collection<Permission> permissions) {  **return** **this**.authorizer.isPermittedAll(principals, permissions);  }  **public** **void** checkPermission(PrincipalCollection principals, String permission) **throws** AuthorizationException {  **this**.authorizer.checkPermission(principals, permission);  }  **public** **void** checkPermission(PrincipalCollection principals, Permission permission) **throws** AuthorizationException {  **this**.authorizer.checkPermission(principals, permission);  }  **public** **void** checkPermissions(PrincipalCollection principals, String... permissions) **throws** AuthorizationException {  **this**.authorizer.checkPermissions(principals, permissions);  }  **public** **void** checkPermissions(PrincipalCollection principals, Collection<Permission> permissions) **throws** AuthorizationException {  **this**.authorizer.checkPermissions(principals, permissions);  }  **public** **boolean** hasRole(PrincipalCollection principals, String roleIdentifier) {  **return** **this**.authorizer.hasRole(principals, roleIdentifier);  }  **public** **boolean**[] hasRoles(PrincipalCollection principals, List<String> roleIdentifiers) {  **return** **this**.authorizer.hasRoles(principals, roleIdentifiers);  }  **public** **boolean** hasAllRoles(PrincipalCollection principals, Collection<String> roleIdentifiers) {  **return** **this**.authorizer.hasAllRoles(principals, roleIdentifiers);  }  **public** **void** checkRole(PrincipalCollection principals, String role) **throws** AuthorizationException {  **this**.authorizer.checkRole(principals, role);  }  **public** **void** checkRoles(PrincipalCollection principals, Collection<String> roles) **throws** AuthorizationException {  **this**.authorizer.checkRoles(principals, roles);  }    **public** **void** checkRoles(PrincipalCollection principals, String... roles) **throws** AuthorizationException {  **this**.authorizer.checkRoles(principals, roles);  } |

#### SessionsSecurityManager

|  |
| --- |
| **public** **abstract** **class** SessionsSecurityManager **extends** AuthorizingSecurityManager {    **private** SessionManager sessionManager;    **public** SessionsSecurityManager() {  **super**();  **//默认sessionManager**  **this**.sessionManager = **new** DefaultSessionManager();  applyCacheManagerToSessionManager();  }    **public** **void** setSessionManager(SessionManager sessionManager) {  **this**.sessionManager = sessionManager;  afterSessionManagerSet();  }  **protected** **void** afterSessionManagerSet() {  applyCacheManagerToSessionManager();  }  **protected** **void** applyCacheManagerToSessionManager() {  **if** (**this**.sessionManager **instanceof** CacheManagerAware) {  ((CacheManagerAware) **this**.sessionManager).setCacheManager(getCacheManager());  }  }  **public** Session start(SessionContext context) **throws** AuthorizationException {  **return** **this**.sessionManager.start(context);  }  **public** Session getSession(SessionKey key) **throws** SessionException {  **return** **this**.sessionManager.getSession(key);  }  **public** **void** destroy() {  LifecycleUtils.*destroy*(getSessionManager());  **this**.sessionManager = **null**;  **super**.destroy();  }  } |

#### DefaultSecurityManager

|  |
| --- |
| **public** **class** DefaultSecurityManager **extends** SessionsSecurityManager {  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(DefaultSecurityManager.**class**);  **protected** RememberMeManager rememberMeManager;  **protected** SubjectDAO subjectDAO;  **protected** SubjectFactory subjectFactory;  **public** DefaultSecurityManager() {  **super**();  **this**.subjectFactory = **new** DefaultSubjectFactory();  **this**.subjectDAO = **new** DefaultSubjectDAO();  }  **//SecurityManager#createSubject实现**  **public** Subject createSubject(SubjectContext subjectContext) {  //create a copy so we don't modify the argument's backing map:  SubjectContext context = copy(subjectContext);  //ensure that the context has a SecurityManager instance, and if not, add one:  context = ensureSecurityManager(context);  //Resolve an associated Session (usually based on a referenced session ID), and place it in the context before  //sending to the SubjectFactory. The SubjectFactory should not need to know how to acquire sessions as the  //process is often environment specific - better to shield the SF from these details:  context = resolveSession(context);  //Similarly, the SubjectFactory should not require any concept of RememberMe - translate that here first  //if possible before handing off to the SubjectFactory:  context = resolvePrincipals(context);  Subject subject = doCreateSubject(context);  //save this subject for future reference if necessary:  //(this is needed here in case rememberMe principals were resolved and they need to be stored in the  //session, so we don't constantly rehydrate the rememberMe PrincipalCollection on every operation).  //Added in 1.2:  save(subject);  **return** subject;  }  **protected** Subject doCreateSubject(SubjectContext context) {  **return** getSubjectFactory().createSubject(context);  }  **protected** **void** save(Subject subject) {  **this**.subjectDAO.save(subject);  } |

#### DefaultWebSecurityManager

|  |
| --- |
| **public** **class** DefaultWebSecurityManager **extends** DefaultSecurityManager **implements** WebSecurityManager {  @Deprecated  **public** **static** **final** String ~~HTTP\_SESSION\_MODE~~ = "http";  @Deprecated  **public** **static** **final** String ~~NATIVE\_SESSION\_MODE~~ = "native";  @Deprecated  **private** String ~~sessionMode~~;  **public** DefaultWebSecurityManager() {  **super**();  ((DefaultSubjectDAO) **this**.subjectDAO).setSessionStorageEvaluator(**new** DefaultWebSessionStorageEvaluator());  **this**.~~sessionMode~~ = ~~HTTP\_SESSION\_MODE~~;  setSubjectFactory(**new** DefaultWebSubjectFactory());  setRememberMeManager(**new** CookieRememberMeManager());  setSessionManager(**new** ServletContainerSessionManager());  }  @SuppressWarnings({"UnusedDeclaration"})  **public** DefaultWebSecurityManager(Realm singleRealm) {  **this**();  setRealm(singleRealm);  }  @SuppressWarnings({"UnusedDeclaration"})  **public** DefaultWebSecurityManager(Collection<Realm> realms) {  **this**();  setRealms(realms);  }  @Override  **protected** SubjectContext createSubjectContext() {  **return** **new** DefaultWebSubjectContext();  }  @Override  //since 1.2.1 for fixing SHIRO-350  **public** **void** setSubjectDAO(SubjectDAO subjectDAO) {  **super**.setSubjectDAO(subjectDAO);  applySessionManagerToSessionStorageEvaluatorIfPossible();  }  //since 1.2.1 for fixing SHIRO-350  @Override  **protected** **void** afterSessionManagerSet() {  **super**.afterSessionManagerSet();  applySessionManagerToSessionStorageEvaluatorIfPossible();  }  //since 1.2.1 for fixing SHIRO-350:  **private** **void** applySessionManagerToSessionStorageEvaluatorIfPossible() {  SubjectDAO subjectDAO = getSubjectDAO();  **if** (subjectDAO **instanceof** DefaultSubjectDAO) {  SessionStorageEvaluator evaluator = ((DefaultSubjectDAO)subjectDAO).getSessionStorageEvaluator();  **if** (evaluator **instanceof** DefaultWebSessionStorageEvaluator) {  ((DefaultWebSessionStorageEvaluator)evaluator).setSessionManager(getSessionManager());  }  }  }  @Override  **protected** SubjectContext copy(SubjectContext subjectContext) {  **if** (subjectContext **instanceof** WebSubjectContext) {  **return** **new** DefaultWebSubjectContext((WebSubjectContext) subjectContext);  }  **return** **super**.copy(subjectContext);  }  **protected** SessionManager createSessionManager(String sessionMode) {  **if** (sessionMode == **null** || !sessionMode.equalsIgnoreCase(~~NATIVE\_SESSION\_MODE~~)) {  ***log***.info("{} mode - enabling ServletContainerSessionManager (HTTP-only Sessions)", ~~HTTP\_SESSION\_MODE~~);  **return** **new** ServletContainerSessionManager();  } **else** {  ***log***.info("{} mode - enabling DefaultWebSessionManager (non-HTTP and HTTP Sessions)", ~~NATIVE\_SESSION\_MODE~~);  **return** **new** DefaultWebSessionManager();  }  }  @Override  **protected** SessionContext createSessionContext(SubjectContext subjectContext) {  SessionContext sessionContext = **super**.createSessionContext(subjectContext);  **if** (subjectContext **instanceof** WebSubjectContext) {  WebSubjectContext wsc = (WebSubjectContext) subjectContext;  ServletRequest request = wsc.resolveServletRequest();  ServletResponse response = wsc.resolveServletResponse();  DefaultWebSessionContext webSessionContext = **new** DefaultWebSessionContext(sessionContext);  **if** (request != **null**) {  webSessionContext.setServletRequest(request);  }  **if** (response != **null**) {  webSessionContext.setServletResponse(response);  }  sessionContext = webSessionContext;  }  **return** sessionContext;  }  @Override  **protected** SessionKey getSessionKey(SubjectContext context) {  **if** (WebUtils.*isWeb*(context)) {  Serializable sessionId = context.getSessionId();  ServletRequest request = WebUtils.*getRequest*(context);  ServletResponse response = WebUtils.*getResponse*(context);  **return** **new** WebSessionKey(sessionId, request, response);  } **else** {  **return** **super**.getSessionKey(context);  }  }  @Override  **protected** **void** beforeLogout(Subject subject) {  **super**.beforeLogout(subject);  removeRequestIdentity(subject);  }  **protected** **void** removeRequestIdentity(Subject subject) {  **if** (subject **instanceof** WebSubject) {  WebSubject webSubject = (WebSubject) subject;  ServletRequest request = webSubject.getServletRequest();  **if** (request != **null**) {  request.setAttribute(ShiroHttpServletRequest.***IDENTITY\_REMOVED\_KEY***, Boolean.***TRUE***);  }  }  }  } |

## realm

### Realm

|  |
| --- |
| **public** **interface** Realm {  String getName();  **boolean** supports(AuthenticationToken token);  AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException;  } |

### Realm的继承实现体系

|  |
| --- |
|  |

#### CachingRealm

|  |
| --- |
| **public** **abstract** **class** CachingRealm **implements** Realm, Nameable, CacheManagerAware, LogoutAware {  **private** String name;  **private** **boolean** cachingEnabled;  **private** CacheManager cacheManager;  …  } |

