47_源码分析：Spring Event 事件通知机制底层原理

1、开篇

上节课带大家把登录互联网教育系统小程序的验证登录流程走了一遍，具体分为三个部分：登录小程序、获取手机号和解密手机号。每个过程都描述的原因和原理，大家在了解的同时也不用担心其实现过程，因为儒猿团队已经在后台代码中完成了这些功能，大家只用专注于登录的逻辑就行了。本节课会通过源码分析看看 Spring Event 事件通知机制底层原理，今天的内容包括以下几个部分：

- Spring Event 简单使用
- Spring Event 流程分析
- Spring 源码分析

2、Spring Event 简单使用

在开始分析 Spring Event 实现原理之前，一起来回顾一下 Spring Event 的简单应用。如图 1 所示，首先注入 ApplicationEventPublisher 对象，sendEvent 方法中可以通过 ApplicationEventPublisher 的 publishEvent 方法将创建的 EventDTO 发送出去，通常而言在这之前可以编写一些业务逻辑，然后把业务逻辑的相关信息封装到 EventDTO 发送出去，从而达到通知其他组件或者模块的目的。接着，EventListener 会监听发送的消息，通过注册 listenEvent 方法获取以后会处理接收到的 EventDTO 对象，并且对其进行处理。

```
@Resource
ApplicationEventPublisher applicationEventPublisher;
// 发送事件
public void sendEvent() {
    EventDTO eventDTO = new EventDTO();
    applicationEventPublisher.publishEvent(eventDTO);
}
// 监听事件
@EventListener
public void listenEvent(EventDTO eventDTO) {
    System.out.println( Thread.currentThread().getName()+" "+eventDTO);
}
```

图 1 Spring Event 用法

3、Spring Event 流程分析

看完 Spring Event 简单使用之后，可以得知 Spring Event 由消息发送者 ApplicationEventPublisher，消息发送体 EventDTO 以及消息接收者 EventListener 组成的。

下面根据图 2 将 Spring Event 的执行流程进行分析，该图从上往下看。

- 首先是 ApplicationEventPublisher 发布 event。
- 发布消息的动作会调用 Multicaster 中的 multicastEvent 方法。
- 然后检查是否有监听器监听该消息，如果没有监听整个流程结束，否则进入到下一步。
- 接着判断 Multicaster 是否有线程池，如果没有主线程直接调用。
- 如果 Multicaster 有线程池，那么多线程调用。
- 最后执行对应的方法，完成 Spring Event 的调用。

