Spring Security3 源码分析

作者: Dead_knight http://dead-knight.iteye.com

Spring Security3 源码分析

目 录

1. Spring Security

1.1 Spring Security3源码分析-FilterChainProxy初始化 4

1.2 Spring Security3源码分析-http标签解析 13

1.3 Spring Security3源码分析-authentication-manager标签解析 17

1.4 Spring Security3源码分析-FilterChainProxy执行过程分析 23

1.5 Spring Security3源码分析-SecurityContextPersistenceFilter分析 26

1.6 Spring Security3源码分析-LogoutFilter分析 33

1.7 Spring Security3源码分析-UsernamePasswordAuthenticationFilter分析 39

1.8 Spring Security3源码分析-RememberMeAuthenticationFilter分析 50

1.9 Spring Security3源码分析-SecurityContextHolderAwareRequestFilter分析 57

1.10 Spring Security3源码分析-FilterSecurityInterceptor分析 60

1.11 Spring Security3源码分析-BasicAuthenticationFilter分析 68

1.12 Spring Security3源码分析-AnonymousAuthenticationFilter分析 71

1.13 Spring Security3源码分析-SessionManagementFilter分析-上 73

1.14 Spring Security3源码分析-SessionManagementFilter分析-下 84

1.15 Spring Security3源码分析-ExceptionTranslationFilter分析 87

1.16 Spring Security3源码分析-RequestCacheAwareFilter分析 92

1.17 Spring Security3源码分析-Filter链排序分析 96

1.18 Spring Security3源码分析-认证授权分析 103

1.19 Spring Security3源码分析-SSL支持 106

1.20 Spring Security3源码分析-CAS支持 111

1.1 Spring Security3源码分析-FilterChainProxy初始化

发表时间: 2012-05-04 关键字: security, spring

很久没有更新博客了，最近对Spring Security做了比较深入的研究。
spring security的教程网上很多：
http://lengyun3566.iteye.com/category/153689
http://wenku.baidu.com/view/b0c0dc0b79563c1ec5da7179.html

以上教程足够应付在实际项目中使用spring security这一安全框架了。如果想深入研究下，网上的资料就很少了，比如：
http://www.blogjava.net/SpartaYew/archive/2011/05/19/spingsecurity3.html
http://www.blogjava.net/youxia/archive/2008/12/07/244883.html
http://www.cnblogs.com/hzhuxin/archive/2011/12/19/2293730.html

但还是没有从filter配置开始进行一步一步分析。
带着很多疑问，逐步拨开spring security3的面纱
spring security在web.xml中的配置为

```
    <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>
```

一看就知道，这是spring的类，这个类位于org.springframework.web-3.0.1.RELEASE.jar这个jar下面，说明这个类本身是和springSecurity无关。DelegatingFilterProxy类继承于抽象类GenericFilterBean,间接地实现了javax.servlet.Filter接口。细节方面就不一一讲述了。看doFilter方法

```
        public void doFilter(ServletRequest request, ServletResponse response, FilterChain filt
                        throws ServletException, IOException {

                // Lazily initialize the delegate if necessary.
                Filter delegateToUse = null;
```

```
            synchronized (this.delegateMonitor) {
                if (this.delegate == null) {
                    WebApplicationContext wac = findWebApplicationContext();
                    if (wac == null) {
                        throw new IllegalStateException("No WebApplicationConte
                    }
                    this.delegate = initDelegate(wac);
                }
                delegateToUse = this.delegate;
            }

            // Let the delegate perform the actual doFilter operation.
            invokeDelegate(delegateToUse, request, response, filterChain);
        }
```

这里做了两件事：

一、initDelegate(wac);//初始化FilterChainProxy

```
        protected Filter initDelegate(WebApplicationContext wac) throws ServletException {
            Filter delegate = wac.getBean(getTargetBeanName(), Filter.class);
            if (isTargetFilterLifecycle()) {
                delegate.init(getFilterConfig());
            }
            return delegate;
        }
```

getTargetBeanName()返回的是Filter的name:springSecurityFilterChain

Filter delegate = wac.getBean(getTargetBeanName(), Filter.class);

这里根据springSecurityFilterChain的bean name直接获取FilterChainProxy的实例。

这里大家会产生疑问，springSecurityFilterChain这个bean在哪里定义的呢？

此时似乎忽略了spring security的bean配置文件了

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xmlns:beans="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/security
       http://www.springframework.org/schema/security/
              spring-security-3.0.xsd">
  <http auto-config="true">
    <intercept-url pattern="/*" access="ROLE_USER"/>
  </http>
  <authentication-manager alias="authenticationManager">
    <authentication-provider>
      <user-service>
        <user authorities="ROLE_USER" name="guest" password="guest"/>
      </user-service>
    </authentication-provider>
  </authentication-manager>
</beans:beans>
```

这是最简单的配置了，同时也是解开springSecurityFilterChain这个bean没有定义的疑问了。
这里主要利用了spring的自定义标签。具体参见：
[url] http://www.w3school.com.cn/schema/schema_example.asp[/url]
首先spring security的标签解析部分的源码包为：
spring-security-config-3.0.2.RELEASE.jar
这个jar包的META-INF目录下面有spring.handlers,spring.schemas两个文件，其中spring.schemas文件主要是标签的规范、约束；而spring.handlers这个文件时真正解析自定义标签的类，这个文件的内容为：

```
http\://www.springframework.org/schema/security=org.springframework.security.config.SecurityNam
```

从这里可以看出来spring security的标签解析由
org.springframework.security.config.SecurityNamespaceHandler
来处理。该类实现接口：NamespaceHandler，spring中自定义标签都要实现该接口，这个接口有三个方法init、parse、decorate，其中init用于自定义标签的初始化，parse用于解析标签，decorate用于装饰。
SecurityNamespaceHandler类的init方法完成了标签解析类的注册工作

```
    public void init() {
        // Parsers
        parsers.put(Elements.LDAP_PROVIDER, new LdapProviderBeanDefinitionParser());
        parsers.put(Elements.LDAP_SERVER, new LdapServerBeanDefinitionParser());
        parsers.put(Elements.LDAP_USER_SERVICE, new LdapUserServiceBeanDefinitionParser());
        parsers.put(Elements.USER_SERVICE, new UserServiceBeanDefinitionParser());
        parsers.put(Elements.JDBC_USER_SERVICE, new JdbcUserServiceBeanDefinitionParser());
        parsers.put(Elements.AUTHENTICATION_PROVIDER, new AuthenticationProviderBeanDefinitionF
        parsers.put(Elements.GLOBAL_METHOD_SECURITY, new GlobalMethodSecurityBeanDefinitionPars
        parsers.put(Elements.AUTHENTICATION_MANAGER, new AuthenticationManagerBeanDefinitionPar
 //       registerBeanDefinitionDecorator(Elements.INTERCEPT_METHODS, new InterceptMethodsBeanD

        // Only load the web-namespace parsers if the web classes are available
        if (ClassUtils.isPresent("org.springframework.security.web.FilterChainProxy", getClass(
            parsers.put(Elements.HTTP, new HttpSecurityBeanDefinitionParser());
            parsers.put(Elements.FILTER_INVOCATION_DEFINITION_SOURCE, new FilterInvocationSecur
            parsers.put(Elements.FILTER_SECURITY_METADATA_SOURCE, new FilterInvocationSecurityM
            filterChainMapBDD = new FilterChainMapBeanDefinitionDecorator();
            //registerBeanDefinitionDecorator(Elements.FILTER_CHAIN_MAP, new FilterChainMapBear
        }
    }
```

可以看出，http的标签解析类注册代码为：

```
parsers.put(Elements.HTTP, new HttpSecurityBeanDefinitionParser());
```

authentication-manager的标签解析类注册代码为：

```
parsers.put(Elements.AUTHENTICATION_MANAGER, new AuthenticationManagerBeanDefinitionParser());
```

HttpSecurityBeanDefinitionParser的parse方法源码为：

```
public BeanDefinition parse(Element element, ParserContext pc) {
        CompositeComponentDefinition compositeDef =
            new CompositeComponentDefinition(element.getTagName(), pc.extractSource(element));
```

```
        pc.pushContainingComponent(compositeDef);
        final Object source = pc.extractSource(element);

        final String portMapperName = createPortMapper(element, pc);
        final UrlMatcher matcher = createUrlMatcher(element);

        HttpConfigurationBuilder httpBldr = new HttpConfigurationBuilder(element, pc, matcher,

        httpBldr.parseInterceptUrlsForEmptyFilterChains();
        httpBldr.createSecurityContextPersistenceFilter();
        httpBldr.createSessionManagementFilters();

        ManagedList<BeanReference> authenticationProviders = new ManagedList<BeanReference>();
        BeanReference authenticationManager = createAuthenticationManager(element, pc, authenti

        httpBldr.createServletApiFilter();
        httpBldr.createChannelProcessingFilter();
        httpBldr.createFilterSecurityInterceptor(authenticationManager);

        AuthenticationConfigBuilder authBldr = new AuthenticationConfigBuilder(element, pc,
                httpBldr.isAllowSessionCreation(), portMapperName);

        authBldr.createAnonymousFilter();
        authBldr.createRememberMeFilter(authenticationManager);
        authBldr.createRequestCache();
        authBldr.createBasicFilter(authenticationManager);
        authBldr.createFormLoginFilter(httpBldr.getSessionStrategy(), authenticationManager);
        authBldr.createOpenIDLoginFilter(httpBldr.getSessionStrategy(), authenticationManager);
        authBldr.createX509Filter(authenticationManager);
        authBldr.createLogoutFilter();
        authBldr.createLoginPageFilterIfNeeded();
        authBldr.createUserServiceInjector();
        authBldr.createExceptionTranslationFilter();

        List<OrderDecorator> unorderedFilterChain = new ArrayList<OrderDecorator>();

        unorderedFilterChain.addAll(httpBldr.getFilters());
```

```
        unorderedFilterChain.addAll(authBldr.getFilters());

        authenticationProviders.addAll(authBldr.getProviders());

        BeanDefinition requestCacheAwareFilter = new RootBeanDefinition(RequestCacheAwareFilter
        requestCacheAwareFilter.getPropertyValues().addPropertyValue("requestCache", authBldr.g
        unorderedFilterChain.add(new OrderDecorator(requestCacheAwareFilter, REQUEST_CACHE_FILT

        unorderedFilterChain.addAll(buildCustomFilterList(element, pc));

        Collections.sort(unorderedFilterChain, new OrderComparator());
        checkFilterChainOrder(unorderedFilterChain, pc, source);

        List<BeanMetadataElement> filterChain = new ManagedList<BeanMetadataElement>();

        for (OrderDecorator od : unorderedFilterChain) {
            filterChain.add(od.bean);
        }

        ManagedMap<BeanDefinition, List<BeanMetadataElement>> filterChainMap = httpBldr.getFilt
        BeanDefinition universalMatch = new RootBeanDefinition(String.class);
        universalMatch.getConstructorArgumentValues().addGenericArgumentValue(matcher.getUniver
        filterChainMap.put(universalMatch, filterChain);

        registerFilterChainProxy(pc, filterChainMap, matcher, source);

        pc.popAndRegisterContainingComponent();
        return null;
    }
```

很多spring security的教程都会说http标签配置了auto-config="true"属性，spring security就会自动配置好了过滤器链。但是这些过滤器是如何添加到链中的呢，教程没说。
但是上面的代码已经告诉我们，就在这里设置的

```
        httpBldr.createSecurityContextPersistenceFilter();
        httpBldr.createSessionManagementFilters();
```

```
        httpBldr.createServletApiFilter();
        httpBldr.createChannelProcessingFilter();
        httpBldr.createFilterSecurityInterceptor(authenticationManager);
        ……
        authBldr.createAnonymousFilter();
        authBldr.createRememberMeFilter(authenticationManager);
        authBldr.createRequestCache();
        authBldr.createBasicFilter(authenticationManager);
        authBldr.createFormLoginFilter(httpBldr.getSessionStrategy(), authenticationManager);
        authBldr.createOpenIDLoginFilter(httpBldr.getSessionStrategy(), authenticationManager);
        authBldr.createX509Filter(authenticationManager);
        authBldr.createLogoutFilter();
        authBldr.createLoginPageFilterIfNeeded();
        authBldr.createUserServiceInjector();
        authBldr.createExceptionTranslationFilter();
```

具体create分析下一篇再细说。接下来完成Filter的排序、并添加到filterChainMap集合中

```
List<OrderDecorator> unorderedFilterChain = new ArrayList<OrderDecorator>();

        unorderedFilterChain.addAll(httpBldr.getFilters());
        unorderedFilterChain.addAll(authBldr.getFilters());

        authenticationProviders.addAll(authBldr.getProviders());

        BeanDefinition requestCacheAwareFilter = new RootBeanDefinition(RequestCacheAwareFilter
        requestCacheAwareFilter.getPropertyValues().addPropertyValue("requestCache", authBldr.g
        unorderedFilterChain.add(new OrderDecorator(requestCacheAwareFilter, REQUEST_CACHE_FILT

        unorderedFilterChain.addAll(buildCustomFilterList(element, pc));

        Collections.sort(unorderedFilterChain, new OrderComparator());
        checkFilterChainOrder(unorderedFilterChain, pc, source);

        List<BeanMetadataElement> filterChain = new ManagedList<BeanMetadataElement>();
```

```
        for (OrderDecorator od : unorderedFilterChain) {
            filterChain.add(od.bean);
        }

        ManagedMap<BeanDefinition, List<BeanMetadataElement>> filterChainMap = httpBldr.getFilt
        BeanDefinition universalMatch = new RootBeanDefinition(String.class);
        universalMatch.getConstructorArgumentValues().addGenericArgumentValue(matcher.getUniver
        filterChainMap.put(universalMatch, filterChain);
```

此时，已经为FilterChainProxy提供了必须的参数，接下来就是该完成FilterChainProxy的bean定义过程了

```
registerFilterChainProxy(pc, filterChainMap, matcher, source);
```

```
    private void registerFilterChainProxy(ParserContext pc, Map<BeanDefinition, List<BeanMetada
        if (pc.getRegistry().containsBeanDefinition(BeanIds.FILTER_CHAIN_PROXY)) {
            pc.getReaderContext().error("Duplicate <http> element detected", source);
        }
        //定义FilterChainProxy的BeanDefinition构造对象
         BeanDefinitionBuilder fcpBldr = BeanDefinitionBuilder.rootBeanDefinition(FilterChainPr
        fcpBldr.getRawBeanDefinition().setSource(source);
        fcpBldr.addPropertyValue("matcher", matcher);
        fcpBldr.addPropertyValue("stripQueryStringFromUrls", Boolean.valueOf(matcher instanceof
        //注入过滤器链
        fcpBldr.addPropertyValue("filterChainMap", filterChainMap);
        BeanDefinition fcpBean = fcpBldr.getBeanDefinition();
        //注册bean
        pc.registerBeanComponent(new BeanComponentDefinition(fcpBean, BeanIds.FILTER_CHAIN_PROX
        //注册bean的alias，其中别名为springSecurityFilterChain
        pc.getRegistry().registerAlias(BeanIds.FILTER_CHAIN_PROXY, BeanIds.SPRING_SECURITY_FILT
    }
```

这里需要说明的是BeanDefinitionBuilder类，该类能够动态创建spring的bean，并通过ParserContext完成bean的注册，而不需要在xml中进行配置。
此时FilterChainProxy实例化过程已经完成。

二、invokeDelegate(delegateToUse, request, response, filterChain);
//调用代理类的doFilter方法

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

执行第一步获取的FilterChainProxy实例的doFilter方法。

通过以上分析，对FilterChainProxy如何产生的，以及Spring Security的标签是如何解析有了大体的认识。
具体标签的解析、Filter链的执行，下次再更新……

1.2 Spring Security3源码分析-http标签解析

发表时间: 2012-05-05 关键字: security, spring

在FilterChainProxy初始化的过程中，大概描述了标签解析的一些步骤，但不够详细

```
<http auto-config="true">
  <remember-me key="workweb" token-validity-seconds="3600" data-source-ref="dataSource"/>
  <form-login login-page="/login.jsp"/>
  <logout logout-success-url="/login.jsp"/>
  <intercept-url pattern="/*" access="ROLE_USER"/>
</http>
```

http标签的解析过程由类org.springframework.security.config.http.HttpSecurityBeanDefinitionParser解析。

```
public BeanDefinition parse(Element element, ParserContext pc) {
        CompositeComponentDefinition compositeDef =
            new CompositeComponentDefinition(element.getTagName(), pc.extractSource(element));
        pc.pushContainingComponent(compositeDef);
        final Object source = pc.extractSource(element);
        //portMapperName、matcher主要提供给SSL相关类使用
        final String portMapperName = createPortMapper(element, pc);
        final UrlMatcher matcher = createUrlMatcher(element);
        //http标签构造器，该构造函数中对intercept-url、create-session子标签
         //进行了预处理，并将所有的intercept-url信息放到List中。
        HttpConfigurationBuilder httpBldr = new HttpConfigurationBuilder(element, pc, matcher,
        //处理List中的intercept-url信息（如pattern、filters），并将结果放到
         //Map集合filterChainMap中
        httpBldr.parseInterceptUrlsForEmptyFilterChains();
        //创建过滤器SecurityContextPersistenceFilter
        httpBldr.createSecurityContextPersistenceFilter();
        //创建过滤器SessionManagementFilter
        httpBldr.createSessionManagementFilters();
        //新建一个空的provider集合
        ManagedList<BeanReference> authenticationProviders = new ManagedList<BeanReference>();
        //通过空的provider集合产生一个ProviderManager的bean定义
```

```
BeanReference authenticationManager = createAuthenticationManager(element, pc, authenti
//创建过滤器SecurityContextHolderAwareRequestFilter
httpBldr.createServletApiFilter();
//判断intercept-url标签是否有requires-channel属性，如果有，则创建过滤器
 //ChannelProcessingFilter
httpBldr.createChannelProcessingFilter();
//创建过滤器FilterSecurityInterceptor
//这个创建过程比较复杂，分别为：
 //1.需要判断是否使用表达式use-expressions
//2.解析intercept-url中的access等属性
 //3.RoleVoter、AffirmativeBased的定义.....
httpBldr.createFilterSecurityInterceptor(authenticationManager);
//下面是与认证有关的过滤器，HttpConfigurationBuilder，
 //AuthenticationConfigBuilder将解析的职责进行了分离
AuthenticationConfigBuilder authBldr = new AuthenticationConfigBuilder(element, pc,
        httpBldr.isAllowSessionCreation(), portMapperName);
//创建过滤器AnonymousAuthenticationFilter，并且构造了provider：
 //AnonymousAuthenticationProvider，供ProviderManager使用
authBldr.createAnonymousFilter();
//判断是否有remember-me标签，如果有，则创建过滤器
 //RememberMeAuthenticationFilter，并且构造了provider：
 //RememberMeAuthenticationProvider供ProviderManager使用
authBldr.createRememberMeFilter(authenticationManager);
//判断是否有request-cache标签，如果有，则构造ref指明的bean定义
 //如果没有，则构造HttpSessionRequestCache缓存
authBldr.createRequestCache();
//创建过滤器BasicAuthenticationFilter
authBldr.createBasicFilter(authenticationManager);
//创建LoginUrlAuthenticationEntryPoint，以及创建过滤器
 //UsernamePasswordAuthenticationFilter
authBldr.createFormLoginFilter(httpBldr.getSessionStrategy(), authenticationManager);
//判断是否使用了openid-login，如果有，则构造openId客户端
 //org.springframework.security.openid.OpenID4JavaConsumer
authBldr.createOpenIDLoginFilter(httpBldr.getSessionStrategy(), authenticationManager);
//判断是否使用了x509，如果有，则创建过滤器
 //X509AuthenticationFilter
authBldr.createX509Filter(authenticationManager);
```

