链滴

Java 和 Spring 中的事件

作者：jchain

原文链接：https://ld246.com/article/1598976604075

来源网站：链滴

事件 简单来说就是发生了什么，然后争对这个发生的操作 怎么处理。比如 页面上有个 按钮，当点击时候，应该怎么处理。看下下面的代码

```
$('#btn').on('click',function(){
    console.log('btn click event')
})
```

通过js我们知道,当点击按钮时,会打印 btn click event 这么一句话。

但不点击时,什么也不会发生。

从上可以分析得到

1. 有 按钮 这么个东东, 发生的动作会作用在它身上， 我们称 这个 按钮为事件源

2. 有 点击(click) 这么个动作,作用在 事件源 上, 可以将这一过程 抽象为事件对象。发生的事件中要含事件源，这样可以在对事件处理的操作中获取到事件源

3. 有 console.log() 这么个操作,这表示发生了这个事件 应该怎么处理的操作，我们将这一过程抽象为件监听

即某个 事件监听器（一个函数或者方法） 会一直监视着某个 事件源（比如按钮）的某个 操作(比如击)

当发生相应的动作后，这个监听器就会执行相应的操作。

需要注意的是：

1. 事件对象中 包含事件源 ， 也就说 事件源是事件对象的一个属性

2. 监听器 需要注册到 事件源上

Java中的事件

在Java中

- 事件对象 被抽象到 java.util.EventObject , 这个对象中包含一个 Object 的属性，也即 事件源 (扩展)
- 事件监听器 被抽象到 java.util.EventListener 中 ， 这是一个标识接口，需要自己定义监听方法
- 事件源 需要自己定义

在Java中 模拟 按钮点击事件 如下

事件对象：

```
public class ClickEventObject extends EventObject {

    /**
     *  这个msg 可随意扩展
     */
    private String msg;

    public ClickEventObject(Object source, String msg) {
        super(source);
        this.msg = msg;
```

```
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

事件监听器

```
public interface ClickEventListener extends EventListener {

    /**
     * 事件处理 (我这里是接口)
     * @param eventObject 事件对象
     */
    void handleEvent(EventObject eventObject);
}
```

事件源

```
public class Button {

    private ClickEventListener clickEventListener;

    /**
     * 注册 监听器
     * 我这里只是用了单个 EventListener ,你可以改成 List<EventListener> 这样一个事件源就可以
册多个监听器了
     * @param clickEventListener 点击事件监听器
     */
    public void registerListener(ClickEventListener clickEventListener) {
        this.clickEventListener = clickEventListener;
    }

    /**
     * 触发事件
     */
    public void triggerClick() {
        // this 表示事件源本身 这样也就是button
        ClickEventObject clickEventObject = new ClickEventObject(this,"点击事件处理");
        this.clickEventListener.handleEvent(clickEventObject);
    }
}
```

测试

```
public static void main(String[] args) {
    // 事件源
    Button button = new Button();

    // 注册监听器
```

```
    button.registerListener(new ClickEventListener() {
        // 实例化 监听器处理
        @Override
        public void handleEvent(EventObject eventObject) {
            if (eventObject instanceof ClickEventObject) {
                System.out.println(((ClickEventObject) eventObject).getMsg());
            }
        }
    });

