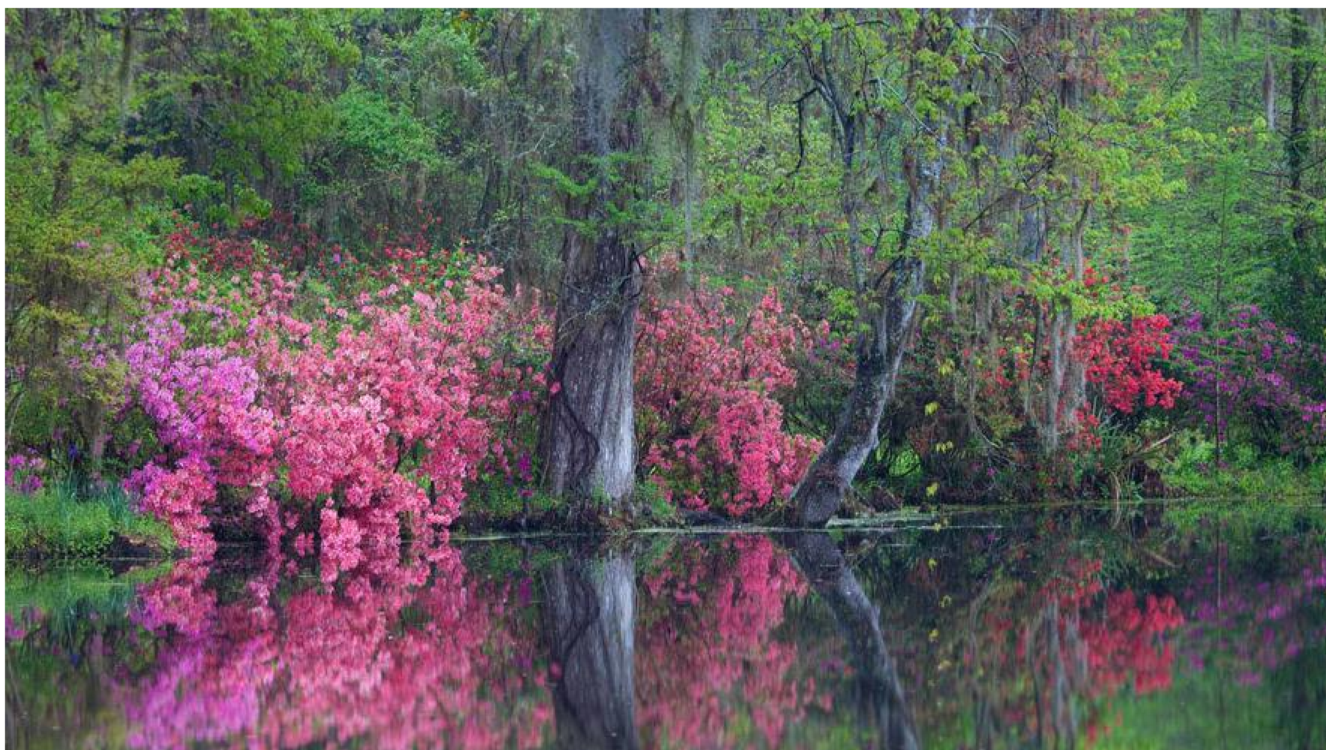
spring 事件监听机制的全方位分析

作者：gitzzzf
原文链接：https://ld246.com/article/1588826383859
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



一、引言

说起 Spring 的事件监听机制，我想用过 Spring 的小伙伴们都不会陌生，我们都了解 Spring 的事件制其实是观察者模式的一种体现。那么 Spring 的事件机制到底是如何实现的呢？实现的方式又有那呢？又是如何实现同步事件和异步事件的呢？下面将逐一揭开面纱。

二、Spring 事件机制的实现原理

1. Spring 事件机制实现的几个重要类

类名	描述
ApplicationEvent	继承自 java.util.EventObject Spring 事件的基类，是个抽象类
ApplicationEventPublisher	事件发布者，封装事件发布功能，调用广播发布事件
ApplicationEventMulticaster	广播，持有观察（也就是 ApplicationListener）的集合，可以向集合类的观察者们通知事件的发生
ApplicationListener	观察者，接收到事件发生的况后，执行相关的业务逻辑， 继承自 java.util.EventListener

