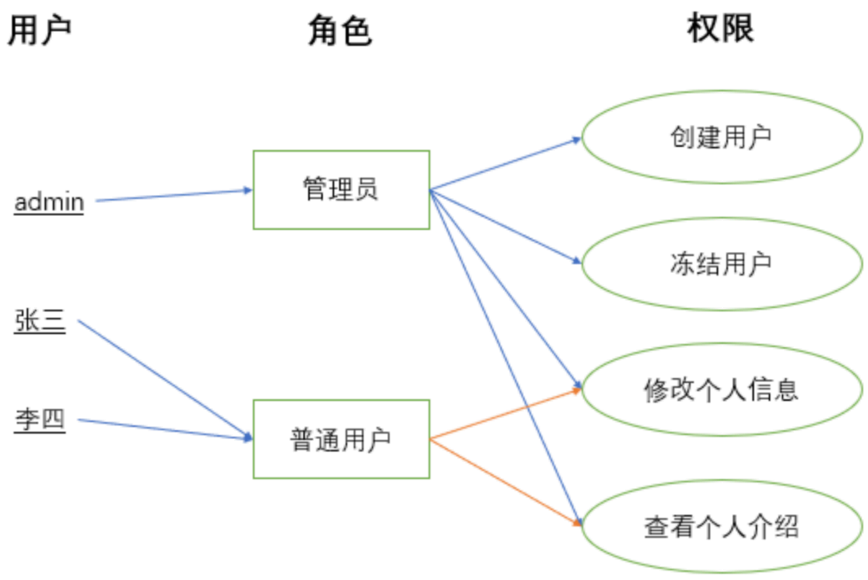
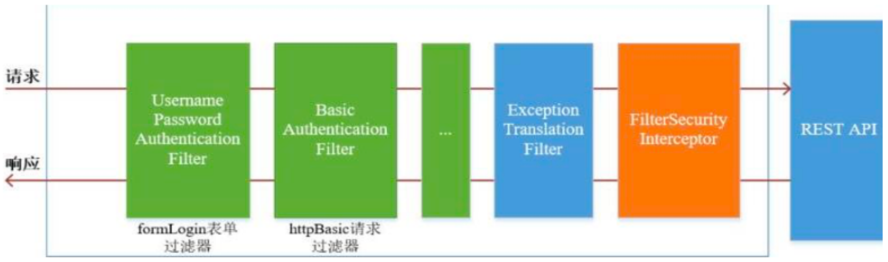
1.SpringSercurity

Spring Security是一套安全框架，可以基于RBAC（基于角色的权限控制）对用户的访问权限进行控制，核心思想是通过一系列的filter chain来进行拦截过滤
用户授予角色，角色授予访问权限



1.1 SpringSercurity基本流程

Spring Security 采取过滤链实现认证与授权，只有当前过滤器通过，才能进入下一个过滤器:



请求
响应
Username Password Authentication Filter
Basic Authentication Filter
...
Exception Translation Filter
FilterSecurity Interceptor
REST API
formLogin表单过滤器
httpBasic请求过滤器

1.2 认证与授权

SpringSercurity的核心功能是用户认证和用户授权。

(1)用户认证指的是:系统通过校验用户名和密码来完成认 证过程。通俗点说就是系统认为用户是否能登录

(2)用户授权指的是:系统会为不同的用户分配不同的角色，而每个角色则对应一系列的权限。通俗点讲就是系统判断用户是否有权限去做某些事情。

1.3 SpringSercurity主要过滤链

SpringSercurity本质是个过滤链，常见的过滤链如下

1.FilterSecurityInterceptor 过滤器:该过滤器是过滤器链的最后一个过滤器，根据资源权限配置来判断当前请求是否有权限访问对应的资源。如果访问受限会抛出相关异常，并由 ExceptionTranslationFilter 过滤器进行捕获和处理。

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

2.ExceptionTranslationFilter 过滤器:该过滤器不需要我们配置，对于前端提交的请求会直接放行，捕获后续抛出的异常并进行处理(例如:权限访问限制)。

```
//异常过滤器，用于处理认证授权过程中抛出去的异常
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter
```

