第十七节 安全架构重构

Java Security

权限 API - Permission

JDK 内建 API

- java.security.Permission

- java.security.PermissionCollection
- java.security.Permissions
- java.security.UnresolvedPermission
- java.io.FilePermission
- java.net.SocketPermission
- java.security.BasicPermission
- java.util.PropertyPermission
- java.lang.RuntimePermission
- java.awt.AWTPermission
- java.net.NetPermission
- java.lang.reflect.ReflectPermission
- java.io.SerializablePermission
- java.security.SecurityPermission
- java.security.AllPermission

- javax.security.auth.AuthPermission

安全策略

安全策略配置文件

文件路径：$JAVA_HOME/jre/lib/security/java.policy

```
grant codeBase "file:${{java.ext.dirs}}/*" {
        permission java.security.AllPermission;
};
```

格式：

permission ${java.security.Permission 实现类} "${permisson.name}";

```
grant {

        permission java.lang.RuntimePermission
"stopThread";

        // allows anyone to listen on dynamic
ports
        permission java.net.SocketPermission
"localhost:0", "listen";

        // "standard" properies that can be
read by anyone

        permission java.util.PropertyPermission
"java.version", "read";
}
```

> java.ext.dirs 系统属性表示：JVM 扩展的 ClassLoader 路径
>
> - Bootstrap ClassLoader
>   - System ClassLoader
>   - App ClassLoader
>   - Ext ClassLoader（JDK 9 开始淘汰）

激活安全管理器 - SecurityManager

```
System.setSecurityManager(new
SecurityManager());
```

自定义权限

需要用到这些 API：

- Java 安全校验方法 - SecurityManager#checkPermission(java.security.Permission)

java.security.AccessController#checkPermission(java.security.Permission)

- Java 鉴权方法 - java.security.AccessController#doPrivileged(java.security.PrivilegedAction) 以及重载

第三方扩展

Apache Tomcat

文档参考：http://tomcat.apache.org/tomcat-7.0-doc/security-manager-howto.html

思考：假设在一台机器上，安装一个版本 JDK，如 JDK 1.8.0，部署了三台 Tomcat，要为它们分别设置权限，如果仍然使用 JDK 中的 java.policy 的话，那么是否会出现逻辑歧义?

Tomcat 自定义 Policy 文件

位置：$CATALINA_HOME/conf/catalina.policy 中

思考：一个 Tomcat JVM 部署了三个 Web Apps，三个 Web Apps 文件目录是否能被访问?

假设 Web App1 目录：web-app1，依次类推，web-app2, web-app3

web-app1 代码：

```
public void
removeWebAppsDirectory(ServletContext
servletContext){
    String rootDirPath =
servletContext.getRealPath("/");
    // $CATALINA_HOME/webapps/web-app1
    File rootDir = new File(rootDirPath);
    File webappsDir = rootDir.getParentFile();
    File webapp2Dir = new File(webappsDir,"web-
app2");
    deleteDirectory(webapp2Dir); // 通过
FilePermission 来限制
}
```

思考：不同 Web Apps 在同一 Tomcat JVM 中，如何写日志?

1. 向当前 Web App 自己目录写，ServletContext 对应的目录
2. 向 Tomcat logs 目录写，利用 ServletContext#log API 实现

Spring Security

安全设计模式

基于拦截模式实现，比如利用 AOP，Servlet Filter

Servlet 技术栈

基于 Servlet 规范中的 Filter API 实现。第一部分来自于 Spring Web 实现。第二部分来自于 Spring Security 实现

Spring Web 实现

org.springframework.web.filter.DelegatingFilterProxy

- 使用场景

  Spring Web 为应用提供了一个 Servlet 容器 Filter 组件和 Spring Filter Bean 桥接的管道

Delegate - 实际 Filter

- targetBeanName Filter Bean
- delegate Filter 对象

Proxy - Filter 代理

```
    <filter>
        <filter-
name>DelegatingFilterProxy</filter-name>
        <filter-
class>org.springframework.web.filter.Delegating
FilterProxy</filter-class>
        <init-param>
            <!-- Filter 配置 = FilterConfig -->
            <param-name>targetBeanName</param-
name>
            <param-value>someFilter</param-
value>
        </init-param>
    </filter>
```

- 生命周期实现
  - 初始化 - initFilterBean()
    - 判断 delegate 对象是否存在，如果不存在，执行下步
    - 判断 targetBeanName 是否存在，如果存在的话，通过名称+类型依赖查找 Filter Bean 实例
      - 获取 WebApplicationContext，依赖于 ContextLoaderListener 或者 DispatcherServlet
        - ContextLoaderListener Root WebApplicationContext
        - DispatcherServlet WebApplicationContext(Child)