2. 一个例子

举个最简单的业务场景。

第一步，构建一个事件 DemoEvent 如下

```
public class DemoEvent extends ApplicationEvent{
```

```
    /**
     * 事件信息
     */
    private String message;

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

第二步，构建一个观察者，并实现监听到事件 DemoEvent 发生后的业务逻辑

```
@Component
public class DemoSpringEventListener implements ApplicationListener<DemoEvent>{

    @Override
    public void onApplicationEvent(DemoEvent event) {
        System.out.println("我已知晓事件发生");
    }
}
```

第三步，通过事件发布者，发布事件

```
/*
@Autowired
private ApplicationEventPublisher publisher;
*/
publisher.publishEvent(new DemoEvent(this, "demoEvent"));
```

这个时候我们即可在终端看到输出文字“我已知晓事件发生”，也就表明观察者已经接收到了 DemoE
ent 事件，并调用方法 onApplicationEvent(DemoEvent event)进行了业务处理。

那么第一个问题来了，发布者调用发布事件的方法后发生了什么?

好的，那我们就顺着方法 publishEvent 跟进去看看。

ApplicationEventPublisher#publishEvent(Object event)

=>

AbstractApplicationContext#publishEvent(Object event)

在这个方法的源码中我们发现这行代码，我们重点关注，

```
getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
```

这个 getApplicationEventMulticaster()其实就是拿到一个广播 ApplicationEventMulticaster，我
知道广播中持有观察者集合，那既然拿到了广播，那下一步是不是通知广播中持有的观察者集合们，
件 Event 发生了？我们接着方法往下看。

ApplicationEventMulticaster#multicastEvent(ApplicationEvent event, ResolvableType eventTy
e)，跟进 ApplicationEventMulticaster 的实现类 SimpleApplicationEventMulticaster 中 multicast
vent 方法。

```
    @Override
    public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventT
```

```
pe) {
        ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
        for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {
            Executor executor = getTaskExecutor();
            if (executor != null) {
                                // 重点关注
                executor.execute(() -> invokeListener(listener, event));
            }
            else {
                                // 重点关注
                invokeListener(listener, event);
            }
        }
    }
```

multicastEvent 方法中，我们发现主要调用 invokeListener 这个方法，我们再跟进去。

```
    protected void invokeListener(ApplicationListener<?> listener, ApplicationEvent event) {
        ErrorHandler errorHandler = getErrorHandler();
        if (errorHandler != null) {
            try {
                                // 重点关注
                doInvokeListener(listener, event);
            }
            catch (Throwable err) {
                errorHandler.handleError(err);
            }
        }
        else {
                        // 重点关注
            doInvokeListener(listener, event);
        }
    }
```

继续跟进 doInvokeListener 方法

```
    private void doInvokeListener(ApplicationListener listener, ApplicationEvent event) {
        try {
                        // 这个地方发现调用了我们实现业务逻辑onApplicationEvent方法
            listener.onApplicationEvent(event);
        }
        catch (ClassCastException ex) {
            // 非重点内容，这里省略
        }
    }
```

我们发现了我们在观察者中实现的业务逻辑的方法 onApplicationEvent( )被调用了，这样一个简单事件发布以及事件监听流程就完成了。

到这个地方你知道了其实重点是在广播类中，广播类去通知他所持有的观察者们去执行响应的业务逻。那么，你是否疑惑广播类是怎么持有这些观察者们的呢？这些观察者们是在什么时候加入进去的呢接下来我们解决这个问题。我们知道，Spring boot 服务启动的入口如下。

```
@SpringBootApplication
public class DemoApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

启动方法中主要调用 SpringApplication.run( )方法，跟随者 run( )方法我们进去看看。

SpringApplication.run()

