Shiro框架Web环境下过滤器结构分析

分类： J2EE开发之权限-shiro2011-12-02 18:21 1988人阅读 评论(6) 收藏 举报

shiro框架websessionexceptionfilter

Shiro的过滤器的配置是结合使用Spring的DelegatingFilterProxy与FactoryBean2种技术来完成自身过滤器的植入的，所以理解Shiro的过滤器首先要理解这2者的使用。

1. DelegatingFilterProxy

Spring提供的一个简便的过滤器的处理方案，它将具体的操作交给内部的Filter对象delegate去处理，而这个delegate对象通过Spring IOC容器获取，这里采用的是Spring的FactoryBean的方式获取这个对象。

DelegatingFilterProxy的配置如下

[html] view plaincopyprint?

```
1. <filter>
    <filter-name>shiroFilter</filter-name>
    <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filte
r-class>
    <init-param>
        <param-name>targetFilterLifecycle</param-name>
        <param-value>true</param-value>
    </init-param>
8. </filter>
```

虽然只配置了这一个filter，但是它并做任何实际的工作，而是把工作交由Spring中容器为bean的名字shiroFilter的类，即ShiroFilterFactoryBean；

2. ShiroFilterFactoryBean

配置如下

[html] view plaincopyprint?

```
1. <bean id="shiroFilter"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    ..
3. </bean>
```

由于它是个FactroyBean，所以上面的delegate真正的对象是通过它的getObject()获取的。

这里是FactoryBean接口获取实例的标准方法

[html] view plaincopyprint?

```
public Object getObject() throws Exception {
        if (instance == null) {
            instance = createInstance();
        }
        return instance;
    }
```

这里是真正创建对象的方法

[html] view plaincopyprint?

```
    protected AbstractShiroFilter createInstance() throws
Exception {

        log.debug("Creating Shiro Filter instance.");

        SecurityManager securityManager =
getSecurityManager();
        if (securityManager == null) {
            String msg = "SecurityManager property must be
set.";
            throw new BeanInitializationException(msg);
        }

        if (!(securityManager instanceof WebSecurityManager))
{
            String msg = "The security manager does not
implement the WebSecurityManager interface.";
            throw new BeanInitializationException(msg);
        }

        FilterChainManager manager =
createFilterChainManager();

        //Expose the constructed FilterChainManager by first
wrapping it in a
        // FilterChainResolver implementation. The
AbstractShiroFilter implementations
        // do not know about FilterChainManagers - only
resolvers:
        PathMatchingFilterChainResolver chainResolver = new
```

```
PathMatchingFilterChainResolver();
            chainResolver.setFilterChainManager(manager);

            //Now create a concrete ShiroFilter instance and
apply the acquired SecurityManager and built
            //FilterChainResolver.  It doesn't matter that the
instance is an anonymous inner class
            //here - we're just using it because it is a concrete
AbstractShiroFilter instance that accepts
            //injection of the SecurityManager and
FilterChainResolver:
            return new SpringShiroFilter((WebSecurityManager)
securityManager, chainResolver);
        }
```

所以真正完成实际工作的过滤器是SpringShiroFilter，这个对象才是真正的delegate。

3. SpringShiroFilter: ShiroFilterFactoryBean的内部类，继承AbstractShiroFilter

[html] view plaincopyprint?

```
    private static final class SpringShiroFilter extends
AbstractShiroFilter {

        protected SpringShiroFilter(WebSecurityManager
webSecurityManager, FilterChainResolver resolver) {
            super();
            if (webSecurityManager == null) {
                throw new
IllegalArgumentException("WebSecurityManager property cannot be
null.");
            }
            setSecurityManager(webSecurityManager);
            if (resolver != null) {
                setFilterChainResolver(resolver);
            }
        }
    }
```

4. OncePerRequestFilter： AbstractShiroFilter的父类

关键方法

[html] view plaincopyprint?

```
    protected abstract void doFilterInternal(ServletRequest
```

```
request, ServletResponse response, FilterChain chain)
            throws ServletException, IOException;
```

