本教程讲述shiro安全框架。

# 1. 初始系统的权限

## 1.1 权限概述

几乎每个系统都要控制应用的访问权限，因为系统必须对用户的访问进行控制，不同角色的人员的功能是不同的。权限系统需要关注下面两个功能：

（1）认证（Authentication）：系统提供的用于识别用户身份的功能。这就是登录所做的功能，即让系统知道“你是谁”。

（2）授权（Authorization）：系统提供的赋予用户访问某个功能的能力，即对用户授权使用某功能，让系统知道“你能做什么”。

认证Authentication和授权Authorization两个英文单词很相似，注意区别。

## 1.2 常见的权限控制方式

（1）拦截URL进行权限控制。

B/S架构的系统基本上都是基于HTTP请求和响应的，因此只要能拦截用户访问的URL，就能控制用户的权限。例如我们之前在系统中配置的过滤器就达到了这样的效果。

（2）标识方法进行权限控制。

这种方式不关心HTTP请求的部分，而是关心应用的Controller部分。如果我们能控制每个Controller方法的访问权限，也就能控制用户的权限。比如，可以在Controller方法上使用注解标识该方法执行所需要的权限。例如在方法上使用“@RequirePrevilege(name = "staff.delete")”，就表示此方法的执行必须要有“staff.delete”这个权限，否则将出现异常。

上述所描述的功能的具体实现实际上需要由“代理模式”来实现。即：当为Controller中方法加上注解后，用户访问的Controller实际为目标Controller的代理对象，代理对象先负责权限的检查，如果检查通过则利用反射执行目标Controller的方法，而如果检查不通过，则抛出异常。

## 1.3 权限模块的数据模型

很多系统的权限模块数据模型基本上都是类似和固定的，一般由如下几张数据库表所构成：

（1）用户表（t\_user）；

（2）角色表（auth\_role）；

（3）用户角色表（user\_role）；

（4）权限表（auth\_privilege）；

（5）角色权限表（role\_privilege）。

其中角色的引入就是为了方便分配权限。

这部分的一些内容在“Java EE Primary -> Part1\_Servlet开发Java Web -> 8.项目要点”教程中已经简单介绍过，其中还提到了粗粒度的权限控制和细粒度的权限控制，可以回顾一下。这样设计的权限模块是比较简单通用的，做特定的系统时，可能要设计的更加复杂，要考虑更多的问题，这是一个大模块。

如果我们自行设计开发权限模块，可以通过过滤器Filter来完成，但是如果要真正做好一个权限模块，一些细节还是比较复杂的，例如细粒度的权限控制、根据URL对应的权限进行拦截、不拦截规定好的一些资源（例如登录页面、商城首页、静态资源CSS和JS等）、菜单和按钮的可见性、权限不足的提示页面、未通过认证再跳转到首页等。

因此下面介绍Java中的一个安全框架shiro，它能很好地进行应用的权限管理，使用方便，框架小巧且功能强大，能满足应用权限模块的要求，很多公司也会采用该安全框架以便提高工作效率。

# 2 shiro简介

## 2.1 关于shiro

shiro是apache开发的一套安全框架，目的是为了简化Java应用系统中权限功能的开发。shiro的官方地址为“http://shiro.apache.org/”，官网是这样介绍shiro的：

“Apache Shiro is a powerful and easy-to-use Java security framework that performs authentication, authorization, cryptography, and session management. With Shiro’s easy-to-understand API, you can quickly and easily secure any application – from the smallest mobile applications to the largest web and enterprise applications.”

我们可以利用shiro做例如下述的功能：

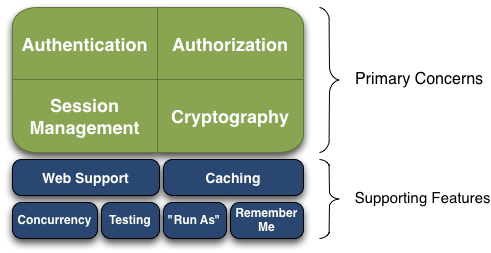
（1）验证用户，对用户执行访问控制（判断用户是否有权限）；

（2）可以使用多个用户数据源，例如一个是Oracle用户库，另一个是MySQL用户库。

（3）单点登录（SSO，Single Sign On），SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

## 2.2 shiro的四大部分

先看官方文档中的这张图：



其中绿色部分描述了shiro的四个主要部分：

（1）身份验证Authentication，简称“登录”。

（2）授权Authorization，即给用户分配权限角色或权限资源。

（3）会话管理Session Management，用户session管理器。

（4）加密Cryptography，把JDK中复杂的加密方式进行封装。

除此之外，shiro还提供了其他特性的支持，即：

（1）Web Support：针对Web应用提供支持；

（2）Caching：提供缓存支持；

（3）Concurrency：多线程的支持；

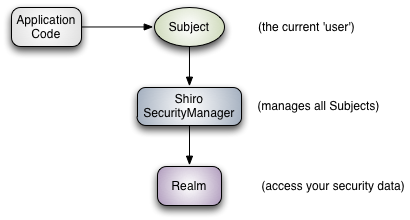
（4）Testing：测试相关功能；

（5）“Run As”：允许用户假设为另一个用户身份（如果允许）的功能；

（6）Remember Me：记住用户身份，提供类似购物车的功能。

## 2.3 shiro的运行流程

下图展示了shiro的运行流程：



箭头的指向顺序可以表示程序中的调用顺序，解释如下：

（1）Application Code：应用程序代码，即开发人员编写的应用代码，可调用“Subject”。

（2）Subject：主体。Subject是与程序进行交互的对象，可以是用户和服务等，通常就理解为用户。

（3）SecurityManager：安全管理器，这是shiro框架的核心对象，用于管理各个组件。

（4）Realm：负责访问安全数据（即用户、角色和权限等这些权限模型的数据）。Realm是应用程序和安全数据的“桥梁”，它获取安全数据以判断Subject所有的权限，Realm有点类似于DAO。

下图就展示了应用程序调用Subject，以及SecurityManager中管理的详细内容：



在Security Manager中，管理了很多内容，包括Session Manager、Authorizer、Realms等。

其中，Realms中提供了一些常用的Realm来连接安全数据的数据源，如常用的JDBC数据源、LDAP数据源等，因为有些安全数据不是存放在数据库中的。

另外，Cryptography中提供的API大幅简化了Java API中繁琐的加密功能；CacheManager缓存管理器，可以通过缓存减少不必要的数据访问，提高应用效率。

