

当天目标

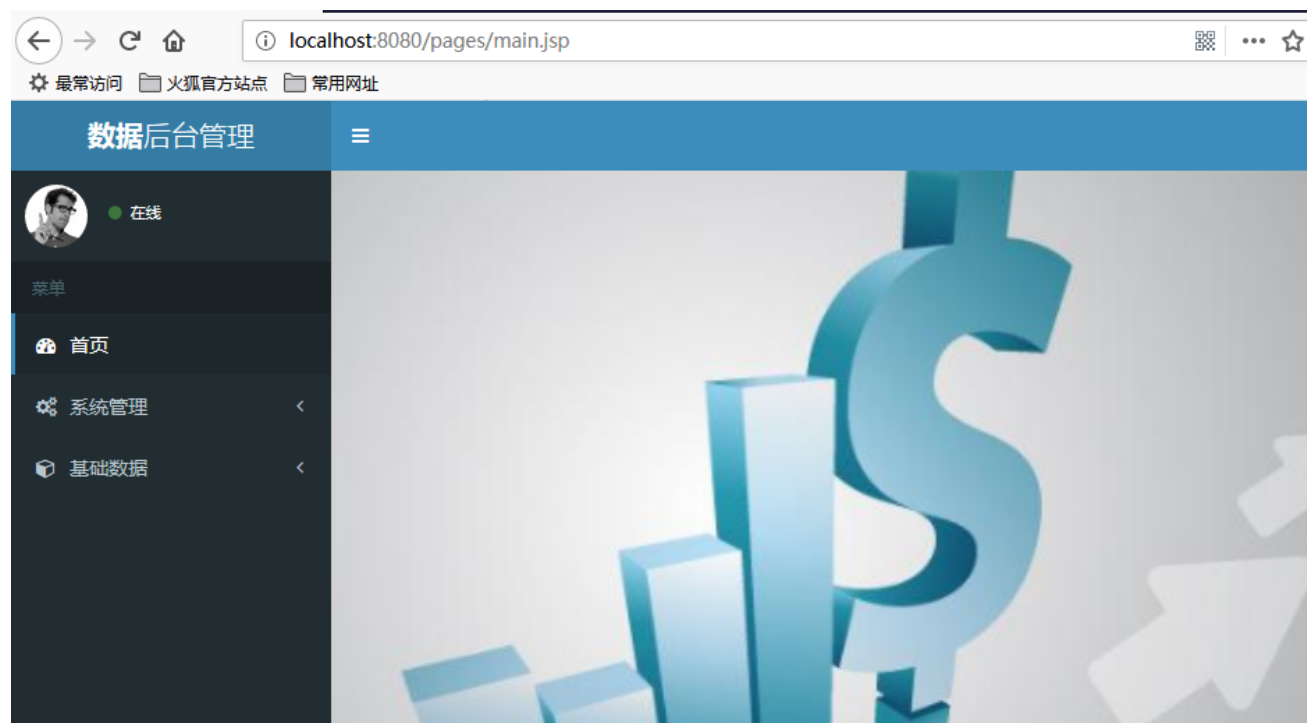
基本掌握SpringSecurity的认证功能实现！

一、案例介绍

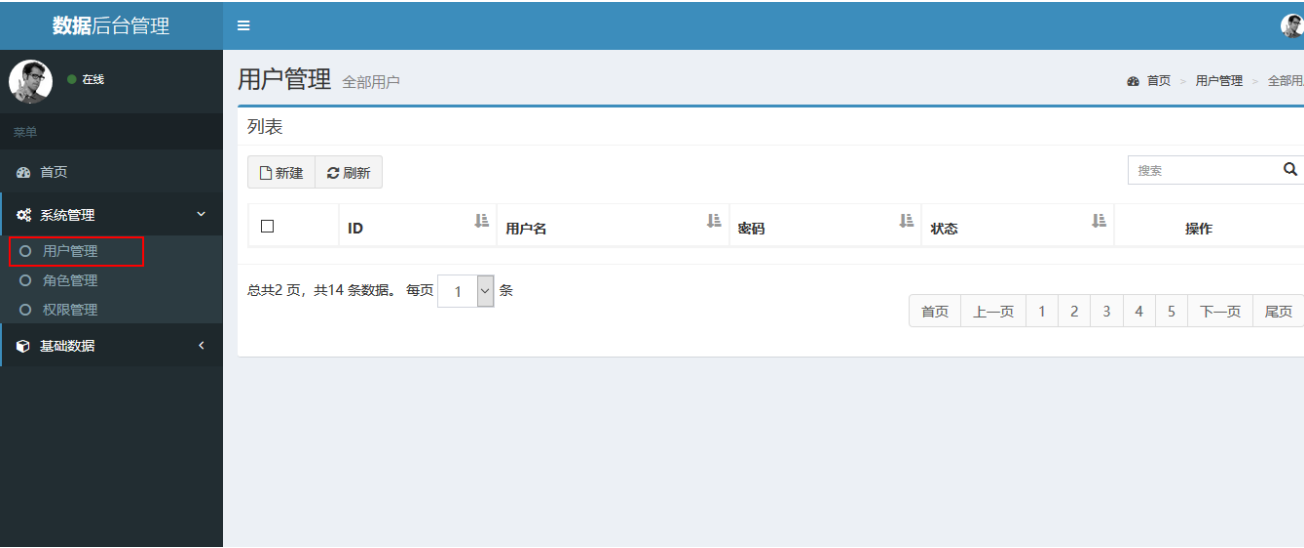
说明：为方便教学，这里已经事先为大家准备好了一个半成品的后台管理系统，而想要完善另一部分，就需要用到我们今天学习的内容SpringSecurity了！

