1. 前言

在使用spring security开发的过程中，获取当前登录用户时常常会用到这样的写法：

```
UserDetails userDetails = (UserDetails)
SecurityContextHolder.getContext().getAuthentication();
```

在多线程环境下，也总是能拿到想要的结果。好奇spring security是如何做到的，通过源码分析下

2. 过滤器链

之前对spring security的过滤器链有一个大概的认识，在配置web.xml时，我们如果要使用spring security需要在web.xml中写入这样一段：

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

这个意味着向`tomcat`容器注册一个`Filter`，它会过滤`/*`所有的请求。`DelegatingFilterProxy`是一个代理类，真正的过滤操作是由`FilterChainProxy`来中的内部类`VirtualFilterChain`来完成的。其中注册了一定数量的`Filter`（默认是12个），来达到对请求的权限管理操作。具体代码：

```
public void doFilter(ServletRequest request, ServletResponse response)
                throws IOException, ServletException {
    //如果每个过滤器都已通过，会转回tomcat中ApplicationFilterChain
    if (currentPosition == size) {
        if (logger.isDebugEnabled()) {
            logger.debug(UrlUtils.buildRequestUrl(firewalledRequest)
                    + " reached end of additional filter chain; proceeding with
original chain");
        }

        // Deactivate path stripping as we exit the security filter chain
        this.firewalledRequest.reset();

        originalChain.doFilter(request, response);
```

```
    }
    else {
        //按顺序取出过滤器
        currentPosition++;

        Filter nextFilter = additionalFilters.get(currentPosition - 1);

        if (logger.isDebugEnabled()) {
            logger.debug(UrlUtils.buildRequestUrl(firewalledRequest)
                    + " at position " + currentPosition + " of " + size
                    + " in additional filter chain; firing Filter: '"
                    + nextFilter.getClass().getSimpleName() + "'");
        }

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

默认12个`Filter`如下：

- `WebAsyncManagerIntegrationFilter`：提供了对`securityContext`和`WebAsyncManager`的集成，其会把`SecurityContext`设置到异步线程中，使其也能获取到用户上下文认证信息
- `SecurityContextPersistenceFilter`：会根据策略获取一个`SecurityContext`放到`SecurityContextHolder`中，并且在请求结束后清空
- `HeaderWriterFilter`：会往该请求的`Header`中添加相应的信息，在`http`标签内部使用`security:headers`来控制
- `LogoutFilter`：匹配`URL`，默认为`/logout`，匹配成功后则用户退出，清除认证信息，如果有自己的退出逻辑，这个过滤器可以`disable`
- `UsernamePasswordAuthenticationFilter`：登录认证过滤器，根据用户名密码进行认证
- `ConcurrentSessionFilter`：`session`同步过滤器，主要有两个功能，一是会刷新当前`session`的最后访问时间，二是判断当前`session`是否失效，失效了的话会做退出操作并触发相应事件。
- `RequestCacheAwareFilter`：重新恢复被打断的请求
- `SecurityContextHolderAwareRequestFilter`：将`request`包装成`HttpServletRequest`
- `AnonymousAuthenticationFilter`：判断`SecurityContext`中是否有一个`Authentication`对象，如果没有创建一个新的（`AnonymousAuthenticationToken`）
- `SessionManagementFilter`：检查`session`在`spring security`中是否是失效了（注意不是在`web`容器中），比如说配置设置了最大`session`数量为1，那么之前的`session`会被设置`expired = true`
- `ExceptionTranslationFilter`：处理`AccessDeniedException`和`AuthenticationException`，为`java exceptions`和`HTTP responses`提供了桥梁
- `FilterSecurityInterceptor`：对`http`资源做权限拦截，我们平时设置的不同角色不同权限访问就是借此`Filter`过滤

3. ThreadLocal

在进入`SecurityContextPersistenceFilter`之前，先再熟悉下`ThreadLocal`这个类，`spring`中许多地方都用到了`ThreadLocal`

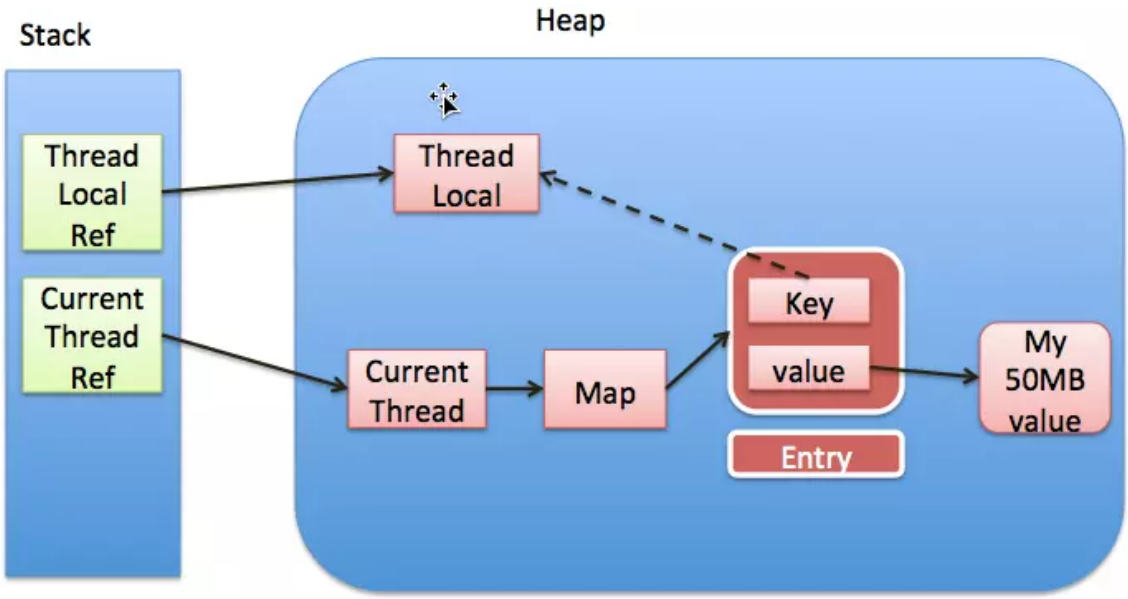
先看一下它的百科：

> JDK 1.2的版本中就提供java.lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序，ThreadLocal并不是一个Thread，而是Thread的局部变量。

其中说的很清楚，`ThreadLocal`并不是一个`Thread`，而是`Thread`的局部变量，我想这也是大部分人将其翻译为“本地线程变量”的原因。

它的数据结构：

每个`Thread`会维护一个本地变量`ThreadLocalMap`，它是`HashMap`的另一种实现，key是`ThreadLocal`变量本身，`value`是要存储的值。`ThreadLocal`本事并不存储任何值，它只是充当了`key`的作用。如下图：



4. SecurityContextPersistenceFilter

最关键的步骤就是`SecurityContextPersistenceFilter`这个过滤器了。其中关键的两个类：

SecurityContextRepository

顾名思义，是存储`SecurityContext`的仓库，默认实现是`HttpSessionSecurityContextRepository`，基于`session`，将`SecurityContext`用key="SPRING_SECURITY_CONTEXT"存入`session`中。