    // 出发点击事件
    button.triggerClick();
}
```

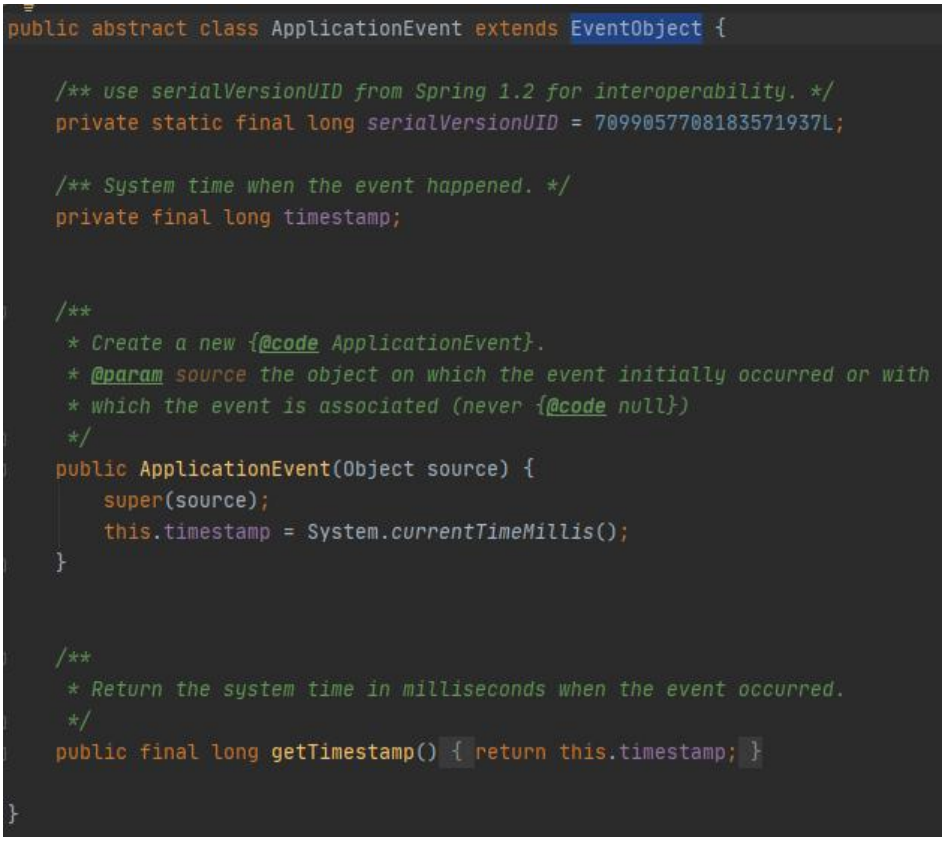
可以看到

- 事件对象只是关联事件源和事件监听器的中间对象，它其中包含 事件源
- 事件监听器需要注册到事件源上

Spring中的事件

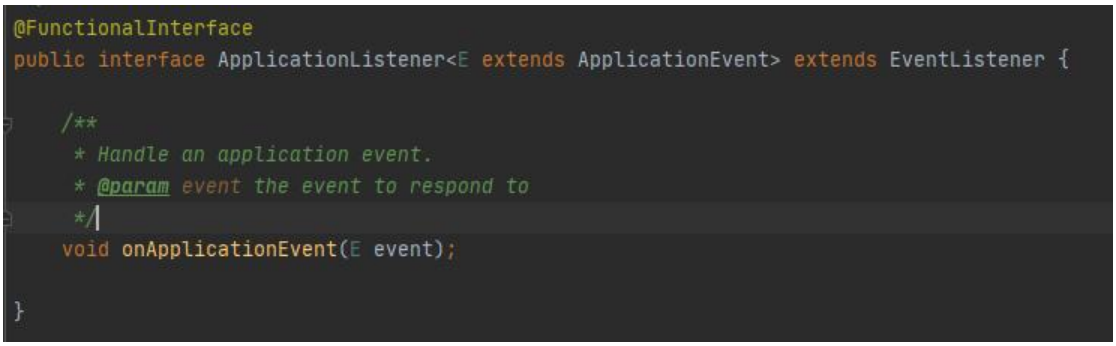
在spring的事件中 其实也是扩展了 java中的 EventListener 和 EventObject

ApplicationEvent 直接继承了 EventObject



```
public abstract class ApplicationEvent extends EventObject {

    /** use serialVersionUID from Spring 1.2 for interoperability. */
    private static final long serialVersionUID = 7099057708183571937L;

    /** System time when the event happened. */
    private final long timestamp;


    /**
     * Create a new {@code ApplicationEvent}.
     * @param source the object on which the event initially occurred or with
     * which the event is associated (never {@code null})
     */
    public ApplicationEvent(Object source) {
        super(source);
        this.timestamp = System.currentTimeMillis();
    }


    /**
     * Return the system time in milliseconds when the event occurred.
     */
    public final long getTimestamp() { return this.timestamp; }