1.1 案例效果图

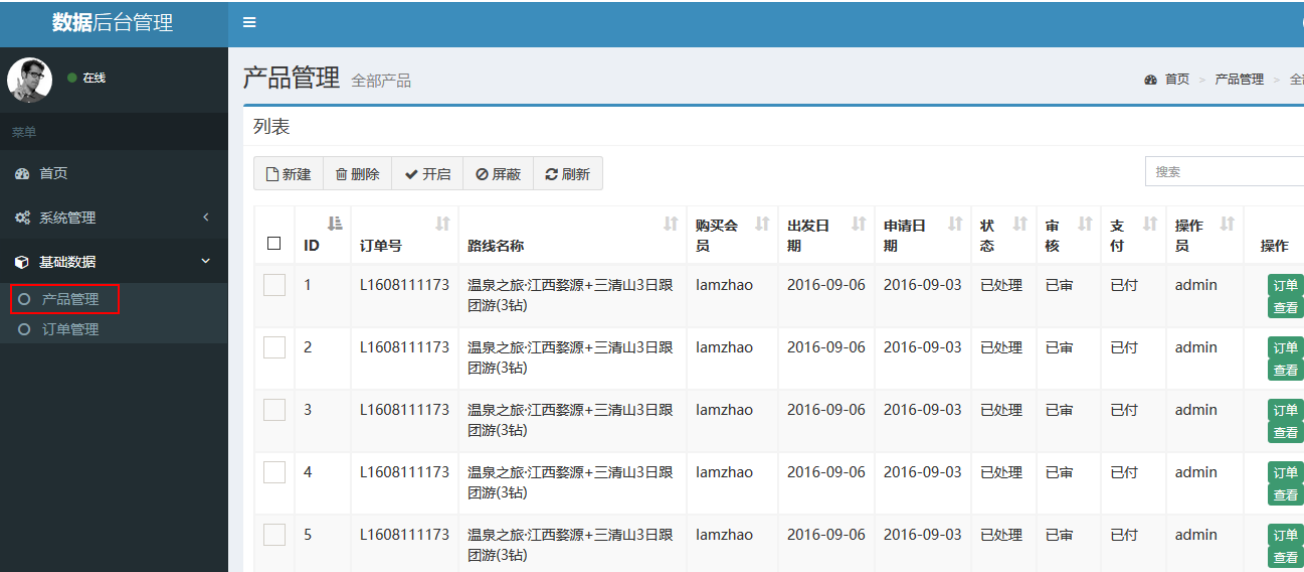
1.1.1 启动项目进入首页



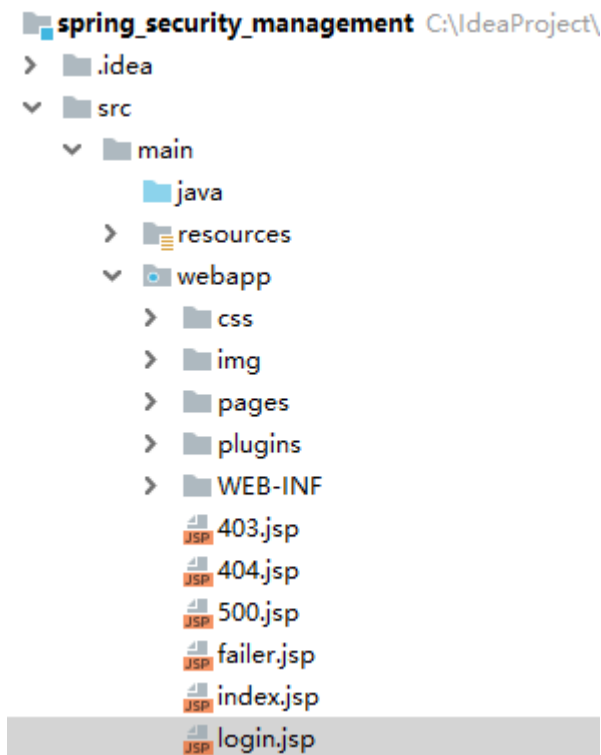
1.1.2 系统管理界面



1.1.3 基础数据界面



1.1.4 项目最终目录结构



1.2 建表语句

注意：这几张表的作用下文有分解！

```
DROP TABLE IF EXISTS `sys_permission`;

CREATE TABLE `sys_permission` (
  `ID` int(11) NOT NULL AUTO_INCREMENT COMMENT '编号',
  `permission_NAME` varchar(30) DEFAULT NULL COMMENT '菜单名称',
  `permission_url` varchar(100) DEFAULT NULL COMMENT '菜单地址',
  `parent_id` int(11) NOT NULL DEFAULT '0' COMMENT '父菜单id',
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `sys_role`;

CREATE TABLE `sys_role` (
  `ID` int(11) NOT NULL AUTO_INCREMENT COMMENT '编号',
  `ROLE_NAME` varchar(30) DEFAULT NULL COMMENT '角色名称',
  `ROLE_DESC` varchar(60) DEFAULT NULL COMMENT '角色描述',
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `sys_role_permission`;

CREATE TABLE `sys_role_permission` (
  `RID` int(11) NOT NULL COMMENT '角色编号',
  `PID` int(11) NOT NULL COMMENT '权限编号',
  PRIMARY KEY (`RID`, `PID`),
  KEY `FK_Reference_12` (`PID`),
```

```

CONSTRAINT `FK_Reference_11` FOREIGN KEY (`RID`) REFERENCES `sys_role` (`ID`),
CONSTRAINT `FK_Reference_12` FOREIGN KEY (`PID`) REFERENCES `sys_permission` (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `sys_user`;

CREATE TABLE `sys_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `password` varchar(120) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '密码',
  `status` int(1) DEFAULT '1' COMMENT '1开启0关闭',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `sys_user_role`;

CREATE TABLE `sys_user_role` (
  `UID` int(11) NOT NULL COMMENT '用户编号',
  `RID` int(11) NOT NULL COMMENT '角色编号',
  PRIMARY KEY (`UID`,`RID`),
  KEY `FK_Reference_10` (`RID`),
  CONSTRAINT `FK_Reference_10` FOREIGN KEY (`RID`) REFERENCES `sys_role` (`ID`),
  CONSTRAINT `FK_Reference_9` FOREIGN KEY (`UID`) REFERENCES `sys_user` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

1.3 页面部分所用技术简单说明

1.3.1 adminLTE介绍

AdminLTE是一款基于Bootstrap的页面模板，可以快速构建出一套美观的后台管理页面。

官网地址：<https://adminlte.io/>

下载地址：<https://github.com/ColorlibHQ/AdminLTE/releases>

由于原版adminLTE是纯英文的，对于多数中国程序员，使用起来不太方便。

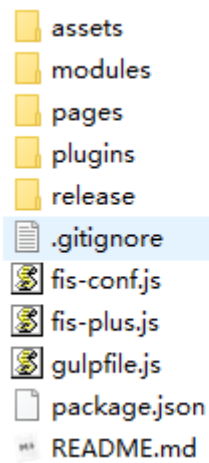
对此：传智播客研究院针对英文版本AdminLTE进行了汉化，并优化与定制了部分页面，方便我们的学习与使用。

下载地址：<https://github.com/itheima2017/adminlte2-itheima>

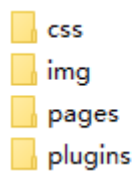
在线浏览：<http://research.itcast.cn/adminlte2-itcast/release/dist/pages/all-admin-index.html>

1.3.2 adminLTE使用

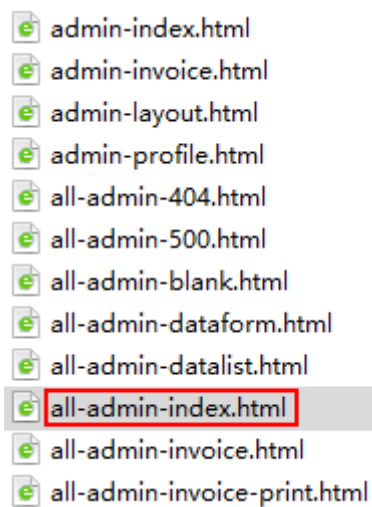
下载传智播客汉化版adminLTE后解压，目录如下：



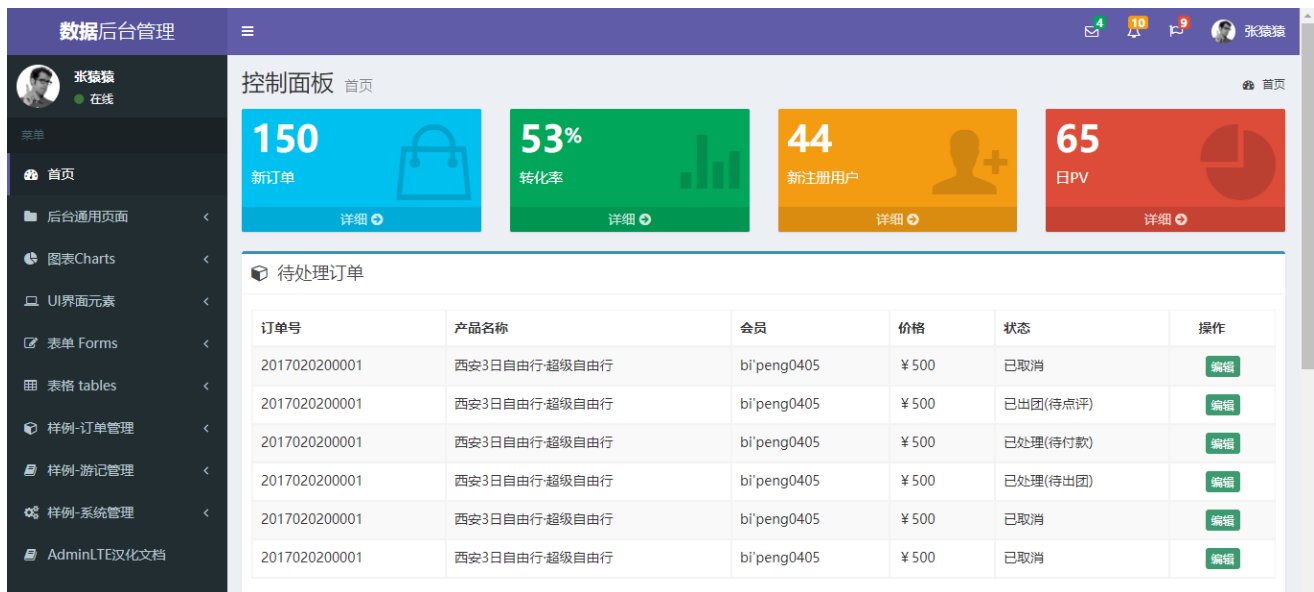
然后点击release，继续点击dist，看到如下目录：



其中css，img和plugins三个文件夹中都是静态资源，pages中就是做好的页面模板。打开pages：

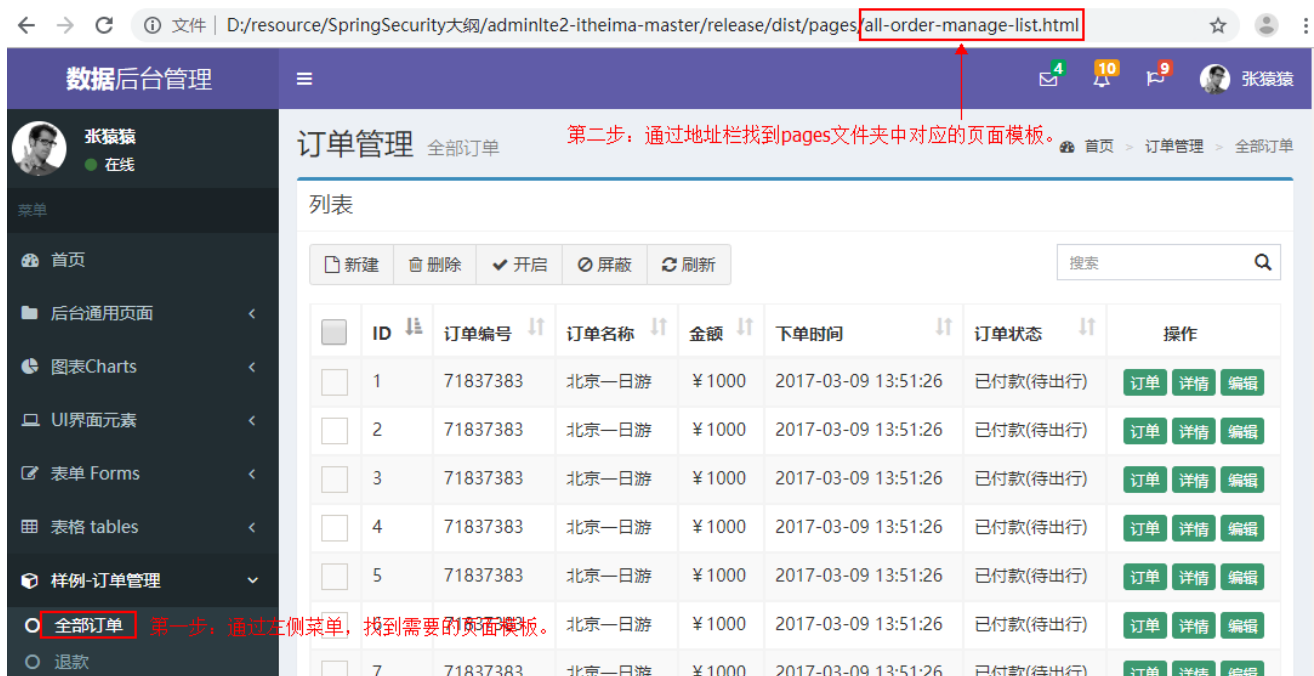


接下来，找到上图所标注的all-admin-index.html，双击打开！



从左侧菜单栏可以逐个浏览页面模板，需要哪个，就从地址栏找到对应的页面，复制粘贴即可使用！

比如：需要一个订单列表页面，具体操作如下图所标注，注意，中间页面上的按钮也是可以点的哦！



学习成本是不是非常低！

当然页面中很多细节还是需要花时间调整的，这里我已经提前做好了一套静态页面，咱们直接拿来使用即可！

1.4 后台部分所用技术简单说明

后台代码采用springmvc实现web层，spring控制业务层事务，mybatis操作数据库，这三个框架大家一定非常熟悉了，这里我就不再赘述！

二、初识权限管理

2.1 权限管理概念

权限管理，一般指根据系统设置的安全规则或者安全策略，用户可以访问而且只能访问自己被授权的资源。权限管理几乎出现在任何系统里面，前提是需要有用户和密码认证的系统。

在权限管理的概念中，有两个非常重要的名词：

认证：通过用户名和密码成功登陆系统后，让系统得到当前用户的角色身份。

授权：系统根据当前用户的角色，给其授予对应可以操作的权限资源。

2.2 完成权限管理需要三个对象

用户：主要包含用户名，密码和当前用户的角色信息，可实现认证操作。

角色：主要包含角色名称，角色描述和当前角色拥有的权限信息，可实现授权操作。

权限：权限也可以称为菜单，主要包含当前权限名称，url地址等信息，可实现动态展示菜单。

注：这三个对象中，用户与角色是多对多的关系，角色与权限是多对多的关系，用户与权限没有直接关系，二者是通过角色来建立关联关系的。

三、初识Spring Security

3.1 Spring Security概念

Spring Security是spring采用AOP思想，基于servlet过滤器实现的安全框架。它提供了完善的认证机制和方法级的授权功能。是一款非常优秀的权限管理框架。

3.2 Spring Security简单入门

Spring Security博大精深，设计巧妙，功能繁杂，一言难尽，咱们还是直接上代码吧！

3.2.1 创建web工程并导入jar包

Spring Security主要jar包功能介绍

spring-security-core.jar

核心包，任何Spring Security功能都需要此包。

spring-security-web.jar

web工程必备，包含过滤器和相关的Web安全基础结构代码。

spring-security-config.jar

用于解析xml配置文件，用到Spring Security的xml配置文件的就要用到此包。

spring-security-taglibs.jar

Spring Security提供的动态标签库，jsp页面可以用。

