6-生命周期内的拦截过滤与监听

1.Servlet域对象与属性变化监听

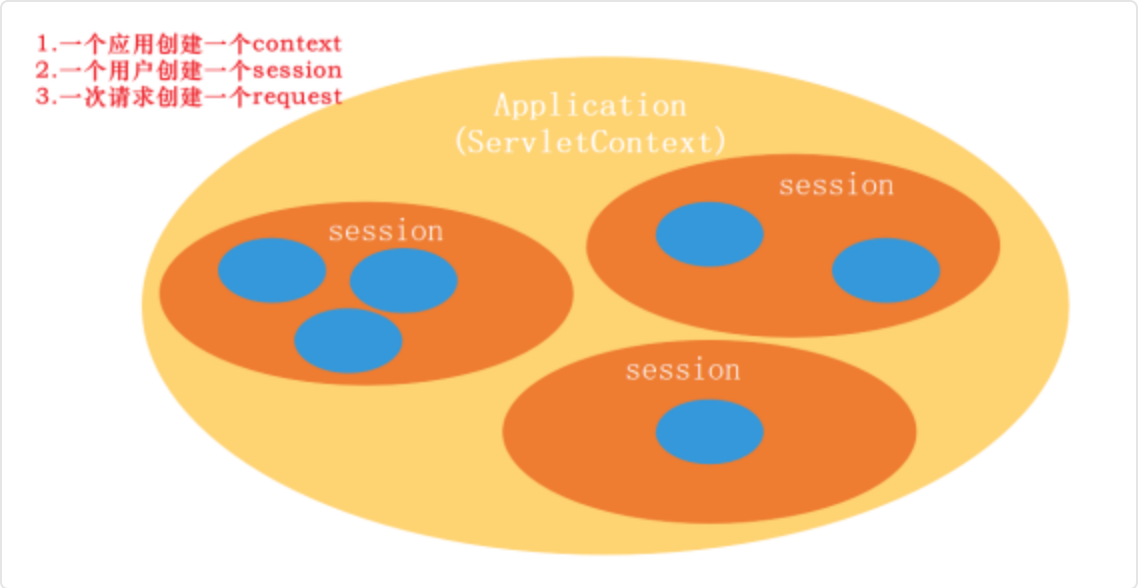
一、监听器定义与实现

1.1 定义

Servlet 监听器是 Servlet 规范中定义的一种特殊类，用于监听 ServletContext、HttpSession 和 ServletRequest 等作用域对象的创建与销毁事件，以及监听这些作用域对象中属性发生修改的事件。监听器使用了设计模式中的观察者模式，它关注特定事物的创建、销毁以及变化并做出回调动作，因此监听器具有**异步**的特性。

Servlet Listener 监听三大域对象的创建和销毁事件，三大对象分别是：

1. ServletContext Listener：application 级别，整个应用只存在一个，所有用户使用一个 ServletContext
2. HttpSession Listener：session 级别，同一个用户的浏览器开启与关闭生命周期内使用的是同一个 session
3. ServletRequest Listener：request 级别，每一个HTTP请求为一个request



除了监听域对象的创建和销毁，还可以监听域对象中属性发生修改的事件。

- HttpSessionAttributeListener
- ServletContextAttributeListener
- ServletRequestAttributeListener

1.2 使用场景

Servlet 规范设计监听器的作用是在事件发生前、发生后进行一些处理，一般可以用来统计在线人数和在线用户、统计网站访问量、系统启动时初始化信息等。

1.3 监听器的实现

Java | 复制代码

```java
@Slf4j
@WebListener
public class CustomListener implements  ServletContextListener,
                                        ServletRequestListener,
                                        HttpSessionListener,
                                        ServletRequestAttributeListener {

    @Override
    public void contextInitialized(ServletContextEvent se) {
        log.info("==============context创建");
    }

    @Override
    public void contextDestroyed(ServletContextEvent se) {
        log.info("==============context销毁");
    }


    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        log.info(" ++++++++++++++++++request监听器：销毁");
    }

    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        log.info(" ++++++++++++++++++request监听器：创建");
    }


    @Override
    public void sessionCreated(HttpSessionEvent se) {
        log.info("----------------session创建");
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        log.info("----------------session销毁");
    }

    public void attributeAdded(ServletRequestAttributeEvent srae) {

        log.info("----------------attributeAdded");
    }

    public void attributeRemoved(ServletRequestAttributeEvent srae) {
```

```
46              log.info("----------------attributeRemoved");
47          }
48
49          public void attributeReplaced(ServletRequestAttributeEvent srae) {
50              log.info("----------------attributeReplaced");
51          }
52
53      }
```

