**声明**

此文档参考自 尚学堂 资料,并对该资料中一些不足之处做了一些调整;此笔记不做商业用途,仅供学习使用,笔者JustryDeng。

Shiro

第一节 简介

Apache Shiro 是 Java 的一个安全（权限）框架。

Shiro 可以非常容易的开发出足够好的应用，其不仅可以用在

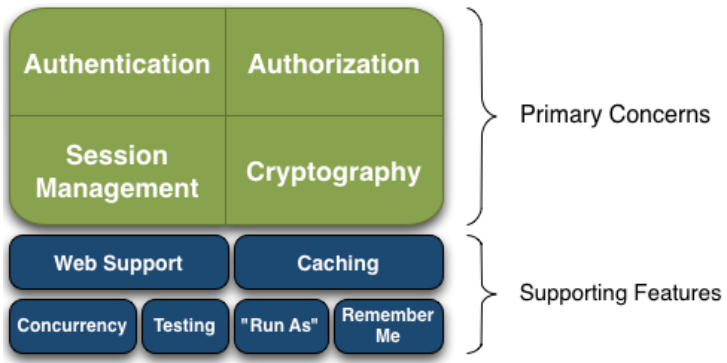
JavaSE 环境，也可以用在 JavaEE环境。

Shiro 可以完成:认证、授权、加密、会话管理、与Web 集成、缓存

等。

下载：

Shiro的基本功能点:



说明:<http://shiro.apache.org/>

Authentication:身份认证(即:登录)，验证用户是都具有相应的身份(如:是否能 登录)。

Authorization:授权(即:权限验证),验证某个已身份认证了的用户，是否具有某 个权限。

Session Manager:会话管理。用户登陆后就是一次会话，在没有退出前，所有信 息都在会话中。

注:会话可以用于普通JavaSE环境，也可用于Web环境。

注:此会话与浏览器的会话不同，此会话是Shiro框架提供的会话。

Cryptography:加密。如:将密码加密后存到数据裤，而非明文存储。

Web Support:Web支持。可以非常容易的集成到Web环境中。

Caching:缓存。如:用户登陆后，缓存用户的信息、角色、权限等，不必每次都 去查询。

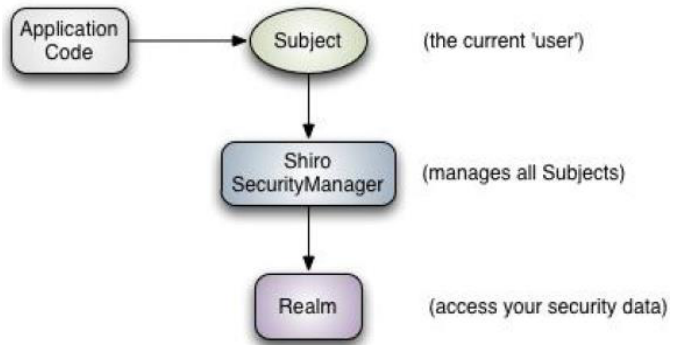
Concurrency:多线程应用的并发验证。如:在一个线程中，开启另一个线程，能 把权限等自动传过去。

Testing:测试支持。

Run As:允许一个用户假装以另一个用户(如果允许的话)的身份进行访问。

Remember Me:记住我。这是非常常见的功能，即一次登录，下次再来的话，不用 登录了。

Shiro的架构（从外部应用程序流程视角来看）:



说明:

Subject:Subject是“登陆者”认证时直接交互的对象(即:Subject是Shiro对 外暴露的API，用户要经过Shrio的验证，就先要走Subject)。由 于Subject只是一个“门面”，所以又可以把Subject看做当前“用 户”，这个用户不一定是一个具体的人，与当前(Shiro“守护”的) 应用交互的任何东西都是Subject,如网络爬虫、机器人等。

注:与Suject的所有交互都会委托给SecurityManager,所以Subject只是一个 门面，SecurityManager才是实际的执行者。

SecurityManager:安全管理器。所有与安全相关的操作，都会与 SecurityManager交互。且其管理着所有的Subject，是 Shrio的核心，它也负责与Shiro其他的组件交互,一定程度 上相当于SpringMVC的DispatcherServlet。

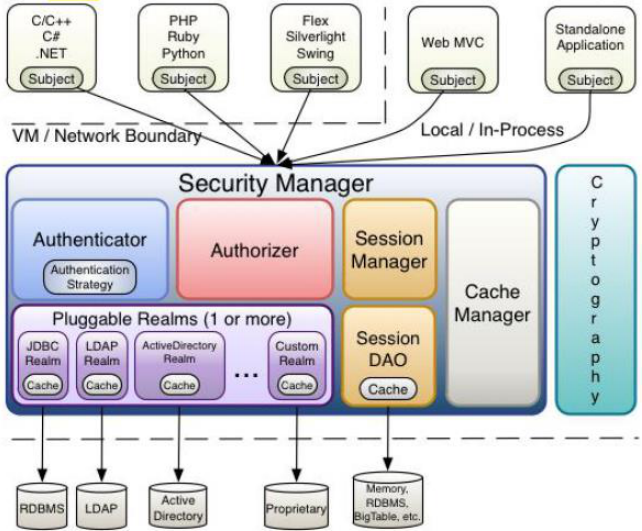
RealM:Shrio从RealM获取到安全数据，如:用户、角色、权限等来进行对应验 证。

如:SecurityManager要验证用户身份，那么它需要先从RealM中获取到相应的 用户数据，然后与用户传过来的数据进行比对，以验证是否合法。也需要从 RealM中得到用户的相应角色、权限，以验证用户是否具有某些权力。

注:RealM可以只有一个，也可以有多个,可以将其认为是安全实体数据源(即:存 放一些安全数据的数据源，如:角色权限等数据)，用于获取安全实体的。

注:RealM可以是JDBC实现，也可以是内存实现等等。由用户提供，所以一般在 应用中都需要实现自己的RealM。

Shiro的架构（从内部模块儿流程视角来看）:



说明:

Subject:可以理解为“用户”（注:前面详细解释了，这里就不再细说了）。提 示:上图中，最上面那一排即为Subject。

SecurityManager:(一定程度上)相当于SpringMVC的DispatcherServlet;是 Shiro的心脏。所有具体的交互都通过SecurityManager进 行控制。它管理着所有的Subject，且负责认证(登录)、授 权(权限)、会话及缓存的管理。

Authenticator:负责Subject用户身份认证(登录)。是一个扩展点，可以自定义 实现。可以使用认证策略，即:什么情况算用户认证通过了。

Authorizer:授权器(又叫:访问控制器),用来决定(已经身份认证了的)用户是否 有某些权限。即:控制用户能访问应用中的哪些功能。

RealM:Shrio从RealM获取到安全数据，如:用户、角色、权限等来进行对应验 证。

如:SecurityManager要验证用户身份，那么它需要先从RealM中获取到相应的 用户数据，然后与用户传过来的数据进行比对，以验证是否合法。也需要从 RealM中得到用户的相应角色、权限，以验证用户是否具有某些权力。

注:RealM可以只有一个，也可以有多个,可以将其认为是安全实体数据源(即:存 放一些安全数据的数据源，如:角色权限等数据)，用于获取安全实体的。

注:RealM可以是JDBC实现，也可以是内存实现等等。由用户提供，所以一般在 应用中都需要实现自己的RealM。

SessionManager:管理Session生命周期的组件。注:shiro不仅可以用于JavaSE 环境，还能用于Web环境。

CacheManeger:缓存控制器。用来管理如用户、角色、权限等的缓存;因为这些数 据基本上很少改变，放到缓存中可以提高访问的性能。

Cryptography:密码模块儿，Shiro提供了一些常见的加密组件用于(如:对用户 登录密码等的)加密解密。

第二节 入门示例之Quickstart

说明:本节简单解释(Shiro提供的快速启动示例文件)Quickstart.java文件，来入门Shiro。

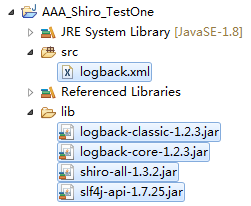
准备:

第一步:创建一个普通Java项目。

2

注:因为只是对一个.java文件的简单说明，所以这里创建一个普通java文件就够用了。

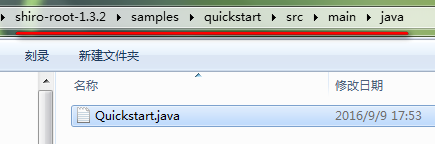
第二步:创建lib文件夹，并引入相关依赖并build path。



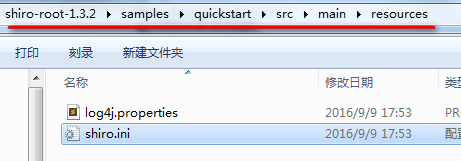
注:shiro-all-1.3.2.jar为主要的shiro依赖包，其余的为日志配置(这里选用slf4j为日志API，logback为slf4j的具体实现)。

第三步:创建package,并将解压后的solr文件夹shiro-root-1.3.2\samples\quickstart\src\main\java中的Quickstart.java和shiro-root-1.3.2\samples\quickstart\src\main\resources中的shiro.ini文件拷贝至package中。

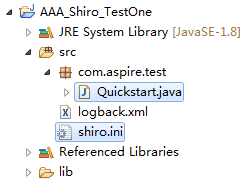
将



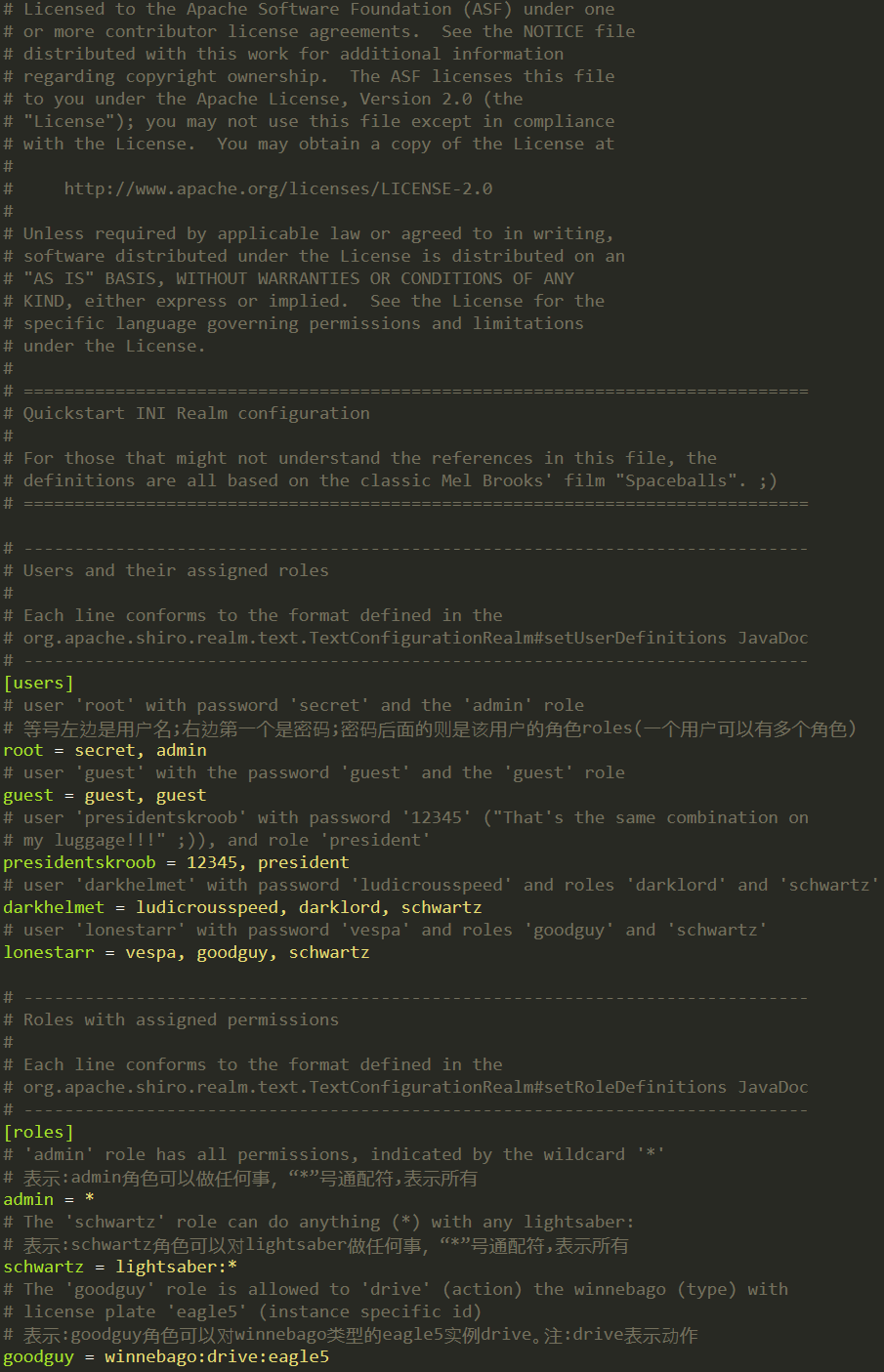
和



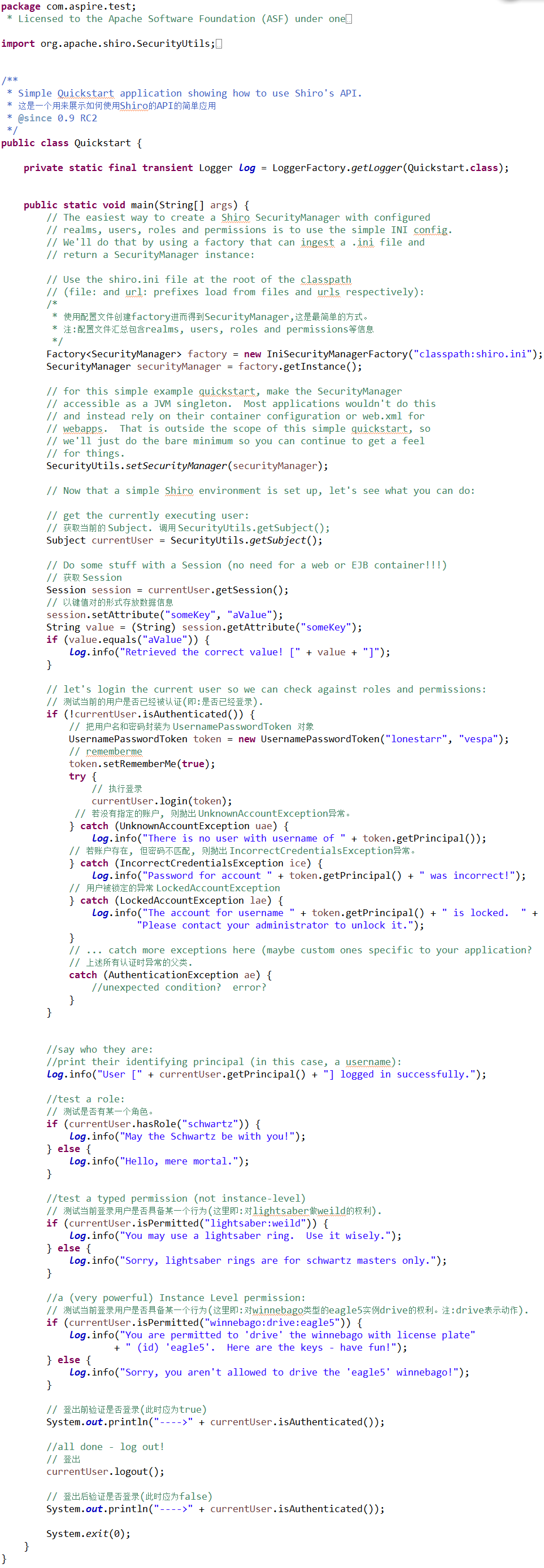
拷贝至:



先给出shiro.ini文件中的内容:



再给出Quickstart.java文件中的内容:



给出文字版:

|  |
| --- |
| **package** com.aspire.test;  **import** org.apache.shiro.SecurityUtils;  **import** org.apache.shiro.authc.\*;  **import** org.apache.shiro.config.IniSecurityManagerFactory;  **import** org.apache.shiro.mgt.SecurityManager;  **import** org.apache.shiro.session.Session;  **import** org.apache.shiro.subject.Subject;  **import** org.apache.shiro.util.Factory;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  /\*\*  \* Simple Quickstart application showing how to use Shiro's API.  \* 这是一个用来展示如何使用Shiro的API的简单应用  \* **@since** 0.9 RC2  \*/  **public** **class** Quickstart {  **private** **static** **final** **transient** Logger ***log*** = LoggerFactory.*getLogger*(Quickstart.**class**);  **public** **static** **void** main(String[] args) {    /\*  \* 使用配置文件创建factory进而得到SecurityManager,这是最简单的方式。  \* 注:配置文件汇总包含realms, users, roles and permissions等信息  \*/  Factory<SecurityManager> factory = **new** IniSecurityManagerFactory("classpath:shiro.ini");  SecurityManager securityManager = factory.getInstance();    SecurityUtils.*setSecurityManager*(securityManager);  // 获取当前的 Subject. 调用 SecurityUtils.getSubject();  Subject currentUser = SecurityUtils.*getSubject*();  // 获取 Session  Session session = currentUser.getSession();  // 以键值对的形式存放数据信息  session.setAttribute("someKey", "aValue");  String value = (String) session.getAttribute("someKey");  **if** (value.equals("aValue")) {  ***log***.info("Retrieved the correct value! [" + value + "]");  }  // 测试当前的用户是否已经被认证(即:是否已经登录).  **if** (!currentUser.isAuthenticated()) {  // 把用户名和密码封装为 UsernamePasswordToken 对象  UsernamePasswordToken token = **new** UsernamePasswordToken("lonestarr", "vespa");  // rememberme  token.setRememberMe(**true**);  **try** {  // 执行登录  currentUser.login(token);  // 若没有指定的账户, 则抛出 UnknownAccountException异常。  } **catch** (UnknownAccountException uae) {  ***log***.info("There is no user with username of " + token.getPrincipal());  // 若账户存在, 但密码不匹配, 则抛出 IncorrectCredentialsException异常。  } **catch** (IncorrectCredentialsException ice) {  ***log***.info("Password for account " + token.getPrincipal() + " was incorrect!");  // 用户被锁定的异常 LockedAccountException  } **catch** (LockedAccountException lae) {  ***log***.info("The account for username " + token.getPrincipal() + " is locked. " +  "Please contact your administrator to unlock it.");  }  // 上述所有认证时异常的父类.  **catch** (AuthenticationException ae) {  //unexpected condition? error?  }  }      ***log***.info("User [" + currentUser.getPrincipal() + "] logged in successfully.");  // 测试是否有某一个角色。  **if** (currentUser.hasRole("schwartz")) {  ***log***.info("May the Schwartz be with you!");  } **else** {  ***log***.info("Hello, mere mortal.");  }  // 测试当前登录用户是否具备某一个行为(这里即:对lightsaber做weild的权利).  **if** (currentUser.isPermitted("lightsaber:weild")) {  ***log***.info("You may use a lightsaber ring. Use it wisely.");  } **else** {  ***log***.info("Sorry, lightsaber rings are for schwartz masters only.");  }  // 测试当前登录用户是否具备某一个行为(这里即:对winnebago类型的eagle5实例drive的权利。注:drive表示动作).  **if** (currentUser.isPermitted("winnebago:drive:eagle5")) {  ***log***.info("You are permitted to 'drive' the winnebago with license plate"  + " (id) 'eagle5'. Here are the keys - have fun!");  } **else** {  ***log***.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");  }  // 登出前验证是否登录(此时应为true)  System.***out***.println("---->" + currentUser.isAuthenticated());    //all done - log out!  // 登出  currentUser.logout();  // 登出后验证是否登录(此时应为false)  System.***out***.println("---->" + currentUser.isAuthenticated());    System.*exit*(0);  }  } |

第三节 Web环境下搭建Shiro

Shiro提供了与Web集成的支持，其通过一个ShiroFilter入口来拦截，需要安全控制的URL,然后进行相应的控制。Shiro类似于SpringMVC、Struts2这种Web框架的前端控制器，是安全控制的入口点其负责读取配置文件(如:.ini配置文件)，然后判断URL是否需要登录/权限等工作。