这个方法有过滤器中调用：

[html] view plaincopyprint?

```
        public final void doFilter(ServletRequest request,
    ServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
            String alreadyFilteredAttributeName =
    getAlreadyFilteredAttributeName();
            if (request.getAttribute(alreadyFilteredAttributeName)
    != null || shouldNotFilter(request)) {
            log.trace("Filter '{}' already executed.
    Proceeding without invoking this filter.", getName());
            // Proceed without invoking this filter...
            filterChain.doFilter(request, response);
        } else {
            // Do invoke this filter...
             log.trace("Filter '{}' not yet executed.
    Executing now.", getName());

    request.setAttribute(alreadyFilteredAttributeName, Boolean.TRUE);

            try {
                doFilterInternal(request, response,
    filterChain);
            } finally {
                // Once the request has finished, we're done
    and we don't
                // need to mark as 'already filtered' any
    more.

    request.removeAttribute(alreadyFilteredAttributeName);
            }
        }
    }
```

doFilterInternal这个方法有2处实现，1是AbstractShiroFilter的实现，2是AdviceFilter的实现。通过查看shiro的内定义的Filter继承结构可以看出，除了SpringShiroFilter这个内部类是继承前者，其他所有的用到的Filter都是继承后者。SpringShiroFilter是每次请求的第一个真正处理实际工作的Filter(主要是创建一个Subject并绑定相关数据)。

5. AbstractShiroFilter：OncePerRequestFilter的第一个子类

[html] view plaincopyprint?

```
        protected void doFilterInternal(ServletRequest servletRequest,
```

```
ServletResponse servletResponse, final FilterChain chain)
            throws ServletException, IOException {

        Throwable t = null;

        try {
            final ServletRequest request =
prepareServletRequest(servletRequest, servletResponse, chain);
            final ServletResponse response =
prepareServletResponse(request, servletResponse, chain);

            final Subject subject = createSubject(request,
response);

            //noinspection unchecked
            subject.execute(new Callable() {
                public Object call() throws Exception {
                    updateSessionLastAccessTime(request,
response);
                    executeChain(request, response, chain);
                    return null;
                }
            });
        } catch (ExecutionException ex) {
            t = ex.getCause();
        } catch (Throwable throwable) {
            t = throwable;
        }

        if (t != null) {
            if (t instanceof ServletException) {
                throw (ServletException) t;
            }
            if (t instanceof IOException) {
                throw (IOException) t;
            }
            //otherwise it's not one of the two exceptions
expected by the filter method signature - wrap it in one:
            String msg = "Filtered request failed.";
            throw new ServletException(msg, t);
        }
    }
```

这段代码表示每次经过AbstractShiroFilter的doFilterInternal方法(具体的类也就是

上面的内部类SpringShiroFilter)都会创建一个新的Subject，具体分析里面的代码可以发现，这个Subject的数据会从SubjectContext或Session中获取过来。**这意味着每次经过Shiro过滤器的HTTP请求，都会创建一次新的Subject.**

Suject里面的数据，主要是从SubjectContext中获取，但是获取方式不一样，如SecurityManager总是从SubjectContext中直接获取，而其他数据则主要从Session中获取。只有在登录操作的时候数据会都从SubjectContext上下文中获取。因为登录成功后还会有一个绑定操作，它会把当前用户的相关信息写入Session中去。

DefaultSecurityManager代码如下：

[html] view plaincopyprint?

```
protected void bind(Subject subject) {
        // TODO consider refactoring to use Subject.Binder.
        // This implementation was copied from
SessionSubjectBinder that was removed
        PrincipalCollection principals =
subject.getPrincipals();
        if (principals != null && !principals.isEmpty()) {
            Session session = subject.getSession();
            bindPrincipalsToSession(principals, session);
        } else {
            Session session = subject.getSession(false);
             if (session != null) {
session.removeAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_
KEY);
             }
        }

        if (subject.isAuthenticated()) {
            Session session = subject.getSession();
session.setAttribute(DefaultSubjectContext.AUTHENTICATED_SESSION_
KEY, subject.isAuthenticated());
        } else {
            Session session = subject.getSession(false);
            if (session != null) {
session.removeAttribute(DefaultSubjectContext.AUTHENTICATED_SESSI
ON_KEY);
            }
        }
    }
```

```
[html] view plaincopyprint?
        private void bindPrincipalsToSession(PrincipalCollection
    principals, Session session) throws IllegalArgumentException {
            if (session == null) {
                throw new IllegalArgumentException("Session
    argument cannot be null.");
            }
            if (CollectionUtils.isEmpty(principals)) {
                throw new IllegalArgumentException("Principals
    cannot be null or empty.");
            }

    session.setAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY
    , principals);
        }
```

