小Hub领读：

权限框架一般都是一堆过滤器、拦截器的组合运用，在shiro中，有多少个内置的过滤器你知道吗？在哪些场景用那些过滤器，这篇文章希望你能对shiro有个新的认识！

别忘了，点个 [**在看**] 支持一下哈~

前两篇原创shiro相关文章：

1、极简入门，Shiro的认证与授权流程解析

2、只需要6个步骤，springboot集成shiro，并完成登录

我们都知道shiro是个认证权限框架，除了登录、退出逻辑我们需要侵入项目代码之外，验证用户是否已经登录、是否拥有权限的代码其实都是过滤器来完成的，可以这么说，shiro其实就是一个过滤器链集合。

那么今天我们详细讨论一下shiro底层到底给我们提供了多少默认的过滤器供我们使用，又都有什么用呢？带着问题，我们先去shiro官网看看对于默认过滤器集的说明。

- http://shiro.apache.org/web.html#default-filters > When running a web-app, Shiro will create some useful default Filter instances and make them available in the [main] section automatically. You can configure them in main as you would any other bean and reference them in your chain definitions. > >The default Filter instances available automatically are defined by the DefaultFilter enum and the enum's name field is the name available for configuration.

翻译过来意思：

当运行web应用程序时，Shiro将创建一些有用的默认过滤器实例，并使它们在[main]部分自动可用。您可以像配置任何其他bean一样在main中配置它们，并在链定义中引用它们。

