

# 吴老师教学讲义



忽然抚尺一下，群响毕绝。撤屏视之，一人、一桌、一椅、一扇、一抚尺而已



吴 青

QQ:16910735

wuqing\_bean@126.com

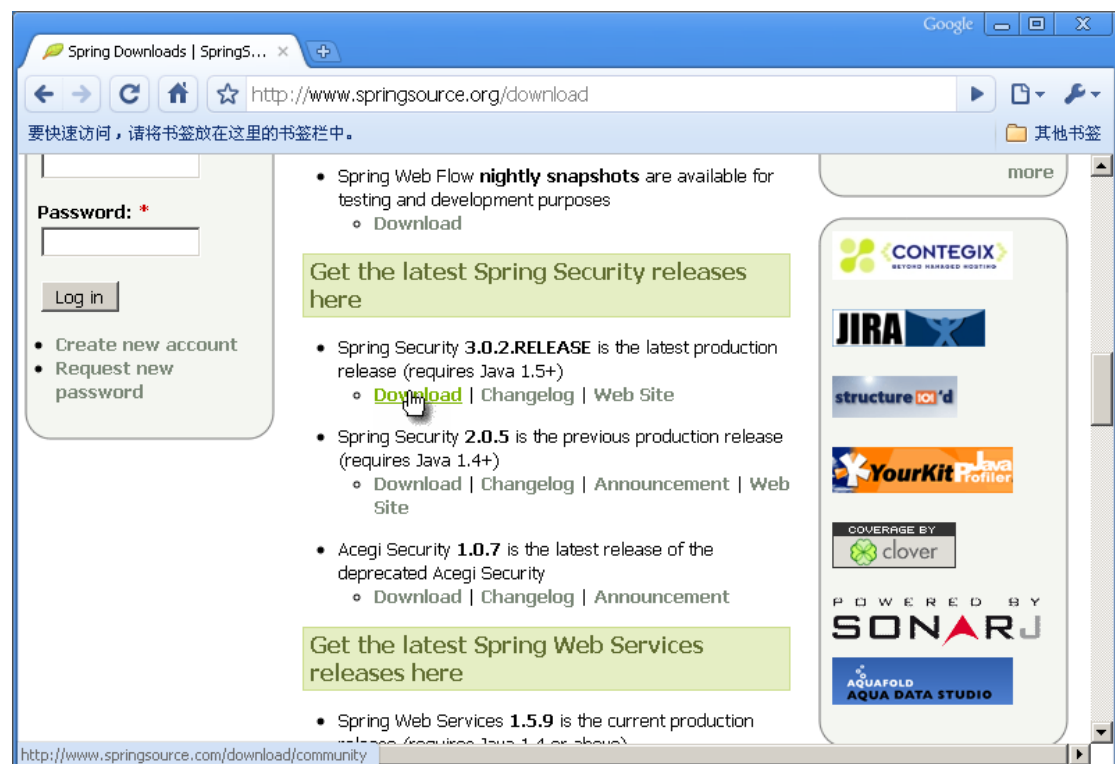
<http://blog.sina.om.cn/accpwulaoshi>

# Spring Security3.0

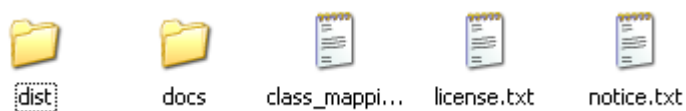
## 什么是 Spring Security?

Spring Security，这是一种基于 Spring AOP 和 Servlet 过滤器<sup>[7]</sup>的安全框架。它提供全面的安全性解决方案，同时在 Web 请求级和方法调用级处理身份确认和授权。在 Spring Framework 基础上，Spring Security 充分利用了依赖注入（DI，Dependency Injection）和面向切面技术。

## 获取 Spring Security 3.0



解压:

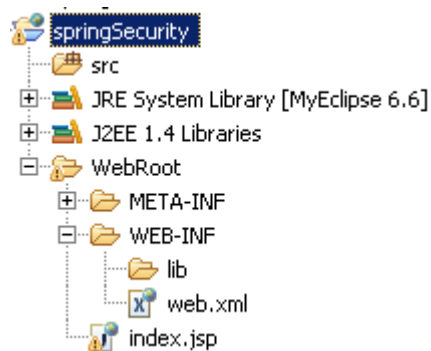


dist : 目录中存放发布包

docs : 存放文档

## 入门

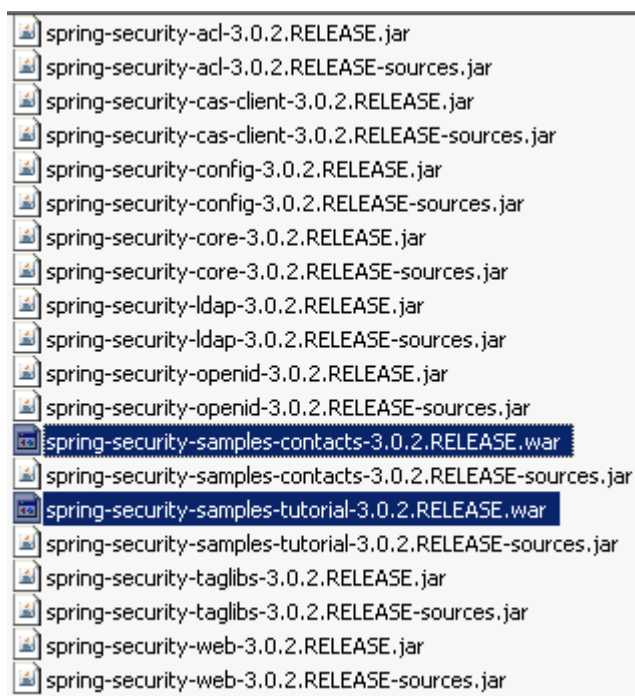
### 新建 web 工程



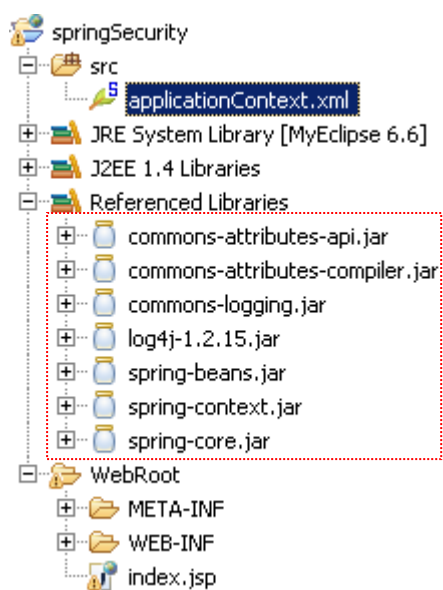
### 为工程添加 Spring 支持

我这里使用的是 MyEclipse6.6 版本，该版本中的 Spring 的版本是 Spring 2.5，而 Spring Security3.0 使用的是 Spring 3.0，这里加入 Spring 支持是为了让工具为我们自动添加 Spring 的配置文件。

Spring Security3.0.2 下载解压后的 dist 目录中有两个 war 包，这两个 war 包是示例程序



将其中的一个示例程序的 .war 扩展名改成.rar，将其解压出来，将压缩包中的 WEB-INF/lib 总的所有的 jar 包拷贝到我们的工程中，在拷贝之前先清除掉 MyEclipse 帮助我们加入的 jar 包。



将这些工具帮助我们加入的 jar 包全

部移除掉，然后加入 .war 中

添加后的结果:



几个包的含义:



spring-security-core-3.0.2.RELEASE.jar

:包含了核心认证和权限控制类和接口， 远程支持和基本供应 API。使用 Spring Security 所必须的。支持单独运行的应用， 远程客户端，方法（服务层）安全和 JDBC 用户供应



spring-security-web-3.0.2.RELEASE.jar

:包含过滤器和对应的 web 安全架构代码。任何需要依赖 servlet API 的。如果需要 Spring Security Web 认证服务和基于 URL 的权限控制都需要它