# 3. shiro的快速使用和运行

在shiro下载页面，提供了shiro全功能包的下载和Maven坐标，同时也提供了shiro各个功能模块的下载和Maven坐标。推荐需要使用什么模块就在项目中引用哪些模块。

需要说明的是，shiro可以在任何Java程序中使用，但这里考虑到实用性，只在Web项目中讲如何以Spring集成的方式使用shiro。我们基于前一章教程中的“ssm\_demo”进行开发，并将项目名称改为“ssm\_shiro\_demo”。

步骤1：我们基于之前做的“ssm\_demo”示例项目来加上shiro框架的使用，先利用Maven引入shiro的类库，暂时只需要引入如下类库（核心库、与Web继承的库和与Spring继承的库）：

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-core</**artifactId**>  <**version**>1.3.2</**version**> </**dependency**> <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-web</**artifactId**>  <**version**>1.3.2</**version**> </**dependency**> <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-spring</**artifactId**>  <**version**>1.3.2</**version**> </**dependency**> |

步骤2：在web.xml中配置一个SpringWeb中提供的一个过滤器“DelegatingFilterProxy”，稍后我们将知道该配置目的在于初始化shiro框架，因此我们将该过滤器命名为“shiroFilter”：

|  |
| --- |
| *<!-- 配置shiro和Spring整合的过滤器 -->* <**filter**>  <**filter-name**>shiroFilter</**filter-name**>  <**filter-class**>org.springframework.web.filter.DelegatingFilterProxy</**filter-class**>  *<!-- 本filter中配置的参数 -->* <**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**> |

但是这样启动后，Web应用程序报这样一个错误导致启动不了：

|  |
| --- |
| org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'shiroFilter' available |

这是因为配置了该过滤器后，Spring要在Spring容器中找到一个名为“shiroFilter”的bean（即bean名和过滤器名相同），该bean就作为过滤器。因此下面我们要在Spring配置文件中配置这样一个bean，而该bean用于初始化shiro并提供安全功能。还需要说明的是，“DelegatingFilterProxy”并不是只能整合shiro的过滤器，它只不过会把该过滤器和Spring中的bean关联起来，详情需要细说，这个涉及到Spring和Web的一些问题，见下面的附录“DelegatingFilterProxy的说明”一章，强烈建议学习，其中也讲了filter中配置参数的含义。

步骤3：配置一个id为“shiroFilter”的bean，在这里，该bean的类型配置为“ShiroFilterFactoryBean”类型，用于创建shiro框架用到的过滤器：

|  |
| --- |
| *<!-- 配置一个工厂bean，用于创建shiro框架用到的过滤器 -->* <**bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean"**>  *<!-- 注入安全过滤器 -->* <**property name="securityManager" ref="securityManager"** />  *<!-- 注入系统的登录页面，这样shiro能自动跳转到首页 -->* <**property name="loginUrl" value="/login.jsp"** />  *<!-- 注入成功页面 -->* <**property name="successUrl" value="/index.jsp"** />  *<!-- 注入权限不足页面 -->* <**property name="unauthorizedUrl" value="/unauthorize.jsp"** />  *<!-- 注入URL拦截规则 -->* <**property name="filterChainDefinitions"**>  <**value**>  /\*\* = authc  </**value**>  </**property**> </**bean**> *<!-- 配置shiro在Web中提供的默认安全管理器bean，上述配置引用该bean -->* <**bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager"** />  *<!-- 最后配置的LifecycleBeanPostProcessor这个的bean的作用是管理shiro中一些bean的生命周期。这是shiro的生命周期管理器 -->* <**bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"**/> |

其中说明如下：

上述在“shiroFilter”这个bean中注入了安全管理器、登录页面、成功页面、权限不足页面、URL拦截规则等这些属性，其中安全管理器和URL拦截规则解释如下：

（1）在注入安全管理器SecurityManager时引用了“securityManager”，因此最后配置了securityManager这个bean。SecurityManager是一个接口，这里使用了一个默认的安全管理器实现类DefaultWebSecurityManager。

（2）在配置URL拦截规则时，为了简单能马上看到效果，简单地配置成了“/\*\* = authc”，意思是“所有路径都使用authc过滤器进行拦截处理”。“authc”是shiro中的一个过滤器名称，它的作用是检查当前用户是否已经认证过了，如果未认证，则跳转到登录页面。shiro中还有其他的过滤器，如下表：

|  |  |  |
| --- | --- | --- |
| 过滤器名称 | 过滤器类（省略了包名） | 说明 |
| anon | AnonymousFilter | 匿名访问过滤器，表示不登录即可访问（例如商城首页等） |
| authc | FormAuthenticationFilter | 认证过滤器 |
| anthcBasic | BasicHttpAuthenticationFilter |  |
| logout | LogoutFilter | 注销登录过滤器 |
| noSessionCreation | NoSessionCreationFilter |  |
| perms | PermissionsAuthorizationFilter | 权限认证过滤器，必须具有某权限才能访问 |
| port | PortFilter |  |
| rest | HttpMethodPermissionFilter |  |
| roles | RolesAuthorizationFilter | 角色认证过滤器，必须具有某角色才能访问 |
| ssl | SslFilter |  |
| user | UserFilter |  |

在配置中，“/\*\*”表示所有路径（“/\*”表示某一级路径，而“/\*\*”就表示所有路径了），这样的话，应用中所有的请求都会跳转到登录页面login.jsp，连登录都无法实现。但这里只是为了快速演示效果，实际中不会这么干。这样启动应用就会发现有上述说明的效果了，一个简单的shiro应用就完成了。

（3）配置中的权限不足页面等可自行新建JSP，其中有简单提示信息即可。

运行项目，我们就会发现，不管请求路径是什么，都会跳转到登录页面，因为任何请求都要认证，所以连登录都无法进行，但这说明我们使用shiro成功了，仅仅是我们现在的配置不符合实际情况而已。

