1. Shiro简介

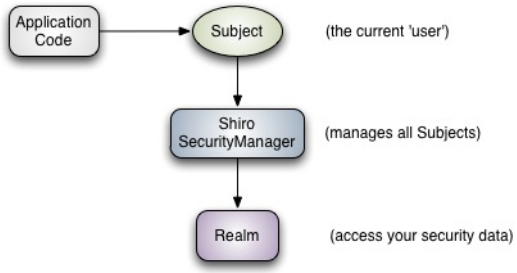
Apache Shiro是Java的一个安全框架。目前，使用Apache Shiro的人越来越多，因为它相当简单，对比Spring Security，可能没有Spring Security做的功能强大，但是在实际工作时可能并不需要那么复杂的东西，所以使用小而简单的Shiro就足够了。对于它俩到底哪个好，这个不必纠结，能更简单的解决项目问题就好了。

Shiro可以的四大基石——身份验证、授权、会话管理和密码:

Shiro不会去维护用户/角色/权限,这些需要我们自己去设计/提供,然后通过Shiro提供的相应的接口注入到Shiro即可。

1.1 shiro基本基本流程

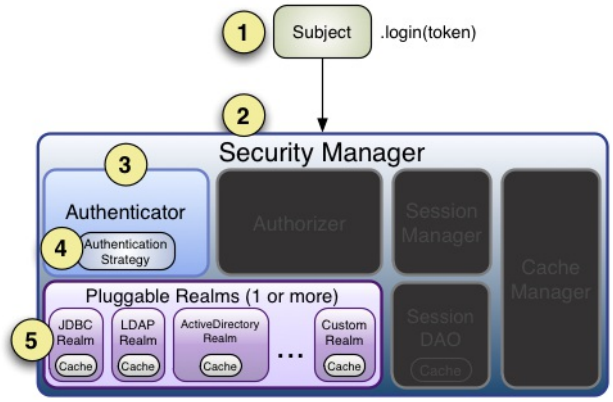
从外部看



也就是说对于我们而言，最简单的一个Shiro应用：

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

从内部看



流程如下：

1. 首先调用Subject.login(token)进行登录，其会自动委托给Security Manager，调用之前必须通过SecurityUtils. setSecurityManager()设置；
2. SecurityManager负责真正的身份验证逻辑；它会委托给Authenticator进行身份验证；
3. Authenticator才是真正的身份验证者，Shiro API中核心的身份认证入口点，此处可以自定义插入自己的实现；
4. Authenticator可能会委托给相应的AuthenticationStrategy进行多Realm身份验证，默认ModularRealmAuthenticator会调用AuthenticationStrategy进行多Realm身份验证；
5. Authenticator会把相应的token传入Realm，从Realm获取身份验证信息，如果没有返回/抛出异常表示身份验证失败了。此处可以配置多个Realm，将按照相应的顺序及策略进行访问。

1.2 shiro特性

1. 接口定义简单明了
2. 身份验证支持多种数据来源(jdbc/LDAP/文件配置等等)，这些来源都是可插拔的。
3. 提供缓存支持可增强应用程序的性能
4. 内置的POJO型企业会话管理，可应用于Web环境，非Web环境，或其任何环境下
5. 更简单的加密接口
6. 是一个非常可靠和低配置的Web框架，能够保护任何url地址或资源，能自动处理登录和注销，执行Remember Me服务，等等
7. 依赖性很低

2. 基于实际应用的shiro使用

2.1 web.xml配置

```
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作用是自动到spring容器查找名字为shiroFilter(filter-name)的bean并把所有Filter的操作委托给它。

ShiroFilter的配置我们可以在spring的配置文件中找到，如下:

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
<!—忽略其他，详见与Spring集成部分 -->
</bean>
```