## 开始配置

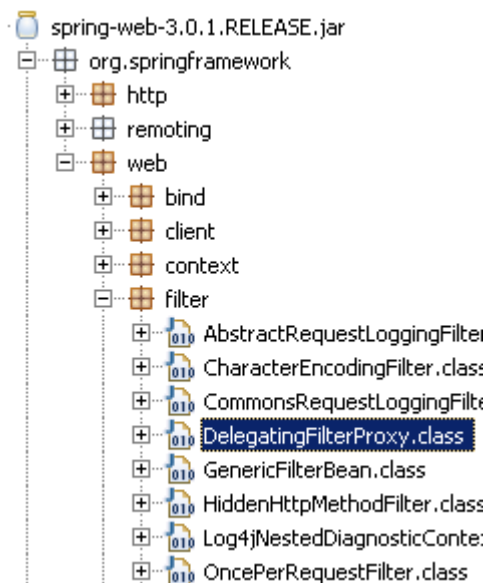
### 首先配置 web.xml 文件

在 web.xml 文件中添加一个过滤器,这个过滤器不在 security 包中. , 它可以代理一个 application context 中定义的 Spring bean 所实现的 filter

DelegatingFilterProxy 做的事情是代理 Filter 的方法, 从 application context 里获得 bean ( 这些 bean 就是 Spring Security 中的核心部分, 过滤器。这些过滤器被定义在了 Spring 容器中 )。 这让 bean 可以获得 spring web application context 的生命周期支持, 使

配置较为轻便。bean 必须实现 `javax.servlet.Filter` 接口，它必须和 `filter-name` 里定义的名称是一样的。

`DelegatingFilterProxy` 这个类在 `spring-web-3.0.0RELEASE.jar` 包中



```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxys
  </filter-class>
</filter>
<!-- 拦截所有的请求 -->
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```



## 1. 在 web.xml 中通过监听器启动 spring 容器

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<!-- 拦截所有的请求 -->
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

## 2. 在 Spring 的配置文件中引入 Spring Security 的 xml 命名空间,这需要修改原来的 applicationContext.xml 文件. 这些代码在下载文档或者示例应用程序中都能够找到

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    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-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-3.0.xsd">
</beans>
```

## Spring Security 配置

以下的代码都在 applicationContext.xml 文件中配置

```
<security:http auto-config="true">
    <security:intercept-url pattern="/" access="ROLE_USER" />
</security:http>
```

这表示,我们要保护应用程序中的所有 URL,只有拥有 ROLE\_USER 角色的用户才能访问。

<http> 元素是所有 web 相关的命名空间功能的上级元素。<intercept-url> 元素定义了 pattern, 用来匹配进入的请求 URL, 上面的表示拦截根, 以及根子目录下的所有的路径。

access 属性定义了请求匹配了指定模式时的需求。使用默认的配置, 这个一般是一个逗号分隔的角色队列, 一个用户中的一个必须被允许访问请求。前缀 "ROLE\_" 表示的是一个用户应该拥有的权限比对。

上面的配置中只有角色是 ROLE\_USER 的用户才能访问, 那么 ROLE\_USER 在哪里定义呢? 所以下面的工作就是定义 ROLE\_USER

```
<security:http auto-config="true">
  <security:intercept-url pattern="/" access="ROLE_USER" />
</security:http>

<!-- 配置认证管理器 -->
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="user" password="user" authorities="ROLE_USER"/>
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

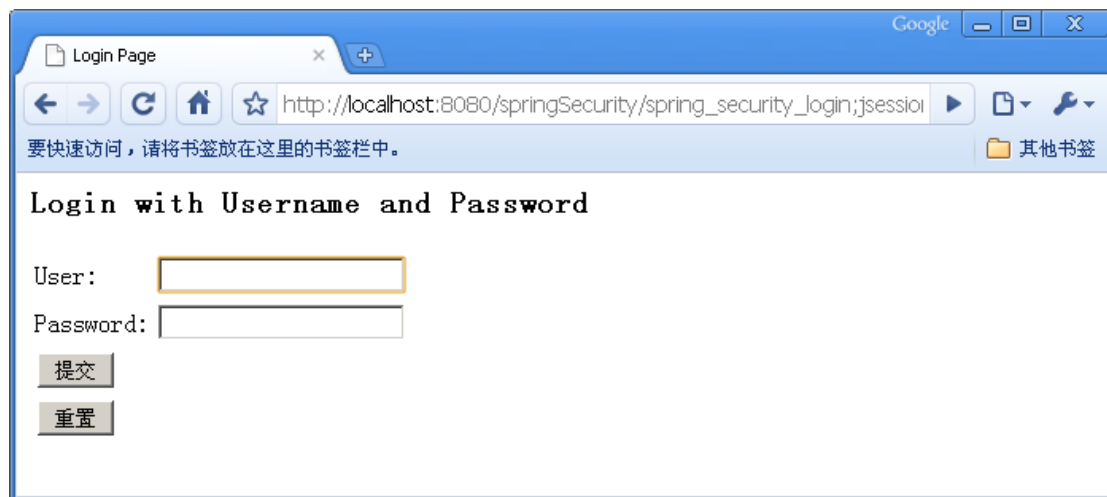
如果用户名为 user, 密码为 user 的用户成功登录了, 它的角色是 ROLE\_USER, 那么他可以访问规定的资源。

## 编写首页

目前为止，项目中只有一个 index.jsp 在这个页面上只需要写一句话

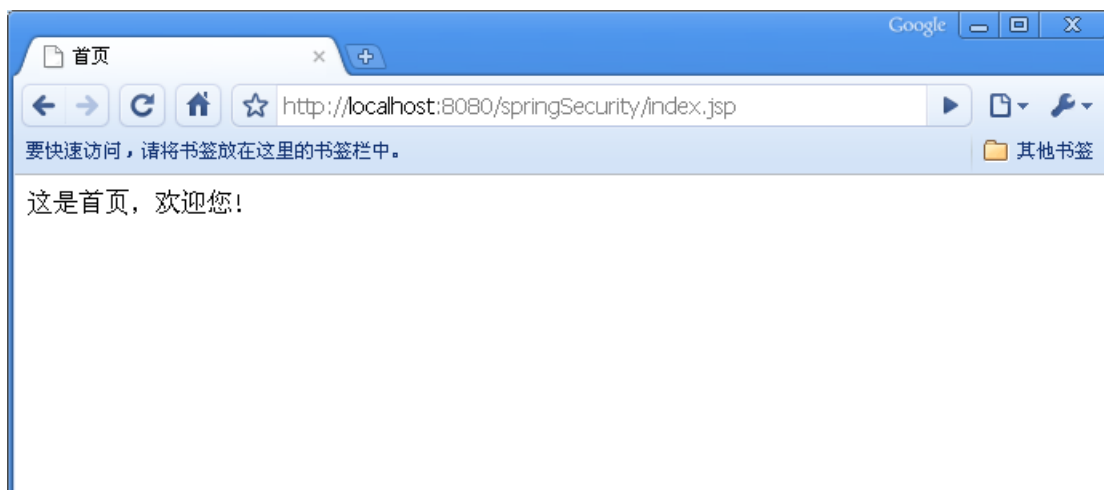
```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>首页</title>
  </head>
  <body>
    这是首页，欢迎您！
  </body>
</html>
```

将项目部署到 web 服务器中，然后访问首页 index.jsp,我们发现首页并没有出现，而是跳转到了 一个登录页面上。因为项目刚启动，第一次访问的时候，没有任何用户登录，而在配置文件中我们拦截的是所有的请求，所以第一次请求被拦截了。



输入刚才在配置文件中配置的用户名：user，密码：user，提交后，发现能够正确的转到

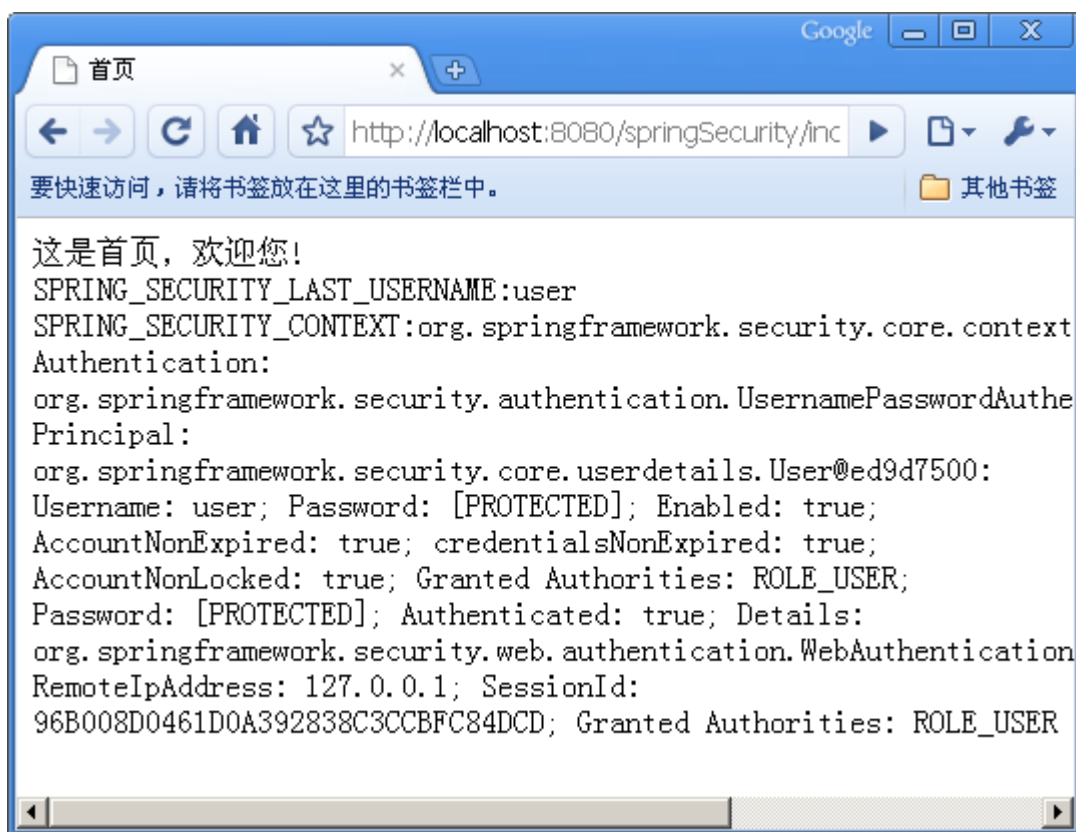
首页



问题: 那个登录页面哪里来的？

当有请求道来的时候，Spring Security 框架开始检查要访问的资源是否有权访问，如果当前登录用户无权或者当前根本就没有用户登录，则 Spring Security 框架就自动产生一个 登录页面。

当在登录页面进行了正确的登录后，Spring Security 会自动进行登录验证，如果成功登录，将用户信息放到 session 中，然后转到先前请求的页面上。我们可以在 index 中将 session 中的 key—value 打印出来，看看 Spring Security 将用户信息如何放到 session 中的：



可以看到：用户使用 的 key 为: SPRING\_SECURITY\_LAST\_USERNAME

## 进阶 I

在入门部分有明显缺陷：

- A) 一般登录页面都是我们自己编写的，并且登录页面是不需要验证直接可以访问的。
- B) 用户名和密码直接写在配置文件中，而实际项目中我们是放在数据库中的。

## 指定登录页面

在前面的例子中，我们在 Spring Security 框架生成的登录页面中直接输入用户名和密码后，Spring Security 框架替我们进行验证的，那么 HTTP 请求的时候，向 web 服务器提交了什么样的数据呢？我们打开生成的登录页面的源代码：