默认筛选器实例由DefaultFilter enum中定义，enum s name字段是可用于配置的名称。

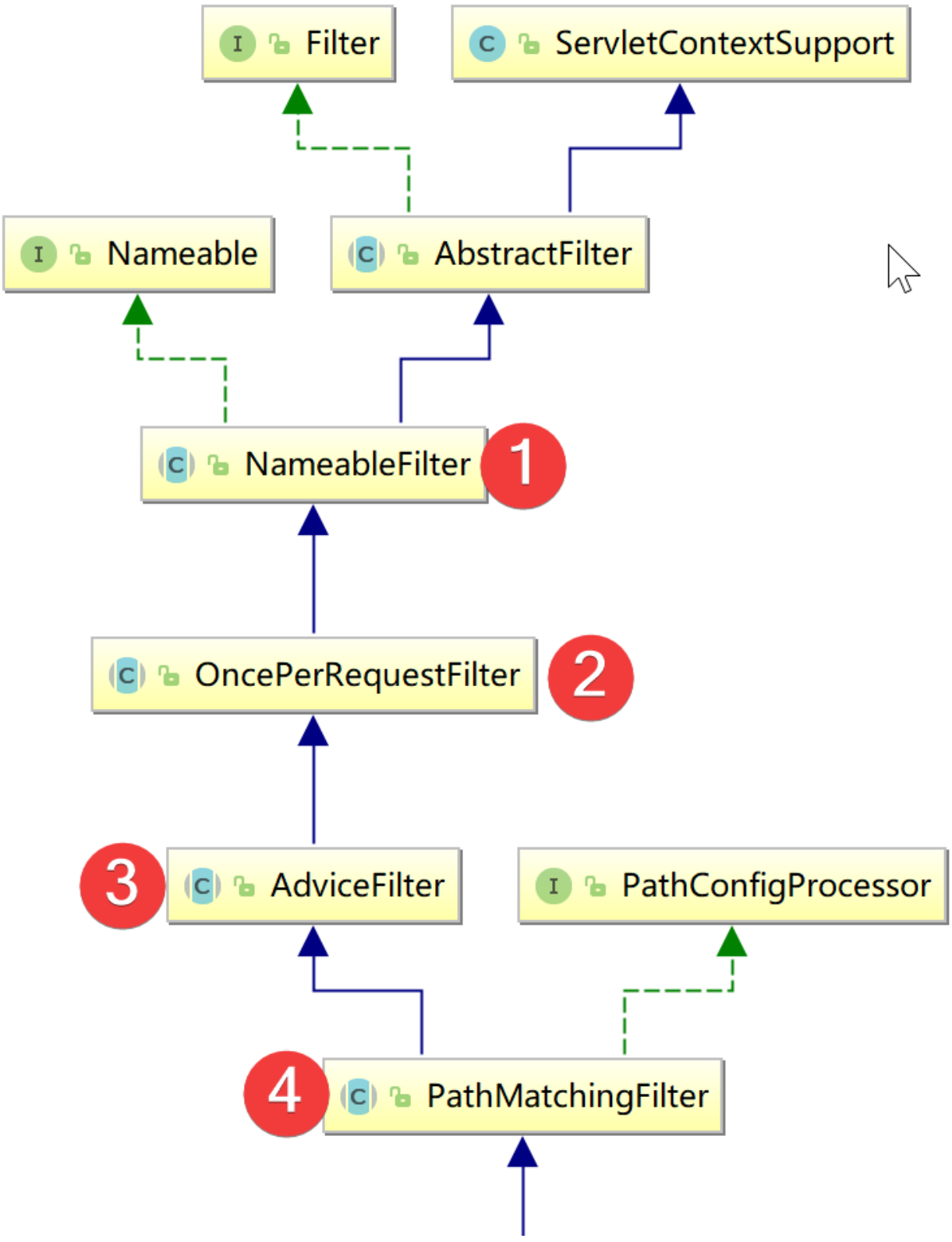
于是我看了一下`DefaultFilter`的源码：

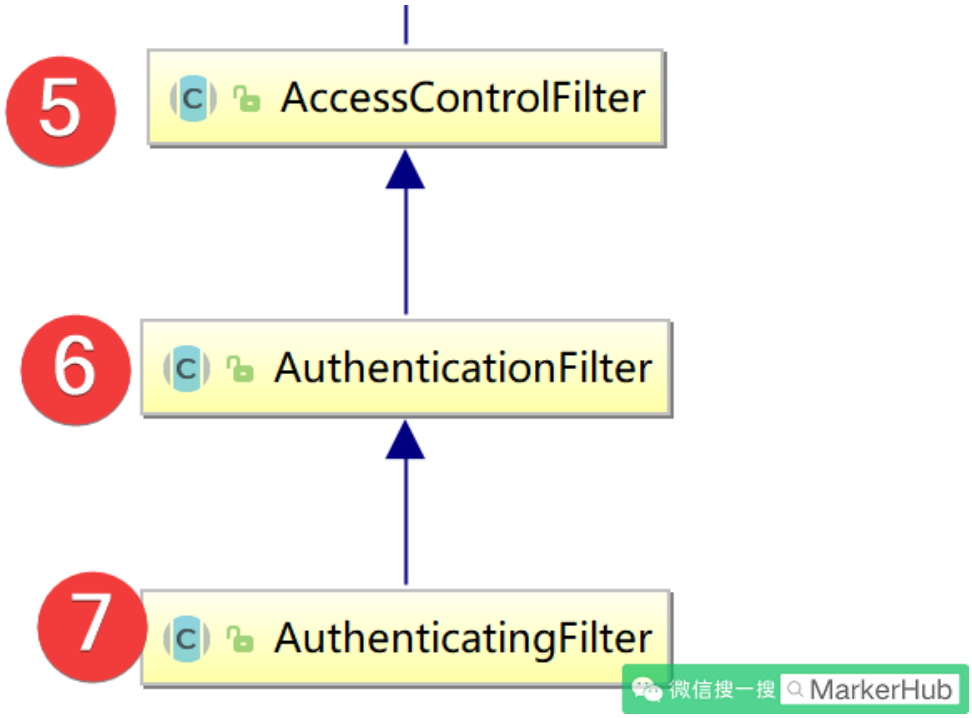
```
public enum DefaultFilter {

    anon(AnonymousFilter.class),
    authc(FormAuthenticationFilter.class),
    authcBasic(BasicHttpAuthenticationFilter.class),
    logout(LogoutFilter.class),
    noSessionCreation(NoSessionCreationFilter.class),
    perms(PermissionsAuthorizationFilter.class),
    port(PortFilter.class),
    rest(HttpMethodPermissionFilter.class),
    roles(RolesAuthorizationFilter.class),
    ssl(SslFilter.class),
```

```
    user(UserFilter.class);

    ...
}
```

