서블릿 필터(Filter)

서블릿은 웹에서 실행되는 프로그램이기 때문에 네트워크 통신의 사이 사이에 특별한 동작을 끼워 넣을 수 있다.
예를 들면, 홈페이지에 접속하기 직전에 이벤트 창을 띄운다든지, 데이터를 입력한 후 실제 저장하는 페이지로 넘어가지 전에 넘겨지는 데이터들에 대하여 한글 처리를 한다든지 등의 작업을 할 수 있다. 또한, 세션이 만들어지거나 삭제될 때 이것을 감지하는 작업도 할 수 있다.
이렇게 **여러가지 동작에 있어서 사이 사이에 끼워져서 실행되는 서블릿의 클래스를 필터**라 부르고 **동작이 발생할 때 감지하는 것을 이벤트**라 부른다.

1. 서블릿 필터

필터는 말 그대로 여과 기능을 수행한다. 웹 프로그램에서도 하나의 페이지에서 다른 페이지로 전달되는 데이터가 필터를 지나 가공되거나 걸러지게 된다.
일반적으로 웹 프로그램은 A->B 라는 식으로 실행 흐름이 있다. 그러나 기존의 흐름에 C 라는 작업을 끼워 넣을 수 있다면 도움이 될 것이다.
예를 들어 A 에서 B 로 넘겨지는 데이터에 인코딩을 한다든지 데이터에 세션을 확인해서 B 페이지를 보여줄지를 결정하는 작업 등을 할 수 있다.

필터는 데이터를 가로채서 처리를 하는 클래스라고 생각하면 된다.
하나의 작업에서 다른 작업으로 넘어갈 때나 어떤 작업이 또 다른 작업으로 넘어갈 때 데이터를 가로채서 처리를 할 수 있다.
요청이나 세션에 담긴 데이터뿐 아니라 해더에도 필터가 적용될 수 있다. 기존 작업이 일어나기 직전이나(전처리) 일어난 직후(후처리) 모두 필터가 적용되는 시점이다.

웹 관련 클래스가 모두 그러하듯이 **필터 클래스의 메서드**도 요청 객체(request)와 응답 객체(response)를 매개변수로 가진다.
여기에 추가적으로 **FilterChain 객체**를 매개 변수로 갖는데, 이유는 필터 기능 자체가 페이지의 분기점에 있기 때문이다.
따라서, **FilterChain 객체**는 필터 기능이 완료되면 다음 페이지로 연결하는 기능에 사용된다. 또한, 서블릿의 일반 클래스처럼 **web.xml 파일에 등록**해야 한다.
web.xml 에 등록할 때 **<filter> 태그를 사용**한다.
필터 관련 클래스로는 **javax.servlet.Filter, javax.servlet.FilterConfig, javax.servlet.FilterChain** 등이 있다.

필터가 웹 프로그램에서 사용되는 경우

- 전달받은 데이터를 인코딩 처리하는 경우
- 세션 데이터를 인증하는 경우
- 이벤트나 공지 등 팝업을 추가 하는 경우

예제(전달받은 데이터를 인코딩하는 경우)

1) web.xml

```
<filter><!--<filter> 태그는 <servlet> 태그보다 앞에 놓여야 한다./ 필터를 지정하는 역할-->
        <filter-name>MyFilter</filter-name><!--getFilterName()-->
        <filter-class>test.fiter.MyFilter</filter-class>
</filter>
<filter-mapping>
        <filter-name>MyFilter</filter-name>
        <url-pattern>/login.do</url-pattern>
        /login.do 서블릿에 접근하기 전에 필터를 거는 것이다.
.       <!--<url-pattern>/*</url-pattern>-->
        <!--URL 패턴에서 '/*' 와 같이 적으면 모든 페이지에 접근하기 전에 해당 필터 클래스가 실행된다.-->
</filter-mapping>
<servlet>
        <servlet-name>Login</servlet-name>
        <servlet-class>test.controller.LoginServlet</servlet-class>
</servlet>
```

