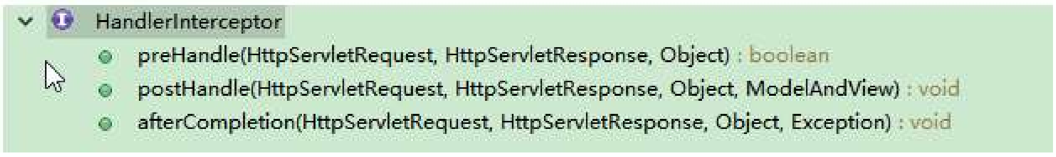
# 10、拦截器

**笔记本:** spring
**创建时间:** 2022/4/4 17:24
**作者:** 雷丰阳

---

SpringMVC提供了拦截器机制；允许运行目标方法之前进行一些拦截工作，或者目标方法运行之后进行一些其他处理；
Filter；javaWeb
HandlerInterceptor：SpringMVC


HandlerInterceptor
preHandle(HttpServletRequest, HttpServletResponse, Object) : boolean
postHandle(HttpServletRequest, HttpServletResponse, Object, ModelAndView) : void
afterCompletion(HttpServletRequest, HttpServletResponse, Object, Exception) : void

preHandle：在目标方法运行之前调用；返回boolean；return true；(chain.doFilter())放行； return false；不放行
postHandle：在目标方法运行之后调用：目标方法调用之后
afterCompletion：在请求整个完成之后；来到目标页面之后；chain.doFilter()放行；资源响应之后；
1）、拦截器是一个接口
2）、实现HandlerInterceptor接口；
3）、配置拦截器
4）、拦截器的运行流程
正常运行流程；
 拦截器的preHandle------目标方法-----拦截器postHandle-----页面-------拦截器的afterCompletion;

```
MyFirstInterceptor...preHandle...
test01....
MyFirstInterceptor...postHandle...
success.jsp....
MyFirstInterceptor...afterCompletion
```

其他流程：
1、只要preHandle不放行就没有以后的流程；
2、只要放行了，afterCompletion都会执行；

---

2、多个拦截器
正常流程：

```
MyFirstInterceptor...preHandle...
MySecondInterceptor...preHandle...
test01....
MySecondInterceptor...postHandle...
MyFirstInterceptor...postHandle...
success.jsp....
MySecondInterceptor...afterCompletion...
MyFirstInterceptor...afterCompletion
```

异常流程：
1、不放行；
  1)、哪一块不放行从此以后都没有；
  MySecondInterceptor不放行；但是他前面已经放行了的拦截器的afterCompletion总会执行；

```
MyFirstInterceptor...preHandle...
MySecondInterceptor...preHandle...
```

```
MyFirstInterceptor...afterCompletion
```

流程：filter的流程；
拦截器的preHandle：是按照顺序执行
拦截器的postHandle：是按照逆序执行
拦截器的afterCompletion：是按照逆序执行；
已经放行了的拦截器的afterCompletion总会执行；

---

```
try {
            ModelAndView mv = null;
            Exception dispatchException = null;

            try {
                processedRequest = checkMultipart(request);
                multipartRequestParsed = processedRequest != request;

                // Determine handler for the current request.拿到方法的执行
链，包含拦截器
                mappedHandler = getHandler(processedRequest);
                if (mappedHandler == null || mappedHandler.getHandler() ==
null) {
                    noHandlerFound(processedRequest, response);
                    return;
                }

                // Determine handler adapter for the current request.
                HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());

                // Process last-modified header, if supported by the
handler.
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                    if (logger.isDebugEnabled()) {
                        String requestUri =
urlPathHelper.getRequestUri(request);
                        logger.debug("Last-Modified value for [" +
requestUri + "] is: " + lastModified);
                    }
                    if (new ServletWebRequest(request,
response).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }

               //拦截器preHandle执行位置;有一个拦截器返回false目标方法以后都不会
执行；直接跳到afterCompletion
                if (!mappedHandler.applyPreHandle(processedRequest,
response)) {
                    return;
                }

                try {
                    // Actually invoke the handler.适配器执行目标方法
                    mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
                }
                finally {
                    if (asyncManager.isConcurrentHandlingStarted()) {
                        return;
                    }
                }

                applyDefaultViewName(request, mv);
                 //目标方法只要正常就会走到postHandle;任何期间有异常
                mappedHandler.applyPostHandle(processedRequest, response,
mv);
```

```
            }
            catch (Exception ex) {
                dispatchException = ex;
            }

            //页面渲染；如果完蛋也是直接跳到afterCompletion；
            processDispatchResult(processedRequest, response, mappedHandler,
mv, dispatchException);
        }
        catch (Exception ex) {
            triggerAfterCompletion(processedRequest, response,
mappedHandler, ex);
        }
        catch (Error err) {
            triggerAfterCompletionWithError(processedRequest, response,
mappedHandler, err);
        }
        finally {
            if (asyncManager.isConcurrentHandlingStarted()) {
                // Instead of postHandle and afterCompletion
                mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest,
response);
                return;
            }
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
```