- 实现**ServletRequestListener接口**，并重写requestDestroyed销毁和requestInitialized方法。一次ServletRequest的requestInitialized方法和requestDestroyed销毁方法的执行代表1次请求的接收与处理完毕。所以比较适合网站资源被访问次数的统计。
- 实现**HttpSessionListener接口**，并重写sessionInitialized初始化和sessionDestroyed销毁方法，可以监听session会话的开启与销毁（用户的上线与下线）。比如：可以用来实现在线用户数量的统计。
- 实现**ServletContextListener接口**，并重写contextInitialized初始化和contextDestroyed销毁方法，可以监听全局应用的初始化和销毁。比如：在系统启动的时候，初始化一些数据到内存中供后续使用。
- 实现**ServletRequestAttributeListener接口**（或HttpSessionAttributeListener或ServletContextAttributeListener）。可以监听到对应的作用域内数据属性的attributeAdded新增、attributeRemoved删除、attributeReplaced替换等动作。

1.4.全局Servlet组件扫描注解

在启动类中加入@ServletComponentScan进行自动注册即可。

二、监听器测试

定义如下的Controller进行访问测试：

```java
@RestController
public class TestController {
    @GetMapping("/hello")
    public String hello(HttpServletRequest request, HttpSession session)
{
        //操作 request 的 attribute
        request.setAttribute("a", "a");
        request.setAttribute("a", "b");
        request.getAttribute("a");
        request.removeAttribute("a");

        //操作 session 的 attribute
        session.setAttribute("a", "a");
        session.getAttribute("a");
        session.invalidate();

        return "hello world---";
    }
}
```

- 当应用启动的时候。“==========context创建”被打印出来，说明触发contextInitialized监听函数
- 访问“http://localhost:8888/hello”，“ ++++++++++++++++++request监听器：创建”被打印出来，说明requestInitialized回调函数被触发
- 紧接着“----------session创建”被打印出来，说明sessionCreated监听函数被触发
- 继续执行request.setAttribute("a", "a");，“----------attributeAdded”被打印出来，说明attributeAdded监听函数被触发
- 继续执行request.setAttribute("a", "b");“----------attributeReplaced”被打印出来，说明attributeReplaced监听函数被触发
- 继续执行完成request.removeAttribute("a");“----------attributeRemoved”被打印出来，说明attributeRemoved监听函数被触发
- 继续执行session.invalidate();，“----------session销毁”被打印出来，说明sessionDestroyed监听函数被触发
- 将controller方法执行完成，"++++++++++++++++++request监听器：销毁"被打印出来，说明requestDestroyed监听函数被触发。
- 当停掉应用的时候，”==========context销毁”被打印出来，说明contextDestroyed监听函

数被触发

从上面的打印结果看：作用域范围是context 大于 request 大于sesion，实际上并不是。因为我们手动调用了session.invalidate();，session才会被销毁。正常情况下session的销毁是由servlet容器根据session超时时间等因素来控制的。