# 4. shiro在项目中的应用

## 4.1 系统的权限模型

用户表已经在ssm\_demo中设计好了，现在要做的就是设计角色表、权限表和关联的两张表。目前设计如下：

（1）用户表（已设计好，user）：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 备注 |
| id | VARCHAR(40) | 主键 | 主键 |
| account | VARCHAR(40) | 无 | 账号 |
| username | VARCHAR(40) | 无 | 姓名 |
| password | VARCHAR(60) | 无 | 密码，MD5加密 |
| age | SMALLINT | 无 | 年龄 |
| gender | VARCHAR(20) | 无 | 性别 |

（2）角色表（role）：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 备注 |
| id | VARCHAR(40) | 主键 | 主键 |
| name | VARCHAR(40) | 无 | 角色名称 |
| keyword | VARCHAR(40) | 无 | 关键字（shiro用） |
| description | VARCHAR(60) | 无 | 角色描述 |

（3）用户角色表（user\_role）：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 备注 |
| user\_id | VARCHAR(40) | 主键，外键 | 用户编号 |
| role\_id | VARCHAR(40) | 主机，外键 | 角色编号 |

（4）权限表（privilege）：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 备注 |
| id | VARCHAR(40) | 主键 | 编号 |
| name | VARCHAR(40) | 无 | 名称 |
| keyword | VARCHAR(40) | 无 | 关键字（shiro用） |
| description | VARCHAR(60) | 无 | 描述 |
| url | VARCHAR(100) | 无 | 路径 |
| ismenu | BIT(1) | 无 | 是否生成菜单 |
| priority | INT | 无 | 优先级编号 |
| parentid | VARCHAR(40) | 无 | 父权限编号 |

上述的权限表为了实现动态显示用户的“功能菜单”，添加了下面三个属性，ismenu表示该功能是否生成菜单，因为有些功能是不要生成菜单的，比如一些细粒度的按钮等。优先级字段是为了对菜单进行排序，以确保用户每次进入功能页面时菜单是保持顺序的。“父权限编号”的作用是为了展示菜单的父子级关系（或者权限模块的关系，例如按钮是哪个权限模块的）。注意的是，如果一个用户有父权限，那么它将有全部子权限的功能。

（5）角色权限表（role\_privilege）：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 备注 |
| role\_id | VARCHAR(40) | 主键，外键 | 角色编号 |
| priv\_id | VARCHAR(40) | 主键，外键 | 权限编号 |

上述5张表的相关SQL如下：

|  |
| --- |
| *# 创建user表* **CREATE TABLE user**(  **id VARCHAR**(40) **PRIMARY KEY**,  **account VARCHAR**(40),  **username VARCHAR**(40),  **password VARCHAR**(60),  **age SMALLINT**,  **gender VARCHAR**(20) ); *# 添加一个用于登录的账户，密码用MD5加密* **INSERT INTO user VALUES** (1, **'zs'**, **'张三'**, *md5*(**'123456'**), 12, **'男'**);  *# 创建角色表* **CREATE TABLE** role(  **id VARCHAR**(40) **PRIMARY KEY**,  **name VARCHAR**(40),  **keyword VARCHAR**(40),  **description VARCHAR**(60) ); *# 插入测试数据* **INSERT INTO** role **VALUES**(1, **'管理员'**, **'admin'**, **'管理员角色'**); **INSERT INTO** role **VALUES**(2, **'普通会员'**, **'member'**, **'普通会员'**); *# 创建用户角色表* **CREATE TABLE** user\_role(  **user\_id VARCHAR**(40),  **role\_id VARCHAR**(40),  **PRIMARY KEY**(**user\_id**, **role\_id**) ); *# 添加外键约束* **ALTER TABLE** user\_role **ADD CONSTRAINT** user\_role\_userid\_fk **FOREIGN KEY**(**user\_id**) **REFERENCES user**(**id**); **ALTER TABLE** user\_role **ADD CONSTRAINT** user\_role\_roleid\_fk **FOREIGN KEY**(**role\_id**) **REFERENCES** role(**id**); *# 添加角色表数据(张三是普通会员)* **INSERT INTO** user\_role **VALUES**(1, 1); *# 权限表* **CREATE TABLE** privilege(  **id VARCHAR**(40) **PRIMARY KEY**,  **name VARCHAR**(40),  **keyword VARCHAR**(40),  **description VARCHAR**(60),  **url VARCHAR**(100),  **ismenu BIT**(1),  **priority INT**,  **parentid VARCHAR**(40) ); *# 添加外键* **ALTER TABLE** privilege **ADD CONSTRAINT** privilege\_self\_fk **FOREIGN KEY**(**parentid**) **REFERENCES** privilege(**id**); *# 添加数据(NULL表示无父节点)* **INSERT INTO** privilege **VALUES**(1, **'查看产品'**, **'viewProducts'**, **'查看产品功能'**, **'/product/getAllProducts.action'**, 1, **'1'**, **NULL**); *# 角色权限表* **CREATE TABLE** role\_privilege(  **role\_id VARCHAR**(40),  **priv\_id VARCHAR**(40),  **PRIMARY KEY**(**role\_id**, **priv\_id**) ); *# 添加外键* **ALTER TABLE** role\_privilege **ADD CONSTRAINT** role\_privilege\_roleid\_fk **FOREIGN KEY**(**role\_id**) **REFERENCES** role(**id**); **ALTER TABLE** role\_privilege **ADD CONSTRAINT** role\_privilege\_privid\_fk **FOREIGN KEY**(**priv\_id**) **REFERENCES** privilege(**id**); *# 添加数据(member角色有查看产品的权限)* **INSERT INTO** role\_privilege **VALUES**(2, 1); |

下面的项目示例中，将用到上述的表和数据。

## 4.2 设置URL拦截策略

首先，我们把上面的URL拦截请求改得合理一点：

|  |
| --- |
| *<!-- 注入URL拦截规则 -->* <**property name="filterChainDefinitions"**>  <**value**>  /css/\*\* = anon  /js/\*\* = anon  /images/\*\* = anon  /login.jsp\* = anon  /user/login.action = anon  /product/getAllProducts.action\* = perms[viewProducts]  /\*\* = authc  </**value**> </**property**> |