#### AuthenticatingRealm

|  |
| --- |
| **public** **abstract** **class** AuthenticatingRealm **extends** CachingRealm **implements** Initializable {    **private** CredentialsMatcher credentialsMatcher;  **private** Cache<Object, AuthenticationInfo> authenticationCache;  **private** **boolean** authenticationCachingEnabled;  **private** String authenticationCacheName;  **private** Class<? **extends** AuthenticationToken> authenticationTokenClass;  **public** AuthenticatingRealm() {  **//默认密码比对为SimpleCredentialsMatcher**  **this**(**null**, **new** SimpleCredentialsMatcher());  }  **public** AuthenticatingRealm(CacheManager cacheManager) {  **this**(cacheManager, **new** SimpleCredentialsMatcher());  }  **public** AuthenticatingRealm(CacheManager cacheManager, CredentialsMatcher matcher) {  authenticationTokenClass = UsernamePasswordToken.**class**;  **this**.authenticationCachingEnabled = **false**;  **int** instanceNumber = ***INSTANCE\_COUNT***.getAndIncrement();  **this**.authenticationCacheName = getClass().getName() + ***DEFAULT\_AUTHORIZATION\_CACHE\_SUFFIX***;  **if** (instanceNumber > 0) {  **this**.authenticationCacheName = **this**.authenticationCacheName + "." + instanceNumber;  }  **if** (cacheManager != **null**) {  setCacheManager(cacheManager);  }  **if** (matcher != **null**) {  setCredentialsMatcher(matcher);  }  }  **//重要**  **public** **final** AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {  AuthenticationInfo info = getCachedAuthenticationInfo(token);  **if** (info == **null**) {  //otherwise not cached, perform the lookup:  **//重要**  info = doGetAuthenticationInfo(token);  ***log***.debug("Looked up AuthenticationInfo [{}] from doGetAuthenticationInfo", info);  **if** (token != **null** && info != **null**) {  cacheAuthenticationInfoIfPossible(token, info);  }  } **else** {  ***log***.debug("Using cached authentication info [{}] to perform credentials matching.", info);  }  **if** (info != **null**) {  **//进行密码校验**  assertCredentialsMatch(token, info);  } **else** {  ***log***.debug("No AuthenticationInfo found for submitted AuthenticationToken [{}]. Returning null.", token);  }  **return** info;  }  \*/  **//重要 -🡪我们通常要实现它**  **protected** **abstract** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException;  **//密码校验**  **protected** **void** assertCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) **throws** AuthenticationException {  CredentialsMatcher cm = getCredentialsMatcher();  **if** (cm != **null**) {  **if** (!cm.doCredentialsMatch(token, info)) {  //not successful - throw an exception to indicate this:  String msg = "Submitted credentials for token [" + token + "] did not match the expected credentials.";  **throw** **new** IncorrectCredentialsException(msg);  }  } **else** {  **throw** **new** AuthenticationException("A CredentialsMatcher must be configured in order to verify " +  "credentials during authentication. If you do not wish for credentials to be examined, you " +  "can configure an " + AllowAllCredentialsMatcher.**class**.getName() + " instance.");  }  }  } |

#### AuthorizingRealm

|  |
| --- |
| **public** **abstract** **class** AuthorizingRealm **extends** AuthenticatingRealm  **implements** Authorizer, Initializable, PermissionResolverAware, RolePermissionResolverAware {  **private** **boolean** authorizationCachingEnabled;  **private** Cache<Object, AuthorizationInfo> authorizationCache;  **private** String authorizationCacheName;  **private** PermissionResolver permissionResolver;  **private** RolePermissionResolver permissionRoleResolver;  **public** AuthorizingRealm(CacheManager cacheManager, CredentialsMatcher matcher) {  **super**();  **if** (cacheManager != **null**) setCacheManager(cacheManager);  **if** (matcher != **null**) setCredentialsMatcher(matcher);  **this**.authorizationCachingEnabled = **true**;  **this**.permissionResolver = **new** WildcardPermissionResolver();  **int** instanceNumber = ***INSTANCE\_COUNT***.getAndIncrement();  **this**.authorizationCacheName = getClass().getName() + ***DEFAULT\_AUTHORIZATION\_CACHE\_SUFFIX***;  **if** (instanceNumber > 0) {  **this**.authorizationCacheName = **this**.authorizationCacheName + "." + instanceNumber;  }  }  **protected** AuthorizationInfo getAuthorizationInfo(PrincipalCollection principals) {  **if** (principals == **null**) {  **return** **null**;  }  AuthorizationInfo info = **null**;  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Retrieving AuthorizationInfo for principals [" + principals + "]");  }  Cache<Object, AuthorizationInfo> cache = getAvailableAuthorizationCache();  **if** (cache != **null**) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Attempting to retrieve the AuthorizationInfo from cache.");  }  Object key = getAuthorizationCacheKey(principals);  info = cache.get(key);  **if** (***log***.isTraceEnabled()) {  **if** (info == **null**) {  ***log***.trace("No AuthorizationInfo found in cache for principals [" + principals + "]");  } **else** {  ***log***.trace("AuthorizationInfo found in cache for principals [" + principals + "]");  }  }  }  **if** (info == **null**) {  // Call template method if the info was not found in a cache  info = doGetAuthorizationInfo(principals);  // If the info is not null and the cache has been created, then cache the authorization info.  **if** (info != **null** && cache != **null**) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Caching authorization info for principals: [" + principals + "].");  }  Object key = getAuthorizationCacheKey(principals);  cache.put(key, info);  }  }  **return** info;  }  **//重要 -🡪我们通常要实现它**  **protected** **abstract** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals);  //这里仅分析其一，  **public** **boolean** isPermitted(PrincipalCollection principals, String permission) {  Permission p = getPermissionResolver().resolvePermission(permission);  **return** isPermitted(principals, p);  }  **public** **boolean** isPermitted(PrincipalCollection principals, Permission permission) {  **//重要**  AuthorizationInfo info = getAuthorizationInfo(principals);//  **return** isPermitted(permission, info);  }  **private** **boolean** isPermitted(Permission permission, AuthorizationInfo info) {  **//获取主体（用户）所有权限**  Collection<Permission> perms = getPermissions(info);  **if** (perms != **null** && !perms.isEmpty()) {  **for** (Permission perm : perms) {  **if** (perm.implies(permission)) {  **return** **true**;  }  }  }  **return** **false**;  }…  **private** Collection<Permission> getPermissions(AuthorizationInfo info) {  Set<Permission> permissions = **new** HashSet<Permission>();  **if** (info != **null**) {  Collection<Permission> perms = info.getObjectPermissions();  **if** (!CollectionUtils.*isEmpty*(perms)) {  permissions.addAll(perms);  }  perms = resolvePermissions(info.getStringPermissions());  **if** (!CollectionUtils.*isEmpty*(perms)) {  permissions.addAll(perms);  }  perms = resolveRolePermissions(info.getRoles());  **if** (!CollectionUtils.*isEmpty*(perms)) {  permissions.addAll(perms);  }  }  **if** (permissions.isEmpty()) {  **return** Collections.*emptySet*();  } **else** {  **return** Collections.*unmodifiableSet*(permissions);  }  } |

## 密码校验

### CredentialsMatcher

|  |
| --- |
| **public** **interface** CredentialsMatcher {  **boolean** doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info);  } |

### CredentialsMatcher概览

|  |
| --- |
|  |

#### SimpleCredentialsMatcher

|  |
| --- |
| **public** **class** SimpleCredentialsMatcher **extends** CodecSupport **implements** CredentialsMatcher {  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(SimpleCredentialsMatcher.**class**);    **protected** Object getCredentials(AuthenticationToken token) {  **return** token.getCredentials();  }  **protected** Object getCredentials(AuthenticationInfo info) {  **return** info.getCredentials();  }  **protected** **boolean** equals(Object tokenCredentials, Object accountCredentials) {  **if** (***log***.isDebugEnabled()) {  ***log***.debug("Performing credentials equality check for tokenCredentials of type [" +  tokenCredentials.getClass().getName() + " and accountCredentials of type [" +  accountCredentials.getClass().getName() + "]");  }  **if** (isByteSource(tokenCredentials) && isByteSource(accountCredentials)) {  **if** (***log***.isDebugEnabled()) {  ***log***.debug("Both credentials arguments can be easily converted to byte arrays. Performing " +"array equals comparison");  }  **byte**[] tokenBytes = toBytes(tokenCredentials);  **byte**[] accountBytes = toBytes(accountCredentials);  **return** Arrays.*equals*(tokenBytes, accountBytes);  } **else** {  **return** accountCredentials.equals(tokenCredentials);  }  }  **public** **boolean** doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {  Object tokenCredentials = getCredentials(token);  Object accountCredentials = getCredentials(info);  **return** equals(tokenCredentials, accountCredentials);  }  }  SimpleCredentialsMatcher适用于明文存储比较。 |

#### HashedCredentialsMatcher

|  |
| --- |
| **public** **class** HashedCredentialsMatcher **extends** SimpleCredentialsMatcher {  **private** String hashAlgorithm;//运算法则  **private** **int** hashIterations;//迭代次数  **private** **boolean** hashSalted;//使用盐?  **private** **boolean** storedCredentialsHexEncoded;//hex编码方式?  **public** HashedCredentialsMatcher() {  **this**.hashAlgorithm = **null**;  **this**.hashSalted = **false**;  **this**.hashIterations = 1;  **this**.storedCredentialsHexEncoded = **true**; //false means Base64-encoded  }  **protected** Object getCredentials(AuthenticationInfo info) {  Object credentials = info.getCredentials();  **byte**[] storedBytes = toBytes(credentials);  **if** (credentials **instanceof** String || credentials **instanceof** **char**[]) {  //account.credentials were a char[] or String, so  //we need to do text decoding first:  **if** (isStoredCredentialsHexEncoded()) {  storedBytes = Hex.*decode*(storedBytes);  } **else** {  storedBytes = Base64.*decode*(storedBytes);  }  }  ~~AbstractHash~~ hash = newHashInstance();  hash.setBytes(storedBytes);  **return** hash;  }  **protected** Object hashProvidedCredentials(AuthenticationToken token, AuthenticationInfo info) {  Object salt = **null**;  **if** (info **instanceof** SaltedAuthenticationInfo) {  salt = ((SaltedAuthenticationInfo) info).getCredentialsSalt();  } **else** {  //retain 1.0 backwards compatibility:  **if** (~~isHashSalted~~()) {  salt = ~~getSalt~~(token);  }  }  **return** hashProvidedCredentials(token.getCredentials(), salt, getHashIterations());  }  @Deprecated  **protected** Object ~~getSalt~~(AuthenticationToken token) {  **return** token.getPrincipal();  }  **protected** Hash hashProvidedCredentials(Object credentials, Object salt, **int** hashIterations) {  String hashAlgorithmName = assertHashAlgorithmName();  **return** **new** SimpleHash(hashAlgorithmName, credentials, salt, hashIterations);  }    @Override  **public** **boolean** doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {  Object tokenHashedCredentials = hashProvidedCredentials(token, info);  Object accountCredentials = getCredentials(info);  **return** equals(tokenHashedCredentials, accountCredentials);  } |

## AuthenticationToken

### AuthenticationToken

|  |
| --- |
| **public** **interface** AuthenticationToken **extends** Serializable {  Object getPrincipal();  Object getCredentials();  } |

### AuthenticationToken体系

|  |
| --- |
|  |

#### HostAuthenticationToken

|  |
| --- |
| **public** **interface** HostAuthenticationToken **extends** AuthenticationToken {  String getHost();  } |

#### RememberMeAuthenticationToken

|  |
| --- |
| **public** **interface** RememberMeAuthenticationToken **extends** AuthenticationToken {  **boolean** isRememberMe();  } |

#### UsernamePasswordToken

|  |
| --- |
| **public** **class** UsernamePasswordToken **implements** HostAuthenticationToken, RememberMeAuthenticationToken {  **private** String username;  **private** **char**[] password;  **private** **boolean** rememberMe = **false**;  **private** String host;  **//重要**  **public** Object getPrincipal() {  **return** getUsername();  }  **//重要**  **public** Object getCredentials() {  **return** getPassword();  } |

## AuthenticationInfo

### AuthenticationInfo

|  |
| --- |
| **public** **interface** AuthenticationInfo **extends** Serializable {  PrincipalCollection getPrincipals();  Object getCredentials();  } |

### AuthenticationInfo体系

|  |
| --- |
|  |

#### SaltedAuthenticationInfo

|  |
| --- |
| **public** **interface** SaltedAuthenticationInfo **extends** AuthenticationInfo {  ByteSource getCredentialsSalt();  } |

#### SimpleAuthenticationInfo

|  |
| --- |
| **public** **class** SimpleAuthenticationInfo **implements** MergableAuthenticationInfo, SaltedAuthenticationInfo {  **protected** PrincipalCollection principals;  **protected** Object credentials;  **protected** ByteSource credentialsSalt;  …  } |

## AuthorizationInfo

### AuthorizationInfo

|  |
| --- |
| **public** **interface** AuthorizationInfo **extends** Serializable {  Collection<String> getRoles();    Collection<String> getStringPermissions();    Collection<Permission> getObjectPermissions();  } |