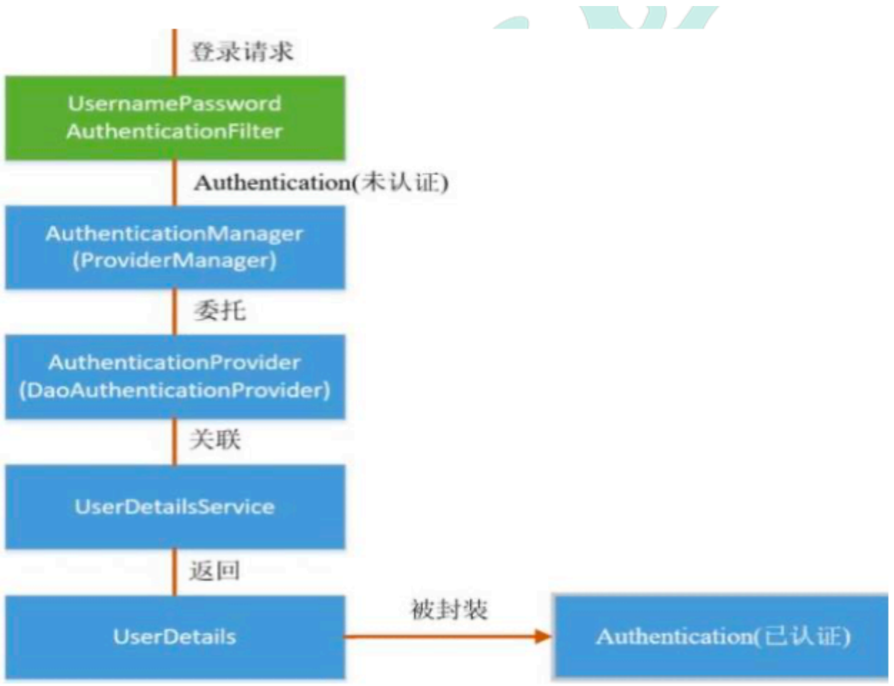
3.UsernamePasswordAuthenticationFilter 过滤器:该过滤器会拦截前端提交的 POST 方式的登录表单请求，并进行身份认证。

```
//对login的post请求做拦截，校验表单的用户名和密码
```

```
org.springframework.security.web.access.ExceptionTranslationFilter
```

2. SpringSecurity 认证流程

2.1 总流程


登录请求
UsernamePassword AuthenticationFilter
Authentication(未认证)
AuthenticationManager (ProviderManager)
委托
AuthenticationProvider (DaoAuthenticationProvider)
关联
UserDetailsService
返回
UserDetails
被封装
Authentication(已认证)

当前端提交的是一个 POST 方式的登录表单请求，就会被该过滤器拦截，并进行身份认证。该过滤器的 doFilter() 方法实现在其抽象父类 AbstractAuthenticationProcessingFilter 中，查看相关源码:

2.2 AbstractAuthenticationProcessingFilter

```java
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
         throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest)req;
    HttpServletResponse response = (HttpServletResponse)res;
    if (!this.requiresAuthentication(request, response)) {
        //1 如果不是post请求，则直接放行
        chain.doFilter(request, response);
    } else {
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Request is to process authentication");
         }
 //2 Authentication用来存储用户认证信息的类
         Authentication authResult;
         try {
             //3 调用子类UsernamePasswordAuthenticationFilter的attemptAuthentication方法
             authResult = this.attemptAuthentication(request, response);
             if (authResult == null) {
                 return;
             }
 //4 session策略处理（如果用户设置session最大并发数，在此判断）
             this.sessionStrategy.onAuthentication(authResult, request, response);
         } catch (InternalAuthenticationServiceException var8) {
             this.logger.error("An internal error occurred while trying to authenticate th
             //5 认证失败调用认证失败处理器
             this.unsuccessfulAuthentication(request, response, var8);
             return;
         } catch (AuthenticationException var9) {
             this.unsuccessfulAuthentication(request, response, var9);
             return;
         }
 //6 continueChainBeforeSuccessfulAuthentication默认为false, 不会进入下一个过滤器
         if (this.continueChainBeforeSuccessfulAuthentication) {
             chain.doFilter(request, response);
         }
 //7 认证成功调用认证成功过滤器
         this.successfulAuthentication(request, response, chain, authResult);
     }
 }
```