2.3 登陆表单

使用shiro的第一步，我们需要收集认证信息，也就是进行登陆操作，shiro的登陆操作也是需要提交一个表单。不同的是form表单中提交的action地址不需要对应到任何controller，只要将地址确定后shiro将自动获取表单中的登陆信息进行认证。

login.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>登录</title>
</head>
<body>
<form action="${ctx}/login.jsp" method="post">
    用户名：<input type="text" name="username"><br/>
    密码：<input type="password" name="password"><br/>
    <input type="submit" value="登录">
</form>
</body>
</html>
```

2.2 结合权限系统进行认证与授权

使用shiro的第三步，需要我们定义个realm,realm是shiro中非常重要的一个概念，它完成了认证与授权的主要操作。定义realm需要继承AuthorizingRealm并重写构造方法、doGetAuthenticationInfo（认证）与doGetAuthorizationInfo（授权）方法

- 定义Realm

```
import org.apache.shiro.authc.*;
import org.apache.shiro.authz.*;
import org.apache.shiro.realm.AuthorizingRealm;
import org.apache.shiro.subject.PrincipalCollection;
import org.apache.shiro.util.ByteSource;

import com.sinosoft.one.sdemo.model.account.User;
import com.sinosoft.one.sdemo.service.account.AccountManager;

public class ShiroDbSimpleRealm extends AuthorizingRealm {

    这是一个数据接口，即我们需要通过外部接口，提供一些比如账号密码等查询数据的操作的接口
    @Autowired
    private AccountManager accountManager;


    public ShiroDbSimpleRealm(){
        super();
        super.setAuthenticationTokenClass(UsernamePasswordToken.class);
    }
    /**
     * 认证、认证信息获取
```

```
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(
            AuthenticationToken token) throws AuthenticationException {
        String username = (String)token.getPrincipal();
        String password = (String)token.getCredentials();
         //通过注入的接口对认证信息进行验证,并获取携带权限信息的用户对象实体
        User user = accountManager.findUserByLoginName(username,password);

        if(user == null) {
            throw new UnknownAccountException();//没找到帐号
        }
        //构建一个认证信息SimpleAuthenticationInfo对象并返回，它携带了用户的具体权限数据。
        SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(
                user,//实体
                user.getPassword(), //密码
                ByteSource.Util.bytes(user.getCredentialsSalt()),//salt用来提供更安全的验证
                getName()  //realmname
        );
        return authenticationInfo;
    }
     /*
     * 授权信息获取
     */
    protected AuthorizationInfo doGetAuthorizationInfo(
    PrincipalCollection principals) {
        User user = (User) principals.fromRealm(getName()).iterator().next();
        if (user != null) {
            SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
            info.addRoles(user.getRoleIdList());
            info.addStringPermissions(user.getTaskIdList());
            return info;
        } else {
            return null;
        }
    }

}
```

- 构造方法

在realm的构造方法中，我们可以直接使用父类的构造方法，但是这里需要为realm指定token(令牌)，shiro自身已经提供了直接可用的令牌，可以理解令牌为用户登录信息与shiro认证信息的之间转换的桥梁，登录信息通过封装成token后让shiro从中获取认证所需的凭据。token可以自定义，具体细节请看最后一章。

- AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)

获取权限信息并通过SimpleAuthenticationInfo对象提交给shiro

我们需要做的是：

1、从token中获取表单中提交的认证信息，shiro自动的会将表单中的用户名密码自动的赋值到token中。 2、获取用户信息，可以通过注外部接口获取。 3、将用户信息封装成authenticationInfo对象并返回，其中authenticationInfo需要携带获取的用户信息中的权限数据

- AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)

根据用户身份获取授权信息。

我们需要做的是：

2、通过PrincipalCollection中获取在认证方法里提交的用户实体。 2、使用用户实体中的例如角色、权限等数据构建一个SimpleAuthorizationInfo返回。shiro将可以在后面具体的权限控制中使用SimpleAuthorizationInfo中的角色及权限数据。

总的来说,shiro与权限系统系统的结合是非常简单的，它只要权限系统提供一个获取具体用户信息的接口，并可以通过接口获取封装好的、携带具体权限信息的实体对象即可。

2.4 与spring的整合配置

第四步，我们需要进行shiro的配置。

- Realm配置

将第三节中设置SimpleRealm设置成bean.