### AuthorizationInfo体系

|  |
| --- |
|  |

#### SimpleAuthorizationInfo

|  |
| --- |
| **public** **class** SimpleAuthorizationInfo **implements** AuthorizationInfo {    **protected** Set<String> roles;    **protected** Set<String> stringPermissions;    **protected** Set<Permission> objectPermissions;    **public** SimpleAuthorizationInfo() {  }    **public** SimpleAuthorizationInfo(Set<String> roles) {  **this**.roles = roles;  }  **public** Set<String> getRoles() {  **return** roles;  }    **public** **void** setRoles(Set<String> roles) {  **this**.roles = roles;  }    **public** **void** addRole(String role) {  **if** (**this**.roles == **null**) {  **this**.roles = **new** HashSet<String>();  }  **this**.roles.add(role);  }    **public** **void** addRoles(Collection<String> roles) {  **if** (**this**.roles == **null**) {  **this**.roles = **new** HashSet<String>();  }  **this**.roles.addAll(roles);  }  **public** Set<String> getStringPermissions() {  **return** stringPermissions;  }    **public** **void** setStringPermissions(Set<String> stringPermissions) {  **this**.stringPermissions = stringPermissions;  }  **public** **void** addStringPermission(String permission) {  **if** (**this**.stringPermissions == **null**) {  **this**.stringPermissions = **new** HashSet<String>();  }  **this**.stringPermissions.add(permission);  }    **public** **void** addStringPermissions(Collection<String> permissions) {  **if** (**this**.stringPermissions == **null**) {  **this**.stringPermissions = **new** HashSet<String>();  }  **this**.stringPermissions.addAll(permissions);  }  **public** Set<Permission> getObjectPermissions() {  **return** objectPermissions;  }  **public** **void** setObjectPermissions(Set<Permission> objectPermissions) {  **this**.objectPermissions = objectPermissions;  }  **public** **void** addObjectPermission(Permission permission) {  **if** (**this**.objectPermissions == **null**) {  **this**.objectPermissions = **new** HashSet<Permission>();  }  **this**.objectPermissions.add(permission);  }    **public** **void** addObjectPermissions(Collection<Permission> permissions) {  **if** (**this**.objectPermissions == **null**) {  **this**.objectPermissions = **new** HashSet<Permission>();  }  **this**.objectPermissions.addAll(permissions);  }  } |

## SessionManager

### SessionManager

|  |
| --- |
| **public** **interface** SessionManager {  Session start(SessionContext context);  Session getSession(SessionKey key) **throws** SessionException;  } |

### SessionManager

|  |
| --- |
|  |

#### AbstractSessionManager

|  |
| --- |
| **public** **abstract** **class** AbstractSessionManager **implements** SessionManager {  **protected** **static** **final** **long** ***MILLIS\_PER\_SECOND*** = 1000;  **protected** **static** **final** **long** ***MILLIS\_PER\_MINUTE*** = 60 \* ***MILLIS\_PER\_SECOND***;  **protected** **static** **final** **long** ***MILLIS\_PER\_HOUR*** = 60 \* ***MILLIS\_PER\_MINUTE***;  **public** **static** **final** **long** ***DEFAULT\_GLOBAL\_SESSION\_TIMEOUT*** = 30 \* ***MILLIS\_PER\_MINUTE***;    **//session 失效时间 默认30分钟**  **private** **long** globalSessionTimeout = ***DEFAULT\_GLOBAL\_SESSION\_TIMEOUT***;  **public** **long** getGlobalSessionTimeout() {  **return** **this**.globalSessionTimeout;  }  **public** **void** setGlobalSessionTimeout(**long** globalSessionTimeout) {  **this**.globalSessionTimeout = globalSessionTimeout;  }  } |

#### AbstractNativeSessionManager

|  |
| --- |
| **public** **abstract** **class** AbstractNativeSessionManager **extends** AbstractSessionManager **implements** NativeSessionManager {    **private** Collection<SessionListener> listeners;  **//重要🡪** **SessionManager#start**  **public** Session start(SessionContext context) {    **//创建session**  Session session = createSession(context);  applyGlobalSessionTimeout(session);  onStart(session, context);  notifyStart(session);  //Don't expose the EIS-tier Session object to the client-tier:    **return** createExposedSession(session, context);  }  **//让子类来实现**  **protected** **abstract** Session createSession(SessionContext context) **throws** AuthorizationException;  **//重要🡪 SessionManager#getSession**  **public** Session getSession(SessionKey key) **throws** SessionException {  Session session = lookupSession(key);  **return** session != **null** ? createExposedSession(session, key) : **null**;  }  **private** Session lookupSession(SessionKey key) **throws** SessionException {  **if** (key == **null**) {  **throw** **new** NullPointerException("SessionKey argument cannot be null.");  }  **return** doGetSession(key);  }  **//让子类来实现**  **protected** **abstract** Session doGetSession(SessionKey key) **throws** InvalidSessionException;  **//DefaultWebSessionManage会重载此方法**  **protected** Session createExposedSession(Session session, SessionContext context) {  **return** **new** DelegatingSession(**this**, **new** DefaultSessionKey(session.getId()));  }    **//DefaultWebSessionManage会重载此方法**  **protected** Session createExposedSession(Session session, SessionKey key) {  **return** **new** DelegatingSession(**this**, **new** DefaultSessionKey(session.getId()));  } |

#### AbstractValidatingSessionManager(验证)

|  |
| --- |
| **public** **static** **final** **long** ***DEFAULT\_SESSION\_VALIDATION\_INTERVAL*** = ***MILLIS\_PER\_HOUR***;  **protected** **boolean** sessionValidationSchedulerEnabled;  **protected** SessionValidationScheduler sessionValidationScheduler;  **protected** **long** sessionValidationInterval;  **public** AbstractValidatingSessionManager() {  **this**.sessionValidationSchedulerEnabled = **true**;  **this**.sessionValidationInterval = ***DEFAULT\_SESSION\_VALIDATION\_INTERVAL***;  }  **protected** **final** Session doGetSession(**final** SessionKey key) **throws** InvalidSessionException {  enableSessionValidationIfNecessary();  ***log***.trace("Attempting to retrieve session with key {}", key);  Session s = retrieveSession(key);  **if** (s != **null**) {  validate(s, key);  }  **return** s;  }  **//让子类来实现**  **protected** **abstract** Session retrieveSession(SessionKey key) **throws** UnknownSessionException;  **protected** Session createSession(SessionContext context) **throws** AuthorizationException {  enableSessionValidationIfNecessary();  **return** doCreateSession(context);  }  **//让子类来实现**  **protected** **abstract** Session doCreateSession(SessionContext initData) **throws** AuthorizationException;  } |

#### DefaultSessionManager

|  |
| --- |
| **public** **class** DefaultSessionManager **extends** AbstractValidatingSessionManager **implements** CacheManagerAware {    **private** SessionFactory sessionFactory;  **protected** SessionDAO sessionDAO; //todo - move SessionDAO up to AbstractValidatingSessionManager?  **private** CacheManager cacheManager;  **private** **boolean** deleteInvalidSessions;  **public** DefaultSessionManager() {  **this**.deleteInvalidSessions = **true**;  **this**.sessionFactory = **new** SimpleSessionFactory();//sessionFactory  **this**.sessionDAO = **new** MemorySessionDAO();//内存sessionDao  }  **//重要**  **protected** Session doCreateSession(SessionContext context) {  Session s = newSessionInstance(context);  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Creating session for host {}", s.getHost());  }  create(s);  **return** s;  }  **protected** Session newSessionInstance(SessionContext context) {  **return** getSessionFactory().createSession(context);//看来session是由sessionFactory来创建的。  }  **protected** **void** create(Session session) {  **if** (***log***.isDebugEnabled()) {  ***log***.debug("Creating new EIS record for new session instance [" + session + "]");  }  sessionDAO.create(session);  }  **//重要**  **protected** Session retrieveSession(SessionKey sessionKey) **throws** UnknownSessionException {  Serializable sessionId = getSessionId(sessionKey);  **if** (sessionId == **null**) {  ***log***.debug("Unable to resolve session ID from SessionKey [{}]. Returning null to indicate a " +  "session could not be found.", sessionKey);  **return** **null**;  }  Session s = retrieveSessionFromDataSource(sessionId);  **if** (s == **null**) {  //session ID was provided, meaning one is expected to be found, but we couldn't find one:  String msg = "Could not find session with ID [" + sessionId + "]";  **throw** **new** UnknownSessionException(msg);  }  **return** s;  }  **protected** Serializable getSessionId(SessionKey sessionKey) {  **return** sessionKey.getSessionId();  }  **protected** Session retrieveSessionFromDataSource(Serializable sessionId) **throws** UnknownSessionException {  **return** sessionDAO.readSession(sessionId);  } |

#### DefaultWebSessionManager

|  |
| --- |
| **public** **class** DefaultWebSessionManager **extends** DefaultSessionManager **implements** WebSessionManager {  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(DefaultWebSessionManager.**class**);  **private** Cookie sessionIdCookie;  **private** **boolean** sessionIdCookieEnabled;  **protected** Session createExposedSession(Session session, SessionContext context) {  **if** (!WebUtils.*isWeb*(context)) {  **return** **super**.createExposedSession(session, context);  }  ServletRequest request = WebUtils.*getRequest*(context);  ServletResponse response = WebUtils.*getResponse*(context);  SessionKey key = **new** WebSessionKey(session.getId(), request, response);  **return** **new** DelegatingSession(**this**, key);  }  **protected** Session createExposedSession(Session session, SessionKey key) {  **if** (!WebUtils.*isWeb*(key)) {  **return** **super**.createExposedSession(session, key);  }  ServletRequest request = WebUtils.*getRequest*(key);  ServletResponse response = WebUtils.*getResponse*(key);  SessionKey sessionKey = **new** WebSessionKey(session.getId(), request, response);  **return** **new** DelegatingSession(**this**, sessionKey);  } |

### SessionFactory

|  |
| --- |
| **public** **interface** SessionFactory {    Session createSession(SessionContext initData);  }  **public** **class** SimpleSessionFactory **implements** SessionFactory {  **public** Session createSession(SessionContext initData) {  **if** (initData != **null**) {  String host = initData.getHost();  **if** (host != **null**) {  **return** **new** SimpleSession(host);  }  }  **return** **new** SimpleSession();  }  } |

## Session

### Session

|  |
| --- |
| **public** **interface** Session {    Serializable getId();    Date getStartTimestamp();    Date getLastAccessTime();    **long** getTimeout() **throws** InvalidSessionException;    **void** setTimeout(**long** maxIdleTimeInMillis) **throws** InvalidSessionException;    String getHost();    **void** touch() **throws** InvalidSessionException;    **void** stop() **throws** InvalidSessionException;    Collection<Object> getAttributeKeys() **throws** InvalidSessionException;    Object getAttribute(Object key) **throws** InvalidSessionException;    **void** setAttribute(Object key, Object value) **throws** InvalidSessionException;    Object removeAttribute(Object key) **throws** InvalidSessionException;  } |

### Session体系

|  |
| --- |
|  |

#### DelegatingSession

|  |
| --- |
| **public** **class** DelegatingSession **implements** Session, Serializable {  //**TODO** - complete JavaDoc  **private** **final** SessionKey key;  //cached fields to avoid a server-side method call if out-of-process:  **private** Date startTimestamp = **null**;  **private** String host = **null**;  /\*\*  \* Handle to the target NativeSessionManager that will support the delegate calls.  \*/  **private** **final** **transient** NativeSessionManager sessionManager;  **public** DelegatingSession(NativeSessionManager sessionManager, SessionKey key) {  **if** (sessionManager == **null**) {  **throw** **new** IllegalArgumentException("sessionManager argument cannot be null.");  }  **if** (key == **null**) {  **throw** **new** IllegalArgumentException("sessionKey argument cannot be null.");  }  **if** (key.getSessionId() == **null**) {  String msg = "The " + DelegatingSession.**class**.getName() + " implementation requires that the " +  "SessionKey argument returns a non-null sessionId to support the " +  "Session.getId() invocations.";  **throw** **new** IllegalArgumentException(msg);  }  **this**.sessionManager = sessionManager;  **this**.key = key;  }  } |