```

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
  <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>5.1.5.RELEASE</version>
</dependency>

```

最终依赖树效果

```

> < 1.8 > C:\Program Files\Java\jdk1.8.0_162
> Maven: org.springframework.security:spring-security-acl:5.1.5.RELEASE
> Maven: org.springframework.security:spring-security-config:5.1.5.RELEASE
> Maven: org.springframework.security:spring-security-core:5.1.5.RELEASE
> Maven: org.springframework.security:spring-security-taglibs:5.1.5.RELEASE
> Maven: org.springframework.security:spring-security-web:5.1.5.RELEASE
> Maven: org.springframework:spring-aop:5.1.6.RELEASE
> Maven: org.springframework:spring-beans:5.1.6.RELEASE
> Maven: org.springframework:spring-context:5.1.6.RELEASE
> Maven: org.springframework:spring-core:5.1.6.RELEASE
> Maven: org.springframework:spring-expression:5.1.6.RELEASE
> Maven: org.springframework:spring-jcl:5.1.6.RELEASE
> Maven: org.springframework:spring-jdbc:5.1.6.RELEASE
> Maven: org.springframework:spring-tx:5.1.6.RELEASE
> Maven: org.springframework:spring-web:5.1.6.RELEASE

```

3.2.2 配置web.xml

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
  <display-name>Archetype Created Web Application</display-name>

  <!--Spring Security过滤器链，注意过滤器名称必须叫springSecurityFilterChain-->
  <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>

</web-app>

```


3.2.3 配置spring-security.xml

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="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.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security.xsd">

    <!--设置可以用spring的el表达式配置Spring Security并自动生成对应配置组件（过滤器）-->
    <security:http auto-config="true" use-expressions="true">
        <!--使用spring的el表达式来指定项目所有资源访问都必须有ROLE_USER或ROLE_ADMIN角色-->
        <security:intercept-url pattern="/**" access="hasAnyRole('ROLE_USER','ROLE_ADMIN')"/>
    </security:http>

    <!--设置Spring Security认证用户信息的来源-->
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user name="user" password="{noop}user"
                               authorities="ROLE_USER" />
                <security:user name="admin" password="{noop}admin"
                               authorities="ROLE_ADMIN" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>

</beans>
```

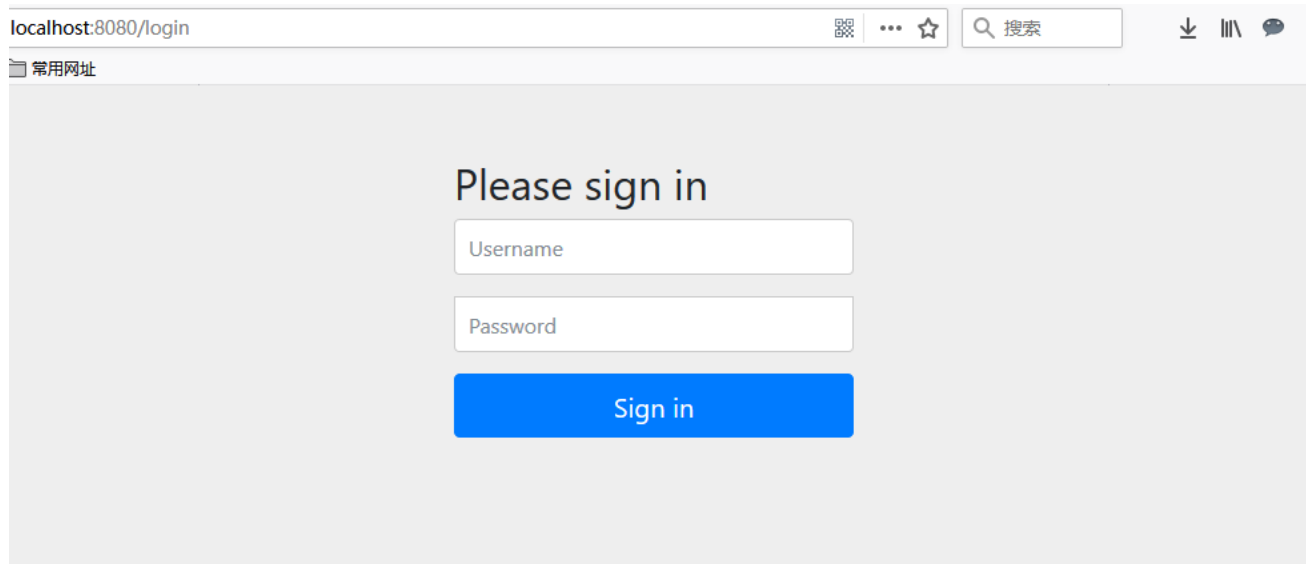
3.2.4 将spring-security.xml配置文件引入到applicationContext.xml中