终于知道我们常用的anon、authc、perms、roles、user过滤器是哪里来的了！这些过滤器我们都是可以直接使用的。但你要弄清楚这些默认过滤器，你还不得不去深入了解一下shiro更底层为我们提供的过滤器，基本我们的这些默认过滤器都是通过继承这几个底层过滤器演变而来的。

那么这些过滤器都有哪些呢？我们来看一个图：

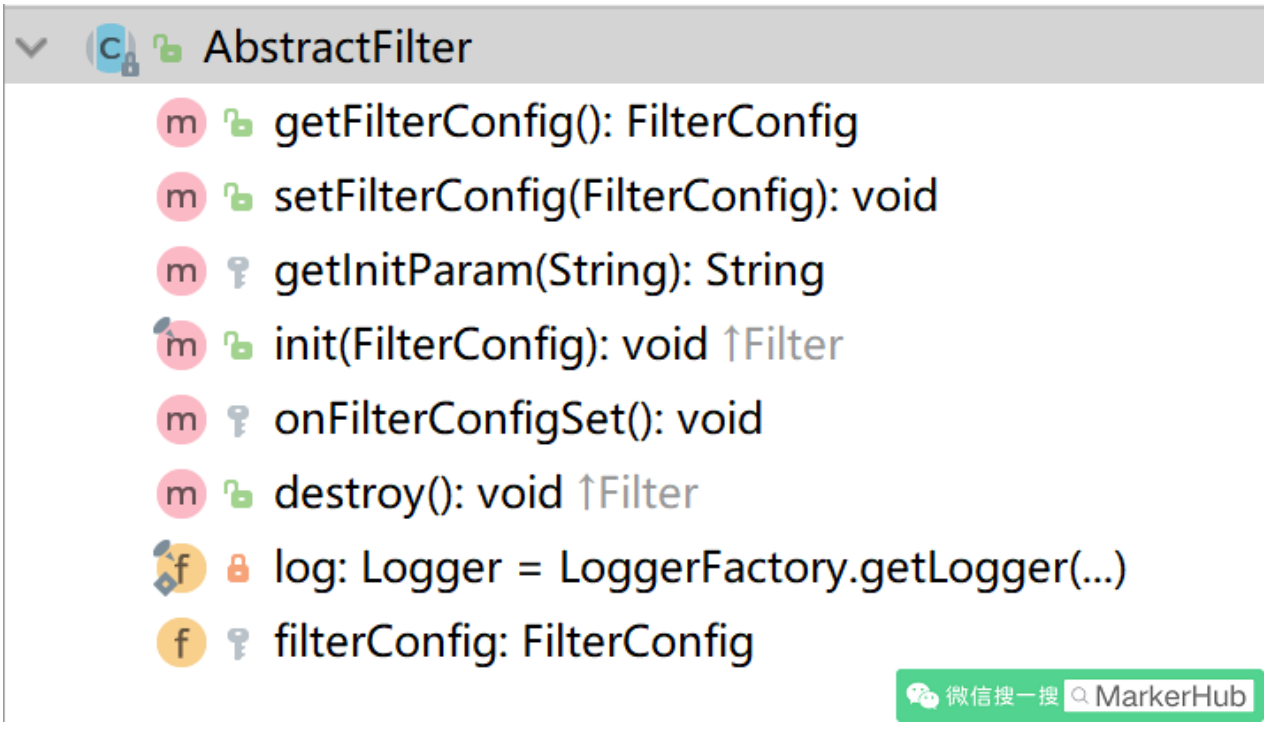




上面我标记了7个我们接下来要介绍的过滤器，我们一个个来介绍，弄清楚这些过滤器之后，相信你对shiro的认识会更深一层了。具体authc、perms、roles等这些默认过滤器与这7个过滤器有什么关系你就会明白。