---

## preHandle

```
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        if (getInterceptors() != null) {
            for (int i = 0; i < getInterceptors().length; i++) {
                HandlerInterceptor interceptor = getInterceptors()[i];

                //preHandle-true-false
                if (!interceptor.preHandle(request, response, this.handler))
{
                    //执行完afterCompletion（）;
                    triggerAfterCompletion(request, response, null);
                    //返回一个false
                    return false;
                }
               //记录一下索引
               //this.interceptorIndex = i;
            }
        }
        return true;
    }
```

---

## postHandle

```
void applyPostHandle(HttpServletRequest request, HttpServletResponse
response, ModelAndView mv) throws Exception {
        if (getInterceptors() == null) {
            return;
        }
        //逆向执行每个拦截器的postHandle
        for (int i = getInterceptors().length - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = getInterceptors()[i];
            interceptor.postHandle(request, response, this.handler, mv);
        }
    }
```

```
private void processDispatchResult(HttpServletRequest request,
HttpServletResponse response,
            HandlerExecutionChain mappedHandler, ModelAndView mv, Exception
exception) throws Exception {

        boolean errorView = false;

        if (exception != null) {
            if (exception instanceof ModelAndViewDefiningException) {
                logger.debug("ModelAndViewDefiningException encountered",
exception);
                mv = ((ModelAndViewDefiningException)
exception).getModelAndView();
            }
            else {
                Object handler = (mappedHandler != null ?
mappedHandler.getHandler() : null);
                mv = processHandlerException(request, response, handler,
exception);
                errorView = (mv != null);
            }
        }

        // Did the handler return a view to render?
        if (mv != null && !mv.wasCleared()) {
             页面渲染
            render(mv, request, response);
            if (errorView) {
                WebUtils.clearErrorRequestAttributes(request);
            }
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("Null ModelAndView returned to
DispatcherServlet with name '" + getServletName() +
                        "': assuming HandlerAdapter completed request
handling");
            }
        }

        if
(WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
            // Concurrent handling started during a forward
            return;
        }

        if (mappedHandler != null) {
               //页面正常执行afterCompletion；即使没走到这，afterCompletion总会
执行；
            mappedHandler.triggerAfterCompletion(request, response, null);
        }
    }
```

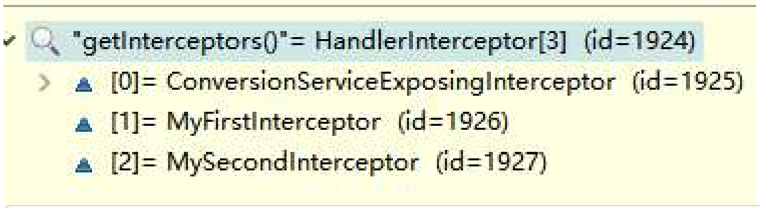
```
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse
response, Exception ex)
            throws Exception {

        if (getInterceptors() == null) {
            return;
        }

          //有记录最后一个放行拦截器的索引，从他开始把之前所有放行的拦截器的
afterCompletion都执行
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = getInterceptors()[i];
            try {
                interceptor.afterCompletion(request, response, this.handler, 
ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw 
exception", ex2);
            }
        }
    }
```