所以正常的作用域生命周期 ServletContext > HttpSession > request

> 在以上的监听测试中，会有一些多余的监听日志被打印，是SpringBoot系统默认帮我们做一些属性的添加与删除设置，从而触发监听，可以忽略掉。

2.Servlet过滤器的实现

一、过滤器

1.1 定义

Servlet 过滤器是可用于 Servlet 编程的 Java 类，目的：

- 在客户端的请求访问后端资源之前，拦截这些请求。
- 在服务器的响应发送回客户端之前，处理这些响应。

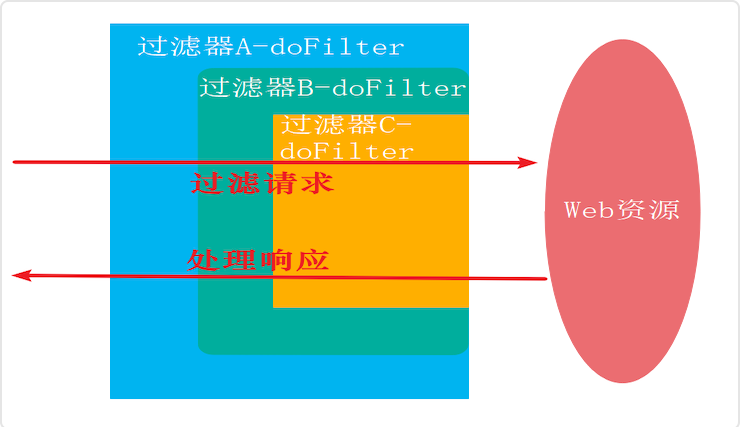
1.2 使用场景

在实际的应用开发中，我们经常使用过滤器做以下一些事情：

- 基于一定的授权逻辑，对HTTP请求进行过滤，从而保证数据访问的安全。比如：判断请求的来源IP是否在系统黑名单中
- 对于一些经过加密的HTTP请求数据，进行统一解密，方便后端资源进行业务处理
- 或者我们社交应用经常需要的敏感词过滤，也可以使用过滤器，将触发敏感词的非法请求过滤掉

过滤器主要的特点在于：**一是可以过滤所有请求，二是它能够改变请求的数据内容。**

1.3 过滤器的实现



注册方式一:利用WebFilter注解配置

@WebFilter 是 Servlet3.0新增的注解，原先实现过滤器，需要在web.xml中进行配置，而现在通过此注解，启动启动时会自动扫描自动注册。

编写Filter类：

```java
//注册器名称为customFilter,拦截的url为所有
@WebFilter(filterName="customFilter",urlPatterns={"/*"})
@Slf4j
public class CustomFilter implements Filter{

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("filter 初始化");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        log.info("customFilter 请求处理之前----doFilter方法之前过滤请求");
        //对request、response进行一些预处理
        //链路 直接传给下一个过滤器
        chain.doFilter(request, response);

        log.info("customFilter 请求处理之后----doFilter方法之后处理响应");
    }

    @Override
    public void destroy() {
        log.info("filter 销毁");
    }
}

```

然后**在启动类加入@ServletComponentScan注解**即可。

使用这种方法当注册多个过滤器时，无法指定过滤器的先后执行顺序。原本使用web.xml配置过滤器时，是可指定执行顺序的，但使用@WebFilter时，没有这个配置属性的(需要配合@Order进行)，所以接下来介绍下通过FilterRegistrationBean进行过滤器的注册。

> **一小技巧一**
>
> **通过对过滤器名称的指定，进行顺序的约定，比如LogFilter和AuthFilter，此时AuthFilter就会比LogFilter先执行，因为首字母A比L排序靠前。**

注册方式二：FilterRegistrationBean方式

FilterRegistrationBean是SpringBoot提供的，此类提供**setOrder方法，可以为filter设置排序值**，让Spring在注册WebFilter之前排序后再依次注册。

```java
@Configuration
public class FilterRegistration {
    @Bean
    public FilterRegistrationBean filterRegistrationBean() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        //Filter可以new，也可以使用依赖注入Bean
        registration.setFilter(new CustomFilter());
        //过滤器名称
        registration.setName("customFilter");
        //拦截路径
        registration.addUrlPatterns("/*");
        //设置顺序
        registration.setOrder(10);
        return registration;
    }
}
```

要注册多个过滤器，就注册多个FilterRegistrationBean即可。启动后效果和第一种是一样的。可以访问应用内的任意资源进行过滤器测试。

二、servlet

2.1定义：

Java程序员十几年前做Web开发的时候，所有的请求都是由Servlet来接受并响应的。每来一个请求，就要写一个Servlet。

这种方式很麻烦，大家就在想能不能根据请求的路径以及参数不同，映射到不同的方法上去执行，这样就可以在一个Servlet类里面处理多个请求，每个请求就是一个方法。这个思想后来就逐渐发展为struts、SpringMVC等框架。

2.2使用场景

目前来看，Servlet使用的场景已经被SpringMVC封装架构全面覆盖，几乎没有什么需要使用原始Servlet进行开发的场景。但是不排除，老项目向SpringBoot项目迁移融合，需要支持Servlet的情况，作为基础也是有必要的。

2.3 实现

我们来看一下在SpringBoot里面如何实现Servlet的编写和使用。

```java
@WebServlet(name = "firstServlet", urlPatterns = "/firstServlet")
@Slf4j
public class FirstServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        log.info("firstServlet");
        resp.getWriter().append("firstServlet");
    }

}
```