1、AbstractFilter

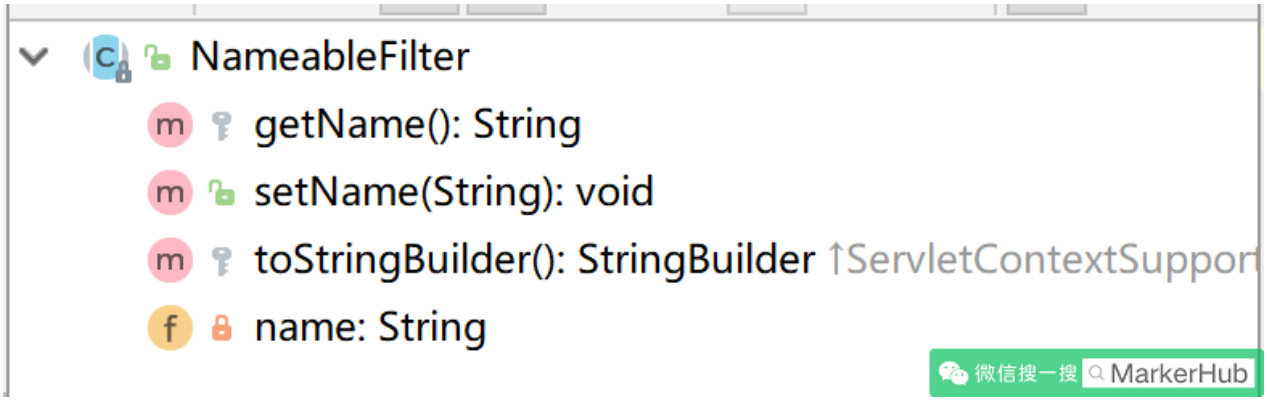
这个过滤器还得说说，shiro最底层的抽象过滤器，虽然我们极少直接继承它，它通过实现`Filter`获得过滤器的特性。



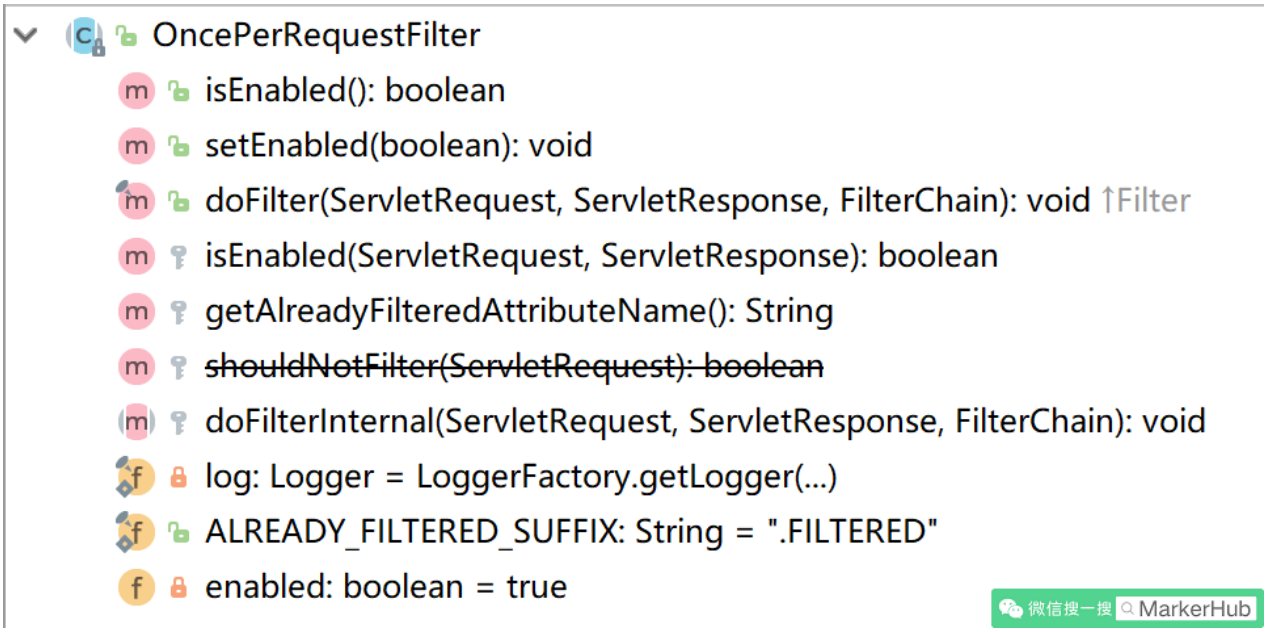
完成一些过滤器基本初始化操作，`FilterConfig`：过滤器配置对象，用于servlet容器在初始化期间将信息传递给其他过滤器。