=>

refreshContext(context) 我们发现这个方法，这个方法是 Spring boot 服务启动时候刷新 Spring 器上下文内容的核心方法

=>

refresh(context)

```
    @Override
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
                    // 非重点关注内容，省略
            try {

                        ......

                        // 重点关注，初始化ApplicationEventMulticaster
                // Initialize event multicaster for this context.
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                onRefresh();

                        // 重点关注
                // Check for listener beans and register them.
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                finishRefresh();
            }

            catch (BeansException ex) {
                // 非重点关注内容，省略
            }

            finally {
                // 非重点关注内容，省略
            }
        }
    }
```

在 Spring boot 启动方法 refresh 中，我们发现了我们重点关注的两个方法调用，第一个 initApplicationEventMulticaster()，也就是初始化了广播类，第二个是 registerListeners( )这个方法，其源码如，

```
    protected void registerListeners() {
        // Register statically specified listeners first.
        for (ApplicationListener<?> listener : getApplicationListeners()) {
                    // 重点关注
            getApplicationEventMulticaster().addApplicationListener(listener);
        }

        // Do not initialize FactoryBeans here: We need to leave all regular beans
        // uninitialized to let post-processors apply to them!
        String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false
);
        for (String listenerBeanName : listenerBeanNames) {
                    // 重点关注
            getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
        }

        // Publish early application events now that we finally have a multicaster...
        Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
        this.earlyApplicationEvents = null;
        if (earlyEventsToProcess != null) {
            for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
                getApplicationEventMulticaster().multicastEvent(earlyEvent);
            }
        }
    }
```

这个地方我们发现了将观察者 ApplicationListener 加入到广播中的方法，那么到这个地方我们也就道了观察者们是如何被加入到广播类中的了。

三、Spring 事件机制的注解实现形式

在上面一部分内容中，是最基础的事件监听实现形式，在观察者类中我们需要继承 ApplicationListener，并实现其 onApplicationEvent( )方法。其实还有一种基于注解的实现形式，使用起来也是简单方。

```
@Component
public class DemoEventListener {
     // 在业务处理方法上添加@EventListener注解即可
    @EventListener
    public void handleDemoEvent(DemoEvent event) {
        System.out.println("我已知晓事件发生");
    }
}
```

1. 了解@EventListener

通过上面的实例，我知道主要在对应的方法上加上@EventListener 注解，就可以实现监听对应的事类 DemoEvent。该注解的描述如下

```
 * Annotation that marks a method as a listener for application events.
...

 * <p>Processing of {@code @EventListener} annotations is performed via
```

```
 * the internal {@link EventListenerMethodProcessor} bean which gets
 * registered automatically when using Java config or manually via the
 * {@code <context:annotation-config/>} or {@code <context:component-scan/>}
 * element when using XML config.
...
```

2. 添加@EventListener 注解的方法如何实现监听事件

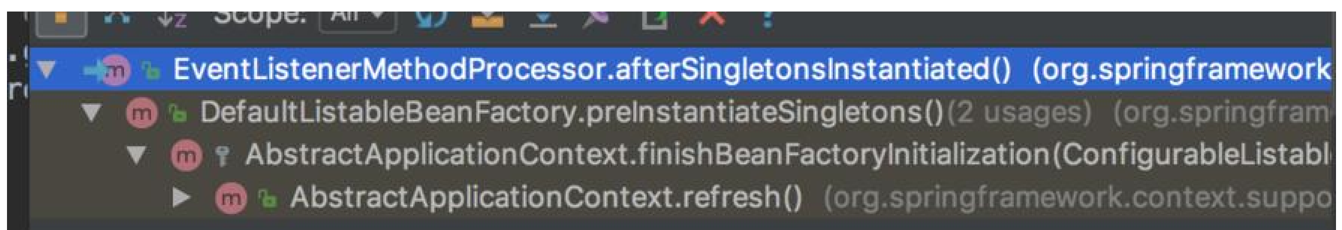
在注解的描述中我们知道它的处理类是 EventListenerMethodProcessor。