#### HttpServletSession

|  |
| --- |
| **public** **class** HttpServletSession **implements** Session {  **private** **static** **final** String ***HOST\_SESSION\_KEY*** = HttpServletSession.**class**.getName() + ".HOST\_SESSION\_KEY";  **private** **static** **final** String ***TOUCH\_OBJECT\_SESSION\_KEY*** = HttpServletSession.**class**.getName() + ".TOUCH\_OBJECT\_SESSION\_KEY";  **private** HttpSession httpSession = **null**;  **public** HttpServletSession(HttpSession httpSession, String host) {  **if** (httpSession == **null**) {  String msg = "HttpSession constructor argument cannot be null.";  **throw** **new** IllegalArgumentException(msg);  }  **if** (httpSession **instanceof** ShiroHttpSession) {  String msg = "HttpSession constructor argument cannot be an instance of ShiroHttpSession. This " +  "is enforced to prevent circular dependencies and infinite loops.";  **throw** **new** IllegalArgumentException(msg);  }  **this**.httpSession = httpSession;  **if** (StringUtils.*hasText*(host)) {  setHost(host);  }  }  } |

#### SimpleSession

|  |
| --- |
| **public** **class** SimpleSession **implements** ValidatingSession, Serializable {// ValidatingSession继承自session    **private** **static** **final** **long** ***serialVersionUID*** = -7125642695178165650L;  //**TODO** - complete JavaDoc  **private** **transient** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(SimpleSession.**class**);  **protected** **static** **final** **long** ***MILLIS\_PER\_SECOND*** = 1000;  **protected** **static** **final** **long** ***MILLIS\_PER\_MINUTE*** = 60 \* ***MILLIS\_PER\_SECOND***;  **protected** **static** **final** **long** ***MILLIS\_PER\_HOUR*** = 60 \* ***MILLIS\_PER\_MINUTE***;  //serialization bitmask fields. DO NOT CHANGE THE ORDER THEY ARE DECLARED!  **static** **int** *bitIndexCounter* = 0;  **private** **static** **final** **int** ***ID\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***START\_TIMESTAMP\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***STOP\_TIMESTAMP\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***LAST\_ACCESS\_TIME\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***TIMEOUT\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***EXPIRED\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***HOST\_BIT\_MASK*** = 1 << *bitIndexCounter*++;  **private** **static** **final** **int** ***ATTRIBUTES\_BIT\_MASK*** = 1 << *bitIndexCounter*++;    // ==============================================================  **private** **transient** Serializable id;  **private** **transient** Date startTimestamp;  **private** **transient** Date stopTimestamp;  **private** **transient** Date lastAccessTime;  **private** **transient** **long** timeout;  **private** **transient** **boolean** expired;  **private** **transient** String host;  **private** **transient** Map<Object, Object> attributes;  **public** SimpleSession() {  **this**.timeout = DefaultSessionManager.***DEFAULT\_GLOBAL\_SESSION\_TIMEOUT***;  **this**.startTimestamp = **new** Date();  **this**.lastAccessTime = **this**.startTimestamp;  }  } |

## Subject

### Suject

|  |
| --- |
| **public** **interface** Subject {    Object getPrincipal();  PrincipalCollection getPrincipals();    **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;    **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;    **void** login(AuthenticationToken token) **throws** AuthenticationException;    **boolean** isAuthenticated();  **boolean** isRemembered();    Session getSession();    Session getSession(**boolean** create);  **void** logout();    <V> V execute(Callable<V> callable) **throws** ExecutionException;    **void** execute(Runnable runnable);    <V> Callable<V> associateWith(Callable<V> callable);    Runnable associateWith(Runnable runnable);    **void** runAs(PrincipalCollection principals) **throws** NullPointerException, IllegalStateException;    **boolean** isRunAs();    PrincipalCollection getPreviousPrincipals();    PrincipalCollection releaseRunAs();    **public** **static** **class** Builder {    **private** **final** SubjectContext subjectContext;    **private** **final** SecurityManager securityManager;    **public** Builder() {  **this**(SecurityUtils.*getSecurityManager*());  }    **public** Builder(SecurityManager securityManager) {  **if** (securityManager == **null**) {  **throw** **new** NullPointerException("SecurityManager method argument cannot be null.");  }  **this**.securityManager = securityManager;  **this**.subjectContext = newSubjectContextInstance();  **if** (**this**.subjectContext == **null**) {  **throw** **new** IllegalStateException("Subject instance returned from 'newSubjectContextInstance' " +  "cannot be null.");  }  **this**.subjectContext.setSecurityManager(securityManager);  }    **protected** SubjectContext newSubjectContextInstance() {  **return** **new** DefaultSubjectContext();  }    **protected** SubjectContext getSubjectContext() {  **return** **this**.subjectContext;  }    **public** Builder sessionId(Serializable sessionId) {  **if** (sessionId != **null**) {  **this**.subjectContext.setSessionId(sessionId);  }  **return** **this**;  }    **public** Builder host(String host) {  **if** (StringUtils.*hasText*(host)) {  **this**.subjectContext.setHost(host);  }  **return** **this**;  }    **public** Builder session(Session session) {  **if** (session != **null**) {  **this**.subjectContext.setSession(session);  }  **return** **this**;  }    **public** Builder principals(PrincipalCollection principals) {  **if** (!CollectionUtils.*isEmpty*(principals)) {  **this**.subjectContext.setPrincipals(principals);  }  **return** **this**;  }    **public** Builder sessionCreationEnabled(**boolean** enabled) {  **this**.subjectContext.setSessionCreationEnabled(enabled);  **return** **this**;  }    **public** Builder authenticated(**boolean** authenticated) {  **this**.subjectContext.setAuthenticated(authenticated);  **return** **this**;  }    **public** Builder contextAttribute(String attributeKey, Object attributeValue) {  **if** (attributeKey == **null**) {  String msg = "Subject context map key cannot be null.";  **throw** **new** IllegalArgumentException(msg);  }  **if** (attributeValue == **null**) {  **this**.subjectContext.remove(attributeKey);  } **else** {  **this**.subjectContext.put(attributeKey, attributeValue);  }  **return** **this**;  }  **public** Subject buildSubject() {  **return** **this**.securityManager.createSubject(**this**.subjectContext);  }  }  } |

### Suject 体系

|  |
| --- |
|  |

#### WebSubject

|  |
| --- |
| **public** **interface** WebSubject **extends** Subject, RequestPairSource {    ServletRequest getServletRequest();    ServletResponse getServletResponse();  } |

#### DelegatingSubject（代理）

|  |
| --- |
| **public** **void** login(AuthenticationToken token) **throws** AuthenticationException {  clearRunAsIdentitiesInternal();  **Subject subject = securityManager.login(this, token);**  PrincipalCollection principals;  String host = **null**;  **if** (subject **instanceof** DelegatingSubject) {  DelegatingSubject delegating = (DelegatingSubject) subject;  //we have to do this in case there are assumed identities - we don't want to lose the 'real' principals:  principals = delegating.principals;  host = delegating.host;  } **else** {  principals = subject.getPrincipals();  }  **if** (principals == **null** || principals.isEmpty()) {  String msg = "Principals returned from securityManager.login( token ) returned a null or " +  "empty value. This value must be non null and populated with one or more elements.";  **throw** **new** IllegalStateException(msg);  }  **this**.principals = principals;  **this**.authenticated = **true**;  **if** (token **instanceof** HostAuthenticationToken) {  host = ((HostAuthenticationToken) token).getHost();  }  **if** (host != **null**) {  **this**.host = host;  }  **Session session = subject.getSession(false);**  **if** (session != **null**) {  **this**.session = decorate(session);  } **else** {  **this**.session = **null**;  }  }  **public** Session getSession(**boolean** create) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("attempting to get session; create = " + create +  "; session is null = " + (**this**.session == **null**) +  "; session has id = " + (**this**.session != **null** && session.getId() != **null**));  }  **if** (**this**.session == **null** && create) {  //added in 1.2:  **if** (!isSessionCreationEnabled()) {  String msg = "Session creation has been disabled for the current subject. This exception indicates " +  "that there is either a programming error (using a session when it should never be " +  "used) or that Shiro's configuration needs to be adjusted to allow Sessions to be created " +  "for the current Subject. See the " + DisabledSessionException.**class**.getName() + " JavaDoc " +  "for more.";  **throw** **new** DisabledSessionException(msg);  }  ***log***.trace("Starting session for host {}", getHost());  **SessionContext sessionContext = createSessionContext();**  **Session session = this.securityManager.start(sessionContext);**  **this.session = decorate(session);**  }  **return** **this**.session;  }  **//当为WebDelegatingSubject将会重载此方法**  **protected** SessionContext createSessionContext() {  SessionContext sessionContext = **new** DefaultSessionContext();  **if** (StringUtils.*hasText*(host)) {  sessionContext.setHost(host);  }  **return** sessionContext;  }  **public** <V> V execute(Callable<V> callable) **throws** ExecutionException {  Callable<V> associated = associateWith(callable);  **try** {  **return** associated.call();  } **catch** (Throwable t) {  **throw** **new** ExecutionException(t);  }  }  **public** <V> Callable<V> associateWith(Callable<V> callable) {  **return** **new** SubjectCallable<V>(**this**, callable);  } |

#### WebDelegatingSubject

|  |
| --- |
| **public** **class** WebDelegatingSubject **extends** DelegatingSubject **implements** WebSubject {  **private** **final** ServletRequest servletRequest;  **private** **final** ServletResponse servletResponse;  **public** WebDelegatingSubject(PrincipalCollection principals, **boolean** authenticated,  String host, Session session,  ServletRequest request, ServletResponse response,  SecurityManager securityManager) {  **this**(principals, authenticated, host, session, **true**, request, response, securityManager);  }  //since 1.2  **public** WebDelegatingSubject(PrincipalCollection principals, **boolean** authenticated,  String host, Session session, **boolean** sessionEnabled,  ServletRequest request, ServletResponse response,  SecurityManager securityManager) {  **super**(principals, authenticated, host, session, sessionEnabled, securityManager);  **this**.servletRequest = request;  **this**.servletResponse = response;  }    @Override  **protected** SessionContext createSessionContext() {  WebSessionContext wsc = **new** DefaultWebSessionContext();  String host = getHost();  **if** (StringUtils.*hasText*(host)) {  wsc.setHost(host);  }  wsc.setServletRequest(**this**.servletRequest);  wsc.setServletResponse(**this**.servletResponse);  **return** wsc;  } |

## Filter

### Filter

|  |
| --- |
| **public** **interface** Filter {  **public** **void** init(FilterConfig filterConfig) **throws** ServletException;    **public** **void** doFilter ( ServletRequest request, ServletResponse response, FilterChain chain ) **throws** IOException, ServletException;  **public** **void** **destroy**();  } |

### Shiro filter 体系

|  |
| --- |
|  |

|  |
| --- |
|  |

#### AbstractFilter

|  |
| --- |
| **public** **abstract** **class** AbstractFilter **extends** ServletContextSupport **implements** Filter {  **private** **static** **transient** **final** Logger ***log*** = LoggerFactory.*getLogger*(AbstractFilter.**class**);  **protected** FilterConfig filterConfig  **public** **final** **void** init(FilterConfig filterConfig) **throws** ServletException {  setFilterConfig(filterConfig);  **try** {  onFilterConfigSet();  } **catch** (Exception e) {  **if** (e **instanceof** ServletException) {  **throw** (ServletException) e;  } **else** {  **if** (***log***.isErrorEnabled()) {  ***log***.error("Unable to start Filter: [" + e.getMessage() + "].", e);  }  **throw** **new** ServletException(e);  }  }  }  **public** **void** setFilterConfig(FilterConfig filterConfig) {  **this**.filterConfig = filterConfig;  setServletContext(filterConfig.getServletContext());  }  **//供子类实现**  **protected** **void** onFilterConfigSet() **throws** Exception {} |