```
<!--引入SpringSecurity主配置文件-->
<import resource="classpath:spring-security.xml"/>
```

3.2.5 运行结果

好了！开始启动项目了，万众期待看到index.jsp中的内容！

三..... 二..... 一..... Duang!



唉！？说好的首页呢！？为何生活不是我想象！？

地址栏中login处理器谁写的！？这个带有歪果仁文字的页面哪来的！？这么丑！？我可以换了它吗！？

稍安勿躁.....咱们先看看这个页面源代码，真正惊心动魄的还在后面呢.....

这不就是一个普通的form表单吗？除了那个_csrf的input隐藏文件！

注意！这可是你想使用自定义页面时，排查问题的一条重要线索！

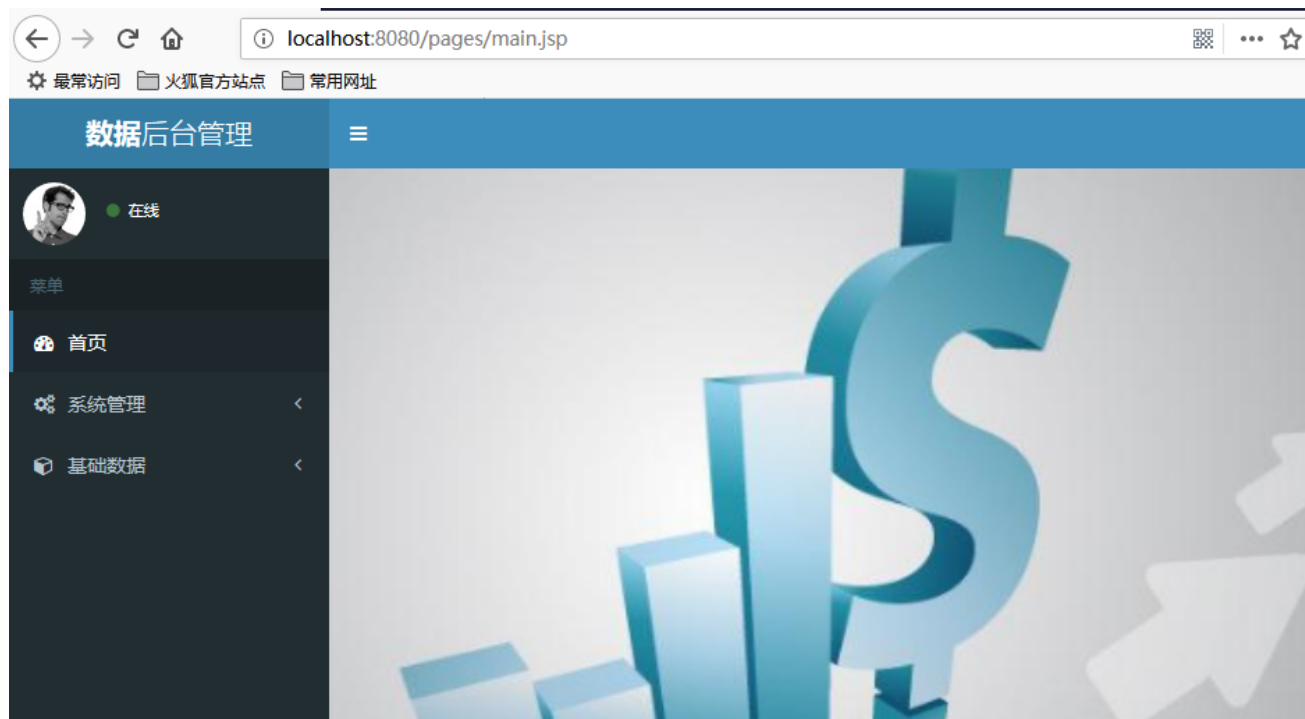
```
<div class="container">
  <form class="form-signin" method="post" action="/login">
    <h2 class="form-signin-heading">Please sign in</h2>
    <p>
      <label class="sr-only" for="username">Username</label>
      <input id="username" class="form-control" type="text" name="username" placeholder="Username" required="" autofocus="">
    </p>
    <p>
      <label class="sr-only" for="password">Password</label>
      <input id="password" class="form-control" type="password" name="password" placeholder="Password" required="">
    </p>
    <input name="_csrf" type="hidden" value="10f3ee8d-398d-480b-b48c-d09ec9c45446">
    <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
  </form>
</div>
```

我们再去看看控制台发生了什么，这里我偷偷在项目中加了日志包和配置文件.....

惊不惊喜！？意不意外！？哪来这么多过滤器啊！？

```
2019-06-25 10:59:25,815 1215 [on(2)-127.0.0.1] INFO web.DefaultSecurityFilterChain - Creating filter chain: any request,
[org.springframework.security.web.context.SecurityContextPersistenceFilter@d7835b3, org.springframework.security.web.context
.request.async.WebAsyncManagerIntegrationFilter@53c277d6, org.springframework.security.web.header.HeaderWriterFilter@759bf616,
org.springframework.security.web.csrf.CsrfFilter@3a28bfb6, org.springframework.security.web.authentication.logout
.LogoutFilter@7631d0ed, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@368d653a,
org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@1c0c1ebb, org.springframework.security.web
.authentication.ui.DefaultLogoutPageGeneratingFilter@6326c783, org.springframework.security.web.authentication.www
.BasicAuthenticationFilter@1b68d5b2, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@4611c750,
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@3a94e0ef, org.springframework.security.web
.authentication.AnonymousAuthenticationFilter@6f5b32e1, org.springframework.security.web.session
.SessionManagementFilter@32c3bffd, org.springframework.security.web.access.ExceptionTranslationFilter@7aa0d5b2, org
.springframework.security.web.access.intercept.FilterSecurityInterceptor@672ed191]
```

最后，我们在这个登录页面上输入用户名user，密码user，点击Sign in，好了，总算再次看到首页了！



一个Spring Security简单入门，已经是凝云重重，举步维艰了，咱们下回再分析吧：)

四、Spring Security过滤器链

4.1 Spring Security常用过滤器介绍

过滤器是一种典型的AOP思想，关于什么是过滤器，就不赘述了，谁还不知道凡是web工程都能用过滤器？

接下来咱们就一起看看Spring Security中这些过滤器都是干啥用的，源码我就不贴出来了，有名字，大家可以自己在idea中Double Shift去。我也会在后续的学习过程中穿插详细解释。

1. org.springframework.security.web.context.SecurityContextPersistenceFilter

首当其冲的一个过滤器，作用之重要，自不必多言。

SecurityContextPersistenceFilter主要是使用SecurityContextRepository在session中保存或更新一个SecurityContext，并将SecurityContext给以后的过滤器使用，来为后续filter建立所需的上下文。SecurityContext中存储了当前用户的认证以及权限信息。

2. org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter

此过滤器用于集成SecurityContext到Spring异步执行机制中的WebAsyncManager

3. org.springframework.security.web.header.HeaderWriterFilter

向请求的Header中添加相应的信息,可在http标签内部使用security:headers来控制

4. org.springframework.security.web.csrf.CsrfFilter

csrf又称跨域请求伪造，SpringSecurity会对所有post请求验证是否包含系统生成的csrf的token信息，如果不包含，则报错。起到防止csrf攻击的效果。

5. org.springframework.security.web.authentication.logout.LogoutFilter

匹配URL为/logout的请求，实现用户退出,清除认证信息。

6. org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter

认证操作全靠这个过滤器，默认匹配URL为/login且必须为POST请求。

7. org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter

如果没有在配置文件中指定认证页面，则由该过滤器生成一个默认认证页面。

8. org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter

由此过滤器可以生产一个默认的退出登录页面

9. org.springframework.security.web.authentication.www.BasicAuthenticationFilter

此过滤器会自动解析HTTP请求中头部名字为Authentication，且以Basic开头的头信息。

10. org.springframework.security.web.savedrequest.RequestCacheAwareFilter

通过HttpSessionRequestCache内部维护了一个RequestCache，用于缓存HttpServletRequest

11. org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter

针对ServletRequest进行了一次包装，使得request具有更加丰富的API

12. org.springframework.security.web.authentication.AnonymousAuthenticationFilter

当SecurityContextHolder中认证信息为空,则会创建一个匿名用户存入到SecurityContextHolder中。
spring security为了兼容未登录的访问，也走了一套认证流程，只不过是一个匿名的身份。

13. org.springframework.security.web.session.SessionManagementFilter

SecurityContextRepository限制同一用户开启多个会话的数量

14. org.springframework.security.web.access.ExceptionTranslationFilter

异常转换过滤器位于整个springSecurityFilterChain的后方，用来转换整个链路中出现的异常

15. org.springframework.security.web.access.intercept.FilterSecurityInterceptor

获取所配置资源访问的授权信息，根据SecurityContextHolder中存储的用户信息来决定其是否有权
限。

好了！这一堆排山倒海的过滤器介绍完了。

那么，是不是spring security一共就这么多过滤器呢？答案是否定的！随着spring-security.xml配置的添加，还会出现新的过滤器。

那么，是不是spring security每次都会加载这些过滤器呢？答案也是否定的！随着spring-security.xml配置的修改，有些过滤器可能会被去掉。

4.2 spring security过滤器链加载原理

通过前面十五个过滤器功能的介绍，对于SpringSecurity简单入门中的疑惑是不是在心中已经有了答案了呀？

但新的问题来了！我们并没有在web.xml中配置这些过滤器啊？它们都是怎么被加载出来的？

友情提示：前方高能预警，吃饭喝水打瞌睡的请睁大眼睛，专注心神！

4.2.1 DelegatingFilterProxy

我们在web.xml中配置了一个名称为springSecurityFilterChain的过滤器DelegatingFilterProxy，接下我直接对DelegatingFilterProxy源码里重要代码进行说明，其中删减掉了一些不重要的代码，大家注意我写的注释就行了！

```
public class DelegatingFilterProxy extends GenericFilterBean {
    @Nullable
    private String contextAttribute;
    @Nullable
    private WebApplicationContext webApplicationContext;
    @Nullable
    private String targetBeanName;
    private boolean targetFilterLifecycle;
    @Nullable
    private volatile Filter delegate; //注：这个过滤器才是真正加载的过滤器
    private final Object delegateMonitor;

    //注：doFilter才是过滤器的入口，直接从这里看！
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
filterChain) throws ServletException, IOException {
        Filter delegateToUse = this.delegate;
        if (delegateToUse == null) {
            synchronized(this.delegateMonitor) {
                delegateToUse = this.delegate;
                if (delegateToUse == null) {
                    WebApplicationContext wac = this.findWebApplicationContext();
                    if (wac == null) {
                        throw new IllegalStateException("No WebApplicationContext found: no
ContextLoaderListener or DispatcherServlet registered?");
                    }
                    //第一步：doFilter中最重要的一步，初始化上面私有过滤器属性delegate
                    delegateToUse = this.initDelegate(wac);
                }

                this.delegate = delegateToUse;
            }
        }
        //第三步：执行FilterChainProxy过滤器
        this.invokeDelegate(delegateToUse, request, response, filterChain);
    }