EventListenerMethodProcessor 中我们发现 afterSingletonsInstantiated( )方法，其核心方法为 Ev ntListenerMethodProcessor#processBean( )，其源码如下

```
  private void processBean(final String beanName, final Class<?> targetType) {
        ...

      Map<Method, EventListener> annotatedMethods = null;
      try {
                  // 重点关注，检查获取带有@EventListener注解的方法信息
        annotatedMethods = MethodIntrospector.selectMethods(targetType,
            (MethodIntrospector.MetadataLookup<EventListener>) method ->
                AnnotatedElementUtils.findMergedAnnotation(method, EventListener.clas
));
      }
      catch (Throwable ex) {
                 // 非重点关注，忽略

      }

      if (CollectionUtils.isEmpty(annotatedMethods)) {
        this.nonAnnotatedClasses.add(targetType);
        if (logger.isTraceEnabled()) {
          logger.trace("No @EventListener annotations found on bean class: " + targetType
getName());
        }
      }
      else {
        // Non-empty set of methods
        ConfigurableApplicationContext context = this.applicationContext;
        Assert.state(context != null, "No ApplicationContext set");
        List<EventListenerFactory> factories = this.eventListenerFactories;
        Assert.state(factories != null, "EventListenerFactory List not initialized");
        for (Method method : annotatedMethods.keySet()) {
          for (EventListenerFactory factory : factories) {
            if (factory.supportsMethod(method)) {
              Method methodToUse = AopUtils.selectInvocableMethod(method, context.
etType(beanName));
                                   // 重点关注，根据带有注解@EventListener的方法构建Applicat
onListener
              ApplicationListener<?> applicationListener =
                  factory.createApplicationListener(beanName, targetType, methodToUse
;
              if (applicationListener instanceof ApplicationListenerMethodAdapter) {
```

```
                    ((ApplicationListenerMethodAdapter) applicationListener).init(context, this
evaluator);
                }
                                        // 在广播类ApplicationEventMulticaster存在的情况下，将app
icationListener添加给广播类
                context.addApplicationListener(applicationListener);
                break;
            }
        }
    }
    if (logger.isDebugEnabled()) {
        logger.debug(annotatedMethods.size() + " @EventListener methods processed o
 bean '" +
                beanName + "': " + annotatedMethods);
    }
  }
 }
}
```

在这个方法中，发现带有@EventListener 注解的方法被构建成 ApplicationListenerMethodAdapte（实现了 ApplicationListener）并添加到了广播类中。再看 afterSingletonsInstantiated( )方法调链，



发现其也是在 Spring boot 服务启动的 refresh( )中被调用，那么带有@EventListener 注解的方法包装成 ApplicationListenerMethodAdapter 类并添加到广播类中这些操作在服务启动过程中就完了。那么在事件发布过程中，广播类一样会通知这些构建的 ApplicationListenerMethodAdapter 察者们，顺着第二章内容，也就知道最终会调用 onApplicationEvent( )方法，在 ApplicationListene MethodAdapter 中重写了该方法如下

```
@Override
public void onApplicationEvent(ApplicationEvent event) {
    processEvent(event);
}
```

跟进 processEvent( )方法

```
public void processEvent(ApplicationEvent event) {
    Object[] args = resolveArguments(event);
    if (shouldHandle(event, args)) {
                // 重点关注
        Object result = doInvoke(args);
        if (result != null) {
            handleResult(result);
        }
        else {
            logger.trace("No result object given - no result to handle");
        }
    }
}
```

跟进 doInvoke( )方法

```
    /**
     * Invoke the event listener method with the given argument values.
     */
    @Nullable
    protected Object doInvoke(Object... args) {
        Object bean = getTargetBean();
        ReflectionUtils.makeAccessible(this.method);
        try {
                    // 这个地方实现反射调用，在本实例中也就是最终调用了业务逻辑方法handleDemoE
ent( )
            return this.method.invoke(bean, args);
        }
        catch (IllegalArgumentException ex) {
                  // 非重点关注，忽略

        }
    }
```