- 注册方法
  - Servlet 标准方式（不会激活 Spring Bean 生命周期）
    - web.xml
    - 注解驱动（Servlet 3.0+）
    - API 编程（Servlet 3.0+）
  -

org.springframework.web.filter.GenericFilterBean

- 使用场景
  - 标准 Servlet 容器场景

        - Servlet 容器生命周期驱动 GenericFilterBean，利用标准 Filter 生命周期
            - 初始化 - init(FilterConfig)
            - 销毁 - destroy()
    - Spring IoC 容器场景
        - Spring Bean 生命周期
            - 初始化 - InitializingBean#afterPropertiesSet()
                - initFilterBean()
            - 销毁 - DisposableBean#destroy()
        - 举例：
            - ```
              @Bean
              public GenericFilterBean
              genericFilterBean() { // 如果它在
              Spring Boot Web，思考一下，会不会出现初始
              化调用两次？

              }
              ```
- 生命周期实现
    - BeanNameAware
    - EnvironmentAware
    - ServletContextAware
    - InitializingBean
        - afterPropertiesSet()
            - initFilterBean() - 模板方法
    - DisposableBean
              ```

    - EnvironmentCapable
- 实现特点
    - 通常扩展 GenericFilterBean 并非一个传统 Spring Bean，但是它需要 Spring Bean 生命周期。GenericFilterBean 的实现往往是通过 web.xml 文件或者 Servlet 3.0+ 注解或 API 注册
- 属性绑定
    - 将 FilterConfig 配置绑定到 GenericFilterBean 实现类的实例属性上

Spring Security 实现

FilterChainProxy

作为 DelegatingFilterProxy 中的 delegate Bean 对象，并且再次将 HTTP 请求委派给 SecurityFilterChain 所关联的多个 Filter。

它们之间的关系：

Servlet 容器 -> DelegatingFilterProxy -> FilterChainProxy -> N * SecurityFilterChain -> M * Filter

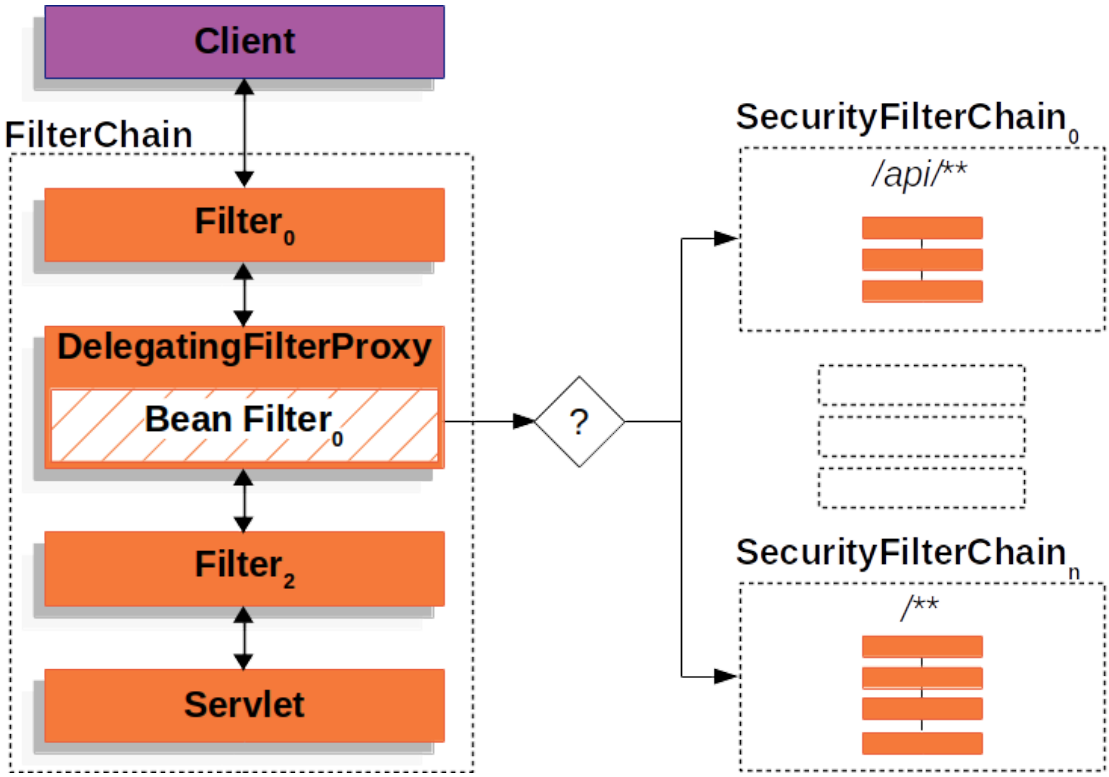
SecurityFilterChain 内部责任链实现，Filter 之间一定存在顺序执行，必然有优先次序。