2、NameableFilter

命名过滤器，给过滤器定义名称！也是比较基层的过滤器了，未拓展其他功能，我们很少会直接继承这个过滤器。为重写doFilter方法。



3、OncePerRequestFilter



重写doFilter方法，保证每个servlet方法只会被过滤一次。可以看到doFilter方法中，第一行代码就是`String alreadyFilteredAttributeName = getAlreadyFilteredAttributeName();`然后通过`request.getAttribute(alreadyFilteredAttributeName) != null`来判断过滤器是否已经被调用过，从而保证过滤器不会被重复调用。

进入方法之前，先标记`alreadyFilteredAttributeName`为True，抽象`doFilterInternal`方法执行之后再remove掉`alreadyFilteredAttributeName`。

```
public final void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
    String alreadyFilteredAttributeName = getAlreadyFilteredAttributeName();
    if ( request.getAttribute(alreadyFilteredAttributeName) != null ) {
        log.trace("Filter '{}' already executed.  Proceeding without invoking this filter.", getName(
        filterChain.doFilter(request, response);
    } else //noinspection deprecation
        if (/* added in 1.2: */ !isEnabled(request, response) ||
        /* retain backwards compatibility: */ shouldNotFilter(request) ) {
        log.debug("Filter '{}' is not enabled for the current request.  Proceeding without invoking t
                getName());
        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);
        }
    }
}
```


所以OncePerRequestFilter过滤器保证只会被一次调用的功能，提供了抽象方法`doFilterInternal`让后面的过滤器可以重写，执行真正的过滤器处理逻辑。

```
protected abstract void doFilterInternal(ServletRequest request,
ServletResponse response, FilterChain chain)
            throws ServletException, IOException;
```

这个过滤器我们已经可以开始在我们的项目继承使用，比如拦截用户请求，判断用户是否已经登录（携带token或cookie信息），如果未登录则返回Json数据告知未登录！

比如： 开源mblog博客项目中，过滤器就是继承OncePerRequestFilter。

- https://gitee.com/mtons/mblog

```
/**
 * 公众号：MarkerHub
**/
public class AuthenticatedFilter extends OncePerRequestFilter {

    // 前端弹窗的js代码
    private static final String JS = "<script type="text/javascript">var
wp=window.parent; if(wp!=null){while(wp.parent&&wp.parent!==wp)
{wp=wp.parent;}wp.location.href='%1$s';}else{window.location.href='%1$s';}
</script>";
    private String loginUrl = "/login";

    // 重写doFilterInternal方法
    @Override
    protected void doFilterInternal(ServletRequest request, ServletResponse
response, FilterChain chain)
            throws ServletException, IOException {
```

```
        Subject subject = SecurityUtils.getSubject();
        // 已经登陆就跳过过滤器
        if (subject.isAuthenticated() || subject.isRemembered()) {
            chain.doFilter(request, response);
        } else {
            // 未登录就返回json或者js代码
            WebUtils.saveRequest(request);
            String path = WebUtils.getContextPath((HttpServletRequest)
request);
            String url = loginUrl;
            if (StringUtils.isNotBlank(path) &amp;&amp; path.length() &gt; 1)
{
                url = path + url;
            }

            if (isAjaxRequest((HttpServletRequest) request)) {
                response.setContentType("application/json;charset=UTF-8");
                response.getWriter().print(JSON.toJSONString(Result.failure("您
还没有登录!")));
            } else {
                response.getWriter().write(new Formatter().format(JS,
url).toString());
            }
        }
    }
}
```