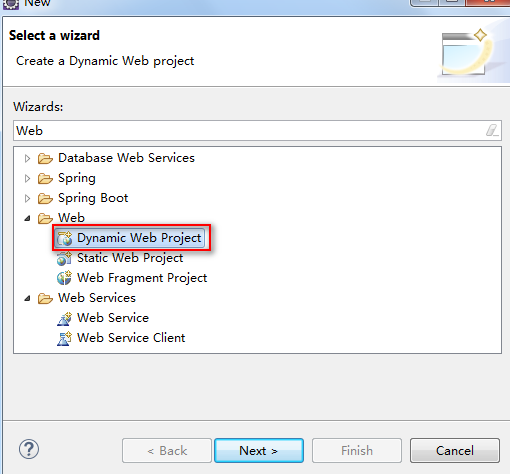
声明:

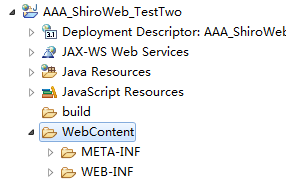
本节先示例一下如何搭建，如果其中个别知识点无法理解的话，可参考下一节的基础知识介绍。

Web环境我们一般都会集成Spring、SpringMVC，所以这里:以集成了Spring、SpringMVC的Web环境再来集成Shiro。

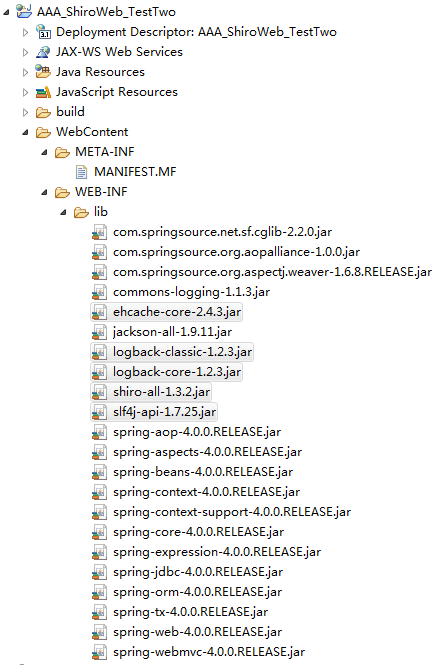
**搭建时,可参考官网下载下来解压后shiro-root-1.3.2\samples\spring示例**。

第一步:创建一个Dynamic Web Project



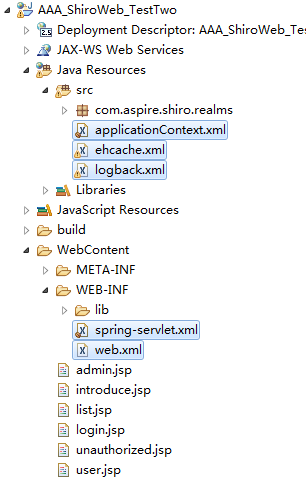


第二步:引入依赖



注:除了引入Spring（含SpringMVC）的依赖外，这里引入了Shiro的依赖、日志的依赖。以及ehcache缓存的依赖。

第三步:配置各种xml文件

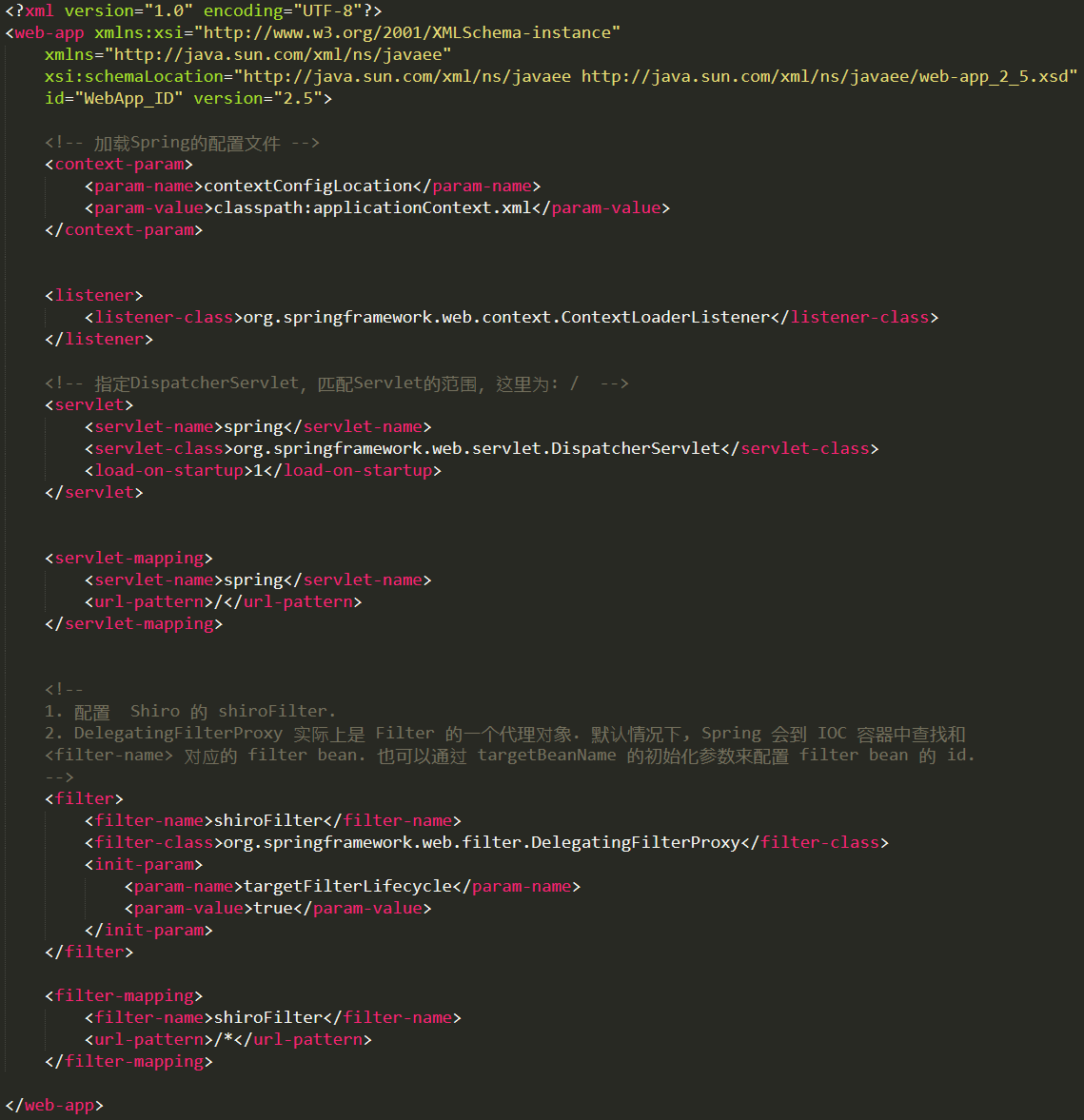


注:这些配置文件从上到下分别是:Spring配置文件、缓存配置文件、日志配置文件、SpringMVC配置文件、WEB配置文件。

注:Shiro的配置，主要配置在Spring配置文件(上图中applicationContext.xml)中,部分在SpringMVC和Web的配置文件中。

给出第三步所涉及的配置文件的内容

web.xml



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"http://java.sun.com/xml/ns/javaee"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*  id=*"WebApp\_ID"* version=*"2.5"*>    <!-- 加载Spring的配置文件 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>    <!-- 指定DispatcherServlet，匹配Servlet的范围，这里为: / -->  <servlet>  <servlet-name>spring</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>spring</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>    <!--  1. 配置 Shiro 的 shiroFilter.  2. DelegatingFilterProxy 实际上是 Filter 的一个代理对象. 默认情况下, Spring 会到 IOC 容器中查找和  <filter-name> 对应的 filter bean. 也可以通过 targetBeanName 的初始化参数来配置 filter bean 的 id.  -->  <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>    </web-app> |

applicationContext.xml



注:此步骤中RealM指向的是自定义的RealM,在第四步的时候给出。

给出文字版

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- ····················· 配置 SecurityManager ····················· -->  <bean id=*"securityManager"*  class=*"org.apache.shiro.web.mgt.DefaultWebSecurityManager"*>  <!-- 缓存 -->  <property name=*"cacheManager"* ref=*"cacheManager"* />  <!-- 认证器 -->  <property name=*"authenticator"* ref=*"authenticator"*></property>  <!-- RealM -->  <property name=*"realms"*>  <!-- 指定使用哪些RealM,注意要对应 -->  <list>  <ref bean=*"jdbcRealm"* />  </list>  </property>  <!-- rememberMeManager匹配 -->  <property name=*"rememberMeManager.cookie.maxAge"* value=*"10"*></property>  </bean>  <!-- 缓存 -->  <!-- 配置 CacheManager. 注:需要加入 ehcache的 jar包及相关配置文件. -->  <bean id=*"cacheManager"*  class=*"org.apache.shiro.cache.ehcache.EhCacheManager"*>  <!-- 指定配置文件 -->  <property name=*"cacheManagerConfigFile"*  value=*"classpath:ehcache.xml"* />  </bean>  <!-- 认证器 -->  <bean id=*"authenticator"*  class=*"org.apache.shiro.authc.pam.ModularRealmAuthenticator"*>  <property name=*"authenticationStrategy"*>  <bean  class=*"org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy"*></bean>  </property>  </bean>  <!-- 配置 Realm 注:直接配置实现了 org.apache.shiro.realm.Realm 接口的 bean -->  <bean id=*"jdbcRealm"* class=*"com.aspire.shiro.realms.ShiroRealm"*>  </bean>    <!-- 配置 LifecycleBeanPostProcessor. 可以自动调用配置在 Spring IOC 容器中 shiro  bean 的生命周期方法. -->  <bean id=*"lifecycleBeanPostProcessor"*  class=*"org.apache.shiro.spring.LifecycleBeanPostProcessor"* />  <!-- 使IOC容器中的bean可以使用 shiro的注解. 注:前提是已经配置了 LifecycleBeanPostProcessor -->  <bean  class=*"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"*  depends-on=*"lifecycleBeanPostProcessor"* />    <bean  class=*"org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"*>  <property name=*"securityManager"* ref=*"securityManager"* />  </bean>  <!-- 配置 ShiroFilter. 注:id必须和 web.xml 文件中配置的org.springframework.web.filter.DelegatingFilterProxy  的 <filter-name> 一致。 -->  <bean id=*"shiroFilter"*  class=*"org.apache.shiro.spring.web.ShiroFilterFactoryBean"*>  <property name=*"securityManager"* ref=*"securityManager"* />  <!-- 当Shiro验证时,如果不满足认证条件,那么打回到这个登录页面 -->  <property name=*"loginUrl"* value=*"/login.jsp"* />  <!-- 当Shiro认证(登陆)成功后,要跳转到的页面 -->  <property name=*"successUrl"* value=*"/list.jsp"* />  <!-- 当Shiro认证(登陆)成功后,如果不满足权限条件,那么跳转至此页面 -->  <property name=*"unauthorizedUrl"* value=*"/unauthorized.jsp"* />  <!-- 配置哪些页面需要受保护. 以及访问这些页面需要的权限. -->  <property name=*"filterChainDefinitions"*>  <value>  # anon表示 无须认证(登录)就可访问  /login.jsp = anon  /introduce.jsp = anon  # 表示 用户认证通过 且 角色含user的用户才可以访问  # 注意角色一定要用双引号引起来  /user.jsp = roles["user"]  # 表示 用户认证通过 且 角色含admin的用户才可以访问  # 注意角色一定要用双引号引起来  /admin.jsp = roles["admin"]  # authc表示需要先认证(登录)  # 注意这个如果有的话,一定要放在最后  /\*\* = authc  </value>  <!--  说明:如果URL匹配时,同时匹配到多个,那么以最上面的那个为准;  例外:如果是非常明确的设定的话,如:上一行是“login.jsp = anon”,  下一行是“login.jsp = authc”那么以下面的那行为准  -->  </property>  </bean>  </beans> |

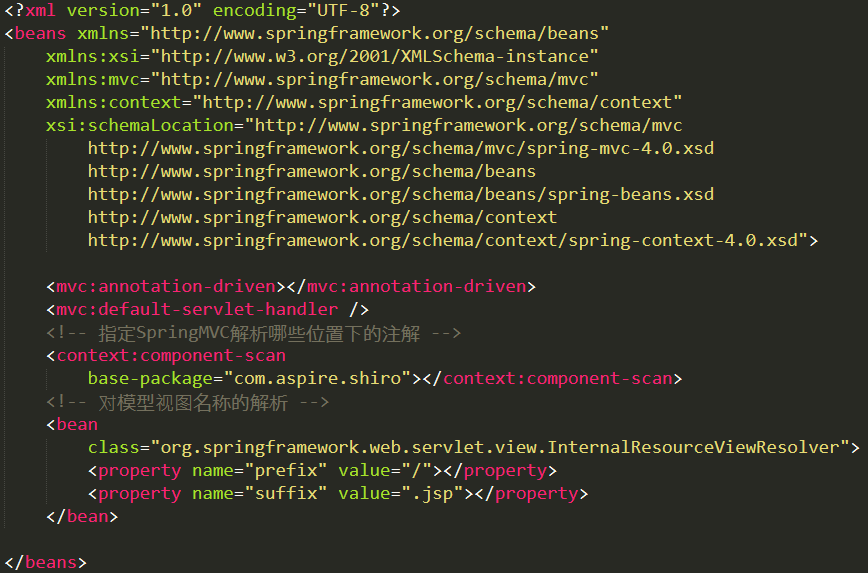
ehcache.xml



给出文字版:

|  |
| --- |
| <ehcache>  <diskStore path=*"java.io.tmpdir"*/>    <cache name=*"authorizationCache"*  eternal=*"false"*  timeToIdleSeconds=*"3600"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  statistics=*"true"*>  </cache>  <cache name=*"authenticationCache"*  eternal=*"false"*  timeToIdleSeconds=*"3600"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  statistics=*"true"*>  </cache>  <cache name=*"shiro-activeSessionCache"*  eternal=*"false"*  timeToIdleSeconds=*"3600"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  statistics=*"true"*>  </cache>    <defaultCache  maxElementsInMemory=*"10000"*  eternal=*"false"*  timeToIdleSeconds=*"120"*  timeToLiveSeconds=*"120"*  overflowToDisk=*"true"*  />  <cache name=*"sampleCache1"*  maxElementsInMemory=*"10000"*  eternal=*"false"*  timeToIdleSeconds=*"300"*  timeToLiveSeconds=*"600"*  overflowToDisk=*"true"*  />  <cache name=*"sampleCache2"*  maxElementsInMemory=*"1000"*  eternal=*"true"*  timeToIdleSeconds=*"0"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  />  </ehcache> |

spring-servlet.xml



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>    <mvc:annotation-driven></mvc:annotation-driven>  <mvc:default-servlet-handler />  <!-- 指定SpringMVC解析哪些位置下的注解 -->  <context:component-scan  base-package=*"com.aspire.shiro"*></context:component-scan>  <!-- 对模型视图名称的解析 -->  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/"*></property>  <property name=*"suffix"* value=*".jsp"*></property>  </bean>  </beans> |

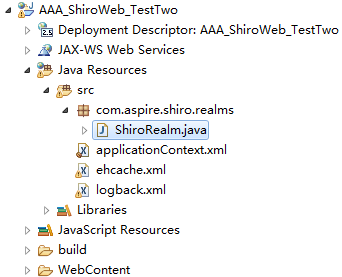
logback.xml



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- scan="true"开启对配置信息的自动扫描(默认时间为60秒扫描一次) 注:当此文件的配置信息发生变化时,此设置的作用就体现出来了,不需要重启服务 -->  <configuration scan=*"true"*>  <!-- 通过property标签,来存放key-value数据,便于后面的动态获取,提高程序的灵活性 -->  <property name=*"logFilePositionDir"*  value=*"C:/Users/dengshuai.ASPIRE/Desktop/log"* />  <!-- >>>>>>>>>>>>>>>>>>>>>>>>>配置appender(可以配置多个)>>>>>>>>>>>>>>>>>>>>>>>>> -->  <!--  name:自取即可,  class:加载指定类(ch.qos.logback.core.ConsoleAppender类会将日志输出到>>>控制台),  patter:指定输出的日志格式  -->  <appender name=*"consoleAppender"*  class=*"ch.qos.logback.core.ConsoleAppender"*>  <encoder>  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}:%L- %msg%n  </pattern>  </encoder>  </appender>  <!--  name:自取即可,  class:加载指定类(ch.qos.logback.core.rolling.RollingFileAppender类会将日志输出到>>>指定的文件中),  patter:指定输出的日志格式 file:指定存放日志的文件(如果无,则自动创建) rollingPolicy:滚动策略>>>每天结束时，都会将该天的日志存为指定的格式的文件  FileNamePattern：文件的全路径名模板 (注:如果最后结尾是gz或者zip等的话,那么会自动打成相应压缩包)  -->  <appender name=*"fileAppender"*  class=*"ch.qos.logback.core.rolling.RollingFileAppender"*>  <!-- 把日志文件输出到:项目启动的目录下的log文件夹(无则自动创建)下 -->  <file>log/logFile.log</file>  <!-- 把日志文件输出到:name为logFilePositionDir的property标签指定的位置下 -->  <!-- <file>${logFilePositionDir}/logFile.log</file> -->  <!-- 把日志文件输出到:当前磁盘下的log文件夹(无则自动创建)下 -->  <!-- <file>/log/logFile.log</file> -->  <rollingPolicy  class=*"ch.qos.logback.core.rolling.TimeBasedRollingPolicy"*>  <!-- TimeBasedRollingPolicy策略会将过时的日志，另存到指定的文件中(无该文件则创建) -->  <!-- 把因为 过时 或 过大 而拆分后的文件也保存到目启动的目录下的log文件夹下 -->  <fileNamePattern>log/logFile.%d{yyyy-MM-dd}.%i.log  </fileNamePattern>  <!-- 设置过时时间(单位:<fileNamePattern>标签中%d里最小的时间单位) -->  <!-- 系统会删除(分离出去了的)过时了的日志文件 -->  <!-- 本人这里:保存以最后一次日志为准,往前7天以内的日志文件 -->  <MaxHistory>  7  </MaxHistory>  <!-- 滚动策略可以嵌套；  这里嵌套了一个SizeAndTimeBasedFNATP策略，  主要目的是: 在每天都会拆分日志的前提下，  当该天的日志大于规定大小时，  也进行拆分并以【%i】进行区分，i从0开始  -->  <timeBasedFileNamingAndTriggeringPolicy  class=*"ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"*>  <maxFileSize>5MB</maxFileSize>  </timeBasedFileNamingAndTriggeringPolicy>  </rollingPolicy>  <encoder>  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}:%L- %msg%n  </pattern>  </encoder>  </appender>  <!-- >>>>>>>>>>>>>>>>>>>>>>>>>>>>>使用appender>>>>>>>>>>>>>>>>>>>>>>>>>>>>> -->  <!-- 指定[哪个包]下使用[哪个appender],并[设定日志级别] -->  <!-- 日志级别: trace|debug|info|warn|error|fatal -->  <logger name=*"com"* level=*"debug"*>  <!-- 指定使用哪个appender -->  <appender-ref ref=*"fileAppender"* />  </logger>  <!-- root:logger的根节点,appender-ref:确定使用哪个appender,将日志信息显示在console -->  <!--  注:如果不指定appender或则指定的是输出到文件的appender的话,  那么SpringBoot启动后,将不会在console打印任何信息  -->  <root level=*"info"*>  <appender-ref ref=*"consoleAppender"* />  </root>  </configuration> |

第四步:编写自己的RealM实现



该类只需实现Realm接口即可。



注:由于本节主要演示Web环境下怎么搭建Shiro,所以方法重写这里就从简处理了。

简单测试一下:

测试步骤:

我们访问http://localhost:8080/AAA\_ShiroWeb\_TestTwo/user.jsp，如何被重定向打回http://localhost:8080/AAA\_ShiroWeb\_TestTwo/login.jsp，则说明Shiro生效了。

测试原理:

通过applicationContext.xml中的配置可知:访问user.jsp首先要验证是否成功认证登陆（注:如果没认证那么重定向至login.jsp）;如果认证(登陆)成功，那么再进一步验证是否有对应的权限，如果没有权限，那么重定向至unauthorized.jsp。

开始测试:

启动tomcat后，如上步骤进行验证，访问:

1

结果重定向回了:

2

注:首次操作会有sessionid。

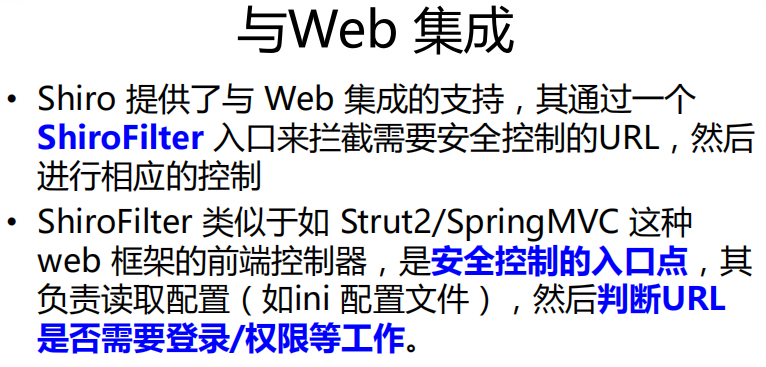
由此可见:Shiro生效了。

第四节 Shiro基础知识

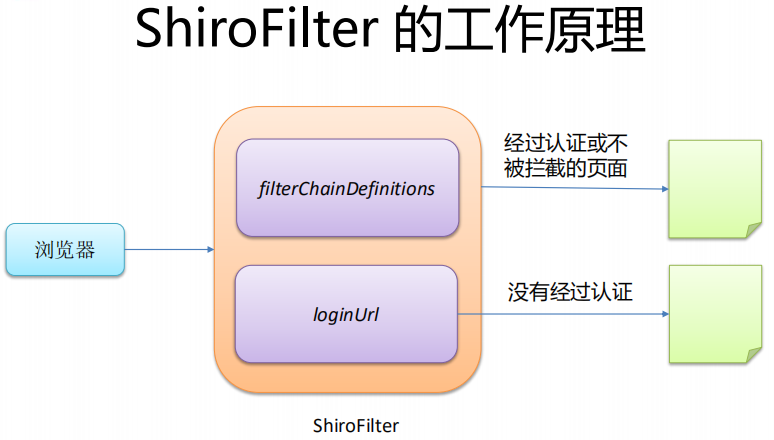
之所以把这一节放在上一节后面，是因为:按照上一节搭建一遍后，再看这一节更容易理解。

声明:本节(甚至本章也是)内容绝大部分是对尚硅谷资料的整理。

与Web的集成

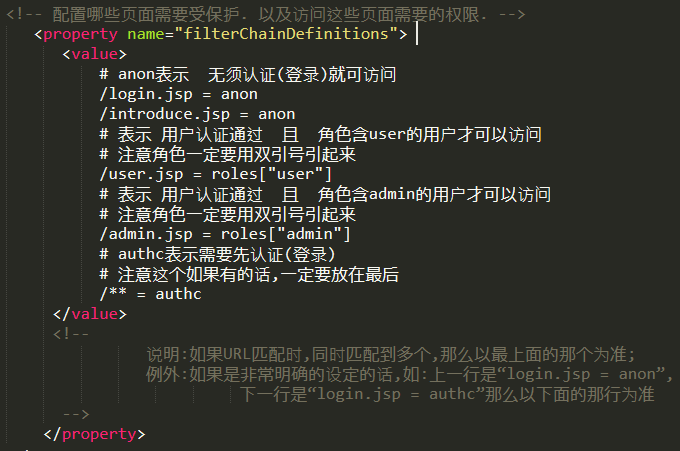


ShiroFilter(更多关于过滤器的知识可参考《程序员成长那个笔记(一)》中的相关章节):





Shiro中过滤配置:



过滤器



注:留意roles[“xxx1,xxx2”]中，角色需要用双引号引起来;哪怕只有一个角色，也需要引起来。

URL匹配模式

提示:URL匹配模式采用Ant风格模式

Ant路径通配符有:? 、\* 、\*\* 。

注:通配符匹配不包括目录分隔符“/”。

? : 匹配一个字符。

示例: /admin?.jsp 将匹配 /admin1.jsp 、 /adminA.jsp 等;但是不匹配 /admin.jsp 或 /admin/

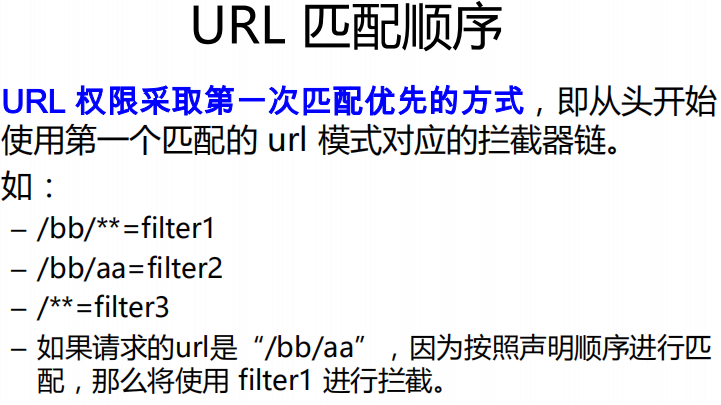
\* : 匹配零个或多个字符串。

示例: /admin\*.jsp 将匹配/admin.jsp 、 /adminA.jsp /adminABC.jsp 等;

\*\* : 匹配该目录下所有的文件(包括其子目录下的所有文件)。

示例: /\*\* 将匹配 /login.jsp 、 /login.jsp 、 /imgs/123.png 等;

URL匹配顺序



注:所以 /\*\* 一般都放在最后。

例外:如果是非常明确的设定的话,如:上一行是“login.jsp = anon”,

下一行是“login.jsp = authc”;那么以下面的那个过滤器为准。

第五节 身份认证流程理论知识

声明:本节末，给出梳理后(最通俗易懂最有用)的身份认证总流程。



身份认证介绍:

身份认证:一般需要提供如身份ID等一些标识信息来标明登陆者的身份。如提供 email,用户名/密码来证明。

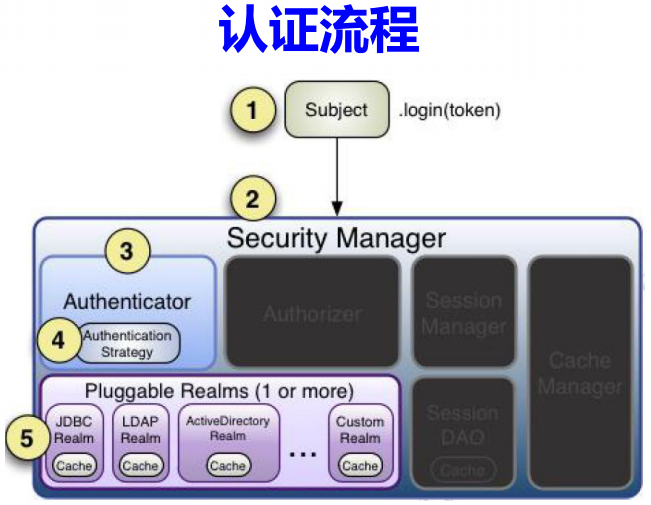
在Shiro中，用户需要提供principals(身份)和credentials(证明)给Shiro， 从而应用能验证用户身份。

Principals:身份，即主体的标识属性，可以是任何属性，如用户名、邮箱等。 一个主题可以有多个principals,但只有一个Primary principals, 一般是用户名/邮箱/手机号。

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

注:最常见的principals和credentials组合就是用户名/密码了。

身份认证认证流程图:



身份认证流程:

第一步:收集用户principals/credentials(即:身份/凭证)，如:用户名/密码。

第二步:调用Subject.login进行登录,如果登录失败，那么将得到相应的 AuthenticationException异常，根据异常提示用户错误信息。

第三步:创建自定义的Realm类，集成

org.apache.shiro.realm.AuthorizingRealm类，实现 doGetAuthenticationInfo()方法。

示例:



对上面的第二步详细介绍:

1:首先调用Subject.login(token)进行登录，其会自动委托给 SecurityManager。

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。

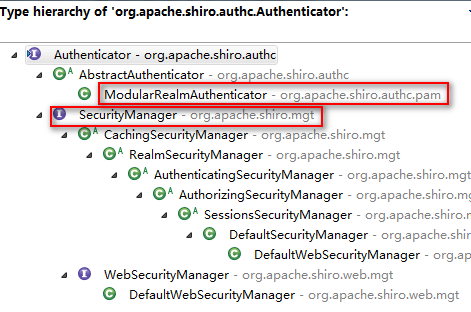
3:Authenticator才是真正的身份验证者，Shiro API中核心的身份认证入口点， 此处可以自定义插入自己的实现。

4:Authenticator有两种方式:一是继承了Authenticator的SecurityManager 接口;二是(间接)实现了Authenticator的ModularRealmAuthenticator类。 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。

注:下面会有专门对Authenticator的介绍。

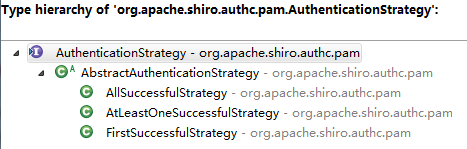
Authenticator认证器:



Authenticator的职责是验证用户账号，是Shiro API中身份验证核心的入口点。如果验证成功，将返回AuthenticationInfo验证信息，此信息中包含了身份及凭证;如果验证失败，则抛出相应的AuthencationException异常。

SecurityManager接口继承了Authenticator;另外Authenticator还有一个ModularRealmAuthenticator实现，ModularRealmAuthenticator委托给多个Realm进行验证，验证规则通过AuthenticationStrategy接口指定。

AuthenticationStrategy认证策略:



AuthenticationStrategy接口的默认实现:

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

FiratSuccessfulStrategy:只要有一个Realm验证成功即可，只返回第一个(成 功验证了)Realm身份的认证信息，其他的忽略。

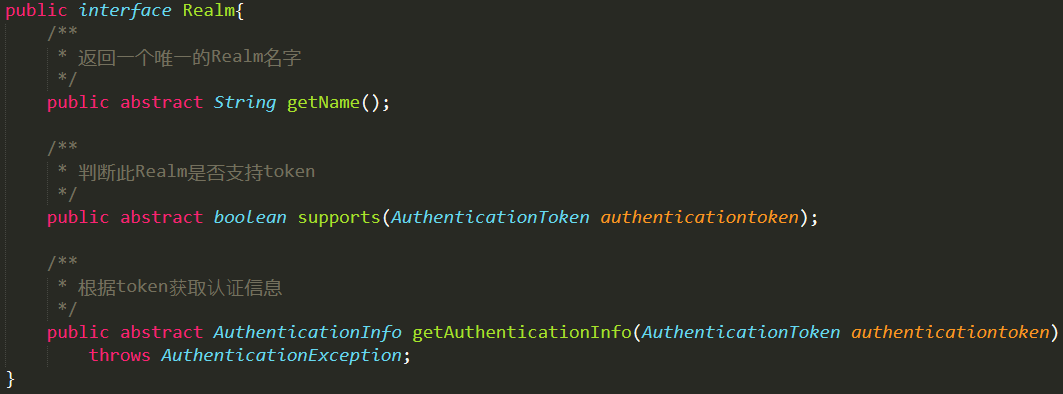
AtLeastOneSuccessfulStrategy:只要有一个Realm验证成功即可，和 FiratSuccessfulStrategy不同，将返回所有(成功验证了)Realm身份验证 成功的认证信息。

注:ModularRealmAuthenticator类默认是采用AtLeastOneSuccessfulStrategy 策略。

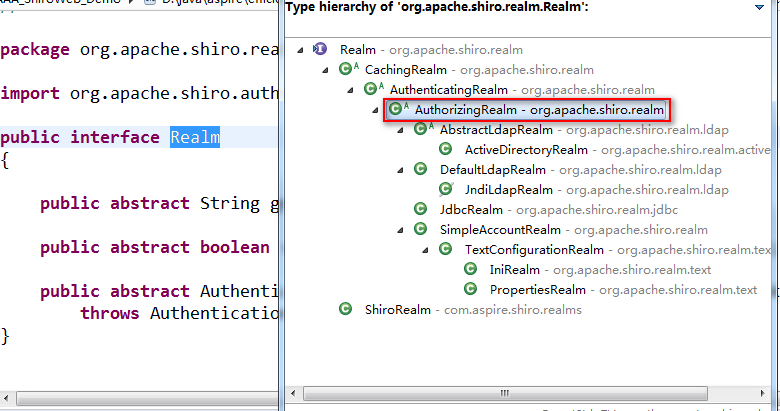
Realm简介:

Shiro从Realm中获取安全数据(如:用户、角色、权限)，即SecurityManager要验证用户身份，那么它需要从Realm获取相应的用户进行比较以确定用户身份是否合法;也需要从Realm得到用户相应的角色/权限进行验证用户是否能进行某一操作。

Realm接口内容如下:



Realm的实现关系如下:



如上图:我们一般不直接实现Realm接口，而是直接继承AuthorizingRealm(授权)。

注:AuthorizingRealm继承了AuthenticatingReals(即身份认证)，而且也继承了CachingReam(带有缓存实现)。

**身份认证总流程:**

第一步:调用 SecurityUtils.getSubject();获取当前的 Subject.

第二步:调用 Subject 的 isAuthenticated() 测试当前的用户是否已经被认 证。即是否已经登录.

第三步:若没有被认证, 则把用户名和密码封装为 UsernamePasswordToken 对 象。

如:

1). 创建一个表单页面

2). 把请求提交到 SpringMVC 的 Handler

3). 获取用户名和密码.

第四步:调用 Subject 的 login(AuthenticationToken) 方法执行登录,参数就传UsernamePasswordToken即可，其是AuthenticationToken接口的实现之一。

第五步:自定义 Realm 的方法

注:

1. . 实际上需要继承 org.apache.shiro.realm.AuthorizingRealm 类

注:其实只是认证的话，我们继承AuthenticatingRealm就可以了，因为后面还需要授权，所以干脆我们这里就直接继承AuthorizingRealm 。

注:AuthorizingRealm类继承了AuthenticatingRealm类。

2). 实现 doGetAuthenticationInfo(AuthenticationToken) 方法.

在该方法里从数据库中获取对应的记录, 返回给 Shiro.

注:此方法的功能是:获取到用户输入的用户名,然后从数据库中查询该用户名，然后根据查询情况，判断用户是否存在、是否合法等;初步验证用户名后，再将用户名（此时数据库中的和用户输入时的用户名都一样）、数据库中查询出来的密码(注:此密码一般在用户注册时就进行了加密存储)、盐值、realmName封装进SimpleAuthenticationInfo对象中返回给Shiro;然后Shiro会对用户登录时的密码(按指定参数)加密，然后比对加密后的密码与 数据库中的密码是否一致。

第六步:由 shiro 完成对密码的比对。

第六节 身份认证流程代码示例

**先给出身份认证总流程:**

第一步:调用 SecurityUtils.getSubject();获取当前的 Subject.

第二步:调用 Subject 的 isAuthenticated() 测试当前的用户是否已经被认 证。即是否已经登录.

第三步:若没有被认证, 则把用户名和密码封装为 UsernamePasswordToken 对 象。

如:

1). 创建一个表单页面

2). 把请求提交到 SpringMVC 的 Handler

3). 获取用户名和密码.

第四步:调用 Subject 的 login(AuthenticationToken) 方法执行登录,参数就传UsernamePasswordToken即可，其是AuthenticationToken接口的实现之一。

第五步:自定义 Realm 的方法

注:

1). 实际上需要继承 org.apache.shiro.realm.AuthorizingRealm 类

注:其实只是认证的话，我们继承AuthenticatingRealm就可以了，因为后面还需要授权，所以干脆我们这里就直接继承AuthorizingRealm 。

注:AuthorizingRealm类继承了AuthenticatingRealm类。

2). 实现 doGetAuthenticationInfo(AuthenticationToken) 方法.

在该方法里从数据库中获取对应的记录, 返回给 Shiro.

注:此方法的功能是:获取到用户输入的用户名,然后从数据库中查询该用户名，然后根据查询情况，判断用户是否存在、是否合法等;初步验证用户名后，再将用户名（此时数据库中的和用户输入时的用户名都一样）、数据库中查询出来的密码(注:此密码一般在用户注册时就进行了加密存储)、盐值、realmName封装进SimpleAuthenticationInfo对象中返回给Shiro;然后Shiro会对用户登录时的密码(按指定参数)加密，然后比对加密后的密码与 数据库中的密码是否一致。

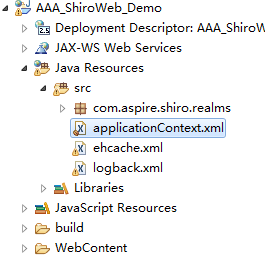
第六步:由 shiro 完成对密码的比对。

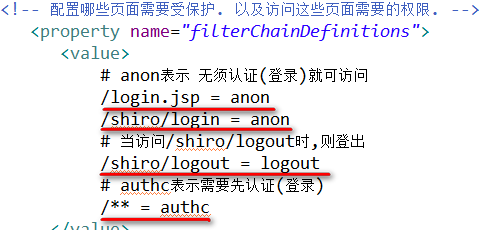
**身份认证总流程代码示例:**

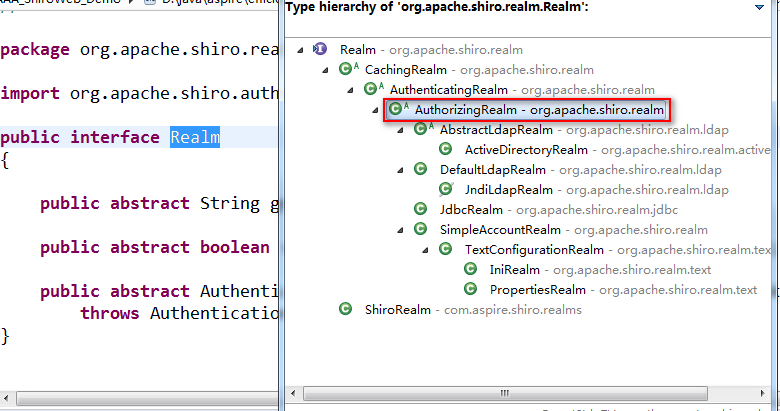
第一步:使用JSP编写一个表单，向后台传入账号密码



第二步:给上一步中的路径*shiro/login*放行，使该路径不需要认证就可访问;在shir配置中(注:shiro配置在applicationContext.xml中)配置:



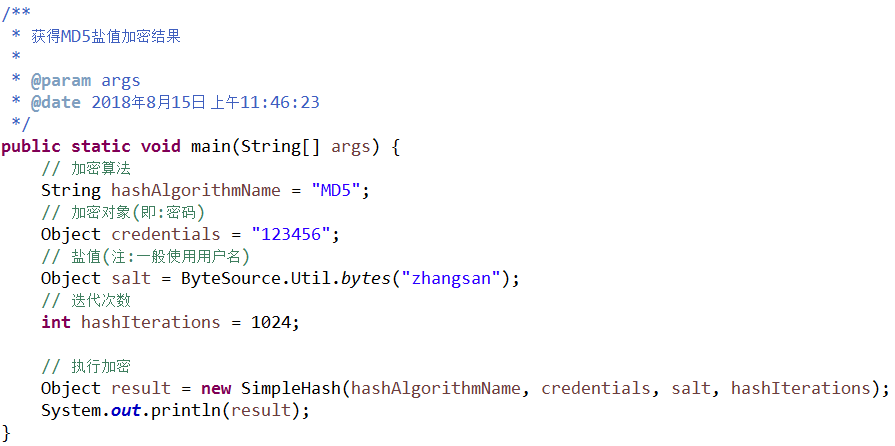


第三步:自定义 Realm 

声明:自定义Realm,可以实现Realm接口或继承AuthebticatingRealm类或继承 AuthorizingRealm类。考虑到下面还要鉴权，我们这里干脆直接继