```
//判断是否配置了logout，如果有，则创建过滤器LogoutFilter
authBldr.createLogoutFilter();
//判断是否配置login-page属性，如果没有，则创建过滤器
 //DefaultLoginPageGeneratingFilter，生成默认登录页面
authBldr.createLoginPageFilterIfNeeded();
//创建UserDetailsServiceInjectionBeanPostProcessor
//动态向x509、openID、rememberme服务注入UserDetailsService
 //主要使用了spring的BeanPostProcessor接口功能
authBldr.createUserServiceInjector();
//创建过滤器ExceptionTranslationFilter
authBldr.createExceptionTranslationFilter();

List<OrderDecorator> unorderedFilterChain = new ArrayList<OrderDecorator>();
//向FilterChain链中添加filters
unorderedFilterChain.addAll(httpBldr.getFilters());
unorderedFilterChain.addAll(authBldr.getFilters());
//向ProviderManager中添加provider
authenticationProviders.addAll(authBldr.getProviders());

BeanDefinition requestCacheAwareFilter = new RootBeanDefinition(RequestCacheAwareFilter
requestCacheAwareFilter.getPropertyValues().addPropertyValue("requestCache", authBldr.g
unorderedFilterChain.add(new OrderDecorator(requestCacheAwareFilter, REQUEST_CACHE_FILT
//添加自定义的Filter，也就是custom-filter标签定义的Filter
unorderedFilterChain.addAll(buildCustomFilterList(element, pc));
//对FilterChian链中的Filter进行排序，排序规则参见SecurityFilters枚举类
Collections.sort(unorderedFilterChain, new OrderComparator());
checkFilterChainOrder(unorderedFilterChain, pc, source);

List<BeanMetadataElement> filterChain = new ManagedList<BeanMetadataElement>();

for (OrderDecorator od : unorderedFilterChain) {
    filterChain.add(od.bean);
}

ManagedMap<BeanDefinition, List<BeanMetadataElement>> filterChainMap = httpBldr.getFilt
BeanDefinition universalMatch = new RootBeanDefinition(String.class);
universalMatch.getConstructorArgumentValues().addGenericArgumentValue(matcher.getUniver
```

```
        filterChainMap.put(universalMatch, filterChain);
        //构造FilterChainProxy的Bean
        registerFilterChainProxy(pc, filterChainMap, matcher, source);

        pc.popAndRegisterContainingComponent();
        return null;
    }
```

至此，大概http标签的解析已经差不多了，虽然每个Filter的BeanDefinition创建过程还没有一一细说,但基本步骤如下：

1.通过Filter的类路径获取BeanDefinitionBuilder对象，如
BeanDefinitionBuilder filterBuilder = BeanDefinitionBuilder.rootBeanDefinition(filterClassName);

2.解析xml标签属性，再通过BeanDefinitionBuilder的addPropertyValue、addPropertyReference等方法设置Filter对应BeanDefinition的属性值、依赖bean

3.注册BeanDefinition。通过
ParserContext.registerBeanComponent(
new BeanComponentDefinition(BeanDefinition,beanId));
完成bean的注册。还可以通过ParserContext.getRegistry().registerAlias
方法注册bean的别名

实际上，标签解析就是构造BeanDefinition，然后注册到bean factory中。而BeanDefinition就是Spring中定义bean的数据结构。

1.3 Spring Security3源码分析-authentication-manager标签解析

发表时间: 2012-05-05 关键字: spring, security

讲解完http标签的解析过程，authentication-manager标签解析部分就很容易理解了
authentication-manager标签在spring的配置文件中的定义一般如下

```
<authentication-manager alias="authenticationManager">
    <authentication-provider user-service-ref="userDetailsManager"/>
</authentication-manager>
```

authentication-manager标签的解析类是：
org.springframework.security.config.authentication.AuthenticationManagerBeanDefinitionParser
具体解析方法parse的代码为

```
public BeanDefinition parse(Element element, ParserContext pc) {
        Assert.state(!pc.getRegistry().containsBeanDefinition(BeanIds.AUTHENTICATION_MANAGER),
                "AuthenticationManager has already been registered!");
        pc.pushContainingComponent(new CompositeComponentDefinition(element.getTagName(), pc.ex
        //构造ProviderManager的BeanDefinition
        BeanDefinitionBuilder providerManagerBldr = BeanDefinitionBuilder.rootBeanDefinition(Pr
        //获取alias属性
        String alias = element.getAttribute(ATT_ALIAS);
        //检查session-controller-ref属性，提示通过标签<concurrent-session-control>取代
        checkForDeprecatedSessionControllerRef(element, pc);
        List<BeanMetadataElement> providers = new ManagedList<BeanMetadataElement>();
        NamespaceHandlerResolver resolver = pc.getReaderContext().getNamespaceHandlerResolver()
        //获取authentication-manager的子节点
        NodeList children = element.getChildNodes();
        //循环节点，一般子节点主要是authentication-provider或者
         //ldap-authentication-provider
        for (int i = 0; i < children.getLength(); i++) {
            Node node = children.item(i);
            if (node instanceof Element) {
                Element providerElt = (Element)node;
                //判断子标签是否有ref属性，如果有，则直接将ref属性
                   //引用的bean id添加到providers集合中
```

```
            if (StringUtils.hasText(providerElt.getAttribute(ATT_REF))) {
                providers.add(new RuntimeBeanReference(providerElt.getAttribute(ATT_REF)));
            } else {
                //如果没有ref属性，则通过子标签的解析类完成标签解析
                   //如子标签：authentication-provider,解析过程在后面
                BeanDefinition provider = resolver.resolve(providerElt.getNamespaceURI()).p
                Assert.notNull(provider, "Parser for " + providerElt.getNodeName() + " retu
                String id = pc.getReaderContext().generateBeanName(provider);
                //注册provider的BeanDefinition
                pc.registerBeanComponent(new BeanComponentDefinition(provider, id));
                //添加注册过的bean到provider集合中
                providers.add(new RuntimeBeanReference(id));
            }
        }
    }

    if (providers.isEmpty()) {
        providers.add(new RootBeanDefinition(NullAuthenticationProvider.class));
    }

    providerManagerBldr.addPropertyValue("providers", providers);
    //添加默认的事件发布类
    BeanDefinition publisher = new RootBeanDefinition(DefaultAuthenticationEventPublisher.c
    String id = pc.getReaderContext().generateBeanName(publisher);
    pc.registerBeanComponent(new BeanComponentDefinition(publisher, id));
    //将事件发布类的bean注入到ProviderManager的
     //authenticationEventPublisher属性中
    providerManagerBldr.addPropertyReference("authenticationEventPublisher", id);
    //注册ProviderManager的bean
    pc.registerBeanComponent(
            new BeanComponentDefinition(providerManagerBldr.getBeanDefinition(), BeanIds.AU

    if (StringUtils.hasText(alias)) {
        pc.getRegistry().registerAlias(BeanIds.AUTHENTICATION_MANAGER, alias);
        pc.getReaderContext().fireAliasRegistered(BeanIds.AUTHENTICATION_MANAGER, alias, pc
    }
```

```
        pc.popAndRegisterContainingComponent();

        return null;
    }
```

通过上面的代码片段，能够知道authentication-manager标签解析的步骤是

1.构造ProviderManager的BeanDefinition

2.循环authentication-manager的子标签，构造provider的BeanDefinition，并添加到providers集合中

3.将第2步的providers设置为ProviderManager的providers属性

4.构造异常事件发布类DefaultAuthenticationEventPublisher的BeanDefinition，并设置为ProviderManager的属性authenticationEventPublisher

5.通过registerBeanComponent方法完成bean的注册任务

authentication-provider标签的解析类为
org.springframework.security.config.authentication.AuthenticationProviderBeanDefinitionParser

```
    public BeanDefinition parse(Element element, ParserContext parserContext) {
        //首先构造DaoAuthenticationProvider的BeanDefinition
        RootBeanDefinition authProvider = new RootBeanDefinition(DaoAuthenticationProvider.clas
        authProvider.setSource(parserContext.extractSource(element));
        //获取password-encoder子标签
        Element passwordEncoderElt = DomUtils.getChildElementByTagName(element, Elements.PASSWC

        if (passwordEncoderElt != null) {
            //如果有password-encoder子标签，把解析任务交给
              //PasswordEncoderParser完成
            PasswordEncoderParser pep = new PasswordEncoderParser(passwordEncoderElt, parserCor
            authProvider.getPropertyValues().addPropertyValue("passwordEncoder", pep.getPasswor
            //如果有salt-source标签，将值注入到saltSource属性中
            if (pep.getSaltSource() != null) {
```

```
            authProvider.getPropertyValues().addPropertyValue("saltSource", pep.getSaltSour
        }
    }
    //下面获取子标签user-service、jdbc-user-service、ldap-user-service
    Element userServiceElt = DomUtils.getChildElementByTagName(element, Elements.USER_SERVI
    Element jdbcUserServiceElt = DomUtils.getChildElementByTagName(element, Elements.JDBC_U
    Element ldapUserServiceElt = DomUtils.getChildElementByTagName(element, Elements.LDAP_U

    String ref = element.getAttribute(ATT_USER_DETAILS_REF);

    if (StringUtils.hasText(ref)) {
        if (userServiceElt != null || jdbcUserServiceElt != null || ldapUserServiceElt != r
            parserContext.getReaderContext().error("The " + ATT_USER_DETAILS_REF + " attrib
                    "elements '" + Elements.USER_SERVICE + "', '" + Elements.JDBC_USER_SERV
                    Elements.LDAP_USER_SERVICE + "'", element);
        }
    } else {
        // Use the child elements to create the UserDetailsService
        AbstractUserDetailsServiceBeanDefinitionParser parser = null;
        Element elt = null;
        //下面的if语句，主要是根据子标签的不同，选择子标签对应的解析器处理
        if (userServiceElt != null) {
            elt = userServiceElt;
            parser = new UserServiceBeanDefinitionParser();
        } else if (jdbcUserServiceElt != null) {
            elt = jdbcUserServiceElt;
            parser = new JdbcUserServiceBeanDefinitionParser();
        } else if (ldapUserServiceElt != null) {
            elt = ldapUserServiceElt;
            parser = new LdapUserServiceBeanDefinitionParser();
        } else {
            parserContext.getReaderContext().error("A user-service is required", element);
        }

        parser.parse(elt, parserContext);
        ref = parser.getId();
        String cacheRef = elt.getAttribute(AbstractUserDetailsServiceBeanDefinitionParser.C
```

```
            if (StringUtils.hasText(cacheRef)) {
                authProvider.getPropertyValues().addPropertyValue("userCache", new RuntimeBeanF
            }
        }
        //将解析后的bean id注入到userDetailsService属性中
        authProvider.getPropertyValues().addPropertyValue("userDetailsService", new RuntimeBear
        return authProvider;
    }
```

如果学习过acegi的配置，应该知道，acegi有这么一段配置

```
   <bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">
      <property name="providers">
         <list>
            <ref local="daoAuthenticationProvider"/>
           <ref local="anonymousAuthenticationProvider"/>
         </list>
      </property>
   </bean>
```

实际上authentication-manager标签所要达到的目标就是构造上面的bean。其中anonymousAuthenticationProvider是在http解析过程添加的。

其实可以完全像acegi那样自定义每个bean。

```
<authentication-manager alias="authenticationManager">
    <authentication-provider user-service-ref="userDetailsManager"/>
</authentication-manager>
```

上面的标签如果用bean来定义，则可以完全由下面的xml来替代。

```
<bean id="org.springframework.security.authenticationManager" class="org.springframework.securi
        <property name="authenticationEventPublisher" ref="defaultAuthenticationEventPublisher"
```

```
    <property name="providers">
            <list>
                    <ref local="daoAuthenticationProvider"/>
                    <ref local="anonymousAuthenticationProvider"/>
            </list>
    </property>
</bean>

<bean id="defaultAuthenticationEventPublisher" class="org.springframework.security.authenti

<bean id="anonymousAuthenticationProvider" class="org.springframework.security.authenticati
    <property name="key"><value>work</value></property>
</bean>

<bean id="daoAuthenticationProvider" class="org.springframework.security.authentication.dac
    <property name="userDetailsService" ref="userDetailsManager"></property>
</bean>
```

需要注意的是anonymousAuthenticationProvider的bean中，需要增加key属性。如果采用authentication-manager标签的方式，key虽然没有定义，在增加AnonymousAuthenticationFilter过滤器中，是通过java.security.SecureRandom.nextLong()来生成的。

显而易见，如果采用bean的方式来定义，非常复杂，而且需要了解底层的组装过程才行，不过能够提高更大的扩展性。采用authentication-manager标签的方式，很简洁，只需要提供UserDetailsService即可。

1.4 Spring Security3源码分析-FilterChainProxy执行过程分析

发表时间: 2012-05-06 关键字: spring, security

通过FilterChainProxy的初始化、自定义标签的分析后，Spring Security需要的运行环境已经准备好了。

这样当用户访问应用时，过滤器就开始工作了。web.xml配置的Filter：org.springframework.web.filter.DelegatingFilterProxy就不介绍了，该类仅仅是初始化一个FilterChainProxy，然后把所有拦截的请求交给FilterChainProxy处理。

FilterChainProxy的doFilter方法如下

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
     //FilterInvocation类封装了request、response、chain
    //在方法中传递
     FilterInvocation fi = new FilterInvocation(request, response, chain);
    //获取http标签中创建的所有Filter
    List<Filter> filters = getFilters(fi.getRequestUrl());
    if (filters == null || filters.size() == 0) {
        if (logger.isDebugEnabled()) {
            logger.debug(fi.getRequestUrl() +
                    filters == null ? " has no matching filters" : " has an empty filter li
        }

        chain.doFilter(request, response);

        return;
    }
    //把实际doFilter任务交给VirtualFilterChain处理
    VirtualFilterChain virtualFilterChain = new VirtualFilterChain(fi, filters);
    virtualFilterChain.doFilter(fi.getRequest(), fi.getResponse());
}
```

现在来分析VirtualFilterChain的doFilter方法：

```
    private static class VirtualFilterChain implements FilterChain {
        private FilterInvocation fi;
        private List<Filter> additionalFilters;
        private int currentPosition = 0;

        private VirtualFilterChain(FilterInvocation filterInvocation, List<Filter> additionalFi
            this.fi = filterInvocation;
            this.additionalFilters = additionalFilters;
        }

        public void doFilter(ServletRequest request, ServletResponse response) throws IOExcepti
            //判断过滤器的个数是否与当前位置相等
            if (currentPosition == additionalFilters.size()) {
                if (logger.isDebugEnabled()) {
                    logger.debug(fi.getRequestUrl()
                        + " reached end of additional filter chain; proceeding with original ch
                }
                fi.getChain().doFilter(request, response);
            } else {
                //当前位置加一
                currentPosition++;
                //根据当前位置从过滤器列表中取出一个Filter
                Filter nextFilter = additionalFilters.get(currentPosition - 1);
                if (logger.isDebugEnabled()) {
                    logger.debug(fi.getRequestUrl() + " at position " + currentPosition + " of
                        + additionalFilters.size() + " in additional filter chain; firing Filte
                        + nextFilter + "'");
                }
               //执行取出Filter的doFilter方法
               nextFilter.doFilter(request, response, this);
            }
        }
    }
```

注意这里

```
nextFilter.doFilter(request, response, this);
```

VirtualFilterChain把自身作为参数传递给doFilter方法，这样doFilter方法最后会调用VirtualFilterChain的doFilter方法，这样控制就又回到了VirtualFilterChain，于是VirtualFilterChain又将当前位置currentPosition前移，调用下一个Filter的doFilter方法。当additionalFilters中所有元素的doFilter都执行完毕，VirtualFilterChain执行fi.getChain().doFilter，而fi.getChain()的值就是FilterChainProxy的doFilter方法中的参数chain的值。

这样我们就理解了FilterChainProxy是如何处理整个Filter链的了。

接下来，就要一一分析每个Filter的功能与作用了。

1.5 Spring Security3源码分析-SecurityContextPersistenceFilter分析

发表时间: 2012-05-06

通过观察Filter的名字，就能大概猜出来这个过滤器的作用，是的，持久化SecurityContext实例。这个过滤器位置是；
org.springframework.security.web.context.SecurityContextPersistenceFilter
废话不说，看源码

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;
    if (request.getAttribute(FILTER_APPLIED) != null) {
        // ensure that filter is only applied once per request
        chain.doFilter(request, response);
        return;
    }

    final boolean debug = logger.isDebugEnabled();

    request.setAttribute(FILTER_APPLIED, Boolean.TRUE);

    if (forceEagerSessionCreation) {
        HttpSession session = request.getSession();

        if (debug && session.isNew()) {
            logger.debug("Eagerly created session: " + session.getId());
        }
    }
    //将request、response对象交给HttpRequestResponseHolder维持
    HttpRequestResponseHolder holder = new HttpRequestResponseHolder(request, response);
    //通过SecurityContextRepository接口的实现类装载SecurityContext实例
    //HttpSessionSecurityContextRepository将产生SecurityContext实例的任务交给SecurityContextH
    //SecurityContextHolder再根据策略模式的不同，
    //把任务再交给相应策略类完成SecurityContext的创建
    //如果没有配置策略名称，则默认为
    //ThreadLocalSecurityContextHolderStrategy，
```

```
        //该类直接通过new SecurityContextImpl()创建实例
        SecurityContext contextBeforeChainExecution = repo.loadContext(holder);

        try {
                //将产生的SecurityContext再通过SecurityContextHolder->
          //ThreadLocalSecurityContextHolderStrategy设置到ThreadLocal中
            SecurityContextHolder.setContext(contextBeforeChainExecution);
            //继续把请求流向下一个过滤器执行
            chain.doFilter(holder.getRequest(), holder.getResponse());

        } finally {
                //先从SecurityContextHolder获取SecurityContext实例
            SecurityContext contextAfterChainExecution = SecurityContextHolder.getContext();
            // Crucial removal of SecurityContextHolder contents - do this before anything else
            //再把SecurityContext实例从SecurityContextHolder中清空
            SecurityContextHolder.clearContext();
            //将SecurityContext实例持久化到session中
            repo.saveContext(contextAfterChainExecution, holder.getRequest(), holder.getRespons
            request.removeAttribute(FILTER_APPLIED);
            if (debug) {
                logger.debug("SecurityContextHolder now cleared, as request processing complete
            }
        }
    }
```

通过源码中的注释，应该可以看出来，这个Filter的作用主要是创建一个空的SecurityContext（如果session中没有SecurityContext实例），然后持久化到session中。
接下来看看repo.loadContext(holder);代码：

```
    public SecurityContext loadContext(HttpRequestResponseHolder requestResponseHolder) {
        HttpServletRequest request = requestResponseHolder.getRequest();
        HttpServletResponse response = requestResponseHolder.getResponse();
        HttpSession httpSession = request.getSession(false);
        //从session中获取SecurityContext
         SecurityContext context = readSecurityContextFromSession(httpSession);
        //如果获取不到SecurityContext，新建一个空的SecurityContext实例
```

```
        if (context == null) {
            if (logger.isDebugEnabled()) {
                logger.debug("No SecurityContext was available from the HttpSession: " + httpSe
                        "A new one will be created.");
            }
            context = generateNewContext();