解释：

上述的设置表示：css、js、images等目录下的内容都不拦截，因为这是静态资源，需要放行。另外，/login.jsp、/user/login.action等页面也允许匿名访问，以便让用户登录系统，路径后带个“\*”表示该路径后加上参数也允许匿名访问，这个根据实际情况来。最后，我们设置了查看产品的URL必须要有“viewProducts”权限才能访问。在实际中，shiro怎么判断用户有没有该权限？因此后续我们会编写代码查出用户的权限（即表中的关键字字段，这对应着一个权限，可在系统中使用，因为关键字、权限编号和权限的URL都是一一对应的关系），然后与shiro交互，这样shiro就知道当前用户是否有该权限，进而进行拦截。

因此，权限表和角色表中的关键字keyword字段就是这样的作用，用于在程序中标记一个权限或角色，而表中的name字段只是为了在前端显示用的。比如上述使用了perms过滤器来规定“/product/getAllProducts.action\*”只能必须由有“viewProducts”权限的用户访问。同样，使用roles过滤器可设置允许访问的用户角色，例如：

“/product/getAllProducts.action\* = roles[member]”表示该路径可由“member”这个标记的角色访问。中括号中是可以加上引号的，但一般省略，例如：/product/delete.action = roles["admin"]。

至于在程序中如何实现，后续将具体讲解。

另外还要注意以下两点：

（1）上述的配置路径是先后顺序的，一个路径可能匹配到其中多个配置，shiro是从上往下匹配的，只要匹配成功就不再往下匹配了。

（2）对于那些仅仅用于服务器端转发的页面（例如WEB-INF下的页面资源），是并不需要在shiro中设置允许访问的权限的，这是因为这些资源并不是用户直接请求的（况且也请求不到），无需通过shiro配置权限，这些转发返回的资源能够正常返回（但是重定向就不同了）。

现在我们运行程序看看是什么样的效果。效果如下：

（1）由于登录的action放开了访问，因此可以成功进行登录。登录后服务器通过转发返回的页面是“funcpage.jsp”，该页面内容成功显示，然而我们并没有在shiro中配置该请求路径，说明了上述第2点描述正确（但如果用户自行访问funcpage.jsp页面是不行的，只会返回到登录页面，因为shiro没有认证通过，详见下述描述）。

（2）但是在登录后的功能页面点击“查看员工信息”时，却又跳转到了登录页面，这是为什么呢？用户登录后，访问此页面无权限应该显示“unauthorize.jsp”无权限页面啊？原来，虽然我们自己编写代码写了“登录”逻辑，但shiro根本不知道，对于shiro来说，用户根本没有进行认证过，即使该路径匹配的是“/product/getAllProducts.action\*”，由于用户都没认证，别谈有没有权限了，直接还是返回登录页面。这也解释了1的问题。

（3）访问其他无匹配的页面同样返回login.jsp页面，也是因为没有进行shiro中的认证。

既然使用了shiro，那么接下来，我们就要通过shiro认证用户，利用shiro完成“登录”，让shiro知道用户是谁，后续还要让shiro知道用户能干什么（即权限）。

## 4.3 使用shiro认证用户

通过4.1节所说，现在我们要通过shiro进行用户的认证。

此时，我们要重新编写UserController中的login方法（原来login中的代码废弃），使用shiro提供的方式进行认证。

具体的代码和解释如下：

|  |
| --- |
| *// 登录功能* @RequestMapping(**"login"**) **public** String login(HttpSession session, String account, String password) {  **try** {  *// 1. 获得当前用户对象(shiro提供的，即Subject对象)* Subject subject = SecurityUtils.*getSubject*(); *// 当前对象的状态是"未认证"。该Subject是一个接口类型  // 2. 调用subject的login方法进行验证  // 但login方法要提供一个token令牌对象。该对象类型是个接口，我们使用UsernamePasswordToken实现类创建该令牌  // 该令牌要传递用户名和密码，给它从表单传递过来的用户名和密码即可（密码要经过MD5加密）* AuthenticationToken token = **new** UsernamePasswordToken(account, MessageDigestUtils.*encrypt*(**"MD5"**, password.getBytes(), **false**));  *// 最后调用subject的login方法* subject.login(token);  *// 我们发现login方法无返回值，那么怎么知道登录认证是否成功呢？  // shiro通过异常来告知我们认证情况的。  // 如果认证不通过，会涉及到两个异常，一个是用户名不存在的异常，一个是密码错误的异常。  // 因此这里使用try catch进行处理。* **return "/funcpage.jsp"**;  } **catch** (Exception e) {  e.printStackTrace();  *// 认证不通过* **return "/login.jsp"**;  } } |

上述代码中只是调用了Subject，但从shiro的运行流程图中我们知道，Subject会去调用SecurityManager，然后SecurityManager调用Realm去查询安全数据（相当于DAO），这样最终通过Realm的结果来判断是否能认证成功。

因此写完上述代码还不够，我们还要编写Realm来访问安全数据（即自己设计的权限模型数据库中的数据），之前我们仅仅是使用了默认的安全管理器，并没有引入Realm，这样是不行的，运行程序进行认证时会报错的。

因此，首先在applicationContext.xml的默认安全管理器配置中，注入Realm。虽然shiro提供了一些Realm的实现（如JdbcReaml），但由于我们采用的是Hibernate、Mybatis等框架，因此这里直接编写自定义的Realm，例如“SSMRealm”类，并且在Spring中也管理这个bean，这样安全管理器和Realm配置如下（shiro过滤器配置无需改动）：

|  |
| --- |
| *<!-- 注入自定义Realm -->* <**bean id="ssmRealm" class="com.zhang.shiro.SSMRealm"** /> *<!-- 配置默认的安全管理器，并引用上述的realm. -->* <**bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager"**>  <**property name="realm" ref="ssmRealm"** /> </**bean**> |

随后，我们在com.zhang.shiro包中编写SSMRealm类。参考JdbcRealm，SSMRealm也要继承“AuthorizingRealm”类，需要实现两个方法，即：

|  |
| --- |
| **package** com.zhang.shiro;  **import** org.apache.shiro.authc.AuthenticationException; **import** org.apache.shiro.authc.AuthenticationInfo; **import** org.apache.shiro.authc.AuthenticationToken; **import** org.apache.shiro.authz.AuthorizationInfo; **import** org.apache.shiro.realm.AuthorizingRealm; **import** org.apache.shiro.subject.PrincipalCollection;  **public class** SSMRealm **extends** AuthorizingRealm { @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {**return null**;  } @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  **return null**;  } } |