---

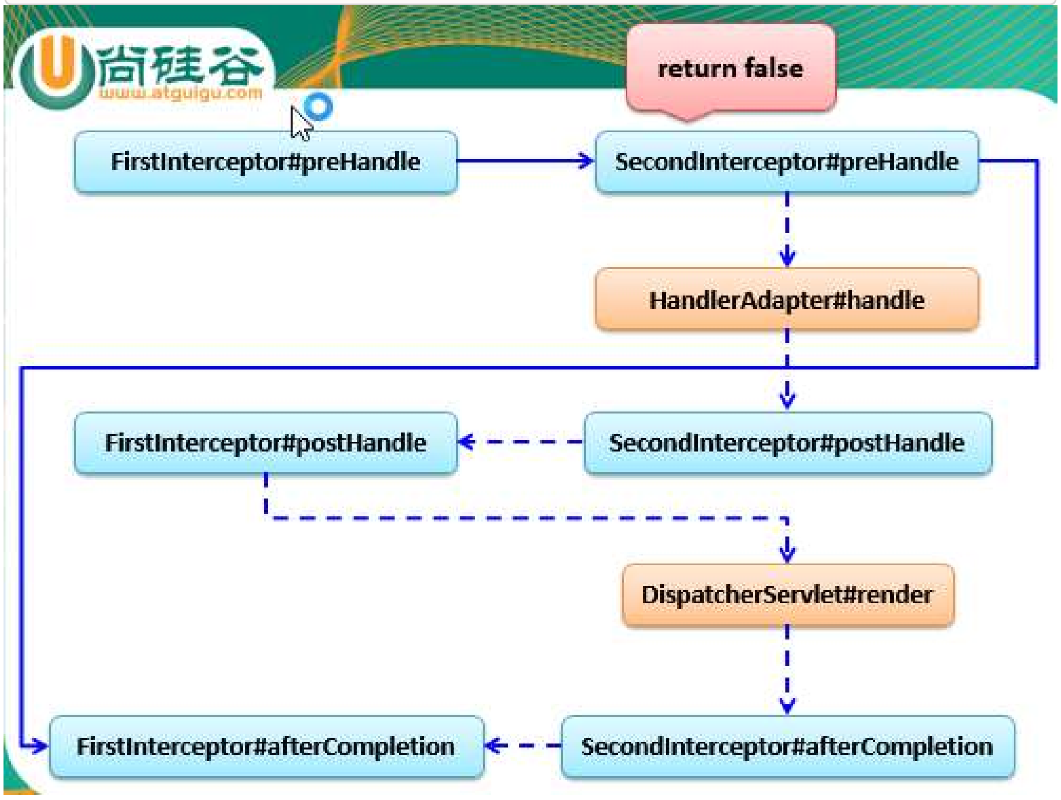
preHandle：



```
"getInterceptors()"= HandlerInterceptor[3]  (id=1924)
    [0]= ConversionServiceExposingInterceptor  (id=1925)
    [1]= MyFirstInterceptor  (id=1926)
    [2]= MySecondInterceptor  (id=1927)
```

```
第一次：ConversionServiceExposingInterceptor  interceptorIndex=0;
第二次：MyFirstInterceptor                    interceptorIndex=1
第三次：MySecondInterceptor          执行afterCompletion()
已经放行了的拦截器的afterCompletion总会执行
```

```
 for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = getInterceptors()[i];
            try {
                interceptor.afterCompletion(request, response, this.handler, 
ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw 
exception", ex2);
            }
        }
```


return false
FirstInterceptor#preHandle
SecondInterceptor#preHandle
HandlerAdapter#handle
FirstInterceptor#postHandle
SecondInterceptor#postHandle
DispatcherServlet#render
FirstInterceptor#afterCompletion
SecondInterceptor#afterCompletion

---

什么时候用Filter什么时候用拦截器？

如果某些功能；需要其他组件配合完成，我们就
使用拦截器；
其他情况可以写filter；