    //第二步：直接看最终加载的过滤器到底是谁
    protected Filter initDelegate(WebApplicationContext wac) throws ServletException {
        //debug得知targetBeanName为：springSecurityFilterChain
        String targetBeanName = this.getTargetBeanName();
        Assert.state(targetBeanName != null, "No target bean name set");
        //debug得知delegate对象为：FilterChainProxy
        Filter delegate = (Filter)wac.getBean(targetBeanName, Filter.class);
        if (this.isTargetFilterLifecycle()) {
            delegate.init(this.getFilterConfig());
        }

        return delegate;
    }
}
```

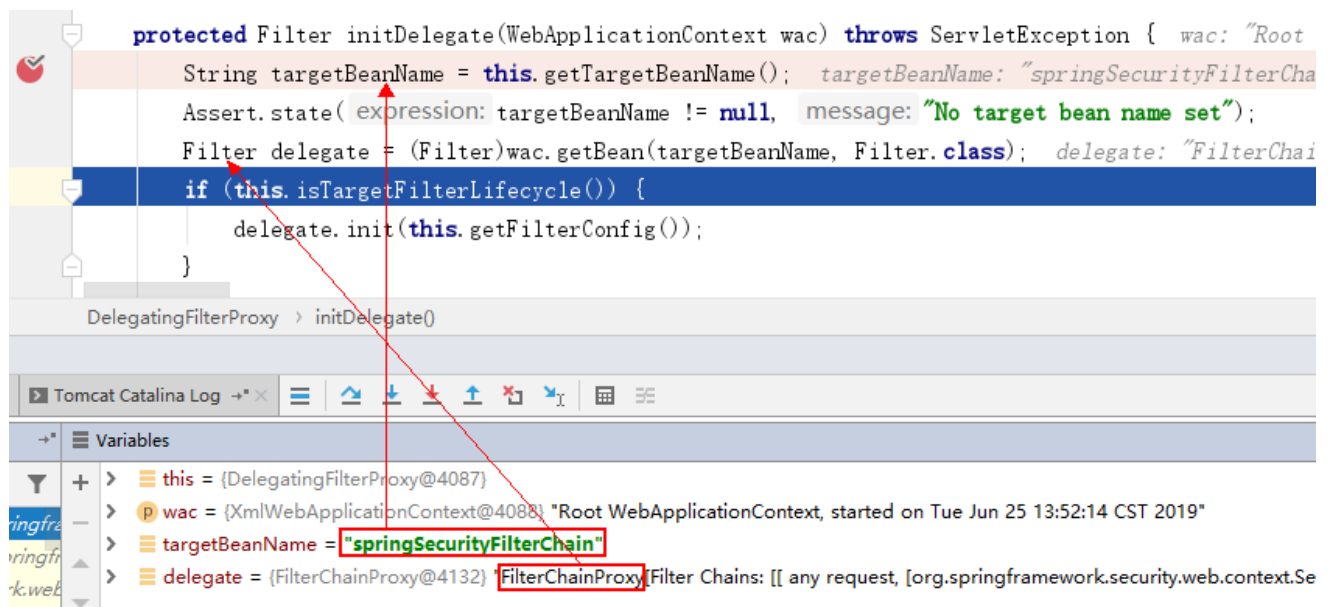
```

    }

    protected void invokeDelegate(Filter delegate, ServletRequest request, ServletResponse
response, FilterChain filterChain) throws ServletException, IOException {
        delegate.doFilter(request, response, filterChain);
    }
}

```

第二步debug结果如下：



由此可知，DelegatingFilterProxy通过springSecurityFilterChain这个名称，得到了一个FilterChainProxy过滤器，最终在第三步执行了这个过滤器。

4.2.2 FilterChainProxy

注意代码注释！注意代码注释！注意代码注释！

```

public class FilterChainProxy extends GenericFilterBean {
    private static final Log logger = LoggerFactory.getLog(FilterChainProxy.class);
    private static final String FILTER_APPLIED =
FilterChainProxy.class.getName().concat(".APPLIED");
    private List<SecurityFilterChain> filterChains;
    private FilterChainProxy.FilterChainValidator filterChainValidator;
    private HttpFirewall firewall;

    // 啊！？ 可以通过一个叫SecurityFilterChain的对象实例化出一个FilterChainProxy对象
    // 这FilterChainProxy又是何方神圣？ 会不会是真正的过滤器链对象呢？ 先留着这个疑问！
    public FilterChainProxy(SecurityFilterChain chain) {
        this(Arrays.asList(chain));
    }

    // 又是SecurityFilterChain这家伙！ 嫌疑更大了！
    public FilterChainProxy(List<SecurityFilterChain> filterChains) {

        this.filterChainValidator = new FilterChainProxy.NullFilterChainValidator();
    }
}

```

```

        this.firewall = new StrictHttpFirewall();
        this.filterChains = filterChains;
    }

    //注: 直接从doFilter看
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
        boolean clearContext = request.getAttribute(FILTER_APPLIED) == null;
        if (clearContext) {
            try {
                request.setAttribute(FILTER_APPLIED, Boolean.TRUE);
                this.doFilterInternal(request, response, chain);
            } finally {
                SecurityContextHolder.clearContext();
                request.removeAttribute(FILTER_APPLIED);
            }
        } else {
            //第一步: 具体操作调用下面的doFilterInternal方法了
            this.doFilterInternal(request, response, chain);
        }
    }

    private void doFilterInternal(ServletRequest request, ServletResponse response, FilterChain
    chain) throws IOException, ServletException {
        FirewalledRequest fwRequest =
        this.firewall.getFirewalledRequest((HttpServletRequest)request);
        HttpServletResponse fwResponse =
        this.firewall.getFirewalledResponse((HttpServletResponse)response);
        //第二步: 封装要执行的过滤器链, 那么多过滤器就在这里被封装进去了!
        List<Filter> filters = this.getFilters((HttpServletRequest)fwRequest);
        if (filters != null && filters.size() != 0) {
            FilterChainProxy.VirtualFilterChain vfc = new
            FilterChainProxy.VirtualFilterChain(fwRequest, chain, filters);
            //第四步: 加载过滤器链
            vfc.doFilter(fwRequest, fwResponse);
        } else {
            if (logger.isDebugEnabled()) {
                logger.debug(UrlUtils.buildRequestUrl(fwRequest) + (filters == null ? " has no
                matching filters" : " has an empty filter list"));
            }
            fwRequest.reset();
            chain.doFilter(fwRequest, fwResponse);
        }
    }

    private List<Filter> getFilters(HttpServletRequest request) {
        Iterator var2 = this.filterChains.iterator();
        //第三步: 封装过滤器链到SecurityFilterChain中!
        SecurityFilterChain chain;
        do {
            if (!var2.hasNext()) {
                return null;
            }

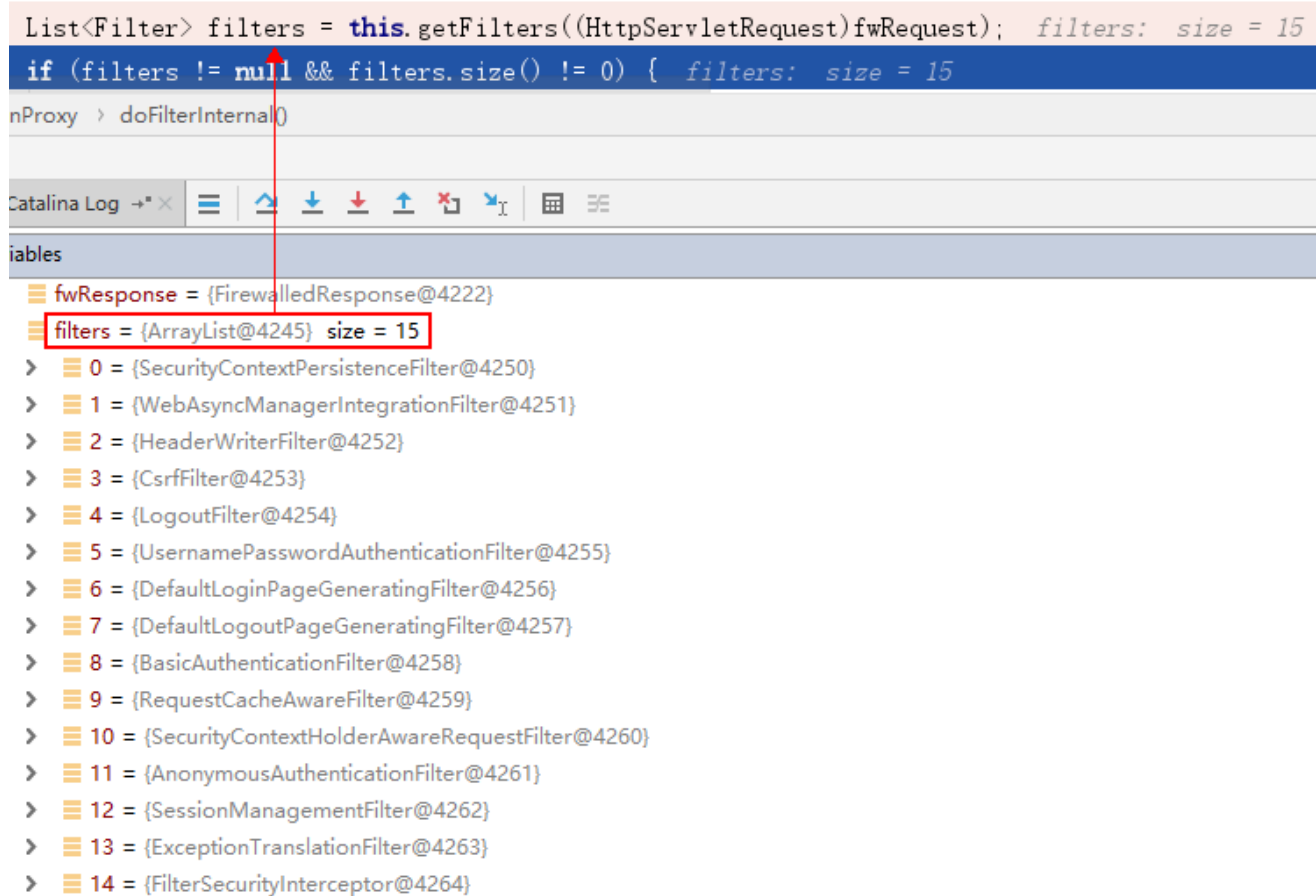
```

```

        chain = (SecurityFilterChain)var2.next();
    } while(!chain.matches(request));
    return chain.getFilters();
}
}

```

第二步debug结果如下图所示，惊不惊喜？十五个过滤器都在这里了！