> DelegatingFilterProxy 不一定为标准 Spring Bean，但是 FilterChainProxy 一定是 Spring 标准 Bean。同时，FilterChainProxy 扩展 GenericFilterBean 实现，意味着，FilterChainProxy 是通过 Spring Bean 生命周期来初始化依赖的组件。

SecurityFilterChain

被 FilterChainProxy 来委派

> 思考：FilterChainProxy 与 SecurityFilterChain 的关联关系？ 1:1 还是 1:N?
>
> 答： FilterChainProxy 与 SecurityFilterChain 数量关系是 1: N。SecurityFilterChain 关联了 M 个 Filter 实现。



接口定义

```
public interface SecurityFilterChain {

    boolean matches(HttpServletRequest
request);

    List<Filter> getFilters();

}
```

SecurityFilterChain 中关联的 M 个 Filter 实现，并且 FIlter 实现之间是有序（因为它通过 List 类型作为 getFilters() 方法返回值）

> 思考：SecurityFilterChain 中关联的 M 个 Filter 实现，是通过 Servlet 容器管理生命周期，还是通过 Spring IoC 容器?

标准实现

org.springframework.security.web.DefaultSecurityFilterChain

- 关联对象
  - org.springframework.security.web.util.matcher.RequestMatcher
  - List -> Filter
- 构建来源

- org.springframework.security.config.annotation.web.builders.HttpSecurity
  - `SecurityBuilder<DefaultSecurityFilterChain>`

Security Filters

被安插到 SecurityFilterChain 之中，并且内建实现存在执行优先次序，通过 API org.springframework.security.config.annotation.web.builders.FilterComparator 控制。

SecurityBuilder 设计模式

- 构建类型

通过泛型参数来决定构建类型，比如：

org.springframework.security.config.annotation.web.builders.HttpSecurity -> DefaultSecurityFilterChain

org.springframework.security.config.annotation.web.builders.WebSecurity -> Filter

- 构建方法
  - org.springframework.security.config.annotation.AbstractSecurityBuilder#build() 方法（唯一实现）

- 模板方法 org.springframework.security.config.annotation.AbstractSecurityBuilder#doBuild()
  - 生命周期 - org.springframework.security.config.annotation.AbstractConfiguredSecurityBuilder
    - 初始化前（模板方法） - beforeInit()
    - 初始化 - init()
      - 迭代 SecurityConfigurer#init(SecurityBuilder) 方法
    - 配置前（模板方法） - beforeConfigure()
    - 配置 - configure()
      - 迭代 SecurityConfigurer#configure(SecurityConfigurer) 方法
      - 其中被迭代 SecurityConfigurer 对象大多来源于 WebSecurityConfigurerAdapter Bean
    - 构建（模板方法） - performBuild()

在 Servlet 技术栈，Spring Security 通过 FilterChainProxy 创建两类 SecurityBuilder，一个是 HttpSecurity，另外一个是 WebSecurity。请注意，是两类，不是两个。

如果是两个的话，那么所有的 SecurityConfigurer Bean 或对象会共享 HttpSecurity 和 WebSecurity 实例。

实际上，HttpSecurity 和 WebSecurity 并非 Spring 单例类型的 Bean。

配置 Spring Security 特性

两个核心配置 API：

- WebSecurity
- HttpSecurity

认证和授权

org.springframework.security.authentication.AuthenticationManager

相关内容

软件开发的原则

1. 最小依赖化

- 依赖第三方 jar
- 依赖 JDK
- 依赖 JVM
- 依赖 OS

如果应用需要仅使用第三方的某一个或极少数类，将这个类复制到当前应用工程中，如：StringUtils

Spring Stack 三大方向

- 传统 Java EE 技术栈（稳定）
  - JDBC、JMS、JPA 等等
- Reactive 技术栈（不稳定）
  - Reactive(Reactor) + Java EE
    - Mono、Flux
- Cloud-Native 技术栈（趋势）
  - Micro-Services
    - Spring Boot
    - Spring Cloud
  - Function

        - Spring Cloud Function
        - Spring Cloud Stream
    - Native
        - Spring Native

作业

如何解决多个 WebSecurityConfigurerAdapter Bean 配置相互冲突的问题?

提示：假设有两个 WebSecurityConfigurerAdapter Bean 定义，并且标注了不同的 @Order，其中一个关闭 CSRF，一个开启 CSRF，那么最终结果如何确定?

背景：Spring Boot 场景下，自动装配以及自定义 Starter 方式非常流行，部分开发人员掌握了 Spring Security 配置方法，并且自定义了自己的实现，解决了 Order 的问题，然而会出现不确定配置因素。