然后在启动类加入@ServletComponentScan注解即可。

3.Spring拦截器及请求链路说明

一、拦截器Interceptor

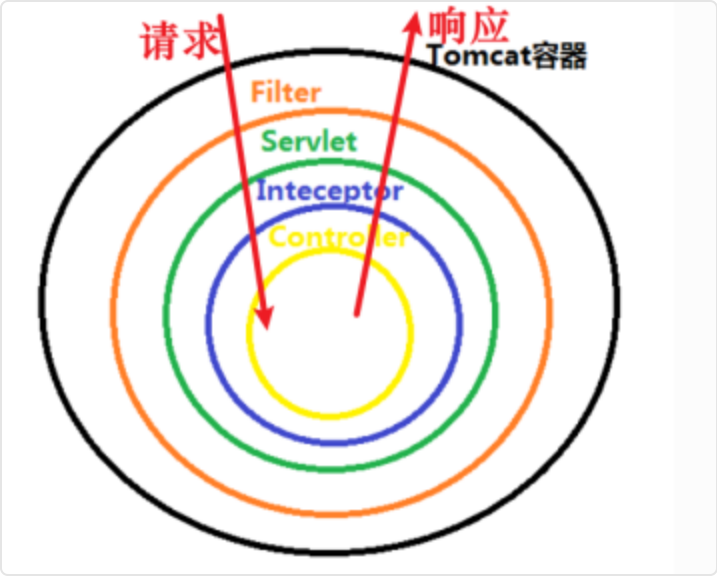
在 Servlet 规范中并没有拦截器的概念，它是在Spring框架内衍生出来的。

Spring中拦截器有三个方法：

- preHandle 表示被拦截的URL对应的控制层方法，执行前的自定义处理逻辑
- postHandle 表示被拦截的URL对应的控制层方法，执行后的自定义处理逻辑，此时还未将modelAndView进行页面渲染。
- afterCompletion 表示此时modelAndView已做页面渲染，执行拦截器的自定义处理。

二、拦截器与过滤器的核心区别

从请求处理的生命周期上看，拦截器Interceptor和过滤器filter的作用是类似的。过滤器能做的事情，拦截器几乎也都能做。



但是二者使用场景还是有一些区别的：

- 规范不同：Filter是在Servlet规范中定义的组件，在servlet容器内生效。而拦截器是Spring框架支

持的，在Spring 上下文中生效。

- 拦截器可以获取并使用Spring IOC容器中的bean，但过滤器就不行。因为过滤器是Servlet的组件，而IOC容器的bean是Spring框架内使用，拦截器恰恰是Spring框架内衍生出来的。
- 拦截器可以访问Spring上下文值对象，如ModelAndView，过滤器不行。基于与上一点同样的原因。
- 过滤器在进入servlet容器之前处理请求，拦截器在servlet容器之内处理请求。过滤器比拦截器的粒度更大，比较适合系统级别的所有API的处理动作。比如：权限认证，Spring Security就大量的使用了过滤器。
- 拦截器相比于过滤器粒度更小，更适合分模块、分范围的统一业务逻辑处理。比如：分模块的、分业务的记录审计日志。

比如：我们在Filter中使用注解，注入一个测试service，结果为null。因为过滤器无法使用Spring IOC容器bean。

三、拦截器的实现

编写自定义拦截器类，此处用一个简单的例子让大家了解拦截器的生命周期。

Java | 复制代码

```java
package com.mqxu.boot.interceptor;

import com.mqxu.boot.service.TestService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.annotation.Resource;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @description: 自定义拦截器类，了解拦截器的生命周期
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Slf4j
@Component
public class CustomHandlerInterceptor implements HandlerInterceptor {

    @Resource
    private TestService testService;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        log.info("preHandle:请求前调用");
        log.info(testService.test());
        //返回 false 则请求中断
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        log.info("postHandle:请求后调用");

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
```

```
41            log.info("afterCompletion:请求调用完成后回调方法，即在视图渲染完成后回调");
42
43        }
44
45    }
```

实现WebMvcConfigurer接口完成拦截器的注册。

Java | 复制代码

```
1   package com.mqxu.boot.interceptor;
2
3   import org.springframework.context.annotation.Configuration;
4   import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
5   import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
6
7   import javax.annotation.Resource;
8
9   /**
10   * @description: 注册拦截器,废弃: public class MyWebMvcConfigurer extends WebMvcConfigurerAdapter
11   * @author: mqxu
12   * @date: 2022-04-04
13   **/
14  @Configuration
15  public class MyWebMvcConfigurer implements WebMvcConfigurer {
16
17      private final String[] excludePath = {"/static"};
18
19      @Resource
20      private CustomHandlerInterceptor customHandlerInterceptor;
21
22      @Override
23      public void addInterceptors(InterceptorRegistry registry) {
24          //注册拦截器 拦截规则
25          //多个拦截器时 以此添加 执行顺序按添加顺序
26
        registry.addInterceptor(customHandlerInterceptor).addPathPatterns("/**").excludePathPatterns(excludePath);
27      }
28  }
```

