Java中的过滤器与拦截器

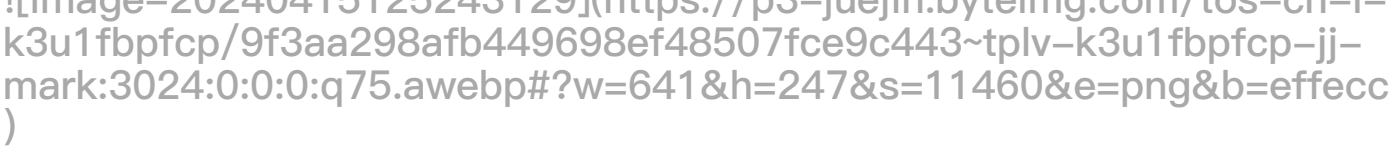
之前提到了登录校验需要在过滤器器和拦截器中进行校验，以JWT令牌来说，前端在获得token之后，后续的请求中都会在请求头中携带JWT令牌到服务端，而服务端需要统一拦截所有的请求，从而判断是否携带的有合法的JWT令牌。这一过程就是在过滤器和拦截器中进行的。

1. 过滤器（Filter）

1.1 什么是过滤器（filter）？

* Filter表示过滤器，是JavaWeb三大组件(Servlet、Filter、Listener)之一。
* 过滤器可以把对资源的请求拦截下来，从而实现一些特殊的功能

+ 使用了过滤器之后，要想访问web服务器上的资源，必须先经过滤器，过滤器处理完毕之后，才可以访问对应的资源。
* 过滤器一般完成一些通用的操作，比如：登录校验、统一编码处理、敏感字符处理等。



1.2 过滤器基本操作

0. 定义过滤器：定义一个类，实现 Filter 接口，并重写其所有方法。Filter在`jakarta.servlet`包下，版本较低的话是`javax.servlet`。对应的参数名可能发生变化，但本质还是一样的。

```
@WebFilter(urlPatterns = "/*")
public class DemoFilter implements Filter {
```

```
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("Demo 过滤器初始化！");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        System.out.println("执行了Demo过滤器！");

        System.out.println("DemoFilter    放行前逻辑.....");
        chain.doFilter(request,response); // 放行
        System.out.println("DemoFilter    放行后逻辑.....");
    }

    @Override
    public void destroy() {
        System.out.println("Demo 过滤器被摧毁！");
    }
}
```

> * `init`方法：过滤器的初始化方法。在web服务器启动的时候会自动的创建Filter过滤器对象，在创建过滤器对象的时候会自动调用init初始化方法，这个方法只会被调用一次。
> * `doFilter`方法：这个方法是在每一次拦截到请求之后都会被调用，所以这个方法是会被调用多次的，每拦截到一次请求就会调用一次doFilter()方法。
> * `destroy`方法： 是销毁的方法。当关闭服务器的时候，它会自动的调用销毁方法destroy，而这个销毁方法也只会被调用一次。
> * 在过滤器Filter中，如果不执行放行操作，将无法访问后面的资源。 放行操作：chain.doFilter(request, response);
> * 一般情况下，只需要重写`doFilter`方法

2. 配置过滤器：Filter类上加 `@WebFilter` 注解，配置拦截资源的路径。引导类上加 `@ServletComponentScan` 开启Servlet组件支持。

上述代码中已经加上了`@WebFilter`注解，urlPatterns为配置过滤器要拦截的请求路径（`/*`表示拦截浏览器的所有请求）

Filter可以根据需求，设置过滤器的拦截路径，配置不同的拦截资源路径：

拦截路径	urlPatterns值示例	含义
拦截具体路径	/login	只有访问 /login 路径时，才会被拦截
目录拦截	/users/*	访问/users下的所有资源，都会被拦截
拦截所有	/*	访问所有资源，都会被拦截