```
<form name='f' action='/springSecurity/j_spring_security_check' method='POST'>
  <table>
    <tr>
      <td>User:</td>
      <td><input type='text' name='j_username' value=''></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type='password' name='j_password' /></td>
    </tr>
    <tr>
      <td colspan='2'><input name="submit" type="submit"/></td>
    </tr>
    <tr>
      <td colspan='2'><input name="reset" type="reset"/></td>
    </tr>
  </table>
</form>
```

由于验证过程是 Spring Security 框架自动完成的，所以在我们的登录页面中表单元素的 name 都是固定的。

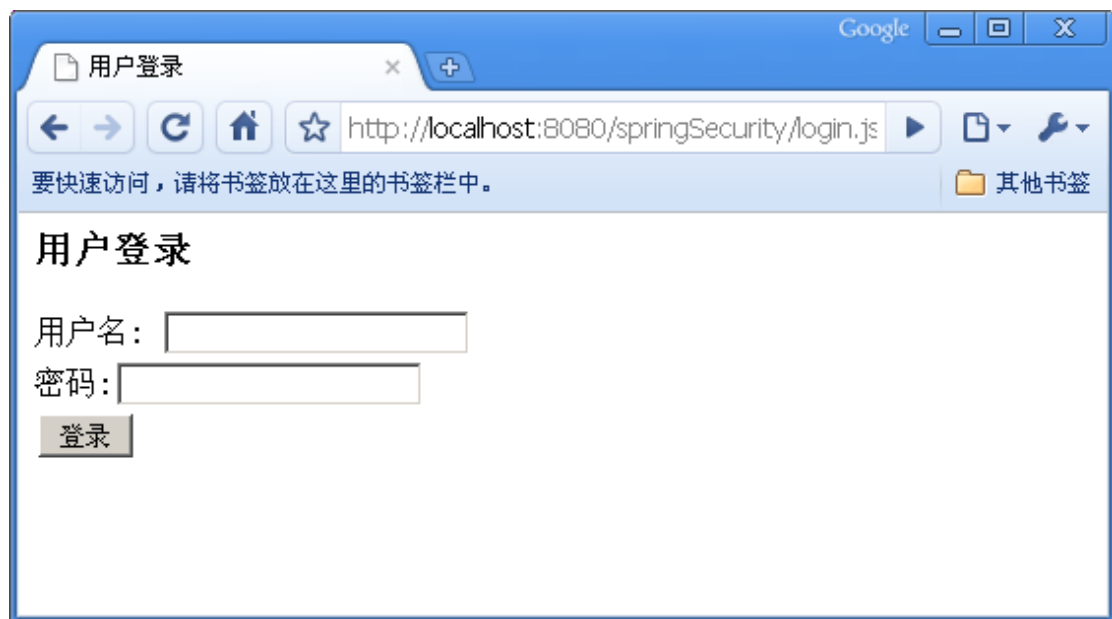
下面是我们自己编写的登录页面: login.jsp

```
<html>
  <head>
    <title>用户登录</title>
  </head>
  <body>
    <h3>用户登录</h3>
    <form action="${pageContext.request.contextPath}/j_spring_security_check"
      method="post">
      用户名: <input type="text" name="j_username" /> <br />
      密码:<input type="password" name="j_password"><br />
      <input type="submit" value="登录" />
    </form>
  </body>
</html>
```

写完登录页面之后，得让 Spring Security 框架知道哪个页面是登录页面，当需要登录的时候，Spring Security 框架会自动跳转到这个登录页面上。所以得在 applicationContext.xml 文件中进行配置

```
<security:http auto-config="true">
  <!--
  login-page:指定登录页面
  -->
  <security:form-login login-page="/login.jsp" />
  <!-- 对登录页面不进行拦截，至于为什么还要在 后面加一个 *
      那是因为 请求这个页面的时候可能会带一些参数
  -->
  <security:intercept-url pattern="/login.jsp*" filters="none" />
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

启动浏览器访问 index.jsp,我们发现自动跳转到 login.jsp 页面上了。

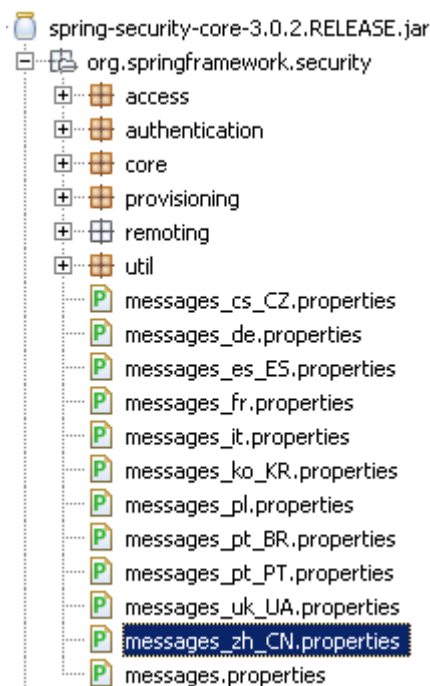


如果输入不正确的用户名或者密码,页面中没有任何错误消息提示,怎样才能将错误消息显示出来呢?