2.3 UsernamePasswordAuthenticationFilter

```java
public class UsernamePasswordAuthenticationFilter extends AbstractAuthenticationProcessing
    public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "username";
    public static final String SPRING_SECURITY_FORM_PASSWORD_KEY = "password";
    private String usernameParameter = "username";//默认表单用户名参数
    private String passwordParameter = "password";//默认表单密码参数
    private boolean postOnly = true;//默认请求方式只能是POST

    public UsernamePasswordAuthenticationFilter() {
        //默认登陆表单提交的地址 /login POST
        super(new AntPathRequestMatcher("/login", "POST"));
    }

     public Authentication attemptAuthentication(HttpServletRequest request,
                HttpServletResponse response) throws AuthenticationException {
        if (this.postOnly && !request.getMethod().equals("POST")) {
            //不是POST请求, 抛出异常
            throw new AuthenticationServiceException
            ("Authentication method not supported: " + request.getMethod());
        } else {
            //获取请求携带的username和password
            String username = this.obtainUsername(request);
            String password = this.obtainPassword(request);
            if (username == null) {
                username = "";
            }

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

            username = username.trim();
//使用表单传入的username和password构造Authentication对象，标记为未登陆
//UsernamePasswordAuthenticationToken extends AbstractAuthenticationToken
//AbstractAuthenticationToken implements Authentication, CredentialsContainer
            UsernamePasswordAuthenticationToken authRequest =
            new UsernamePasswordAuthenticationToken(username, password);
//将请求中的一些属性设置到Authentication对象中，如sessionId、remoteAddress
            this.setDetails(request, authRequest);
//调用ProviderManager类的authenticate进行身份验证
//ProviderManager implements AuthenticationManager
            return this.getAuthenticationManager().authenticate(authRequest);
        }
    }
```

2.4 UsernamePasswordAuthenticationToken

```
//封装未认证的用户信息
public UsernamePasswordAuthenticationToken(Object principal, Object credentials) {
    super((Collection)null);
    this.principal = principal;
    this.credentials = credentials;
    this.setAuthenticated(false);//标记为未认证
}
//封装已认证的用户信息
public UsernamePasswordAuthenticationToken(Object principal, Object credentials,
                                   Collection<? extends GrantedAuthority> authorities) {
     super(authorities);
     this.principal = principal;
     this.credentials = credentials;
     super.setAuthenticated(true);//标记为已认证
 }
```

2.5 Authentication

```
public interface Authentication extends Principal, Serializable {
    //用户权限集合
     Collection<? extends GrantedAuthority> getAuthorities();
    //用户密码
    Object getCredentials();
    //请求携带的一些信息 如sessionId、remoteAddress
    Object getDetails();
    //未认证时为前端请求传入的用户名 认证成功后为认证用户信息的userDetails对象
    Object getPrincipal();
     //是否被认证
     boolean isAuthenticated();
     //设置是否被认证
     void setAuthenticated(boolean var1) throws IllegalArgumentException;
 }
```

2.6 ProviderManager

ProviderManager 是 AuthenticationManager 接口的实现类，该接口是认证相关的核心接口，也是认证的入口。在实际开发中，我们可能有多种不同的认证方式，例如:用户名+ 密码、邮箱+密码、手机号+验证码等，而这些认证方式的入口始终只有一个，那就是 AuthenticationManager。在该接口的常用实现类

ProviderManager 内部会维护一个 List<AuthenticationProvider>列表，存放多种认证方式，实际上这是委托者模式 (Delegate)的应用。每种认证方式对应着一个 AuthenticationProvider， AuthenticationManager 根据认证方式的不同(根据传入的 Authentication 类型判断)委托对应的 AuthenticationProvider 进行用户认证。

```
public Authentication authenticate(Authentication authentication) throws AuthenticationExc
    //1 获取传入Authentication的类型，这里是UsernamePasswordAuthenticationToken
    Class<? extends Authentication> toTest = authentication.getClass();
    AuthenticationException lastException = null;
    AuthenticationException parentException = null;
    Authentication result = null;
    Authentication parentResult = null;
    boolean debug = logger.isDebugEnabled();
    //2 获取认证方式列表 List<AuthenticationProvider>的迭代器
     Iterator var8 = this.getProviders().iterator();

     while(var8.hasNext()) {
         AuthenticationProvider provider = (AuthenticationProvider)var8.next();
 //3 判断当前的AuthenticationProvider的迭代器是否适用UsernamePasswordAuthenticationToken类型
         if (provider.supports(toTest)) {
             if (debug) {
                 logger.debug("Authentication attempt using " + provider.getClass().getName
             }
 //4 成功找到适配当前认证方式的AuthenticationProvider，此处为DaoAuthenticationProvider
             try {
 //5 使用DaoAuthenticationProvider的authenticate认证
 //如果认证成功，返回标记认证成功的Authentication对象
                 result = provider.authenticate(authentication);
                 if (result != null) {
  //6 认证成功之后 将传入的Authentication对象中的details对象拷贝到已认证的Authentication对象
                     this.copyDetails(authentication, result);
                     break;
                 }
             } catch (InternalAuthenticationServiceException | AccountStatusException var1
                 this.prepareException(var13, authentication);
                 throw var13;
             } catch (AuthenticationException var14) {
                 lastException = var14;
             }
         }
     }

     if (result == null && this.parent != null) {
         try {
  //7. 认证失败 使用父类型AuthenticationManager进行验证
             result = parentResult = this.parent.authenticate(authentication);
         } catch (ProviderNotFoundException var11) {
         } catch (AuthenticationException var12) {
             parentException = var12;
```

```
                lastException = var12;
            }
        }

        if (result != null) {
    //8 认证成功之后 去除result的敏感信息，要求相关类实现CredentialsContainer接口
            if (this.eraseCredentialsAfterAuthentication && result instanceof CredentialsConte
               //去掉敏感信息: CredentialsContainer接口的eraseCredentials
                 ((CredentialsContainer)result).eraseCredentials();
            }
    //9 发布认证成功事件
            if (parentResult == null) {
                this.eventPublisher.publishAuthenticationSuccess(result);
            }

            return result;
        } else {
    //10 验证失败抛出异常信息
            if (lastException == null) {
                lastException = new ProviderNotFoundException(this.messages.getMessage("Provi
            }

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

            throw lastException;
        }
    }
```