#### NameableFilter

|  |
| --- |
| **public** **abstract** **class** NameableFilter **extends** AbstractFilter **implements** Nameable {  **private** String name;  //默认过滤器名字是web.xml配置的name  **protected** String getName() {  **if** (**this**.name == **null**) {  FilterConfig config = getFilterConfig();  **if** (config != **null**) {  **this**.name = config.getFilterName();  }  }  **return** **this**.name;  }  //当然你也可以自定义name  **public** **void** setName(String name) {  **this**.name = name;  } |

#### OncePerRequestFilter

|  |
| --- |
| **public** **abstract** **class** OncePerRequestFilter **extends** NameableFilter {  **public** **static** **final** String ***ALREADY\_FILTERED\_SUFFIX*** = ".FILTERED";  //默认对于某过滤器一次请求只走一次  **private** **boolean** enabled = **true**;  **protected** String getAlreadyFilteredAttributeName() {  String name = getName();  **if** (name == **null**) {  name = getClass().getName();  }  **return** name + ***ALREADY\_FILTERED\_SUFFIX***;  }  **public** **final** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)  **throws** ServletException, IOException {  String alreadyFilteredAttributeName = getAlreadyFilteredAttributeName();  **if** ( request.getAttribute(alreadyFilteredAttributeName) != **null** ) {**//已经走过（言外之意enabled=true），跳过**  ***log***.trace("Filter '{}' already executed. Proceeding without invoking this filter.", getName());  filterChain.doFilter(request, response);  } **else** //noinspection deprecation //  **if** (/\* added in 1.2: \*/ !isEnabled(request, response) ||  /\* retain backwards compatibility: \*/ ~~shouldNotFilter~~(request) ) {**//还没走过，若enable=false跳过**  ***log***.debug("Filter '{}' is not enabled for the current request. Proceeding without invoking this filter.",  getName());  filterChain.doFilter(request, response);  } **else** {**//还没走过，enable=true,则….**  // Do invoke this filter...  ***log***.trace("Filter '{}' not yet executed. Executing now.", getName());  request.setAttribute(alreadyFilteredAttributeName, Boolean.***TRUE***);  **try** {  doFilterInternal(request, response, filterChain);  } **finally** {  // Once the request has finished, we're done and we don't  // need to mark as 'already filtered' any more.  request.removeAttribute(alreadyFilteredAttributeName);  }  }  **//子类来实现**  **protected** **abstract** **void** doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain) **throws** ServletException, IOException;    } |

#### AbstractShiroFilter

|  |
| --- |
| **public** **abstract** **class** AbstractShiroFilter **extends** OncePerRequestFilter {  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(AbstractShiroFilter.**class**);  **private** **static** **final** String ***STATIC\_INIT\_PARAM\_NAME*** = "staticSecurityManagerEnabled";  **private** WebSecurityManager securityManager;  **private** FilterChainResolver filterChainResolver;  **private** **boolean** staticSecurityManagerEnabled;  **protected** **final** **void** onFilterConfigSet() **throws** Exception {  //added in 1.2 for SHIRO-287:  applyStaticSecurityManagerEnabledConfig();  init();  ensureSecurityManager();  //added in 1.2 for SHIRO-287:  **if** (isStaticSecurityManagerEnabled()) {  **//非常重要🡪**  SecurityUtils.*setSecurityManager*(getSecurityManager());  }  }  **private** **void** applyStaticSecurityManagerEnabledConfig() {  String value = getInitParam(***STATIC\_INIT\_PARAM\_NAME***);  **if** (value != **null**) {  Boolean b = Boolean.*valueOf*(value);  **if** (b != **null**) {  setStaticSecurityManagerEnabled(b);  }  }  }  **public** **void** init() **throws** Exception {}  **private** **void** ensureSecurityManager() {  WebSecurityManager securityManager = getSecurityManager();  **if** (securityManager == **null**) {  ***log***.info("No SecurityManager configured. Creating default.");  securityManager = createDefaultSecurityManager();  setSecurityManager(securityManager);  }  }  **protected** WebSecurityManager createDefaultSecurityManager() {  **return** **new** DefaultWebSecurityManager();**//重要**  }    **//包装原生的httprequest**  @SuppressWarnings({"UnusedDeclaration"})  **protected** ServletRequest prepareServletRequest(ServletRequest request, ServletResponse response, FilterChain chain) {  ServletRequest toUse = request;  **if** (request **instanceof** HttpServletRequest) {  HttpServletRequest http = (HttpServletRequest) request;  toUse = wrapServletRequest(http);  }  **return** toUse;  }  **protected** ServletRequest wrapServletRequest(HttpServletRequest orig) {  **return** **new** ShiroHttpServletRequest(orig, getServletContext(), isHttpSessions());  }  **//包装原生的httpresponse**  @SuppressWarnings({"UnusedDeclaration"})  **protected** ServletResponse prepareServletResponse(ServletRequest request, ServletResponse response, FilterChain chain) {  ServletResponse toUse = response;  **if** (!isHttpSessions() && (request **instanceof** ShiroHttpServletRequest) &&  (response **instanceof** HttpServletResponse)) {  //the ShiroHttpServletResponse exists to support URL rewriting for session ids. This is only needed if  //using Shiro sessions (i.e. not simple HttpSession based sessions):  toUse = wrapServletResponse((HttpServletResponse) response, (ShiroHttpServletRequest) request);  }  **return** toUse;  }  **protected** ServletResponse wrapServletResponse(HttpServletResponse orig, ShiroHttpServletRequest request) {  **return** **new** ShiroHttpServletResponse(orig, getServletContext(), request);  }    **//创建subject-------🡪非常重要**  **protected** WebSubject createSubject(ServletRequest request, ServletResponse response) {  **return** **new** WebSubject.Builder(getSecurityManager(), request, response).buildWebSubject();  }      **//创建subject-------🡪非常重要**  **protected** **void** doFilterInternal(ServletRequest servletRequest, ServletResponse servletResponse, **final** FilterChain chain)  **throws** ServletException, IOException {  Throwable t = **null**;  **try** {  **final** ServletRequest request = prepareServletRequest(servletRequest, servletResponse, chain);  **final** ServletResponse response = prepareServletResponse(request, servletResponse, chain);    **//非常重要**  **final** Subject subject = createSubject(request, response);  //noinspection unchecked  subject.execute(**new** Callable() {  **public** Object call() **throws** Exception {  updateSessionLastAccessTime(request, response);  executeChain(request, response, chain);  **return** **null**;  }  });  } **catch** (ExecutionException ex) {  t = ex.getCause();  } **catch** (Throwable throwable) {  t = throwable;  }  **if** (t != **null**) {  **if** (t **instanceof** ServletException) {  **throw** (ServletException) t;  }  **if** (t **instanceof** IOException) {  **throw** (IOException) t;  }  //otherwise it's not one of the two exceptions expected by the filter method signature - wrap it in one:  String msg = "Filtered request failed.";  **throw** **new** ServletException(msg, t);  }  }  **protected** **void** executeChain(ServletRequest request, ServletResponse response, FilterChain origChain)  **throws** IOException, ServletException {  //获取shiro 包装origChain的FilterChain  //首先执行权限过滤器再执行origChain的过滤器  FilterChain chain = getExecutionChain(request, response, origChain);  chain.doFilter(request, response);  }  **protected** FilterChain getExecutionChain(ServletRequest request, ServletResponse response, FilterChain origChain) {  FilterChain chain = origChain;  FilterChainResolver resolver = getFilterChainResolver();  **if** (resolver == **null**) {  ***log***.debug("No FilterChainResolver configured. Returning original FilterChain.");  **return** origChain;  }  //获取shiro的filterChain.  FilterChain resolved = resolver.getChain(request, response, origChain);  **if** (resolved != **null**) {  ***log***.trace("Resolved a configured FilterChain for the current request.");  chain = resolved;  } **else** {  ***log***.trace("No FilterChain configured for the current request. Using the default.");  }  **return** chain;  } |

#### ShiroFilter

|  |
| --- |
| **public** **class** ShiroFilter **extends** AbstractShiroFilter {    @Override  **public** **void** init() **throws** Exception {  WebEnvironment env = WebUtils.*getRequiredWebEnvironment*(getServletContext());  setSecurityManager(env.getWebSecurityManager());  FilterChainResolver resolver = env.getFilterChainResolver();  **if** (resolver != **null**) {  setFilterChainResolver(resolver);  }  }  } |

#### AdviceFilter

|  |
| --- |
| **public** **abstract** **class** AdviceFilter **extends** OncePerRequestFilter {    **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(AdviceFilter.**class**);  **protected** **boolean** preHandle(ServletRequest request, ServletResponse response) **throws** Exception {  **return** **true**;  }    @SuppressWarnings({"UnusedDeclaration"})  **protected** **void** postHandle(ServletRequest request, ServletResponse response) **throws** Exception {  }    @SuppressWarnings({"UnusedDeclaration"})  **public** **void** afterCompletion(ServletRequest request, ServletResponse response, Exception exception) **throws** Exception {  }  **protected** **void** executeChain(ServletRequest request, ServletResponse response, FilterChain chain) **throws** Exception {  chain.doFilter(request, response);  }    **public** **void** doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** ServletException, IOException {  Exception exception = **null**;  **try** {  **//执行本过滤器前置增强**  **boolean** continueChain = preHandle(request, response);  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Invoked preHandle method. Continuing chain?: [" + continueChain + "]");  }    **//本过滤器前置增强结果为true,则继续执行其余过滤器，否则其余过滤器不再执行**  **// 这里的if判断决定了是否执行余下所有的过滤器**  **if** (continueChain) {  executeChain(request, response, chain);  }    **//执行本过滤器的后置增强**  postHandle(request, response);  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Successfully invoked postHandle method");  }  } **catch** (Exception e) {  exception = e;  } **finally** {  **//清理，执行最终增强**  cleanup(request, response, exception);  }  }  **protected** **void** cleanup(ServletRequest request, ServletResponse response, Exception existing)  **throws** ServletException, IOException {  Exception exception = existing;  **try** {  //  afterCompletion(request, response, exception);  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Successfully invoked afterCompletion method.");  }  } **catch** (Exception e) {  **if** (exception == **null**) {  exception = e;  } **else** {  ***log***.debug("afterCompletion implementation threw an exception. This will be ignored to " +  "allow the original source exception to be propagated.", e);  }  }  **if** (exception != **null**) {  **if** (exception **instanceof** ServletException) {  **throw** (ServletException) exception;  } **else** **if** (exception **instanceof** IOException) {  **throw** (IOException) exception;  } **else** {  **if** (***log***.isDebugEnabled()) {  String msg = "Filter execution resulted in an unexpected Exception " +  "(not IOException or ServletException as the Filter API recommends). " +  "Wrapping in ServletException and propagating.";  ***log***.debug(msg);  }  **throw** **new** ServletException(exception);  }  }  }  } |