## 国际化输出

Spring Security 框架将所有的错误信息都定义成了异常,并且提供了国际化的资源文件。这个资源文件在 spring-security-core-xxx.jar 文件中





在配置文件中指定使用的资源文件。这里定义的 messageSource 对象被 spring Security 框架内部使用。

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename"
    value="classpath:org/springframework/security/messages_zh_CN" />
</bean>
```

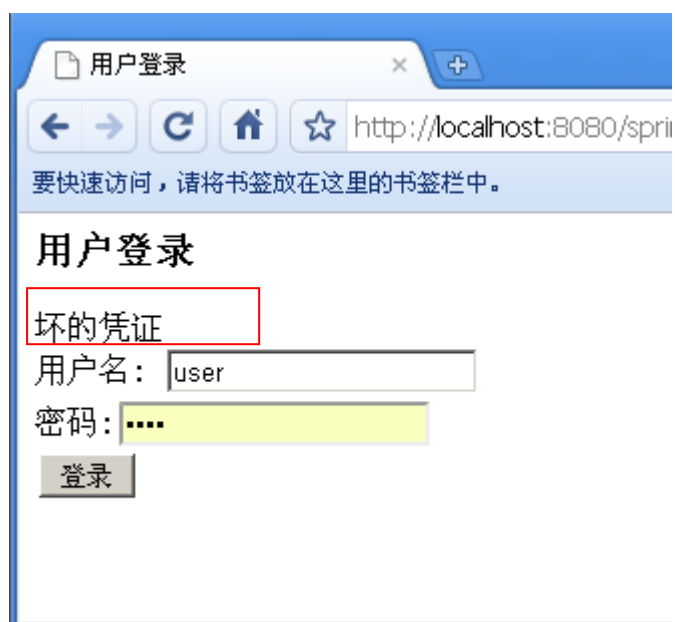
在 jsp 文件中取出错误消息，Spring Security 框架将抛出的异常对象放到了 session 范围中，

key 是: SPRING\_SECURITY\_LAST\_EXCEPTION,取出的是异常对象，所以还要调用

getMessage()方法取出真正的错误信息

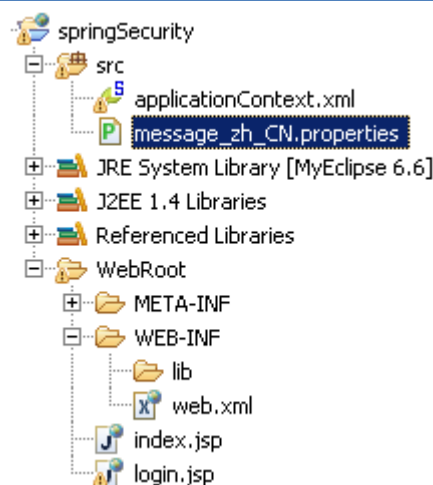
```
<h3>用户登录</h3>
${sessionScope.SPRING_SECURITY_LAST_EXCEPTION.message}
<form action="${pageContext.request.contextPath}/j_spring_security_check"
method="post">
    用户名: <input type="text" name="j_username" /> <br />
    密码:<input type="password" name="j_password"><br />

    <input type="submit" value="登录" />
</form>
```



自定义错误消息

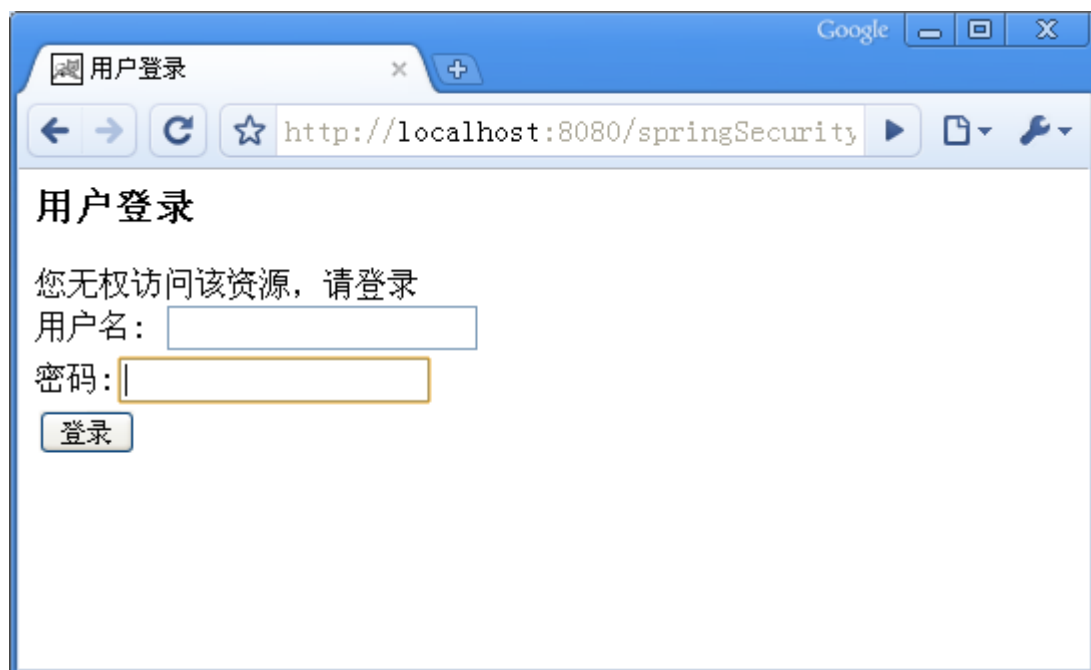
在 src 中新建一个 message\_zh\_CN.properties 文件



AbstractUserDetailsAuthenticationProvider.badCredentials=您无权访问该资源，请登录

在 applicationContext.xml 中指定资源文件。

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename"
    value="classpath:message_zh_CN" />
</bean>
```



## 对密码进行加密

上面的配置中，密码以明文的形式出现在了配置文件中，这是非常危险的。所以我们可以将真实密码经过加密之后的结果放到配置文件中。这需要在配置文件中指明使用什么样的加密算法。

MD5 是一种不可逆的加密算法，验证的时候，将用户输入的密码经过 MD5 加密之后的结果再与配置文件中配置的密文进行比较，如果相同则通过验证。

*admin 经过 MD5 加密之后的结果是：21232f297a57a5a743894a0e4a801fc3*

于是作如下配置：

```
<!-- 配置认证管理器 -->
<security:authentication-manager>
  <security:authentication-provider>
    <security:password-encoder hash="md5" />
    <security:user-service>
      <security:user name="admin" password="21232f297a57a5a743894a0e4a801fc3"
        authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

## 将权限放入数据库保存

前面将用户名和密码，还有权限都定义在了配置文件中，而实际项目中需要将这些放到数据库数据库中，那么这些信息在数据库中的表结构是什么，能不能自己定义呢？spring Security 将表结构已经定义好了，我们可以参考 发行文档中的 附录 A

---

```
/*用户表*/

create table users

(

    username varchar_ignorecase(50) not null primary key, /*用户名*/

    password varchar_ignorecase(50) not null,             /*密码*/

    enabled boolean not null                               /*是否禁用*/

);

/*权限表*/

create table authorities

(

    username varchar_ignorecase(50) not null,

    authority varchar_ignorecase(50) not null,

    constraint fk_authorities_users foreign key(username) references

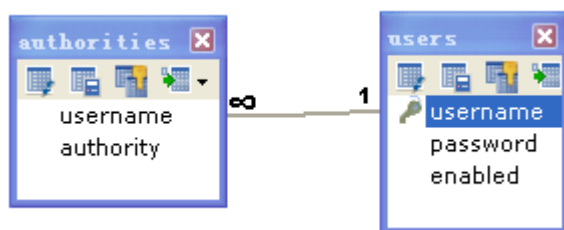
users(username)

);

create unique index ix_auth_username on authorities

(username,authority);
```

---



插入数据:

```
INSERT INTO users(username,PASSWORD,enabled)
```

```
VALUES('admin','21232f297a57a5a743894a0e4a801fc3',1);
```

/\*密码是字符串 admin 加密后的结果\*/

```
INSERT INTO users(username,PASSWORD,enabled)
```

```
VALUES('user','ee11cbb19052e40b07aac0ca060c23ee',1);
```

/\*密码是字符串 user 加密后的结果 \*/

```
INSERT INTO authorities VALUES('admin','ROLE_ADMIN');
```

```
INSERT INTO authorities VALUES('user','ROLE_USER');
```

	username	password	enabled
<input type="checkbox"/>	admin	21232f297a57a5a743894a0e4a801fc3	<input type="checkbox"/>
<input type="checkbox"/>	user	ee11cbb19052e40b07aac0ca060c23ee	<input type="checkbox"/>

---

	username	authority
<input type="checkbox"/>	admin	ROLE_ADMIN
<input type="checkbox"/>	user	ROLE_USER

---

·修改配置文件

A) 在配置文件中加入数据源,因为要访问数据库,所以还要加入 MySQL 的驱动

---