2.7 AbstractAuthenticationToken

```
//去掉敏感信息
public void eraseCredentials() {
    //密码置为null
    this.eraseSecret(this.getCredentials());
    //Principal在已认证的Authentication是UserDetails实现类,
    //如果该实现类想要去除敏感信息，需要实现CredentialsContainer接口的eraseCredentials方法
    this.eraseSecret(this.getPrincipal());
    this.eraseSecret(this.details);
}
```

2.8 AbstractAuthenticationProcessingFilter

```
//认证成功之后的处理
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse re
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Authentication success. Updating SecurityContextHolder to conta
    }

    SecurityContextHolder.getContext().setAuthentication(authResult);
//rememberMe处理
```

```java
    this.rememberMeServices.loginSuccess(request, response, authResult);
     if (this.eventPublisher != null) {
         //发布认证成功事件
         this.eventPublisher.publishEvent(new InteractiveAuthenticationSuccessEvent(authRe
     }
     //调用认证成功处理器
     this.successHandler.onAuthenticationSuccess(request, response, authResult);
 }
 //认证失败之后的处理
 protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse
     //清除该线程在SecurityContextHolder中对应的对象SecurityContext
     SecurityContextHolder.clearContext();
     if (this.logger.isDebugEnabled()) {
         this.logger.debug("Authentication request failed: " + failed.toString(), failed);
         this.logger.debug("Updated SecurityContextHolder to contain null Authentication")
         this.logger.debug("Delegating to authentication failure handler " + this.failureH
     }
 //rememberMe处理
     this.rememberMeServices.loginFail(request, response);
     //调用认证失败处理器
     this.failureHandler.onAuthenticationFailure(request, response, failed);
 }
```

2.9 MyUserDetailsService

实现UserDetailsService接口，UserDetailsService接口为框架的接口

```java
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private UsersMapper usersMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExceptio
        //调用userMapper的方法，根据用户名查询数据库
        QueryWrapper<Users>  wrapper  = new  QueryWrapper();
         wrapper.eq("username",username);
         Users user = usersMapper.selectOne(wrapper);
         if (user == null){
             throw new UsernameNotFoundException("用户找不到");
         }
         List<GrantedAuthority> grantedAuthorities = AuthorityUtils.commaSeparatedStringTo
         return new User(username,new BCryptPasswordEncoder().encode(user.getPassword()),g
    }
}
```

3 授权流程

3.1 ExceptionTranslationFilter

该过滤器是用于处理异常的，不需要我们配置，对于前端提交的请求会直接放行，捕获后续抛出的异常并进行处理(例如:权限访问限制)。具体源码如下:

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

    try {
        //前端请求直接放行
        chain.doFilter(request, response);
        this.logger.debug("Chain processed normally");
    } catch (IOException var9) {
         throw var9;
     } catch (Exception var10) {
         //捕获出现的异常进行处理
         Throwable[] causeChain = this.throwableAnalyzer.determineCauseChain(var10);
         RuntimeException ase = (AuthenticationException)this.throwableAnalyzer.getFirstTh
         if (ase == null) {
             //访问受限的资源抛出的异常
             ase = (AccessDeniedException)this.throwableAnalyzer.getFirstThrowableOfType(A
         }
         ......
     }