```
Object contextFromSession = httpSession.getAttribute(springSecurityContextKey);
```

SecurityContextHolder

在请求之间保存`SecurityContext`，提供了一系列的静态方法。使用了策略设计模式，默认使用的策略是`ThreadLocalSecurityContextHolderStrategy`，这其中便是使用`ThreadLocal`进行存储的。

该过滤器会先从`SecurityContextRepository`中获取`SecurityContext`，将其设置到`SecurityContextHolder`中，之后会转到下一个过滤器。在请求结束之后，清空`SecurityContextHolder`，并将请求后的`SecurityContext`在保存到`SecurityContextRepository`中。 对应源码：

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

```
        //省略若干语句...

    HttpRequestResponseHolder holder = new HttpRequestResponseHolder(request,
            response);
    //从session中根据key取出SecurityContext，如果没有会创建一个新的
    SecurityContext contextBeforeChainExecution = repo.loadContext(holder);

    try {
            //设置到ThreadLoacal中
        SecurityContextHolder.setContext(contextBeforeChainExecution);

        chain.doFilter(holder.getRequest(), holder.getResponse());

    }
    finally {
        SecurityContext contextAfterChainExecution = SecurityContextHolder
                .getContext();
        // Crucial removal of SecurityContextHolder contents - do this before
anything
        // else.
        SecurityContextHolder.clearContext();
        repo.saveContext(contextAfterChainExecution, holder.getRequest(),
                holder.getResponse());
        request.removeAttribute(FILTER_APPLIED);

        if (debug) {
            logger.debug("SecurityContextHolder now cleared, as request
processing completed");
        }
    }
}
```

注意：这个`finally`语句块是在`reques t`经过`Filter`，到达`DispatcherServlet`完成业务处理之后才会运行的。所以我们可以在`controller`中直接使用文章开头的方式获取到当前登录用户。

5. 总结

最后再来梳理一遍流程：

浏览器发起一个`Http`请求到达`web`容器，比如`Tomcat`，`Tomcat`将其封装成一个`Request`，先经过`Filter`，其中经过`spring security`的`SecurityContextPersistenceFilter`，从`session`中取出`SecurityContext`（如果没有就创建新的）存入当前线程的`ThreadLocalMap`中，因为是当前线程，所以不同的线程之间根本互不影响。之后完成`Servlet`调用，执行`finally`语句块，清除当前线程中`ThreadLocalMap`对应的`SecurityContext`，再将其覆盖`session`中之前的部分。

这里多说一句，为什么在每次请求之后要清空当前线程呢？看一下`spring`的官方`api`说明：

> In an application which receives concurrent requests in a single session, the same SecurityContext instance will be shared between threads. Even though a ThreadLocal is being used, it is the same instance that is retrieved from the HttpSession for each thread. This has implications if you wish to temporarily change the context under which a thread is running. If you just use SecurityContextHolder.getContext(), and call setAuthentication(anAuthentication) on the returned context object, then the Authentication object will change in all concurrent threads which share the same SecurityContext instance. You can customize the behaviour of SecurityContextPersistenceFilter to create a completely new SecurityContext for each

> request, preventing changes in one thread from affecting another. Alternatively you can create a new instance just at the point where you temporarily change the context. The method SecurityContextHolder.createEmptyContext() always returns a new context instance.

简而言之，因为`SecurityContext`是放在`session`中的，所有一个`session`下的`request`都是共享一个`SecurityContext`，也就是会有多个`Thread`共享一个`SecurityContext`。如果我们在某一个线程中只是想临时对`SecurityContext`做点更改，那么其他线程中`SecurityContext`也会受到影响，这是不被允许的。

Author：NaraLuwan