}
```

ApplicationListener 直接继承了 EventListener ,并且定义了 onApplicationEvent 方法，需要注意里使用了泛型指定了特定的 EventObject



```java
@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {

    /**
     * Handle an application event.
     * @param event the event to respond to
     */
    void onApplicationEvent(E event);

}
```

在spring中写一个 按钮hover 的事件如下

事件对象

```java
public class HoverEvent extends ApplicationEvent {
    /**
     * 自定义的消息， 这个可以随意扩展
     */
    private String message;

    public HoverEvent(Object source, String message) {
        super(source);
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

事件监听器

```java
@Component
public class HoverListener implements ApplicationListener<HoverEvent> {

    /**
     * 事件处理
     * @param hoverEvent hover事件
     */
    @Override
    public void onApplicationEvent(HoverEvent hoverEvent) {
        System.out.println(hoverEvent.getMessage());
    }
}
```

事件源

```java
@Component
public class Button {
    @Autowired
```

```
    ApplicationContext applicationContext;

    public void triggerHover(String message) {
        applicationContext.publishEvent(new HoverEvent(this,message));
    }
}
```

测试

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicati
nContext(Config.class);
    Button button = applicationContext.getBean(Button.class);
    button.triggerHover("Hover事件处理");
}
```

其中 Config.class 是spring 启动时的 配置类,这里直接将 事件源 和 事件监听器 交给spring管理

```
@ComponentScan("com.slarn.event.spring")
@Configuration
    public class Config {
}
```

从上面代码可以看到在 事件源 中，注入了 ApplicationContext，触发 triggerHover 后 ，将操作交了 applicationContext.publishEvent(EventObject) 方法

从这里可以猜测，applicationContext 是触发事件者，在 publishEvent 中必然有 调用 最终的事件理方法,即最终会调用 HoverListener 的 onApplicationEvent(EventObject) 方法

跟踪代码可以发现 其调用栈如下：

```
org.springframework.context.ApplicationEventPublisher#publishEvent(org.springframework.c
ntext.ApplicationEvent)
    --> org.springframework.context.support.AbstractApplicationContext#publishEvent(java.la
g.Object, org.springframework.core.ResolvableType)
      --> org.springframework.context.support.AbstractApplicationContext#getApplicationEven
Multicaster
       --> org.springframework.context.event.ApplicationEventMulticaster#multicastEvent(org.
pringframework.context.ApplicationEvent, org.springframework.core.ResolvableType)
         --> org.springframework.context.event.SimpleApplicationEventMulticaster#invokeListe
er
          -- > org.springframework.context.event.SimpleApplicationEventMulticaster#doInvokeL
stener
```

在 SimpleApplicationEventMulticaster#doInvokeListener 的方法中 可以看到 最终执行了 listener.nApplicationEvent(event); 方法，完成事件的回调。

到这里你可能会发现一个问题就是,之前说 事件监听器 是要注册到事件源上，但是在上面的 Button 中却没有这个操作，这是怎么回事呢？是在哪里注册的呢？

带着这个问题，我们重新看下上面的调用栈 ，最终是执行了 listener.onApplicationEvent(event) ，么这个 listener是怎么来的呢？

往上找发现有如下代码

```
org.springframework.context.event.SimpleApplicationEventMulticaster#multicastEvent()
```

```
@Override
public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType
 {
    ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
    Executor executor = getTaskExecutor();
    for (ApplicationListener<?> listener : getApplicationListeners(event, type)) {
        if (executor != null) {
            executor.execute(() -> invokeListener(listener, event));
        }
        else {
            invokeListener(listener, event);
        }
    }
}
```

由此可以看到 getApplicationListeners(event, type) 可以获取到所有的监听器，也可以看出来 一事件源上 可能有多个监听器。

这里就是 获取所有的监听器，然后遍历执行每个 listener的 onApplicationEvent 方法

继续跟踪 getApplicationListeners()，发现他是 org.springframework.context.event.AbstractApplicationEventMulticaster#getApplicationListeners() 在这个地方调用的 然后会发现 AbstractApplicationEventMulticaster 其实是 ApplicationEventMulticaster 的实现类,打开这个 ApplicationEventMuticaster 瞄一眼，发现了什么