如果我们在CustomHandlerInterceptor ， 注入一个测试service，结果是可以正确依赖注入并使用该Service的。

```
@Slf4j
@Component
public class CustomHandlerInterceptor implements HandlerInterceptor {

@Resource
TestBeanService testBeanService;  testBeanService: TestBeanService@7888
```

四、请求链路说明

随便请求一个系统内的API（因为我们配置的过滤器拦截器拦截所有请求），通过输出结果分析一下拦截器、过滤器中各接口函数的执行顺序。

Plain Text | 复制代码

```
CustomFilter  : customFilter 请求处理之前----doFilter方法之前过滤请求
CustomHandlerInterceptor  : preHandle:请求前调用
CustomHandlerInterceptor  : postHandle:请求后调用
CustomHandlerInterceptor  : afterCompletion:请求调用完成后回调方法，即在视图渲染完成后回调
CustomFilter  : customFilter 请求处理之后----doFilter方法之后处理响应
```

请求链路调用顺序图如下所示：

4.自定义事件的发布与监听

一、事件监听介绍:

1.1.事件监听的角色

首先我们要理解事件监听中需要的几个角色

- 事件发布者 （即事件源）
- 事件监听者
- 事件本身

1.2. 事件监听的使用场景

举一个简单的例子：比如居委会发布停水通知。居委会就是事件源、停水就是事件本身、该居委会的辖区居民就是事件监听者。这个例子，有这样几个特点：

- 异步处理：居委会工作人员发布通知之后，就可以去忙别的工作了，不会原地等待所有居民的反馈。
- 解耦：居委会和居民之间是解耦的，互相不干扰对方的工作状态与生活状态。
- 不规律性：停水事件发生频率是不规律的，触发规则相对随机。

当你在一个系统的业务需求中，满足上面的几个特点中的2点，就应该考虑使用事件监听机制实现业务需求。

实现事件监听机制有很多方法，比如：

- 使用消息队列中间件的发布订阅模式
- JDK自带的java.util.EventListener
- Spring环境下的实现事件发布监听的方法

二、代码具体实现

2.1.自定义事件

继承自ApplicationEvent抽象类，然后定义自己的构造器。

```java
/**
 * @description: 自定义事件:继承ApplicationEvent抽象类，并定义自己的构造器
 * @author: mqxu
 * @date: 2022-04-04
 **/
public class MyEvent extends ApplicationEvent {

    public MyEvent(Object source) {
        super(source);
    }
}
```

2.2.自定义事件监听器

springboot进行事件监听有四种方式

1.写代码向ApplicationContext中添加监听器

2.使用Component注解将监听器装载入spring容器

3.在application.properties中配置监听器

4.通过@EventListener注解实现事件监听

方式1

首先创建MyListener1类

```java
/**
 * @description: 自定义事件监听器方式1：实现ApplicationListener接口
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Slf4j
public class MyListener1 implements ApplicationListener<MyEvent> {
    @Override
    public void onApplicationEvent(MyEvent event) {
        log.info(String.format("%s 监听到事件源：%s.", MyListener1.class.getName(), event.getSource()));
    }
}
```

然后在SpringBoot应用启动类中获取ConfigurableApplicationContext上下文，装载监听

```java
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        //获取ConfigurableApplicationContext上下文
        ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);
        //装载监听
        context.addApplicationListener(new MyListener1());
    }
}
```

方式2（推荐）

创建MyListener2类，并使用@Component注解将该类装载入spring容器中

```java

/**
 * @description: 自定义事件监听器方式2：使用@Component注解将该类装载入spring容器中
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Component
@Slf4j
public class MyListener2 implements ApplicationListener<MyEvent> {

    @Override
    public void onApplicationEvent(MyEvent event) {
        log.info(String.format("%s 监听到事件源：%s.", MyListener2.class.getName(), event.getSource()));
    }

}
```