```
    <bean id="simpleRealm" class="com.sinosoft.one.sdemo.service.shiro.SimpleRealm" />
```

- SecurityManager配置

```
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager" depends-on="userDao">
    <property name="realm" ref="ShiroDbSimpleRealm" />
</bean>
```

SecurityManager是shiro的信息主体，在securityManager的配置中，引入了simpleRealm。同时又依赖注入了“userDao”，这样做的原因是在2.2节中simpleRealm注入了一个AccountManager接口，而AccountManager如果还注入了其他bean那么需要通过depends-on进行注入，这是需要我们注意的，否则当我们再simpleRealm中的AccountManager时，AccountManager里的其他bean无论是使用@Autowired或者是get方法都无法进行注入。

例如： @Component public class AccountManager {

```
    private UserDao userDao;

}
```

- 过滤器配置

  ```
  <!-- 基于Form表单的身份验证过滤器 -->
  <bean id="formAuthenticationFilter" class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter">
      <property name="usernameParam" value="userCode"/>
      <property name="passwordParam" value="passWord"/>
  </bean>
  ```

  其中username和password分别对应登录表单中username和password。

- ShiroFilter配置

  ```
  <bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
      <property name="securityManager" ref="securityManager"/>
      <property name="loginUrl" value="/login.jsp"/>
      <property name="filters">
          <util:map>
              <entry key="authc" value-ref="formAuthenticationFilter"/>
          </util:map>
      </property>
      <property name="filterChainDefinitions">
          <value>
              /index.jsp = anon
              /login.jsp = authc
              /logout = logout
              /** = user
          </value>
      </property>
  </bean>
  ```

  1. loginUrl：登录的url,该url需要和form表单中所提交的url匹配。
  2. filters：在验证授权时使用的过滤器，entry中的key用来对应的过滤器，它可以有多个，在下面的filterChainDefinitions中使用到。同时shiro提供了许多默认的认证过滤器（具体信息可以参考org.apache.shiro.web.filter.mgt.DefaultFilter），我们也可以自定义过滤器，具体如何自定义将在后面进行描述。
  3. filterChainDefinitions 的匹配原则：filterChainDefinitions的优先级是从上至下,如果获取的请求匹配就会使用“=”号右边的过滤器进行认证。

2.5 权限控制

shiro的权限来源是在**定义Realm**一节中，我们所定义的Realm中的doGetAuthorizationInfo方法，在前面也提到doGetAuthorizationInfo方法中需要把权限信息加入到SimpleAuthorizationInfo中并返回，在进行权限控制时，shiro进行权限判断的依据就是这个SimpleAuthorizationInfo里的数据。

Shiro支持三种权限控制方式，分别为url控制，页面标签控制，编码控制。

- **URL控制**

方式一

```
在ShiroFilter中的filterChainDefinitions中配置，配置方法为

    资源路径 = 过滤器名称
    资源路径 = 过滤器名称1[param1,param2],过滤器名称2[param3,param4]

    例如：/static/js/* = user 这时访问/static/js/下的资源就需要是登录用户才可以
如果过滤器支持更精确的控制，如权限，角色等那么我们可以这样设置

    /service/account/* = roles[role1,role2],表示访问/service/account/下的资源就需要有role1和role2才可以

    /docs/** = perms[document:read]，表示访问/docs/**下的路径需要有权限document:read。
```

具体的URL配置控制可以参照****文档

方式二

自定义一个过滤器，虽然shiro提供了一种最基本的，即在配置文件中定义好url的权限控制的方式。但是通常在我们的权限系统中被访问的URL与对应的权限之间的关系，是有可能根据数据库中的数据改变的，也就是说是在系统运行期间可变的。那么通过配置的方式就达不到我们所需的URL控制，所以可以通过自定一个过滤器实现。

```
public class SimpleFilter extends AuthenticatingFilter {

@Override
protected AuthenticationToken createToken(ServletRequest request,
        ServletResponse response) throws Exception {
    String username = getUsername(request);
    String password = getPassword(request);
    boolean rememberMe = isRememberMe(request);
    String host = getHost(request);
    return createToken(username, password, rememberMe, host);
}