        }
        //这里需要注意一下，response装饰器类重新包装了response
        requestResponseHolder.setResponse(new SaveToSessionResponseWrapper(response, request,
                httpSession != null, context.hashCode()));

        return context;
    }
```

进一步分析generateNewContext方法

```
    SecurityContext generateNewContext() {
        SecurityContext context = null;
        //创建SecurityContext实例并返回
        if (securityContextClass == null) {
            context = SecurityContextHolder.createEmptyContext();

            return context;
        }

        try {
            context = securityContextClass.newInstance();
        } catch (Exception e) {
            ReflectionUtils.handleReflectionException(e);
        }
        return context;
    }
```

实际上，SecurityContextHolder类也是把创建SecurityContext任务交给具体的SecurityContextHolderStrategy实现类处理，SecurityContextHolder类有一个静态初始化过程

```
    static {
        initialize();
    }

.....
    private static void initialize() {
        if ((strategyName == null) || "".equals(strategyName)) {
            // Set default
            strategyName = MODE_THREADLOCAL;
        }
        //默认的SecurityContextHolderStrategy实现类为
         //ThreadLocalSecurityContextHolderStrategy
        if (strategyName.equals(MODE_THREADLOCAL)) {
            strategy = new ThreadLocalSecurityContextHolderStrategy();
        } else if (strategyName.equals(MODE_INHERITABLETHREADLOCAL)) {
            strategy = new InheritableThreadLocalSecurityContextHolderStrategy();
        } else if (strategyName.equals(MODE_GLOBAL)) {
            strategy = new GlobalSecurityContextHolderStrategy();
        } else {
            // Try to load a custom strategy
            try {
                Class<?> clazz = Class.forName(strategyName);
                Constructor<?> customStrategy = clazz.getConstructor();
                strategy = (SecurityContextHolderStrategy) customStrategy.newInstance();
            } catch (Exception ex) {
                ReflectionUtils.handleReflectionException(ex);
            }
        }

        initializeCount++;
    }
```

现在来看ThreadLocalSecurityContextHolderStrategy源码

```
final class ThreadLocalSecurityContextHolderStrategy implements SecurityContextHolderStrategy {
    //~ Static fields/initializers =====================================================

    private static final ThreadLocal<SecurityContext> contextHolder = new ThreadLocal<SecurityC

    //~ Methods ========================================================================

    public void clearContext() {
        contextHolder.set(null);
    }

    public SecurityContext getContext() {
        SecurityContext ctx = contextHolder.get();

        if (ctx == null) {
            ctx = createEmptyContext();
            contextHolder.set(ctx);
        }

        return ctx;
    }

    public void setContext(SecurityContext context) {
        Assert.notNull(context, "Only non-null SecurityContext instances are permitted");
        contextHolder.set(context);
    }
    //直接new一个SecurityContextImpl对象，
     //SecurityContextImpl类实现SecurityContext接口
    public SecurityContext createEmptyContext() {
        return new SecurityContextImpl();
    }
}
```

分析到这里，整个过程也清楚了。不过在filter原路返回时，还需要保存这个SecurityContext实例到session中，并且通过SecurityContextHolder将ThreadLocalSecurityContextHolderStrategy中ThreadLocal维持的SecurityContext实例清空。

```
        //将SecurityContext实例持久化到session中
          repo.saveContext(contextAfterChainExecution, holder.getRequest(), holder.getRespons
```

```
public void saveContext(SecurityContext context, HttpServletRequest request, HttpServletRes
    //由于之前response装饰器类SaveToSessionResponseWrapper
    //重新装饰了response
    SaveContextOnUpdateOrErrorResponseWrapper responseWrapper = (SaveContextOnUpdateOrError
    // saveContext() might already be called by the response wrapper
    // if something in the chain called sendError() or sendRedirect(). This ensures we only
    // once per request.
    if (!responseWrapper.isContextSaved() ) {
        //SaveToSessionResponseWrapper保存SecurityContext实例
        responseWrapper.saveContext(context);
    }
}
```

SaveToSessionResponseWrapper的saveContext方法源码：

```
    protected void saveContext(SecurityContext context) {
        // See SEC-776
        if (authenticationTrustResolver.isAnonymous(context.getAuthentication())) {
            if (logger.isDebugEnabled()) {
                logger.debug("SecurityContext contents are anonymous - context will not be
            }
            return;
        }

        HttpSession httpSession = request.getSession(false);

        if (httpSession == null) {
            httpSession = createNewSessionIfAllowed(context);
        }
```

```
            // If HttpSession exists, store current SecurityContextHolder contents but only if
            // the SecurityContext has actually changed (see JIRA SEC-37)
            if (httpSession != null && context.hashCode() != contextHashBeforeChainExecution) {
                //保存SecurityContext到session中
                httpSession.setAttribute(SPRING_SECURITY_CONTEXT_KEY, context);

                if (logger.isDebugEnabled()) {
                    logger.debug("SecurityContext stored to HttpSession: '" + context + "'");
                }
            }
        }
```

1.6 Spring Security3源码分析-LogoutFilter分析

发表时间: 2012-05-06

LogoutFilter过滤器对应的类路径为
org.springframework.security.web.authentication.logout.LogoutFilter
通过这个类的源码可以看出，这个类有两个构造函数

```
    public LogoutFilter(LogoutSuccessHandler logoutSuccessHandler, LogoutHandler... handlers) {
        Assert.notEmpty(handlers, "LogoutHandlers are required");
        this.handlers = Arrays.asList(handlers);
        Assert.notNull(logoutSuccessHandler, "logoutSuccessHandler cannot be null");
        this.logoutSuccessHandler = logoutSuccessHandler;
    }

    public LogoutFilter(String logoutSuccessUrl, LogoutHandler... handlers) {
        Assert.notEmpty(handlers, "LogoutHandlers are required");
        this.handlers = Arrays.asList(handlers);
        Assert.isTrue(!StringUtils.hasLength(logoutSuccessUrl) ||
                UrlUtils.isValidRedirectUrl(logoutSuccessUrl), logoutSuccessUrl + " isn't a val
        SimpleUrlLogoutSuccessHandler urlLogoutSuccessHandler = new SimpleUrlLogoutSuccessHandl
        if (StringUtils.hasText(logoutSuccessUrl)) {
            urlLogoutSuccessHandler.setDefaultTargetUrl(logoutSuccessUrl);
        }
        logoutSuccessHandler = urlLogoutSuccessHandler;
    }
```

这两个构造函数的参数，是从哪里传递的呢？没错，就是之前解析http标签通过创建LogoutFilter过滤器的bean定义时通过构造参数注入进来的。下面的部分源码为LogoutFilter的bean定义

```
public BeanDefinition parse(Element element, ParserContext pc) {
        String logoutUrl = null;
        String successHandlerRef = null;
        String logoutSuccessUrl = null;
        String invalidateSession = null;

        BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(LogoutFilter.c
```

```
if (element != null) {
    //分别解析logout标签的属性
    Object source = pc.extractSource(element);
    builder.getRawBeanDefinition().setSource(source);
    logoutUrl = element.getAttribute(ATT_LOGOUT_URL);
    successHandlerRef = element.getAttribute(ATT_LOGOUT_HANDLER);
    WebConfigUtils.validateHttpRedirect(logoutUrl, pc, source);
    logoutSuccessUrl = element.getAttribute(ATT_LOGOUT_SUCCESS_URL);
    WebConfigUtils.validateHttpRedirect(logoutSuccessUrl, pc, source);
    invalidateSession = element.getAttribute(ATT_INVALIDATE_SESSION);
}

if (!StringUtils.hasText(logoutUrl)) {
    logoutUrl = DEF_LOGOUT_URL;
}
//向LogoutFilter中注入属性值filterProcessesUrl
builder.addPropertyValue("filterProcessesUrl", logoutUrl);

if (StringUtils.hasText(successHandlerRef)) {
    if (StringUtils.hasText(logoutSuccessUrl)) {
        pc.getReaderContext().error("Use " + ATT_LOGOUT_URL + " or " + ATT_LOGOUT_HANDL
                pc.extractSource(element));
    }
    //如果successHandlerRef不为空，就通过构造函数注入到LogoutFilter中
    builder.addConstructorArgReference(successHandlerRef);
} else {
    // Use the logout URL if no handler set
    if (!StringUtils.hasText(logoutSuccessUrl)) {
        //如果logout-success-url没有定义，则采用默认的/
        logoutSuccessUrl = DEF_LOGOUT_SUCCESS_URL;
    }
    //通过构造函数注入logoutSuccessUrl值
    builder.addConstructorArgValue(logoutSuccessUrl);
}

if (!StringUtils.hasText(invalidateSession)) {
```

```
            invalidateSession = DEF_INVALIDATE_SESSION;
        }
        //默认Logout的Handler是SecurityContextLogoutHandler
        ManagedList handlers = new ManagedList();
        SecurityContextLogoutHandler sclh = new SecurityContextLogoutHandler();
        if ("true".equals(invalidateSession)) {
            sclh.setInvalidateHttpSession(true);
        } else {
            sclh.setInvalidateHttpSession(false);
        }
        handlers.add(sclh);
        //如果有remember me服务，需要添加remember的handler
        if (rememberMeServices != null) {
            handlers.add(new RuntimeBeanReference(rememberMeServices));
        }
        //继续将handlers通过构造参数注入到LogoutFilter的bean中
        builder.addConstructorArgValue(handlers);

        return builder.getBeanDefinition();
    }
```

此时应该能知道，在LogoutFilter的bean实例化时，两个类变量logoutSuccessUrl、List<LogoutHandler> handlers已经通过构造函数注入到LogoutFilter的实例中来了。
接下来，继续看doFilter部分的源码

```
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        //判断是否需要退出，主要通过请求的url是否是filterProcessesUrl值来识别
        if (requiresLogout(request, response)) {
            //通过SecurityContext实例获取认证信息
            Authentication auth = SecurityContextHolder.getContext().getAuthentication();

            if (logger.isDebugEnabled()) {
                logger.debug("Logging out user '" + auth + "' and transferring to logout destir
```

```
            }
            //循环LogoutHandler处理退出任务
            for (LogoutHandler handler : handlers) {
                handler.logout(request, response, auth);
            }
            //退出成功后，进行redirect操作
            logoutSuccessHandler.onLogoutSuccess(request, response, auth);

            return;
        }

        chain.doFilter(request, response);
    }
```

这时，可能会产生疑问。上一个过滤器SecurityContextPersistenceFilter不是只产生了一个空的SecurityContext么？就是一个没有认证信息的SecurityContext实例

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
```

这个返回的不是null么？产生这个疑问，肯定是被SecurityContextPersistenceFilter过滤器分析时误导的。实际上，每个过滤器只处理自己负责的事情，LogoutFilter只负责拦截j_spring_security_logout这个url（如果没有配置logout的url），其他的url全部跳过。其实退出功能肯定是登录到应用之后才会使用到的，登录对应的Filter肯定会把认证信息添加到SecurityContext中去的，后面再分析。

继续看LogoutHandler是如何处理退出任务的

```
            for (LogoutHandler handler : handlers) {
                handler.logout(request, response, auth);
            }
```

这里的handler至少有一个SecurityContextLogoutHandler，
如果有remember me服务，就还有一个Handler。remember me的handler有两种，
如果配置了持久化信息，如（token-repository-ref、data-source-ref属性）这种的handler为：
org.springframework.security.web.authentication.rememberme.PersistentTokenBasedRememberMeServices
如果没有配置，那么handler就是：

org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices

先来看SecurityContextLogoutHandler

```
//完成两个任务1.让session失效；2.清除SecurityContext实例
public void logout(HttpServletRequest request, HttpServletResponse response, Authentication
    Assert.notNull(request, "HttpServletRequest required");
    if (invalidateHttpSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            session.invalidate();
        }
    }

    SecurityContextHolder.clearContext();
}
```

再来看remember me的handler

1.配置了持久化属性时的handler：PersistentTokenBasedRememberMeServices

```
//也完成两个任务1.清除cookie；2.从持久化中清除remember me数据
public void logout(HttpServletRequest request, HttpServletResponse response, Authentication
    super.logout(request, response, authentication);

    if (authentication != null) {
        //如果定义了token-repository-ref属性，则通过依赖的持久化bean清除
          //如果定义了data-source-ref属性，直接通过
          //JdbcTokenRepositoryImpl清除数据，也就是执行delete操作
        tokenRepository.removeUserTokens(authentication.getName());
    }
}
```

2.未配置持久化属性的handler：TokenBasedRememberMeServices

这个handler没有覆盖父类的logout方法，所以直接调用父类的logout方法，仅仅清除cookie

退出成功后执行onLogoutSuccess操作，完成redirect

```
logoutSuccessHandler.onLogoutSuccess(request, response, auth);
```

这个语句是直接redirect到logout标签中的logout-success-url属性定义的url

至此，整个logoutFilter任务已经完成了，总结一下，主要任务为
1.从SecurityContext中获取Authentication，然后调用每个handler处理logout
2.退出成功后跳转到指定的url

1.7 Spring Security3源码分析-UsernamePasswordAuthenticationFilter分析

发表时间: 2012-05-06 关键字: spring, security

UsernamePasswordAuthenticationFilter过滤器对应的类路径为
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter
实际上这个Filter类的doFilter是父类AbstractAuthenticationProcessingFilter的

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {

    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;
    //判断form-login标签是否包含login-processing-url属性
      //如果没有采用默认的url：j_spring_security_check
    //如果拦截的url不需要认证，直接跳过
    if (!requiresAuthentication(request, response)) {
        chain.doFilter(request, response);

        return;
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Request is to process authentication");
    }

    Authentication authResult;

    try {
        //由子类完成认证
        authResult = attemptAuthentication(request, response);
        if (authResult == null) {
            // return immediately as subclass has indicated that it hasn't completed auther
            return;
        }
        //session策略处理认证信息
          //sessionStrategy是通过session-management标签中定义的
          //session管理策略构造的SessionAuthenticationStrategy
```

```
            //具体的session管理比较复杂，部分后面单个篇幅讲解
            sessionStrategy.onAuthentication(authResult, request, response);
        }
        catch (AuthenticationException failed) {
            // Authentication failed
            //认证失败处理
            unsuccessfulAuthentication(request, response, failed);

            return;
        }

        // Authentication success
        if (continueChainBeforeSuccessfulAuthentication) {
            chain.doFilter(request, response);
        }
        //认证成功处理
         //1.向SecurityContext中设置Authentication认证信息
         //2.如果有remember me服务，则查找请求参数中是否包含_spring_security_remember_me，如果该参数
         //3.发布认证成功事件
         //4.执行跳转
        successfulAuthentication(request, response, authResult);
    }
```

子类UsernamePasswordAuthenticationFilter的认证方法attemptAuthentication

```
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse
        //只处理post提交的请求
        if (postOnly && !request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException("Authentication method not supported: " +
        }
        //获取用户名、密码数据
        String username = obtainUsername(request);
        String password = obtainPassword(request);

        if (username == null) {
            username = "";
```

```
        }

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

        username = username.trim();
        //构造未认证的UsernamePasswordAuthenticationToken
        UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationTok

        // Place the last username attempted into HttpSession for views
        HttpSession session = request.getSession(false);
        //如果session不为空，添加username到session中
        if (session != null || getAllowSessionCreation()) {
            request.getSession().setAttribute(SPRING_SECURITY_LAST_USERNAME_KEY, TextEscapeUtil
        }

        // Allow subclasses to set the "details" property
        //设置details，这里就是设置org.springframework.security.web.
        //authentication.WebAuthenticationDetails实例到details中
        setDetails(request, authRequest);
        //通过AuthenticationManager:ProviderManager完成认证任务
        return this.getAuthenticationManager().authenticate(authRequest);
    }
```

这里的authenticationManager变量也是通过解析form-login标签，构造bean时注入的，具体解析类为：
org.springframework.security.config.http.AuthenticationConfigBuilder
代码片段为：

```
    void createFormLoginFilter(BeanReference sessionStrategy, BeanReference authManager) {

        Element formLoginElt = DomUtils.getChildElementByTagName(httpElt, Elements.FORM_LOGIN);

        if (formLoginElt != null || autoConfig) {
            FormLoginBeanDefinitionParser parser = new FormLoginBeanDefinitionParser("/j_spring
                    AUTHENTICATION_PROCESSING_FILTER_CLASS, requestCache, sessionStrategy);
```

```
            parser.parse(formLoginElt, pc);
            formFilter = parser.getFilterBean();
            formEntryPoint = parser.getEntryPointBean();
        }

        if (formFilter != null) {
            formFilter.getPropertyValues().addPropertyValue("allowSessionCreation", new Boolear
            //设置authenticationManager的bean依赖
            formFilter.getPropertyValues().addPropertyValue("authenticationManager", authManage



            // Id is required by login page filter
            formFilterId = pc.getReaderContext().generateBeanName(formFilter);
            pc.registerBeanComponent(new BeanComponentDefinition(formFilter, formFilterId));
            injectRememberMeServicesRef(formFilter, rememberMeServicesId);
        }
    }
```

继续看ProviderManager代码。实际上authenticate方法由ProviderManager的父类定义，并且authenticate方法内调用子类的doAuthentication方法，记得这是设计模式中的模板模式

```
    public Authentication doAuthentication(Authentication authentication) throws Authenticatior
        Class<? extends Authentication> toTest = authentication.getClass();
        AuthenticationException lastException = null;
        Authentication result = null;
        //循环ProviderManager中的providers，由具体的provider执行认证操作
        for (AuthenticationProvider provider : getProviders()) {
                System.out.println("AuthenticationProvider: " + provider.getClass().getName());
            if (!provider.supports(toTest)) {
                continue;
            }

            logger.debug("Authentication attempt using " + provider.getClass().getName());
```

```
            try {
                result = provider.authenticate(authentication);

                if (result != null) {
                    //复制details
                    copyDetails(authentication, result);
                    break;
                }
            } catch (AccountStatusException e) {
                // SEC-546: Avoid polling additional providers if auth failure is due to invali
                eventPublisher.publishAuthenticationFailure(e, authentication);
                throw e;
            } catch (AuthenticationException e) {
                lastException = e;
            }
        }

        if (result == null && parent != null) {
            // Allow the parent to try.
            try {
                result = parent.authenticate(authentication);
            } catch (ProviderNotFoundException e) {
                // ignore as we will throw below if no other exception occurred prior to callir
                // may throw ProviderNotFound even though a provider in the child already handl
            } catch (AuthenticationException e) {
                lastException = e;
            }
        }

        if (result != null) {
            eventPublisher.publishAuthenticationSuccess(result);
            return result;
        }

        // Parent was null, or didn't authenticate (or throw an exception).

        if (lastException == null) {
```

```
            lastException = new ProviderNotFoundException(messages.getMessage("ProviderManager.
                        new Object[] {toTest.getName()}, "No AuthenticationProvider found for {
        }
        //由注入进来的org.springframework.security.authentication.DefaultAuthenticationEventPubl
        eventPublisher.publishAuthenticationFailure(lastException, authentication);

        throw lastException;
    }
```

ProviderManager类中的providers由哪些provider呢？如果看完authentication-manager标签解析的讲解，应该知道注入到providers中的provider分别为：
org.springframework.security.authentication.dao.DaoAuthenticationProvider
org.springframework.security.authentication.AnonymousAuthenticationProvider
其他的provider根据特殊情况，再添加到providers中的，如remember me功能的provider
org.springframework.security.authentication.RememberMeAuthenticationProvider

可以看出来，ProviderManager仅仅是管理provider的，具体的authenticate认证任务由各自provider来完成。

现在来看DaoAuthenticationProvider的认证处理，实际上authenticate由父类AbstractUserDetailsAuthenticationProvider完成。代码如下