承AuthorizingRealm类，既能认证又能授权。

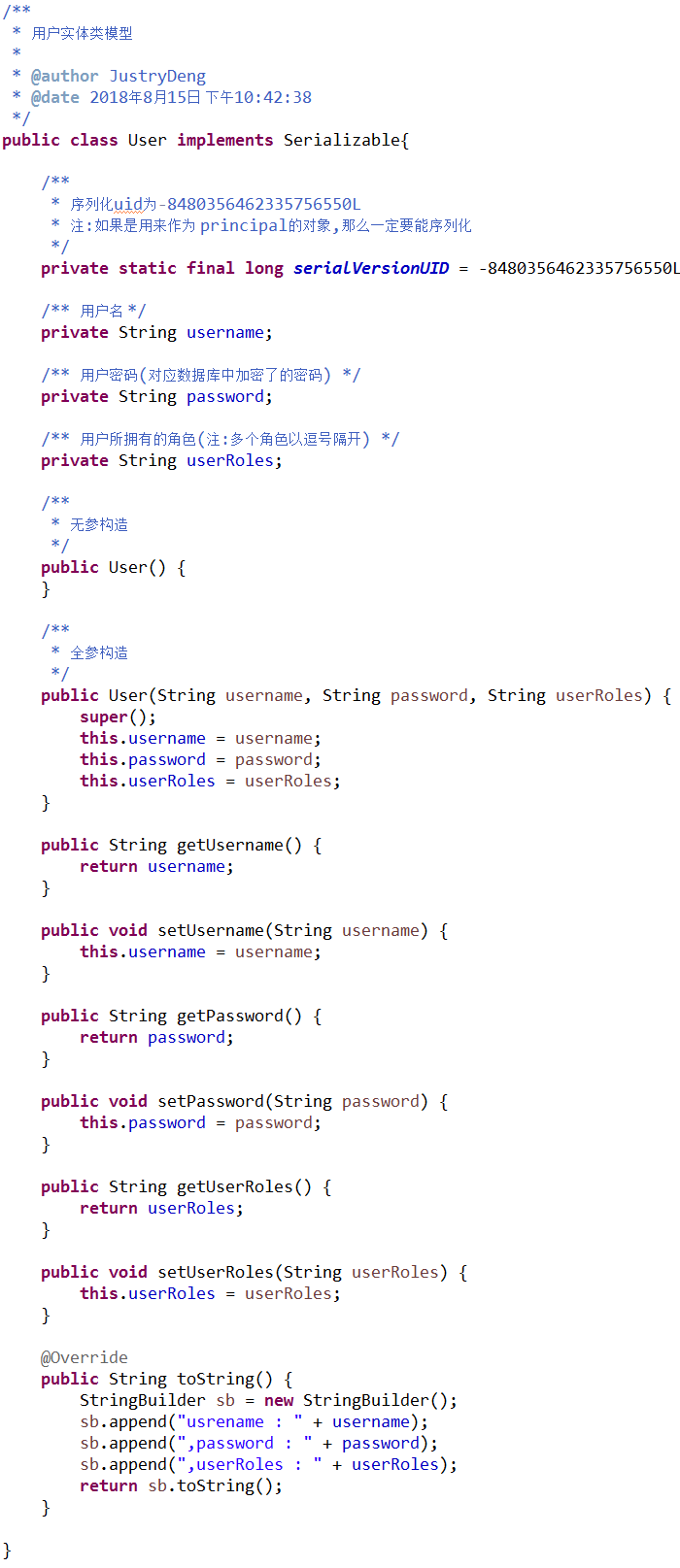
先给出如何给数据进行MD5加密:



给出文字版:

|  |
| --- |
| **public** **static** **void** main(String[] args) {  // 加密算法  String hashAlgorithmName = "MD5";  // 加密对象(即:密码)  Object credentials = "123456";  // 盐值(注:一般使用用户名)  Object salt = ByteSource.Util.*bytes*("zhangsan");  // 迭代次数  **int** hashIterations = 1024;  // 执行加密  Object result = **new** SimpleHash(hashAlgorithmName, credentials, salt, hashIterations);  System.***out***.println(result);  } |

先给出等下要用到的实体类:



自定义 Realm 的方法:

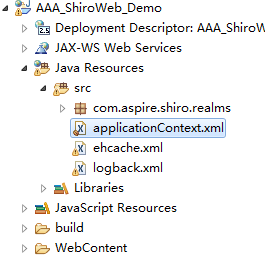


注:上图中的Object credentials = "098d2c478e9c11555ce2823231e02ec1";密码是本人将“123456”使用MD5加密后得到的，这里为了快速测试就直接写出来，就不从数据库查询了。

给出文字版:

|  |
| --- |
| **import** org.apache.shiro.authc.AuthenticationException;  **import** org.apache.shiro.authc.AuthenticationInfo;  **import** org.apache.shiro.authc.AuthenticationToken;  **import** org.apache.shiro.authc.SimpleAuthenticationInfo;  **import** org.apache.shiro.authc.UsernamePasswordToken;  **import** org.apache.shiro.authz.AuthorizationInfo;  **import** org.apache.shiro.realm.AuthorizingRealm;  **import** org.apache.shiro.subject.PrincipalCollection;  **import** org.apache.shiro.util.ByteSource;  **import** com.aspire.shiro.model.User;  /\*\*  \* 编写自己的Realm  \*  \* **@author** JustryDeng  \* **@date** 2018年8月13日 下午2:18:43  \*/  **public** **class** ShiroRealm **extends** AuthorizingRealm {  @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)  **throws** AuthenticationException {  System.***out***.println(".......................................ShiroRealm");  // 1. 把 AuthenticationToken 拆箱转换为 UsernamePasswordToken  UsernamePasswordToken upToken = (UsernamePasswordToken) token;  // 2. 从 UsernamePasswordToken 中来获取 username  String username = upToken.getUsername();  /\*  \* 3. 调用数据库的方法, 从数据库中查询 username对应的用户记录  \* 注:一般的 , 用户名 什么的 最好唯一  \* 4. 若用户不存在, 则可以抛出 UnknownAccountException 异常  \* 5. 根据用户信息的情况, 决定是否需要抛出其他的 AuthenticationException 异常.  \* 6. principal:认证的用户实体信息.(可以为 username、手机号、邮箱等，也可以是一个携带用户信息的对象模型)  \* 注:也可以是数据表对应的用户的实体类对象,在鉴权时可以冲这个对象中回去到其对应有哪些权限.  \* 注:用户对象信息本应该是从数据库中查询出来的,这里为了快速测试，直接new一个  \* 注:用于存放用户信息的模型,必须能后实例化。即:必须实现Serializable接口  \* 注: 这里假设从数据库查出来了某个用户的数据,假设User类的实例principal中的就是查出来的数据  \*/  User principal = **null** ;  **if**(username != **null** && username.equals("zhangsan")) {  principal = **new** User(username, "2a0d136ceacafe198ea64ac09daaf1b6", "admin,user");  }  **if**(username != **null** && username.equals("lisi")) {  principal = **new** User(username, "8c702ae443795331c91cfab48f3f3833", "user");  }      // 6.credentials: 凭证(一般都是密码).  // credentials本应该是查询出来的;这里为了快速测试,我们直接写  Object credentials = principal.getPassword();  // 8. realmName: 当前 realm 对象的 name. 调用父类的 getName() 方法即可  // 这里获取到的是:com.aspire.shiro.realms.ShiroRealm\_0  String realmName = getName();    /\*  \* 9. 盐值.  \* 注:如果多个用户的密码一样，那么一般情况下加密结果也一样;  \* 注:通过使用不同的盐值来确保即便密码都一样,加密结果也会不一样  \* 注:盐值 最好保证其唯一性。  \* 注:由于一般情况下,用户名是唯一的，所以我们一般使用用户名来计算盐值  \*/  ByteSource credentialsSalt = ByteSource.Util.*bytes*(username);  /\*  \* 实例化对象.  \* 注意:如果不加密,那么就是直接比对的明文  \* 注意:SimpleAuthenticationInfo加密,是指:将用户登录时输入的密码(盐值)加密后,与数据库取出来的密码进行比对  \* 所以,这里的加密并不是对从数据库取出来的credentials进行加密!从数据库取出来的credentials应该是  \* 之前录入数据库时已经加密好了的.  \*/  SimpleAuthenticationInfo info = **null**;  info = **new** SimpleAuthenticationInfo(principal, credentials, credentialsSalt, realmName);  **return** info;  }  @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  **return** **null**;  }    } |

配置加密算法:





并在SecurityManager中指定使用Realm:



注:加密是指对当前用户此次登录时输入密码进行对应的加密,然后与数据库中 已经存好了的(之前已经加密好了的)密码进行对比。所以这里配置时，加密 算法类型、迭代次数、盐值等要与给数据库里面的密码加密时的对应参数一 致。

追注:一般在配置文件中配置加密算法类型、迭代次数，在代码里面配置盐 值。

注:如果数据库中存的密码是明文，那么这里就不要设置加密了。

第四步:由 shiro 完成对密码的比对;登录测试。

我们再来看一下applicationContext.xml中Shiro的相关配置:



可知:先进入login.jsp登录>>>登录成功后进入list.jsp页面>>>点击list.jsp页面里面的5标签，进入/shiro/login进行登出。

登录前，访问任何其他需要认证的界面，就会重定向到登录页面;登陆后，访问任何其他需要认证的界面，能进去，说明身份认证通过。

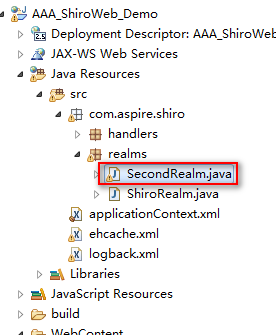
注:一次登录成功后,Shiro就会记住。如果用户此时再进入登录页面，就算输入错误的账号密码也能进。如果想测试，那么需要先登出。

声明:上诉步骤结束后，身份认证流程就走完了。下面再深入介绍一下多Realm的情况。

多Realm的情况:

说明:有时，由于某些原因，我们会把用户账号密码存在多个数据库中或多个表中;这些数据库(或表)中采用的加密算法不一样。此时我们就要对这些数据库(或表)中的数据密码等都进行校验认证。

之前我们已经创建了一个Realm的实现ShiroRealm了,现在我们再创建一个Realm的实现:



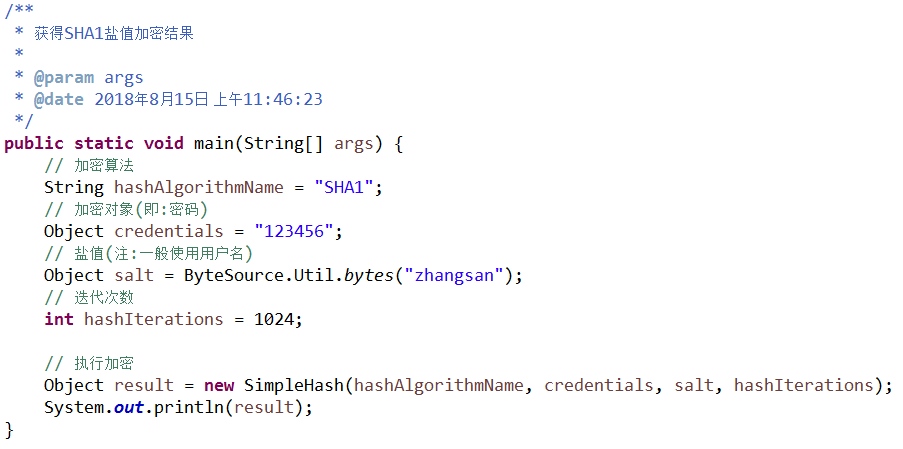
其内部代码和前面的ShiroRealm一样，只是:从数据库取出来的加密后的密码有差异(ShiroRealm中的密码是以MD5加密的，SecondRealm中的密码是SHA1加密):



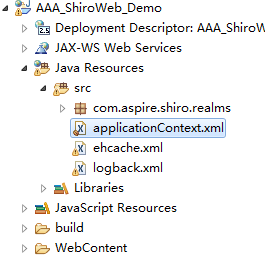
给出文字版:

|  |
| --- |
| **import** org.apache.shiro.authc.AuthenticationException;  **import** org.apache.shiro.authc.AuthenticationInfo;  **import** org.apache.shiro.authc.AuthenticationToken;  **import** org.apache.shiro.authc.SimpleAuthenticationInfo;  **import** org.apache.shiro.authc.UsernamePasswordToken;  **import** org.apache.shiro.authz.AuthorizationInfo;  **import** org.apache.shiro.realm.AuthorizingRealm;  **import** org.apache.shiro.subject.PrincipalCollection;  **import** org.apache.shiro.util.ByteSource;  **import** com.aspire.shiro.model.User;  /\*\*  \* 编写自己的Realm  \*  \* **@author** JustryDeng  \* **@date** 2018年8月13日 下午2:18:43  \*/  **public** **class** SecondRealm **extends** AuthorizingRealm {  @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)  **throws** AuthenticationException {  System.***out***.println(".......................................SecondRealm");  // 1. 把 AuthenticationToken 拆箱转换为 UsernamePasswordToken  UsernamePasswordToken upToken = (UsernamePasswordToken) token;  // 2. 从 UsernamePasswordToken 中来获取 username  String username = upToken.getUsername();  /\*  \* 3. 调用数据库的方法, 从数据库中查询 username对应的用户记录  \* 注:一般的 , 用户名 什么的 最好唯一  \* 4. 若用户不存在, 则可以抛出 UnknownAccountException 异常  \* 5. 根据用户信息的情况, 决定是否需要抛出其他的 AuthenticationException 异常.  \* 6. principal:认证的用户实体信息.(可以为 username、手机号、邮箱等，也可以是一个携带用户信息的对象模型)  \* 注:也可以是数据表对应的用户的实体类对象,在鉴权时可以冲这个对象中回去到其对应有哪些权限.  \* 注:用户对象信息本应该是从数据库中查询出来的,这里为了快速测试，直接new一个  \* 注:用于存放用户信息的模型,必须能后实例化。即:必须实现Serializable接口  \* 注: 这里假设从数据库查出来了某个用户的数据,假设User类的实例principal中的就是查出来的数据  \*/  User principal = **null** ;  **if**(username != **null** && username.equals("zhangsan")) {  principal = **new** User(username, "ecf821d32b47a3eaf1fe58fd588bbd38303becf5", "admin,user");  }  **if**(username != **null** && username.equals("lisi")) {  principal = **new** User(username, "99bd908b05647fdca69c12510c5f1355337264f7", "user");  }      // 6.credentials: 凭证(一般都是密码).  // credentials本应该是查询出来的;这里为了快速测试,我们直接写  Object credentials = principal.getPassword();  // 8. realmName: 当前 realm 对象的 name. 调用父类的 getName() 方法即可  // 这里获取到的是:com.aspire.shiro.realms.ShiroRealm\_0  String realmName = getName();    /\*  \* 9. 盐值.  \* 注:如果多个用户的密码一样，那么一般情况下加密结果也一样;  \* 注:通过使用不同的盐值来确保即便密码都一样,加密结果也会不一样  \* 注:盐值 最好保证其唯一性。  \* 注:由于一般情况下,用户名是唯一的，所以我们一般使用用户名来计算盐值  \*/  ByteSource credentialsSalt = ByteSource.Util.*bytes*(username);  /\*  \* 实例化对象.  \* 注意:如果不加密,那么就是直接比对的明文  \* 注意:SimpleAuthenticationInfo加密,是指:将用户登录时输入的密码(盐值)加密后,与数据库取出来的密码进行比对  \* 所以,这里的加密并不是对从数据库取出来的credentials进行加密!从数据库取出来的credentials应该是  \* 之前录入数据库时已经加密好了的.  \*/  SimpleAuthenticationInfo info = **null**;  info = **new** SimpleAuthenticationInfo(principal, credentials, credentialsSalt, realmName);  **return** info;  }  @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection arg0) {  // 给用户赋予角色(即:鉴权)的逻辑写在这个方法里  **return** **null**;  }  } |

注:对数据进行MD5、SHA1等加密，方式都一样，只需要将hashAlgorithmName改为对应的算法名字即可。



配置加密算法:



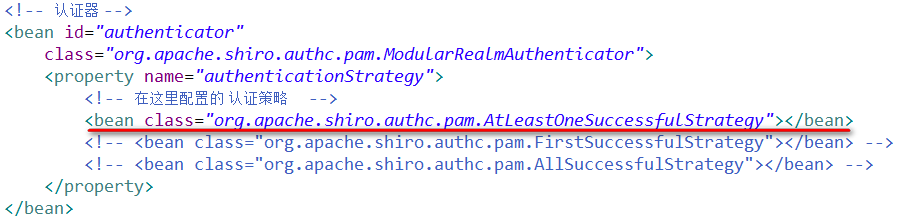


并在SecurityManager中指定使用Realm:



注:多个Realms时，认证身份时，默认是采用AtLeastOneSuccessfulStrategy策略。

多个Realm时，配置身份认证策略:



给出此节结束时，applicationContext.xml中的配置:



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- ····················· 配置 SecurityManager ····················· -->  <bean id=*"securityManager"*  class=*"org.apache.shiro.web.mgt.DefaultWebSecurityManager"*>  <!-- 缓存 -->  <property name=*"cacheManager"* ref=*"cacheManager"* />  <!-- 认证器 -->  <property name=*"authenticator"* ref=*"authenticator"*></property>  <!-- RealM -->  <property name=*"realms"*>  <!-- 指定使用哪些RealM,注意要对应 -->  <list>  <ref bean=*"jdbcRealm"* />  <ref bean=*"secondRealm"* />  </list>  </property>  <!-- rememberMeManager匹配 -->  <property name=*"rememberMeManager.cookie.maxAge"* value=*"10"*></property>  </bean>  <!-- 缓存 -->  <!-- 配置 CacheManager. 注:需要加入 ehcache的 jar包及相关配置文件. -->  <bean id=*"cacheManager"*  class=*"org.apache.shiro.cache.ehcache.EhCacheManager"*>  <!-- 指定配置文件 -->  <property name=*"cacheManagerConfigFile"*  value=*"classpath:ehcache.xml"* />  </bean>  <!-- 认证器 -->  <bean id=*"authenticator"*  class=*"org.apache.shiro.authc.pam.ModularRealmAuthenticator"*>  <property name=*"authenticationStrategy"*>  <!-- 在这里配置的 认证策略 -->  <bean class=*"org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy"*></bean>  <!-- <bean class="org.apache.shiro.authc.pam.FirstSuccessfulStrategy"></bean> -->  <!-- <bean class="org.apache.shiro.authc.pam.AllSuccessfulStrategy"></bean> -->  </property>  </bean>  <!-- 配置 Realm -->    <!-- MD5加密 -->  <bean id=*"jdbcRealm"* class=*"com.aspire.shiro.realms.ShiroRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <!-- 设置加密算法为MD5 -->  <property name=*"hashAlgorithmName"* value=*"MD5"*></property>  <!-- 设置加密迭代次数为1024次 -->  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- SHA1加密 -->  <bean id=*"secondRealm"* class=*"com.aspire.shiro.realms.SecondRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <property name=*"hashAlgorithmName"* value=*"SHA1"*></property>  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- 配置 LifecycleBeanPostProcessor. 可以自动调用配置在 Spring IOC 容器中 shiro  bean 的生命周期方法. -->  <bean id=*"lifecycleBeanPostProcessor"*  class=*"org.apache.shiro.spring.LifecycleBeanPostProcessor"* />  <!-- 使IOC容器中的bean可以使用 shiro的注解. 注:前提是已经配置了 LifecycleBeanPostProcessor -->  <bean  class=*"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"*  depends-on=*"lifecycleBeanPostProcessor"* />    <bean  class=*"org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"*>  <property name=*"securityManager"* ref=*"securityManager"* />  </bean>  <!-- 配置 ShiroFilter. 注:id必须和 web.xml 文件中配置的org.springframework.web.filter.DelegatingFilterProxy  的 <filter-name> 一致。 -->  <bean id=*"shiroFilter"*  class=*"org.apache.shiro.spring.web.ShiroFilterFactoryBean"*>  <property name=*"securityManager"* ref=*"securityManager"* />  <!-- 当Shiro验证时,如果不满足认证条件,那么打回到这个登录页面 -->  <property name=*"loginUrl"* value=*"/login.jsp"* />  <!-- 当Shiro认证(登陆)成功后,要跳转到的页面 -->  <property name=*"successUrl"* value=*"/list.jsp"* />  <!-- 当Shiro认证(登陆)成功后,如果不满足权限条件,那么跳转至此页面 -->  <property name=*"unauthorizedUrl"* value=*"/unauthorized.jsp"* />  <!-- 配置哪些页面需要受保护. 以及访问这些页面需要的权限. -->  <property name=*"filterChainDefinitions"*>  <value>  # anon表示 无须认证(登录)就可访问  /login.jsp = anon  /shiro/login = anon  # 当访问/shiro/logout时,则登出  /shiro/logout = logout  # authc表示需要先认证(登录)  /\*\* = authc  </value>  <!--  说明:如果URL匹配时,同时匹配到多个,那么以最上面的那个为准;  例外:如果是非常明确的设定的话,如:上一行是“login.jsp = anon”,  下一行是“login.jsp = authc”那么以下面的那行为准  -->  </property>  </bean>  </beans> |

第七节 授权流程理论知识

授权基本知识介绍:

授权,也叫访问控制，即在应用中控制谁访问哪些资源(如访问页面/编辑数据/页面操作等)。在授权中需要了解几个关键的对象:主体(Subject)、资源(Resource)、权限(Permission)、角色(Role)。

主体(Subject):访问应用的用户，在Shiro中使用Subject代表该用户。用户只 有授权后才能访问相应的资源。

资源(Resource):在应用中用户可以访问URL，比如访问JSP页面、查看/编辑某 些数据、访问某个资源的权利。即权限表示在应用中用户能不能 访问某个资源，如:访问用户列表页面查看/新增/修改/删除用户 数据(即很多时候都是CRUD(增查改删)式权限控制等)。权限代 表了用户有没有操作某个资源的权利，即反映在某个资源上的操 作允不允许。

Shiro支持粗粒度权限(如用户模块儿的所有权限)和细粒度权限(操作某个用户 的权限，即实力级别)。

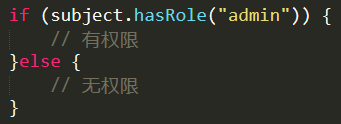
角色(Role):权限的集合，一般情况下会赋予用户角色而不是权限，即这样的用 户可以拥有一组权限，赋予权限时比较方便。典型的如:项目经理、 技术总监、CTO、开发工程师等都是角色，不同的角色拥有一组不同 的权限。

授权方式:

Shiro支持三种方式的授权,分别是:

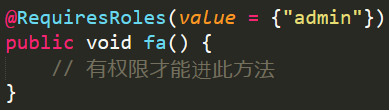
◎编程式:通过写if/else授权代码块儿完成

如:



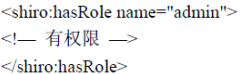
◎注解式:通过在执行的Java方法上防止相应的注解来完成，没有权限则抛出相 应的异常。

如:



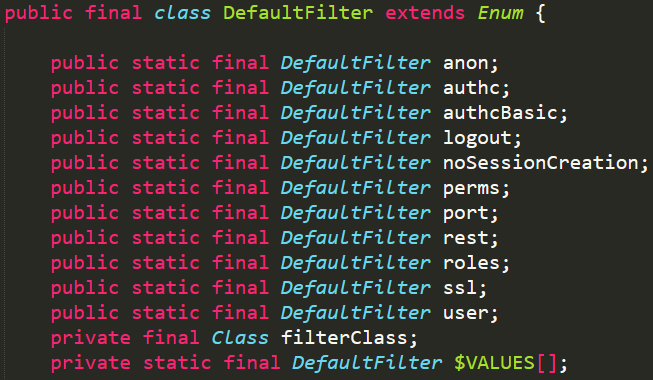
◎JSP/GSP标签:在JSP/GSP页面通过相应的标签来完成

如:

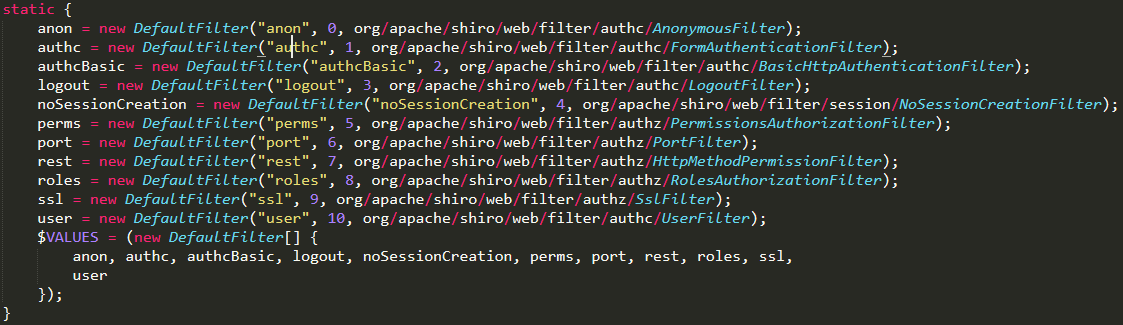


默认过滤器:

Shiro内置了很多默认的过滤器，比如身份认证、授权等相关的。这些过滤器可详见org.apache.shiro.web.filter.mgt包下的枚举类:



它们的默认实现是:



上图中权限相关的过滤器介绍:

|  |  |
| --- | --- |
| 授权相关过滤器 | 说明 |
| roles | 角色授权过滤器，验证用户是否拥有某些角色  如: \*/admin/\* = roles[admin] |
| perms | 权限授权过滤器，验证用户是否拥有某些权限  如: /user/\*\* = perms[user:create] |
| port | 端口过滤器  如: /test = port[80] 如果用户访问该页面用的不是80端口，那么将自动重定向到80端口进行访问 |
| rest | rest风格过滤器，自动更具请求方法构建权限字符串  GET = read、POST = create、 PUT = update、DELETE = delete、 HEAD =read、TRACE = read 、 OPTIONS =read、 MKCOL =create构建权限字符串。  如: /users = rest[user] ,会自动拼出“user:read,user:create,user:update,user:delete”权限字符串进行权限匹配 |
| ssl | SSL过滤器，只有请求协议时HTTPS才能通过;否则自动跳转回HTTPS端口 |
| noSessionCreation | 不创建会话过滤器，调用subject.getSession(false)不会有什么问题，但是如果subject.getSession(true)将抛出异常 |

Permissions实例级访问控制

规则:域:操作:被付诸实施的实例，

即为:对哪个域资源的哪个实例可以进行什么操作。

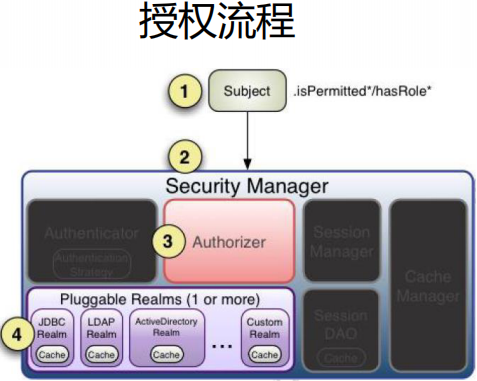
“:”表示资源/操作/实例的分割;“,”表示操作的分割，“\*”表示任意资源/操作/实例。

注:使用通配示例:user:edit:\* 、user:\*:\* 、user:\*:manager

注:部分省略通配符示例: user:query等价于user:query:\* 、user等价于user:\*:\*

注:通配符只能从字符串的结尾处省略不见，如: user:edit等价于user:edit:\*，而不等价于user:\*:edit

授权流程:



授权流程:

第一步:首先调用Subject.isPermitted\*/hasRole\*接口，其会委托给 SecurityManager，而SecurityManager接着会委托给Authorizer授权 器；

第二步:Authorizer是真正的授权者,如果调用如isPermitted（“user:view”）, 其首先会通过PermissionResolver把字符串转换成相应的Permission 实例。

第三步:在进行授权之前，其会调用相应的Realm获取Subject相应的角色/权限 用于匹配传入的角色/权限。

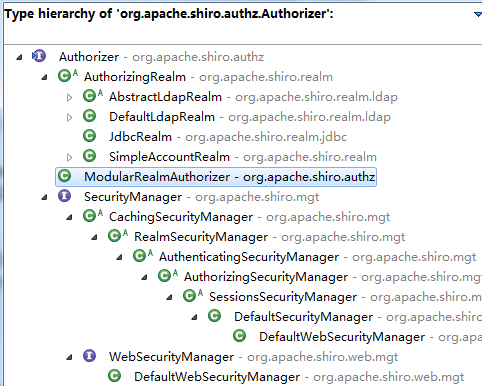
第四步:Authorizer会判断Realm的角色/权限是否和传入的匹配，如果有多个 Realm，会委托给ModularRealmAuthorizer进行循环判断,如果匹配，则 返回true,否则返回false表示授权失败。

ModularRealmAuthorizer:

授权器的集成实现关系



授权器的集成实现关系详细展开



ModularRealmAuthorizer进行多Realm匹配流程:

第一步:首先检查Realm是否实现了Authorizer;

第二步:如果实现了Authorizer,那么接着调用其相应的

isPermitted\*/hasRole\*接口进行匹配。

第三步:如果有一个Realm匹配那么将返回true,否则返回false。

权限注解:

@RequiresAuthentication:表示当前Subject需要先通过login身份验证才行; 即Subject.isAuthenticated()返回true。

@RequiresUser:表示当前Subject需要先身份验证或先通过记住我登陆才行。

@RequiresRoles(value = {“admin”, “user”}, logical = Logical.AND): 表示当前Subject同时需要admin和user的角色。

@RequiresPermissions（value = {“user:a”, “user:b”},）:表示当前Subject 需要user:a或user:b的权限。

自定义拦截器:

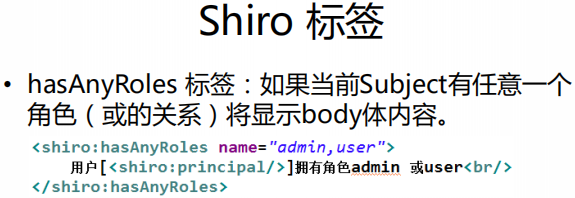
通过自定义拦截器可以扩展功能,如:动态URL-角色/权限访问控制的实现、根据Subject身份信息获取用户信息绑定到Request(即设置通用数据)、验证码验证、在线用户信息的保存等。

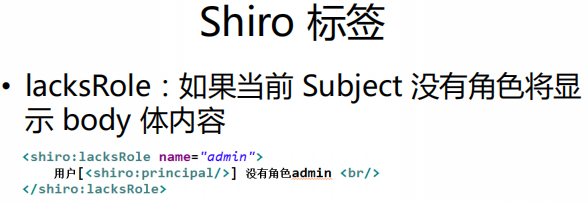
Shiro的JSTL标签(部分):



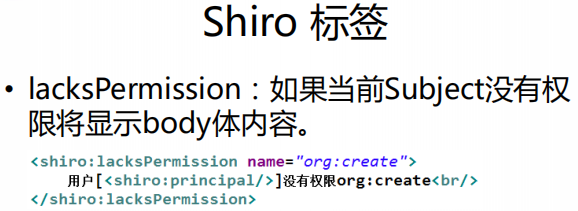












第八节 授权流程代码示例

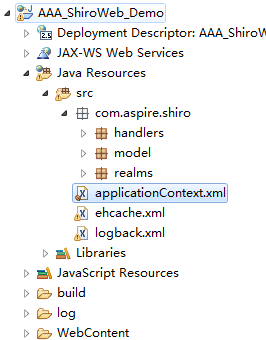
说明:在用户登录经过认证后,就会接着走授权逻辑。这时，我们可以根据在认证时，放入SimpleAuthenticationInfo中的principal来获取该用户的某个信息，再根据该信息内容，授权该用户对应的权限。

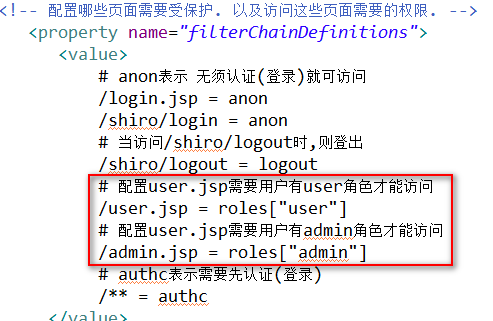
注:授权是在认证之后的。

注:Shiro过滤时，限制了访问某些资源需要哪些角色;所以授权，就是一个给用户添加角色的过程。

示例代码:

准备工作:首先配置哪些资源需要哪些角色(权限)才能访问





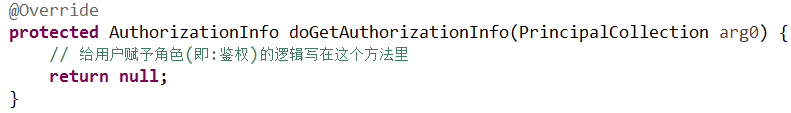
注:roles[“”]一定要有双引号。

注:本节末，会对这种配置进行优化。

第一步:写自己的Realm继承AuthorizingRealm类，并重写doGetAuthenticationInfo和doGetAuthorizationInfo这两个方法;其中doGetAuthenticationInfo方法时我们在前面认证时就已经重写了的;所以授权时，我们只需要重写doGetAuthorizationInfo方法即可。

注:如果有多个认证Reaml的话，那么会按照配置的Reaml的顺序，先将所有Reaml中的认证方法走完之后。

注:只有访问那些需要权限的路径时，才会走鉴权(授权)方法。即:程序在跳转时，判断是否该路径是否需要鉴权、需要哪些权限。



第二步:编写具体实现



给出文字版:

|  |
| --- |
| @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {    /\*  \* 1.从principals中获取到第一个principal  \* 注:多个Realm时,Shiro返回给此方法的参数principals的规则是由身份认证策略控制的(可详见:身份认证策略)  \* 注:默认的策略为AtLeastOneSuccessfulStrategy,那么这里只能获取  \* 到对应的通过了身份认证的Realm中的principal  \* 注:就算有多个Realm,一般而言这些Realm中的principal都应该是一样的,所以  \* 我们读取信息时，一般拿其中的某一个进行读取就行  \*/  Object principal = principals.getPrimaryPrincipal();    /\*  \* 2.从配置applicationContext.xml中可知:我们写的ShiroReaml是第一个,  \* 而ShiroReaml中的principal,我们传的是User对象  \* 所以这里获取到的principal即为该User对象,并获取到对应其角色信息  \*/  User user = (User)principal;  // 获取该用户(带的)角色信息  String[] roles = **null**;  **if**(user.getUserRoles() != **null**) {  roles = (user.getUserRoles()).split(",");  }    // 3.创建一个Set,来放置用户拥有的角色  java.util.Set<String> rolesSet = **new** java.util.HashSet<>();  **for** (String role : roles) {  rolesSet.add(role);  }  // 4.创建 SimpleAuthorizationInfo, 并将办好角色的Set放入.  SimpleAuthorizationInfo info = **new** SimpleAuthorizationInfo(rolesSet);    // 5.返回 SimpleAuthorizationInfo对象.  **return** info;  } |

注:到此步骤为止,鉴权就算是完成了,下面给出一些补充。

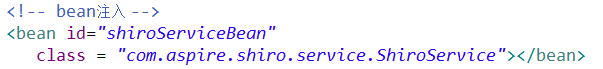
给出使用注解来验证权限的示例:

第一步:在Service层中创建方法，并在需要权限(角色)的方法上添加相应注解，如:

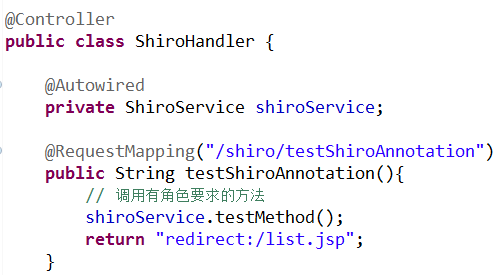


注:别忘了把这个类注入容器。

这里给出如何通过配置applicationContext.xml的方式将其注入:



第二步:在controller层调用方法:



第三步:测试一下

先使用一个同时有“user”和“admin”角色的用户登录，访问该URL,可看见控制台输出:

1

说明,放行了；

登出后,再使用一个只有“user”角色的用户登录，访问该URL,可看见控制台输出异常信息:

2

有上图中的异常信息可知,该用户由于没有满足角色(权限)要求；所以调用失败。

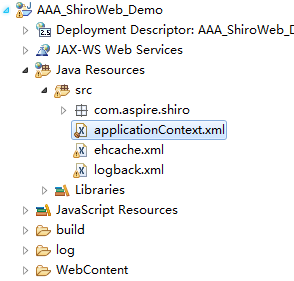
注:在进行下面的解时前，这里先解释一下Spring与SpringMVC容器之间的关系:

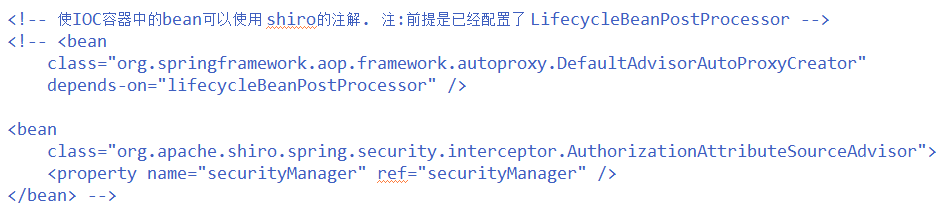
在Spring整体框架的核心概念中，容器是核心思想，就是用来管理Bean的整个生命周期的，而在一个项目中，容器不一定只有一个，Spring中可以包括多个容器，而且容器有上下层关系，目前最常见的一种场景就是在一个项目中引入Spring和SpringMVC这两个框架，那么它其实就是两个容器，Spring是父容器，SpringMVC是其子容器，并且在Spring父容器中注册的Bean对于SpringMVC容器中是可见的，而在SpringMVC容器中注册的Bean对于Spring父容器中是不可见的，也就是子容器可以看见父容器中的注册的Bean，反之就不行(即:类与类之间的继承关系)。最常用的bean的范畴:controller在SpringMVC容器中，而Service、Daod等在Spring容器中。

更多可参考:<http://www.cnblogs.com/junzi2099/p/8042476.html>

注:在上面的示例代码中，@RequiresRoles注解是写在service层中的;如果service层中的目标方法上要开启@Transactional声明式事务注解,那么我们就不能再把@RequiresRoles等注解放在该方法上了(注:这是因为那样做会发生重复代理的问题)，此时我们应该把Shiro的鉴权注解放在Controller层中的方法上。但是由于我们之前是将Shiro的关于扫描注解的配置放在Spring中的,所以其不能扫描到Controler层中的方法(因为controller是属于SpringMVC容器的)，所以如果我们想把Shiro注解写在controller中的方法上，那么我们需要将Shiro的关于扫描注解的配置放在SprinMVC的配置中:

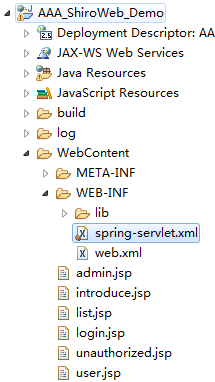
①将Spring的配置文件applicationContext.xml中的Shiro的关于扫描注解的配置删除或注释掉。

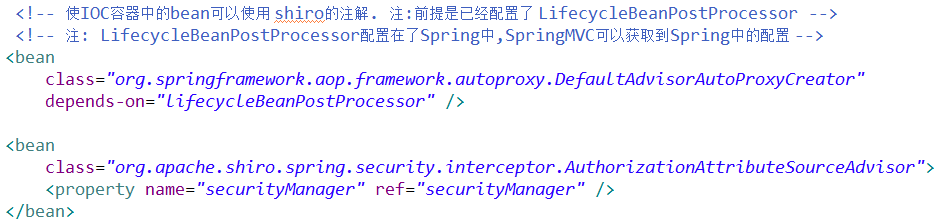




注:LifecycleBeanPostProcessor依然配置在Spring中也行。

②在SpringMVC的配置文件spring-servlet.xml中加上Shiro的关于扫描注解配置。

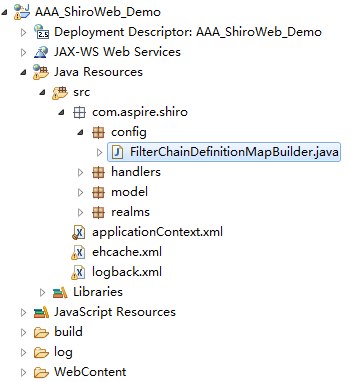




此时，在controller层里面的方法里使用Shiro的鉴权注解，可知原来不生效的鉴权注解现在生效了，即:成功。

给出上述对准备工作的优化:

第一步:编写一个配置类



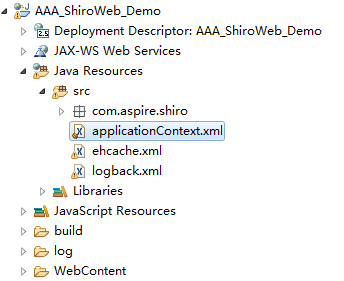
第二步:把资源过滤信息配置在这个类中

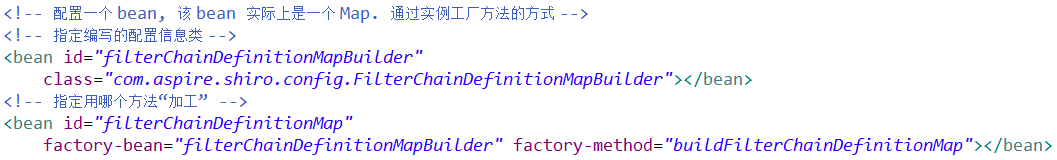


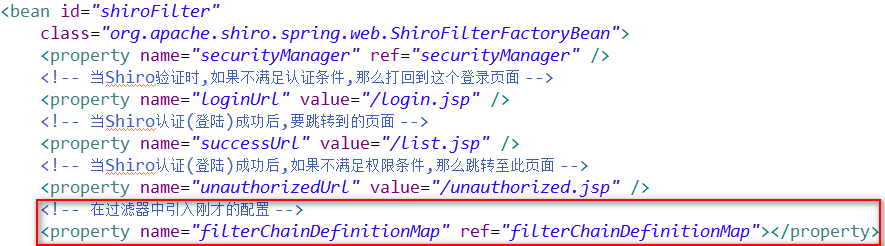
给出文字版:

|  |
| --- |
| **import** java.util.LinkedHashMap;  **public** **class** FilterChainDefinitionMapBuilder {  /\*\*  \* 资源配置信息  \* 注:配置信息一定要放在LinkedHashMap中进行返回  \*  \* **@return** LinkedHashMap  \* **@date** 2018年8月16日 上午2:38:56  \*/  **public** LinkedHashMap<String, String> buildFilterChainDefinitionMap() {  LinkedHashMap<String, String> map = **new** LinkedHashMap<>();  map.put("/login.jsp", "anon");  map.put("/shiro/login", "anon");  map.put("/shiro/logout", "logout");  // 注意roles[user]这里的话,角色不要再用引号引起来了,直接写即可  map.put("/user.jsp", "authc,roles[user]");  map.put("/admin.jsp", "authc,roles[admin]");  map.put("/\*\*", "authc");  **return** map;  }  } |

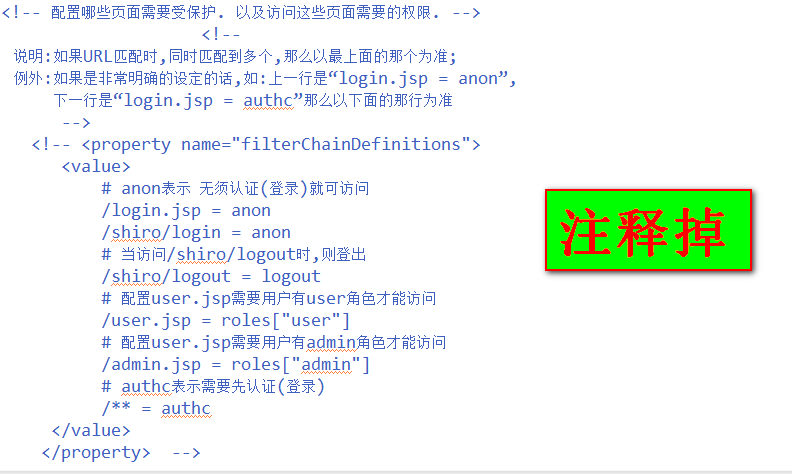
第三步:注入这个类，并引用这个类的配置





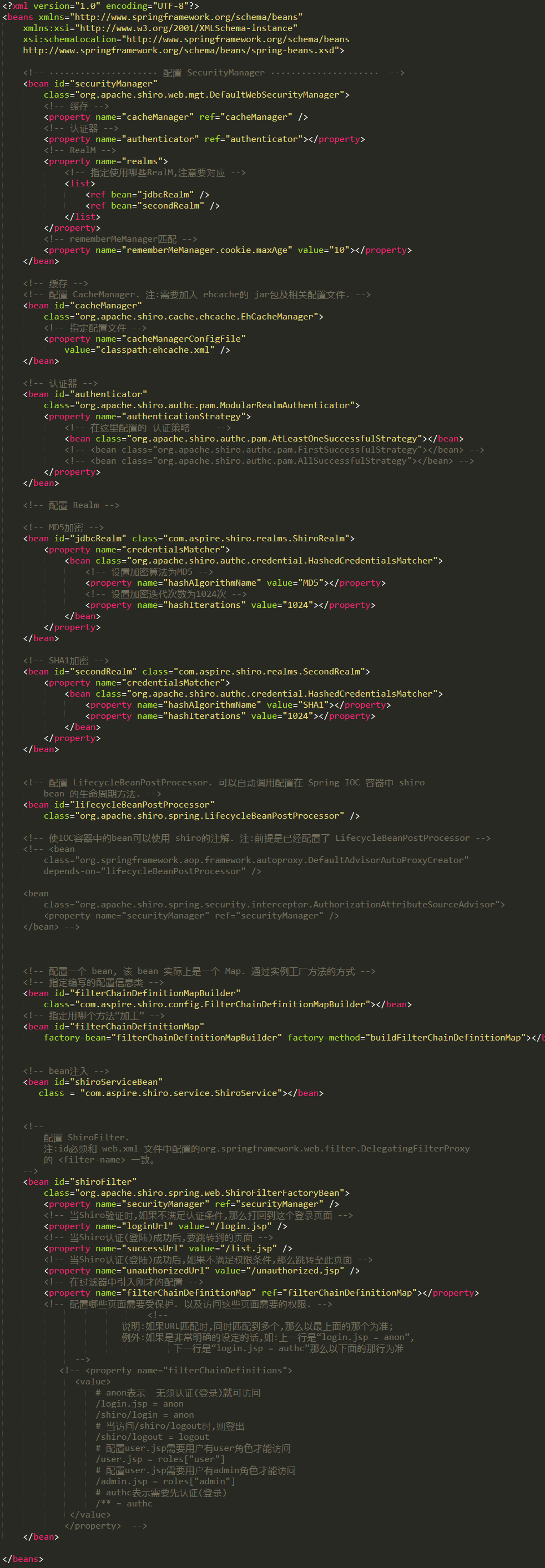


第四步:注释掉多余的配置



注:关于如何使用Shiro的JSTL标签这里就不再演示了。

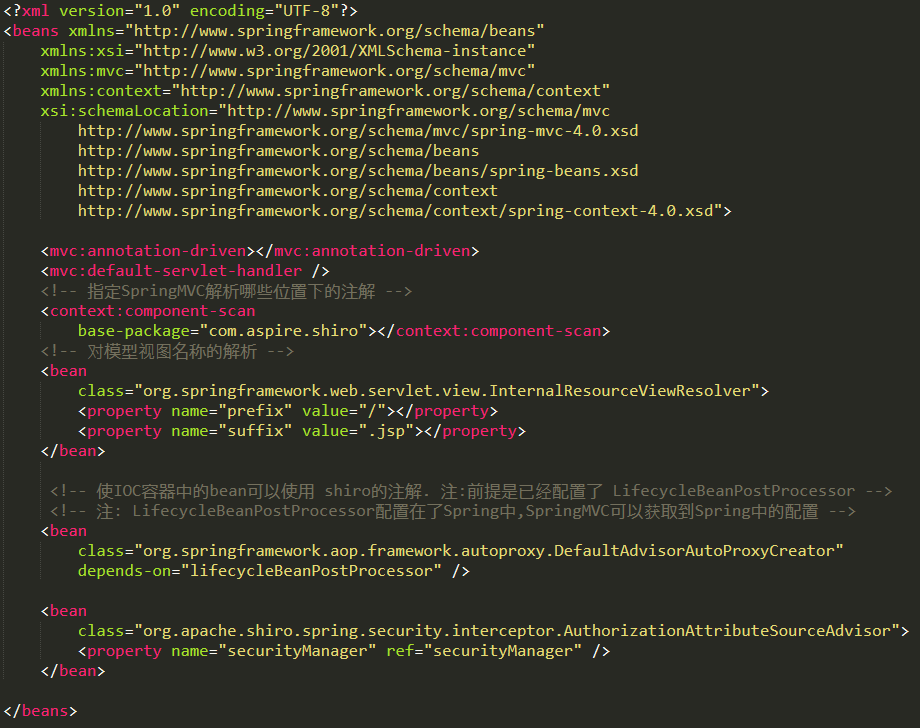
给出到这一步为止，Spring的配置文件applicationContext.xml文件的配置:



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- ····················· 配置 SecurityManager ····················· -->  <bean id=*"securityManager"*  class=*"org.apache.shiro.web.mgt.DefaultWebSecurityManager"*>  <!-- 缓存 -->  <property name=*"cacheManager"* ref=*"cacheManager"* />  <!-- 认证器 -->  <property name=*"authenticator"* ref=*"authenticator"*></property>  <!-- RealM -->  <property name=*"realms"*>  <!-- 指定使用哪些RealM,注意要对应 -->  <list>  <ref bean=*"jdbcRealm"* />  <ref bean=*"secondRealm"* />  </list>  </property>  <!-- rememberMeManager匹配 -->  <property name=*"rememberMeManager.cookie.maxAge"* value=*"10"*></property>  </bean>  <!-- 缓存 -->  <!-- 配置 CacheManager. 注:需要加入 ehcache的 jar包及相关配置文件. -->  <bean id=*"cacheManager"*  class=*"org.apache.shiro.cache.ehcache.EhCacheManager"*>  <!-- 指定配置文件 -->  <property name=*"cacheManagerConfigFile"*  value=*"classpath:ehcache.xml"* />  </bean>  <!-- 认证器 -->  <bean id=*"authenticator"*  class=*"org.apache.shiro.authc.pam.ModularRealmAuthenticator"*>  <property name=*"authenticationStrategy"*>  <!-- 在这里配置的 认证策略 -->  <bean class=*"org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy"*></bean>  <!-- <bean class="org.apache.shiro.authc.pam.FirstSuccessfulStrategy"></bean> -->  <!-- <bean class="org.apache.shiro.authc.pam.AllSuccessfulStrategy"></bean> -->  </property>  </bean>  <!-- 配置 Realm -->    <!-- MD5加密 -->  <bean id=*"jdbcRealm"* class=*"com.aspire.shiro.realms.ShiroRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <!-- 设置加密算法为MD5 -->  <property name=*"hashAlgorithmName"* value=*"MD5"*></property>  <!-- 设置加密迭代次数为1024次 -->  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- SHA1加密 -->  <bean id=*"secondRealm"* class=*"com.aspire.shiro.realms.SecondRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <property name=*"hashAlgorithmName"* value=*"SHA1"*></property>  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- 配置 LifecycleBeanPostProcessor. 可以自动调用配置在 Spring IOC 容器中 shiro  bean 的生命周期方法. -->  <bean id=*"lifecycleBeanPostProcessor"*  class=*"org.apache.shiro.spring.LifecycleBeanPostProcessor"* />  <!-- 使IOC容器中的bean可以使用 shiro的注解. 注:前提是已经配置了 LifecycleBeanPostProcessor -->  <!-- <bean  class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"  depends-on="lifecycleBeanPostProcessor" />    <bean  class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">  <property name="securityManager" ref="securityManager" />  </bean> -->        <!-- 配置一个 bean, 该 bean 实际上是一个 Map. 通过实例工厂方法的方式 -->  <!-- 指定编写的配置信息类 -->  <bean id=*"filterChainDefinitionMapBuilder"*  class=*"com.aspire.shiro.config.FilterChainDefinitionMapBuilder"*></bean>  <!-- 指定用哪个方法“加工” -->  <bean id=*"filterChainDefinitionMap"*  factory-bean=*"filterChainDefinitionMapBuilder"* factory-method=*"buildFilterChainDefinitionMap"*></bean>      <!-- bean注入 -->  <bean id=*"shiroServiceBean"*  class = *"com.aspire.shiro.service.ShiroService"*></bean>  <!-- 配置 ShiroFilter. 注:id必须和 web.xml 文件中配置的org.springframework.web.filter.DelegatingFilterProxy  的 <filter-name> 一致。 -->  <bean id=*"shiroFilter"*  class=*"org.apache.shiro.spring.web.ShiroFilterFactoryBean"*>  <property name=*"securityManager"* ref=*"securityManager"* />  <!-- 当Shiro验证时,如果不满足认证条件,那么打回到这个登录页面 -->  <property name=*"loginUrl"* value=*"/login.jsp"* />  <!-- 当Shiro认证(登陆)成功后,要跳转到的页面 -->  <property name=*"successUrl"* value=*"/list.jsp"* />  <!-- 当Shiro认证(登陆)成功后,如果不满足权限条件,那么跳转至此页面 -->  <property name=*"unauthorizedUrl"* value=*"/unauthorized.jsp"* />  <!-- 在过滤器中引入刚才的配置 -->  <property name=*"filterChainDefinitionMap"* ref=*"filterChainDefinitionMap"*></property>  <!-- 配置哪些页面需要受保护. 以及访问这些页面需要的权限. -->  <!--  说明:如果URL匹配时,同时匹配到多个,那么以最上面的那个为准;  例外:如果是非常明确的设定的话,如:上一行是“login.jsp = anon”,  下一行是“login.jsp = authc”那么以下面的那行为准  -->  <!-- <property name="filterChainDefinitions">  <value>  # anon表示 无须认证(登录)就可访问  /login.jsp = anon  /shiro/login = anon  # 当访问/shiro/logout时,则登出  /shiro/logout = logout  # 配置user.jsp需要用户有user角色才能访问  /user.jsp = roles["user"]  # 配置user.jsp需要用户有admin角色才能访问  /admin.jsp = roles["admin"]  # authc表示需要先认证(登录)  /\*\* = authc  </value>  </property> -->  </bean>  </beans> |

给出到这一步为止，SpringMVC的配置文件spring-servlet.xml文件的配置:



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>    <mvc:annotation-driven></mvc:annotation-driven>  <mvc:default-servlet-handler />  <!-- 指定SpringMVC解析哪些位置下的注解 -->  <context:component-scan  base-package=*"com.aspire.shiro"*></context:component-scan>  <!-- 对模型视图名称的解析 -->  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/"*></property>  <property name=*"suffix"* value=*".jsp"*></property>  </bean>    <!-- 使IOC容器中的bean可以使用 shiro的注解. 注:前提是已经配置了 LifecycleBeanPostProcessor -->  <!-- 注: LifecycleBeanPostProcessor配置在了Spring中,SpringMVC可以获取到Spring中的配置 -->  <bean  class=*"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"*  depends-on=*"lifecycleBeanPostProcessor"* />    <bean  class=*"org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"*>  <property name=*"securityManager"* ref=*"securityManager"* />  </bean>  </beans> |

第九节 会话

提示:此会话虽然是Shiro自己提供的,但是其余Web的会话有很多共通相似之处;阅读此章节时，可参考Web的会话。

基本介绍:

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

回话相关的API:

声明:下述方法调用时，应该用对应类的实例调用;这里为了更直观，直接用类名打点了。

Subject.getSession():即可获取会话;其等价于Subject.getSession(true)，即如果当前没有创建Session对象则会创建一个，有的话直接获取。

注:Subject.getSession(false),如果当前没有创建Session则返回null。

Session.getId():获取当前会话的唯一标识。

Session.getHost():获取当前Subject的主机地址。

Session.getTinmeout(毫秒)/Session.getTinmeout(毫秒):获取/设置当前session的过期时间。

Session.getStartTimestamp()/Session.getLastAccessTime():获取会话的启 动时间/获取会话的最后访问时间。

注:如果是JavaSE，那么需要自己手动调用Session.touch()去更新最后访问时间;如果是Web应用，每次进入ShiroFilter都会自动调用Session.touch()来更新最后访问时间。

Session.touch()/Session.stop():更新会话最后访问时间/销毁会话;当Subject.logout()时，会自动调用Session.stop()来销毁会话。

注:如果在Web中，调用HttpSession.invalidate()也会自动调用Shiro的Session.stop()方法进行销毁Shiro的会话。

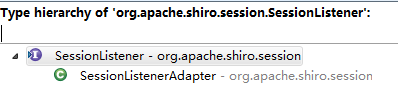
Session.setAttribute(key,value):往会话中放置一些key-value数据;在整个会话范围内都可以对这些属性进行操作。

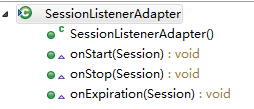
Session.getAttribute(key):从会话中获取某个对应的value;在整个会话范围内都可以对这些属性进行操作。

Session.removeAttribute(key):删除会话中的某个ke-value;在整个会话范围内都可以对这些属性进行操作。

会话监听器

会话监听器可用于监听会话创建事件、会话过期事件、会话停止事件。





Session持久化:

为什么需要持久化:

　　客户端访问了某个能开启会话功能的资源， web服务器就会创建一个与该客户端对应的HttpSession对象，每个HttpSession对象都要站用一定的内存空间。如果在某一时间段内访问站点的用户很多，web服务器内存中就会积累大量的HttpSession对象，消耗大量的服务器内存，即使用户已经离开或者关闭了浏览器，web服务器仍要保留与之对应的HttpSession对象，在他们超时之前，一直占用web服务器内存资源。

　　web服务器通常将那些暂时不活动但未超时的HttpSession对象转移到文件系统或数据库或中间件中保存，服务器要使用他们时再将他们从文件系统或数据库中装载入内存，这种技术称为Session的持久化。

　　将HttpSession对象保存到文件系统或数据库中，需要采用序列化的方式将HttpSession对象中的每个属性对象保存到文件系统或数据库中；将HttpSession对象从文件系统或数据库中装载如内存时，需要采用反序列化的方式，恢复HttpSession对象中的每个属性对象。所以存储在HttpSession对象中的每个属性对象必须实现Serializable接口。

Session的持久化的作用：

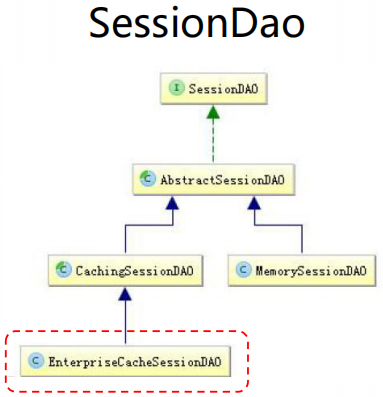
　　1.提高服务器内存的利用率，保证那些暂停活动的客户端在会话超时之前继续原来的会话。

　　2，在多台web服务器协同对外提供服务的集群系统中，使用Session的持久化技术，某台服务器可以将其中发生改变的Session对象复制给其他服务器。保证了在某台服务器停止工作后可以由其他服务器来接替它与客户端的会话。

　　3，在一个web应用程序重启时，服务器也会持久化该应用程序中所有HttpSession对象，保证客户端的会话活动仍可以继续。

SessionDao:

SessionDao的主要功能是:将session写入数据库中，对session进行CRUD增删改查操作。



AbstractSessionDAO提供了SessionDAO的基础实现,如:生成会话ID等。

CachingSessionDAO提供了对开发者透明的会话缓存的功能，需要设置相应的CacheManager。

MemorySessionDAO直接在内存中进行会话维护。

EnterpriseCacheSessionDAO提供了缓存功能的会话维护，默认情况下使用MapCache实现，内部使用ConcurrentHashMap保存缓存的会话。

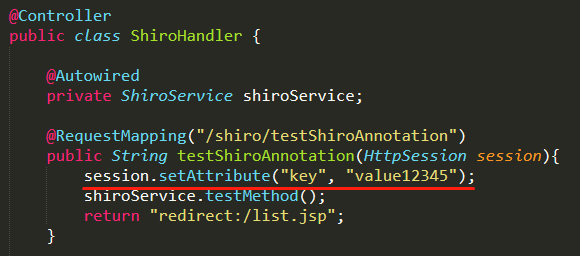
Session的使用:

我们后端之前要获取Web的Session的话，只能在后端的Controller层获取;但是使用Shiro提供的Session的话，在后端的controller、service、dao层等都能获取到Session。但是我们建议:在controller层，还是使用原生的javax.servlet.http.HttpSession,在service、dao层再使用Shiro提供的Session。

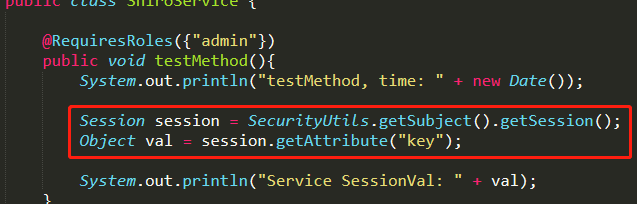
注:用原生的Seesion存放的数据，我们在service、dao层用Shiro的Session也是能够取出来的。

Session节本使用示例:

在Controller层使用原生的Session存



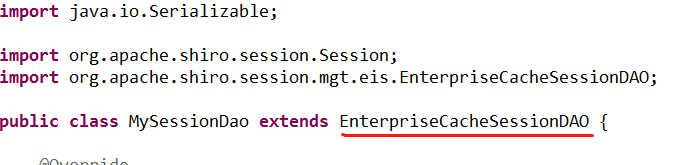
在Service层使用Shiro的Session取



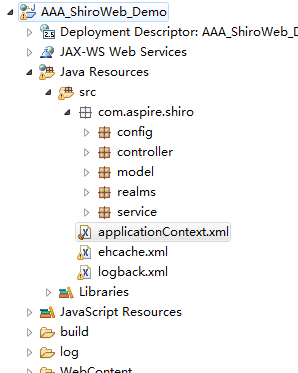
注:上图中的工具类org.apache.shiro.SecurityUtils是由Shiro提供的。

编写我们自己的SessionDao，来管理(即:增删改查)Session:

第一步:创建一个类来继承EnterpriseCacheSessionDAO类（注:这一步暂时不重写类中的方法，在下面的步骤再详细介绍重写哪些方法、怎么重写）



第二步:在Spring的配置文件applicationContext.xml中做相关配置



提示:先给出图片版，文字版在后面统一给出

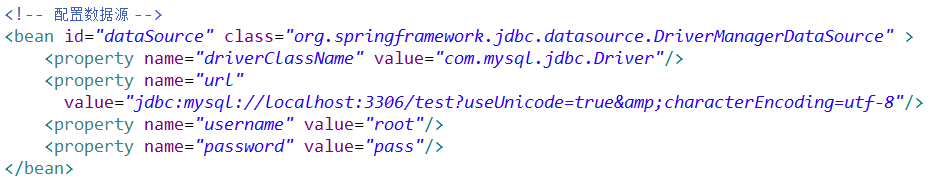
配置session ID生成器、sessionDAO、session会话管理器



在SecurityManager中引用session会话管理器

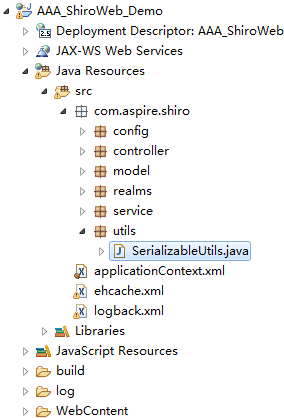


配置数据源:



注:连库操作的几个相关jar包依赖也要build path到项目里，这里就不给出图示了。

第三步:编写Session序列化/反序列化 工具类(等下需要用到)

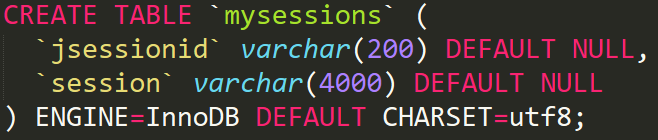




给出文字版:

|  |
| --- |
| **import** org.apache.shiro.codec.Base64;  **import** org.apache.shiro.session.Session;  **import** java.io.ByteArrayInputStream;  **import** java.io.ByteArrayOutputStream;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutputStream;  /\*\*  \* Session序列化工具类  \*/  **public** **class** SerializableUtils {  **public** **static** String serialize(Session session) {  **try** {  ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  ObjectOutputStream oos = **new** ObjectOutputStream(bos);  oos.writeObject(session);  **return** Base64.*encodeToString*(bos.toByteArray());  } **catch** (Exception e) {  **throw** **new** RuntimeException("serialize session error", e);  }  }  **public** **static** Session deserialize(String sessionStr) {  **try** {  ByteArrayInputStream bis = **new** ByteArrayInputStream(  Base64.*decode*(sessionStr));  ObjectInputStream ois = **new** ObjectInputStream(bis);  **return** (Session) ois.readObject();  } **catch** (Exception e) {  **throw** **new** RuntimeException("deserialize session error", e);  }  }  } |

第四步:创建存放session的表



注:具体的数据类型定义，可根据实际情况来定义。

给出文字版:

|  |
| --- |
| CREATE TABLE `mysessions` (  `jsessionid` varchar(200) DEFAULT NULL,  `session` varchar(4000) DEFAULT NULL  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

注:连库从操作需要配置数据库的连接等，这里就不列出来了。

第五步:重写EnterpriseCacheSessionDAO类中的CRUD方法



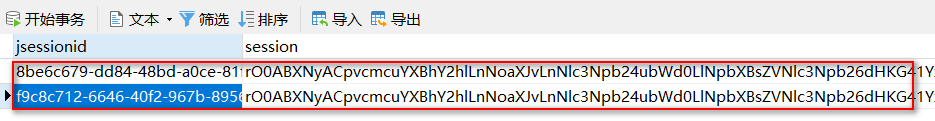
注:JdbcTemplate可以在配置文件中配置并指明其使用相关的DataSource构造。由于本人在xml中配置不成功，所以直接在代码里面生成JdbcTemplate了。

给出文字版:

|  |
| --- |
| **import** java.io.Serializable;  **import** java.util.List;  **import** org.apache.shiro.session.Session;  **import** org.apache.shiro.session.mgt.ValidatingSession;  **import** org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **import** org.springframework.jdbc.core.JdbcTemplate;  **import** org.springframework.jdbc.datasource.DriverManagerDataSource;  **import** com.aspire.shiro.utils.SerializableUtils;  **public** **class** MySessionDao **extends** EnterpriseCacheSessionDAO {  /\*\* 从Spring配置文件(在该配置文件中配置了数据源)中获取到DataSource \*/  **private** **static** ApplicationContext *ac* = **new** ClassPathXmlApplicationContext("applicationContext.xml");  **private** **static** DriverManagerDataSource *dataSource* = (DriverManagerDataSource) *ac*.getBean("dataSource");  /\*\* 获取JdbcTemplate实例 \*/  **private** **static** JdbcTemplate *jdbcTemplate* = **new** JdbcTemplate(*dataSource*);  @Override  **protected** Serializable doCreate(Session session) {  Serializable jsessionId = generateSessionId(session);  assignSessionId(session, jsessionId);  /\*  \* jsessionid就是用来判断当前用户对应于哪个session的。  \* 即:数据库表中的id对应jsessionid;只要定位到了jsessionid,就相当于定位到了session  \*  \*/  String sql = "insert into mysessions(jsessionid, session) values(?,?)";  *jdbcTemplate*.update(sql, jsessionId, SerializableUtils.*serialize*(session));  **return** jsessionId;  }  @Override  **protected** Session doReadSession(Serializable jsessionId) {  String sql = "select session from mysessions where jsessionid=?";  List<String> sessionStrList = *jdbcTemplate*.queryForList(sql, String.**class**, jsessionId);  **if** (sessionStrList.size() == 0)  **return** **null**;  **return** SerializableUtils.*deserialize*(sessionStrList.get(0));  }  @Override  **protected** **void** doUpdate(Session session) {  **if** (session **instanceof** ValidatingSession && !((ValidatingSession) session).isValid()) {  **return**;  }  Serializable jsessionId = generateSessionId(session);  String sql = "update mysessions set session=? where jsessionid=?";  *jdbcTemplate*.update(sql, SerializableUtils.*serialize*(session), jsessionId);  }  @Override  **protected** **void** doDelete(Session session) {  Serializable jsessionId = generateSessionId(session);  String sql = "delete from mysessions where jsessionid=?";  *jdbcTemplate*.update(sql, jsessionId);  }  } |

第六步:此时我们就可以使用上面的我们自己创建的SessionDao来实现对Session的增删改查了。

启动项目，随便跑一下，即可看见数据库表中多出了session数据了:



注:用户首次进入页面(无论是否登录)，会想表里记录一条会话数据，如果这时，用户进行了登录,那么还是使用原来那个会话;如果用户登陆后，进行了登出,那么此时会使用新产生的会话,即会向表中插入新的一条会话。

追注1:触发一次登出,那么就会插入一条会话;所以“登出”按钮不应暴露给游客;即:只有用户登陆后才可以登出。

追注2:关于会话的产生问题,可在实际使用时进一步熟悉。

会话验证:

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

处于性能考虑，一般情况下都是获取会话时来验证会话是否过期并停止会话的;但是如果在web环境中，如果用户不主动退出的话，是不知道会话是否过期的，此时需要定期的检测是否过期，如果过期了则销毁会话，则调用方法进行销毁。

Shiro提供了会话验证调度器SessionValidationScheduler。

Shiro也提供了使用Quartz会话验证调度器:QuartzSessionValidationScheduler。

给出到这里为止，Spring配置文件的内容:



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- ····················· 配置 SecurityManager ····················· -->  <bean id=*"securityManager"*  class=*"org.apache.shiro.web.mgt.DefaultWebSecurityManager"*>  <!-- 缓存 -->  <property name=*"cacheManager"* ref=*"cacheManager"* />  <!-- rememberMeManager -->  <property name=*"rememberMeManager"* ref=*"rememberMeManager"*></property>  <!-- 认证器 -->  <property name=*"authenticator"* ref=*"authenticator"*></property>  <!-- RealM -->  <property name=*"realms"*>  <!-- 指定使用哪些RealM,注意要对应 -->  <list>  <ref bean=*"jdbcRealm"* />  <ref bean=*"secondRealm"* />  </list>  </property>  <!--  sessionManager,如果要将session持久化,那么需要ref="sessionManager"  注:其余的辅助配置在下面已经配置好了  -->  <property name=*"sessionManager"* ref=*"sessionManager"*></property>  </bean>  <!-- 缓存 -->  <!-- 配置 CacheManager. 注:需要加入 ehcache的 jar包及相关配置文件. -->  <bean id=*"cacheManager"*  class=*"org.apache.shiro.cache.ehcache.EhCacheManager"*>  <!-- 指定配置文件 -->  <property name=*"cacheManagerConfigFile"*  value=*"classpath:ehcache.xml"* />  </bean>    <!-- rememberMeCookie配置-->  <bean id=*"rememberMeCookie"* class=*"org.apache.shiro.web.servlet.SimpleCookie"*>  <!-- 设置Cookie在客户端浏览器中保存内容的名字，由用户自己来设置 -->  <constructor-arg value=*"rememberMe"*/>  <!-- 保证该系统不会受到跨域的脚本操作攻击 -->  <property name=*"httpOnly"* value=*"true"*/>  <!-- 定义该Cookie的过期时间,单位为秒。如果设置为-1标识浏览器关闭就失效 -->  <property name=*"maxAge"* value=*"60"*/>  </bean>    <!-- rememberMe管理器-->  <bean id=*"rememberMeManager"*  class=*"org.apache.shiro.web.mgt.CookieRememberMeManager"*>  <!-- RememberMe对应加密后的Cookie的cipherKey -->  <property name=*"cipherKey"*  value=*"#{T(org.apache.shiro.codec.Base64).decode('4AvVhmFLUs0KTA3Kprsdag==')}"* />  <property name=*"cookie"* ref=*"rememberMeCookie"*/>  </bean>  <!-- 认证器 -->  <bean id=*"authenticator"*  class=*"org.apache.shiro.authc.pam.ModularRealmAuthenticator"*>  <property name=*"authenticationStrategy"*>  <!-- 在这里配置的 认证策略 -->  <bean class=*"org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy"*></bean>  <!-- <bean class="org.apache.shiro.authc.pam.FirstSuccessfulStrategy"></bean> -->  <!-- <bean class="org.apache.shiro.authc.pam.AllSuccessfulStrategy"></bean> -->  </property>  </bean>  <!-- 配置 Realm -->    <!-- MD5加密 -->  <bean id=*"jdbcRealm"* class=*"com.aspire.shiro.realms.ShiroRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <!-- 设置加密算法为MD5 -->  <property name=*"hashAlgorithmName"* value=*"MD5"*></property>  <!-- 设置加密迭代次数为1024次 -->  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- SHA1加密 -->  <bean id=*"secondRealm"* class=*"com.aspire.shiro.realms.SecondRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <property name=*"hashAlgorithmName"* value=*"SHA1"*></property>  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- 配置 LifecycleBeanPostProcessor. 可以自动调用配置在 Spring IOC 容器中 shiro  bean 的生命周期方法. -->  <bean id=*"lifecycleBeanPostProcessor"*  class=*"org.apache.shiro.spring.LifecycleBeanPostProcessor"* />    <!-- 配置一个 bean, 该 bean 实际上是一个 Map. 通过实例工厂方法的方式 -->  <!-- 指定编写的配置信息类 -->  <bean id=*"filterChainDefinitionMapBuilder"*  class=*"com.aspire.shiro.config.FilterChainDefinitionMapBuilder"*></bean>  <!-- 指定用哪个方法“加工” -->  <bean id=*"filterChainDefinitionMap"*  factory-bean=*"filterChainDefinitionMapBuilder"* factory-method=*"buildFilterChainDefinitionMap"*>  </bean>      <!-- bean注入 -->  <bean id=*"shiroServiceBean"*  class = *"com.aspire.shiro.service.ShiroService"*></bean>  <!--  配置 ShiroFilter.  注:id必须和 web.xml 文件中配置的org.springframework.web.filter.DelegatingFilterProxy  的 <filter-name> 一致。  -->  <bean id=*"shiroFilter"*  class=*"org.apache.shiro.spring.web.ShiroFilterFactoryBean"*>  <property name=*"securityManager"* ref=*"securityManager"* />  <!-- 当Shiro验证时,如果不满足认证条件,那么打回到这个登录页面 -->  <property name=*"loginUrl"* value=*"/login.jsp"* />  <!-- 当Shiro认证(登陆)成功后,要跳转到的页面 -->  <property name=*"successUrl"* value=*"/list.jsp"* />  <!-- 当Shiro认证(登陆)成功后,如果不满足权限条件,那么跳转至此页面 -->  <property name=*"unauthorizedUrl"* value=*"/unauthorized.jsp"* />  <!-- 在过滤器中引入刚才的配置 -->  <property name=*"filterChainDefinitionMap"* ref=*"filterChainDefinitionMap"*></property>  </bean>    <!-- Session ID 生成器-->  <bean id=*"sessionIdGenerator"*  class=*"org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator"*/>    <!-- Session DAO. 继承 EnterpriseCacheSessionDAO -->  <bean id=*"sessionDAO"*  class=*"com.aspire.shiro.realms.MySessionDao"*>  <!-- 缓存引用ehcache.xml中配置的shiro-activeSessionCache -->  <property name=*"activeSessionsCacheName"* value=*"shiro-activeSessionCache"*/>  <!-- EnterpriseCacheSessionDAO需要Session ID 生成器,我们引用上面配置的生成器 -->  <property name=*"sessionIdGenerator"* ref=*"sessionIdGenerator"*/>  </bean>    <!-- 会话管理器-->  <bean id=*"sessionManager"* class=*"org.apache.shiro.web.session.mgt.DefaultWebSessionManager"*>  <!-- session失效时间30分钟 -->  <property name=*"globalSessionTimeout"* value=*"1800000"*/>  <!-- 删除失效了的session -->  <property name=*"deleteInvalidSessions"* value=*"true"*/>  <!-- 定时检查session是否失效 -->  <property name=*"sessionValidationSchedulerEnabled"* value=*"true"*/>  <!-- 指定引用上面配置的session -->  <property name=*"sessionDAO"* ref=*"sessionDAO"*/>  </bean>    <!-- 配置数据源 -->  <bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"* >  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>  <property name=*"url"*  value=*"jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=utf-8"*/>  <property name=*"username"* value=*"root"*/>  <property name=*"password"* value=*"pass"*/>  </bean>  </beans> |

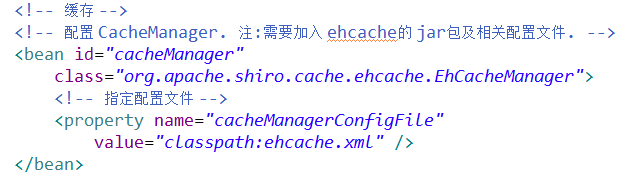
第十节 缓存

CacheManagerAware接口

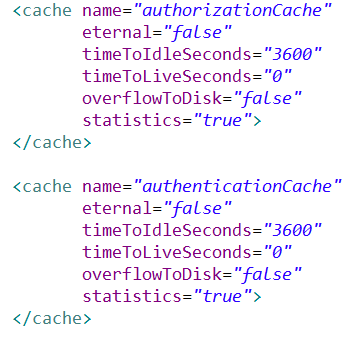
Shiro内部相应的组件(DefaultSecurityManager)会自动检测相应的对象(如Realm、Session等)是否实现了CacheManagerAware并自动注入相应的CacheManager。



从上图我们还可知，Shiro提供了CachingRealm，其实现了CacheManagerAware接口，提供了缓存的一些基础实现。AuthenticatingRealm及AuthorizingRealm分别提供了对AuthenticationInfo和AuthorizationInfo信息的缓存。我们编写的继承了AuthorizingRealm的类默认是有被CacheManager管理的。我们在Spring的配置文件applicationContext.xml中对CacheManager有配置，如图所示:



由上图可知，我们引入了ehcache.xml文件，来讲所有缓存相关的配置放在里面，ehcache.xml中对应身份认证、授权的配置是这样的:



Session缓存

如SecurityManager实现了SessionSecurityManager,其会判断SessionManager是否实现了CacheManagerAware接口，如果实现了则会把CacheManager设置给它。

SesssionManager也会判断相应的SessionDAO(如继承自CachingSessionDAO)是否实现了CacheManagerAware,如果实现了会把CacheManager设置给它。

设置了缓存的SessionManager,查询时会先查缓存，如果找不到才查数据库。

第十一节 RememberMe

简述

Shiro提供了记住我(RememberMe)的功能,比如访问淘宝等一些网站时，关闭了浏览器，下次再打开时还是能记住你是谁，下次访问时无需再登录即可访问。

认证和记住我

Subject.isAuthenticated():表示用户进行了身份验证登录的，即:使用Subject.login进行了登录，进入网站的。

Subject.isRemembered():表示用户是通过记住我进入网站的，此时可能并不是真正的你(如你的朋友使用你的电脑，或者你的Cookie被窃取)在访问。

注:如果用户是通过Subject.isAuthenticated()进行身份认证登录进入的，那么此时Subject.isAuthenticated() == true,而Subject.isRemembered() == false;如果用户是利用Subject.isRemembered()记住我进入的，那么此时Subject.isRemembered() == true;而Subject.isAuthenticated() == false。

user过滤器与authc过滤器的比较:

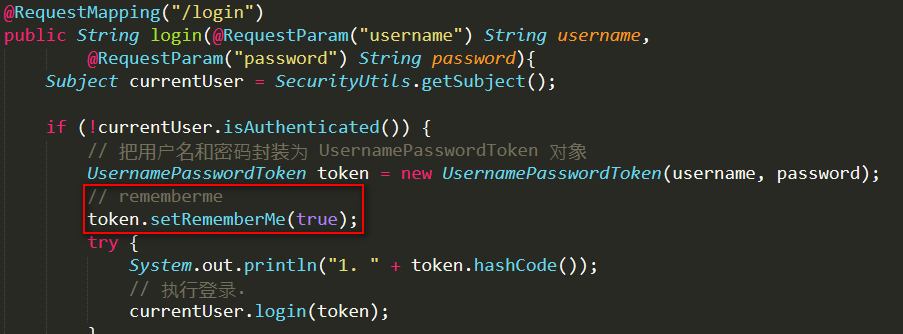
user过滤器:既会放行通过身份认证登录进入的用户，又会放行通过rememberMe进入的用户。

authc过滤器:会放行通过身份认证登录进入的用户，但不会放行通过rememberMe既然怒的用户。

使用RememberMe基本流程如下:

第一步:首先在登录页面选中RememberMe,然后进行登录;如果是浏览器登录，会把RememberMe用来记录信息的Cookie写到客户端并保存下来。

注:在登录时，一般会给用户提供一个选择按钮，提示是否选中RememberMe，如果选中了，那么用户登录进入后端方法时，会调用UsernamePasswordToken的.setRememberMe(true);方法来开启RememberMe,示例:



第二步:关闭浏览器再重新打开，会发现浏览器还是记住你的。

第三步:访问一般的网页服务器还是知道你是谁，且能正常访问。

第四步:但是比如我们访问淘宝时，如果要查看我的订单或进行支付时，此时还是需要再进行身份认证的，以确保当前用户还是你。

我们可以这样对RememberMe进行一些配置:

在Spring的配置文件applicationContext.xml中，配置:



给出文字版:

|  |
| --- |
| <!-- rememberMeCookie配置-->  <bean id=*"rememberMeCookie"* class=*"org.apache.shiro.web.servlet.SimpleCookie"*>  <!-- 设置Cookie在客户端浏览器中保存内容的名字，由用户自己来设置 -->  <constructor-arg value=*"rememberMe"*/>  <!-- 保证该系统不会受到跨域的脚本操作攻击 -->  <property name=*"httpOnly"* value=*"true"*/>  <!-- 定义该Cookie的过期时间,单位为秒。如果设置为-1标识浏览器关闭就失效 -->  <property name=*"maxAge"* value=*"60"*/>  </bean>  <!-- rememberMe管理器-->  <bean id=*"rememberMeManager"*  class=*"org.apache.shiro.web.mgt.CookieRememberMeManager"*>  <!-- RememberMe对应加密后的Cookie的cipherKey -->  <property name=*"cipherKey"*  value=*"#{T(org.apache.shiro.codec.Base64).decode('4AvVhmFLUs0KTA3Kprsdag==')}"* />  <property name=*"cookie"* ref=*"rememberMeCookie"*/>  </bean> |

并在SecurityManager中引入RememberMeManager:



注:如果之前在SecurityManager中简单配置了RememberMe，那么到这一步时，可以将之前在SecurityManager中对RememberMe的简单配置，挪到这里(上上图)名为rememberMeCookie的配置中;在SecurityManager中只需要引入对RememberMeManager即可。

测试一下:

测试准备工作:对一个url配置user过滤器。



测试第一步:启动服务，访问http://localhost:8080/AAA\_ShiroWeb\_Demo/rememberMe.jsp页面;此时由于没有登录,打回到登录页面

测试第二步:进行登录。

注:此时后端登录代码处，启用rememberMe



测试第三步:登录进去后，我们再访问http://localhost:8080/AAA\_ShiroWeb\_Demo/rememberMe.jsp,这时，是能够进去的。

测试第四步:关闭浏览器后，再重新打开该浏览器此时，无需登陆，直接访问

http://localhost:8080/AAA\_ShiroWeb\_Demo/rememberMe.jsp,这时，我们也是能进去的。

测试第五步:再关闭浏览器，等待。直到时长超过了我们设置的RememberMe对应的Cookie的有效时长(在xml中对rememberMe的这个属性进行的设置，前面图中，本人设置的是60秒),这时再访问http://localhost:8080/AAA\_ShiroWeb\_Demo/rememberMe.jsp就又会打回到登录页面。

由此可见:RememberMe配置成功。

建议:

访问一般网页，如个人主页之类的，我们使用user拦截器即可，user拦截器只要用户登录过即可访问。即:isAuthenticated()和isRemembered()登录的都可以。

访问特殊网页，如我的订单，订单提交页面，我们使用authc拦截器即可。Authc拦截器会判断用户是否是通过Subject.login（）登陆的(即:Subject.isAuthenticated() == true)，如果是才放行，否则会跳转到登录页面叫你重新登录。即:只有isAuthenticated()为true登陆的才可以。

第十二节 给出此时的各个配置文件

声明:在本章中，Shiro的配置主要配置在Spring的配置文件即applicationContext.xml中，少许配置在了Tomcat的配置文件即web.xml和SpringMVC的配置文件spring-servlet.xml中。如果要改善的话，那么可以把一下可以挪动的shiro的配置单独放在一个配置文件汇中，对该文件 进行引入即可。

Tomcat的配置文件web.xml:



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"http://java.sun.com/xml/ns/javaee"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*  id=*"WebApp\_ID"* version=*"2.5"*>    <!-- 加载其它的配置文件 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <!-- 加载Spring的配置文件 -->  <param-value>  classpath:applicationContext.xml  </param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>    <!-- 指定DispatcherServlet,匹配Servlet的范围，这里为: / -->  <servlet>  <servlet-name>spring</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>spring</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>    <!--  1. 配置 Shiro 的 shiroFilter.  2. DelegatingFilterProxy 实际上是 Filter 的一个代理对象. 默认情况下, Spring 会到 IOC 容器中查找和  <filter-name> 对应的 filter bean. 也可以通过 targetBeanName 的初始化参数来配置 filter bean 的 id.  -->  <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>    </web-app> |

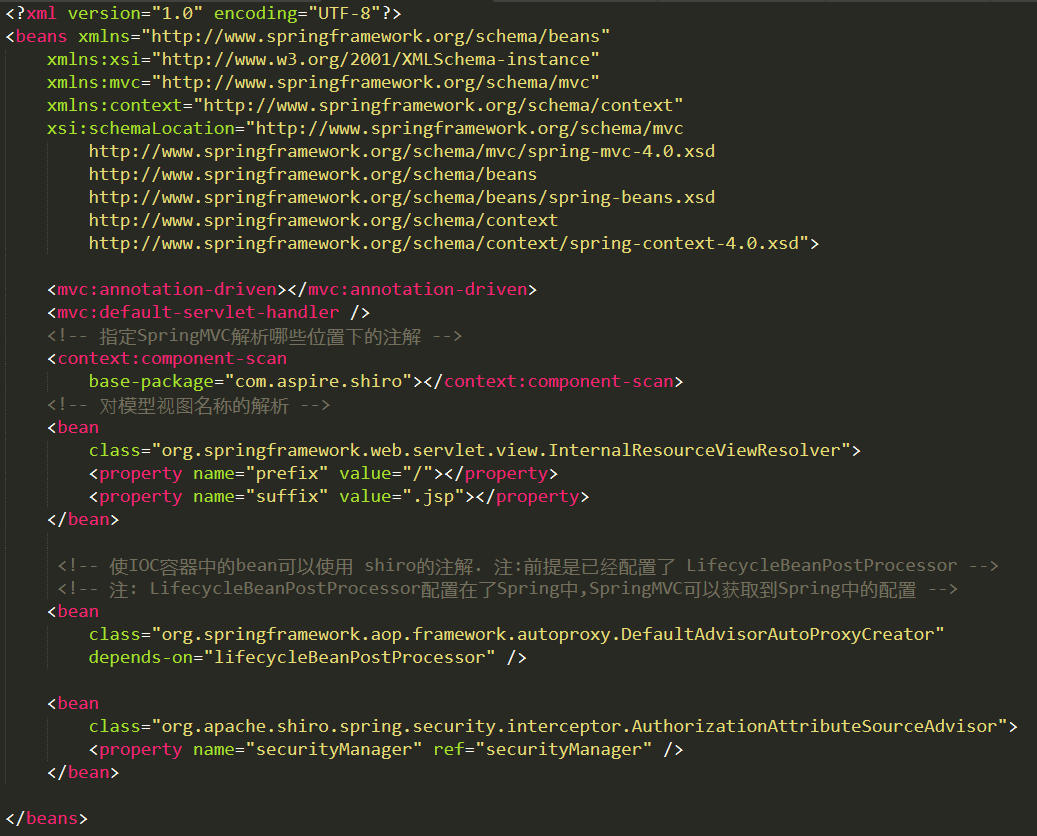
Spring的配置文件即applicationContext.xml



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- ····················· 配置 SecurityManager ····················· -->  <bean id=*"securityManager"*  class=*"org.apache.shiro.web.mgt.DefaultWebSecurityManager"*>  <!-- 缓存 -->  <property name=*"cacheManager"* ref=*"cacheManager"* />  <!-- rememberMeManager -->  <property name=*"rememberMeManager"* ref=*"rememberMeManager"*></property>  <!-- 认证器 -->  <property name=*"authenticator"* ref=*"authenticator"*></property>  <!-- RealM -->  <property name=*"realms"*>  <!-- 指定使用哪些RealM,注意要对应 -->  <list>  <ref bean=*"jdbcRealm"* />  <ref bean=*"secondRealm"* />  </list>  </property>  <!--  sessionManager,如果要将session持久化,那么需要ref="sessionManager"  注:其余的辅助配置在下面已经配置好了  -->  <property name=*"sessionManager"* ref=*"sessionManager"*></property>  </bean>  <!-- 缓存 -->  <!-- 配置 CacheManager. 注:需要加入 ehcache的 jar包及相关配置文件. -->  <bean id=*"cacheManager"*  class=*"org.apache.shiro.cache.ehcache.EhCacheManager"*>  <!-- 指定配置文件 -->  <property name=*"cacheManagerConfigFile"*  value=*"classpath:ehcache.xml"* />  </bean>    <!-- rememberMeCookie配置-->  <bean id=*"rememberMeCookie"* class=*"org.apache.shiro.web.servlet.SimpleCookie"*>  <!-- 设置Cookie在客户端浏览器中保存内容的名字，由用户自己来设置 -->  <constructor-arg value=*"rememberMe"*/>  <!-- 保证该系统不会受到跨域的脚本操作攻击 -->  <property name=*"httpOnly"* value=*"true"*/>  <!-- 定义该Cookie的过期时间,单位为秒。如果设置为-1标识浏览器关闭就失效 -->  <property name=*"maxAge"* value=*"60"*/>  </bean>    <!-- rememberMe管理器-->  <bean id=*"rememberMeManager"*  class=*"org.apache.shiro.web.mgt.CookieRememberMeManager"*>  <!-- RememberMe对应加密后的Cookie的cipherKey -->  <property name=*"cipherKey"*  value=*"#{T(org.apache.shiro.codec.Base64).decode('4AvVhmFLUs0KTA3Kprsdag==')}"* />  <property name=*"cookie"* ref=*"rememberMeCookie"*/>  </bean>  <!-- 认证器 -->  <bean id=*"authenticator"*  class=*"org.apache.shiro.authc.pam.ModularRealmAuthenticator"*>  <property name=*"authenticationStrategy"*>  <!-- 在这里配置的 认证策略 -->  <bean class=*"org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy"*></bean>  <!-- <bean class="org.apache.shiro.authc.pam.FirstSuccessfulStrategy"></bean> -->  <!-- <bean class="org.apache.shiro.authc.pam.AllSuccessfulStrategy"></bean> -->  </property>  </bean>  <!-- 配置 Realm -->    <!-- MD5加密 -->  <bean id=*"jdbcRealm"* class=*"com.aspire.shiro.realms.ShiroRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <!-- 设置加密算法为MD5 -->  <property name=*"hashAlgorithmName"* value=*"MD5"*></property>  <!-- 设置加密迭代次数为1024次 -->  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- SHA1加密 -->  <bean id=*"secondRealm"* class=*"com.aspire.shiro.realms.SecondRealm"*>  <property name=*"credentialsMatcher"*>  <bean class=*"org.apache.shiro.authc.credential.HashedCredentialsMatcher"*>  <property name=*"hashAlgorithmName"* value=*"SHA1"*></property>  <property name=*"hashIterations"* value=*"1024"*></property>  </bean>  </property>  </bean>    <!-- 配置 LifecycleBeanPostProcessor. 可以自动调用配置在 Spring IOC 容器中 shiro  bean 的生命周期方法. -->  <bean id=*"lifecycleBeanPostProcessor"*  class=*"org.apache.shiro.spring.LifecycleBeanPostProcessor"* />    <!-- 配置一个 bean, 该 bean 实际上是一个 Map. 通过实例工厂方法的方式 -->  <!-- 指定编写的配置信息类 -->  <bean id=*"filterChainDefinitionMapBuilder"*  class=*"com.aspire.shiro.config.FilterChainDefinitionMapBuilder"*></bean>  <!-- 指定用哪个方法“加工” -->  <bean id=*"filterChainDefinitionMap"*  factory-bean=*"filterChainDefinitionMapBuilder"* factory-method=*"buildFilterChainDefinitionMap"*>  </bean>      <!-- bean注入 -->  <bean id=*"shiroServiceBean"*  class = *"com.aspire.shiro.service.ShiroService"*></bean>  <!--  配置 ShiroFilter.  注:id必须和 web.xml 文件中配置的org.springframework.web.filter.DelegatingFilterProxy  的 <filter-name> 一致。  -->  <bean id=*"shiroFilter"*  class=*"org.apache.shiro.spring.web.ShiroFilterFactoryBean"*>  <property name=*"securityManager"* ref=*"securityManager"* />  <!-- 当Shiro验证时,如果不满足认证条件,那么打回到这个登录页面 -->  <property name=*"loginUrl"* value=*"/login.jsp"* />  <!-- 当Shiro认证(登陆)成功后,要跳转到的页面 -->  <property name=*"successUrl"* value=*"/list.jsp"* />  <!-- 当Shiro认证(登陆)成功后,如果不满足权限条件,那么跳转至此页面 -->  <property name=*"unauthorizedUrl"* value=*"/unauthorized.jsp"* />  <!-- 在过滤器中引入刚才的配置 -->  <property name=*"filterChainDefinitionMap"* ref=*"filterChainDefinitionMap"*></property>  </bean>    <!-- Session ID 生成器-->  <bean id=*"sessionIdGenerator"*  class=*"org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator"*/>    <!-- Session DAO. 继承 EnterpriseCacheSessionDAO -->  <bean id=*"sessionDAO"*  class=*"com.aspire.shiro.realms.MySessionDao"*>  <!-- 缓存引用ehcache.xml中配置的shiro-activeSessionCache -->  <property name=*"activeSessionsCacheName"* value=*"shiro-activeSessionCache"*/>  <!-- EnterpriseCacheSessionDAO需要Session ID 生成器,我们引用上面配置的生成器 -->  <property name=*"sessionIdGenerator"* ref=*"sessionIdGenerator"*/>  </bean>    <!-- 会话管理器-->  <bean id=*"sessionManager"* class=*"org.apache.shiro.web.session.mgt.DefaultWebSessionManager"*>  <!-- session失效时间30分钟 -->  <property name=*"globalSessionTimeout"* value=*"1800000"*/>  <!-- 删除失效了的session -->  <property name=*"deleteInvalidSessions"* value=*"true"*/>  <!-- 定时检查session是否失效 -->  <property name=*"sessionValidationSchedulerEnabled"* value=*"true"*/>  <!-- 指定引用上面配置的session -->  <property name=*"sessionDAO"* ref=*"sessionDAO"*/>  </bean>    <!-- 配置数据源 -->  <bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"* >  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>  <property name=*"url"*  value=*"jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=utf-8"*/>  <property name=*"username"* value=*"root"*/>  <property name=*"password"* value=*"pass"*/>  </bean>  </beans> |

SpringMVC的配置文件spring-servlet.xml



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>    <mvc:annotation-driven></mvc:annotation-driven>  <mvc:default-servlet-handler />  <!-- 指定SpringMVC解析哪些位置下的注解 -->  <context:component-scan  base-package=*"com.aspire.shiro"*></context:component-scan>  <!-- 对模型视图名称的解析 -->  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/"*></property>  <property name=*"suffix"* value=*".jsp"*></property>  </bean>    <!-- 使IOC容器中的bean可以使用 shiro的注解. 注:前提是已经配置了 LifecycleBeanPostProcessor -->  <!-- 注: LifecycleBeanPostProcessor配置在了Spring中,SpringMVC可以获取到Spring中的配置 -->  <bean  class=*"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"*  depends-on=*"lifecycleBeanPostProcessor"* />    <bean  class=*"org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"*>  <property name=*"securityManager"* ref=*"securityManager"* />  </bean>  </beans> |

缓存配置文件ehcache.xml



给出文字版:

|  |
| --- |
| <ehcache>  <diskStore path=*"java.io.tmpdir"*/>    <cache name=*"authorizationCache"*  eternal=*"false"*  timeToIdleSeconds=*"3600"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  statistics=*"true"*>  </cache>  <cache name=*"authenticationCache"*  eternal=*"false"*  timeToIdleSeconds=*"3600"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  statistics=*"true"*>  </cache>    <cache name=*"shiro-activeSessionCache"*  eternal=*"false"*  timeToIdleSeconds=*"3600"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  statistics=*"true"*>  </cache>    <defaultCache  maxElementsInMemory=*"10000"*  eternal=*"false"*  timeToIdleSeconds=*"120"*  timeToLiveSeconds=*"120"*  overflowToDisk=*"true"*  />  <cache name=*"sampleCache1"*  maxElementsInMemory=*"10000"*  eternal=*"false"*  timeToIdleSeconds=*"300"*  timeToLiveSeconds=*"600"*  overflowToDisk=*"true"*  />  <cache name=*"sampleCache2"*  maxElementsInMemory=*"1000"*  eternal=*"true"*  timeToIdleSeconds=*"0"*  timeToLiveSeconds=*"0"*  overflowToDisk=*"false"*  />  </ehcache> |

日志配置文件logback.xml



给出文字版:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- scan="true"开启对配置信息的自动扫描(默认时间为60秒扫描一次) 注:当此文件的配置信息发生变化时,此设置的作用就体现出来了,不需要重启服务 -->  <configuration scan=*"true"*>  <!-- 通过property标签,来存放key-value数据,便于后面的动态获取,提高程序的灵活性 -->  <property name=*"logFilePositionDir"*  value=*"C:/Users/dengshuai.ASPIRE/Desktop/log"* />  <!-- >>>>>>>>>>>>>>>>>>>>>>>>>配置appender(可以配置多个)>>>>>>>>>>>>>>>>>>>>>>>>> -->  <!--  name:自取即可,  class:加载指定类(ch.qos.logback.core.ConsoleAppender类会将日志输出到>>>控制台),  patter:指定输出的日志格式  -->  <appender name=*"consoleAppender"*  class=*"ch.qos.logback.core.ConsoleAppender"*>  <encoder>  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}:%L- %msg%n  </pattern>  </encoder>  </appender>  <!--  name:自取即可,  class:加载指定类(ch.qos.logback.core.rolling.RollingFileAppender类会将日志输出到>>>指定的文件中),  patter:指定输出的日志格式 file:指定存放日志的文件(如果无,则自动创建) rollingPolicy:滚动策略>>>每天结束时，都会将该天的日志存为指定的格式的文件  FileNamePattern：文件的全路径名模板 (注:如果最后结尾是gz或者zip等的话,那么会自动打成相应压缩包)  -->  <appender name=*"fileAppender"*  class=*"ch.qos.logback.core.rolling.RollingFileAppender"*>  <!-- 把日志文件输出到:项目启动的目录下的log文件夹(无则自动创建)下 -->  <file>log/logFile.log</file>  <!-- 把日志文件输出到:name为logFilePositionDir的property标签指定的位置下 -->  <!-- <file>${logFilePositionDir}/logFile.log</file> -->  <!-- 把日志文件输出到:当前磁盘下的log文件夹(无则自动创建)下 -->  <!-- <file>/log/logFile.log</file> -->  <rollingPolicy  class=*"ch.qos.logback.core.rolling.TimeBasedRollingPolicy"*>  <!-- TimeBasedRollingPolicy策略会将过时的日志，另存到指定的文件中(无该文件则创建) -->  <!-- 把因为 过时 或 过大 而拆分后的文件也保存到目启动的目录下的log文件夹下 -->  <fileNamePattern>log/logFile.%d{yyyy-MM-dd}.%i.log  </fileNamePattern>  <!-- 设置过时时间(单位:<fileNamePattern>标签中%d里最小的时间单位) -->  <!-- 系统会删除(分离出去了的)过时了的日志文件 -->  <!-- 本人这里:保存以最后一次日志为准,往前7天以内的日志文件 -->  <MaxHistory>  7  </MaxHistory>  <!-- 滚动策略可以嵌套；  这里嵌套了一个SizeAndTimeBasedFNATP策略，  主要目的是: 在每天都会拆分日志的前提下，  当该天的日志大于规定大小时，  也进行拆分并以【%i】进行区分，i从0开始  -->  <timeBasedFileNamingAndTriggeringPolicy  class=*"ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"*>  <maxFileSize>5MB</maxFileSize>  </timeBasedFileNamingAndTriggeringPolicy>  </rollingPolicy>  <encoder>  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}:%L- %msg%n  </pattern>  </encoder>  </appender>  <!-- >>>>>>>>>>>>>>>>>>>>>>>>>>>>>使用appender>>>>>>>>>>>>>>>>>>>>>>>>>>>>> -->  <!-- 指定[哪个包]下使用[哪个appender],并[设定日志级别] -->  <!-- 日志级别: trace|debug|info|warn|error|fatal -->  <logger name=*"com"* level=*"debug"*>  <!-- 指定使用哪个appender -->  <appender-ref ref=*"fileAppender"* />  </logger>  <!-- root:logger的根节点,appender-ref:确定使用哪个appender,将日志信息显示在console -->  <!--  注:如果不指定appender或则指定的是输出到文件的appender的话,  那么SpringBoot启动后,将不会在console打印任何信息  -->  <root level=*"info"*>  <appender-ref ref=*"consoleAppender"* />  </root>  </configuration> |