然后再启动类上需要加上`@ServletComponentScan` 开启Servlet组件支持。

```
 @ServletComponentScan // 开启Servlet组件支持
 @SpringBootApplication
 public class FilterInterceptorApplication {
     public static void main(String[] args) {
         SpringApplication.run(FilterInterceptorApplication.class, args);
     }
 }
```

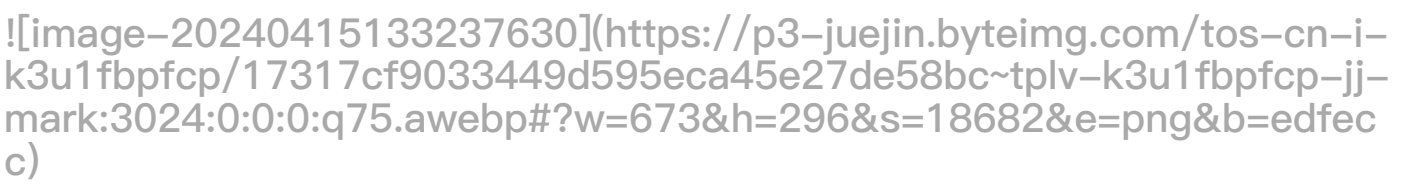
执行流程

使用ApiFox请求登录接口测试，结果如下，可以清晰看清过滤器的执行流程：

过滤器链

过滤器链指的是在一个web应用程序当中，可以配置多个过滤器，多个过滤器就形成了一个过滤器链。

其执行流程如下所示：

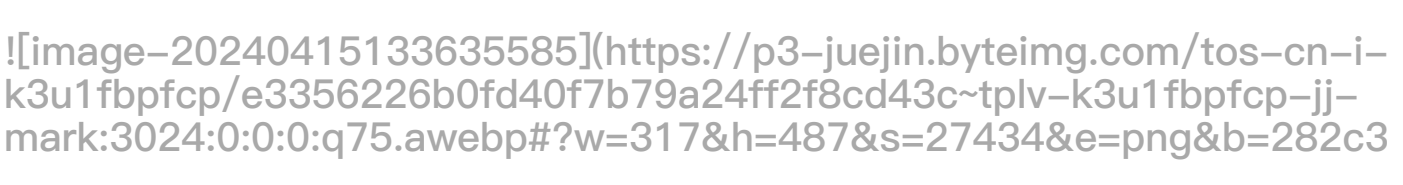


在添加一个AFilter过滤器，查看以一下效果：

AFilter.java

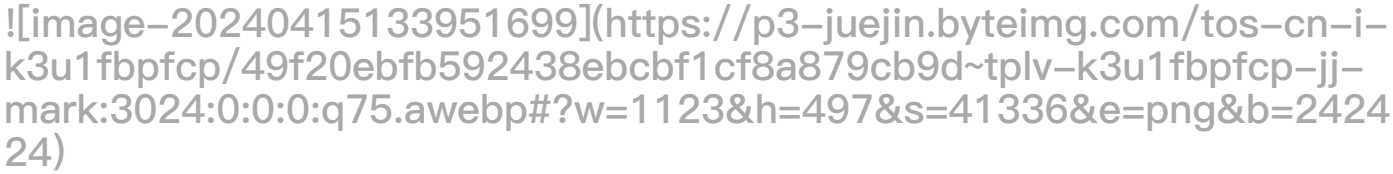
```
@WebFilter("/*")
public class AFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("A 过滤器初始化！");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        System.out.println("执行了A过滤器！");
        System.out.println("AFilter放行前逻辑.....");
        chain.doFilter(request,response); // 放行
        System.out.println("AFilter放行后逻辑.....");
    }

    @Override
    public void destroy() {
        System.out.println("A 过滤器被摧毁！");
    }
}
```

执行结果如下：



4)