```
    public Authentication authenticate(Authentication authentication) throws AuthenticationExce
        .....
         //获取登录的用户名
        String username = (authentication.getPrincipal() == null) ? "NONE_PROVIDED" : authentic

        boolean cacheWasUsed = true;
        //如果配置了缓存，从缓存中获取UserDetails实例
        UserDetails user = this.userCache.getUserFromCache(username);

        if (user == null) {
            cacheWasUsed = false;

            try {
```

```
            //如果UserDetails为空，则由具体子类DaoAuthenticationProvider
            //根据用户名、authentication获取UserDetails
            user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authenticat
        } catch (UsernameNotFoundException notFound) {
            if (hideUserNotFoundExceptions) {
                throw new BadCredentialsException(messages.getMessage(
                        "AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad cr
            } else {
                throw notFound;
            }
        }

        Assert.notNull(user, "retrieveUser returned null - a violation of the interface cor
    }

    try {
        //一些认证检查（账号是否可用、是否过期、是否被锁定）
        preAuthenticationChecks.check(user);
        //额外的密码检查（salt、passwordEncoder）
        additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authenti
    } catch (AuthenticationException exception) {
        if (cacheWasUsed) {
            // There was a problem, so try again after checking
            // we're using latest data (i.e. not from the cache)
            cacheWasUsed = false;
            user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authenticat
            preAuthenticationChecks.check(user);
            additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) auth
        } else {
            throw exception;
        }
    }
    //检查账号是否过期
    postAuthenticationChecks.check(user);
    //添加UserDetails到缓存中
    if (!cacheWasUsed) {
        this.userCache.putUserInCache(user);
```

```
        }

        Object principalToReturn = user;

        if (forcePrincipalAsString) {
            principalToReturn = user.getUsername();
        }
        //返回成功认证后的Authentication
        return createSuccessAuthentication(principalToReturn, authentication, user);
    }
```

继续看DaoAuthenticationProvider的retrieveUser方法

```
    protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationTok
            throws AuthenticationException {
        UserDetails loadedUser;

        try {
            //最关键的部分登场了
              //UserDetailService就是authentication-provider标签中定义的
              //属性user-service-ref
            loadedUser = this.getUserDetailsService().loadUserByUsername(username);
        }
        catch (DataAccessException repositoryProblem) {
            throw new AuthenticationServiceException(repositoryProblem.getMessage(), repository
        }

        if (loadedUser == null) {
            throw new AuthenticationServiceException(
                    "UserDetailsService returned null, which is an interface contract violatior
        }
        return loadedUser;
    }
```

实际上，只要实现UserDetailsService接口的loadUserByUsername方法，就完成了登录认证的工作

```
<authentication-manager alias="authenticationManager">
    <authentication-provider user-service-ref="userDetailsManager"/>
</authentication-manager>
```

很多教程上说配置JdbcUserDetailsManager这个UserDetailsService，实际上该类的父类JdbcDaoImpl方法loadUserByUsername代码如下：

```
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException, Da
        //根据username从数据库中查询User数据
        List<UserDetails> users = loadUsersByUsername(username);

        if (users.size() == 0) {
            throw new UsernameNotFoundException(
                    messages.getMessage("JdbcDaoImpl.notFound", new Object[]{username}, "Userna
        }

        UserDetails user = users.get(0); // contains no GrantedAuthority[]

        Set<GrantedAuthority> dbAuthsSet = new HashSet<GrantedAuthority>();
        //添加授权信息
        if (enableAuthorities) {
            dbAuthsSet.addAll(loadUserAuthorities(user.getUsername()));
        }
        //是否使用了Group
        if (enableGroups) {
            dbAuthsSet.addAll(loadGroupAuthorities(user.getUsername()));
        }

        List<GrantedAuthority> dbAuths = new ArrayList<GrantedAuthority>(dbAuthsSet);

        addCustomAuthorities(user.getUsername(), dbAuths);

        if (dbAuths.size() == 0) {
            throw new UsernameNotFoundException(
```

```
                    messages.getMessage("JdbcDaoImpl.noAuthority",
                            new Object[] {username}, "User {0} has no GrantedAuthority"), userr
        }

        return createUserDetails(username, user, dbAuths);
    }

    //usersByUsernameQuery查询语句可配置
     //直接从数据库中查询该username对应的数据，并构造User对象
    protected List<UserDetails> loadUsersByUsername(String username) {
        return getJdbcTemplate().query(usersByUsernameQuery, new String[] {username}, new RowMa
            public UserDetails mapRow(ResultSet rs, int rowNum) throws SQLException {
                String username = rs.getString(1);
                String password = rs.getString(2);
                boolean enabled = rs.getBoolean(3);
                return new User(username, password, enabled, true, true, true, AuthorityUtils.N
            }

        });
    }

.....
    protected UserDetails createUserDetails(String username, UserDetails userFromUserQuery,
            List<GrantedAuthority> combinedAuthorities) {
        String returnUsername = userFromUserQuery.getUsername();

        if (!usernameBasedPrimaryKey) {
            returnUsername = username;
        }
        //根据用户名、密码、enabled、授权列表构造UserDetails实例User
        return new User(returnUsername, userFromUserQuery.getPassword(), userFromUserQuery.isEr
                true, true, true, combinedAuthorities);
    }
```

其他的provider，如RememberMeAuthenticationProvider、AnonymousAuthenticationProvider的认证处理都很简单，首先判

断是否支持Authentication，不支持直接返回null，支持也不处理直接返回该Authentication

这里需要强调一下，DaoAuthenticationProvider只支持UsernamePasswordAuthenticationToken这个Authentication。如果对其他的Authentication，DaoAuthenticationProvider是不做处理的

1.8 Spring Security3源码分析-RememberMeAuthenticationFilter分析

发表时间: 2012-05-06

RememberMeAuthenticationFilter过滤器对应的类路径为
org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFilter
看主要的doFilter方法

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;
    //判断SecurityContext实例中是否存在Authentication认证信息
     //没有认证的情况，才会做autoLogin的操作
    if (SecurityContextHolder.getContext().getAuthentication() == null) {
        //具体autoLogin过程后面分析，主要返回一个认证实体Authentication
        Authentication rememberMeAuth = rememberMeServices.autoLogin(request, response);

        if (rememberMeAuth != null) {
            // Attempt authenticaton via AuthenticationManager
            try {
                //通过authenticationManager对该认证实体验证
                   //之前登录的Filter已经说明了，authenticationManager将
                   //将具体的认证工作交给provider完成
                   //而provider只负责自己关心的认证实体
                   //所以返回的认证实体RememberMeAuthenticationToken由
                   //RememberMeAuthenticationProvider完成认证的工作
                   //而这个provider也仅仅判断key是否相同，然后直接返回Authentication
                rememberMeAuth = authenticationManager.authenticate(rememberMeAuth);

                //设置认证信息到SecurityContext中
                SecurityContextHolder.getContext().setAuthentication(rememberMeAuth);
                //登录成功后的处理
                onSuccessfulAuthentication(request, response, rememberMeAuth);
                // Fire event
                if (this.eventPublisher != null) {
                    eventPublisher.publishEvent(new InteractiveAuthenticationSuccessEvent(
                            SecurityContextHolder.getContext().getAuthentication(), this.ge
```

```
                }
            } catch (AuthenticationException authenticationException) {
            ……
            }
        }

        chain.doFilter(request, response);
    } else {
        ……
        chain.doFilter(request, response);
    }
}
```

下面看具体的autoLogin执行过程。
autoLogin是由RememberMeServices接口定义的方法，那么在RememberMeAuthenticationFilter类中的RememberMeServices实例也是通过解析http标签时注入到Filter中的。具体解析过程代码就不分析了。
如果remember-me标签有data-source-ref或token-repository-ref属性，RememberMeServices接口实现类为
PersistentTokenBasedRememberMeServices
其他情况下，没有设置services-ref属性，RememberMeServices接口实现类为
TokenBasedRememberMeServices
这两个实现类都继承了AbstractRememberMeServices抽象类

而autoLogin是AbstractRememberMeServices类完成的，看源码

```
public final Authentication autoLogin(HttpServletRequest request, HttpServletResponse respo
    //从cookie中获取remember me的相关信息
     //cookie名称是SPRING_SECURITY_REMEMBER_ME_COOKIE
    //cookie的value实际上是经过Base64做了两次编码算法处理的结果
     //对应的是数据库中persistent_logins表中的series、token的数据
     //数据库中保存的值是经过一次Base64编码的处理
    String rememberMeCookie = extractRememberMeCookie(request);

    if (rememberMeCookie == null) {
        return null;
    }
```

```
        logger.debug("Remember-me cookie detected");

        UserDetails user = null;

        try {
            //所以这里只要解码一次就可以了
              //这个数组中包含两个字符串，分别对应persistent_logins
            //表中的series、token的数据
            String[] cookieTokens = decodeCookie(rememberMeCookie);
            //执行autoLogin的处理
            user = processAutoLoginCookie(cookieTokens, request, response);
            //账号状态检查，主要检查是否被锁，是否可用等
            userDetailsChecker.check(user);

            logger.debug("Remember-me cookie accepted");
            //返回UserDetails的实现类RememberMeAuthenticationToken
            //注意这里构造的不是之前登录的那个UsernamePasswordAuthenticationToken了
            return createSuccessfulAuthentication(request, user);
        } catch (CookieTheftException cte) {
            cancelCookie(request, response);
            throw cte;
        } catch (UsernameNotFoundException noUser) {
            logger.debug("Remember-me login was valid but corresponding user not found.", noUse
        } catch (InvalidCookieException invalidCookie) {
            logger.debug("Invalid remember-me cookie: " + invalidCookie.getMessage());
        } catch (AccountStatusException statusInvalid) {
            logger.debug("Invalid UserDetails: " + statusInvalid.getMessage());
        } catch (RememberMeAuthenticationException e) {
            logger.debug(e.getMessage());
        }

        cancelCookie(request, response);
        return null;
    }
```

继续跟踪processAutoLoginCookie方法，该方法由具体子类实现，下面的代码从PersistentTokenBasedRememberMeServices中截取

```
protected UserDetails processAutoLoginCookie(String[] cookieTokens, HttpServletRequest requ

    if (cookieTokens.length != 2) {
        throw new InvalidCookieException("Cookie token did not contain " + 2 +
                " tokens, but contained '" + Arrays.asList(cookieTokens) + "'");
    }
    final String presentedSeries = cookieTokens[0];
    final String presentedToken = cookieTokens[1];
    //根据cookie中获取的Series数据从持久化载体（一般为数据库）中
     //获取该Series对应的登录信息，构造PersistentRememberMeToken实例
    PersistentRememberMeToken token = tokenRepository.getTokenForSeries(presentedSeries);
    //此处省略一部分验证……
     //重新产生一个token并由当前时间构造新的PersistentRememberMeToken实例
    PersistentRememberMeToken newToken = new PersistentRememberMeToken(token.getUsername(),
            token.getSeries(), generateTokenData(), new Date());

    try {
        //根据Series更新persistent_logins
        tokenRepository.updateToken(newToken.getSeries(), newToken.getTokenValue(), newToke
        //重新设置cookie
        addCookie(newToken, request, response);
    } catch (DataAccessException e) {
        logger.error("Failed to update token: ", e);
        throw new RememberMeAuthenticationException("Autologin failed due to data access pr
    }
    //直接根据username通过UserDetailsService实例的loadUserByUsername方法
     //获取UserDetails对象并返回
    UserDetails user = getUserDetailsService().loadUserByUsername(token.getUsername());

    return user;
}
```

这里需要注意的是getUserDetailsService方法返回的userDetailsService变量从哪里注入进来的呢？通过解析remember标签构造bean时注入的么？不是的，进一步观察remember标签解析代码

```
public BeanDefinition parse(Element element, ParserContext pc) {
    //此处省略了获取标签属性、判断等代码
    boolean isPersistent = dataSourceSet | tokenRepoSet;
    //如果配置了data-source-ref或token-repository-ref属性
    if (isPersistent) {
        Object tokenRepo;
        services = new RootBeanDefinition(PersistentTokenBasedRememberMeServices.class);

        if (tokenRepoSet) {
            tokenRepo = new RuntimeBeanReference(tokenRepository);
        } else {
            //设置data-source-ref属性的情况
            tokenRepo = new RootBeanDefinition(JdbcTokenRepositoryImpl.class);
            ((BeanDefinition)tokenRepo).getPropertyValues().addPropertyValue("dataSource",
                    new RuntimeBeanReference(dataSource));
        }
        //tokenRepository主要处理存储的功能
        services.getPropertyValues().addPropertyValue("tokenRepository", tokenRepo);
    } else if (!servicesRefSet) {
        services = new RootBeanDefinition(TokenBasedRememberMeServices.class);
    }

    if (services != null) {
        //注意：这里仅仅判断是否配置了user-service-ref属性
          //如果配置了该属性，就会注入userDetailsService
        //实际上很少配置该标签
        if (userServiceSet) {
            services.getPropertyValues().addPropertyValue("userDetailsService", new Runtime
        }

        if ("true".equals(element.getAttribute(ATT_SECURE_COOKIE))) {
            services.getPropertyValues().addPropertyValue("useSecureCookie", true);
        }
```

```
            if (tokenValiditySet) {
                Integer tokenValidity = new Integer(tokenValiditySeconds);
                if (tokenValidity.intValue() < 0 && isPersistent) {
                    pc.getReaderContext().error(ATT_TOKEN_VALIDITY + " cannot be negative if us
                            " a persistent remember-me token repository", source);
                }
                services.getPropertyValues().addPropertyValue("tokenValiditySeconds", tokenVali
            }
            services.setSource(source);
            services.getPropertyValues().addPropertyValue("key", key);
            servicesName = pc.getReaderContext().generateBeanName(services);
            pc.registerBeanComponent(new BeanComponentDefinition(services, servicesName));
        } else {
            servicesName = rememberMeServicesRef;
        }

        if (StringUtils.hasText(element.getAttribute(ATT_SERVICES_ALIAS))) {
            pc.getRegistry().registerAlias(servicesName, element.getAttribute(ATT_SERVICES_ALIA
        }

        BeanDefinition filter = createFilter(pc, source);
        pc.popAndRegisterContainingComponent();

        return filter;
    }
```

上面的解析过程已经很明确了，确实没有在这里注入userDetailsService。如果细心的话，会发现在解析http标签时，有这么一段代码

```
authBldr.createUserServiceInjector();
```

没错，就是这个方法完成动态注入的了

```
    void createUserServiceInjector() {
        BeanDefinitionBuilder userServiceInjector = BeanDefinitionBuilder.rootBeanDefinition(Us
        userServiceInjector.addConstructorArgValue(x509ProviderId);
        userServiceInjector.addConstructorArgValue(rememberMeServicesId);
        userServiceInjector.addConstructorArgValue(openIDProviderId);
        userServiceInjector.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
        pc.getReaderContext().registerWithGeneratedName(userServiceInjector.getBeanDefinition()
    }
```

UserDetailsServiceInjectionBeanPostProcessor实现了两个接口BeanPostProcessor, BeanFactoryAware，既可以动态改变bean，又能直接获取到BeanFactory对象再从ioc容器中获取具体的bean。
主要看postProcessBeforeInitialization方法

```
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansExc
        if(beanName == null) {
            return bean;
        }
        if (beanName.equals(x509ProviderId)) {
            injectUserDetailsServiceIntoX509Provider((PreAuthenticatedAuthenticationProvider) b
        } else if (beanName.equals(rememberMeServicesId)) {
            injectUserDetailsServiceIntoRememberMeServices(bean);
        } else if (beanName.equals(openIDProviderId)) {
            injectUserDetailsServiceIntoOpenIDProvider(bean);
        }

        return bean;
    }
```

只对x509ProviderId、rememberMeServicesId、openIDProviderId的bean动态注入userDetailsService实例。

1.9 Spring Security3源码分析-SecurityContextHolderAwareRequestFilter分析

发表时间: 2012-05-07 关键字: securitycontextholderawarerequestfilter, spring, security

SecurityContextHolderAwareRequestFilter过滤器对应的类路径为
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter

从类名称可以猜出这个过滤器主要是包装请求对象request的，看源码

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
    chain.doFilter(new SecurityContextHolderAwareRequestWrapper((HttpServletRequest) req, r
}
```

SecurityContextHolderAwareRequestWrapper类对request包装的目的主要是实现servlet api的一些接口方法isUserInRole、getRemoteUser

```
//从SecurityContext中获取认证实体Authentication
private Authentication getAuthentication() {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();

    if (!authenticationTrustResolver.isAnonymous(auth)) {
        return auth;
    }

    return null;
}

//实现getRemoteUser方法。首先获取认证实体，再从认证实体中获取登录账号
@Override
public String getRemoteUser() {
    Authentication auth = getAuthentication();

    if ((auth == null) || (auth.getPrincipal() == null)) {
        return null;
```

```
        }

        if (auth.getPrincipal() instanceof UserDetails) {
            return ((UserDetails) auth.getPrincipal()).getUsername();
        }

        return auth.getPrincipal().toString();
    }

    //实现getUserPrincipal方法
    @Override
    public Principal getUserPrincipal() {
        Authentication auth = getAuthentication();

        if ((auth == null) || (auth.getPrincipal() == null)) {
            return null;
        }

        return auth;
    }

    //判断是否授权。这里注意一下rolePrefix，就是角色的前缀
    private boolean isGranted(String role) {
        Authentication auth = getAuthentication();

        if( rolePrefix != null ) {
            role = rolePrefix + role;
        }

        if ((auth == null) || (auth.getPrincipal() == null)) {
            return false;
        }

        Collection<GrantedAuthority> authorities = auth.getAuthorities();

        if (authorities == null) {
            return false;
```

```
        }


        for (GrantedAuthority grantedAuthority : authorities) {
            if (role.equals(grantedAuthority.getAuthority())) {
                return true;
            }
        }

        return false;
    }

    //实现isUserInRole
    @Override
    public boolean isUserInRole(String role) {
        return isGranted(role);
    }
```

这个过滤器看起来很简单。目的仅仅是实现java ee中servlet api一些接口方法。
一些应用中直接使用getRemoteUser方法、isUserInRole方法，在使用spring security时其实就是通过这个过滤器来实现的。

1.10 Spring Security3源码分析-FilterSecurityInterceptor分析

发表时间: 2012-05-07

FilterSecurityInterceptor过滤器对应的类路径为
org.springframework.security.web.access.intercept.FilterSecurityInterceptor
这个filter是filterchain中比较复杂，也是比较核心的过滤器，主要负责授权的工作
在看这个filter源码之前，先来看看spring是如何构造filter这个bean的
具体的构造过程的代码片段为

```
//这个方法源自HttpConfigurationBuilder类
void createFilterSecurityInterceptor(BeanReference authManager) {
    //判断是否配置了use-expressions属性
    boolean useExpressions = FilterInvocationSecurityMetadataSourceParser.isUseExpressions(
    //根据intercept-url标签列表创建授权需要的元数据信息。后面仔细分析
    BeanDefinition securityMds = FilterInvocationSecurityMetadataSourceParser.createSecurit

    RootBeanDefinition accessDecisionMgr;
    //创建voter列表
    ManagedList<BeanDefinition> voters =  new ManagedList<BeanDefinition>(2);
    //如果是使用了表达式，使用WebExpressionVoter
    //没使用表达式，就使用RoleVoter、AuthenticatedVoter
    if (useExpressions) {
        voters.add(new RootBeanDefinition(WebExpressionVoter.class));
    } else {
        voters.add(new RootBeanDefinition(RoleVoter.class));
        voters.add(new RootBeanDefinition(AuthenticatedVoter.class));
    }
    //定义授权的决策管理类AffirmativeBased
    accessDecisionMgr = new RootBeanDefinition(AffirmativeBased.class);
    //添加依赖的voter列表
    accessDecisionMgr.getPropertyValues().addPropertyValue("decisionVoters", voters);
    accessDecisionMgr.setSource(pc.extractSource(httpElt));