2) 필터 클래스

```
package test.filter;

import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class MyFilter implements Filter {
        // 필터는 Filter 인터페이스를 상속 구현해야 한다.
        public void init(FilterConfig fc) throws ServletException{ } // 필터 초기화 작업
    // init() 메서드의 매개변수는 FilterConfig 객체이다.
```

```
	public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) throws IOException, ServletException {
// 1. request 파라미터를 이용하여 요청의 필터 작업 수행
// doFilter() 메서드를 사용하고, 매개 변수로는
반드시 ServletRequest, ServletResponse, FilterChain 세 가지를 사용한다.

		HttpServletRequest h_request = (HttpServletRequest)request;
		String method = h_request.getMethod();
		if(method.equalsIgnoreCase("POST")) {
			request.setCharacterEncoding("euc-kr");
		}

		chain.doFilter(request, response);
// 2. 체인의 다음 필터 처리
// 3. response 를 이용하여 응답의 필터링 작업 수행
// 필터 메서드 내용부의 마지막 코드는 현재까지 작업한 내용을 적용하고 연결된 페이지로
이동하도록 만들어 준다. 이런 역할을 하는 메서드가 chain 객체의 doFilter()이다.
// 세번째 매개변수인 FilterChain 클래스의 객체인 chain 을 이용해서 다른 필터나 서블릿과
연결하는 코드를 반드시 작성해야 한다.
}

		public void destroy() {
// 4. 주로 필터가 사용한 자원을 반납
		}
}
```

web.xml 필터 설정하는 방법

```
<web-app>
    <filter><!-- 웹 어플리케이션에서 사용될 필터를 지정하는 역할 -->
        <filter-name>FilterName</filter-name><!-- getFilterName() -->
        <filter-class>javacan.filter.FilterClass</filter-class>
        <init-param><!-- getInitParameter() -->
            <param-name>paramName</param-name>
         <!-- getInitParameter(String name)-->paramName 은 파라미터 name 변수의 key 값-->
         <!-- 필터가 초기화될 때, 즉 필터의 init() 메소드가 호출될 때 전달되는 파라미터 값,
              이는 서블릿의 초기화 파라미터와 비슷한 역할을 하며 주로 필터를 사용하기
              전에 초기화해야 하는 객체나 자원을 할당할 때 필요한 정보를 제공하기 위해
              사용된다.-->
```

```
            <param-value>value</param-value>
            <!--getInitParameter(String name)에서 name 값--> value 는 name 변수의 value 값 -->
        </init-param>
    </filter>
    <filter-mapping> <!-- 특정 자원에 대해 어떤 필터를 사용할지를 지정 -->
        <filter-name>FilterName</filter-name>
        <url-pattern>*.jsp</url-pattern>
        <!-- 클라이언트가 jsp 확장자를 갖는 자원을 요청할 경우 FilterName 가 사용되도록 지정,
            클라이언트가 요청한 특정 URI 에 대해서 필터링을 할 때 사용 -->
    </filter-mapping>
</web-app>
```

'/'로 시작하고 '/*'로 끝나는 url-pattern 은 모든 경로 매핑을 위해서 사용된다.
'*.확장자'로 시작하는 url-pattern 은 확장자에 대한 매핑을 할 때 사용된다.
오직 '/'만 포함하는 경우 어플리케이션의 기본 서블릿으로 매핑한다.
나머지 다른 문자열은 문자열에 대한 정확한 매핑을 위해서 사용된다.

```
<filter-mapping>
        <filter-name>AuthCheckFilter</filter-name>
        <servlet-name>FileDownload</servlet-name>
</filter-mapping>

<servlet>
        <servlet-name>FileDownload</servlet-name>
        ...
</servlet>
```

<url-pattern> 태그를 사용하지 않고 대신 <servlet-name> 태그를 사용함으로써 특정 서블릿에 대한 요청에 대해서 필터를 적용할 수도 있다. 예를 들면 다음과 같이 이름이 FileDownload 인 서블릿에 대해서 AuthCheckFilter 를 필터로 사용하도록 할 수 있다.

```
<filter-mapping>
        <filter-name>AuthCheckFilter</filter-name>
        <servlet-name>FileDownload</servlet-name>
        <dispather>INCLUDE</dispather>
</filter-mapping>
```

<dispather> 태그는 실행되는 자원을 클라이언트가 요청한 것인지, 아니면 RequestDispather 의 forward()를 통해서 이동한 것인지, 아니면 include() 통해서 포함되는 것인지에 따라서 필터를 적용하도록 지정할 수 있다.