未登录情况，ajax请求过滤器返回`您还没有登录!`提示，web请求则返回一段js代码，前端渲染会跳出一个登陆窗口，这也就是未什么大家常遇到的点击登录，当前跳出一个登陆弹窗的一种实现方式！

效果：

4、AdviceFilter

看到Advice，很自然想到切面环绕编程，一般有pre、post、after几个方法。所以这个AdviceFilter过滤器就是提供了和AOP相似的切面功能。

继承OncePerRequestFilter过滤器重写doFilterInternal方法，我们可以先看看：

```
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);
    }
}
```


可以看到上面4个序号：

- 1、preHandle 前置过滤，默认true
- 2、executeChain 执行真正代码过滤逻辑->chain.doFilter
- 3、postHandle 后置过滤
- 4、cleanup 其实主要逻辑是afterCompletion方法

于是，我们从OncePerRequestFilter的一个doFilterInternal分化成了切面编程，更容易前后控制执行逻辑。所以如果继承AdviceFilter时候，我们可以重写preHandle方法，判断用户是否满足已登录或者其他业务逻辑，返回false时候表示不通过过滤器。

5、PathMatchingFilter

请求路径匹配过滤器，通过匹配请求url，判断请求是否需要过滤，如果url未在需要过滤的集合内，则跳过，否则进入`isFilterChainContinued`的onPreHandle方法。

我们可以看下代码：

```
protected boolean preHandle(ServletRequest request, ServletResponse response) throws

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

    for (String path : this.appliedPaths.keySet()) {
        // If the path does match, then pass on to the subclass implementation for sp
        //(first match 'wins'):
        if (pathsMatch(path, request)) {
            log.trace("Current requestURI matches pattern '{}'.  Determining filter c
            Object config = this.appliedPaths.get(path);
            return isFilterChainContinued(request, response, path, config);
        }
    }

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



```
private boolean isFilterChainContinued(ServletRequest request, ServletResponse response,
                                       String path, Object pathConfig) throws Exception {

    if (isEnabled(request, response, path, pathConfig)) { //isEnabled check added in 1.2
        if (log.isTraceEnabled()) {
            log.trace("Filter '{}' is enabled for the current request under path '{}' with
                    "Delegating to subclass implementation for 'onPreHandle' check.",
                    new Object[]{getName(), path, pathConfig});
        }
        //The filter is enabled for this specific request, so delegate to subclass implemen
        //so they can decide if the request should continue through the chain or not:
        return onPreHandle(request, response, pathConfig);
    }

    if (log.isTraceEnabled()) {
        log.trace("Filter '{}' is disabled for the current request under path '{}' with con
                "The next element in the FilterChain will be called immediately.",
                new Object[]{getName(), path, pathConfig});
    }
    //This filter is disabled for this specific request,
    //return 'true' immediately to indicate that the filter will not process the request
    //and let the request/response to continue through the filter chain:
    return true;
}
```


从上面3个步骤中可以看到，PathMatchingFilter提供的功能是：自定义匹配url，匹配上的请求最终跳转到`onPreHandle`方法。

这个过滤器为后面的常用过滤器提供的基础，比如我们在config中配置如下

```
/login = anon
/admin/* = authc
```

拦截/login请求，经过AnonymousFilter过滤器，我们可以看下

- org.apache.shiro.web.filter.authc.AnonymousFilter

```
public class AnonymousFilter extends PathMatchingFilter {

    /**
     * 公众号：MarkerHub
    **/
    @Override
    protected boolean onPreHandle(ServletRequest request, ServletResponse
response, Object mappedValue) {
        // Always return true since we allow access to anyone
        return true;
    }
}
```

AnonymousFilter重写了onPreHandle方法，只不过直接返回了true，说明拦截的链接可以直接通过，不需要其他拦截逻辑。

而authc->FormAuthenticationFilter也是间接继承了PathMatchingFilter。