    // Set up the access manager reference for http
    String accessManagerId = httpElt.getAttribute(ATT_ACCESS_MGR);
    //如果未定义access-decision-manager-ref属性，就使用默认的
     //AffirmativeBased
```

```
        if (!StringUtils.hasText(accessManagerId)) {
            accessManagerId = pc.getReaderContext().generateBeanName(accessDecisionMgr);
            pc.registerBeanComponent(new BeanComponentDefinition(accessDecisionMgr, accessManag
        }
        //创建FilterSecurityInterceptor过滤器
        BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(FilterSecurity
        //添加决策管理器
        builder.addPropertyReference("accessDecisionManager", accessManagerId);
        //添加认证管理类
        builder.addPropertyValue("authenticationManager", authManager);

        if ("false".equals(httpElt.getAttribute(ATT_ONCE_PER_REQUEST))) {
            builder.addPropertyValue("observeOncePerRequest", Boolean.FALSE);
        }
        //添加授权需要的安全元数据资源
        builder.addPropertyValue("securityMetadataSource", securityMds);
        BeanDefinition fsiBean = builder.getBeanDefinition();
        //向ioc容器注册bean
        String fsiId = pc.getReaderContext().generateBeanName(fsiBean);
        pc.registerBeanComponent(new BeanComponentDefinition(fsiBean,fsiId));

        // Create and register a DefaultWebInvocationPrivilegeEvaluator for use with taglibs et
        BeanDefinition wipe = new RootBeanDefinition(DefaultWebInvocationPrivilegeEvaluator.cla
        wipe.getConstructorArgumentValues().addGenericArgumentValue(new RuntimeBeanReference(fs

        pc.registerBeanComponent(new BeanComponentDefinition(wipe, pc.getReaderContext().genera

        this.fsi = new RuntimeBeanReference(fsiId);
    }
```

现在再仔细分析创建元数据资源的bean过程

```
    static BeanDefinition createSecurityMetadataSource(List<Element> interceptUrls, Element elt
        //创建Url处理类，有两个实现：AntUrlPathMatcher、RegexUrlPathMatcher
        UrlMatcher matcher = HttpSecurityBeanDefinitionParser.createUrlMatcher(elt);
        boolean useExpressions = isUseExpressions(elt);
```

```
        //解析intercept-url标签，构造所有需要拦截url的map信息
         //map中的key：RequestKey的bean定义，value：SecurityConfig的bean定义
        ManagedMap<BeanDefinition, BeanDefinition> requestToAttributesMap = parseInterceptUrlsF
                interceptUrls, useExpressions, pc);
        BeanDefinitionBuilder fidsBuilder;

        if (useExpressions) {
            //定义表达式处理类的bean
            Element expressionHandlerElt = DomUtils.getChildElementByTagName(elt, Elements.EXPF
            String expressionHandlerRef = expressionHandlerElt == null ? null : expressionHandl

            if (StringUtils.hasText(expressionHandlerRef)) {
                logger.info("Using bean '" + expressionHandlerRef + "' as web SecurityExpressio
            } else {
                BeanDefinition expressionHandler = BeanDefinitionBuilder.rootBeanDefinition(Def
                expressionHandlerRef = pc.getReaderContext().generateBeanName(expressionHandler
                pc.registerBeanComponent(new BeanComponentDefinition(expressionHandler, express
            }
            //定义表达式类型的FilterInvocationSecurityMetadataSource
            fidsBuilder = BeanDefinitionBuilder.rootBeanDefinition(ExpressionBasedFilterInvocat
            //通过构造函数注入依赖
            fidsBuilder.addConstructorArgValue(matcher);
            fidsBuilder.addConstructorArgValue(requestToAttributesMap);
            fidsBuilder.addConstructorArgReference(expressionHandlerRef);
        } else {
            //定义非表达式类型的FilterInvocationSecurityMetadataSource
            fidsBuilder = BeanDefinitionBuilder.rootBeanDefinition(DefaultFilterInvocationSecur
            //通过构造函数注入依赖
            fidsBuilder.addConstructorArgValue(matcher);
            fidsBuilder.addConstructorArgValue(requestToAttributesMap);
        }

        fidsBuilder.addPropertyValue("stripQueryStringFromUrls", matcher instanceof AntUrlPathM
        fidsBuilder.getRawBeanDefinition().setSource(pc.extractSource(elt));

        return fidsBuilder.getBeanDefinition();
    }
```

通过以上的bean构造过程，FilterSecurityInterceptor所依赖的决策管理器、认证管理器、安全元数据资源都具备了，该让FilterSecurityInterceptor干活了，其源码为

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
    //封装request, response, chain，方便参数传递、增加代码阅读性
    FilterInvocation fi = new FilterInvocation(request, response, chain);
    invoke(fi);
}

public void invoke(FilterInvocation fi) throws IOException, ServletException {
    if ((fi.getRequest() != null) && (fi.getRequest().getAttribute(FILTER_APPLIED) != null)
            && observeOncePerRequest) {
        if (fi.getRequest() != null) {
            fi.getRequest().setAttribute(FILTER_APPLIED, Boolean.TRUE);
        }
        //执行父类beforeInvocation，类似于aop中的before
        InterceptorStatusToken token = super.beforeInvocation(fi);

        try {
            //filter传递
            fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
        } finally {
            //执行父类的afterInvocation，类似于aop中的after
            super.afterInvocation(token, null);
        }
    }
}
```

继续看父类的beforeInvocation方法，其中省略了一些不重要的代码片段

```
protected InterceptorStatusToken beforeInvocation(Object object) {
    //根据SecurityMetadataSource获取配置的权限属性
    Collection<ConfigAttribute> attributes = this.obtainSecurityMetadataSource().getAttribu
    //省略……
```

```
        //判断是否需要对认证实体重新认证，默认为否
        Authentication authenticated = authenticateIfRequired();

        // Attempt authorization
        try {
            //决策管理器开始决定是否授权，如果授权失败，直接抛出AccessDeniedException
            this.accessDecisionManager.decide(authenticated, object, attributes);
        }
        catch (AccessDeniedException accessDeniedException) {
            publishEvent(new AuthorizationFailureEvent(object, attributes, authenticated,
                    accessDeniedException));

            throw accessDeniedException;
        }
    }
```

增加说明

```
Collection<ConfigAttribute> attributes = this.obtainSecurityMetadataSource().getAttributes(obje
```

这里获取的是权限列表信息，比如说有这个配置
<security:intercept-url pattern="/index.jsp*" access="ROLE_USER,ROLE_ADMIN"/>
如果现在发起一个请求时index.jsp，那么根据这个请求返回的attributes集合就是分别包含ROLE_USER,ROLE_ADMIN属性的两个SecurityConfig对象

至于请求url如何匹配的，大家可以通过阅读DefaultFilterInvocationSecurityMetadataSource类的源码，实际上，这里用到了spring的路径匹配工具类org.springframework.util.AntPathMatcher
AntPathMatcher匹配方式的通配符有三种：
?（匹配任何单字符），*（匹配0或者任意数量的字符），**（匹配0或者更多的目录）

由于之前在bean的定义过程已经知道决策管理器是AffirmativeBased，接着看AffirmativeBased的决策过程

```
    public void decide(Authentication authentication, Object object, Collection<ConfigAttribute
            throws AccessDeniedException {
        int deny = 0;
```

```
        //循环voters，实际上是RoleVoter、AuthenticatedVoter
        for (AccessDecisionVoter voter : getDecisionVoters()) {
            //把具体的决策任务交给voter处理
              //voter只返回-1、0、1，只有为1才算授权成功
            int result = voter.vote(authentication, object, configAttributes);

            if (logger.isDebugEnabled()) {
                logger.debug("Voter: " + voter + ", returned: " + result);
            }

            switch (result) {
            case AccessDecisionVoter.ACCESS_GRANTED:
                return;

            case AccessDecisionVoter.ACCESS_DENIED:
                deny++;

                break;

            default:
                break;
            }
        }
        //只要有一个voter拒绝了，则直接抛出访问拒绝异常
        if (deny > 0) {
            throw new AccessDeniedException(messages.getMessage("AbstractAccessDecisionManager.
                    "Access is denied"));
        }

        // To get this far, every AccessDecisionVoter abstained
        checkAllowIfAllAbstainDecisions();
    }
```

实际上，有三种决策管理器，分别为AffirmativeBased、ConsensusBased、UnanimousBased，各自决策的区别是：

AffirmativeBased：只要有一个voter投同意票，就授权成功

ConsensusBased：只要投同意票的大于投反对票的，就授权成功

UnanimousBased：需要一致通过才授权成功具体决策规则很简单，只是根据voter返回的结果做处理

接下来，分别看RoleVoter、AuthenticatedVoter的源码

RoleVoter：

```
public int vote(Authentication authentication, Object object, Collection<ConfigAttribute> a
    int result = ACCESS_ABSTAIN;
    //从认证实体中获取所有的权限列表
    Collection<GrantedAuthority> authorities = extractAuthorities(authentication);
    //循环intercept-url配置的access权限列表
    for (ConfigAttribute attribute : attributes) {
        if (this.supports(attribute)) {
            result = ACCESS_DENIED;

            // Attempt to find a matching granted authority
            //循环认证实体所拥有的权限列表
            for (GrantedAuthority authority : authorities) {
                if (attribute.getAttribute().equals(authority.getAuthority())) {
                    //只要有相同的权限，直接返回成功1
                    return ACCESS_GRANTED;
                }
            }
        }
    }

    return result;
}
```

AuthenticatedVoter：

```
public int vote(Authentication authentication, Object object, Collection<ConfigAttribute> a
    int result = ACCESS_ABSTAIN;

    for (ConfigAttribute attribute : attributes) {
        if (this.supports(attribute)) {
            result = ACCESS_DENIED;
```

```
                if (IS_AUTHENTICATED_FULLY.equals(attribute.getAttribute())) {
                    if (isFullyAuthenticated(authentication)) {
                        return ACCESS_GRANTED;
                    }
                }

                if (IS_AUTHENTICATED_REMEMBERED.equals(attribute.getAttribute())) {
                    if (authenticationTrustResolver.isRememberMe(authentication)
                        || isFullyAuthenticated(authentication)) {
                        return ACCESS_GRANTED;
                    }
                }

                if (IS_AUTHENTICATED_ANONYMOUSLY.equals(attribute.getAttribute())) {
                    if (authenticationTrustResolver.isAnonymous(authentication) || isFullyAuthe
                        || authenticationTrustResolver.isRememberMe(authentication)) {
                        return ACCESS_GRANTED;
                    }
                }
            }
        }

        return result;
    }
```

由于RoleVoter在list列表中的位置处于AuthenticatedVoter前面，只要RoleVoter通过，就不会再执行AuthenticatedVoter了。实际上AuthenticatedVoter只会对IS_AUTHENTICATED_FULLY、IS_AUTHENTICATED_REMEMBERED、IS_AUTHENTICATED_ANONYMOUSLY三种权限做vote处理。

1.11 Spring Security3源码分析-BasicAuthenticationFilter分析

发表时间: 2012-05-08 关键字: security, spring

BasicAuthenticationFilter过滤器对应的类路径为
org.springframework.security.web.authentication.www.BasicAuthenticationFilter

Basic验证方式相比较而言用的不是太多。spring security也支持basic的方式，配置如下

```
<security:http auto-config="true">
    <!-- <security:form-login login-page="/login.jsp"/>-->
    <security:http-basic/>
    <security:logout logout-success-url="/login.jsp" invalidate-session="true"/>
    <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_USER,ROLE_ADMIN"/>
    <security:intercept-url pattern="/**" access="ROLE_USER,ROLE_ADMIN"/>
</security:http>
```

如果选择basic方式，需要把form-login标签的定义给注释掉。

接下来看BasicAuthenticationFilter的执行过程

```
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
            throws IOException, ServletException {
        final boolean debug = logger.isDebugEnabled();
        final HttpServletRequest request = (HttpServletRequest) req;
        final HttpServletResponse response = (HttpServletResponse) res;
        //basic登录时，会产生Authorization的header信息
         //Authorization的值是Basic eXVxaW5nc29uZzox
        //eXVxaW5nc29uZzox是经过base编码的一串字符
        String header = request.getHeader("Authorization");
        if ((header != null) && header.startsWith("Basic ")) {
            byte[] base64Token = header.substring(6).getBytes("UTF-8");
            //经过base解码后，token值为username:password这种方式
            String token = new String(Base64.decode(base64Token), getCredentialsCharset(request
            String username = "";
```

```
            String password = "";
            int delim = token.indexOf(":");

            if (delim != -1) {
                username = token.substring(0, delim);
                password = token.substring(delim + 1);
            }

            if (debug) {
                logger.debug("Basic Authentication Authorization header found for user '" + use
            }
            //下面的执行过程基本和login方式一样，认证、授权等过程
            if (authenticationIsRequired(username)) {
                UsernamePasswordAuthenticationToken authRequest =
                        new UsernamePasswordAuthenticationToken(username, password);
                authRequest.setDetails(authenticationDetailsSource.buildDetails(request));

                Authentication authResult;

                try {
                    authResult = authenticationManager.authenticate(authRequest);
                } catch (AuthenticationException failed) {
                    // Authentication failed
                    if (debug) {
                        logger.debug("Authentication request for user: " + username + " failed:
                    }

                    SecurityContextHolder.getContext().setAuthentication(null);

                    rememberMeServices.loginFail(request, response);

                    onUnsuccessfulAuthentication(request, response, failed);

                    if (ignoreFailure) {
                        chain.doFilter(request, response);
                    } else {
                        authenticationEntryPoint.commence(request, response, failed);
```

```
                }

                return;
            }

            // Authentication success
            if (debug) {
                logger.debug("Authentication success: " + authResult.toString());
            }

            SecurityContextHolder.getContext().setAuthentication(authResult);

            rememberMeServices.loginSuccess(request, response, authResult);

            onSuccessfulAuthentication(request, response, authResult);
        }
    }

    chain.doFilter(request, response);
}
```

1.12 Spring Security3源码分析-AnonymousAuthenticationFilter分析

发表时间: 2012-05-08 关键字: anonymousauthenticationfilter, security

AnonymousAuthenticationFilter过滤器对应的类路径为org.springframework.security.web.authentication.AnonymousAuthenticationFilter AnonymousAuthenticationFilter过滤器是在UsernamePasswordAuthenticationFilter、BasicAuthenticationFilter、RememberMeAuthenticationFilter这些过滤器后面的，所以如果这三个过滤器都没有认证成功，则为当前的SecurityContext中添加一个经过匿名认证的token，但是通过servlet的getRemoteUser等方法是获取不到登录账号的。因为SecurityContextHolderAwareRequestFilter过滤器在AnonymousAuthenticationFilter前面。

```
//省略了日志部分
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
    //applyAnonymousForThisRequest永远返回ture
    if (applyAnonymousForThisRequest((HttpServletRequest) req)) {
        //如果当前SecurityContext中没有认证实体
        if (SecurityContextHolder.getContext().getAuthentication() == null) {
            //产生一个匿名认证实体，并保存到SecurityContext中
            SecurityContextHolder.getContext().setAuthentication(createAuthentication((Http
        } else {
        }
    }

    chain.doFilter(req, res);
}

protected Authentication createAuthentication(HttpServletRequest request) {
    //产生匿名认证token，注意这里的key、userAttribute是通过解析标签注入的
    AnonymousAuthenticationToken auth = new AnonymousAuthenticationToken(key, userAttribute
            userAttribute.getAuthorities());
    auth.setDetails(authenticationDetailsSource.buildDetails(request));

    return auth;
}
```

anonymous标签配置为。

```
<anonymous granted-authority="ROLE_ANONYMOUS" enabled="true" username="test"/>
```

这里username属性容易混淆，username默认为anonymousUser，实际上是注入到UserAttribute的password变量中的。
granted-authority属性注入到UserAttribute的authorities授权列表

1.13 Spring Security3源码分析-SessionManagementFilter分析-上

发表时间: 2012-05-08

SessionManagementFilter过滤器对应的类路径为
org.springframework.security.web.session.SessionManagementFilter
这个过滤器看名字就知道是管理session的了，http标签是自动配置时，默认是添加SessionManagementFilter过滤器到filterChainProxy中的，如果不想使用这个过滤器，需要做如下配置

```
<security:http auto-config="true">
  <security:session-management session-fixation-protection="none"/>
</security:http>
```