#### PathMatchingFilter

|  |
| --- |
| **public** **abstract** **class** PathMatchingFilter **extends** AdviceFilter **implements** PathConfigProcessor {  **例如对于配置**“  …  /act/rest/service/editor/\*\* = perms[act:model:edit]”,  /act/rest/service/editor/manage/\*\* = perms[act:model:add]”,  …  ”  **//**PermissionsAuthorizationFilter的 appliedPaths为：  {“/act/rest/service/editor/\*\*”:” act:model:add”,” /act/rest/service/editor/manage/\*\*”:” act:model:add”}  **protected** PatternMatcher pathMatcher = **new** AntPathMatcher();  **protected** Map<String, Object> appliedPaths = **new** LinkedHashMap<String, Object>();  **//前置增强，若没有配置则前置增强返回true**  **protected** **boolean** preHandle(ServletRequest request, ServletResponse response) **throws** Exception {  **if** (**this**.appliedPaths == **null** || **this**.appliedPaths.isEmpty()) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("appliedPaths property is null or empty. This Filter will passthrough immediately.");  }  **return** **true**;  }    **for** (String path : **this**.appliedPaths.keySet()) {  // If the path does match, then pass on to the subclass implementation for specific checks  //(first match 'wins'):  **if** (pathsMatch(path, request)) {  ***log***.trace("Current requestURI matches pattern '{}'. Determining filter chain execution...", path);  Object config = **this**.appliedPaths.get(path);  **return** isFilterChainContinued(request, response, path, config);  }  }  //no path matched, allow the request to go through:  **return** **true**;  }  **protected** **boolean** pathsMatch(String path, ServletRequest request) {  String requestURI = getPathWithinApplication(request);  ***log***.trace("Attempting to match pattern '{}' with current requestURI '{}'...", path, requestURI);  **return** pathsMatch(path, requestURI);  }  **protected** String getPathWithinApplication(ServletRequest request) {  **return** WebUtils.*getPathWithinApplication*(WebUtils.*toHttp*(request));  }  **protected** **boolean** pathsMatch(String pattern, String path) {  **return** pathMatcher.matches(pattern, path);  }  @SuppressWarnings({"JavaDoc"})  **private** **boolean** isFilterChainContinued(ServletRequest request, ServletResponse response,  String path, Object pathConfig) **throws** Exception {  **if** (isEnabled(request, response, path, pathConfig)) { //isEnabled check added in 1.2  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Filter '{}' is enabled for the current request under path '{}' with config [{}]. " +  "Delegating to subclass implementation for 'onPreHandle' check.",  **new** Object[]{getName(), path, pathConfig});  }  //The filter is enabled for this specific request, so delegate to subclass implementations  //so they can decide if the request should continue through the chain or not:  **return** onPreHandle(request, response, pathConfig);  }  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Filter '{}' is disabled for the current request under path '{}' with config [{}]. " +  "The next element in the FilterChain will be called immediately.",  **new** Object[]{getName(), path, pathConfig});  }  //This filter is disabled for this specific request,  //return 'true' immediately to indicate that the filter will not process the request  //and let the request/response to continue through the filter chain:  **return** **true**;  }  **//供子类来实现**  **protected** **boolean** onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue) **throws** Exception {  **return** **true**;  } |

#### AccessControlFilter

|  |
| --- |
| **public** **abstract** **class** AccessControlFilter **extends** PathMatchingFilter {  **public** **static** **final** String ***DEFAULT\_LOGIN\_URL*** = "/login.jsp";  **public** **static** **final** String ***GET\_METHOD*** = "GET";  **public** **static** **final** String ***POST\_METHOD*** = "POST";  **private** String loginUrl = ***DEFAULT\_LOGIN\_URL***;  **//为何能使用SecurityUtils来获取Subject,请参见 AbstractShiroFilter**  **protected** Subject getSubject(ServletRequest request, ServletResponse response) {  **return** SecurityUtils.*getSubject*();  }  **//归属于前置增强（是否允许访问，访问是否已处理）**  **public** **boolean** onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue) **throws** Exception {  //这里非常重要，一定要理解  //1、isAccessAllowed和onAccessDenied共同决定当前请求是否要走其余的过滤器。  //2、如果isAccessAllowed=true，则不会再执行“onAccessDenied”  //3、如果isAccessAllowed=false，则结果由onAccessDenied来决定  **return** isAccessAllowed(request, response, mappedValue) || onAccessDenied (request, response, mappedValue);  }  **//归属于前置增强（是否允许通过当前过滤器）**  **protected** **abstract** **boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) **throws** Exception;  **//归属于前置增强（当不允许通过当前过滤器，是否已处理）**  **protected** **boolean** onAccessDenied(ServletRequest request, ServletResponse response, Object mappedValue) **throws** Exception {  **return** onAccessDenied(request, response);  }    //保存当前请求，重定向到登陆页面  **protected** **void** saveRequestAndRedirectToLogin(ServletRequest request, ServletResponse response) **throws** IOException {  saveRequest(request);  redirectToLogin(request, response);  }    **protected** **void** saveRequest(ServletRequest request) {  WebUtils.*saveRequest*(request);  }    **protected** **void** redirectToLogin(ServletRequest request, ServletResponse response) **throws** IOException {  String loginUrl = getLoginUrl();  WebUtils.*issueRedirect*(request, response, loginUrl);  }  **protected** **boolean** isLoginRequest(ServletRequest request, ServletResponse response) {  **return** pathsMatch(getLoginUrl(), request);  } |

#### AuthenticationFilter

|  |
| --- |
| **public** **abstract** **class** AuthenticationFilter **extends** AccessControlFilter {  //**TODO** - complete JavaDoc  **public** **static** **final** String ***DEFAULT\_SUCCESS\_URL*** = "/";  **private** String successUrl = ***DEFAULT\_SUCCESS\_URL***;  **//subject 已验证通过，则允许通过该过滤器**  **protected** **boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {  Subject subject = getSubject(request, response);  **return** subject.isAuthenticated();  }  **//确保登陆成功后重定向**  **protected** **void** issueSuccessRedirect(ServletRequest request, ServletResponse response) **throws** Exception {  WebUtils.*redirectToSavedRequest*(request, response, getSuccessUrl());  } |

#### AuthenticatingFilter

|  |
| --- |
| **public** **abstract** **class** AuthenticatingFilter **extends** AuthenticationFilter {  **public** **static** **final** String ***PERMISSIVE*** = "permissive";  @Override  //1、subject 已验证身份 ，则返回ture  //2、subject 未验证身份 如果是登陆请求，返回false  //3、subject 未验证身份 如果是非登陆请求且有权限标识返回true  //4、subject 未验证身份 如果是非登陆请求且无权限标识返回false  **protected** **boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {  **return** **super**.isAccessAllowed(request, response, mappedValue) ||  (!isLoginRequest(request, response) && isPermissive(mappedValue));// isLoginRequest是  AccessControlFilter方法  }  **//有配置 则true，否则false**  //mappedValue为[[act:model:edit], [act:model:add]]  **protected** **boolean** isPermissive(Object mappedValue) {  **if**(mappedValue != **null**) {  String[] values = (String[]) mappedValue;  **return** Arrays.*binarySearch*(values, ***PERMISSIVE***) >= 0;  }  **return** **false**;  }  **// 子类FormAuthenticationFilter将调用该方法**  **protected** **boolean** executeLogin(ServletRequest request, ServletResponse response) **throws** Exception {  AuthenticationToken token = createToken(request, response);  **if** (token == **null**) {  String msg = "createToken method implementation returned null. A valid non-null AuthenticationToken " +  "must be created in order to execute a login attempt.";  **throw** **new** IllegalStateException(msg);  }  **try** {  Subject subject = getSubject(request, response);  subject.login(token);  //是否通过当前过滤器，由onLoginSuccess决定，默认是不通过（请看子类FormAuthenticationFileter的该方法）。  **return** onLoginSuccess(token, subject, request, response);//此方法  } **catch** (AuthenticationException e) {  //有异常，则由onLoginFailure决定，是否通过当前过滤器，默认通过（请看子类FormAuthenticationFileter的该方法）  **return** onLoginFailure(token, e, request, response);  }  }  //默认情况下，会继续走shiro其余过滤器和origin过滤器  **protected** **boolean** onLoginSuccess(AuthenticationToken token, Subject subject,  ServletRequest request, ServletResponse response) **throws** Exception {  **return** **true**;  }  //默认情况下，不会继续走shiro其余过滤器和origin过滤器  **protected** **boolean** onLoginFailure(AuthenticationToken token, AuthenticationException e,  ServletRequest request, ServletResponse response) {  **return** **false**;  } |

#### FormAuthenticationFilter

|  |
| --- |
| **public** **class** FormAuthenticationFilter **extends** AuthenticatingFilter {  **public** **static** **final** String ***DEFAULT\_ERROR\_KEY\_ATTRIBUTE\_NAME*** = "shiroLoginFailure";  **public** **static** **final** String ***DEFAULT\_USERNAME\_PARAM*** = "username";  **public** **static** **final** String ***DEFAULT\_PASSWORD\_PARAM*** = "password";  **public** **static** **final** String ***DEFAULT\_REMEMBER\_ME\_PARAM*** = "rememberMe";  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(FormAuthenticationFilter.**class**);  **private** String usernameParam = ***DEFAULT\_USERNAME\_PARAM***;  **private** String passwordParam = ***DEFAULT\_PASSWORD\_PARAM***;  **private** String rememberMeParam = ***DEFAULT\_REMEMBER\_ME\_PARAM***;  **private** String failureKeyAttribute = ***DEFAULT\_ERROR\_KEY\_ATTRIBUTE\_NAME***;  //归属前置增强，当不允许通过当前过滤器时  **protected** **boolean** onAccessDenied(ServletRequest request, ServletResponse response) **throws** Exception {  **if** (isLoginRequest(request, response)) {**//是登陆请求uri**  **if** (isLoginSubmission(request, response)) {**//是登陆提交**  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Login submission detected. Attempting to execute login.");  }  **return** executeLogin(request, response);//该方法在AuthenticatingFilter中  } **else** {**//是登陆页面---会继续执行shiro的其他过滤器和原生过滤器**  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Login page view.");  }  //allow them to see the login page ;)  **return** **true**; //允许通过  }  } **else** {**//不是登录请求---重定向后续shiro的过滤器不会再执行（因为返回false），但会执行原生过滤器链后续的过滤器。**  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Attempting to access a path which requires authentication. Forwarding to the " +  "Authentication url [" + getLoginUrl() + "]");  }  saveRequestAndRedirectToLogin(request, response);//保存当前请求，待登录成功后跳转  **return** **false**; //不允许通过  }  }  **//返回false，则不能通过当前过滤器(这将决定后续过滤器不再执行)，请注意和**AuthenticatingFilter的此方法**比较**  **protected** **boolean** onLoginSuccess(AuthenticationToken token, Subject subject,  ServletRequest request, ServletResponse response) **throws** Exception {  issueSuccessRedirect(request, response);  //we handled the success redirect directly, prevent the chain from continuing:  **return** **false**;  }  **//返回true，则能通过当前过滤器(这将决定后续过滤器依旧执行)，为什么，我认为应该返回false，这里返回true有什么特别意义吗？….**  **//请和**AuthenticatingFilter的此方法**比较**  **protected** **boolean** onLoginFailure(AuthenticationToken token, AuthenticationException e,  ServletRequest request, ServletResponse response) {  setFailureAttribute(request, e);  //login failed, let request continue back to the login page:  **return** **true**;  } |

#### AuthorizationFilter

|  |
| --- |
| **public** **abstract** **class** AuthorizationFilter **extends** AccessControlFilter {  **private** String unauthorizedUrl;    **public** String getUnauthorizedUrl() {  **return** unauthorizedUrl;  }    **public** **void** setUnauthorizedUrl(String unauthorizedUrl) {  **this**.unauthorizedUrl = unauthorizedUrl;  }  //总是返回false  **protected** **boolean** onAccessDenied(ServletRequest request, ServletResponse response) **throws** IOException {  Subject subject = getSubject(request, response);  // If the subject isn't identified, redirect to login URL  **if** (subject.getPrincipal() == **null**) {//未进行身份验证，则保存当前请求，和redirect 到login页面请求  saveRequestAndRedirectToLogin(request, response);  } **else** {//已经进行身份验证  // If subject is known but not authorized, redirect to the unauthorized URL if there is one  // If no unauthorized URL is specified, just return an unauthorized HTTP status code  String unauthorizedUrl = getUnauthorizedUrl();  //SHIRO-142 - ensure that redirect \_or\_ error code occurs - both cannot happen due to response commit:  **if** (StringUtils.*hasText*(unauthorizedUrl)) {//若配置了未授权页，则…  WebUtils.*issueRedirect*(request, response, unauthorizedUrl);  } **else** {//若未配置未授权页，则…  WebUtils.*toHttp*(response).sendError(HttpServletResponse.***SC\_UNAUTHORIZED***);  }  }  **return** **false**;  } |