```

List<Filter> filters = this.getFilters((HttpServletRequest)fwRequest); filters: size = 15
if (filters != null && filters.size() != 0) { filters: size = 15
nProxy > doFilterInternal()
Catalina Log → ×
Variables
fwResponse = {FirewalledResponse@4222}
filters = {ArrayList@4245} size = 15
> 0 = {SecurityContextPersistenceFilter@4250}
> 1 = {WebAsyncManagerIntegrationFilter@4251}
> 2 = {HeaderWriterFilter@4252}
> 3 = {CsrfFilter@4253}
> 4 = {LogoutFilter@4254}
> 5 = {UsernamePasswordAuthenticationFilter@4255}
> 6 = {DefaultLoginPageGeneratingFilter@4256}
> 7 = {DefaultLogoutPageGeneratingFilter@4257}
> 8 = {BasicAuthenticationFilter@4258}
> 9 = {RequestCacheAwareFilter@4259}
> 10 = {SecurityContextHolderAwareRequestFilter@4260}
> 11 = {AnonymousAuthenticationFilter@4261}
> 12 = {SessionManagementFilter@4262}
> 13 = {ExceptionTranslationFilter@4263}
> 14 = {FilterSecurityInterceptor@4264}

```

再看第三步，怀疑这么久！原来这些过滤器还真是都被封装进SecurityFilterChain中了。

4.2.3 SecurityFilterChain

最后看SecurityFilterChain，这是个接口，实现类也只有一个，这才是web.xml中配置的过滤器链对象！

```

//接口
public interface SecurityFilterChain {
    boolean matches(HttpServletRequest var1);

    List<Filter> getFilters();
}
//实现类
public final class DefaultSecurityFilterChain implements SecurityFilterChain {
    private static final Log logger = LogFactory.getLog(DefaultSecurityFilterChain.class);
    private final RequestMatcher requestMatcher;
    private final List<Filter> filters;

    public DefaultSecurityFilterChain(RequestMatcher requestMatcher, Filter... filters) {

```



```

        this(requestMatcher, Arrays.asList(filters));
    }

    public DefaultSecurityFilterChain(RequestMatcher requestMatcher, List<Filter> filters) {
        logger.info("Creating filter chain: " + requestMatcher + ", " + filters);
        this.requestMatcher = requestMatcher;
        this.filters = new ArrayList(filters);
    }

    public RequestMatcher getRequestMatcher() {
        return this.requestMatcher;
    }

    public List<Filter> getFilters() {
        return this.filters;
    }

    public boolean matches(HttpServletRequest request) {
        return this.requestMatcher.matches(request);
    }

    public String toString() {
        return "[" + this.requestMatcher + ", " + this.filters + "]";
    }
}

```

总结：通过此章节，我们对SpringSecurity工作原理有了一定的认识。但理论千万条，功能第一条，探寻底层，是为了更好的使用框架。

那么，言归正传！到底如何使用自己的页面来实现SpringSecurity的认证操作呢？要完成此功能，首先要有一套自己的页面！

五、SpringSecurity使用自定义认证页面

5.1 在SpringSecurity主配置文件中指定认证页面配置信息

```

<?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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="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.xsd

```

```

        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">

<!--直接释放无需经过SpringSecurity过滤器的静态资源-->
<security:http pattern="/css/**" security="none"/>
<security:http pattern="/img/**" security="none"/>
<security:http pattern="/plugins/**" security="none"/>
<security:http pattern="/failer.jsp" security="none"/>
<security:http pattern="/favicon.ico" security="none"/>

<!--设置可以用spring的el表达式配置Spring Security并自动生成对应配置组件（过滤器）-->
<security:http auto-config="true" use-expressions="true">
    <!--指定login.jsp页面可以被匿名访问-->
    <security:intercept-url pattern="/login.jsp" access="permitAll()"/>
    <!--使用spring的el表达式来指定项目所有资源访问都必须有ROLE_USER或ROLE_ADMIN角色-->
    <security:intercept-url pattern="/**" access="hasAnyRole('ROLE_USER','ROLE_ADMIN')"/>
    <!--指定自定义的认证页面-->
    <security:form-login login-page="/login.jsp"
                        login-processing-url="/login"
                        default-target-url="/index.jsp"
                        authentication-failure-url="/failer.jsp"/>
    <!--指定退出登录后跳转的页面-->
    <security:logout logout-url="/logout"
                    logout-success-url="/login.jsp"/>
</security:http>

<!--设置Spring Security认证用户信息的来源-->
<security:authentication-manager>
    <security:authentication-provider>
        <security:user-service>
            <security:user name="user" password="{noop}user"
                          authorities="ROLE_USER" />
            <security:user name="admin" password="{noop}admin"
                          authorities="ROLE_ADMIN" />
        </security:user-service>
    </security:authentication-provider>
</security:authentication-manager>

</beans>

```

修改认证页面的请求地址



login.jsp ×

```
<form action="${pageContext.request.contextPath}/login"
method="post">
  <div class="form-group has-feedback">
    <input type="text" name="username" class="form-control"
placeholder="用户名"> <span
class="glyphicon glyphicon-envelope form-control-feedback"></span>
  </div>
  <div class="form-group has-feedback">
    <input type="password" name="password" class="form-control"
placeholder="密码"> <span
class="glyphicon glyphicon-lock form-control-feedback"></span>
```

再次启动项目后就可以看到自定义的酷炫认证页面了！

ITCAST后台管理系统

登录系统

☐ 记住 下次自动登录[忘记密码](#)

然后你开开心心的输入了用户名user，密码user，就出现了如下的界面：



403什么异常？这是SpringSecurity中的权限不足！这个异常怎么来的？还记得上面SpringSecurity内置认证页面源码中的那个_csrf隐藏input吗？问题就在这了！

5.2 SpringSecurity的csrf防护机制

CSRF (Cross-site request forgery) 跨站请求伪造，是一种难以防范的网络攻击方式。

5.2.1 SpringSecurity中CsrfFilter过滤器说明

```
public final class CsrfFilter extends OncePerRequestFilter {
    public static final RequestMatcher DEFAULT_CSRF_MATCHER = new
CsrfFilter.DefaultRequiresCsrfMatcher();
    private final Log logger = LoggerFactory.getLog(this.getClass());
    private final CsrfTokenRepository tokenRepository;
    private RequestMatcher requireCsrfProtectionMatcher;
    private AccessDeniedHandler accessDeniedHandler;

    public CsrfFilter(CsrfTokenRepository csrfTokenRepository) {
        this.requireCsrfProtectionMatcher = DEFAULT_CSRF_MATCHER;
        this.accessDeniedHandler = new AccessDeniedHandlerImpl();
        Assert.notNull(csrfTokenRepository, "csrfTokenRepository cannot be null");
        this.tokenRepository = csrfTokenRepository;
    }
    //通过这里可以看出SpringSecurity的csrf机制把请求方式分成两类来处理
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException {
        request.setAttribute(HttpServletResponse.class.getName(), response);
        CsrfToken csrfToken = this.tokenRepository.loadToken(request);
        boolean missingToken = csrfToken == null;

        if (missingToken) {
```

```

        csrfToken = this.tokenRepository.generateToken(request);
        this.tokenRepository.saveToken(csrfToken, request, response);
    }

    request.setAttribute(CsrfToken.class.getName(), csrfToken);
    request.setAttribute(csrfToken.getParameterName(), csrfToken);
    //第一类: "GET", "HEAD", "TRACE", "OPTIONS"四类请求可以直接通过
    if (!this.requireCsrfProtectionMatcher.matches(request)) {
        filterChain.doFilter(request, response);
    } else {
        //第二类: 除去上面四类, 包括POST都要被验证携带token才能通过
        String actualToken = request.getHeader(csrfToken.getHeaderName());
        if (actualToken == null) {
            actualToken = request.getParameter(csrfToken.getParameterName());
        }

        if (!csrfToken.getToken().equals(actualToken)) {
            if (this.logger.isDebugEnabled()) {
                this.logger.debug("Invalid CSRF token found for " +
                    UrlUtils.buildFullRequestUrl(request));
            }

            if (missingToken) {
                this.accessDeniedHandler.handle(request, response, new
                    MissingCsrfTokenException(actualToken));
            } else {
                this.accessDeniedHandler.handle(request, response, new
                    InvalidCsrfTokenException(csrfToken, actualToken));
            }