<dispather> 태그가 가질 수 있는 값은 다음과 같다.

REQUEST: 클라이언트의 요청인 경우에 필터를 사용한다.
FORWARD: forward()를 통해서 제어를 이동하는 경우에 필터를 사용한다.
INCLUDE: include()를 통해서 포함하는 경우에 필터를 사용한다.

2. Filter 인터페이스

Filter 인터페이스는 다음과 같은 메서드를 선언하고 있으며, 필터 기능을 제공할 클래스는 Filter 인터페이스를 알맞게 구현해 주어야 한다.

1) public void **init**(FilterConfig filterConfig) throws ServletException
--> 필터를 초기화할 때 호출한다.

2) public void **doFilter**(ServletRequest request, ServletResponsereponse, FilterChain chain) throws java.io.IOException, ServletException
--> 체인을 따라 다음에 존재하는 필터로 이동한다. 체인의 가장 마지막에는 클라이언트가 요청한 최종 자원이 위치한다.

3) public void **destory**()
--> 필터가 웹 컨테이너에서 삭제될 때 호출된다.

위 메서드에서 필터의 역할을 하는 메서드가 바로 doFilter() 메서드이다. 서블릿 컨테이너는 사용자가 특정한 자원을 요청했을 때 그 자원 사이에 필터가 존재할 경우 필터 객체의 doFilter() 메서드를 호출하며 바로 이 시점부터 필터가 작동하기 시작한다.

```
public   class FirstFilter implements Filter{
          public void init(FilterConfigfilterConfig) throws ServletException{
                    // 필터 초기화 작업
          }
          public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException{
          // 1. request 파라미터를 이용하여 요청객체의 필터 작업 수행
          ....
          // 2. 체인의 다음 필터 처리
          chain.doFilter(request, response);
          // 3. response 를 이용하여 응답객체의 필터링 작업 수행
          ....
          }
          // 4. 주로 필터가 사용한 자원을 반납
          public void destory{
          }
```

```
}
```

위 코드에서 Filter 인터페이스의 doFilter() 메서드는 요청이 있을 때마다 매번 실행된다. 예를 들어, 클라이언트가 요청한 자원이 필터를 거치는 경우 클라이언트의 요청이 있을 때마다 doFilter() 메서드가 호출되며, doFilter() 메서드는 JSP/서블릿과 마찬가지로 요청에 대해서 알맞은 작업을 처리하게 된다.

※ doFIilter() 메서드 내에서 이루어지는 작업의 순서

1) request 파라미터를 이용하여 클라이언트의 요청객체 필터링: 1 단계에서는 RequestWrapper 클래스를 사용하여 클라이언트의 요청을 변경한다.
2) chain.doFiter() 메서드 호출: 2 단계에서는 요청객체의 필터링 결과를 다음 필터에 전달한다.
3) response 파라미터를 사용하여 클라이언트로 가는 응답객체 필터링: 3 단계에서는 체인을 통해서 전달된 응답 데이터를 변경하여 그 결과를 클라이언트에 전송한다.

※ FilterConfig 가 제공하는 메서드

메서드	리턴 타입	설명
getFilterName()	String	설정파일에서 <filter-name>에서 지정한 필터의 이름을 리턴한다.
getInitParameter(String name)	String	설정파일의 <init-param>에서 지정한 초기화 파라미터의 값을 읽어온다. 존재하지 않을 경우 null 을 리턴한다.
getInitParameterNames()	Enumeration<String>	초기화 파라미터의 이름 목록을 구한다.
getServletContext()	ServletContext	서블릿 컨텍스트 객체를 구한다.