其中，doGetAuthenticationInfo是认证方法，doGetAuthorizationInfo是授权方法，注意区分！

在进行认证时，Subject最终会通过安全管理器调用到这个Realm中的doGetAuthenticationInfo方法，其中的token参数就是我们在subject中设置的token，经过一步步地执行传递了过来。

如果doGetAuthenticationInfo认证方法返回null，则安全管理器会认为当前用户名不存在，因此会抛出UnknownAccountException异常。因此如果我们现在什么都不编写，启动应用进行登录的话，Controller中就能输出UnknownAccountException异常，表示用户名不存在。

那么doGetAuthenticationInfo到底是怎么认证的呢？shiro的认证过程是这样的：我们先要在doGetAuthenticationInfo中根据账号查出用户信息（账号通过token已经传递过来了），把用户信息中的密码交给安全管理器进行认证即可。由安全管理器负责比对数据库中查询出的密码和用户提交的密码（用户比较的密码在token中，安全管理器本身就可访问，不用再传递token给安全管理器）。

由于认证中需要查询数据库，即会用到DAO，因此SSMRealm中也需引用DAO。SSMRealm中的认证代码和解释如下：

|  |
| --- |
| **package** com.zhang.shiro;  **import** com.zhang.domain.User; **import** com.zhang.mapper.UserMapper; **import** org.apache.shiro.authc.\*; **import** org.apache.shiro.authz.AuthorizationInfo; **import** org.apache.shiro.realm.AuthorizingRealm; **import** org.apache.shiro.subject.PrincipalCollection;  **import** javax.annotation.Resource;  **public class** SSMRealm **extends** AuthorizingRealm {   *// 引用UserMapper* @Resource  **private** UserMapper **userMapper**;   *// 认证的方法* @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {  *// 1. 先把token强换成UsernamePasswordToken类型以使用，本来就是这个类型。* UsernamePasswordToken upToken = (UsernamePasswordToken) token;  *// 2. 通过token中的账号（调用getUsername方法）查询用户信息* User user = **userMapper**.getUserByAccount(upToken.getUsername());  *// 3. 判断user是否存在，存在的话就进行认证* **if** (user == **null**) {  **return null**;  } **else** {  *// 认证。需要把需要的信息包装成AuthenticationInfo并返回  // 这里使用SimpleAuthenticationInfo实现类  // 其中有三个参数：  // 参数1：签名。可传递任意对象，程序可以在任意位置访问当前放入的对象。这里就把User对象放进去，方便使用。  // 参数2：凭证。即把查询的密码传递进去即可。安全管理器负责比对。  // 参数3：当前realm的名字，传递类名即可，为了区分应用中的不同realms，虽然本应用中只有一个realm。* **return new** SimpleAuthenticationInfo(user, user.getPassword(), **this**.getClass().getSimpleName());  }  }   *// 授权的方法* @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  **return null**;  } } |

最终，认证方法返回AuthenticationInfo给安全管理器，由安全管理器进行认证。安全管理器比对成功了就会修改当前的subject状态为已认证，否则会抛出用户不存在或者是密码错误这两种异常。

需要注意的是，我们把User作为签名传递进去SimpleAuthenticationInfo，认证以后就能在任意地方拿到该对象使用了，十分方便。例如在login方法中就需要拿到这个User对象并放在session域中，以便登录成功后，页面能显示用户信息。示例代码如下：

|  |
| --- |
| *// 登录功能* @RequestMapping(**"login"**) **public** String login(HttpSession session, String account, String password) {  **if** (account == **null** || password == **null**) {  **return "/login.jsp"**;  }  **try** {  *// 进行认证的代码（之前已解释过）* Subject subject = SecurityUtils.*getSubject*(); *// 当前对象的状态是"未认证"。* AuthenticationToken token = **new** UsernamePasswordToken(account, MessageDigestUtils.*encrypt*(**"MD5"**, password.getBytes(), **false**));  subject.login(token);  *// 认证成功后，则可以获取认证信息对象中存储的User对象。* User user = (User) subject.getPrincipal(); *// 通过subject的getPrincipal方法拿到对象，类型是Object，强转即可* session.setAttribute(**"user"**, user);  **return "/funcpage.jsp"**;  } **catch** (Exception e) {  e.printStackTrace();  *// 认证不通过* **return "/login.jsp"**;  } } |

可以看出，以后只要在任意位置通过subject的getPrincipal方法拿到当前登录的用户对象就能使用，当然也能通过session的方式获取。

现在再运行程序，发现登录正常，而且访问“/product/getAllProducts.action”页面时也提示了权限不足，说明我们编写正确。下面，就是要考虑进行授权了。

## 4.4 使用shiro进行授权

现在我们需要编写SSMRealm中的doGetAuthorizationInfo方法进行授权。同样，如果认证方法返回null，则安全管理器认为当前subject（用户）没有被授予任何权限，只能访问无需权限的页面，因此用户无法查看产品信息页面。如果要给用户授予权限，可以通过“SimpleAuthorizationInfo”类型对象授予权限并将此对象返回（SimpleAuthorizationInfo 是AuthorizationInfo 的一个实现类，对象还是会返回到安全管理器中）。可调用SimpleAuthorizationInfo对象的addStringPermission方法和addRole方法分别为当前用户添加某权限或某角色。

例如下述代码就为当前用户添加了“viewProducts”权限和“admin”角色：

|  |
| --- |
| **package** com.zhang.shiro;  **import** com.zhang.domain.User; **import** com.zhang.mapper.UserMapper; **import** org.apache.shiro.authc.\*; **import** org.apache.shiro.authz.AuthorizationInfo; **import** org.apache.shiro.authz.SimpleAuthorizationInfo; **import** org.apache.shiro.realm.AuthorizingRealm; **import** org.apache.shiro.subject.PrincipalCollection;  **import** javax.annotation.Resource;  **public class** SSMRealm **extends** AuthorizingRealm {   *// 引用UserMapper* @Resource  **private** UserMapper **userMapper**;   *// 认证的方法* @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {  *// 认证过程省略...前面已讲过* }  }   *// 授权的方法* @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  *// 方法的返回值类型是AuthorizationInfo，即授权信息对象  // 我们使用它的一个实现类SimpleAuthorizationInfo。* SimpleAuthorizationInfo info = **new** SimpleAuthorizationInfo();  *// 调用addStringPermission方法进行授予权限* info.addStringPermission(**"viewProducts"**); *// 为当前用户授予viewProducts权限  // 调用addRole方法授予角色* info.addRole(**"member"**); *// 为当前用户授予member角色。  // 最后返回info即可* **return** info;  } } |