        } else {
            filterChain.doFilter(request, response);
        }
    }
}

public void setRequireCsrfProtectionMatcher(RequestMatcher requireCsrfProtectionMatcher) {
    Assert.notNull(requireCsrfProtectionMatcher, "requireCsrfProtectionMatcher cannot be
        null");
    this.requireCsrfProtectionMatcher = requireCsrfProtectionMatcher;
}

public void setAccessDeniedHandler(AccessDeniedHandler accessDeniedHandler) {
    Assert.notNull(accessDeniedHandler, "accessDeniedHandler cannot be null");
    this.accessDeniedHandler = accessDeniedHandler;
}

private static final class DefaultRequiresCsrfMatcher implements RequestMatcher {
    private final HashSet<String> allowedMethods;

    private DefaultRequiresCsrfMatcher() {
        this.allowedMethods = new HashSet(Arrays.asList("GET", "HEAD", "TRACE", "OPTIONS"));
    }
}

```

```
public boolean matches(HttpServletRequest request) {  
    return !this.allowedMethods.contains(request.getMethod());  
}  
}  
}
```

通过源码分析，我们明白了，自己的认证页面，请求方式为POST，但却没有携带token，所以才出现了403权限不足的异常。那么如何处理这个问题呢？

方式一：直接禁用csrf，不推荐。

方式二：在认证页面携带token请求。

5.2.2 禁用csrf防护机制

在SpringSecurity主配置文件中添加禁用csrf防护的配置。

```
<!--设置可以用spring的el表达式配置Spring Security并自动生成对应配置组件（过滤器）-->  
<security:http auto-config="true" use-expressions="true">  
    <!--指定login.jsp页面可以被匿名访问-->  
    <security:intercept-url pattern="/login.jsp" access="permitAll()"/>  
    <!--使用spring的el表达式来指定项目所有资源访问都必须有ROLE_USER或ROLE_ADMIN角色-->  
    <security:intercept-url pattern="/**" access="hasAnyRole('ROLE_USER','ROLE_ADMIN')"/>  
    <!--指定自定义的认证页面-->  
    <security:form-login login-page="/login.jsp"  
        login-processing-url="/login"  
        default-target-url="/index.jsp"  
        authentication-failure-url="/failer.jsp"/>  
    <!--指定退出登录后跳转的页面-->  
    <security:logout logout-url="/logout"  
        logout-success-url="/login.jsp"/>  
    <!--禁用csrf防护机制-->  
    <security:csrf disabled="true"/>  
</security:http>
```

5.2.3 在认证页面携带token请求

```
login.jsp
1 <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
2 <%@taglib uri="http://www.springframework.org/security/tags" prefix="security"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head...>
26
27 <body class="hold-transition login-page">
28   <div class="login-box">
29     <div class="login-logo">
30       <a href="#"><b>ITCAST</b>后台管理系统</a>
31     </div>
32     <!-- /.login-logo -->
33     <div class="login-box-body">
34       <p class="login-box-msg">登录系统</p>
35       <form action="${pageContext.request.contextPath}/login" method="post">
36         <security:csrfInput/ > 在认证form表单内携带token
37         <div class="form-group has-feedback">
38           <input type="text" name="username" class="form-control"
39             placeholder="用户名"> <span
40             class="glyphicon glyphicon-envelope form-control-feedback"></span>
```

注：HttpSessionCsrfTokenRepository对象负责生成token并放入session域中。

六、SpringSecurity使用数据库数据完成认证

6.1 认证流程分析

UsernamePasswordAuthenticationFilter

先看主要负责认证的过滤器UsernamePasswordAuthenticationFilter，有删减，注意注释。

```
public class UsernamePasswordAuthenticationFilter extends AbstractAuthenticationProcessingFilter
{
    public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "username";
    public static final String SPRING_SECURITY_FORM_PASSWORD_KEY = "password";
    private String usernameParameter = "username";
    private String passwordParameter = "password";
    private boolean postOnly = true;

    public UsernamePasswordAuthenticationFilter() {
        super(new AntPathRequestMatcher("/login", "POST"));
    }

    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse
    response) throws AuthenticationException {
        //必须为POST请求
        if (this.postOnly && !request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException("Authentication method not supported: " +
            request.getMethod());
        } else {
```

```

        String username = this.obtainUsername(request);
        String password = this.obtainPassword(request);
        if (username == null) {
            username = "";
        }

        if (password == null) {
            password = "";
        }

        username = username.trim();
        //将填写的用户名和密码封装到了UsernamePasswordAuthenticationToken中
        UsernamePasswordAuthenticationToken authRequest = new
UsernamePasswordAuthenticationToken(username, password);
        this.setDetails(request, authRequest);
        //调用AuthenticationManager对象实现认证
        return this.getAuthenticationManager().authenticate(authRequest);
    }
}
}

```

AuthenticationManager

由上面源码得知，真正认证操作在AuthenticationManager里面！

然后看AuthenticationManager的实现类ProviderManager：

```

public class ProviderManager implements AuthenticationManager, MessageSourceAware,
InitializingBean {
    private static final Log logger = LogFactory.getLog(ProviderManager.class);
    private AuthenticationEventPublisher eventPublisher;
    private List<AuthenticationProvider> providers;
    protected MessageSourceAccessor messages;
    private AuthenticationManager parent;
    private boolean eraseCredentialsAfterAuthentication;

    //注意AuthenticationProvider这个对象，SpringSecurity针对每一种认证，什么qq登录啊，
    //用户名密码登陆啊，微信登录啊都封装了一个AuthenticationProvider对象。
    public ProviderManager(List<AuthenticationProvider> providers) {
        this(providers, (AuthenticationManager)null);
    }

    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
        Class<? extends Authentication> toTest = authentication.getClass();
        AuthenticationException lastException = null;
        AuthenticationException parentException = null;
        Authentication result = null;
        Authentication parentResult = null;
        boolean debug = logger.isDebugEnabled();
        Iterator var8 = this.getProviders().iterator();

        //循环所有AuthenticationProvider，匹配当前认证类型。
    }
}

```



```

        while(var8.hasNext()) {
            AuthenticationProvider provider = (AuthenticationProvider)var8.next();
            if (provider.supports(toTest)) {
                if (debug) {
                    logger.debug("Authentication attempt using " +
provider.getClass().getName());
                }
                try {
                    //找到了对应认证类型就继续调用AuthenticationProvider对象完成认证业务。
                    result = provider.authenticate(authentication);
                    if (result != null) {
                        this.copyDetails(authentication, result);
                        break;
                    }
                } catch (AccountStatusException var13) {
                    this.prepareException(var13, authentication);
                    throw var13;
                } catch (InternalAuthenticationServiceException var14) {
                    this.prepareException(var14, authentication);
                    throw var14;
                } catch (AuthenticationException var15) {
                    lastException = var15;
                }
            }
        }

        if (result == null && this.parent != null) {
            try {
                result = parentResult = this.parent.authenticate(authentication);
            } catch (ProviderNotFoundException var11) {
            } catch (AuthenticationException var12) {
                parentException = var12;
                lastException = var12;
            }
        }

        if (result != null) {
            if (this.eraseCredentialsAfterAuthentication && result instanceof
CredentialsContainer) {
                ((CredentialsContainer)result).eraseCredentials();
            }

            if (parentResult == null) {
                this.eventPublisher.publishAuthenticationSuccess(result);
            }

            return result;
        } else {
            if (lastException == null) {
                lastException = new
ProviderNotFoundException(this.messages.getMessage("ProviderManager.providerNotFound", new
Object[] {toTest.getName()}), "No AuthenticationProvider found for {0}");
            }
        }
    }

```

```

        if (parentException == null) {
            this.prepareException((AuthenticationException)lastException, authentication);
        }

        throw lastException;
    }
}

```

AbstractUserDetailsAuthenticationProvider

咱们继续再找到AuthenticationProvider的实现类AbstractUserDetailsAuthenticationProvider:

```

public class DaoAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider {
    private static final String USER_NOT_FOUND_PASSWORD = "userNotFoundPassword";
    private PasswordEncoder passwordEncoder;
    private volatile String userNotFoundEncodedPassword;
    private UserDetailsService userDetailsService;
    private UserDetailsPasswordService userDetailsPasswordService;

    protected final UserDetails retrieveUser(String username,
UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {
        this.prepareTimingAttackProtection();
        try {
            //重点来了！主要就在这里了！
            //可别忘了，咱们为什么要翻源码，是想用自己数据库中的数据实现认证操作啊！
            //UserDetails就是SpringSecurity自己的用户对象。
            //this.getUserDetailsService()其实就是得到UserDetailsService的一个实现类
            //loadUserByUsername里面就是真正的认证逻辑
            //也就是说我们可以直接编写一个UserDetailsService的实现类，告诉SpringSecurity就可以了！
            //loadUserByUsername方法中只需要返回一个UserDetails对象即可
            UserDetails loadedUser = this.getUserDetailsService().loadUserByUsername(username);

            //若返回null，就抛出异常，认证失败。
            if (loadedUser == null) {
                throw new InternalAuthenticationServiceException("UserDetailsService returned
null, which is an interface contract violation");
            } else {
                //若有得到了UserDetails对象，返回即可。
                return loadedUser;
            }
        } catch (UsernameNotFoundException var4) {
            this.mitigateAgainstTimingAttack(authentication);
            throw var4;
        } catch (InternalAuthenticationServiceException var5) {
            throw var5;
        } catch (Exception var6) {
            throw new InternalAuthenticationServiceException(var6.getMessage(), var6);
        }
    }
}

```

```
}
```

AbstractUserDetailsAuthenticationProvider中authenticate返回值

按理说到此已经知道自定义认证方法的怎么写了，但咱们把返回的流程也大概走一遍，上面不是说到返回了一个UserDetails对象对象吗？跟着它，就又回到了AbstractUserDetailsAuthenticationProvider对象中authenticate方法的最后一行了。