可能会有人产生疑问了，为什么是A先执行，而不是Demo先执行？其实默认是根据`类名的字母排序`来的，若仅将AFilter改为ZFilter，看看是什么效果：



可以看到输出的信息，A过滤器（现在类名为ZFilter）就是后执行的。

1.3 Filter登录校验

现在，结合登录校验来看，新建一个LoginCheckFilter。

LoginCheckFilter.java

```
@WebFilter(urlPatterns = "/*")
public class LoginCheckFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        // 转为HttpServletRequest，HttpServletResponse
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        // 获取url
        String requestURL = httpRequest.getRequestURL().toString();
        // 若为登录请求
        if (requestURL.contains("login")) {
            // 直接放行
            chain.doFilter(request, response);
            return;
        }
        // 否则需要进行jwt校验
        String token = httpRequest.getHeader("token");
        if (StringUtils.hasLength(token)) {
            try {
```

```
                JwtUtil.ParseJwt(token);
                // 未出错则放行
                chain.doFilter(request, response);
                return;
            } catch (Exception e) {
                System.out.println("令牌不合法");
            }
        }
        // 返回提示信息
        Result responseResult = Result.error("未登录");
        String jsonString = JSONObject.toJSONString(responseResult);
        // 设定字符集，返回响应
        response.setContentType("application/json;charset=utf-8");
        response.getWriter().write(jsonString);
    }
}
```

Controller类中定义了两个接口：

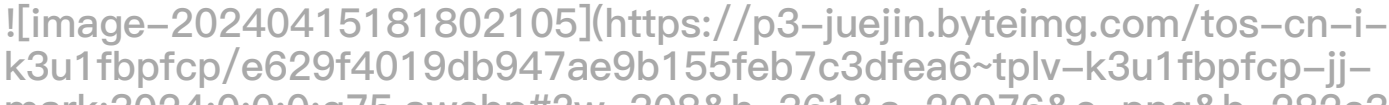
```
@RestController
public class JWTController {

    /**
     * 登录(生成jwt令牌)
     */
    @PostMapping("/login")
    public Result login(String username, String password) {

        // 登录成功后，生成jwt令牌
        Map<String, Object> claims = new HashMap<>();
        claims.put("username", username);
        claims.put("password", password);
        String jwt = JwtUtil.generateJwt(claims);

        return Result.success(jwt);
    }

    /**
     * 获取数据
     */
    @GetMapping("/data")
    public Result getData() {
        Map<String, Object> data = new HashMap<>();
```

```
        data.put("姓名", "小明");
        data.put("性别", "男");
        data.put("职业", "程序猿");
        data.put("城市", "深圳");
        data.put("愿望", "挣钱植发！");
        return Result.success(data);
    }
}
```

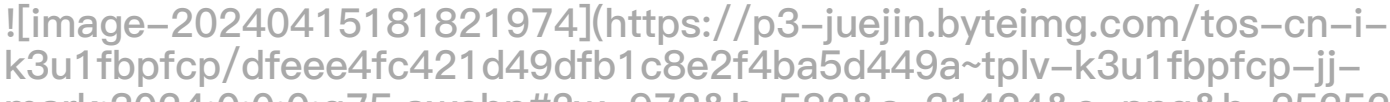
测试，先请求登录接口：

成功返回了JWT令牌。然后携带jwt去请求获取数据接口，直接在Headers中添加`token`参数，这个`token`名称不是固定的，但代码中需要和这里对应。

成功获得数据，若是修改一下jwt，再次请求呢?



终端显示：



这就是说明解析失败了，提示了错误信息。未经过校验，不能获取到相关数据

。

2. 拦截器（Interceptor）

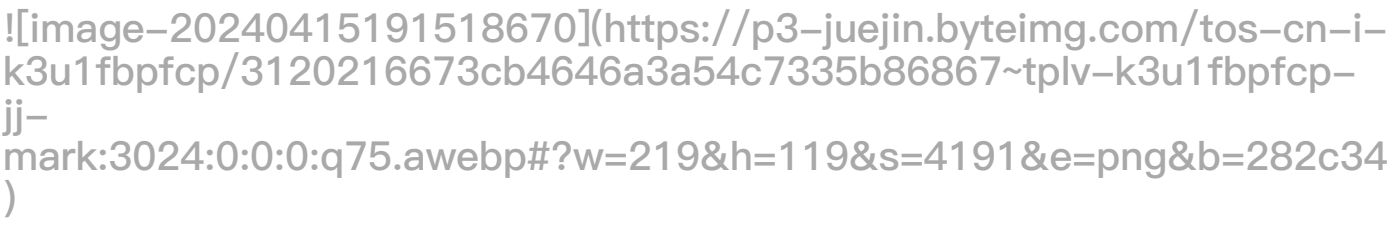
2.1 什么是拦截器?