#### PermissionsAuthorizationFilter

|  |
| --- |
| **public** **class** PermissionsAuthorizationFilter **extends** AuthorizationFilter {  **public** **boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) **throws** IOException {  Subject subject = getSubject(request, response);  String[] perms = (String[]) mappedValue;  //默认允许通过  **boolean** isPermitted = **true**;    //当前请求uri有权限配置，则当前subject必须有权限，否则不允许访问  **if** (perms != **null** && perms.length > 0) {  **if** (perms.length == 1) {  **if** (!subject.isPermitted(perms[0])) {  isPermitted = **false**;  }  } **else** {  **if** (!subject.isPermittedAll(perms)) {  isPermitted = **false**;  }  }  }    **return** isPermitted;  }  } |

#### NamedFilterList

|  |
| --- |
| **public** **interface** NamedFilterList **extends** List<Filter> {    String getName();    FilterChain proxy(FilterChain filterChain);  } |

|  |
| --- |
| **public** **class** SimpleNamedFilterList **implements** NamedFilterList {  **private** String name;  **private** List<Filter> backingList;    **public** SimpleNamedFilterList(String name) {  **this**(name, **new** ArrayList<Filter>());  }    **public** SimpleNamedFilterList(String name, List<Filter> backingList) {  **if** (backingList == **null**) {  **throw** **new** NullPointerException("backingList constructor argument cannot be null.");  }  **this**.backingList = backingList;  setName(name);  }  **public** FilterChain proxy(FilterChain orig) {  **return** **new** ProxiedFilterChain(orig, **this**);  }  //…  } |

#### ProxiedFilterChain

|  |
| --- |
| **public** **class** ProxiedFilterChain **implements** FilterChain {  //**TODO** - complete JavaDoc  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(ProxiedFilterChain.**class**);  **private** FilterChain orig;  **private** List<Filter> filters;  **private** **int** index = 0;  **public** ProxiedFilterChain(FilterChain orig, List<Filter> filters) {  **if** (orig == **null**) {  **throw** **new** NullPointerException("original FilterChain cannot be null.");  }  **this**.orig = orig;  **this**.filters = filters;  **this**.index = 0;  }  **public** **void** doFilter(ServletRequest request, ServletResponse response) **throws** IOException, ServletException {‘  //若shiro权限过滤器没有全部通过，则其余的原生过滤器皆不在走….  //对于shiro的权限过滤器“前置处理”没有成功执行，则其余的shiro过滤器皆不在走  **if** (**this**.filters == **null** || **this**.filters.size() == **this**.index) {  //we've reached the end of the wrapped chain, so invoke the original one:  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Invoking original filter chain.");  }  **this**.orig.doFilter(request, response);  } **else** {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Invoking wrapped filter at index [" + **this**.index + "]");  }  **this**.filters.get(**this**.index++).doFilter(request, response, **this**);  }  }  } |

# 与spring结合

## Web.xml配置

|  |
| --- |
| <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

|  |
| --- |
| **public** **class** DelegatingFilterProxy **extends** GenericFilterBean {  **private** String contextAttribute;  **private** WebApplicationContext webApplicationContext;  **private** String targetBeanName;  **private** **boolean** targetFilterLifecycle = **false**;  **private** **volatile** Filter delegate;  **private** **final** Object delegateMonitor = **new** Object();  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)  **throws** ServletException, IOException {  // Lazily initialize the delegate if necessary.  Filter delegateToUse = **this**.delegate;  **if** (delegateToUse == **null**) {  **synchronized** (**this**.delegateMonitor) {  **if** (**this**.delegate == **null**) {  /**/获取WebApplicationContext**  WebApplicationContext wac = findWebApplicationContext();  **if** (wac == **null**) {  **throw** **new** IllegalStateException("No WebApplicationContext found: no ContextLoaderListener registered?");  }  //  **this**.delegate = initDelegate(wac);  }  delegateToUse = **this**.delegate;  }  }  // Let the delegate perform the actual doFilter operation.  invokeDelegate(delegateToUse, request, response, filterChain);  }    **protected** Filter initDelegate(WebApplicationContext wac) **throws** ServletException {  **Filter delegate = wac.getBean(getTargetBeanName(), Filter.class);**  **if** (isTargetFilterLifecycle()) {  delegate.init(getFilterConfig());//初始化..  }  **return** delegate;  }  **protected** **void** invokeDelegate(  Filter delegate, ServletRequest request, ServletResponse response, FilterChain filterChain)  **throws** ServletException, IOException {  delegate.doFilter(request, response, filterChain);  }  @Override  **protected** **void** initFilterBean() **throws** ServletException {  **synchronized** (**this**.delegateMonitor) {  **if** (**this**.delegate == **null**) {  // If no target bean name specified, use filter name.  **if** (**this**.targetBeanName == **null**) {  ***this.targetBeanName = getFilterName();***  }  // Fetch Spring root application context and initialize the delegate early,  // if possible. If the root application context will be started after this  // filter proxy, we'll have to resort to lazy initialization.  WebApplicationContext wac = findWebApplicationContext();  **if** (wac != **null**) {  **this**.delegate = initDelegate(wac);  }  }  }  } |

|  |
| --- |
| 总结：web.xml中配置的shiro过滤器，是代理过滤器；代理那个过滤器？代理spring容器中beanname=代理过滤器名字的bean。 |

## Spring 配置

|  |
| --- |
| <!—  结合3.1我们知道，真正的过滤器就是它-🡪 ShiroFilterFactoryBean!  他是真正的过滤器吗？no,真正的过滤器是SpringShiroFilter。  而ShiroFilterFactoryBean真正的作用就是配置过滤器必须的信息，例如securtiymanager 等.  -->  <bean id=*"shiroFilter"* class=*"org.apache.shiro.spring.web.ShiroFilterFactoryBean"*>  <property name=*"securityManager"* ref=*"securityManager"* /><!--  <property name="loginUrl" value="${cas.server.url}?service=${cas.project.url}${adminPath}/cas" /> -->  <property name=*"loginUrl"* value=*"${adminPath}/login"* />  <property name=*"successUrl"* value=*"${adminPath}?login"* />  <property name=*"filters"*>  <map>  <entry key=*"cas"* value-ref=*"casFilter"*/>  <entry key=*"authc"* value-ref=*"formAuthenticationFilter"*/>  </map>  </property>  <property name=*"filterChainDefinitions"*>  <ref bean=*"shiroFilterChainDefinitions"*/>  </property>  </bean>  <!—securityManager 配置-->  <!—  defaultSecurityManager 配置默认的sessionManager是ServletContainerSessionManager  ServletContainerSessionManager创建的session 是http session  Jeesite 采用了自定义sessionManager  这里我们采用jeesite，后续再比对他们之间的区别  -->  <bean id=*"securityManager"* class=*"org.apache.shiro.web.mgt.DefaultWebSecurityManager"*>  <property name=*"realm"* ref=*"systemAuthorizingRealm"* />  <property name=*"sessionManager"* ref=*"sessionManager"* />  <property name=*"cacheManager"* ref=*"shiroCacheManager"* />  </bean>  <!—shiro 缓存配置-->  <bean id=*"shiroCacheManager"* class=*"org.apache.shiro.cache.ehcache.EhCacheManager"*>  <property name=*"cacheManager"* ref=*"cacheManager"*/>  </bean>  <!—shiro 注解启用-->  <!-- 保证实现了Shiro内部lifecycle函数的bean执行 -->  <bean id=*"lifecycleBeanPostProcessor"* class=*"org.apache.shiro.spring.LifecycleBeanPostProcessor"*/>  <bean class=*"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"* depends-  on=*"lifecycleBeanPostProcessor"*>  <property name=*"proxyTargetClass"* value=*"true"* />  </bean>  <bean class=*"org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"*>  <property name=*"securityManager"* ref=*"securityManager"*/>  </bean> |

### ShiroFilterFactoryBean

|  |
| --- |
| **public** **class** **ShiroFilterFactoryBean** **implements** FactoryBean, BeanPostProcessor {  **private** SecurityManager securityManager;//安全管理器  **private** Map<String, Filter> filters;//重载或新增过滤器  //url<->过滤器 配置  **private** Map<String, String> filterChainDefinitionMap; //urlPathExpression\_to\_comma-delimited-filter-chain-definition  **private** String loginUrl;//登录url  **private** String successUrl;//登陆成功默认跳转url  **private** String unauthorizedUrl;//未授权url--403  **private** AbstractShiroFilter instance;//真真正正的过滤器  **public** ShiroFilterFactoryBean() {  **this**.filters = **new** LinkedHashMap<String, Filter>();  **this**.filterChainDefinitionMap = **new** LinkedHashMap<String, String>(); //order matters!  }  **public** Object getObject() **throws** Exception {  **if** (instance == **null**) {  instance = createInstance();  }  **return** instance;  }  **protected** AbstractShiroFilter createInstance() **throws** Exception {  ***log***.debug("Creating Shiro Filter instance.");  SecurityManager securityManager = getSecurityManager();  **if** (securityManager == **null**) {  String msg = "SecurityManager property must be set.";  **throw** **new** BeanInitializationException(msg);  }  **if** (!(securityManager **instanceof** WebSecurityManager)) {  String msg = "The security manager does not implement the WebSecurityManager interface.";  **throw** **new** BeanInitializationException(msg);  }  FilterChainManager manager = createFilterChainManager();  //Expose the constructed FilterChainManager by first wrapping it in a  // FilterChainResolver implementation. The AbstractShiroFilter implementations  // do not know about FilterChainManagers - only resolvers:  PathMatchingFilterChainResolver chainResolver = **new** PathMatchingFilterChainResolver();  chainResolver.setFilterChainManager(manager);  //Now create a concrete ShiroFilter instance and apply the acquired SecurityManager and built  //FilterChainResolver. It doesn't matter that the instance is an anonymous inner class  //here - we're just using it because it is a concrete AbstractShiroFilter instance that accepts  //injection of the SecurityManager and FilterChainResolver:  **return** **new** SpringShiroFilter((WebSecurityManager) securityManager, chainResolver);  }  **protected** FilterChainManager createFilterChainManager() {  DefaultFilterChainManager manager = **new** DefaultFilterChainManager();  Map<String, Filter> defaultFilters = manager.getFilters();  //apply global settings if necessary:  **for** (Filter filter : defaultFilters.values()) {  applyGlobalPropertiesIfNecessary(filter);  }  //Apply the acquired and/or configured filters:  Map<String, Filter> filters = getFilters();  **if** (!CollectionUtils.*isEmpty*(filters)) {  **for** (Map.Entry<String, Filter> entry : filters.entrySet()) {  String name = entry.getKey();  Filter filter = entry.getValue();  applyGlobalPropertiesIfNecessary(filter);  **if** (filter **instanceof** Nameable) {  ((Nameable) filter).setName(name);  }  //'init' argument is false, since Spring-configured filters should be initialized  //in Spring (i.e. 'init-method=blah') or implement InitializingBean:  manager.addFilter(name, filter, **false**);  }  }  //build up the chains:  Map<String, String> chains = getFilterChainDefinitionMap();  **if** (!CollectionUtils.*isEmpty*(chains)) {  **for** (Map.Entry<String, String> entry : chains.entrySet()) {  String url = entry.getKey();  String chainDefinition = entry.getValue();  manager.createChain(url, chainDefinition);  }  }  **return** manager;  } |