```
public abstract class AbstractUserDetailsAuthenticationProvider implements
AuthenticationProvider, InitializingBean, MessageSourceAware {

    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {

        //最后一行返回值，调用了createSuccessAuthentication方法，此方法就在下面！
        return this.createSuccessAuthentication(principalToReturn, authentication, user);
    }

    //诶！？怎么又封装了一次UsernamePasswordAuthenticationToken，开局不是已经封装过了吗？
    protected Authentication createSuccessAuthentication(Object principal, Authentication
authentication, UserDetails user) {
        //那就从构造方法点进去看看，这才干啥了。
        UsernamePasswordAuthenticationToken result = new
UsernamePasswordAuthenticationToken(principal, authentication.getCredentials(),
this.authoritiesMapper.mapAuthorities(user.getAuthorities()));
        result.setDetails(authentication.getDetails());
        return result;
    }
}
```

UsernamePasswordAuthenticationToken

来到UsernamePasswordAuthenticationToken对象发现里面有两个构造方法

```
public class UsernamePasswordAuthenticationToken extends AbstractAuthenticationToken {
    private static final long serialVersionUID = 510L;
    private final Object principal;
    private Object credentials;

    //认证成功前，调用的是这个带有两个参数的。
    public UsernamePasswordAuthenticationToken(Object principal, Object credentials) {
        super((Collection)null);
        this.principal = principal;
        this.credentials = credentials;
        this.setAuthenticated(false);
    }

    //认证成功后，调用的是这个带有三个参数的。

    public UsernamePasswordAuthenticationToken(Object principal, Object credentials,
```

```

Collection<? extends GrantedAuthority> authorities) {
    //看看父类干了什么!
    super(authorities);
    this.principal = principal;
    this.credentials = credentials;
    super.setAuthenticated(true);
}
}

```

AbstractAuthenticationToken

再点进去super(authorities)看看:

```

public abstract class AbstractAuthenticationToken implements Authentication,
CredentialsContainer {
    private final Collection<GrantedAuthority> authorities;
    private Object details;
    private boolean authenticated = false;

    public AbstractAuthenticationToken(Collection<? extends GrantedAuthority> authorities) {
        //这时两个参数那个分支!
        if (authorities == null) {
            this.authorities = AuthorityUtils.NO_AUTHORITIES;
        } else {
            //三个参数的, 看这里!
            Iterator var2 = authorities.iterator();
            //原来是多个了添加权限信息的步骤
            GrantedAuthority a;
            do {
                if (!var2.hasNext()) {
                    ArrayList<GrantedAuthority> temp = new ArrayList(authorities.size());
                    temp.addAll(authorities);
                    this.authorities = Collections.unmodifiableList(temp);
                    return;
                }
                a = (GrantedAuthority)var2.next();
            } while(a != null);
            //若没有权限信息, 是会抛出异常的!
            throw new IllegalArgumentException("Authorities collection cannot contain any null
elements");
        }
    }
}

```

由此, 咱们需要牢记自定义认证业务逻辑返回的UserDetails对象中一定要放置权限信息啊!

现在可以结束源码分析了吧? 先不要着急!

咱们回到最初的地方UsernamePasswordAuthenticationFilter, 你好好看了, 这可是个过滤器, 咱们分析这么久, 都没提到doFilter方法, 你不觉得心里不踏实?

可是这里面也没有doFilter呀? 那就从父类找!

AbstractAuthenticationProcessingFilter

点开AbstractAuthenticationProcessingFilter, 删掉不必要的代码!

```
public abstract class AbstractAuthenticationProcessingFilter extends GenericFilterBean
implements ApplicationEventPublisherAware, MessageSourceAware {

    //doFilter再次!
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws
IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest)req;
        HttpServletResponse response = (HttpServletResponse)res;
        if (!this.requiresAuthentication(request, response)) {
            chain.doFilter(request, response);
        } else {
            if (this.logger.isDebugEnabled()) {
                this.logger.debug("Request is to process authentication");
            }
            Authentication authResult;
            try {
                authResult = this.attemptAuthentication(request, response);
                if (authResult == null) {
                    return;
                }

                this.sessionStrategy.onAuthentication(authResult, request, response);
            } catch (InternalAuthenticationServiceException var8) {
                this.logger.error("An internal error occurred while trying to authenticate the
user.", var8);
                this.unsuccessfulAuthentication(request, response, var8);
                return;
            } catch (AuthenticationException var9) {
                this.unsuccessfulAuthentication(request, response, var9);
                return;
            }

            if (this.continueChainBeforeSuccessfulAuthentication) {
                chain.doFilter(request, response);
            }

            this.successfulAuthentication(request, response, chain, authResult);
        }
    }

    protected boolean requiresAuthentication(HttpServletRequest request, HttpServletResponse
response) {
        return this.requiresAuthenticationRequestMatcher.matches(request);
    }

    //成功走successfulAuthentication
    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse
```

```

response, FilterChain chain, Authentication authResult) throws IOException, ServletException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Authentication success. Updating SecurityContextHolder to
contain: " + authResult);
    }
    //认证成功, 将认证信息存储到SecurityContext中!
    SecurityContextHolder.getContext().setAuthentication(authResult);
    //登录成功调用rememberMeServices
    this.rememberMeServices.loginSuccess(request, response, authResult);
    if (this.eventPublisher != null) {
        this.eventPublisher.publishEvent(new
InteractiveAuthenticationSuccessEvent(authResult, this.getClass()));
    }

    this.successHandler.onAuthenticationSuccess(request, response, authResult);
}
//失败走unsuccessfulAuthentication
protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse
response, AuthenticationException failed) throws IOException, ServletException {
    SecurityContextHolder.clearContext();
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Authentication request failed: " + failed.toString(), failed);
        this.logger.debug("Updated SecurityContextHolder to contain null Authentication");
        this.logger.debug("Delegating to authentication failure handler " +
this.failureHandler);
    }

    this.rememberMeServices.loginFail(request, response);
    this.failureHandler.onAuthenticationFailure(request, response, failed);
}
}
}

```

可见AbstractAuthenticationProcessingFilter这个过滤器对于认证成功与否, 做了两个分支, 成功执行successfulAuthentication, 失败执行unsuccessfulAuthentication。

在successfulAuthentication内部, 将认证信息存储到了SecurityContext中。并调用了loginSuccess方法, 这就是常见的“记住我”功能! 此功能具体应用, 咱们后续再研究!

6.2 初步实现认证功能

6.2.1 让我们自己的UserService接口继承UserDetailsService

毕竟SpringSecurity是只认UserDetailsService的:

```
public interface UserService extends UserDetailsService {

    public void save(SysUser user);

    public List<SysUser> findAll();

    public Map<String, Object> toAddRolePage(Integer id);

    public void addRoleToUser(Integer userId, Integer[] ids);
}
```

6.2.2 编写loadUserByUsername业务

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    SysUser sysUser = userDao.findByName(username);
    if(sysUser==null){
        //若用户名不对, 直接返回null, 表示认证失败。
        return null;
    }
    List<SimpleGrantedAuthority> authorities = new ArrayList<>();
    List<SysRole> roles = sysUser.getRoles();
    for (SysRole role : roles) {
        authorities.add(new SimpleGrantedAuthority(role.getRoleName()));
    }
    //最终需要返回一个SpringSecurity的UserDetails对象, {noop}表示不加密认证。
    return new User(sysUser.getUsername(), "{noop}" + sysUser.getPassword(), authorities);
}
```

6.2.3 在SpringSecurity主配置文件中指定认证使用的业务对象

```
<!--设置Spring Security认证用户信息的来源-->
<security:authentication-manager>
    <security:authentication-provider user-service-ref="userServiceImpl">
        </security:authentication-provider>
    </security:authentication-manager>
```

6.3 加密认证

6.3.1 在IOC容器中提供加密对象

```

<!--加密对象-->
<bean id="passwordEncoder"
class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>
<!--设置Spring Security认证用户信息的来源-->
<security:authentication-manager>
    <security:authentication-provider user-service-ref="userServiceImpl">
        <!--指定认证使用的加密对象-->
        <security:password-encoder ref="passwordEncoder"/>
    </security:authentication-provider>
</security:authentication-manager>

```

6.3.2 修改认证方法

去掉{noop}

```

@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    SysUser sysUser = userDao.findByName(username);
    if(sysUser==null){
        //若用户名不对, 直接返回null, 表示认证失败。
        return null;
    }
    List<SimpleGrantedAuthority> authorities = new ArrayList<>();
    List<SysRole> roles = sysUser.getRoles();
    for (SysRole role : roles) {
        authorities.add(new SimpleGrantedAuthority(role.getRoleName()));
    }
    //最终需要返回一个SpringSecurity的UserDetails对象, {noop}表示不加密认证。
    return new User(sysUser.getUsername(), sysUser.getPassword(), authorities);
}

```

6.3.3 修改添加用户的操作

```

@Service
@Transactional
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    @Autowired
    private RoleService roleService;

    @Autowired
    private BCryptPasswordEncoder passwordEncoder;
    @Override
    public void save(SysUser user) {
        //对密码进行加密, 然后再入库
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        userDao.save(user);
    }
}

```



```
//.....  
}
```

6.3.4 手动将数据库中用户密码改为加密后的密文

1 结果 2 个配置文件 3 信息 4 表数据 5 信息			
<input type="checkbox"/>	id	username	password
<input type="checkbox"/>	2	xiaozhi	\$2a\$10\$c93ROC6RTt1rpS67NLiP5OhQRkLWVoGDylrRI
*	(Auto)	(NULL)	(NULL)