方式3

首先创建MyListener3类

```java
/**
 * @description: 自定义事件监听器方式3：在application.properties中配置监听
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Slf4j
public class MyListener3 implements ApplicationListener<MyEvent> {

    @Override
    public void onApplicationEvent(MyEvent event) {
        log.info(String.format("%s 监听到事件源：%s.", MyListener3.class.getName(), event.getSource()));
    }
}
```

然后在application.yml中配置监听

```yaml
context:
listener:
    classes: com.mqxu.boot.listener.MyListener3
```

方式4（推荐）

创建MyListener4类，该类无需实现ApplicationListener接口，使用@EventListener装饰具体方法

YAML | 复制代码

```yaml
/**
 * @description: 自定义事件监听器方式4：使用@EventListener装饰具体方法
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Slf4j
@Component
public class MyListener4 {
    @EventListener
    public void listener(MyEvent event) {
        log.info(String.format("%s 监听到事件源：%s.", MyListener4.class.getName(), event.getSource()));
    }
}
```

三、测试监听事件的发布

有了applicationContext，想在哪发布事件就在哪发布事件，我们在启动主类发布事件

Java | 复制代码

```java
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        //获取ConfigurableApplicationContext上下文
        ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);
        //装载监听
        context.addApplicationListener(new MyListener1());
        // 发布事件
        context.publishEvent(new MyEvent("测试事件"));
    }
}
```

启动后，日志打印如下。（下面截图是在启动类发布事件后的截图，在单元测试里面监听器1监听不到，执行顺序问题）：

```
16:31:16 [restartedMain] [INFO ] c.m.b.l.MyListener3.onApplicationEvent - com.mqxu.boot.listener.MyListener3 监听到事件源：测试事件.
16:31:16 [restartedMain] [INFO ] c.m.b.l.MyListener2.onApplicationEvent - com.mqxu.boot.listener.MyListener2 监听到事件源：测试事件.
16:31:16 [restartedMain] [INFO ] c.m.b.l.MyListener4.listener - com.mqxu.boot.listener.MyListener4 监听到事件源：测试事件.
16:31:16 [restartedMain] [INFO ] c.m.b.l.MyListener1.onApplicationEvent - com.mqxu.boot.listener.MyListener1 监听到事件源：测试事件.
```

由日志打印可以看出，SpringBoot四种事件的实现方式监听是有序的。无论执行多少次都是这个顺序。

5.应用启动的监听

一、简介

SpringBoot提供了两个接口：CommandLineRunner、ApplicationRunner，用于启动应用时做特殊处理，这些代码会在SpringApplication的run()方法运行完成之前被执行。

相对于之前介绍的Spring的ApplicationListener接口自定义监听器、Servlet的ServletContextListener监听器。

使用二者的好处在于，可以方便地使用应用启动参数，根据参数不同做不同的初始化操作。

二、常用场景介绍

实现CommandLineRunner、ApplicationRunner接口，通常用于应用启动前的特殊代码执行，比如：

- 将系统常用的数据加载到内存
- 应用上一次运行的垃圾数据清理
- 系统启动成功后的通知的发送

三、小实验

通过@Component定义方式实现

CommandLineRunner：参数是字符串数组

```java
@Component
@Slf4j
public class CommandLineStartupRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        log.info("CommandLineStartupRunner传入参数: {}",
Arrays.toString(args));
    }
}
```

ApplicationRunner：参数被放入ApplicationArguments，通过getOptionNames()、getOptionValues()、getSourceArgs()获取参数

```java
@Component
@Slf4j
public class AppStartupRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) {
        log.info("ApplicationRunner参数名称: {}", args.getOptionNames());
        log.info("ApplicationRunner参数值: {}",
args.getOptionValues("age"));
        log.info("ApplicationRunner参数: {}",
Arrays.toString(args.getSourceArgs()));
    }
}
```