### SpringShiroFilter

|  |
| --- |
| **public** **class** ShiroFilterFactoryBean **implements** FactoryBean, BeanPostProcessor {  **private** **static** **final** **class** SpringShiroFilter **extends** AbstractShiroFilter {  **protected** SpringShiroFilter(WebSecurityManager webSecurityManager, FilterChainResolver resolver) {  **super**();  **if** (webSecurityManager == **null**) {  **throw** **new** IllegalArgumentException("WebSecurityManager property cannot be null.");  }  setSecurityManager(webSecurityManager);  **if** (resolver != **null**) {  setFilterChainResolver(resolver);  }  }  }  } |

### DefaultFilterChainManager

|  |
| --- |
| **public** **class** DefaultFilterChainManager **implements** FilterChainManager {  **private** **static** **transient** **final** Logger ***log*** = LoggerFactory.*getLogger*(DefaultFilterChainManager.**class**);  **private** FilterConfig filterConfig;  **private** Map<String, Filter> filters; //pool of filters available for creating chains  //key是url，value是此url对应的过滤器（可能是多个，NamedFilterList是对多个过滤器的封装）。  **private** Map<String, NamedFilterList> filterChains; //key: chain name, value: chain  //默认使用此构造方法  **public** DefaultFilterChainManager() {  **this**.filters = **new** LinkedHashMap<String, Filter>();  **this**.filterChains = **new** LinkedHashMap<String, NamedFilterList>();  addDefaultFilters(**false**);  }  **public** DefaultFilterChainManager(FilterConfig filterConfig) {  **this**.filters = **new** LinkedHashMap<String, Filter>();  **this**.filterChains = **new** LinkedHashMap<String, NamedFilterList>();  setFilterConfig(filterConfig);  addDefaultFilters(**true**);  }  //默认调用此 init=false  **protected** **void** addDefaultFilters(**boolean** init) {  **for** (DefaultFilter defaultFilter : DefaultFilter.*values*()) {  addFilter(defaultFilter.name(), defaultFilter.newInstance(), init, **false**);  }  }  //默认调用此 init=false  **public** **void** addFilter(String name, Filter filter, **boolean** init) {  addFilter(name, filter, init, **true**);  }  **protected** **void** addFilter(String name, Filter filter, **boolean** init, **boolean** overwrite) {  Filter existing = getFilter(name);  **if** (existing == **null** || overwrite) {//不存在或者需要重写  **if** (filter **instanceof** Nameable) {  ((Nameable) filter).setName(name);  }  **if** (init) {  initFilter(filter);  }  **this**.filters.put(name, filter);  }  }  //chainName 即配置的url；chainDefinition即配置的过滤器  **public** **void** createChain(String chainName, String chainDefinition) {  **if** (!StringUtils.*hasText*(chainName)) {  **throw** **new** NullPointerException("chainName cannot be null or empty.");  }  **if** (!StringUtils.*hasText*(chainDefinition)) {  **throw** **new** NullPointerException("chainDefinition cannot be null or empty.");  }  **if** (***log***.isDebugEnabled()) {  ***log***.debug("Creating chain [" + chainName + "] from String definition [" + chainDefinition + "]");  }  //parse the value by tokenizing it to get the resulting filter-specific config entries  //  //e.g. for a value of  //  // "authc, roles[admin,user], perms[file:edit]"  //  // the resulting token array would equal  //  // { "authc", "roles[admin,user]", "perms[file:edit]" }  //  String[] filterTokens = splitChainDefinition(chainDefinition);  //each token is specific to each filter.  //strip the name and extract any filter-specific config between brackets [ ]  **for** (String token : filterTokens) {  String[] nameConfigPair = toNameConfigPair(token);  //now we have the filter name, path and (possibly null) path-specific config. Let's apply them:  addToChain(chainName, nameConfigPair[0], nameConfigPair[1]);  }  }  //例如"foo, bar[baz], blah[x, y]"分割的结果是“foo”, “bar[baz]”, “blah[x, y]”  **protected** String[] splitChainDefinition(String chainDefinition) {  **return** StringUtils.*split*(chainDefinition, StringUtils.***DEFAULT\_DELIMITER\_CHAR***, '[', ']', **true**, **true**);  }  例如：  //"foo" 将被分解为String[]{"foo",null}  //"bar[baz]"将被分解为String[]{"bar","baz"}  //"blah[x, y]"将被分解为String[]{"blah","x,y"}  **protected** String[] toNameConfigPair(String token) **throws** ConfigurationException {  **try** {  String[] pair = token.split("\\[", 2);  String name = StringUtils.*clean*(pair[0]);  **if** (name == **null**) {  **throw** **new** IllegalArgumentException("Filter name not found for filter chain definition token: " + token);  }  String config = **null**;  **if** (pair.length == 2) {  config = StringUtils.*clean*(pair[1]);  //if there was an open bracket, it assumed there is a closing bracket, so strip it too:  config = config.substring(0, config.length() - 1);  config = StringUtils.*clean*(config);  //backwards compatibility prior to implementing SHIRO-205:  //prior to SHIRO-205 being implemented, it was common for end-users to quote the config inside brackets  //if that config required commas. We need to strip those quotes to get to the interior quoted definition  //to ensure any existing quoted definitions still function for end users:  **if** (config != **null** && config.startsWith("\"") && config.endsWith("\"")) {  String stripped = config.substring(1, config.length() - 1);  stripped = StringUtils.*clean*(stripped);  //if the stripped value does not have any internal quotes, we can assume that the entire config was  //quoted and we can use the stripped value.  **if** (stripped != **null** && stripped.indexOf('"') == -1) {  config = stripped;  }  //else:  //the remaining config does have internal quotes, so we need to assume that each comma delimited  //pair might be quoted, in which case we need the leading and trailing quotes that we stripped  //So we ignore the stripped value.  }  }    **return** **new** String[]{name, config};  } **catch** (Exception e) {  String msg = "Unable to parse filter chain definition token: " + token;  **throw** **new** ConfigurationException(msg, e);  }  }  **public** **void** addToChain(String chainName, String filterName, String chainSpecificFilterConfig) {  **if** (!StringUtils.*hasText*(chainName)) {  **throw** **new** IllegalArgumentException("chainName cannot be null or empty.");  }  Filter filter = getFilter(filterName);  **if** (filter == **null**) {  **throw** **new** IllegalArgumentException("There is no filter with name '" + filterName +  "' to apply to chain [" + chainName + "] in the pool of available Filters. Ensure a " +  "filter with that name/path has first been registered with the addFilter method(s).");  }  applyChainConfig(chainName, filter, chainSpecificFilterConfig);  NamedFilterList chain = ensureChain(chainName);  chain.add(filter);  }  **protected** **void** applyChainConfig(String chainName, Filter filter, String chainSpecificFilterConfig) {  **if** (***log***.isDebugEnabled()) {  ***log***.debug("Attempting to apply path [" + chainName + "] to filter [" + filter + "] " +  "with config [" + chainSpecificFilterConfig + "]");  }  **if** (filter **instanceof** PathConfigProcessor) {  ((PathConfigProcessor) filter).processPathConfig(chainName, chainSpecificFilterConfig);  } **else** {  **if** (StringUtils.*hasText*(chainSpecificFilterConfig)) {  //they specified a filter configuration, but the Filter doesn't implement PathConfigProcessor  //this is an erroneous config:  String msg = "chainSpecificFilterConfig was specified, but the underlying " +  "Filter instance is not an 'instanceof' " +  PathConfigProcessor.**class**.getName() + ". This is required if the filter is to accept " +  "chain-specific configuration.";  **throw** **new** ConfigurationException(msg);  }  }  }  **protected** NamedFilterList ensureChain(String chainName) {  NamedFilterList chain = getChain(chainName);  **if** (chain == **null**) {  chain = **new** SimpleNamedFilterList(chainName);  **this**.filterChains.put(chainName, chain);  }  **return** chain;  }  **public** FilterChain proxy(FilterChain original, String chainName) {  NamedFilterList configured = getChain(chainName);  **if** (configured == **null**) {  String msg = "There is no configured chain under the name/key [" + chainName + "].";  **throw** **new** IllegalArgumentException(msg);  }  **return** configured.proxy(original);  } |

### FilterChainResolver

|  |
| --- |
| **public** **interface** FilterChainResolver {    FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain originalChain);  } |

#### PathMatchingFilterChainResolver

|  |
| --- |
| **public** **class** PathMatchingFilterChainResolver **implements** FilterChainResolver {  **private** **static** **transient** **final** Logger ***log*** = LoggerFactory.*getLogger*(PathMatchingFilterChainResolver.**class**);  **private** FilterChainManager filterChainManager;  **private** PatternMatcher pathMatcher;  //默认，但filterChainManager被重置  **public** PathMatchingFilterChainResolver() {  **this**.pathMatcher = **new** AntPathMatcher();  **this**.filterChainManager = **new** DefaultFilterChainManager();  }  //非常重要  **public** FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain originalChain) {  FilterChainManager filterChainManager = getFilterChainManager();  **if** (!filterChainManager.hasChains()) {  **return** **null**;  }  String requestURI = getPathWithinApplication(request);  //the 'chain names' in this implementation are actually path patterns defined by the user. We just use them  //as the chain name for the FilterChainManager's requirements  **for** (String pathPattern : filterChainManager.getChainNames()) {  // If the path does match, then pass on to the subclass implementation for specific checks:  **if** (pathMatches(pathPattern, requestURI)) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Matched path pattern [" + pathPattern + "] for requestURI [" + requestURI + "]. " +  "Utilizing corresponding filter chain...");  }  **return** filterChainManager.proxy(originalChain, pathPattern);  }  }  **return** **null**;  }  } |

### DefaultFilter

|  |
| --- |
| **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**);  **public** Filter newInstance() {  **return** (Filter) ClassUtils.*newInstance*(**this**.filterClass);  }  } |

# Shiro原理

shiro的过滤器分为调度过滤器和权限集合过滤器。

shiro调度过滤器：AbstractShiroFilter，这个过滤器的作用是管理其它权限过滤器的执行。

shiro权限集合过滤器：是一个具有名字的权限过滤器列表，name是uri,list是配置拦截uri的权限过滤器组合。

1. 当请求被shiro调度过滤器拦截时，调度过滤器会找出匹配此次请求的权限集合过滤器，

权限集合过滤器会生成一个代理过滤器链（proxyFilterChain），代理过滤器链就依次开始执行匹配本次请求的权限过滤器，执行完后开始执行原过滤器链的过滤器。

配置顺序问题

尽量等出过滤器是第一个，然后是认证过滤器，再然后是权限过滤器，这样可以提升一点性能问题。不过你随便配置顺序也不会有问题的。

过滤器执行顺序

|  |
| --- |
|  |

Origin[index]:表示原生过滤器

AbstractShiroFilter:调度过滤器

Shiro[1]+shiro[2]+…:组合过滤器

Shiro[index]:表示shiro权限过滤器

请认真查看

Abstract. doFilterInternal---

ProxiedFilterChain.doFilter

AdviceFilter. doFilterInternal()---

ProxiedFilterChain.doFilter

AdviceFilter. doFilterInternal()

…

说明问题：

访问受保护的资源（即必须登陆才能访问的资源）时，重定向到登陆页面。

假设受保护资源是a.jsp，当没有登陆就访问a.jsp，

根据配置那么会先走认证过滤器FormAuthenticationFilter（按照上述最佳顺序方式配置），此时它会重定向到登陆页面，后续过滤器一个也不会执行（包括shiro的过滤器和原生过滤器）