其实在之前的过滤器中有使用到session策略了，但是没有细说。
SessionManagementFilter提供两大类功能：
1.session固化保护-通过session-fixation-protection配置
2.session并发控制-通过concurrency-control配置
下面看SessionManagementFilter的bean是如何创建的

```
    void createSessionManagementFilters() {
        Element sessionMgmtElt = DomUtils.getChildElementByTagName(httpElt, Elements.SESSION_MA
        Element sessionCtrlElt = null;

        String sessionFixationAttribute = null;
        String invalidSessionUrl = null;
        String sessionAuthStratRef = null;
        String errorUrl = null;
        //如果配置了标签，解析标签的属性、子标签
        if (sessionMgmtElt != null) {
            sessionFixationAttribute = sessionMgmtElt.getAttribute(ATT_SESSION_FIXATION_PROTECT
            invalidSessionUrl = sessionMgmtElt.getAttribute(ATT_INVALID_SESSION_URL);
            sessionAuthStratRef = sessionMgmtElt.getAttribute(ATT_SESSION_AUTH_STRATEGY_REF);
            errorUrl = sessionMgmtElt.getAttribute(ATT_SESSION_AUTH_ERROR_URL);
            sessionCtrlElt = DomUtils.getChildElementByTagName(sessionMgmtElt, Elements.CONCURR
            //判断是否配置了concurrency-control子标签
            if (sessionCtrlElt != null) {
```

```
            //配置了并发控制标签则创建并发控制过滤器和session注册的bean定义
            createConcurrencyControlFilterAndSessionRegistry(sessionCtrlElt);
        }
    }

    if (!StringUtils.hasText(sessionFixationAttribute)) {
        sessionFixationAttribute = OPT_SESSION_FIXATION_MIGRATE_SESSION;
    } else if (StringUtils.hasText(sessionAuthStratRef)) {
        pc.getReaderContext().error(ATT_SESSION_FIXATION_PROTECTION + " attribute cannot be
                " in combination with " + ATT_SESSION_AUTH_STRATEGY_REF, pc.extractSource(s
    }

    boolean sessionFixationProtectionRequired = !sessionFixationAttribute.equals(OPT_SESSIO

    BeanDefinitionBuilder sessionStrategy;
    //如果配置了concurrency-control子标签
    if (sessionCtrlElt != null) {
        assert sessionRegistryRef != null;
        //session控制策略为ConcurrentSessionControlStrategy
        sessionStrategy = BeanDefinitionBuilder.rootBeanDefinition(ConcurrentSessionControl
        sessionStrategy.addConstructorArgValue(sessionRegistryRef);

        String maxSessions = sessionCtrlElt.getAttribute("max-sessions");
        //添加最大session数
        if (StringUtils.hasText(maxSessions)) {
            sessionStrategy.addPropertyValue("maximumSessions", maxSessions);
        }
        String exceptionIfMaximumExceeded = sessionCtrlElt.getAttribute("error-if-maximum-e

        if (StringUtils.hasText(exceptionIfMaximumExceeded)) {
            sessionStrategy.addPropertyValue("exceptionIfMaximumExceeded", exceptionIfMaxim
        }
    } else if (sessionFixationProtectionRequired || StringUtils.hasText(invalidSessionUrl)
            || StringUtils.hasText(sessionAuthStratRef)) {
        //如果没有配置concurrency-control子标签
          //session控制策略是SessionFixationProtectionStrategy
        sessionStrategy = BeanDefinitionBuilder.rootBeanDefinition(SessionFixationProtectio
```

```
        } else {
            //<session-management session-fixation-protection="none"/>
            sfpf = null;
            return;
        }
        //创建SessionManagementFilter，并设置依赖的bean、property
        BeanDefinitionBuilder sessionMgmtFilter = BeanDefinitionBuilder.rootBeanDefinition(Sess
        RootBeanDefinition failureHandler = new RootBeanDefinition(SimpleUrlAuthenticationFailu
        if (StringUtils.hasText(errorUrl)) {
            failureHandler.getPropertyValues().addPropertyValue("defaultFailureUrl", errorUrl);
        }
        sessionMgmtFilter.addPropertyValue("authenticationFailureHandler", failureHandler);
        sessionMgmtFilter.addConstructorArgValue(contextRepoRef);

        if (!StringUtils.hasText(sessionAuthStratRef)) {
            BeanDefinition strategyBean = sessionStrategy.getBeanDefinition();

            if (sessionFixationProtectionRequired) {
                sessionStrategy.addPropertyValue("migrateSessionAttributes",
                        Boolean.valueOf(sessionFixationAttribute.equals(OPT_SESSION_FIXATION_MI
            }
            sessionAuthStratRef = pc.getReaderContext().generateBeanName(strategyBean);
            pc.registerBeanComponent(new BeanComponentDefinition(strategyBean, sessionAuthStrat
        }

        if (StringUtils.hasText(invalidSessionUrl)) {
            sessionMgmtFilter.addPropertyValue("invalidSessionUrl", invalidSessionUrl);
        }

        sessionMgmtFilter.addPropertyReference("sessionAuthenticationStrategy", sessionAuthStra

        sfpf = (RootBeanDefinition) sessionMgmtFilter.getBeanDefinition();
        sessionStrategyRef = new RuntimeBeanReference(sessionAuthStratRef);
    }

    //创建并发控制Filter和session注册的bean
    private void createConcurrencyControlFilterAndSessionRegistry(Element element) {
```

```
final String ATT_EXPIRY_URL = "expired-url";
final String ATT_SESSION_REGISTRY_ALIAS = "session-registry-alias";
final String ATT_SESSION_REGISTRY_REF = "session-registry-ref";

CompositeComponentDefinition compositeDef =
    new CompositeComponentDefinition(element.getTagName(), pc.extractSource(element));
pc.pushContainingComponent(compositeDef);

BeanDefinitionRegistry beanRegistry = pc.getRegistry();

String sessionRegistryId = element.getAttribute(ATT_SESSION_REGISTRY_REF);
//判断是否配置了session-registry-ref属性，用于扩展
 //默认情况下使用SessionRegistryImpl类管理session的注册
if (!StringUtils.hasText(sessionRegistryId)) {
    // Register an internal SessionRegistryImpl if no external reference supplied.
    RootBeanDefinition sessionRegistry = new RootBeanDefinition(SessionRegistryImpl.cla
    sessionRegistryId = pc.getReaderContext().registerWithGeneratedName(sessionRegistry
    pc.registerComponent(new BeanComponentDefinition(sessionRegistry, sessionRegistryId
}

String registryAlias = element.getAttribute(ATT_SESSION_REGISTRY_ALIAS);
if (StringUtils.hasText(registryAlias)) {
    beanRegistry.registerAlias(sessionRegistryId, registryAlias);
}
//创建并发session控制的Filter
BeanDefinitionBuilder filterBuilder =
        BeanDefinitionBuilder.rootBeanDefinition(ConcurrentSessionFilter.class);
//注入session的注册实现类
filterBuilder.addPropertyReference("sessionRegistry", sessionRegistryId);

Object source = pc.extractSource(element);
filterBuilder.getRawBeanDefinition().setSource(source);
filterBuilder.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);

String expiryUrl = element.getAttribute(ATT_EXPIRY_URL);

if (StringUtils.hasText(expiryUrl)) {
```

```
            WebConfigUtils.validateHttpRedirect(expiryUrl, pc, source);
            filterBuilder.addPropertyValue("expiredUrl", expiryUrl);
        }

        pc.popAndRegisterContainingComponent();

        concurrentSessionFilter = filterBuilder.getBeanDefinition();
        sessionRegistryRef = new RuntimeBeanReference(sessionRegistryId);
    }
```

接着看SessionManagementFilter过滤器执行过程

```
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        //省略……
         //判断当前session中是否有SPRING_SECURITY_CONTEXT属性
        if (!securityContextRepository.containsContext(request)) {
            Authentication authentication = SecurityContextHolder.getContext().getAuthenticatic

            if (authentication != null && !authenticationTrustResolver.isAnonymous(authenticati
                try {
                    //再通过sessionStrategy执行session固化、并发处理
                       //与UsernamePasswordAuthenticationFilter时处理一样，后面会仔细分析。
                    sessionStrategy.onAuthentication(authentication, request, response);
                } catch (SessionAuthenticationException e) {
                    SecurityContextHolder.clearContext();
                    failureHandler.onAuthenticationFailure(request, response, e);
                    return;
                }
                //把SecurityContext设置到当前session中
                securityContextRepository.saveContext(SecurityContextHolder.getContext(), reque
            } else {
                if (request.getRequestedSessionId() != null && !request.isRequestedSessionIdVal
```

```
                if (invalidSessionUrl != null) {
                    request.getSession();
                    redirectStrategy.sendRedirect(request, response, invalidSessionUrl);

                    return;
                }
            }
        }

        chain.doFilter(request, response);
    }
```

如果项目需要使用session的并发控制，需要做如下的配置

```
<session-management invalid-session-url="/login.jsp">
    <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" expired-url="/login.
</session-management>
```

session-fixation-protection属性支持三种不同的选项允许你使用
none：使得session固化攻击失效（未配置其他属性）
migrateSession：当用户经过认证后分配一个新的session，它保证原session的所有属性移到新session中
newSession：当用户认证后，建立一个新的session，原（未认证时）session的属性不会进行移到新session中来

如果使用了标签concurrency-control，那么filterchainProxy中会添加新的过滤器ConcurrentSessionFilter。这个过滤器的顺序在SecurityContextPersistenceFilter之前。说明未创建空的认证实体时就需要对session进行并发控制了
看ConcurrentSessionFilter执行过程

```
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
```

```
        HttpServletResponse response = (HttpServletResponse) res;

        HttpSession session = request.getSession(false);
        if (session != null) {
            //这个SessionInformation是在执行SessionManagementFilter时通过sessionRegistry构造的并且
            SessionInformation info = sessionRegistry.getSessionInformation(session.getId());
            //如果当前session已经注册了
            if (info != null) {
                //如果当前session失效了
                if (info.isExpired()) {
                    // Expired - abort processing
                    //强制退出
                    doLogout(request, response);
                    //目标url为expired-url标签配置的属性值
                    String targetUrl = determineExpiredUrl(request, info);
                    //跳转到指定url
                    if (targetUrl != null) {
                        redirectStrategy.sendRedirect(request, response, targetUrl);

                        return;
                    } else {
                        response.getWriter().print("This session has been expired (possibly due
                                "logins being attempted as the same user).");
                        response.flushBuffer();
                    }

                    return;
                } else {
                    // Non-expired - update last request date/time
                    //session未失效，刷新时间
                    info.refreshLastRequest();
                }
            }
        }

        chain.doFilter(request, response);
    }
```

那么分析完ConcurrentSessionFilter过滤器的执行过程，具体有什么作用呢？
简单点概括就是：从session缓存中获取当前session信息，如果发现过期了，就跳转到expired-url配置的url或者响应session失效提示信息。当前session有哪些情况会导致session失效呢？这里的失效并不是指在web容器中session的失效，而是spring security把登录成功的session封装为SessionInformation并放到注册类缓存中，如果SessionInformation的expired变量为true，则表示session已失效。
所以，ConcurrentSessionFilter过滤器主要检查SessionInformation的expired变量的值

为了能清楚解释session 并发控制的过程，现在引入UsernamePasswordAuthenticationFilter过滤器，因为该过滤器就是对登录账号进行认证的，并且在分析UsernamePasswordAuthenticationFilter过滤器时，也没有详细讲解session的处理。
UsernamePasswordAuthenticationFilter的doFilter是由父类AbstractAuthenticationProcessingFilter完成的，截取部分重要代码

```
        try {
            //由子类UsernamePasswordAuthenticationFilter认证
              //之前已经详细分析
            authResult = attemptAuthentication(request, response);
            if (authResult == null) {
                // return immediately as subclass has indicated that it hasn't completed auther
                return;
            }
            //由session策略类完成session固化处理、并发控制处理
              //如果当前认证实体的已注册session数超出最大并发的session数
              //这里会抛出AuthenticationException
            sessionStrategy.onAuthentication(authResult, request, response);
        }
        catch (AuthenticationException failed) {
            // Authentication failed
            //捕获到异常，直接跳转到失败页面或做其他处理
            unsuccessfulAuthentication(request, response, failed);

            return;
        }
```

session处理的方法就是这一语句

```
sessionStrategy.onAuthentication(authResult, request, response);
```

如果是采用了并发控制session，则sessionStrategy为ConcurrentSessionControlStrategy类，具体源码：

```
    public void onAuthentication(Authentication authentication, HttpServletRequest request,
            HttpServletResponse response) {
        //检查是否允许认证
        checkAuthenticationAllowed(authentication, request);

        // Allow the parent to create a new session if necessary
        //执行父类SessionFixationProtectionStrategy的onAuthentication，完成session固化工作。其实就
        super.onAuthentication(authentication, request, response);
        //向session注册类SessionRegistryImpl注册当前session、认证实体
         //实际上SessionRegistryImpl维护两个缓存列表，分别是
         //1.sessionIds（Map）：key=sessionid，value=SessionInformation
        //2.principals（Map）：key=principal，value=HashSet
        sessionRegistry.registerNewSession(request.getSession().getId(), authentication.getPrir
    }
    //检查是否允许认证通过，如果通过直接返回，不通过，抛出AuthenticationException
    private void checkAuthenticationAllowed(Authentication authentication, HttpServletRequest r
            throws AuthenticationException {
        //获取当前认证实体的session集合
        final List<SessionInformation> sessions = sessionRegistry.getAllSessions(authenticatior

        int sessionCount = sessions.size();
        //获取的并发session数（由max-sessions属性配置）
        int allowedSessions = getMaximumSessionsForThisUser(authentication);
        //如果当前认证实体的已注册session数小于max-sessions，允许通过
        if (sessionCount < allowedSessions) {
            // They haven't got too many login sessions running at present
            return;
        }
        //如果allowedSessions配置为-1，说明未限制并发session数，允许通过
        if (allowedSessions == -1) {
            // We permit unlimited logins
            return;
```

```
        }
        //如果当前认证实体的已注册session数等于max-sessions
        //判断当前的session是否已经注册过了，如果注册过了，允许通过
        if (sessionCount == allowedSessions) {
            HttpSession session = request.getSession(false);

            if (session != null) {
                // Only permit it though if this request is associated with one of the already
                for (SessionInformation si : sessions) {
                    if (si.getSessionId().equals(session.getId())) {
                        return;
                    }
                }
            }
            // If the session is null, a new one will be created by the parent class, exceeding
        }
        //以上条件都不满足时，进一步处理
        allowableSessionsExceeded(sessions, allowedSessions, sessionRegistry);
    }

    protected void allowableSessionsExceeded(List<SessionInformation> sessions, int allowableSe
            SessionRegistry registry) throws SessionAuthenticationException {
        //判断配置的error-if-maximum-exceeded属性，如果为true，抛出异常
        if (exceptionIfMaximumExceeded || (sessions == null)) {
            throw new SessionAuthenticationException(messages.getMessage("ConcurrentSessionCont
                    new Object[] {new Integer(allowableSessions)},
                    "Maximum sessions of {0} for this principal exceeded"));
        }
        //如果配置的error-if-maximum-exceeded为false，接下来就是取出最先注册的session信息（这里是封
        // Determine least recently used session, and mark it for invalidation
        SessionInformation leastRecentlyUsed = null;

        for (int i = 0; i < sessions.size(); i++) {
            if ((leastRecentlyUsed == null)
                    || sessions.get(i).getLastRequest().before(leastRecentlyUsed.getLastRequest
                leastRecentlyUsed = sessions.get(i);
            }
```

```
        }
        leastRecentlyUsed.expireNow();
    }
```

经过以上分析，可以这么理解

如果concurrency-control标签配置了error-if-maximum-exceeded="true"，max-sessions="1"，那么第二次登录时，是登录不了的。如果error-if-maximum-exceeded="false"，那么第二次是能够登录到系统的，但是第一个登录的账号再次发起请求时，会跳转到expired-url配置的url中（如果没有配置，则显示This session has been expired (possibly due to multiple concurrent logins being attempted as the same user).提示信息）

由于篇幅过长，SessionManagementFilter、org.springframework.security.web.session.HttpSessionEventPublisher就放到下部分再分析了

1.14 Spring Security3源码分析-SessionManagementFilter分析-下

发表时间: 2012-05-08 关键字: spring, security3

很多spring security3资料在介绍session的并发控制都要求配置HttpSessionEventPublisher的监听器，如下

```
    <listener>
        <listener-class>
org.springframework.security.web.session.HttpSessionEventPublisher
         </listener-class>
    </listener>
```

这个监听器实现了HttpSessionListener接口，主要监听sessionCreated、sessionDestroyed事件。看源码

```
public class HttpSessionEventPublisher implements HttpSessionListener {
    private static final Log log = LogFactory.getLog(HttpSessionEventPublisher.class);
    ApplicationContext getContext(ServletContext servletContext) {
        return WebApplicationContextUtils.getWebApplicationContext(servletContext);
    }

    public void sessionCreated(HttpSessionEvent event) {
        HttpSessionCreatedEvent e = new HttpSessionCreatedEvent(event.getSession());

        if (log.isDebugEnabled()) {
            log.debug("Publishing event: " + e);
        }
        //通过ApplicationContext的事件发布机制发布sessionCreated事件
        getContext(event.getSession().getServletContext()).publishEvent(e);
    }

    public void sessionDestroyed(HttpSessionEvent event) {
        HttpSessionDestroyedEvent e = new HttpSessionDestroyedEvent(event.getSession());

        if (log.isDebugEnabled()) {
            log.debug("Publishing event: " + e);
        }
        //通过ApplicationContext的事件发布机制发布sessionDestroyed事件
```

```
        getContext(event.getSession().getServletContext()).publishEvent(e);
    }
}
```

由于ApplicationContext继承了ApplicationEventPublisher接口，所以ApplicationContext具有发布事件的能力。
Spring中与事件有关的接口和类主要包括ApplicationEvent、ApplicationListener。
定义一个事件的类需要继承ApplicationEvent或者ApplicationContextEvent抽象类。针对一种事件，需要特定的监听器，监听器需要实现ApplicationListener接口。当监听器接收到一个事件的时候，就会执行它的onApplicationEvent()方法。

上面的代码仅仅发布了session事件，针对session事件，谁去监听并处理呢？
spring security中监听session事件的类是
SessionRegistryImpl（org.springframework.security.core.session.SessionRegistryImpl）
这个类实现了ApplicationListener<SessionDestroyedEvent>接口，但是这个类在哪里注册到ioc容器中的呢。
如果看完上一篇的分析，应该知道是解析标签concurrency-control时创建这个bean并注册到ioc容器的。如果没有配置这个标签，这个类的bean就不会产生。

接着看SessionRegistryImpl类中的处理事件的方法onApplicationEvent

```
    //实际上SessionRegistryImpl仅仅处理SessionDestroyedEvent
    //接收到session失效事件，从当前缓存中清除SessionInformation
    public void onApplicationEvent(SessionDestroyedEvent event) {
        String sessionId = event.getId();
        removeSessionInformation(sessionId);
    }
    //根据失效的sessionid清除SessionInformation对象
    public void removeSessionInformation(String sessionId) {
        Assert.hasText(sessionId, "SessionId required as per interface contract");

        SessionInformation info = getSessionInformation(sessionId);

        if (info == null) {
            return;
        }
```

```
        //从sessionIds中清除SessionInformation
        sessionIds.remove(sessionId);

        Set<String> sessionsUsedByPrincipal = principals.get(info.getPrincipal());

        if (sessionsUsedByPrincipal == null) {
            return;
        }
        //从principals清除SessionInformation对应的认证实体信息
        synchronized (sessionsUsedByPrincipal) {
            sessionsUsedByPrincipal.remove(sessionId);

            if (sessionsUsedByPrincipal.size() == 0) {
                principals.remove(info.getPrincipal());
            }
        }
    }
```

现在，HttpSessionEventPublisher监听器的目的就很明显了。这个过滤器首先监听session失效的事件（web容器配置的timeout、直接调用session.invalidate方法），然后由SessionRegistryImpl处理session失效事件，从自己维护的集合缓存中清除已经失效的session信息以及session对应的认证实体信息，避免造成oom。

这里要重点区分下，上一篇分析的ConcurrentSessionFilter过滤器也是处理session失效，但是它所处理的仅仅封装后的SessionInformation类，如果这个类满足失效条件，再执行session.invalidate强制失效。

1.15 Spring Security3源码分析-ExceptionTranslationFilter分析

发表时间: 2012-05-09 关键字: security, spring

ExceptionTranslationFilter过滤器对应的类路径为
org.springframework.security.web.access.ExceptionTranslationFilter
从类名就看出这个过滤器用于异常翻译的。但是从这个过滤器在filterchain中的位置来看，它仅仅处于倒数第三的位置（这个filter后面分为是FilterSecurityInterceptor、SwitchUserFilter），所以ExceptionTranslationFilter只能捕获到后面两个过滤器所抛出的异常。
这里需要强调一下，spring security中的异常类基本上都继承RuntimeException。

接着看ExceptionTranslationFilter执行过程

```
//doFilter拦截到请求时，不做处理。仅仅处理后面filter所抛出的异常
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;

    try {
        chain.doFilter(request, response);
    }
    catch (IOException ex) {
        throw ex;
    }
    catch (Exception ex) {
        //这里主要是从异常堆栈中提取SpringSecurityException
        Throwable[] causeChain = throwableAnalyzer.determineCauseChain(ex);
        RuntimeException ase = (AuthenticationException)
                throwableAnalyzer.getFirstThrowableOfType(AuthenticationException.class, ca

        if (ase == null) {
            ase = (AccessDeniedException)throwableAnalyzer.getFirstThrowableOfType(AccessDe
        }
        //如果提取到安全异常，则进行处理
        if (ase != null) {
            handleException(request, response, chain, ase);
        } else {
```

```
            //没有安全异常，继续抛出
            // Rethrow ServletExceptions and RuntimeExceptions as-is
            if (ex instanceof ServletException) {
                throw (ServletException) ex;
            }
            else if (ex instanceof RuntimeException) {
                throw (RuntimeException) ex;
            }
            throw new RuntimeException(ex);
        }
    }
}
//处理安全异常
private void handleException(HttpServletRequest request, HttpServletResponse response, Filt
        RuntimeException exception) throws IOException, ServletException {
    //如果是认证异常，由sendStartAuthentication处理
    if (exception instanceof AuthenticationException) {
        sendStartAuthentication(request, response, chain, (AuthenticationException) excepti
    }
    //如果是访问拒绝异常，由访问拒绝处理类的handle处理
    else if (exception instanceof AccessDeniedException) {
        if (authenticationTrustResolver.isAnonymous(SecurityContextHolder.getContext().getA
            sendStartAuthentication(request, response, chain, new InsufficientAuthenticatic
                    "Full authentication is required to access this resource"));
        }
        else {
            accessDeniedHandler.handle(request, response, (AccessDeniedException) exceptior
        }
    }
}
```