@Override
protected boolean onAccessDenied(ServletRequest request,
        ServletResponse response) throws Exception {
    try {

        if (isLoginRequest(request, response)) {// 是否登录的请求
            System.out.println("请求");
            if (isLoginSubmission(request, response)) {// 是否登陆请求的提交
                System.out.println("提交");
                return executeLogin(request, response);// 拦截的是登陆链接则返回executeLogin
            } else {
                return true;
            }
        } else {
            saveRequestAndRedirectToLogin(request, response);
            return false;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}


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;
}

@Override
protected boolean isAccessAllowed(ServletRequest request,
        ServletResponse response, Object mappedValue) {
    Subject subject = getSubject(request, response);
    if (subject.isAuthenticated()
            || (!isLoginRequest(request, response) && isPermissive(mappedValue))) {
        User user = (User) subject.getPrincipals().getPrimaryPrincipal();
        for (String url : user.getUrls()) {
            String requesturi = WebUtils
                    .getRequestUri((HttpServletRequest) request);
            if (requesturi.toString().equals(url)) {
                return true;
            }
        }

    }
    return false;
}

public boolean isLoginSubmission(ServletRequest request,
        ServletResponse response) {
    return (request instanceof HttpServletRequest)
            && WebUtils.toHttp(request).getMethod()
                    .equalsIgnoreCase(POST_METHOD);
}

protected String getUsername(ServletRequest request) {
    return WebUtils.getCleanParam(request, "userCode");
}

protected String getPassword(ServletRequest request) {
```

```
    return WebUtils.getCleanParam(request, "passWord");
}

}
```

自定义的filter听过继承抽象类AuthenticatingFilter实现，需要实现两个抽象方法createToken与onAccessDenied，这两个方法的逻辑参照formAuthenticationFilter实现即可，自定义filter的其他方法逻辑及自定义filter的详细内容可以见最后一章。在此处完成了createToken与onAccessDenied方法后，需要实现通过用户数据控制URL访问需要实现isAccessAllowed方法。

isAccessAllowed方法需要完成以下操作

1、isAccessAllowed方法在父类中已有的逻辑不改变，即：

```
逻辑判断：
subject.isAuthenticated() ||(!isLoginRequest(request, response) && isPermissi(mappedValue))
```

2、通过获取shiro当前的登陆用户subject，subject中将带有我们在认证阶段设置进去的用户实体，具体见2.2节。再从用户实体中将可访问的URL信息取出与当前URL进行对比，isAccessAllowed方法返回true则表示该请求通过，反回false则反之。

自定义filter还需要对登陆成功后进行一些处理onLoginSuccess，也可参照formAuthenticationFilte中的完成即可。自定义filter的详细介绍请看最后一章。

- 页面标签控制

  Shiro提供了比较完整的权限控制标签，下边我们来介绍一下

  引入

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

  使用

  ```
  guest标签：用户没有身份验证时显示相应信息
  <shiro:guest>
      Hi there!  Please <a href="login.jsp">登录</a> | <a href="signup.jsp">注册</a> today!
  </shiro:guest>

  user标签：用户拥有身份验证时显示相应信息
  <shiro:user>
      欢迎 <shiro:principal/> 登录!
  </shiro:user>

  hasPermission标签:拥有对应的权限才能显示标签内的信息
  <shiro:hasPermission name="user:create">
      <shiro:principal/>拥有user:create权限
  </shiro:hasPermission>

  hasRole标签:拥有对应的权限才能显示标签内的信息
  <shiro:hasRole标签 name="role1">
      <shiro:principal/>拥有role1角色
  </shiro:hasRole标签>
  ```

  更详细的信息请参考 http://shiro.apache.org/web.html#Web-taglibrary

- **编码控制**

shiro还可以在java代码中进行权限控制 **方式一** //Subject是shiro中的用户，可以理解为我们的登陆用户shiro通过它来表示。它携带了所有权限信息、sisson信息。 Subject subject = SecurityUtils.getSubject();
if(subject.hasRole("admin")) {
//TODO:有权限
} else {
//TODO:无权限
}
方式二 在代码中使用注解进行权限控制，可以用于类/属性/方法

