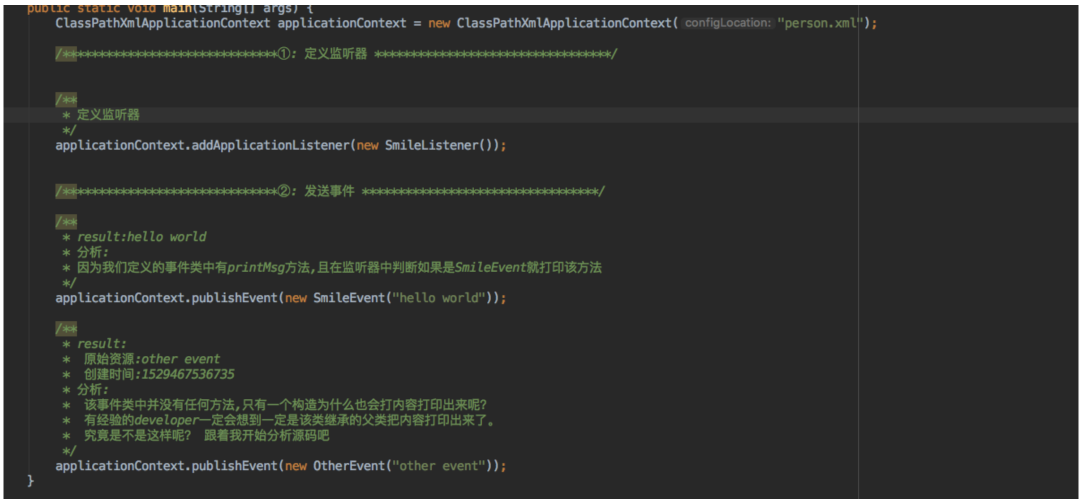
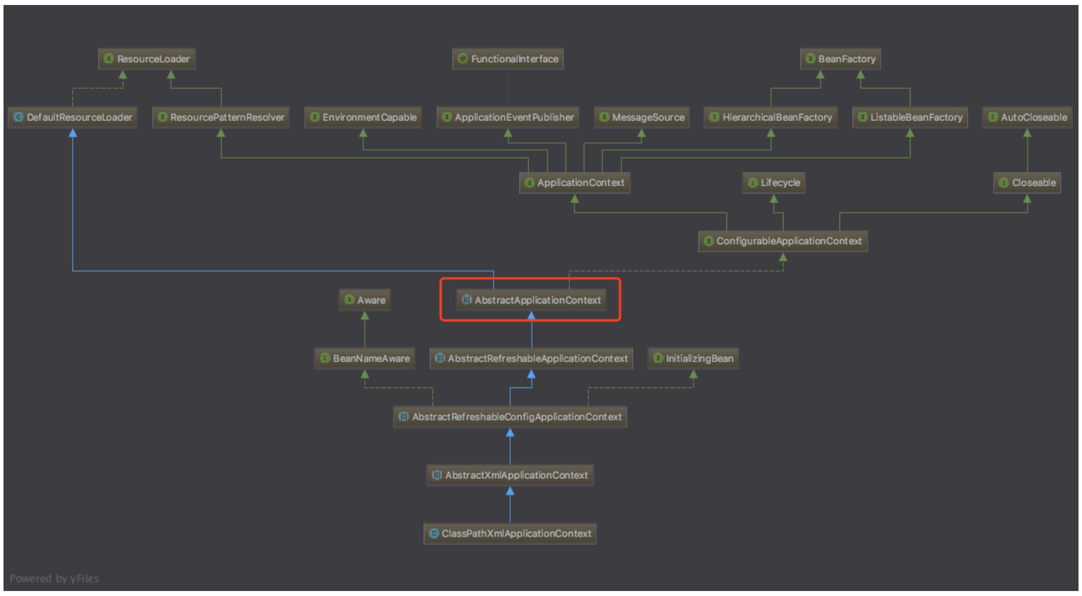
在我们对事件模式有一个认知后我们来阅读源码,来寻找Spring是否也是按照我们的思路去实现呢?



```java
public static void main(String[] args) {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext( configLocation: "person.xml");

    /*******************************①: 定义监听器 ***********************************/


    /**
     * 定义监听器
     */
    applicationContext.addApplicationListener(new SmileListener());


    /*******************************②: 发送事件 ***********************************/

    /**
     * result:hello world
     * 分析:
     * 因为我们定义的事件类中有printMsg方法,且在监听器中判断如果是SmileEvent就打印该方法
     */
    applicationContext.publishEvent(new SmileEvent("hello world"));

    /**
     * result:
     *  原始资源:other event
     *  创建时间:1529467536735
     * 分析:
     *  该事件类中并没有任何方法,只有一个构造为什么也会打内容打印出来呢?
     *  有经验的developer一定会想到一定是该类继承的父类把内容打印出来了。
     *  究竟是不是这样呢?  跟着我开始分析源码吧
     */
    applicationContext.publishEvent(new OtherEvent("other event"));
}
```