```
<bean id="datasource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
    <property name="driverClassName"
        value="com.mysql.jdbc.Driver" >
    </property>
    <property name="url"
        value="jdbc:mysql://localhost:3306/springDB?useUnicode=true&characterEncoding=utf8" >
    </property>
    <property name="username" value="root" ></property>
    <property name="password" value="root" ></property>
</bean>
```

---

B) 将原来的 user-service 注释起来,再提供一个 userService,使用刚刚配置的 dataSource

---

```
<!-- 配置认证管理器 -->
<security:authentication-manager>
    <security:authentication-provider>
        <security:password-encoder hash="md5" />
        <!--
        <security:user-service>
            <security:user name="admin"
password="21232f297a57a5a743894a0e4a801fc3"
            authorities="ROLE_USER" />
        </security:user-service>
        -->
        <security:jdbc-user-service data-source-ref="datasource" />
    </security:authentication-provider>
</security:authentication-manager>
```

---

在做实验之前，首先看看权限相关的配置:

```
<security:http auto-config="true">
  <!--
    login-page:指定登录页面
  -->
  <security:form-login login-page="/login.jsp" />
  <!-- 对登录页面不进行拦截，至于为什么还要在后面加一个*
        那是因为 请求这个页面的时候可能会带一些参数
  -->
  <security:intercept-url pattern="/login.jsp" filters="none" />
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

如果我们需要完成这样的功能：

- A) 系统中除了 login.jsp 可以直接访问以外，其它的页面都需要权限才能进入
- B) index.jsp 页面 ROLE\_USER 和 ROLE\_ADMIN 都可以访问;
- C) admin.jsp 页面只有 ROLE\_ADMIN 权限可以访问

index.jsp 页面内容如下:

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>首页</title>
  </head>
  <body>
    这是首页，欢迎您！ <br />

    <a href="admin.jsp"> 进入admin.jsp页面</a>
  </body>
</html>
```

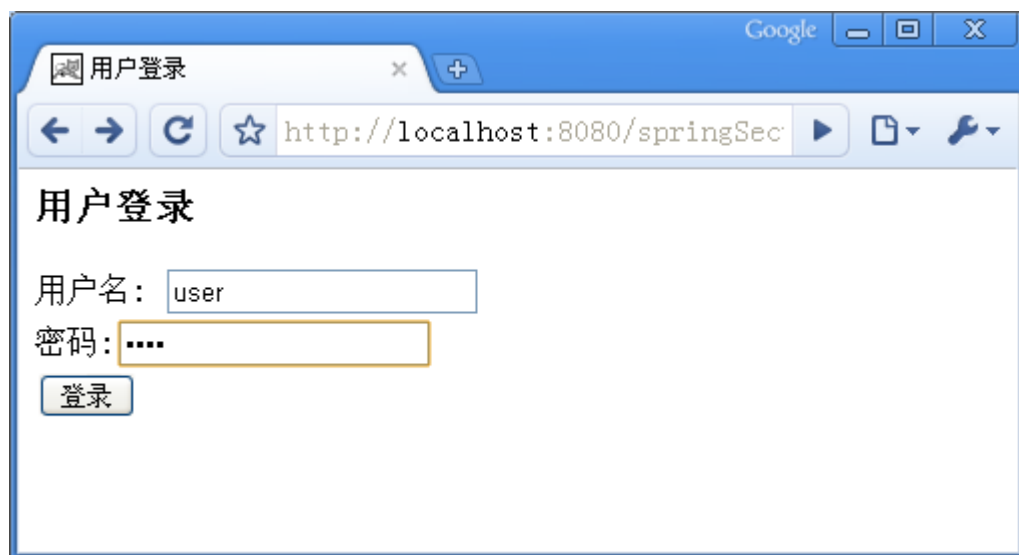
admin.jsp 页面内容:



```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>管理员页面</title>
</head>
<body>
  这是管理员页面，欢迎您！ <br />
</body>
</html>
```

上面的配置能满足要求吗？下面做实验验证：

启动项目，直接访问 login.jsp,输入用户名 user，密码 user



登录之后：



然后访问 admin.jsp 页面



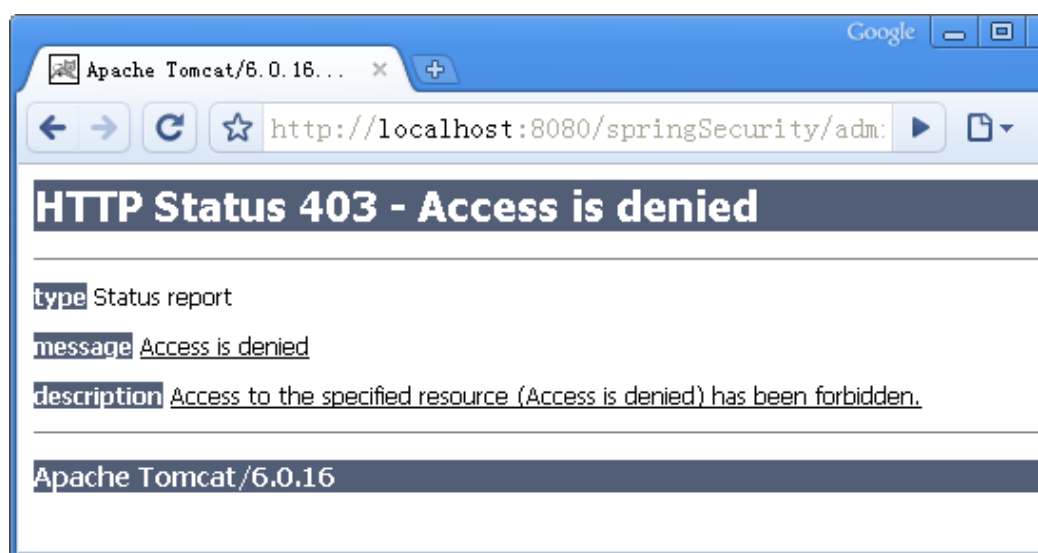
可以访问，这里无法满足我满上面的要求，为什么？

我们仔细观察上面的配置，admin.jsp 并没有配置访问权限，所以我们在配置文件中添加配

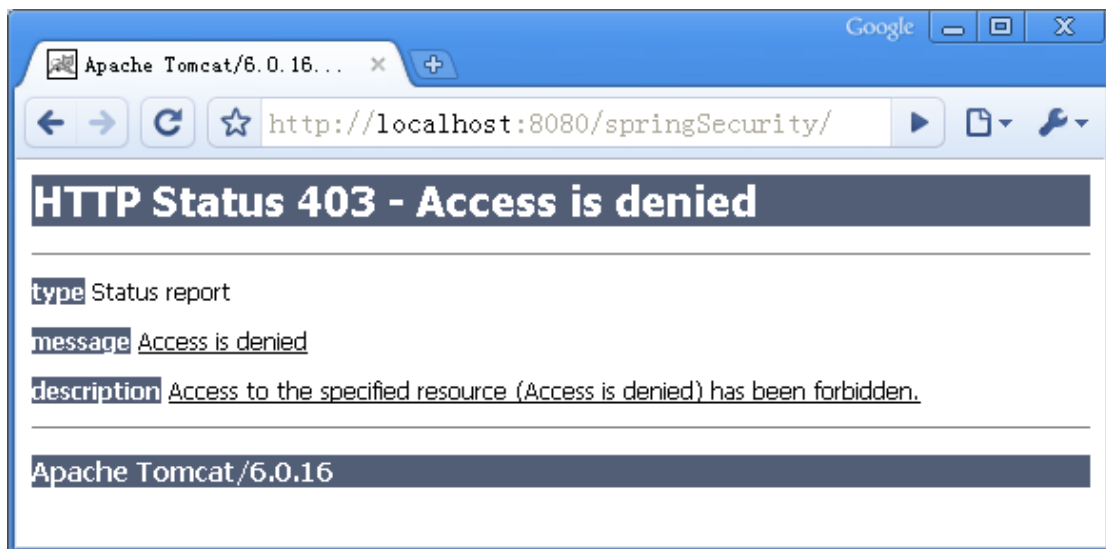
置：

```
<security:http auto-config="true">
  <!--
  login-page:指定登录页面
  -->
  <security:form-login login-page="/login.jsp" />
  <!-- 对登录页面不进行拦截，至于为什么还要在 后面加一个*
  那是因为 请求这个页面的时候可能会带一些参数
  -->
  <security:intercept-url pattern="/login.jsp*" filters="none" />
  <security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN" />
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