```
当前用户需拥有制定权限TASKID才可执行
@RequiresPermissions("taskID")
public void createAccount(Account account) {
    //this method will only be invoked by a Subject
    //that is permitted to create an account
    ...
}

@RequiresRoles("roleID")
public void createAccount(Account account) {
    //this method will only be invoked by a Subject
    //that is permitted to create an account
    ...
```

```
}
```

更多标签、注解见**shiro配详解**

3 shiro进阶

3.1 使登陆信息携带更多的认证要素

有时候我们在进行登陆认证时，提交的认证信息不单单是用户名和密码。这时就需要对shiro进行适当的扩展，而shiro也提供了对应的扩展接口供我们使用。接下来将介绍如何扩展shiro的认证方式，已达到我们的目的。

自定义token

首先我们知道shiro的认证信息是通过提交token，并使用token中携带认证信息进行认证。那么我们需要构建一个符合我们要求的token.shiro提供一个AuthenticationToken的接口，通过实现它即可。

```
import org.apache.shiro.authc.AuthenticationToken;

public class LoginToken implements AuthenticationToken{

private static final long serialVersionUID = 1L;

private String userCode;

private String passWord;

private String comCode;

private String sysFlag;

public LoginToken(){
}

public LoginToken(String userCode,String passWord,String comCode,String sysFlag ){
    this.userCode=userCode;
    this.passWord=passWord;
    this.comCode=comCode;
    this.sysFlag=sysFlag;
}

public Object getPrincipal() {
    return getUserCode();
}

public Object getCredentials() {
    return getPassWord();
}

public void clear() {
    this.userCode = null;
    this.passWord = null;
    this.comCode= null;
    this.sysFlag= null;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(getClass().getName());
    sb.append(" - ");
    sb.append(userCode);
    sb.append(" - ");
    sb.append(passWord);
    sb.append(" - ");
    sb.append(comCode);
    sb.append(" - ");
    sb.append(sysFlag);
    return sb.toString();
}

public String getUserCode() {
    return userCode;
}

public void setUserCode(String userCode) {
    this.userCode = userCode;
}
```

```
public String getPassWord() {
    return passWord;
}

public void setPassWord(String passWord) {
    this.passWord = passWord;
}

public String getComCode() {
    return comCode;
}

public void setComCode(String comCode) {
    this.comCode = comCode;
}

public String getSysFlag() {
    return sysFlag;
}

public void setSysFlag(String sysFlag) {
    this.sysFlag = sysFlag;
}

}
```

以上就是一个简单的扩展，在用户名密码的基础上多了一个机构属性。实际意义就是在登陆认证时需要提交机构代码。同时继承AuthenticationToken时需要实现getPrincipal()方法及getCredentials()方法。getPrincipal方法是需要返回认证的主体，例子中使用用户名作为主体。getCredentials方法需要返回的是授权所需的凭据，这里使用的是密码。注意：自定义的token需要在realm的构造方法中引入，详见2.2节**定义Realm**

编写与token对应的表单

构建完自定义的token后，我们所对应的登陆所提交的from表单项也需要进行匹配。与以上的token的属性进行一一对应。

```
<form id="userLogin" method="post">
    <input type="text"  name="userCode" />
    <input type="password" name="passWord"/>
    <input type="text"  name="comcode" />
    <input type="text"  name="sysFlag" />
</form>
```

自定义filter

这里所说的filter并不是在web.xml中所配置的filter，而是shiro对请求进行权限拦截及对请求进行各种操作处理的“处理器”。例如登陆操作、登陆成功后的操作、登陆失败后的操作等。

同样的shiro自身提供了默认的filter供我们选择，也为我们实现个性化提供了接口。自定义的filter通过继承c抽象类AuthenticatingFilter，并重写其中的方法达到我们的需要。