3. 요청 및 응답 래퍼 클래스

필터가 필터로서의 재기능을 하기 위해서는 클라이언트의 요청을 변경하고, 또한 클라이언트로 가는 응답을 변경할 수 있어야 할 것이다. 이러한 변경을 할 수 있도록 해주는 것이 바로 ServletRequestWrapper 와 ServletResponseWrapper 이다.
서블릿 요청/응답 래퍼 클래스를 이용함으로써 클라이언트의 요청 정보를 변경하여 최종 자원인 서블릿/JSP/HTML/기타 자원에 전달할 수 있고, 또한 최종 자원으로부터의 응답 결과를 변경하여 새로운 응답 정보를 클라이언트에 보낼 수 있게 된다.

서블릿 요청/응답 래퍼 클래스로서의 역할을 수행하기 위해서는 javax.servlet 패키지에 정의되어 있는 ServletRequestWrapper 클래스와 ServletResponseWrapper 클래스를 **상속**받으면 된다.
하지만, 대부분의 경우 HTTP 프로토콜에 대한 요청/응답을 필터링하기 때문에 이 두 클래스를 상속받아 알맞게 구현한 HttpServletRequestWrapper 클래스와 HttpServletResponseWrapper 클래스를 상속받는 경우가 대부분이다.

HttpServletRequestWrapper 클래스와 **HttpServletResponseWrapper** 클래스는 모두 **javax.servlet.http** 패키지에 정의되어 있으며, 이 두 클래스는 각각 HttpServletRequest 인터페이스와 HttpServletResponse 인터페이스에 정의되어 있는 모든 메소드를 이미 구현해 놓고 있다. 필터를 통해서 변경하고 싶은 정보가 있을 경우 그 정보를 추출하는 메소드를 알맞게 오버라이딩하여 필터의 doFilter() 메소드에 넘겨주기만 하면 된다.
예를 들어, 클라이언트가 전송한 "company" 파라미터 값을 무조건 "JavaCan.com"으로 변경하는 요청 래퍼 클래스는 다음과 같이 HttpServletRequestWrapper 클래스를 상속받은 후에 getParameter() 메소드를 알맞게 구현하면 된다.

```
import java.util.Collections;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;

// 요청 레퍼 클래스로 동작하기 위해 HttpServletRequestWrapper 클래스를 상속받는다.
public class NullParameterRequestWrapper extends HttpServletRequestWrapper {

	private Map<String, String[]>parameterMap = null;
	public NullParameterRequestWrapper(HttpServletRequest request) {
		super(request);
		// 생성자는 전달받은 request 의 파라미터의 정보를 parameterMap 에 저장한다.
		parameterMap = new HashMap<String, String[]>(request.getParameterMap());

	}

	// checkNull() 메서드는 검사할 파라미터의 이름 목록을 인자로 전달받는다.
	//인자로 전달받은 각각의 이름을 검사해서 해당 이름의 파라미터가 존재하지 않으면
	// 기본값으로 빈 문자열을 저장한다.
	public void checkNull(String[] parameterNames) {
		for (inti = 0; i<parameterNames.length; i++) {
			if (!parameterMap.containsKey(parameterNames[i])) {
				String[] values = new String[] { "" };
				parameterMap.put(parameterNames[i], values);
			}
		}
	}
```

```
	@Override
	public String getParameter(String name) {
		String[] values = getParameterValues(name);
		if (values != null &&values.length> 0)
			return values[0];
		return null;
	}

	@Override
	public Map<String, String[]>getParameterMap() {
		returnparameterMap;
	// 파라미터와 관련된 메서드를 구현해서 parameterMap 으로부터
	// 파라미터값을 읽어오도록 한다.
	}

	@Override
	public Enumeration<String>getParameterNames() {
		returnCollections.enumeration(parameterMap.keySet());
	}

	@Override
	public String[] getParameterValues(String name) {
		return (String[]) parameterMap.get(name);
	}
}
```