这时我们重启应用，再查看产品页面就能访问了，即使把URL拦截改成：

|  |
| --- |
| /product/getAllProducts.action\* = roles[member] |

也能正常访问，因为我们给当前用户添加了member角色。

上述的代码是硬编码的，实际中我们应该查看数据库找出当前用户所拥有的角色和权限。

如果我们在授权方法中添加断点或输出，观察该方法何时执行，则会发现每次访问需要权限的页面时，都会执行该授权方法。如果以后查询数据库获取用户权限，这样做显然耗费资源，因此后续我们会讲shiro的缓存功能，能够把用户的权限进行缓存，减少对安全数据的访问。

# 5. shiro提供的权限控制方式

## 5.1 URL拦截权限控制

这就是我们前面所讲的控制方式，主要就是在Spring配置文件中配置好了URL及其对应的权限，即类似下述的配置：

|  |
| --- |
| *<!-- 注入URL拦截规则 -->* <**property name="filterChainDefinitions"**>  <**value**>  /css/\*\* = anon  /js/\*\* = anon  /images/\*\* = anon  /login.jsp\* = anon  /user/login.action = anon  /product/getAllProducts.action\* = perms[viewProducts]  /\*\* = authc  </**value**> </**property**> |

其他不再赘述。

## 5.2 方法注解权限控制

顾名思义，该种方式就是通过注解的方式在Controller方法上标识该方法执行所需的权限或角色。使用步骤如下：

（1）开启注解，需要在Spring中进行如下配置：

|  |
| --- |
| *<!-- 步骤1：让Spring自动创建代理。depend-on为我们之前配置的shiro生命周期管理器，表示该bean需要依赖shiro的生命周期管理器，即在lifecycleBeanPostProcessor这个bean创建后才会创建这个bean -->* <**bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" depends-on="lifecycleBeanPostProcessor"**>  *<!-- 该值为true表示强制使用CGLIB创建代理类 -->* <**property name="proxyTargetClass" value="true"** /> </**bean**> *<!-- 其实上述的配置相当于以下的配置： <aop:config proxy-target-class="true" /> 就是直接强制使用CGLIB代理，之前讲过 --> <!-- 步骤2：开启Shiro Spring AOP权限注解的支持 -->* <**bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"**>  *<!-- 引用安全管理器 -->* <**property name="securityManager" ref="securityManager"**/> </**bean**> |

为什么要强制使用CGLIB代理？因为Spring默认会使用JDK代理（接口代理），而Controller中并没有采用实现接口的方式，就算Controller使用了接口（例如在Struts中使用的DrivenModel等），也和Controller中的访问方法不对应，因此无法代理。所以这里最好强制设置为CGLIB代理。

另外注意如果使用“aop:config”的配置方式，则要在pom.xml中引入aspectj-weaver的jar包：

|  |
| --- |
| <**dependency**>  <**groupId**>org.aspectj</**groupId**>  <**artifactId**>aspectjweaver</**artifactId**>  <**version**>1.8.13</**version**>  <**scope**>compile</**scope**> </**dependency**> |

在切面类AuthorizationAttributeSourceAdvisor中，没有给bean提供id，因为我们并不需要引用它，shiro会自动通过类型获得该bean。

但是上述在applicationContext.xml中的配置并不会生效！这是因为applicationContext.xml文件中并没有扫描Controller层的注解，Controller的扫描是在springmvc.xm中控制的，因此我们将上面的配置移到springmvc.xml中即可。但是这样Service层就不能使用shiro的注解了（有时，远程调用时需要在Service层上做权限控制，例如WebService的SOAP调用时，就是在Service层），因此我们把这两个配置在applicationCotext.xml也保留一份即可，注意bean的id不要冲突即可。

接下来，我们就能在Controller的方法上使用shiro中提供的注解了，常用的有：

@RequiresAuthentication：表示方法的执行需要认证，相当于“authc”；

@RequiresPermissions("perms")：表示方法的执行需要“perms”权限；

@RequiresRoles("roles")：表示方法的执行需要“roles”权限。

例如：

|  |
| --- |
| **package** com.zhang.controller;  **import** com.zhang.service.IProductService; **import** org.apache.shiro.authz.annotation.RequiresAuthentication; **import** org.apache.shiro.authz.annotation.RequiresGuest; **import** org.apache.shiro.authz.annotation.RequiresPermissions; **import** org.springframework.stereotype.Controller; **import** org.springframework.web.bind.annotation.RequestMapping;  **import** javax.annotation.Resource; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpSession;  @Controller @RequestMapping(**"product"**) **public class** ProductController {  @Resource  **private** IProductService **productService**;   @RequestMapping(**"getAllProducts"**)  @RequiresPermissions(**"viewProducts"**)  **public** String getAllProducts(HttpServletRequest request) {  request.setAttribute(**"productList"**, **productService**.getAllProducts());  **return "/WEB-INF/product\_list.jsp"**;  } } |

表示执行getAllProducts方法必须要有viewProducts权限。

但这里要注意，使用注解控制时，如果用户没有权限的话（或没有角色），shiro不是直跳转到权限不足页面，而是抛出异常，直接显示“500”错误页面。原来是shiro使用注解时如果无权限（或无角色）就会报出UnauthorizedException异常。那我们不是已经配置过权限不足页面吗？这是因为前面配置的权限不足页面是针对URL拦截方式来说的。因此这里需要处理全局异常，即要捕获Controller的异常类型，并返回到友好页面。

这个功能在Struts功能较好实现，实际上在Spring中也能较好完成，只要在springmvc.xml中配置如下的异常处理即可：

|  |
| --- |
| *<!-- 处理异常 -->* <**bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver"**>  <**property name="exceptionMappings"**>  <**props**>  *<!-- 登录（认证不成功会抛出此异常） -->* <**prop key="org.apache.shiro.authz.UnauthenticatedException"**>  redirect:/login.jsp  </**prop**>  *<!-- 授权 -->* <**prop key="org.apache.shiro.authz.UnauthorizedException"**>  redirect:/unauthorize.jsp  </**prop**>  </**props**>  </**property**> </**bean**> |