通过@Bean定义方式实现

这种方式可以指定执行顺序，注意前两个Bean是CommandLineRunner，最后一个Bean是ApplicationRunner 。

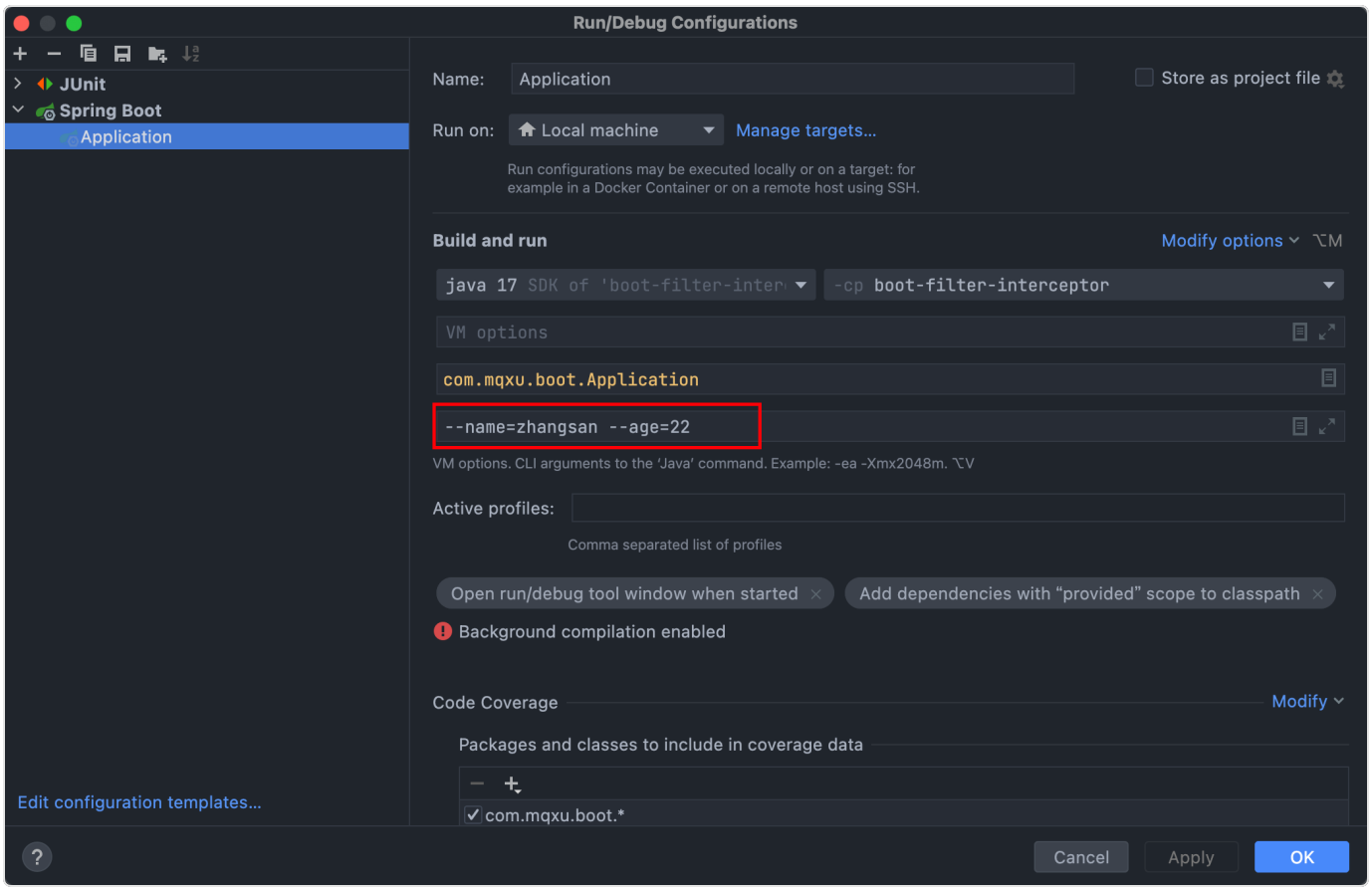
Java | 复制代码

```java
package com.mqxu.boot.runner;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;

import java.util.Arrays;

/**
 * @description:
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Configuration
@Slf4j
public class BeanRunner {
    @Bean
    @Order(1)
    public CommandLineRunner runner1() {
        return new CommandLineRunner() {
            @Override
            public void run(String... args) {
                log.info("BeanCommandLineRunner run1()" +
Arrays.toString(args));
            }
        };
    }

    @Bean
    @Order(2)
    public CommandLineRunner runner2() {
        return new CommandLineRunner() {
            @Override
            public void run(String... args) {
                log.info("BeanCommandLineRunner run2()" +
Arrays.toString(args));
            }
        };
    }

    @Bean
```

```
44        @Order(3)
45        public ApplicationRunner runner3() {
46            return new ApplicationRunner() {
47                @Override
48                public void run(ApplicationArguments args) {
49                    log.info("BeanApplicationRunner run3()" +
    Arrays.toString(args.getSourceArgs()));
50                }
51            };
52        }
53    }
```