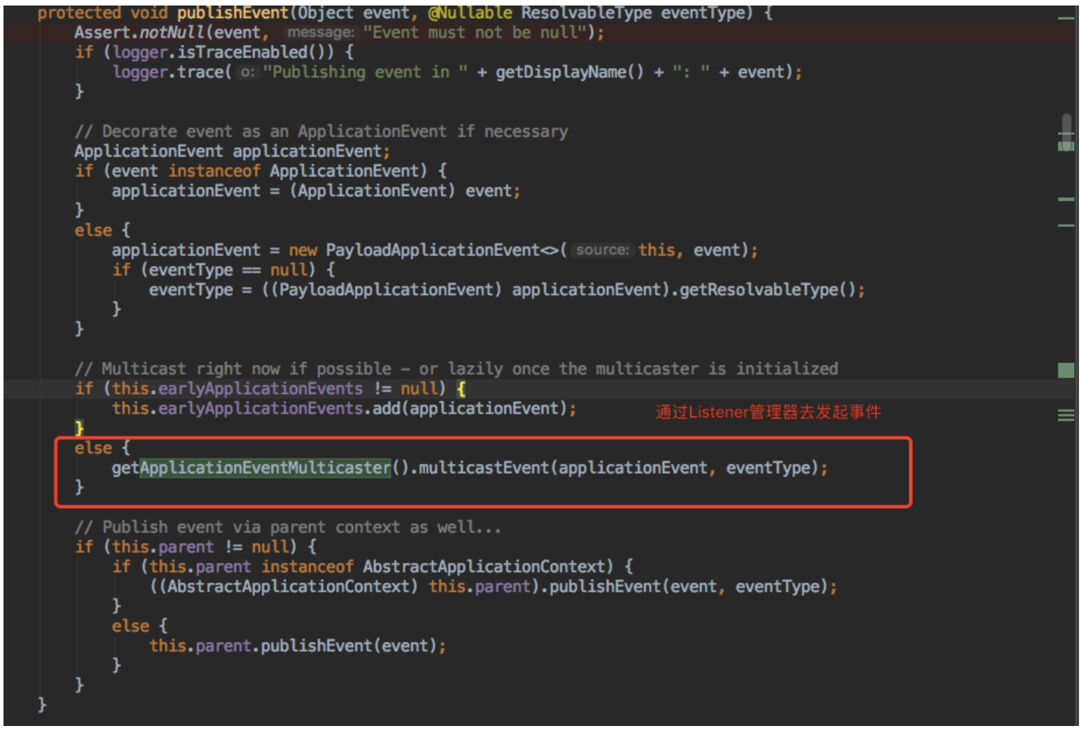


addApplicationListener() 跟进

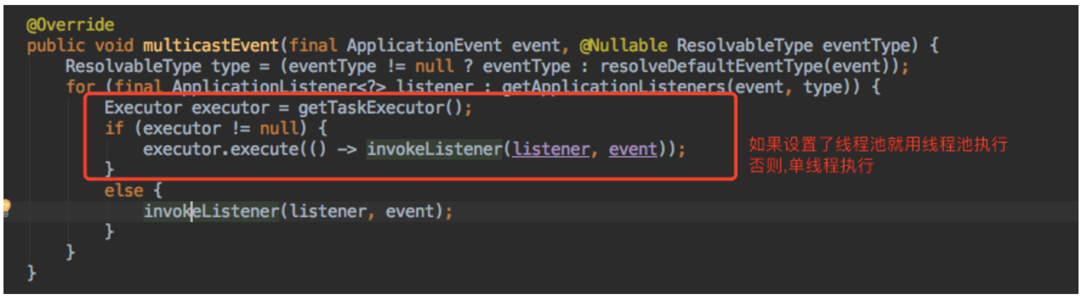
我们发现了一点线索即找到了一个管理Listener的set集合个事件管理器

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader
        implements ConfigurableApplicationContext {
      ...
    /** Helper class used in event publishing */
    @Nullable
    private ApplicationEventMulticaster applicationEventMulticaster;
    /** Statically specified listeners */
    private final Set<ApplicationListener<?>> applicationListeners = new LinkedHashSet<>
();
     ...
    /**
     * applicationEventMulticaster 是对Listener一个管理器
     * applicationListeners 是一个Listener的Set集合
     */
    @Override
    public void addApplicationListener(ApplicationListener<?> listener) {
        Assert.notNull(listener, "ApplicationListener must not be null");
        if (this.applicationEventMulticaster != null) {
            this.applicationEventMulticaster.addApplicationListener(listener);
        }
        else {
            this.applicationListeners.add(listener);
        }
    }
```

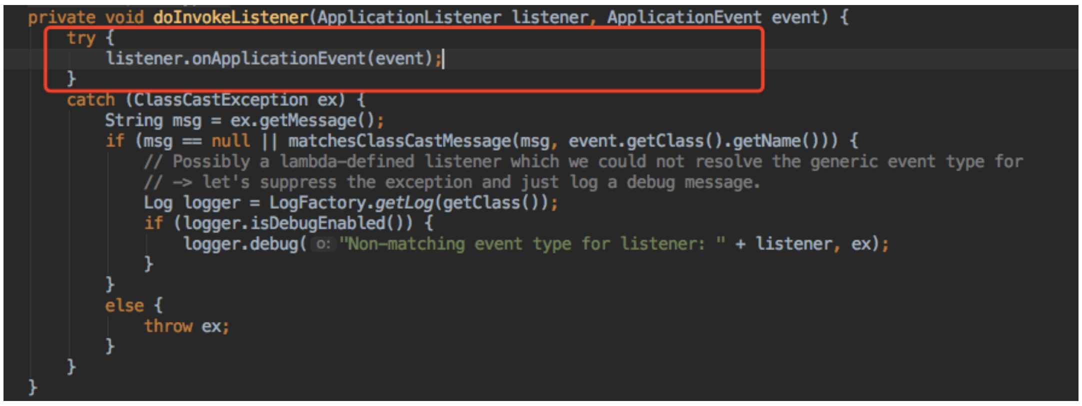
publishEvent跟进


```
protected void publishEvent(Object event, @Nullable ResolvableType eventType) {
    Assert.notNull(event, message: "Event must not be null");
    if (logger.isTraceEnabled()) {
        logger.trace( o: "Publishing event in " + getDisplayName() + ": " + event);
    }

    // Decorate event as an ApplicationEvent if necessary
    ApplicationEvent applicationEvent;
    if (event instanceof ApplicationEvent) {
        applicationEvent = (ApplicationEvent) event;
    }
    else {
        applicationEvent = new PayloadApplicationEvent<>( source: this, event);