使用注解后，关注的重点就在于控制Controller方法的访问了。

## 5.3 使用JSP页面标签控制权限

这种方式就是在JSP页面上，根据用户的权限选择显示内容。例如用户没有该权限，就不显示这个菜单或不显示这个按钮，而不是让用户点击了菜单后跳转到“权限不足”页面。

可使用shiro提供的shiro标签来完成这样的功能。首先在JSP页面中引入shiro标签库：

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

然后使用“shiro:hasPermission”和“shiro:hasRole”等标签来控制页面的显示，例如（funcpage.jsp页面）：

|  |
| --- |
| <**div**>欢迎你，**${**user.username**}**</**div**> <**ul**>  *<%-- 当用户有viewProducts这个权限时才显示其中的内容 --%>* <**shiro:hasPermission name="viewProducts"**>  <**li**>  <**a href="${**basePath**}/product/getAllProducts.action"**>查看产品信息</**a**>  </**li**>  </**shiro:hasPermission**> </**ul**> |

上述就是标签的使用。

## 5.4 使用代码来控制权限

shiro中也提供直接用代码来检查用户是否有某种权限（或角色），但是这种方式一般不使用，实际中前面三种方式结合使用即可。

使用代码控制权限时要调用subject的checkPermission()或checkRole()等方法检查权限，如果用户没有权限，同样会抛出UnauthorizedException异常。例如：

|  |
| --- |
| **public** String edit() {  Subject subject = SecurityUtils.*getSubject*();  subject.checkPermission(**"edit"**);  **return "/edit.jsp"**; } |

# 6. shiro缓存的使用

我们之前说实际中是通过数据库查询用户的所有权限（具体方法是在自定义Realm的授权方法中，查询出用户角色，再遍历角色中的权限，调用addStringPermission方法添加权限关键字以授权），因此最好通过shiro把用户的权限进行缓存，避免每次请求时都进行授权操作而频繁地查询数据库。

这里使用ehcache来做shiro中权限的缓存。

（1）在pom.xml引入shiro对ehcache的支持：

|  |
| --- |
| *<!-- shiro对ehcache的支持 -->* <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-ehcache</**artifactId**>  <**version**>1.3.2</**version**>  <**scope**>compile</**scope**> </**dependency**> |

（2）首先在Spring中配置shiro的缓存管理器，并注入到安全管理器中，例如：

|  |
| --- |
| *<!-- 配置shiro的缓存管理器 -->* <**bean id="ehCacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager"**>  *<!-- 注入ehcache的配置文件 -->* <**property name="cacheManagerConfigFile" value="classpath:ehcache.xml"** /> </**bean**> *<!-- 配置默认的安全管理器，并引用上述的realm. -->* <**bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager"**>  <**property name="realm" ref="ssmRealm"** />  *<!-- 注入缓存管理器 -->* <**property name="cacheManager" ref="ehCacheManager"** /> </**bean**> |

（3）由于上面指定了ehcache的配置文件为“classpath:ehcache.xml”，因此我们在resources资源目录中新建ehcache.xml文件，配置如下（其实和之前ehcache的使用都是类似的，下面只进行了默认的配置）：

|  |
| --- |
| <**ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"**>  <**diskStore path="java.io.tmpdir"** />  <**defaultCache  maxElementsInMemory="10000"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  overflowToDisk="true"  maxElementsOnDisk="10000000"  diskPersistent="false"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"** /> </**ehcache**> |

这样配置后shiro就有缓存功能了。运行应用无问题显现。

# 7. 总结

如果我们想动态地显示用户的功能菜单，即不同的用户登入页面，显示的菜单是不同的，那么有两种方式：

（1）登录后的JSP页面默认查询列出所有的功能（权限）菜单，然后每个菜单使用“shiro:hasPermission”包裹即可，这样就能控制菜单的显示。

（2）不使用shiro的功能，直接从数据库中查出本用户对应的所有功能菜单进行显示。

另总结：

（1）实际上权限是在系统开发好该权限的功能后才进行维护的，跟编码是相关的，因此“权限（功能）”在后台是做不到“动态”的。路径、关键字、权限名实际都是该权限的标识，使用哪个的效果都一样，只是在程序适合用关键字，在显示时适合用权限名等。

（2）本身shiro就比较轻量，只是安全框架，使用上基本的功能都好实现，而且框架考虑的比较全，一般不自己写过滤器。

（3）现在应该能体会到Spring框架的好处：各种资源一注入整合到Spring中，该框架就搭起来了，就可以马上使用了。Spring就像是老大、是中心，各种框架通过它配置资源并进行工作，使用灵活方便。想象一下，如果没有Spring进行管理，各个框架的使用可能都很麻烦。

（4）一些其他的、细节的知识需要查看shiro相关的资料深入学习，例如我们提供的和网上的。

# 8. 附录 DelegatingFilterProxy的说明

本章详细说明如下几个问题：为什么使用DelegatingFilterProxy；DelegatingFilterProxy中参数的作用。

## F.1 为什么使用DelegatingFilterProxy

在web.xml中配置shiro过滤器时，我们配置的是DelegatingFilterProxy，而DelegatingFilterProxy是Spring中提供的过滤器，本身上和shiro没有什么关系。那我们配置过滤器为什么不能直接配置和shiro相关的过滤器，而要配置DelegatingFilterProxy这个Spring中提供的过滤器呢？

如果你把本教程看完再回顾一下，可能会对你理解这些有帮助。因为shiro与Spring整合后，Spring也会管理shiro需要的对象。这样问题就来了，如果我们不使用DelegatingFilterProxy，而使用普通的“shiro过滤器”（姑且认为有该过滤器），那么Spring就无法将Spring中管理的对象注入到这个普通的过滤器中（注意这个表述，绝对不能按照网上说的“filter中不能注入bean”，而应该是“bean不能注入到filter中”！！！）。为什么呢？