```
public class LoginFilter extends AuthenticatingFilter {

    @Override
    protected AuthenticationToken createToken(ServletRequest request,
        ServletResponse response) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected boolean onAccessDenied(ServletRequest request,
        ServletResponse response) throws Exception {
        // TODO Auto-generated method stub
        return false;
    }
}
```

createToken、onAccessDenied是两个必要的重写方法。 继承之后，我们可以参考FormAuthenticationFilter这个shiro提供的直接可以使用的filter，它许多方法即可直接复制进行使用。包括filter的构造方法，默认属性等。

一下提供一些自定义filter中常用的方法及属性。

定义变量

```
public static final String DEFAULT_ERROR_KEY_ATTRIBUTE_NAME = "登陆失败";
private static final Logger log = LoggerFactory.getLogger(LoginFilter.class);
public static final String DEFAULT_USERNAME_PARAM = "userCode";
public static final String DEFAULT_PASSWORD_PARAM = "passWord";
```

```
private String failureKeyAttribute = DEFAULT_ERROR_KEY_ATTRIBUTE_NAME;
private String userCodeParam = DEFAULT_USERNAME_PARAM;
private String passWordParam = DEFAULT_PASSWORD_PARAM;
private String comCodeParam = DEFAULT_COMCODE_PARAM;
```

构造方法 public LoginFilter() { setLoginUrl(DEFAULT*LOGIN*URL);//父类定义的默认登陆链接'loging.jsp' }

```
public void setLoginUrl(String loginUrl) {
    String previous = getLoginUrl();
    if (previous != null) {
        this.appliedPaths.remove(previous);
    }
    super.setLoginUrl(loginUrl);
    this.appliedPaths.put(getLoginUrl(), null);
}
```

必须重写的构造token的方法

```
public AuthenticationToken createToken(ServletRequest request,
        ServletResponse response) throws Exception {
    String userCode = getUserCode(request);
    String passWord = getPassWord(request);
    String comCode = getComCode(request);
    return new LoginToken(userCode, passWord, comCode,sysFlag);//返回自定义的TOKEN实例
}
```

必须重写的方法,对所有被拦截的URL请求的处理

```
public boolean onAccessDenied(ServletRequest request,
        ServletResponse response) throws Exception  {

        if (isLoginRequest(request, response)) {//是否登录的请求
            if (isLoginSubmission(request, response)) {//是否登陆请求的提交
                return executeLogin(request, response);//拦截的是登陆链接则返回executeLogin
            } else {
            return true;
            }
        } else {
            saveRequestAndRedirectToLogin(request, response);
            return false;
        }
}
```

登陆成功的处理，即登陆成功后的跳转操作。

```
public boolean onLoginSuccess(AuthenticationToken token,
        Subject subject, ServletRequest request, ServletResponse response)
        throws Exception {
    issueSuccessRedirect(request, response, subject);
    // we handled the success redirect directly, prevent the chain from
    // continuing:
    return false ;
}
```

登陆失败的处理，登陆失败后的跳转操作。

```
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;
}
```

对每个URL的过滤处理，一般如果我们需要对每个URL进行一些权限控制，或者其他的操作那么可以通过复写isAccessAllowed方法。可以见

```
@Override
protected boolean isAccessAllowed(ServletRequest request,
        ServletResponse response, Object mappedValue) {
    Subject subject = getSubject(request, response);
    if (subject.isAuthenticated()
            || (!isLoginRequest(request, response) && isPermissive(mappedValue))) {
        User user = (User) subject.getPrincipals().getPrimaryPrincipal();
        for (String url : user.getUrls()) {
            String requesturi = WebUtils
                    .getRequestUri((HttpServletRequest) request);
            if (requesturi.toString().equals(url)) {
```