重新启动项目后，点击链接，发现访问被拒绝.这样满足了我们的要求



再以 admin 用户进行登录，登录后我们也发现:



我们要求 admin 用户可以访问 index.jsp,为什么这里被拒绝了昵?看看配置文件:

```
<security:intercept-url pattern="/login.jsp*" filters="none" />
<security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN" />
<security:intercept-url pattern="/**" access="ROLE_USER" />
```

我们发现 index.jsp 实际上匹配的是/\*\*,要求 ROLE\_USER 才能访问,ROLE\_ADMIN 就自然不能访问了

修改配置文件:

```
<security:intercept-url pattern="/login.jsp*" filters="none" />
<security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN" />
<security:intercept-url pattern="/index.jsp*" access="ROLE_ADMIN,ROLE_USER" />
<security:intercept-url pattern="/**" access="ROLE_USER" />
```

自定义访问被拒绝页面

修改后满足了我们的要求。但是 403 错误信息不太友好，如何自定义访问拒绝页面呢？

首先新建一个页面：accessDenied.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<html>
<head>
<title>无权访问</title>
</head>
<body>
    您的访问被拒绝，无权访问该资源！
</body>
</html>
```

修改配置文件，指定被拒绝的页面：

```
<security:http auto-config="true" access-denied-page="/accessDenied.jsp">
    <!--
    login-page:指定登录页面
    -->
    <security:form-login login-page="/login.jsp" />
    <!-- 对登录页面不进行拦截，至于为什么还要在 后面加一个*
    那是因为 请求这个页面的时候可能会带一些参数
    -->
    <security:intercept-url pattern="/login.jsp*" filters="none" />
    <security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN" />
    <security:intercept-url pattern="/index.jsp*" access="ROLE_ADMIN,ROLE_USER" />
    <security:intercept-url pattern="/*" access="ROLE_USER" />
</security:http>
```



## 获取登录用户信息

第一种方式：使用 java 代码：

了解 Spring Security 框架的核心共享组件：

被称为"shared"的组件，是指它在框架中占有很重要的位置，框架离开它们就没法运行。内置的一系列的过滤器中都用到了这些共享组件。这些 java 类表达了维持系统的构建代码块，所以理解他们的位置是非常重要的，即使你不需要直接跟他们打交道。

### ■ SecurityContextHolder

最基础的对象就是 SecurityContextHolder。我们把当前应用程序的当前安全环境的细节存储到它里边了。默认情况下，SecurityContextHolder 使用 ThreadLocal 存储这些信息，这意味着，安全环境在同一个线程执行的方法一直是有效的。我们把安全主体和系统交互的信息都保存在 SecurityContextHolder 中了

查看该类的源代码：我们发现在这个类中有一个静态属性：

SecurityContextHolderStrategy strategy,在 SecurityContextHolder 初始化的时候，为它指定了实例对象。

还有一个静态方法：

```
public static void setContext(SecurityContext context) {  
    strategy.setContext(context);  
}
```

#### ■ SecurityContext:

Security 上下文，是一个接口

```
public interface SecurityContext extends Serializable {  
    //~ Methods =====  
  
    * Obtains the currently authenticated principal, or an  
    Authentication getAuthentication();  
  
    * Changes the currently authenticated principal, or re  
    void setAuthentication(Authentication authentication);  
}
```

#### ■ Authentication :

也是一个接口。我们把安全主体和系统交互的信息都保存在

SecurityContextHolder 中了。 Spring Security 使用一个 Authentication 对应来表现这  
些信息。 虽然你通常不需要自己创建一个 Authentication 对象，直接通过

SecurityContextHolder 获取上下文对象，然后通过上下文对象 ( SecurityContext ) 获取即可

#### ■ UserDetails

UserDetails 是一个 Spring Security 的核心接口。它代表一个主体 ( 包含于用户相关的信息 )。

在 Authentication 接口中有一个方法 `Object getPrincipal()`; 这个方法返回的是一个安全主题，大多数情况下，这个对象可以强制转换成 UserDetails 对象，获取到 UserDetails 对象之后，就可以通过这个对象的 `getUserName()`方法获取当前用户名。

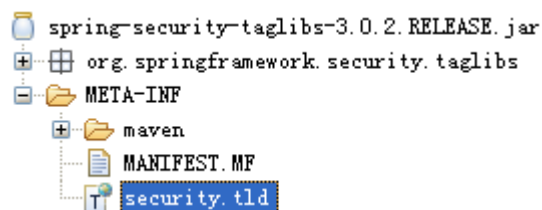
下面的代码演示了在**任何地方，如何获取当前的用户名**：

```
//获取上下文
SecurityContext secCtx=SecurityContextHolder.getContext();
//获取认证对象
Authentication auth=secCtx.getAuthentication();
//在认证对象中获取主体对象
Object principal=auth.getPrincipal();
String userName="";
if(principal instanceof UserDetails){
    userName=((UserDetails)principal).getUsername();
}else{
    userName=principal.toString();
}
```

第二种方式:使用 Spring Security 框架的标签库.

标签库存放在 spring-security-taglibs-XXX.jar 中





在 jsp 页面中使用标签库的时候，首先引入标签库：下面在 index.jsp 中使用标签库输出登录用户

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%@taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>首页</title>
</head>
<body>
  这是首页，欢迎<sec:authentication property="name" /> ! <br />
  <a href="admin.jsp"> 进入admin.jsp页面</a>
</body>
</html>
```

## 有选择的显示具备权限的链接

上面“进入 admin.jsp” 这个链接不管是 user 登录还是 admin 登录，都显示出来了。实际上只有 admin 有权限执行这个链接，所以最好是 admin 登录的时候，链接显示，而 user 登录的时候，不显示链接。这个可以使用 标签库可以做到

```
<body>
  这是首页，欢迎<sec:authentication property="name" /> ! <br />
```

```
<sec:authorize ifAllGranted="ROLE_ADMIN" >
  <a href="admin.jsp">进入admin.jsp页面</a>
</sec:authorize>
</body>
```

sec:authorize 标签的属性:

- A) ifAllGranted:只有当前用户拥有所有指定的权限时，才能显示标签体的内容（相当于“与”的关系）
- B) ifAnyGranted:当前用户拥有指定的权限中的一个的时候，就能显示标签内部内容(相当于“或”的关系)
- C) ifNotGranted:没有指定的权限的时候，显示标签体的内容（相当于“非”的关系）

也可以写成下面的形式:

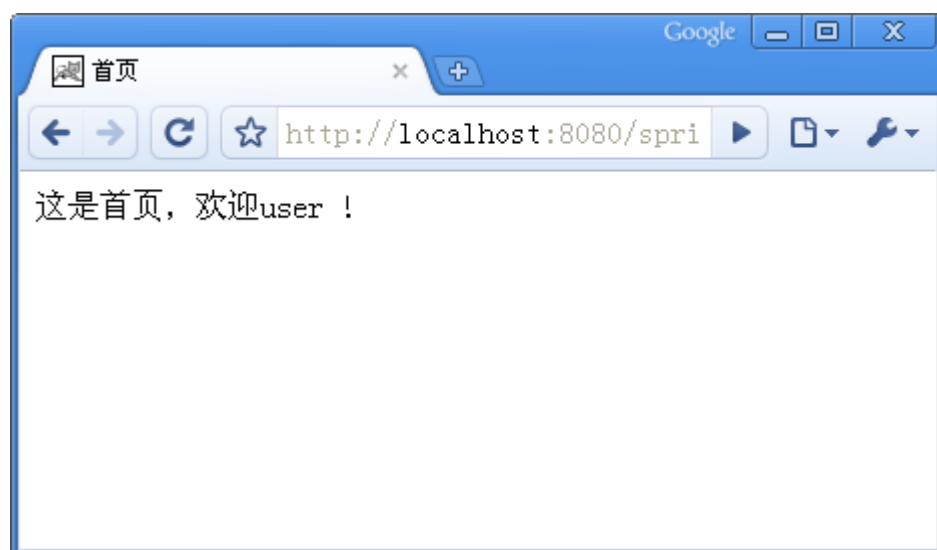
当前用户如果能访问/admin.jsp,则显示标签体的内容

```
<sec:authorize url="/admin.jsp" >
  <a href="admin.jsp">进入admin.jsp页面</a>
</sec:authorize>
```

admin 登录:



user 登录:



## 进阶 II

### 理解安全过滤器链

Spring Security 的 web 架构是完全基于标准的 servlet 过滤器的。它没有在互联网使用 servlet 或任何其他基于 servlet 的框架（比如 spring mvc），所以它没有与任何特定的 web 技术强行关联。它只管处理 `HttpServletRequest` 和 `HttpServletResponse`，不关心请求时来自浏览器，web 服务客户端，还是一个 AJAX 应用。

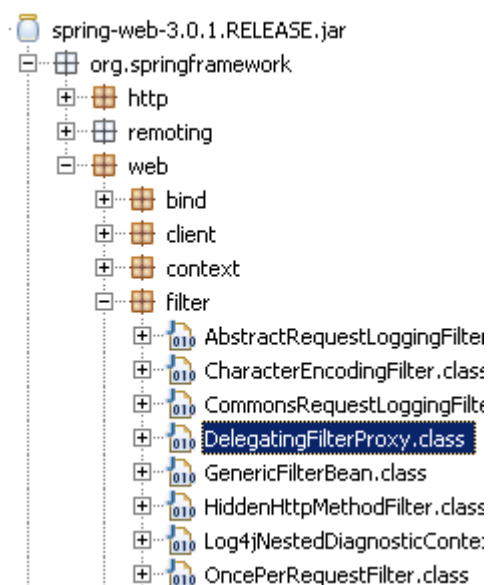
Spring Security 一启动就会包含一批负责各种安全管理的过滤器，这些过滤器组成过滤器链。

那么这些过滤器如何启动？在 `web.xml` 文件中我们看到了这样的配置：

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

这里配置了一个过滤器，名字是 `DelegatingFilterProx`，这个类在

`spring-web-3.0.0RELEASE.jar` 包中



它实际上是一个代理类，这个过滤器里没有实现过滤器的任何逻辑。

DelegatingFilterProxy 做的事情是代理 Filter 的方法，从 spring 容器【application context】里获得 bean。这让 bean 可以获得 spring web application context 的生命周期支持，使配置较为轻便。bean 必须实现 javax.servlet.Filter 接口，它必须和 filter-name 里定义的名称是一样的。查看 DelegatingFilterProxy 的 javadoc 获得更多信息。

刚才说到 Spring Security 需要一些过滤器，这些过滤器如果在 web.xml 中配置的话，配置起来就很繁琐了，这些过滤器还必须按照顺序来配置。所以为了方便，就只在 web.xml 文件中配置了一个代理过滤器，它就相当于是一个入口，web 容器启动的时候，由它到 spring 容器中启动 spring Security 需要的过滤器链。

既然如此，我们就需要在 spring 容器中配置过滤器了。我们知道，代理类需要有个代理目标，web.xml 文件中的 DelegatingFilterProxy 所代理的目标类是什么呢？这个类就是：

org.springframework.security.web.FilterChainProxy ,在 spring 容器中声明这个类,然后将要启动的过滤器链配置进去就完成了配置任务,下面是在 applicationContext.xml 文件中的配置:

```
<bean id="filterChainProxy"
class="org.springframework.security.web.FilterChainProxy">

    <security:filter-chain-map path-type="ant">
        <security:filter-chain pattern="/webServices/**" filters="
            securityContextPersistenceFilterWithASCFalse,
            basicAuthenticationFilter,
            exceptionTranslationFilter,
            filterSecurityInterceptor" />
        <security:filter-chain pattern="/**" filters="
            securityContextPersistenceFilterWithASCFalse,
            formLoginFilter,
            exceptionTranslationFilter,
            filterSecurityInterceptor" />
    </security:filter-chain-map>
</bean>
```

事实上,在前面的应用中,我们并没有看到像这样的配置,这是怎么回事?

其实我们是用下面的配置替代了上面的配置:

```
<security:http auto-config="true" >

</security:http>
```

如果我们不想用 自动配置的方式，我们可以自己编写过滤器，然后按照上面的方式进行配置。Spring Security 框架提供的过滤器已经很完善了，一般没有必要自己来定义，除非是特殊需求。

下面来看看这些过滤器：



#### ■ HttpSessionContextIntegrationFilter

【org.springframework.security.context.HttpSessionContextIntegrationFilter】:

第一个起作用的过滤器，首先判断用户的 session 中是否已经存在一个 SecurityContext 了，如果存在，就把 SecurityContext 拿出来，放到 SecurityContextHolder 中，供 Spring Security 其他部分使用。如果不存在，就创建一个 SecurityContext 出来，还是放到 SecurityContextHolder 中，供其它部分使用。

在所有过滤器执行完毕后，清空 SecurityContextHolder。

■ LogoutFilter 【org.springframework.security.ui.logout.LogoutFilter】：

只处理注销请求，默认为/j\_spring\_security\_logout。在用户发送注销请求的时候，销毁用户 session，清空 SecurityContextHolder，然后重定向到注销成功页面。

■ AuthenticationProcessingFilter

【org.springframework.security.ui.webapp.AuthenticationProcessingFilter】

处理 form 登陆的过滤器，与 form 登陆有关的所有操作都是在此进行的。默认情况下只处理/j\_spring\_security\_check 请求，这个请求应该是用户使用 form 登陆后的提交地址。此过滤器执行的基本操作时，通过用户名和密码判断用户是否有效，如果登录成功就跳转到成功页面（可能是登陆之前访问的受保护页面，也可能是默认的成功页面），如果登录失败，就跳转到失败页面。



#### ■ DefaultLoginPageGeneratingFilter

【org.springframework.security.ui.webapp.DefaultLoginPageGeneratingFilter】

此过滤器用来生成一个默认的登录页面，默认的访问地址为/spring\_security\_login，这个默认的登录页面虽然支持用户输入用户名，密码，也支持 rememberMe 功能，但是因为太难看了，只能是在演示时做个样子，不可能直接用在实际项目中

#### ■ BasicProcessingFilter

【org.springframework.security.ui.basicauth.BasicProcessingFilter】

此过滤器用于进行 basic 验证，功能与 AuthenticationProcessingFilter 类似，只是验证的方式不同

#### ■ SecurityContextHolderAwareRequestFilter

【org.springframework.security.wrapper.SecurityContextHolderAwareRequestFilter】

此过滤器用来包装客户的请求。目的是在原始请求的基础上，为后续程序提供一些额外的数据。比如 getRemoteUser()时直接返回当前登陆的用户名之类的。

#### ■ RememberMeProcessingFilter

【org.springframework.security.ui.rememberme.RememberMeProcessingFilter】

此过滤器实现 RememberMe 功能，当用户 cookie 中存在 rememberMe 的标记，此过滤器会根据标记自动实现用户登陆，并创建 SecurityContext，授予对应的权限。

#### ■ AnonymousProcessingFilter

【org.springframework.security.providers.anonymous.AnonymousProcessingFilter】

为了保证操作统一性，当用户没有登陆时，默认为用户分配匿名用户的权限。有关匿名登录功能的详细信息

#### ■ ExceptionTranslationFilter

【org.springframework.security.ui.ExceptionTranslationFilter】

此过滤器的作用是处理中 FilterSecurityInterceptor 抛出的异常，然后将请求重定向到对应页面，或返回对应的响应错误代码。

#### ■ SessionFixationProtectionFilter

【org.springframework.security.ui.SessionFixationProtectionFilter】

防御会话伪造攻击。

#### ■ FilterSecurityInterceptor

【org.springframework.security.intercept.web.FilterSecurityInterceptor】

用户的权限控制都包含在这个过滤器中。功能一：如果用户尚未登陆，则抛出 AuthenticationCredentialsNotFoundException “尚未认证异常”。

功能二：如果用户已登录，但是没有访问当前资源的权限，则抛出 AccessDeniedException “拒绝访问异常”。

功能三：如果用户已登录，也具有访问当前资源的权限，则放行。

至此，我们完全展示了默认情况下 Spring Security 中使用到的过滤器，以及每个过滤器的应用场景和显示功能。

## 管理会话

应用中常常有这样的情形：一个用户正确的登录了系统，并没有使用系统提供的“退出”功能，而是直接关掉浏览器，或者长时间不适用系统，Tomcat 默认情况下会保存 session 30 分钟，30 分钟后，会话就会过期。