```
public interface ApplicationEventMulticaster {

    /**
     * Add a listener to be notified of all events.
     * @param listener the listener to add
     */
    void addApplicationListener(ApplicationListener<?> listener);

    /**
     * Add a listener bean to be notified of all events.
     * @param listenerBeanName the name of the listener bean to add
     */
    void addApplicationListenerBean(String listenerBeanName);

    /**
     * Remove a listener from the notification list.
     * @param listener the listener to remove
     */
    void removeApplicationListener(ApplicationListener<?> listener);

    /**
     * Remove a listener bean from the notification list.
     * @param listenerBeanName the name of the listener bean to remove
     */
    void removeApplicationListenerBean(String listenerBeanName);

    /**
     * Remove all listeners registered with this multicaster.
     * <p>After a remove call, the multicaster will perform no action
```

```
     * on event notification until new listeners are registered.
     */
    void removeAllListeners();

    /**
     * Multicast the given application event to appropriate listeners.
     * <p>Consider using {@link #multicastEvent(ApplicationEvent, ResolvableType)}
     * if possible as it provides better support for generics-based events.
     * @param event the event to multicast
     */
    void multicastEvent(ApplicationEvent event);

    /**
     * Multicast the given application event to appropriate listeners.
     * <p>If the {@code eventType} is {@code null}, a default type is built
     * based on the {@code event} instance.
     * @param event the event to multicast
     * @param eventType the type of event (can be {@code null})
     * @since 4.2
     */
    void multicastEvent(ApplicationEvent event, @Nullable ResolvableType eventType);

}
```

其中定义了一个 void addApplicationListener(ApplicationListener<?> listener); 方法，看到这里会想到什么？ 通过名字我们可以猜测 这里有可能就是 注册监听器 的地方，为了验证，我们直接在这法的实现类上 打上一个断点

这个方法的实现类为

org.springframework.context.event.AbstractApplicationEventMulticaster#addApplicationListener