```
public class FormAuthenticationFilter extends AuthenticatingFilter
```

所以，需要拦截某个链接进行业务逻辑过滤的可以继承PathMatchingFilter方法拓展哈。

6、AccessControlFilter

访问控制过滤器。继承PathMatchingFilter过滤器，重写onPreHandle方法，又分出了两个抽象方法来控制

```
public boolean onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue) {
    return isAccessAllowed(request, response, mappedValue) || onAccessDenied(request, response, ma
}
```

- isAccessAllowed 是否允许访问
- onAccessDenied 是否拒绝访问

所以，我们现在可以通过重写这个抽象两个方法来控制过滤逻辑。另外多提供了3个方法，方便后面的过滤器使用。

```
/**
 * 公众号：MarkerHub
**/
protected void saveRequestAndRedirectToLogin(ServletRequest request,
ServletResponse response) throws IOException {
    saveRequest(request);
    redirectToLogin(request, response);
}

protected void saveRequest(ServletRequest request) {
    WebUtils.saveRequest(request);
}
```

```
protected void redirectToLogin(ServletRequest request, ServletResponse
response) throws IOException {
    String loginUrl = getLoginUrl();
    WebUtils.issueRedirect(request, response, loginUrl);
}
```

其中redirectToLogin提供了调整到登录页面的逻辑与实现，为后面的过滤器发现未登录跳转到登录页面提供了基础。

这个过滤器，我们可以灵活运用。

7、AuthenticationFilter

继承AccessControlFilter，重写了isAccessAllowed方法，通过判断用户是否已经完成登录来判断用户是否允许继续后面的逻辑判断。这里可以看出，从这个过滤器开始，后续的判断会与用户的登录状态相关，直接继承这些过滤器，我们不需要再自己手动去判断用户是否已经登录。并且提供了登录成功之后跳转的方法。

```
public abstract class AuthenticationFilter extends AccessControlFilter {
    public void setSuccessUrl(String successUrl) {
        this.successUrl = successUrl;
    }

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

8、AuthenticatingFilter

继承AuthenticationFilter，提供了自动登录、是否登录请求等方法。

```
/**
 * 公众号：MarkerHub
**/
public abstract class AuthenticatingFilter extends AuthenticationFilter {
    public static final String PERMISSIVE = "permissive";

    //TODO - complete JavaDoc

    protected boolean executeLogin(ServletRequest request, ServletResponse
response) throws Exception {
        AuthenticationToken token = createToken(request, response);
        if (token == null) {
            String msg = "createToken method implementation returned null. A
valid non-null AuthenticationToken " +
                    "must be created in order to execute a login attempt.";
```

```
            throw new IllegalStateException(msg);
        }
        try {
            Subject subject = getSubject(request, response);
            subject.login(token);
            return onLoginSuccess(token, subject, request, response);
        } catch (AuthenticationException e) {
            return onLoginFailure(token, e, request, response);
        }
    }

    protected abstract AuthenticationToken createToken(ServletRequest request,
ServletResponse response) throws Exception;

    /**
     * 公众号：MarkerHub
    **/
    @Override
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse
response, Object mappedValue) {
        return super.isAccessAllowed(request, response, mappedValue) ||
                (!isLoginRequest(request, response) &amp;amp;&amp;amp;
isPermissive(mappedValue));
    }
    ...
}
```

- executeLogin 执行登录
- onLoginSuccess 登录成功跳转
- onLoginFailure 登录失败跳转
- createToken 创建登录的身份token
- isAccessAllowed 是否允许被访问
- isLoginRequest 是否登录请求