```
                return true;
            }
        }

    }
    return false;
}
```

必要的get和set方法

```
public String getUserCode(ServletRequest request) {
    return WebUtils.getCleanParam(request, getUserCodeParam());
}

public String getPassWord(ServletRequest request) {
    return WebUtils.getCleanParam(request, getPassWordParam());
}

public String getComCode(ServletRequest request) {
    return WebUtils.getCleanParam(request, getComCodeParam());
}

public String getUserCodeParam() {
    return userCodeParam;
}

public void setUserCodeParam(String userCodeParam) {
    this.userCodeParam = userCodeParam;
}

public String getPassWordParam() {
    return passWordParam;
}

public void setPassWordParam(String passWordParam) {
    this.passWordParam = passWordParam;
}

public String getComCodeParam() {
    return comCodeParam;
}

public void setComCodeParam(String comCodeParam) {
    this.comCodeParam = comCodeParam;
}

public String getFailureKeyAttribute() {
    return failureKeyAttribute;
}

public void setFailureKeyAttribute(String failureKeyAttribute) {
    this.failureKeyAttribute = failureKeyAttribute;
}
```

以上提供的自定义filter的方法可根据需要进行修改。

构建玩自定义的filter后需要设置到shiro的配置里

定义filter成一个spring的bean。

```
<bean id="loginFilter" class="com.sinosoft.one.newRms.client.shiro.LoginFilter"></bean>
```

将bean使用shiroshiro控制中

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <property name="loginUrl" value="/login.jsp"/>
    <property name="successUrl" value="/login"/>
    <property name="filters">
        <map>
            <entry key="loginfilter" value-ref="loginFilter"/>
        </map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            / =anon
            /login/company/** =anon
            /login/checkUser** = anon
            /login/logout = anon
```

```
            /js/** = anon
            /css/** = anon
            /images/** = anon

            /** = loginfilter
        </value>
    </property>
</bean>
```

注意在使用自定token后，需要在Realm的doGetAuthenticationInfo方法里将token替换。

3.2 认证加密

在涉及到密码存储问题上，应该加密/生成密码摘要存储，而不是存储明文密码。我们的实际项目中涉及到密码的地方大多会进行加密处理，这时我们需要重写Shiro的CredentialsMatcher。

我们的密码加密一般使用的是MD5或者SHA-1，下边我们来介绍一下使用它们两种加密方式怎么重写CredentialsMatcher。

Shiro中CredentialsMatcher的实现PasswordMatcher及HashedCredentialsMatcher，一般我们用HashedCredentialsMatcher来做散列加密。

示例：

假定我们保存用户的时候密码的加密方式是这样的：

```
private void entryptPassword(User user) {
    byte[] salt = Digests.generateSalt(8);
    user.setSalt(Encodes.encodeHex(salt));

    byte[] hashPassword = Digests.sha1(user.getPlainPassword().getBytes(), salt, 1024);
    user.setPassword(Encodes.encodeHex(hashPassword));
}
```

我们的CredentialsMatcher就可以这样写

```
/**
 * 继承HashedCredentialsMatcher，并且重写doCredentialsMatch、
 * 因为我们保存的是经过1024次 sha-1 hash后的密码，所以在认证的时候我们需要将我们从请求获取的密码经过1024次 sha-1 hash后做匹配
 * @author gongwt
 */
public class OverwriteHashedCredentialsMatcher extends HashedCredentialsMatcher{
    private String algorithm ;//设置我们使用的加密算法，这里以SHA-1为例
    private Integer interations;//设置我们的迭代次数

    public OverwriteHashedCredentialsMatcher(String algorithm ,Integer interations) {
        this.algorithm = algorithm;
        this.interations = interations;
    }
    @Override
    public boolean doCredentialsMatch(AuthenticationToken token,
            AuthenticationInfo info) {
        super.setHashAlgorithmName(algorithm);
        super.setHashIterations(interations);
        return super.doCredentialsMatch(token, info);
    }
}
```

下边，将我们重写的CredentialsMatch与我们的Realm关联起来，以前面章节的SimpleRealm为例,我们在SimpleRealm中增添一下代码即可。

```
@PostConstruct
public void initCredentialsMatcher() {
    OverwriteHashedCredentialsMatcher matcher = new OverwriteHashedCredentialsMatcher("SHA-1",1024);
    setCredentialsMatcher(matcher);
}
```

附录

<一些名词解释?> <参考文档?>