其他登录相关的信息绑定到SubjectContext的操作代码如下，每个set方法的调用都将数据保存到SubjectContext：

```
[html] view plaincopyprint?
        protected Subject createSubject(AuthenticationToken token,
    AuthenticationInfo info, Subject existing) {
            SubjectContext context = createSubjectContext();
            context.setAuthenticated(true);
            context.setAuthenticationToken(token);
            context.setAuthenticationInfo(info);
            if (existing != null) {
                context.setSubject(existing);
            }
            return createSubject(context);
        }
```

6. AdviceFilter：OncePerRequestFilter的第二个子类

它是全部的验证与授权Filter的父类，其doFilterInternal方法承担此类过滤器的核心逻辑。

```
[java] view plaincopyprint?
1.  public void doFilterInternal(ServletRequest request,
    ServletResponse response, FilterChain chain)
            throws ServletException, IOException {

            Exception exception = null;

            try {

                boolean continueChain = preHandle(request,
```

```
response);
            if (log.isTraceEnabled()) {
                log.trace("Invoked preHandle method.
Continuing chain?: [" + continueChain + "]");
            }

            if (continueChain) {
                executeChain(request, response, chain);
            }

            postHandle(request, response);
            if (log.isTraceEnabled()) {
                log.trace("Successfully invoked postHandle
method");
            }

        } catch (Exception e) {
            exception = e;
        } finally {
            cleanup(request, response, exception);
        }
    }
```

从上面的代码可以看出，其核心的逻辑是3个部分: preHandle, executeChain,postHandle。后2者都只有该类中有唯一的实现，子类并不覆盖，而preHandle则由一个子类PathMatchingFilter中覆盖，代码如下：

[java] view plaincopyprint?

```
1.  public boolean preHandle(ServletRequest request,
ServletResponse response) throws Exception {

        if (this.appliedPaths == null ||
this.appliedPaths.isEmpty()) {
            if (log.isTraceEnabled()) {
                log.trace("appliedPaths property is null or
empty.  This Filter will passthrough immediately.");
            }
            return true;
        }

        for (String path : this.appliedPaths.keySet()) {
            // If the path does match, then pass on to the
subclass implementation for specific checks
            //(first match 'wins'):
            if (pathsMatch(path, request)) {
                if (log.isTraceEnabled()) {
```

```
                    log.trace("Current requestURI matches
pattern [" + path + "].  Performing onPreHandle check...");
                }
                Object config = this.appliedPaths.get(path);
                return onPreHandle(request, response,
config);
            }
        }

        //no path matched, allow the request to go through:
        return true;
    }
```

这个方法根据用户请求的地址是否与该Filter配置的地址匹配来决定是否调用内部的onPreHandler方法。从shiroFilter中的属性filterChainDefinitions配置中可以看出，shiro默认的那些过滤器如user,roles,perms等等都可以统一使用这种方式，对于内部的处理则分别由各个Filter的onPreHandler(其实是由内部的isAccessAllowed和onAccessDenied方法)来决定了。

举2个例子

第一个是AuthenticationFilter的isAccessAllowed方法，它只检测用户是否通过验证

[java] view plaincopyprint?

```
1. protected boolean isAccessAllowed(ServletRequest request,
ServletResponse response, Object mappedValue) {
        Subject subject = getSubject(request, response);
        return subject.isAuthenticated();
    }
```

第二个是RolesAuthorizationFilter的isAccessAllowed方法，它检测用户的角色是否满足

[java] view plaincopyprint?

```
1. public boolean isAccessAllowed(ServletRequest request,
ServletResponse response, Object mappedValue) throws IOException
{

        Subject subject = getSubject(request, response);
        String[] rolesArray = (String[]) mappedValue;

        if (rolesArray == null || rolesArray.length == 0) {
            //no roles specified, so nothing to check - allow
access.
            return true;
```

```
        }

        Set<String> roles = CollectionUtils.asSet(rolesArray);
        return subject.hasAllRoles(roles);
    }
```