先分析如何处理认证异常

```
//处理认证异常
protected void sendStartAuthentication(HttpServletRequest request, HttpServletResponse resp
        AuthenticationException reason) throws ServletException, IOException {
```

```
        // SEC-112: Clear the SecurityContextHolder's Authentication, as the
        // existing Authentication is no longer considered valid
        //首先把SecurityContext中的认证实体置空
        SecurityContextHolder.getContext().setAuthentication(null);
        //通过cache保存当前的请求信息（分析RequestCacheAwareFilter时再深入）
        requestCache.saveRequest(request, response);
        logger.debug("Calling Authentication entry point.");
        //由认证入口点开始处理
        authenticationEntryPoint.commence(request, response, reason);
    }
```

这里补充一下

authenticationEntryPoint是由配置http标签时，通过什么认证入口来决定注入相应的入口点bean的。请看下面的对应关系列表

form-login认证:LoginUrlAuthenticationEntryPoint

http-basic认证:BasicAuthenticationEntryPoint

openid-login认证:LoginUrlAuthenticationEntryPoint

x509认证:Http403ForbiddenEntryPoint

就不一一分析每个EntryPoint了，着重看一下LoginUrlAuthenticationEntryPoint

```
    //主要目的是完成跳转任务
     //创建该bean时，只注入了loginFormUrl属性，其他类变量均为默认值
    public void commence(HttpServletRequest request, HttpServletResponse response, Authenticati
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        String redirectUrl = null;
        //默认为false
        if (useForward) {
            if (forceHttps && "http".equals(request.getScheme())) {
                redirectUrl = buildHttpsRedirectUrlForRequest(httpRequest);
            }

            if (redirectUrl == null) {
```

```
            String loginForm = determineUrlToUseForThisRequest(httpRequest, httpResponse, a
            RequestDispatcher dispatcher = httpRequest.getRequestDispatcher(loginForm);
            dispatcher.forward(request, response);
            return;
        }
    } else {
        //返回的url为loginFormUrl配置的值，如果未配置，跳转到默认登录页面/spring_security_login
        redirectUrl = buildRedirectUrlToLoginPage(httpRequest, httpResponse, authException)

    }
    redirectStrategy.sendRedirect(httpRequest, httpResponse, redirectUrl);
}
```

接着分析访问拒绝类异常的处理过程，看AccessDeniedHandlerImpl的handle方法

```
public void handle(HttpServletRequest request, HttpServletResponse response, AccessDeniedEx
        throws IOException, ServletException {
    if (!response.isCommitted()) {
        //如果配置了access-denied-page属性，跳转到指定的url
        if (errorPage != null) {
            // Put exception into request scope (perhaps of use to a view)
            request.setAttribute(SPRING_SECURITY_ACCESS_DENIED_EXCEPTION_KEY, accessDeniedE

            // Set the 403 status code.
            response.setStatus(HttpServletResponse.SC_FORBIDDEN);

            // forward to error page.
            RequestDispatcher dispatcher = request.getRequestDispatcher(errorPage);
            dispatcher.forward(request, response);
        //如果没有配置，则直接响应403禁止访问的错误信息到浏览器端
        } else {
            response.sendError(HttpServletResponse.SC_FORBIDDEN, accessDeniedException.getM
        }
    }
}
```

通过以上分析，可以大体上认识到ExceptionTranslationFilter主要拦截两类安全异常：认证异常、访问拒绝异常。而且仅仅是捕获FilterSecurityInterceptor、SwitchUserFilter以及自定义拦截器的异常。所以在自定义拦截器时，需要注意在链中的顺序。

在上面分析过程中，有requestCache.saveRequest(request, response);的语句，具体requestCache的用途下篇分析。

1.16 Spring Security3源码分析-RequestCacheAwareFilter分析

发表时间: 2012-05-09 关键字: spring, security

RequestCacheAwareFilter过滤器对应的类路径为
org.springframework.security.web.savedrequest.RequestCacheAwareFilter

这个filter的用途官方解释是
用于用户登录成功后，重新恢复因为登录被打断的请求
这个解释也有几点需要说明
被打算的请求：简单点说就是出现了AuthenticationException、AccessDeniedException两类异常
重新恢复：既然能够恢复，那肯定请求信息被保存到cache中了

首先看被打断请求是如何保存到cache中的
实际上，上一篇的ExceptionTranslationFilter分析已经提到了
requestCache.saveRequest(request, response)
是的，如果出现AuthenticationException或者是匿名登录的抛出了AccessDeniedException，都会把当前request保存到cache中。这里的cache是HttpSessionRequestCache，接着看HttpSessionRequestCache的saveRequest方法

```
public void saveRequest(HttpServletRequest request, HttpServletResponse response) {
    //由于构造HttpSessionRequestCache的bean时，没有设置justUseSavedRequestOnGet属性，所以该属性
    if (!justUseSavedRequestOnGet || "GET".equals(request.getMethod())) {
        //构造DefaultSavedRequest，并且设置到session中
        DefaultSavedRequest savedRequest = new DefaultSavedRequest(request, portResolver);

        if (createSessionAllowed || request.getSession(false) != null) {
            request.getSession().setAttribute(DefaultSavedRequest.SPRING_SECURITY_SAVED_REQ
        }
    }

}
```

这里应该知道，实际上被打断的请求被封装成DefaultSavedRequest对象保存到session中了

分析完保存被打断的请求，接着就分析如何恢复被打断的请求了。RequestCacheAwareFilter过滤器就是完成恢复的工作。看doFilter方法

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
    //根据当前session取出DefaultSavedRequest，如果有被打断的请求，就把当前请求与被打断请求做匹配
    HttpServletRequest wrappedSavedRequest =
        requestCache.getMatchingRequest((HttpServletRequest)request, (HttpServletResponse)r
    chain.doFilter(wrappedSavedRequest == null ? request : wrappedSavedRequest, response);
}
```

继续看HttpSessionRequestCache处理过程

```
//从当前session中提取DefaultSavedRequest对象
public SavedRequest getRequest(HttpServletRequest currentRequest, HttpServletResponse respo
    HttpSession session = currentRequest.getSession(false);

    if (session != null) {
        return (DefaultSavedRequest) session.getAttribute(DefaultSavedRequest.SPRING_SECURI
    }

    return null;
}
//清除被打断请求
public void removeRequest(HttpServletRequest currentRequest, HttpServletResponse response)
    HttpSession session = currentRequest.getSession(false);

    if (session != null) {
        logger.debug("Removing DefaultSavedRequest from session if present");
        session.removeAttribute(DefaultSavedRequest.SPRING_SECURITY_SAVED_REQUEST_KEY);
    }
}
//请求匹配
public HttpServletRequest getMatchingRequest(HttpServletRequest request, HttpServletRespons
    DefaultSavedRequest saved = (DefaultSavedRequest) getRequest(request, response);
    //如果没有被打断请求，直接返回null，不做处理
    if (saved == null) {
        return null;
    }
```

```
        //如果当前请求与被打断请求不匹配，直接返回null，不做处理
        if (!saved.doesRequestMatch(request, portResolver)) {
            logger.debug("saved request doesn't match");
            return null;
        }
        //清除被打断请求
        removeRequest(request, response);
        //重新包装当前请求为被打断请求的各项信息
        return new SavedRequestAwareWrapper(saved, request);
    }
```

接着分析doesRequestMatch方法，看请求是如何匹配的

```
    //就是比较request与cache中被打断请求的各项信息是否相同
    //这里有个疑惑（由于被打断请求包括POST、GET提交方式的，而这里要求必须为GET方式的请求才会匹配成功）
    public boolean doesRequestMatch(HttpServletRequest request, PortResolver portResolver) {

        if (!propertyEquals("pathInfo", this.pathInfo, request.getPathInfo())) {
            return false;
        }

        if (!propertyEquals("queryString", this.queryString, request.getQueryString())) {
            return false;
        }

        if (!propertyEquals("requestURI", this.requestURI, request.getRequestURI())) {
            return false;
        }

        if (!"GET".equals(request.getMethod()) && "GET".equals(method)) {
            // A save GET should not match an incoming non-GET method
            return false;
        }

        if (!propertyEquals("serverPort", new Integer(this.serverPort), new Integer(portResolve
```

```
        {
            return false;
        }

        if (!propertyEquals("requestURL", this.requestURL, request.getRequestURL().toString())
            return false;
        }

        if (!propertyEquals("scheme", this.scheme, request.getScheme())) {
            return false;
        }

        if (!propertyEquals("serverName", this.serverName, request.getServerName())) {
            return false;
        }

        if (!propertyEquals("contextPath", this.contextPath, request.getContextPath())) {
            return false;
        }

        if (!propertyEquals("servletPath", this.servletPath, request.getServletPath())) {
            return false;
        }

        return true;
    }
```

这里为何对POST请求匹配不成功，目前还不知道具体设计思路。知道后会进行补充

1.17 Spring Security3源码分析-Filter链排序分析

发表时间: 2012-05-09 关键字: spring, security

通过前面Spring Security提供的各种Filter的分析，大体上知道每个Filter具体的用途了。
Spring Security一共提供了20个Filter，我目前只分析了13个（如果http的auto-config="true"，那默认的filter列表都包含在这13个里面了），另外7个在后面的源码分析中碰到时会逐个讲解。
在分析http标签时，已经提到filter排序的问题了，但是没有深入。

现在再回头看filter是如何排序的。下面的代码片段截取自HttpSecurityBeanDefinitionParser类，

```
        //定义未排序filter集合。该集合中的对象为OrderDecorator实例。
        List<OrderDecorator> unorderedFilterChain = new ArrayList<OrderDecorator>();
        //添加http、认证相关的filter集合
        unorderedFilterChain.addAll(httpBldr.getFilters());
        unorderedFilterChain.addAll(authBldr.getFilters());
        //定义RequestCacheAwareFilter过滤器，并添加到unorderedFilterChain中
        BeanDefinition requestCacheAwareFilter = new RootBeanDefinition(RequestCacheAwareFilter
        requestCacheAwareFilter.getPropertyValues().addPropertyValue("requestCache", authBldr.g
        unorderedFilterChain.add(new OrderDecorator(requestCacheAwareFilter, REQUEST_CACHE_FILT
        //添加自定义filter
        unorderedFilterChain.addAll(buildCustomFilterList(element, pc));
        //根据排序规则进行排序
        Collections.sort(unorderedFilterChain, new OrderComparator());
        //检查每个filter与前一个filter的位置是否相同
         //这里的检查主要是防止自定义filter直接配置position属性，造成与默认的filter产生order冲突
        checkFilterChainOrder(unorderedFilterChain, pc, source);
        //重新定义filterChain，把经过排序的filter依次添加到filterChain集合中
        List<BeanMetadataElement> filterChain = new ManagedList<BeanMetadataElement>();

        for (OrderDecorator od : unorderedFilterChain) {
            filterChain.add(od.bean);
        }
```

细心的同学会发现httpBldr.getFilters()、authBldr.getFilters()两个方法返回的就是OrderDecorator对象的集合列表。并且OrderDecorator对象已经将filter与SecurityFilters中的order关联了起来

```
        unorderedFilterChain.addAll(httpBldr.getFilters());
        unorderedFilterChain.addAll(authBldr.getFilters());
```

顺便看一下创建自定义过滤器部分

```
    List<OrderDecorator> buildCustomFilterList(Element element, ParserContext pc) {
        List<Element> customFilterElts = DomUtils.getChildElementsByTagName(element, Elements.C
        List<OrderDecorator> customFilters = new ArrayList<OrderDecorator>();

        final String ATT_AFTER = "after";
        final String ATT_BEFORE = "before";
        final String ATT_POSITION = "position";
        //循环自定义标签列表custom-filter
        for (Element elt: customFilterElts) {
            String after = elt.getAttribute(ATT_AFTER);
            String before = elt.getAttribute(ATT_BEFORE);
            String position = elt.getAttribute(ATT_POSITION);

            String ref = elt.getAttribute(ATT_REF);

            if (!StringUtils.hasText(ref)) {
                pc.getReaderContext().error("The '" + ATT_REF + "' attribute must be supplied",
            }

            RuntimeBeanReference bean = new RuntimeBeanReference(ref);

            if(WebConfigUtils.countNonEmpty(new String[] {after, before, position}) != 1) {
                pc.getReaderContext().error("A single '" + ATT_AFTER + "', '" + ATT_BEFORE + "'
                        ATT_POSITION + "' attribute must be supplied", pc.extractSource(elt));
            }
            //如果指定了position，直接将filter与order产生关联关系
            if (StringUtils.hasText(position)) {
                customFilters.add(new OrderDecorator(bean, SecurityFilters.valueOf(position)));
            //如果指定了after，将filter与after值加一产生关联关系
            } else if (StringUtils.hasText(after)) {
```

```
                SecurityFilters order = SecurityFilters.valueOf(after);
                if (order == SecurityFilters.LAST) {
                    customFilters.add(new OrderDecorator(bean, SecurityFilters.LAST));
                } else {
                    customFilters.add(new OrderDecorator(bean, order.getOrder() + 1));
                }
            //如果指定了before，将filter与before-1产生关联关系
            } else if (StringUtils.hasText(before)) {
                SecurityFilters order = SecurityFilters.valueOf(before);
                if (order == SecurityFilters.FIRST) {
                    customFilters.add(new OrderDecorator(bean, SecurityFilters.FIRST));
                } else {
                    customFilters.add(new OrderDecorator(bean, order.getOrder() - 1));
                }
            }
        }

        return customFilters;
    }
```

这里用到三个重要的与排序相关的类及枚举，分别是OrderDecorator、OrderComparator以及SecurityFilters枚举

首先看SecurityFilters枚举定义

```
enum SecurityFilters {
    FIRST (Integer.MIN_VALUE),
    //order=100
    CHANNEL_FILTER,
    //order=200
    CONCURRENT_SESSION_FILTER,
    //依次递增.....
    SECURITY_CONTEXT_FILTER,
    LOGOUT_FILTER,
    X509_FILTER,
    PRE_AUTH_FILTER,
```

```
    CAS_FILTER,
    FORM_LOGIN_FILTER,
    OPENID_FILTER,
    LOGIN_PAGE_FILTER,
    DIGEST_AUTH_FILTER,
    BASIC_AUTH_FILTER,
    REQUEST_CACHE_FILTER,
    SERVLET_API_SUPPORT_FILTER,
    REMEMBER_ME_FILTER,
    ANONYMOUS_FILTER,
    SESSION_MANAGEMENT_FILTER,
    EXCEPTION_TRANSLATION_FILTER,
    FILTER_SECURITY_INTERCEPTOR,
    SWITCH_USER_FILTER,
    LAST (Integer.MAX_VALUE);
    //这里设置100，主要给自定义过滤器提供after、before的预留位置
    //也就是说，在某个默认的过滤器前后只能自定义99个过滤器，虽然可能性几乎为0
    private static final int INTERVAL = 100;
    private final int order;
    //返回的order值=序号*间隔100
    private SecurityFilters() {
        order = ordinal() * INTERVAL;
    }

    private SecurityFilters(int order) {
        this.order = order;
    }
    //主要通过该方法返回Filter的位置
    public int getOrder() {
       return order;
    }
}
```

由此可见，该类维护了Spring Security中每个filter的顺序

接着看OrderDecorator类。这个类实现org.springframework.core.Ordered接口

```
class OrderDecorator implements Ordered {
    BeanMetadataElement bean;
    int order;
    //构造函数传递两个参数1.bean定义；2.filter在链中的位置
    public OrderDecorator(BeanMetadataElement bean, SecurityFilters filterOrder) {
        this.bean = bean;
        this.order = filterOrder.getOrder();
    }

    public OrderDecorator(BeanMetadataElement bean, int order) {
        this.bean = bean;
        this.order = order;
    }
    //实现接口方法getOrder
    public int getOrder() {
        return order;
    }

    public String toString() {
        return bean + ", order = " + order;
    }
}
```

OrderComparator类的路径是org.springframework.core.OrderComparator，实际上是spring core包的一个比较器，可以顺便看下OrderComparator源码。下面截取的只是部分核心代码

```
        public int compare(Object o1, Object o2) {
                boolean p1 = (o1 instanceof PriorityOrdered);
                boolean p2 = (o2 instanceof PriorityOrdered);
                if (p1 && !p2) {
                        return -1;
                }
                else if (p2 && !p1) {
                        return 1;
                }
                //前面几行代码主要针对PriorityOrdered，这里不做分析
```