- 我们可以配置 Spring Security 检测失效的 session ID，并把用户转发到对应的 URL 上，我们可以通过配置 session-management 元素

```
<security:http auto-config="true" access-denied-page="/accessDenied.jsp" >
  <security:form-login login-page="/login.jsp" />
  <security:intercept-url pattern="/login.jsp*" filters="none" />
  <security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN" />
  <security:intercept-url pattern="/index.jsp*" access="ROLE_ADMIN,ROLE_USER" />
  <security:intercept-url pattern="/*" access="ROLE_USER" />

  <!-- 会话管理配置 -->
  <security:session-management invalid-session-url="/sessionTimeout.html">
    </security:session-management>
</security:http>
```

SessionTimeout.html 文件

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

  <head>

    <title>会话超时</title>

  </head>

  <body>

    您的会话超时了！

  </body>

</html>
```

## ■ 在页面中提供“退出”链接

Spring Security 在启动的时候，启动了一个过滤器 LogoutFilter，它能够接收请求 /j\_spring\_security\_logout，此时可以将当前登录的用户“踢”出系统

在首页中添加一个连接

```
<body>
这是首页，欢迎<sec:authentication property="name" /> ! <br />

<sec:authorize ifAllGranted="ROLE_ADMIN" >
    <a href="admin.jsp">进入admin.jsp页面</a>
</sec:authorize>

<a href="j_spring_security_logout">退出系统</a>
```

指定退出系统后，跳转的地方：

```
<security:form-login login-page="/login.jsp" />
<security:logout logout-success-url="/login.jsp"/>
<security:intercept-url pattern="/login.jsp*" filters="none" />
<security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN" />
<security:intercept-url pattern="/index.jsp*" access="ROLE_ADMIN,ROLE_USER" />
<security:intercept-url pattern="/**" access="ROLE_USER" />
```

## ■ 同步会话控制

多个用户不能使用同一个账号同时登陆系统。为了实现这个功能，我们首先要在 web.xml 文件中添加一个监听器，这个监听器会在 session 创建和销毁的时候通知 Spring Security。

```
<listener>

<listener-class>

org.springframework.security.web.session.HttpSessionEventPublisher

</listener-class>

</listener>
```

然后，在 applicationContext 中添加下面的配置，这将防止一个用户重复登录好几次-第  
二次登录会让第一次登录失效。

```
<!-- 会话管理配置 -->
<security:session-management invalid-session-url="/sessionTimeout.html">
  <security:concurrency-control max-sessions="1"/>
</security:session-management>
```

通常我们更想防止第二次登录，这时候我们可以使用

```
<!-- 会话管理配置 -->
<security:session-management invalid-session-url="/sessionTimeout.html">
  <security:concurrency-control max-sessions="1"
    error-if-maximum-exceeded="true"
  />
</security:session-management>
```

### 用户登录

Maximum sessions of 1 for this principal exceeded

用户名:

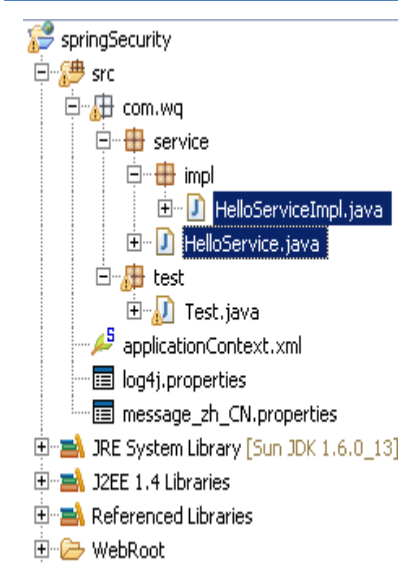
密码:

## 进阶 III

### 保护业务代码的执行

Spring Security 使用 AOP 对方法调用进行了权限控制，这部分内容来源于 Spring 提供的 AOP 功能，Spring Security 进行了自己的封装，我们可以使用声明和编程两种方式进行权限管理

为应用添加一个业务接口和业务的实现类:



The image shows the Eclipse IDE project structure for 'springSecurity'. The 'src' directory contains a package 'com.wq' with sub-packages 'service' and 'test'. The 'service' package contains an 'impl' sub-package with 'HelloServiceImpl.java' and 'HelloService.java'. The 'test' package contains 'Test.java'. Other files in the project include 'applicationContext.xml', 'log4j.properties', and 'message\_zh\_CN.properties'. The 'HelloService.java' file is highlighted, showing its content:

```
public interface HelloService {  
    public String sayHello(String name);  
    public String sayBye(String name);  
}
```

The 'HelloServiceImpl.java' file is also shown, implementing the 'HelloService' interface:

```
public class HelloServiceImpl implements HelloService {  
  
    @Override  
    public String sayBye(String name) {  
        return "再见, "+name+"您正确执行了sayBye方法";  
    }  
  
    @Override  
    public String sayHello(String name) {  
        return "您好, "+name+"您正确执行了sayHello方法";  
    }  
}
```

#### ■ 使用 annotation 保护业务方法的执行

第一步:首先在 applicationContext.xml 文件中启用 annotation

```
<!-- 启用annotation -->  
<security:global-method-security  
    secured-annotations="enabled"  
    jsr250-annotations="enabled" />
```

第二步:在业务接口中使用注解对方法调用加以限制

```
public interface HelloService {  
  
    @Secured({"ROLE_USER", "ROLE_ADMIN"})  
    public String sayHello(String name);  
  
    @Secured("ROLE_ADMIN")  
    public String sayBye(String name);  
}
```



---

管理员两个方法都可以调用，而 RSER 只能调用 sayHello 方法

---

第三步:调用这些业务方法。在这个例子中，为了简单起见，我不使用任何第三方 MVC 框架，编写一个 Servlet，直接在 Servlet 中从 Spring 容器中取出业务对象，然后对它进行调用。

在 Spring 容器 applicationContext.xml 中部署业务类

---

```
<!-- 业务方法 -->
<bean id="helloService" class="com.wq.service.impl.HelloServiceImpl
```

---

添加 Servlet，Servlet 的映射是/hello.do 以下是 Servlet 中 doGet 方法中的部分方法

---

```
out.println(" <BODY>");
// 取得Spring容器
ApplicationContext ctx = WebApplicationContextUtils
    .getWebApplicationContext(this.getServletContext());

//取出业务类
HelloService helloService=ctx.getBean("helloService",HelloService.class);
String method=request.getParameter("method");
String name=request.getParameter("name");

//调用sayBye方法
if("sayBye".equals(method)){
    out.println(helloService.sayBye(name));
}
//调用sayHello方法
if("sayHello".equals(method)){
    out.println(helloService.sayHello(name));
}

out.println(" </BODY>");
```

---

在 index.jsp 中添加链接

```
<a href="hello.do?method=sayHello&name=<sec:authentication property="name" />
问好
</a>

<br />
<a href="hello.do?method=sayBye&name=<sec:authentication property="name" />
再见
</a>
```

为了让 admin 和 user 都能够访问 hello.do,在 applicationContext 中需要放行:

```
<security:intercept-url pattern="/hello.do*" access="ROLE_ADMIN,ROLE_USER"/>
```

以 user 身份登录后执行 “问好”



以 user 身份登录后执行 “再见”



以 admin 身份登录后发现两者都能执行

- 使用配置的方式，在 applicationContext.xml 配置文件中使用 **protect-pointcut** 添加安全切点。

使用 annotation 的方式很明显非常麻烦，需要在每一个业务方法中指定权限。如果使用 protect-pointcut 添加切入点，定义一个切入点表达式，就会非常方便。关于切入点表达式可以参考 《spring 参考手册》 6.2 章节

切入点表达式语法格式：

---

`execution ( modifiers-pattern? ret-type-pattern declaring-type-pattern?`

`name-pattern ( param-pattern ) throws-pattern? )`

其中用 ? 标注的是可选部分

---

去掉代码中的 annotation,然后在 applicationContext 中修改代码如下:

```
<security:global-method-security>

    <!--
    ROLE_USER:可以访问sayHello方法
    -->
    <security:protect-pointcut access="ROLE_USER,ROLE_ADMIN"
    expression="execution(* com.wq.service.*.sayHello(..))"/>
    <!--
    第一个*: 表示返回类型
    第二个*: 表示任意的类
    完整的意义:ROLE ADMIN角色可以访问:
    com.wq.service包中的任意类中以say开头的方法,任意的参数类型,
    任意的返回类型
    -->
    <security:protect-pointcut access="ROLE_ADMIN"
    expression="execution(* com.wq.service.*.say*(..))"/>

</security:global-method-security>
```