什么是拦截器?

* 是一种动态拦截方法调用的机制，类似于过滤器。
* 拦截器是Spring框架中提供的，用来动态拦截控制器方法的执行。

拦截器的作用：

* 拦截请求，在指定方法调用前后，根据业务需要执行预先设定的代码。



* 当我们打开浏览器来访问部署在web服务器当中的web应用时，此时所定义的过滤器会拦截到这次请求。拦截到这次请求之后，它会先执行放行前的逻辑，然后再执行放行操作。而由于当前是基于springboot开发的，所以放行之后是进入到了spring的环境当中，也就是要来访问我们所定义的controller当中的接口方法。
* Tomcat（web服务器）并不识别所编写的Controller程序，但是它识别Servlet程序，所以在Spring的Web环境中提供了一个非常核心的Servlet：DispatcherServlet（前端控制器），所有请求都会先进行到DispatcherServlet，再将请求转给Controller。
* 当定义了拦截器后，会在执行Controller的方法之前，请求被拦截器拦截住。执行`preHandle()`方法，这个方法执行完成后需要返回一个布尔类型的值，如果返回true，就表示放行本次操作，才会继续访问controller中的方法；如果返回false，则不会放行（controller中的方法也不会执行）。
* 在controller当中的方法执行完毕之后，再回过来执行`postHandle()`这个方法以及`afterCompletion()` 方法，然后再返回给DispatcherServlet，最终再来执行过滤器当中放行后的这一部分逻辑的逻辑。执行完毕之后，最终给浏览器响应数据。

2.2 拦截器基本操作

0. 定义拦截器

```
 @Component
 public class LoginCheckInterceptor implements HandlerInterceptor {
     @Override
     public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
         System.out.println("preHandle");
         return true; // 放行
     }

     @Override
     public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView
modelAndView) throws Exception {
         System.out.println("postHandle");
     }

     @Override
     public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
         System.out.println("afterCompletion");
     }
 }
```

> * `preHandle`：目标资源方法执行前执行。 返回true：放行 返回false：不放行
> * `postHandle`：目标资源方法执行后执行
> * `afterCompletion`：视图渲染完毕后执行，最后执行

2. 配置拦截器：实现WebMvcConfigurer接口，并重写addInterceptors方法

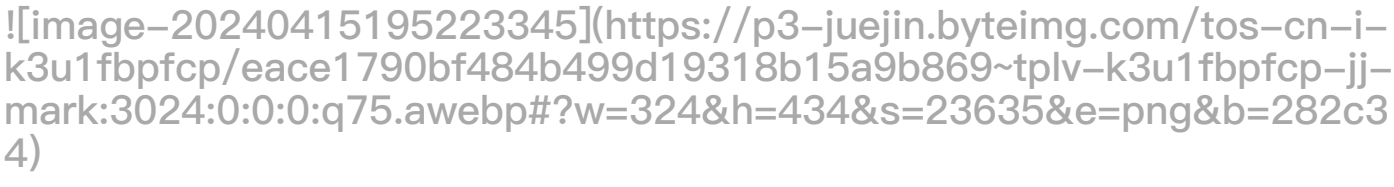
InterceptorConfig.java

```
 @Configuration
 public class InterceptorConfig implements WebMvcConfigurer {
     // 拦截器注入
     @Autowired
     private LoginCheckInterceptor loginCheckInterceptor;

     @Override
     public void addInterceptors(InterceptorRegistry registry) {
         // 注册自定义拦截器对象
         // addPathPatterns 设置拦截器拦截的请求路径（ /** 表示拦截所有请
求）
         // excludePathPatterns 设置拦截器不拦截的请求路径

registry.addInterceptor(loginCheckInterceptor).addPathPatterns("/**").ex
cludePathPatterns("/login");
     }
 }
```