这和Listener、Filter和Servlet的启动顺序有关，应当牢记它们的启动顺序是“理发师（LFS）”，即顺序是Listener、Filter和Servlet。我们在学习Spring时知道，Spring的生命周期是在ServletContext范围中，SpringIOC容器在ServletContextListener监听器创建时就创建了，接着就会创建好需要Spring管理的bean（即例如applicationContext.xml中描述的单例bean，如果没有设置延迟创建对象的话）。

也就是说SpringIOC容器及其中一些bean的创建会先于Filter过滤器创建，那么当Servlet容器创建好Filter后，Filter中是可以拿到SpringIOC中的bean的，通过以下方式就能拿到（之前讲Spring与Web整合原理时讲过，通过WebApplicationContextUtils工具类），例如：

|  |
| --- |
| **package** com.zhang.filter;  **import** com.zhang.service.IUserService; **import** org.springframework.context.ApplicationContext; **import** org.springframework.web.context.support.WebApplicationContextUtils;  **import** javax.servlet.\*; **import** javax.servlet.annotation.WebFilter; **import** java.io.IOException;  @WebFilter(urlPatterns = **"/\*"**) **public class** MyFilter **implements** Filter {  @Override  **public void** init(FilterConfig config) **throws** ServletException {  *// 获得Spring容器中的对象* ServletContext context = config.getServletContext();  ApplicationContext ac = WebApplicationContextUtils.*getWebApplicationContext*(context);  IUserService userService = (IUserService) ac.getBean(**"userService"**);  System.***out***.println(**"---普通过滤器拿到Spring对象---"**);  System.***out***.println(userService);  }   @Override  **public void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  chain.doFilter(request, response);  }   @Override  **public void** destroy() {  } } |

虽然Filter中能够拿到SpringIOC中的bean，但这显然不是我们想要的结果，因为上述都是根据指定的bean名称获得对象的。我们还是希望在applicationContext.xml配置文件中，能直接像以前一样，给过滤器注入需要的属性，且能引用SpringIOC中已有的bean，这对于shiro过滤器来说是非常重要的，当然对于其他一些需要这样功能的框架来说也同样重要。

而上述的需求无法直接实现，因为IOC容器早于Filter创建，而且Filter并不是Spring负责创建的。而要实现这样的功能，可能在每个过滤器中都要编写大量的代码来适应Spring以提供灵活性，这是不现实的。因此，Spring提供了DelegatingFilterProxy来解决这一问题。

例如这样的配置：

|  |
| --- |
| <**filter**>  <**filter-name**>myFilter</**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**>myFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**> </**filter-mapping**> |

应用启动时，DelegatingFilterProxy就会在SpringIOC中查找和filterName名相同的bean，并把这个bean作为“真正的过滤器”，通过代理的方式执行这个“真正过滤器”。DelegatingFilterProxy本身也是一个Filter，用户的请求路径如果匹配到DelegatingFilterProxy中配置的路径，则DelegatingFilterProxy负责把请求再委派给真正的过滤器去处理。

当然，DelegatingFilterProxy还会把真正的过滤器和Spring中的bean关联起来，这是很重要的，解决了我们上面提到的问题，这样就能在Spring配置文件中，创建bean的同时注入属性了，且能引用bean。

为了举例，我们把上面的“myFilter”过滤器写好。

（1）编写MyFilter过滤器类：

|  |
| --- |
| **package** com.zhang.filter;  **import** com.zhang.service.IUserService; **import** javax.servlet.\*; **import** java.io.IOException;  **public class** MyFilter **implements** Filter {   *// userService属性* **private** IUserService **userService**;   @Override  **public void** init(FilterConfig config) **throws** ServletException {  System.***out***.println(**"---普通过滤器拿到Spring对象---"**);  System.***out***.println(**userService**); *// 直接输出属性对象* }   @Override  **public void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  System.***out***.println(**"放行了"**);  chain.doFilter(request, response);  }   @Override  **public void** destroy() {  System.***out***.println(**"销毁了myFilter"**);   }   *// userService属性的getter和setter方法* **public** IUserService getUserService() {  **return userService**;  }   **public void** setUserService(IUserService userService) {  **this**.**userService** = userService;  } } |

（2）在Spring配置文件applicationContext.xml中配置myFilter这个bean（和web.xml中配置的DelegatingFilterProxy的过滤器名一致），并且在其中注入userService属性：

|  |
| --- |
| <**bean id="myFilter" class="com.zhang.filter.MyFilter"**>  <**property name="userService" ref="userService"** /> </**bean**> |

最后启动程序，则会发现我们的MyFilter中能输出userService对象，说明注入属性成功，且“放行”和“销毁”都执行成功，证明了上述所言。

当然，一个应用中可在web.xml中配置多个DelegatingFilterProxy，说白了就是通过DelegatingFilterProxy让Spring去管理Filter的生命周期，而不再是由Servlet容器去管理了！总之，DelegatingFilterProxy是一个代理类，拦截的请求会首先发到这个filter代理，然后再按照"filter-name"委派到spring中的这个bean过滤器去执行处理。

## F.2 DelegatingFilterProxy过滤器中配置的参数

之前，我们在DelegatingFilterProxy配置了这样的参数：

|  |
| --- |
| <**init-param**>  <**param-name**>targetFilterLifecycle</**param-name**>  <**param-value**>true</**param-value**> </**init-param**> |

在DelegatingFilterProxy中配置targetFilterLifecycle这个参数的含义是是否让Spring来管理“真正过滤器”中的init和destroy方法。如果设置为true，则init和destroy由Spring来管理执行，在运行时这两个方法会有效果，但如果设置为false，则init和destroy将失效，即不会执行，该参数默认就为false。如果我们把myFilter的该参数设置为false，则不会输出userService对象和“销毁了myFilter”的信息。

另外，DelegatingFilterProxy还可配置一个叫做“targetBeanName”的参数，使用该参数可指定“真正过滤器”的bean名称，不再是默认的过滤器名了。例如：

|  |
| --- |
| <**filter**>  <**filter-name**>myFilter</**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**>  <**init-param**>  <**param-name**>targetBeanName</**param-name**>  <**param-value**>yourFilter</**param-value**>  </**init-param**> </**filter**> |

则Spring中会去找“yourFilter”这个bean作为过滤器类。

最后，关于DelegatingFilterProxy具体是如何实现的，可自行看源码，或参考网址：

http://www.cnblogs.com/hzhuxin/archive/2011/12/19/2293730.html

其中的评论很精彩。