 }
```

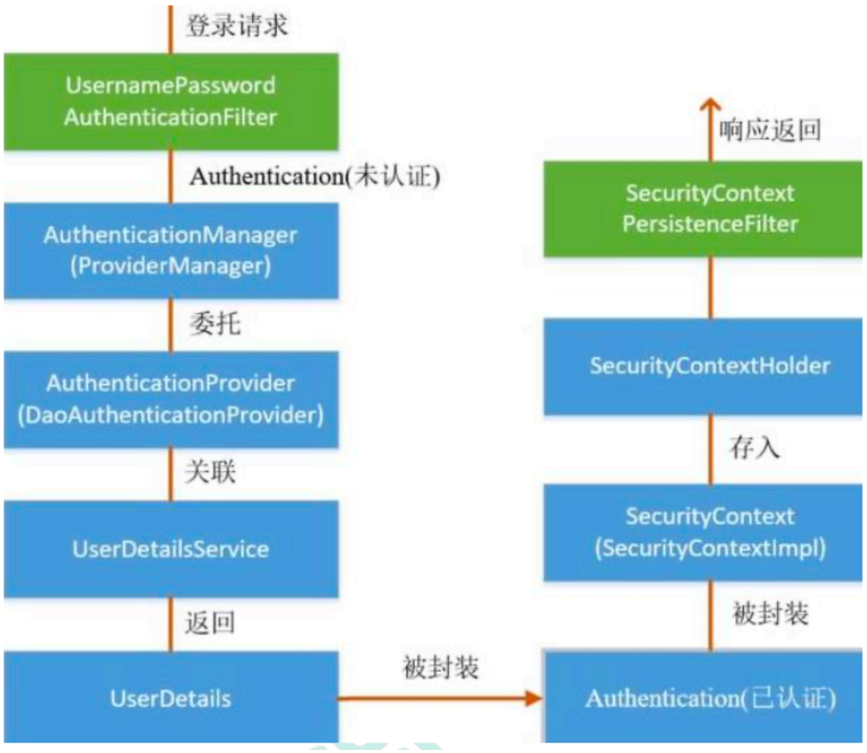
3.2 FilterSecurityInterceptor

FilterSecurityInterceptor 是过滤器链的最后一个过滤器，根据资源权限配置来判断当前请求是否有权限访问对应的资源。如果访问受限会抛出相关异常，最终所抛出的异常会由前一个过滤器 ExceptionTranslationFilter 进行捕获和处理。具体源码如下:

```java
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    FilterInvocation fi = new FilterInvocation(request, response, chain);
    this.invoke(fi);
}
public void invoke(FilterInvocation fi) throws IOException, ServletException {
    if (fi.getRequest() != null && fi.getRequest().getAttribute("__spring_security_filterS
        fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
    } else {
        if (fi.getRequest() != null && this.observeOncePerRequest) {
            fi.getRequest().setAttribute("__spring_security_filterSecurityInterceptor_fil
        }
 //根据资源权限设置来判断当前请求是否有权限访问对象资源, 如果不能访问 则抛出对应异常
         InterceptorStatusToken token = super.beforeInvocation(fi);

         try {
             //访问相关资源, 通过springmvc的核心组件DispatcherServlet 进行访问
             fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
         } finally {
             super.finallyInvocation(token);
         }

         super.afterInvocation(token, (Object)null);
    }

}
```
需要注意，Spring Security 的过滤器链是配置在 SpringMVC 的核心组件 DispatcherServlet 运行之前。也就是说，请求通过 Spring Security 的所有过滤器， 不意味着能够正常访问资源，该请求还需要通过 SpringMVC 的拦截器链。

4 SpringSecurity 请求间共享认证信息

一般认证成功后的用户信息是通过 Session 在多个请求之间共享，那么 Spring Security 中是如何实现将已认证的用户信息对象 Authentication 与 Session 绑定的进行具体分析。


登录请求
UsernamePassword AuthenticationFilter
Authentication(未认证)
AuthenticationManager (ProviderManager)
委托
AuthenticationProvider (DaoAuthenticationProvider)
关联
UserDetailsService
返回
UserDetails
被封装
Authentication(已认证)
被封装
SecurityContext (SecurityContextImpl)
存入
SecurityContextHolder
SecurityContext PersistenceFilter
响应返回

4.1 认证成功