拦截器的请求路径规则和过滤器不太一样，如下所示，：

拦截路径	含义	举例
/*	一级路径	能匹配/login，/data，不能匹配 /data/xx，/data/xx/xx,...
/**	任意级路径	能匹配/data，/data/xx，/data/xx/xx,...
/data/*	/data下的一级路径	能匹配/data/xx，不能匹配 /data/xx/xx，/data
/data/**	/data下的任意级路径	能匹配 /data，/data/xx，/data/xx/xx，不能匹配/login/xx

测试一下（这里将之前过滤器的`@WebFilter`注解注释掉了）：



我们打开注释，来看看执行顺序：

过滤器和拦截器的执行顺序大家应该都清楚了，主要区别以下两点：

* 接口规范不同：过滤器需要实现Filter接口，而拦截器需要实现HandlerInterceptor接口。
* 拦截范围不同：过滤器Filter会拦截所有的资源，而Interceptor只会拦截Spring环境中的资源。

2.3 Interceptor登录校验

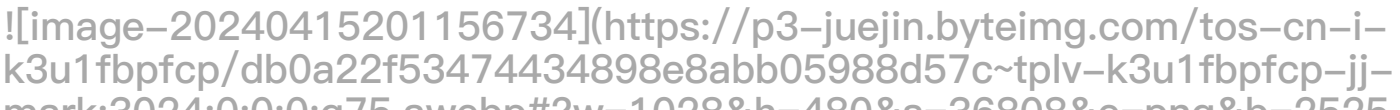
其实逻辑和过滤器是一样的，代码如下：

```
@Component
public class LoginCheckInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        // 获取url
        String requestURL = request.getRequestURL().toString();
        // 若为登录请求
        if (requestURL.contains("login")) {
            // 直接放行
            return true;
        }
        // 否则需要进行jwt校验
        String token = request.getHeader("token");
        if (StringUtils.hasLength(token)) {
            try {
                JwtUtil.ParseJwt(token);
                // 未出错则放行
                return true;
            } catch (Exception e) {
                System.out.println("令牌不合法");
            }
        }
```

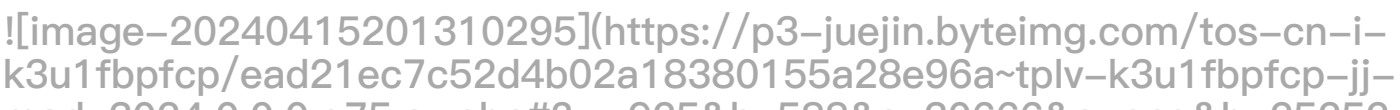
```
            // 返回提示信息
            Result responseResult = Result.error("未登录");
            String jsonString = JSONObject.toJSONString(responseResult);
            // 设定字符集，返回响应
            response.setContentType("application/json;charset=utf-8");
            response.getWriter().write(jsonString);
            return false;
        }
    }

```

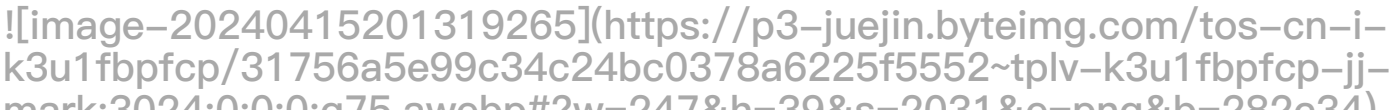
登录请求：

获取数据请求（携带token）：



更改错误的token值请求：





到此也就验证了所开发的登录校验的拦截器也是没问题的。

登录校验的过滤器和拦截器，只需要使用其中的一种就可以了。