```
                //分别获取Ordered接口实现类的getOrder方法得到order值
            int i1 = getOrder(o1);
            int i2 = getOrder(o2);
               //对得到的order进行比较
            return (i1 < i2) ? -1 : (i1 > i2) ? 1 : 0;
        }
          //获取Ordered接口的实现类，获取getOrder值
        protected int getOrder(Object obj) {
            return (obj instanceof Ordered ? ((Ordered) obj).getOrder() : Ordered.LOWEST_PR
        }
```

通过以上的分析，可以总结如下

1.由SecurityFilters维持位置order
2.由OrderDecorator维持filter与order的对应关系
3.由OrderComparator负责比较OrderDecorator的先后顺序

附上默认的过滤器顺序列表

order过滤器名称

100 ChannelProcessingFilter
200 ConcurrentSessionFilter
300 SecurityContextPersistenceFilter
400 LogoutFilter
500 X509AuthenticationFilter
600 RequestHeaderAuthenticationFilter
700 CasAuthenticationFilter
800 UsernamePasswordAuthenticationFilter
900 OpenIDAuthenticationFilter
1000 DefaultLoginPageGeneratingFilter
1100 DigestAuthenticationFilter
1200 BasicAuthenticationFilter
1300 RequestCacheAwareFilter

1400 SecurityContextHolderAwareRequestFilter

1500 RememberMeAuthenticationFilter

1600 AnonymousAuthenticationFilter

1700 SessionManagementFilter

1800 ExceptionTranslationFilter

1900 FilterSecurityInterceptor

2000 SwitchUserFilter

以上标注红色的都已经分析完毕

1.18 Spring Security3源码分析-认证授权分析

发表时间: 2012-05-09

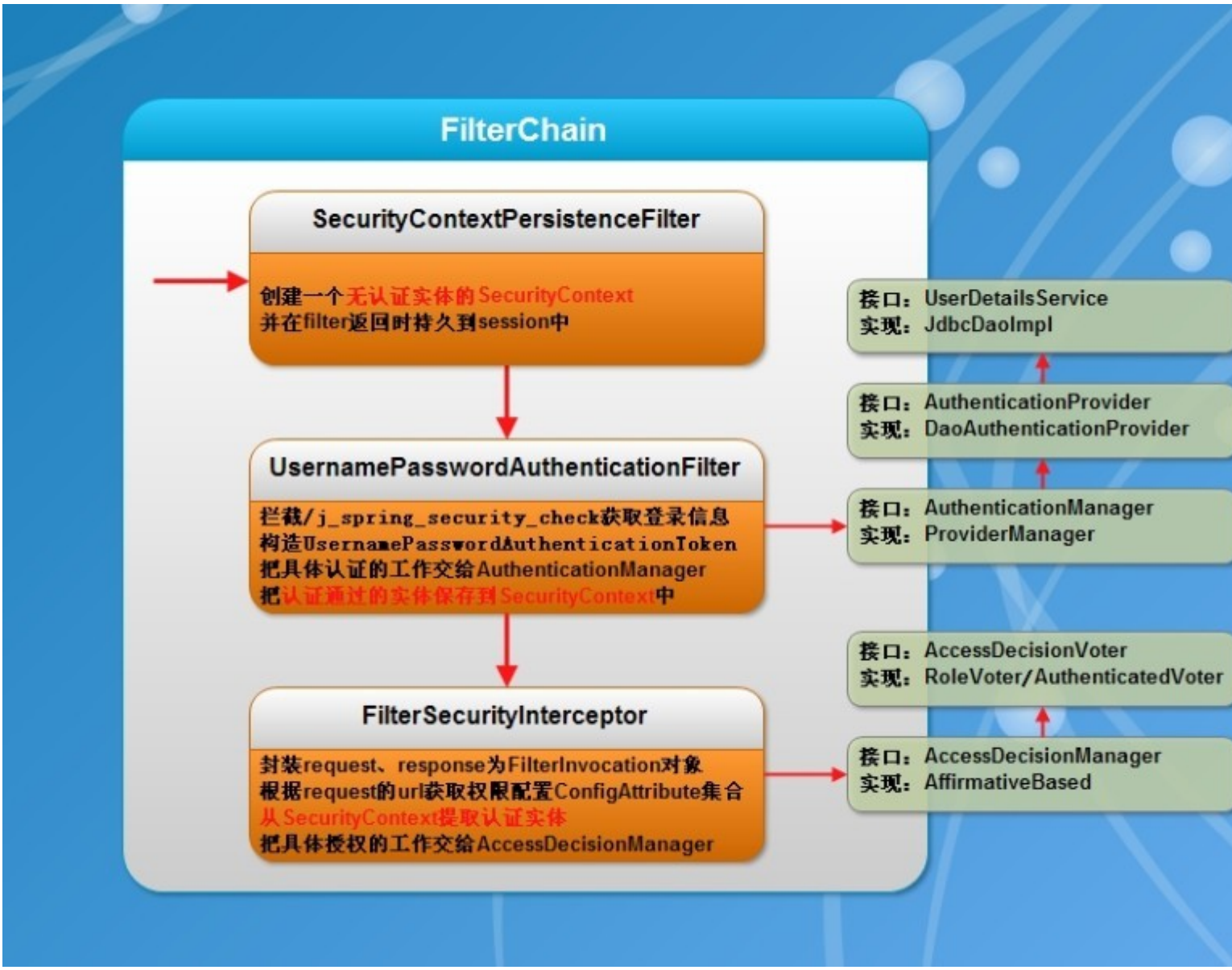
前面分析了FilterChainProxy执行过程，也对常用的filter逐一深入介绍了，但似乎忽略了Spring Security的核心功能：认证和授权。
虽然在介绍过滤器时也把认证、授权的具体过程深入分析了，但一直没有从整体设计的角度来观察认证、授权过程。

虽然前面介绍了很多过滤器处理过程，但个人认为真正对认证授权起决定影响的过滤器只有三个。
分别是

SecurityContextPersistenceFilter：创建安全上下文SecurityContext
UsernamePasswordAuthenticationFilter：完成认证处理，并把认证通过的实体保存到SecurityContext中
FilterSecurityInterceptor：完成授权处理

上面的UsernamePasswordAuthenticationFilter只是其中一种form-login认证方式，还有很多其他的方式，这里不一一描述了。仅仅选择form-login方式来简化观点的本质。

从下面的过滤器流向图应该能清晰描述这几个过滤器的核心处理过程



认证过程：

授权过程：


FilterSecurityInterceptor
AccessDecisionManager
AccessDecisionVoter

1.19 Spring Security3源码分析-SSL支持

发表时间: 2012-05-10

Sping Security3对于SSL的支持仅仅表现在对需要拦截的url（标签intercept-url）设置requires-channel=https属性。

如果一个url设置了requires-channel为https，那么该url在http的访问会直接重定向到https的通道中去。后面再具体分析。

首先需要在应用中配置SSL的支持，具体配置方法可参考

http://lengyun3566.iteye.com/blog/1141347

Sping Security3支持SSL分别表现下面几个类

类名称	用途描述
ChannelProcessingFilter	通道处理过滤器。只要intercept-url标签中包含requires-channel属性，该过滤器就被创建
ChannelDecisionManagerImpl	通道决策管理器。该管理器包含两个ChannelProcessor实例用于处理安全、不安全两种Channel方式
SecureChannelProcessor	安全通道处理器
InsecureChannelProcessor	不安全通道处理器
AbstractRetryEntryPoint	抽象的通道重操作入口点，是entrypoint的父类
RetryWithHttpEntryPoint	如果当前以安全通道访问不安全通道，也可以通过http的入口点重定向到不安全通道中
RetryWithHttpsEntryPoint	如果当前以不安全通道访问安全通道，就要通过https的入口点重定向到安全通道中
PortMapperImpl	端口映射处理。主要是针对非默认端口（80、8080、443、8443）的情况

看ChannelProcessingFilter过滤器的作用

ChannelProcessingFilter首先检查当前请求的url是否已配置了requires-channel属性，如果没配置，不处理。如果配置了，就把决策权交给ChannelDecisionManagerImpl处理。

ChannelProcessingFilter对应类路径：
org.springframework.security.web.access.channel.ChannelProcessingFilter

具体源码如下

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;

    FilterInvocation fi = new FilterInvocation(request, response, chain);
    //获取url的权限配置信息
    Collection<ConfigAttribute> attr = this.securityMetadataSource.getAttributes(fi);

    if (attr != null) {
        if (logger.isDebugEnabled()) {
            logger.debug("Request: " + fi.toString() + "; ConfigAttributes: " + attr);
        }
        //把决策权交给channelDecisionManager处理
        channelDecisionManager.decide(fi, attr);

        if (fi.getResponse().isCommitted()) {
            return;
        }
    }

    chain.doFilter(request, response);
}
```

接着看ChannelDecisionManagerImpl的作用

ChannelDecisionManagerImpl根据requires-channel的值做相应处理，requires-channel值有以下三种

any：任何通道都支持。决策管理器不做处理

https：只支持安全通道。决策管理器把决策任务交给ChannelProcessor列表循环处理

http：只支持http。决策管理器把决策任务交给ChannelProcessor列表循环处理

ChannelDecisionManagerImpl的源码为：

```
public void decide(FilterInvocation invocation, Collection<ConfigAttribute> config) throws

    Iterator<ConfigAttribute> attrs = config.iterator();
    //判断是否为any值
    while (attrs.hasNext()) {
        ConfigAttribute attribute = attrs.next();
        if (ANY_CHANNEL.equals(attribute.getAttribute())) {
            return;
        }
    }
    //循环ChannelProcessor列表执行decide
    for (ChannelProcessor processor : channelProcessors) {
        processor.decide(invocation, config);

        if (invocation.getResponse().isCommitted()) {
            break;
        }
    }
}
```

继续看ChannelProcessor 的作用

实际上在构造ChannelDecisionManager的bean时，已经注入了两个ChannelProcessor ，分别是SecureChannelProcessor、InsecureChannelProcessor

先看SecureChannelProcessor（负责处理安全通道）执行过程

```
public void decide(FilterInvocation invocation, Collection<ConfigAttribute> config) throws
    Assert.isTrue((invocation != null) && (config != null), "Nulls cannot be provided");
    for (ConfigAttribute attribute : config) {
        if (supports(attribute)) {
            if (!invocation.getHttpRequest().isSecure()) {
                entryPoint.commence(invocation.getRequest(), invocation.getResponse());
```

```
            }
        }
    }
}
```

根据当前的请求是否安全，进行相应的处理。实际工作的是抽象的父类AbstractRetryEntryPoint的commence完成

AbstractRetryEntryPoint的commence方法源码：

```
public void commence(HttpServletRequest request, HttpServletResponse res) throws IOExceptic
    String pathInfo = request.getPathInfo();
    String queryString = request.getQueryString();
    String contextPath = request.getContextPath();
    String destination = request.getServletPath() + ((pathInfo == null) ? "" : pathInfo)
        + ((queryString == null) ? "" : ("?" + queryString));

    String redirectUrl = contextPath;
    //获取当前请求所在端口
 Integer currentPort = new Integer(portResolver.getServerPort(request));
    //根据当前端口获得映射的端口（需要配置port-mappings标签），如果是http的访问，则获取映射的http
 Integer redirectPort = getMappedPort(currentPort);
    //如果获取到匹配端口，则根据当前请求构造重定向请求的url
    if (redirectPort != null) {
        boolean includePort = redirectPort.intValue() != standardPort;

        redirectUrl = scheme + request.getServerName() + ((includePort) ? (":" + redirectPo
            + destination;
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Redirecting to: " + redirectUrl);
    }
    //执行重定向操作
    res.sendRedirect(res.encodeRedirectURL(redirectUrl));
}
```

通过以上分析，应该很清楚的知道：

如果以**http**的方式登录到应用中，再访问配置了**requires-channel=https**的url时，就会重定向到https的通道去，以SSL方式访问。

如果以**https**的方式登录到应用中，再访问配置了**requires-channel=http**的url时，就会重定向到http的通道去，以不安全的方式访问。

1.20 Spring Security3源码分析-CAS支持

发表时间: 2012-05-13

Spring Security3对CAS的支持主要在这个spring-security-cas-client-3.0.2.RELEASE.jar包中

Spring Security和CAS集成的配置资料很多。这里讲解的比较详细

http://lengyun3566.iteye.com/blog/1358323

配置方面，主要为下面的部分：

```
<security:http auto-config="true" entry-point-ref="casAuthEntryPoint" access-denied-page="/er
            <security:custom-filter ref="casAuthenticationFilter" position="CAS_FILTER"/>
            <security:form-login login-page="/login.jsp"/>
            <security:logout logout-success-url="/login.jsp"/>
            <security:intercept-url pattern="/admin.jsp*" access="ROLE_ADMIN"/>
            <security:intercept-url pattern="/index.jsp*" access="ROLE_USER,ROLE_ADMIN"/>
            <security:intercept-url pattern="/home.jsp*" access="ROLE_USER,ROLE_ADMIN"/>
            <security:intercept-url pattern="/**" access="ROLE_USER,ROLE_ADMIN"/>
    </security:http>

    <security:authentication-manager alias="authenticationmanager">
            <security:authentication-provider ref="casAuthenticationProvider"/>
    </security:authentication-manager>

    <bean id="casAuthenticationProvider" class="org.springframework.security.cas.authentica
            <property name="ticketValidator" ref="casTicketValidator"/>
            <property name="serviceProperties" ref="casService"/>
            <property name="key" value="docms"/>
            <property name="authenticationUserDetailsService" ref="authenticationUserDetail
    </bean>

    <bean id="casAuthEntryPoint" class="org.springframework.security.cas.web.CasAuthenticat
            <property name="loginUrl" value="https://server:8443/cas/login"/>
            <property name="serviceProperties" ref="casService"/>
    </bean>
```

```
<bean id="casService" class="org.springframework.security.cas.ServiceProperties">
        <property name="service" value="http://localhost:8888/docms/j_spring_cas_securi
</bean>

<bean id="casAuthenticationFilter" class="org.springframework.security.cas.web.CasAuthe
        <property name="authenticationManager" ref="authenticationmanager"/>
</bean>

<bean id="casTicketValidator" class="org.jasig.cas.client.validation.Cas20ServiceTicket
        <constructor-arg value="https://server:8443/cas/"/>
</bean>

<bean id="authenticationUserDetailsService" class="org.springframework.security.core.us
        <property name="userDetailsService" ref="userDetailsManager"/>
</bean>
```

这里需要强调一下http标签的entry-point-ref属性，因为之前没有着重的介绍，英文的意思是入口点引用。为什么需要这个入口点呢。这个入口点其实仅仅是被ExceptionTranslationFilter引用的。前面已经介绍过ExceptionTranslationFilter过滤器的作用是异常翻译，在出现认证异常、访问异常时，通过入口点决定redirect、forward的操作。比如现在是form-login的认证方式，如果没有通过UsernamePasswordAuthenticationFilter的认证就直接访问某个被保护的url，那么经过ExceptionTranslationFilter过滤器处理后，先捕获到访问拒绝异常，并把跳转动作交给入口点来处理。form-login的对应入口点类为LoginUrlAuthenticationEntryPoint，这个入口点类的commence方法会redirect或forward到指定的url（form-login标签的login-page属性）

清楚了entry-point-ref属性的意义。那么与CAS集成时，如果访问一个受保护的url，就通过CAS认证对应的入口点org.springframework.security.cas.web.CasAuthenticationEntryPoint类redirect到loginUrl属性所配置的url中，即一般为CAS的认证页面（比如：https://server:8443/cas/login）。

下面为CasAuthenticationEntryPoint类的commence方法。其主要任务就是构造跳转的url，再执行redirect动作。根据上面的配置，实际上跳转的url为：https://server:8443/cas/login?service=http%3A%2F%2Flocalhost%3A8888%2Fdocms%2Fj_spring_cas_security_check

```
    public final void commence(final HttpServletRequest servletRequest, final HttpServletRespor
            final AuthenticationException authenticationException) throws IOException, ServletE

        final String urlEncodedService = createServiceUrl(servletRequest, response);
        final String redirectUrl = createRedirectUrl(urlEncodedService);

        preCommence(servletRequest, response);
        response.sendRedirect(redirectUrl);
    }
```

接下来继续分析custom-filter ref="casAuthenticationFilter" position="CAS_FILTER"

这是一个自定义标签，并且在过滤器链中的位置是CAS_FILTER。这个过滤器在何时会起作用呢？带着这个疑问继续阅读源码

CasAuthenticationFilter对应的类路径是

org.springframework.security.cas.web.CasAuthenticationFilter

这个类与UsernamePasswordAuthenticationFilter一样，都继承于AbstractAuthenticationProcessingFilter。实际上所有认证过滤器都继承这个抽象类，其过滤器本身只要实现attemptAuthentication方法即可。

CasAuthenticationFilter的构造方法直接向父类的构造方法传入/j_spring_cas_security_check用于判断当前请求的url是否需要进一步的认证处理

```
    public CasAuthenticationFilter() {
        super("/j_spring_cas_security_check");
    }
```

CasAuthenticationFilter类的attemptAuthentication方法源码如下

```
    public Authentication attemptAuthentication(final HttpServletRequest request, final HttpSer
            throws AuthenticationException {
        //设置用户名为有状态标识符
        final String username = CAS_STATEFUL_IDENTIFIER;
```

```
        //获取CAS认证成功后返回的ticket
        String password = request.getParameter(this.artifactParameter);

        if (password == null) {
            password = "";
        }
        //构造UsernamePasswordAuthenticationToken对象
        final UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticat

        authRequest.setDetails(authenticationDetailsSource.buildDetails(request));
        //由认证管理器完成认证工作
        return this.getAuthenticationManager().authenticate(authRequest);
    }
```

在之前的源码分析中，已经详细分析了认证管理器AuthenticationManager认证的整个过程，这里就不再赘述了。

由于AuthenticationManager是依赖于具体的AuthenticationProvider的，所以接下来看

```
<security:authentication-manager alias="authenticationmanager">
<security:authentication-provider ref="casAuthenticationProvider"/>
</security:authentication-manager>
```

注意这里的ref属性定义。如果没有使用CAS认证，此处一般定义user-service-ref属性。这两个属性的区别在于

ref：直接将ref依赖的bean注入到AuthenticationProvider的providers集合中

user-service-ref：定义DaoAuthenticationProvider的bean注入到AuthenticationProvider的providers集合中，并且DaoAuthenticationProvider的变量userDetailsService由user-service-ref依赖的bean注入。

由此可见，采用CAS认证时，AuthenticationProvider只有AnonymousAuthenticationProvider和CasAuthenticationProvider

继续分析CasAuthenticationProvider是如何完成认证工作的

```
public Authentication authenticate(Authentication authentication) throws AuthenticationExce
    //省略若干判断
    CasAuthenticationToken result = null;
    //注意这里的无状态条件。主要用于无httpsession的环境中。如soap调用
    if (stateless) {
        // Try to obtain from cache
        //通过缓存来存储认证实体。主要避免每次请求最新ticket的网络开销
        result = statelessTicketCache.getByTicketId(authentication.getCredentials().toStrir
    }

    if (result == null) {
        result = this.authenticateNow(authentication);
        result.setDetails(authentication.getDetails());
    }

    if (stateless) {
        // Add to cache
        statelessTicketCache.putTicketInCache(result);
    }

    return result;
}

//完成认证工作
private CasAuthenticationToken authenticateNow(final Authentication authentication) throws
    try {
        //通过cas client的ticketValidator完成ticket校验，并返回身份断言
        final Assertion assertion = this.ticketValidator.validate(authentication.getCredent
       //根据断言信息构造UserDetails
        final UserDetails userDetails = loadUserByAssertion(assertion);
       //检查账号状态
       userDetailsChecker.check(userDetails);
       //构造CasAuthenticationToken
        return new CasAuthenticationToken(this.key, userDetails, authentication.getCredenti
    } catch (final TicketValidationException e) {
        throw new BadCredentialsException(e.getMessage(), e);
    }
```

```
    }

    //通过注入的authenticationUserDetailsService根据token中的认证主体即用户名获取UserDetails
    protected UserDetails loadUserByAssertion(final Assertion assertion) {
        final CasAssertionAuthenticationToken token = new CasAssertionAuthenticationToken(asser
        return this.authenticationUserDetailsService.loadUserDetails(token);
    }
```

需要注意的是为什么要定义authenticationUserDetailsService这个bean。由于CAS需要authentication-manager标签下定义<security:authentication-provider ref="casAuthenticationProvider"/>，而不是之前所介绍的
user-service-ref属性，所以这里仅仅定义了一个provider，而没有注入UserDetailsService，所以这里需要单独定义authenticationUserDetailsService这个bean，并注入到CasAuthenticationProvider中。

这里需要对CasAuthenticationToken、CasAssertionAuthenticationToken单独解释一下

CasAuthenticationToken：一个成功通过的CAS认证，与UsernamePasswordAuthenticationToken一样，都是继承于AbstractAuthenticationToken，并且最终会保存到SecurityContext上下文、session中

CasAssertionAuthenticationToken：一个临时的认证对象用于辅助获取UserDetails

配置文件中几个bean定义这里就不一一分析了，都是为了辅助完成CAS认证、跳转的工作。

现在，可以对整个CAS认证的过程总结一下了：

1.客户端发起一个请求，试图访问系统系统中受保护的url

2.各filter链进行拦截并做相应处理，由于没有通过认证，ExceptionTranslationFilter过滤器会捕获到访问拒绝异常，并把该异常交给入口点处理

3.CAS 认证对应的入口点直接跳转到CAS Server端的登录界面，并携带参数service（一般为url：...../j_spring_cas_security_check)

4.CAS Server对登录信息进行处理，如果登录成功，就跳转到应用系统中service指定的url，并携带ticket

5.应用系统中的各filter链再次对该url拦截，此时CasAuthenticationFilter拦截到j_spring_cas_security_check，就会对ticket进行验证，验证成功返回一个身份断言，再通过身份断言从当前应

用系统中获取对应的UserDetails、GrantedAuthority。此时，如果步骤1中受保护的url权限列表有一个权限存在于GrantedAuthority列表中，说明有权限访问，直接响应客户端所试图访问的url