```
SecurityContextHolder.getContext().setAuthentication(authResult);
```

4.2 SecurityContextPersistenceFilter

前面提到过，在 UsernamePasswordAuthenticationFilter 过滤器认证成功之后，会在认证成功的处理方法中将已认证的用户信息对象 Authentication 封装进 SecurityContext，并存入 SecurityContextHolder。

之后，响应会通过 SecurityContextPersistenceFilter 过滤器，该过滤器的位置在所有过滤器的最前面，请求到来先进它，响应返回最后一个通过它，所以在该过滤器中处理已认证的用户信息对象 Authentication 与 Session 绑定。

认证成功的响应通过 SecurityContextPersistenceFilter 过滤器时，会从 SecurityContextHolder 中取出封装了已认证用户信息对象 Authentication 的 SecurityContext，放进 Session 中。当请求再次到来时，请求首先经过该过滤器，该过滤器会判断当前请求的 Session 是否存有 SecurityContext 对象，如果有则将该对象取出再次 放入 SecurityContextHolder 中，之后该请求所在的线程获得认证用户信息，后续的资源访问不需要进行身份认证;当响应再次返回时，该过滤器同样从 SecurityContextHolder 取出 SecurityContext 对象，放入 Session 中。具体源码如下:

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
       throws IOException, ServletException {
       ...
       //请求到来时，检查当前session中是否存有SecurityContext对象
       //如果有则取出来，如果没有，创建一个空的SecurityContext对象
        HttpRequestResponseHolder holder = new HttpRequestResponseHolder(request, response
        SecurityContext contextBeforeChainExecution = this.repo.loadContext(holder);
        boolean var13 = false;

         try {
             var13 = true;
             //将上面的SecurityContext放在SecurityContextHolder中
             SecurityContextHolder.setContext(contextBeforeChainExecution);
             //进入下一个过滤器
             chain.doFilter(holder.getRequest(), holder.getResponse());
             var13 = false;
         } finally {
             if (var13) {
                 SecurityContext contextAfterChainExecution = SecurityContextHolder.getCon
                 SecurityContextHolder.clearContext();
                 this.repo.saveContext(contextAfterChainExecution, holder.getRequest(), ho
                 request.removeAttribute("__spring_security_scpf_applied");
                 if (debug) {
                     this.logger.debug("SecurityContextHolder now cleared, as request proc
                 }

             }
         }
     //响应返回时，从SecurityContextHolder取出SecurityContext
         SecurityContext contextAfterChainExecution = SecurityContextHolder.getContext();
     //移除SecurityContextHolder的ecurityContext对象
         SecurityContextHolder.clearContext();
      //放入session中
            this.repo.saveContext(contextAfterChainExecution, holder.getRequest(), holder.
         request.removeAttribute("__spring_security_scpf_applied");
         if (debug) {
             this.logger.debug("SecurityContextHolder now cleared, as request processing c
         }

     }
 }
```

5 过滤器如何加载?

1. 使用spring serciurity配置过滤器DelegatingFilterProxy,获得多个过滤器

springboot项目无需配置

```
public class DelegatingFilterProxy extends GenericFilterBean {
     public void doFilter(ServletRequest request, ServletResponse response,
           FilterChain filterChain) throws ServletException, IOException {
            ....
            delegateToUse = this.initDelegate(wac);
            .....
        }
        protected Filter initDelegate(WebApplicationContext wac) throws ServletException {
            ....
            //targetBeanName:FilterChainProxy
            Filter delegate = (Filter)wac.getBean(targetBeanName, Filter.class);
            ....
         }
}
public class FilterChainProxy extends GenericFilterBean {
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
                ....
            this.doFilterInternal(request, response, chain);
                ....
        }

        private void doFilterInternal(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
                ....

            List<Filter> filters = this.getFilters((HttpServletRequest)firewallRequest);
                    ....
        }
    //加载所有的过滤器链
        private List<Filter> getFilters(HttpServletRequest request) {
            Iterator var3 = this.filterChains.iterator();

            do {
                //SecurityFilterChain:过滤器链
                chain = (SecurityFilterChain)var3.next();
                if (logger.isTraceEnabled()) {
                    ++count;
                    logger.trace(LogMessage.format("Trying to match request against %s (%d/%d
                }
            } while(!chain.matches(request));
                }
}
```