四、执行测试

在启动配置中加入如下参数，保存后启动应用



测试输出结果：

```
16:34:46 [restartedMain] [INFO ] c.m.b.r.AppStartupRunner.run - ApplicationRunner参数名称: [name, age]
16:34:46 [restartedMain] [INFO ] c.m.b.r.AppStartupRunner.run - ApplicationRunner参数值: [22]
16:34:46 [restartedMain] [INFO ] c.m.b.r.AppStartupRunner.run - ApplicationRunner参数: [--name=zhangsan, --age=22]
16:34:46 [restartedMain] [INFO ] c.m.b.r.BeanRunner.run - BeanApplicationRunner run3()[--name=zhangsan, --age=22]
16:34:46 [restartedMain] [INFO ] c.m.b.r.CommandLineStartupRunner.run - CommandLineStartupRunner传入参数: [--name=zhangsan, --age=22]
16:34:46 [restartedMain] [INFO ] c.m.b.r.BeanRunner.run - BeanCommandLineRunner run1()[--name=zhangsan, --age=22]
16:34:46 [restartedMain] [INFO ] c.m.b.r.BeanRunner.run - BeanCommandLineRunner run2()[--name=zhangsan, --age=22]
```

从测试结果上看

- ApplicationRunner执行优先级高于CommandLineRunner
- 以Bean的形式运行的Runner优先级要低于Component注解加implements Runner接口的方式
- Order注解只能保证同类的CommandLineRunner或ApplicationRunner的执行顺序，不能跨类保证顺序

五、总结

CommandLineRunner、ApplicationRunner的核心用法是一致的，就是用于应用启动前的特殊代码执行。ApplicationRunner的执行顺序先于CommandLineRunner；ApplicationRunner将参数封装成了对象，提供了获取参数名、参数值等方法，操作上会方便一些。

6.类初始化监听

有些初始化动作，并不一定在应用初始化的时候进行，因为这个时候初始化，经常有些Bean还未形成对象，有些properties属性值还没完成注入，导致我们的初始化动作需要的一些必要条件没有准备好，所谓的初始化也就无法正确进行。

我们经常使用的一些初始化动作，可以在bean进行初始化的时候进行，如下代码：

Java | 复制代码

```java
package com.mqxu.boot.listener;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

/**
 * @description: 类初始化监听
 * @author: mqxu
 * @date: 2022-04-04
 **/
@Component
@Slf4j
public class BeanInitListener implements InitializingBean {
    static {
        log.info("类初始化静态代码块");
    }

    public BeanInitListener() {
        log.info("类初始化构造方法");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        log.info("类初始化 afterPropertiesSet 方法");
    }

    @PostConstruct
    void method() {
        log.info("类初始化 postConstruct 注解方法!");
    }
}
```

上面代码经常被我们使用到的是：

- postConstruct 注解方法，该注解所注释的方法会在Bean对象构建完成之后去执行
- 实现InitializingBean 接口的afterPropertiesSet方法， 通过这个方法名也可以知道该方法是在属性被设置之后执行。

将上面的代码类放入一个SpringBoot应用，并启动应用，执行顺序（日志输出顺序）如下:

```
16:34:46 [restartedMain] [INFO ] c.m.b.l.BeanInitListener.<clinit> - 类初始化静态代码块
16:34:46 [restartedMain] [INFO ] c.m.b.l.BeanInitListener.<init> - 类初始化构造方法
16:34:46 [restartedMain] [INFO ] c.m.b.l.BeanInitListener.method - 类初始化 postConstruct 注解方法!
16:34:46 [restartedMain] [INFO ] c.m.b.l.BeanInitListener.afterPropertiesSet - 类初始化 afterPropertiesSet 方法
```

结论:

- 静态代码块会在类加载器加载这个类时执行一次
- 非静态代码块会在这个类的构造方法被执行的时候执行,构造方法每被执行一次, 非静态代码块都会被执行一次(可以理解为非静态代码块的内容会被copy到构造方法内容的最前面)
- afterPropertiesSet方法和被@PostConstruct方法会在BeanInitTester实例被创建并且BeanInitTester类中的所有实例属性都被初始化之后执行. 而afterPropertiesSet方法会在被@PostConstruct方法标注的方法之后执行.
- 顺序: 静态代码块->非静态代码块->构造方法->@PostConstruct方法->afterPropertiesSet方法