```
@Override
public void addApplicationListener(ApplicationListener<?> listener) {
    synchronized (this.retrievalMutex) {
        // Explicitly remove target for a proxy, if registered already,
        // in order to avoid double invocations of the same listener.
        Object singletonTarget = AopProxyUtils.getSingletonTarget(listener);
        if (singletonTarget instanceof ApplicationListener) {
            this.defaultRetriever.applicationListeners.remove(singletonTarget);
        }
        this.defaultRetriever.applicationListeners.add(listener);
        this.retrieverCache.clear();
    }
}
```

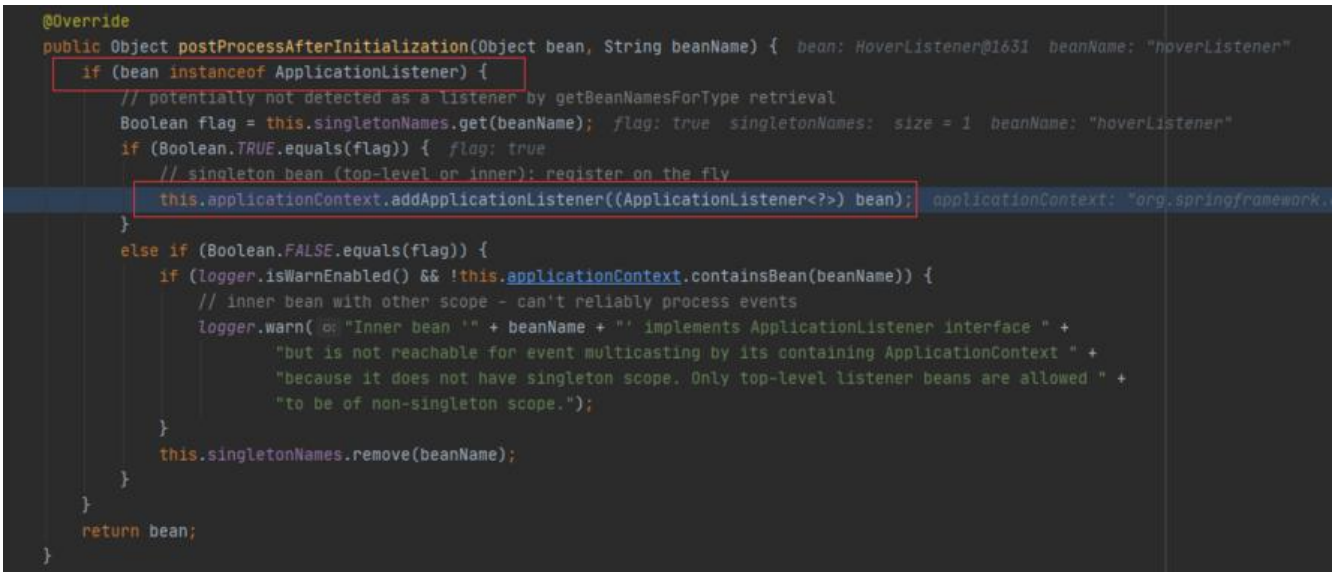
如果你对 spring bean 生命周期有一定了解的话，通过调试 跟踪调用栈 可以发现：

在 spring 初始化bean [实例化，填充属性之后] 的时候（也就是执行AbstractAutowireCapableBeanactory#initializeBean() 方法）

会在初始化方法调用之后 调用 applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);

在这个方法中 获取所有的后置处理器（BeanPostProcessor），然后执行每个后置处理器的 postProcessAfterInitialization()方法

其中有一个后置处理器为 ApplicationListenerDetector，在他的 postProcessAfterInitialization() 法中 有如下代码



最终在这个操作里面 把 类型为 ApplicationListener 的 listener 加入到 org.springframework.context.event.AbstractApplicationEventMulticaster.ListenerRetriever#applicationListeners 这个Set集中。

这样在发布（publishEvent）的时候 从这里获取 所有的listener 然后遍历调用 listener的 onApplicationEvent 就和上面对应起来了

你可能会问 这个 ApplicationListenerDetector 这个后置处理器 上面为什么能够获取到，其实 这是spring内部 手动注册的，注册的代码在 ApplicationContext刷新refresh()方式时

org.springframework.context.support.AbstractApplicationContext#registerBeanPostProcessor

--> org.springframework.context.support.PostProcessorRegistrationDelegate#registerBeanPostProcessors(org.springframework.beans.factory.config.ConfigurableListableBeanFactory, org.springframework.context.support.AbstractApplicationContext)

在这个最后一段代码为：

beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));

另外细心一下你会发现:

- applicationContext 的 publishEvent 方法 其实是 ApplicationEventPublisher 这个接口指定的法，表示发布事件 （事件触发者），在 AbstractApplicationContext 中对这方法 进行了实现
- ApplicationEvent 实现 有 ApplicationContextEvent,ContextRefreshedEvent 这个里面你会现 source 其实换成了 ApplicationContext，也就是说 ApplicationContext 就是事件源

总结一下：

1. spring 在实例化 bean 之前 会手动注册一个后置处理器 ApplicationListenerDetector
2. 在实例化 bean ， 填充属性 之后的 初始化bean中 initializeBean 方法的 postProcessAfterInitialization 方法中会获取 所有的后置处理器 遍历执行处理，其中就包括上面的 ApplicationListenerDetector
3. 在 ApplicationListenerDetector 中 会判断 bean 是否是 ApplicationListener，如果是的话就

这个bean(listener) 加入到org.springframework.context.event.AbstractApplicationEventMulticaster.ListenerRetriever#applicationListeners 这个Set集合中

4. AbstractApplicationEventMulticaster 这个类 是 AbstractApplicationContext 抽象类中的一个性（并且不为空，在refresh()方法的initApplicationEventMulticaster() 中初始化了一个SimpleApplicationEventMulticaster 的多播器），也就是说 ApplicationContext 有AbstractApplicationEventMlticaster，即也能能获取到所有的 listeners

5. ApplicationContext 继承了 ApplicationEventPublisher，同时 AbstractApplicationContext 也现了 ApplicationEventPublisher的 publishEvent

6. 在这个 publishEvent() 方法中就获取到 所有的 listeners 然后遍历 执行 listener.onApplicationEent(event);