这个方法提供了自动登录的课程，比如我们获取到token之后实行自动登录，这场景还是很场景的。

比如在开源项目renren-fast中，就是这样处理的：

- https://gitee.com/renrenio/renren-fast

```
public class OAuth2Filter extends AuthenticatingFilter {

    @Override
    protected AuthenticationToken createToken(ServletRequest request,
ServletResponse response) throws Exception {
        //获取请求token
        String token = getRequestToken((HttpServletRequest) request);

        if(StringUtils.isBlank(token)){
            return null;
```

```
        }

        return new OAuth2Token(token);
    }

    @Override
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse
response, Object mappedValue) {
        if(((HttpServletRequest)
request).getMethod().equals(RequestMethod.OPTIONS.name())){
            return true;
        }

        return false;
    }

    @Override
    protected boolean onAccessDenied(ServletRequest request, ServletResponse
response) throws Exception {
        //获取请求token，如果token不存在，直接返回401
        String token = getRequestToken((HttpServletRequest) request);
        if(StringUtils.isBlank(token)){
            HttpServletResponse httpResponse = (HttpServletResponse) response;
            httpResponse.setHeader("Access-Control-Allow-Credentials",
"true");
            httpResponse.setHeader("Access-Control-Allow-Origin",
HttpContextUtils.getOrigin());

            String json = new
Gson().toJson(R.error(HttpStatus.SC_UNAUTHORIZED, "invalid token"));

            httpResponse.getWriter().print(json);

            return false;
        }

        return executeLogin(request, response);
    }

/**
 *公众号：MarkerHub
**/
    @Override
    protected boolean onLoginFailure(AuthenticationToken token,
AuthenticationException e, ServletRequest request, ServletResponse response) {
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        httpResponse.setContentType("application/json;charset=utf-8");
        httpResponse.setHeader("Access-Control-Allow-Credentials", "true");
```

```
        httpResponse.setHeader("Access-Control-Allow-Origin",
HttpContextUtils.getOrigin());
        try {
            //处理登录失败的异常
            Throwable throwable = e.getCause() == null ? e : e.getCause();
            R r = R.error(HttpStatus.SC_UNAUTHORIZED, throwable.getMessage());

            String json = new Gson().toJson(r);
            httpResponse.getWriter().print(json);
        } catch (IOException e1) {

        }

        return false;
    }

    /**
     * 获取请求的token
     */
    private String getRequestToken(HttpServletRequest httpRequest){
        //从header中获取token
        String token = httpRequest.getHeader("token");

        //如果header中不存在token，则从参数中获取token
        if(StringUtils.isBlank(token)){
            token = httpRequest.getParameter("token");
        }

        return token;
    }


}
```

在`onAccessDenied`方法校验通过之后执行`executeLogin`方法完成自动登录！

9、FormAuthenticationFilter

基于form表单的账号密码自动登录的过滤器，我们只需要看这个方法就明白，和renren-fast的实现相似：

```
public class FormAuthenticationFilter extends AuthenticatingFilter {

    public static final String DEFAULT_USERNAME_PARAM = "username";
    public static final String DEFAULT_PASSWORD_PARAM = "password";
    public static final String DEFAULT_REMEMBER_ME_PARAM = "rememberMe";

    protected AuthenticationToken createToken(ServletRequest request,
ServletResponse response) {
```

```
        String username = getUsername(request);
        String password = getPassword(request);
        return createToken(username, password, request, response);
    }
  /**
   * 公众号：MarkerHub
  **/
    protected boolean onAccessDenied(ServletRequest request, ServletResponse
response) throws Exception {
        if (isLoginRequest(request, response)) {
            if (isLoginSubmission(request, response)) {
                if (log.isTraceEnabled()) {
                    log.trace("Login submission detected.  Attempting to
execute login.");
                }
                return executeLogin(request, response);
            } else {
                if (log.isTraceEnabled()) {
                    log.trace("Login page view.");
                }
                //allow them to see the login page ;)
                return true;
            }
        } else {
            if (log.isTraceEnabled()) {
                log.trace("Attempting to access a path which requires
authentication.  Forwarding to the " +
                        "Authentication url [" + getLoginUrl() + "]");
            }

            saveRequestAndRedirectToLogin(request, response);
            return false;
        }
    }
}
```

onAccessDenied调用executeLogin方法。默认的token是UsernamepasswordToken。

结束语

好了，今天先到这里啦，讲了多好内置的过滤器，代码有点多，你们可以用电脑打开文章，然后仔细研究，并回想自己使用shiro过滤器的时候，是不是和我讲的场景一样，结合起来。