至此，基于注解@EventListener 的观察者梳理完成。

四、异步事件的实现形式

1. 实现自定义 bean applicationEventMulticaster

前面几部分内容主要讲了 Spring 事件机制的原理，细心的小伙伴在对照源码看的过程中应该发现了
么一段内容。

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

也就是事件发布后，广播通知观察者事件发生的方法，其中 Executor executor = getTaskExecutor();
方法就是尝试获取线程池，如果有线程池设置，则在线程池中调用 invokeListener(listener, event);
法，异步实现事件的监听。默认情况线程池是没有设置的。那么有对应的设置方法吗？当然有，这个
方获取的线程池也就是广播的 taskExecutor 属性。那么我们给这个线程池属性赋值不就可以了吗？
然是的，那么问题也就转变成了在初始化广播的时候给线程池属性赋值的问题了。上面内容中讲到广
的初始化是在 Spring boot 服务启动的时候完成的，在 refresh()方法中调用 initApplicationEventMu
ticaster() 完成，我们细看一下这个方法的源码。

```
    /**
     * Initialize the ApplicationEventMulticaster.
     * Uses SimpleApplicationEventMulticaster if none defined in the context.
     * @see org.springframework.context.event.SimpleApplicationEventMulticaster
        * 初始化广播类，如果容器上下文中没有定义该bean的话，那么默认使用SimpleApplicationEv
ntMulticaster
     */
    protected void initApplicationEventMulticaster() {
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
                // public static final String APPLICATION_EVENT_MULTICASTER_BEAN_NAME = "app
icationEventMulticaster";   常量对应的bean name值
        if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
            this.applicationEventMulticaster =
                    beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, Applicat
onEventMulticaster.class);
            if (logger.isTraceEnabled()) {
                logger.trace("Using ApplicationEventMulticaster [" + this.applicationEventMulticaste
 + "]");
            }
        }
        else {
            this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactor
);
            beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.
pplicationEventMulticaster);
            if (logger.isTraceEnabled()) {
                logger.trace("No '" + APPLICATION_EVENT_MULTICASTER_BEAN_NAME + "' bean,
sing " +
                        "[" + this.applicationEventMulticaster.getClass().getSimpleName() + "]");
            }
        }
    }
```

通过源码我们发现，服务启动时，会先找 bean name 为 "applicationEventMulticaster" 的 bean，么问题就简单了，我们只要自定义个 bean name 为 applicationEventMulticaster 的 bean，并给属性 taskExecutor 赋上自定义的线程池即可，这个时候就能实现异步事件处理了，实例代码如下

```
@Configuration
public class CustomSpringEventBroadcast {

    @Bean("applicationEventMulticaster")
    public SimpleApplicationEventMulticaster applicationEventMulticaster(){
        SimpleApplicationEventMulticaster eventMulticaster = new SimpleApplicationEventMulti
aster();
        // 自定义线程池工具类
        Executor executor = ThreadPoolConfigUtil.newThreadTriggerPool("spring-event-executo
");
       // 给广播类的线程池属性赋值
        eventMulticaster.setTaskExecutor(executor);
        return eventMulticaster;
    }
}
```

这个时候，观察者的业务逻辑就是交给自定义的线程池去处理了。

2. @Async

当然除了自定义广播类这个方式以外，还有一种方式也能实现事件的异步处理也就是使用 Spring 提的异步调用注解@Async，使用的方式是在监听类上面添加@Async 注解即可。并且通过实现 Async onfigurer 接口中 getAsyncExecutor() 方法，也能实现使用自定义的线程池。关于@Async 注解的码分析本文中就不展开了，可以参看这篇文章，文章中对@Async 注解进行了详细的分析。

五、总结

文章到此，所述内容已经结束了，总结一下文章主要讲了三个部分内容：

- Spring 事件机制的基础原理，这部分内容主要在第二章中；
- Spring 事件的注解（@EventListener）实现形式，这部分内容主要在第三章中；
- Spring 事件机制的异步实现形式，这里主要讲了实现自定义广播"applicationEventMulticaster"个 bean 并给其线程池属性赋值以及使用异步@Async 的两种方法，这部分内容主要在第四章。