        if (eventType == null) {
            eventType = ((PayloadApplicationEvent) applicationEvent).getResolvableType();
        }
    }

    // Multicast right now if possible - or lazily once the multicaster is initialized
    if (this.earlyApplicationEvents != null) {
        this.earlyApplicationEvents.add(applicationEvent);
    }
    else {
        getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
    }

    // Publish event via parent context as well...
    if (this.parent != null) {
        if (this.parent instanceof AbstractApplicationContext) {
            ((AbstractApplicationContext) this.parent).publishEvent(event, eventType);
        }
        else {
            this.parent.publishEvent(event);
        }
    }
}
```

multicastEvent跟进



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



```
private void doInvokeListener(ApplicationListener listener, ApplicationEvent event) {
    try {
        listener.onApplicationEvent(event);
    }
    catch (ClassCastException ex) {
        String msg = ex.getMessage();
        if (msg == null || matchesClassCastMessage(msg, event.getClass().getName())) {
            // Possibly a lambda-defined listener which we could not resolve the generic event type for
            // -> let's suppress the exception and just log a debug message.
            Log logger = LogFactory.getLog(getClass());
            if (logger.isDebugEnabled()) {
                logger.debug( o: "Non-matching event type for listener: " + listener, ex);
            }
        }
        else {
            throw ex;
        }
    }
}
```

通过以上代码的不断跟进我们了解了Spring的事件是怎么实现的？在这里我们不追究细节。感兴趣的同学可以深入了解。