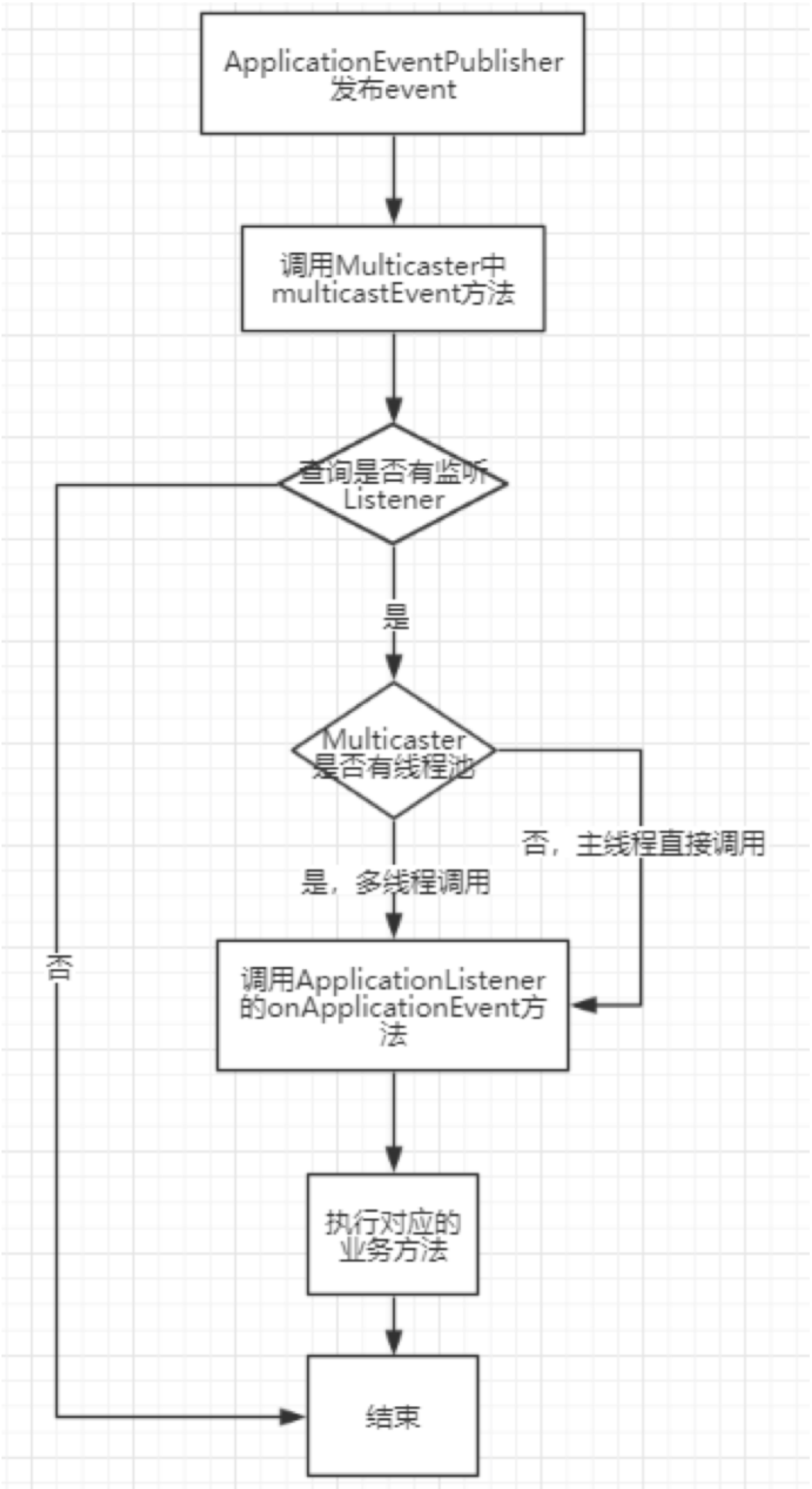
ApplicationEventPublisher
发布event
调用Multicaster中
multicastEvent方法
查询是否有监听
Listener
是
Multicaster
是否有线程池
否，主线程直接调用
是，多线程调用
否
调用ApplicationListener
的onApplicationEvent方
法
执行对应的
业务方法
结束

图 2 Spring Event 流程图

4、Spring 源码分析

在这整个流程中，Multicaster 广播器主要负责发送事件到各个订阅的 Listener 方法中，整个流程的核心节点。在分析整个 Event 之前需要对在 Spring 启动时，如何注入广播器等等工作做一个简单分析。Spring 的启动注入流程如图 3 代码所示如下，我们将特别注意的函数用红色框体标注处理，在后面进行讲解。

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();
        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);
        try {
            // Allows post-processing of the bean factory in context subclasses
            postProcessBeanFactory(beanFactory);
            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);
            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);
            // Initialize message source for this context.
            initMessageSource();
            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();
            // Initialize other special beans in specific context subclasses.
            onRefresh();
            // 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) {
            if (logger.isWarnEnabled()) {…
            }
            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();
            // Reset 'active' flag.
            cancelRefresh(ex);
            // Propagate exception to caller.
            throw ex;
        }
        finally {
            resetCommonCaches();
        }
    }
}
```

图 3 refresh

- initApplicationEventMulticaster 方法：主要对上下文中的对注册事件的广播器进行初始化。
- registerListeners()方法：检查消息监听器并且注册这些监听器。
- finishBeanFactoryInitialization(beanFactory)：初始化含有@EventListener 注解的 Bean 包装到 ApplicationListener 中，然后将它们注入到 Spring IOC 容器。这里初始化的 Bean 指的是所有的非懒加载的 Bean。

下面就对这三个方法的源代码进行解读。

initApplicationEventMulticaster 方法中重要代码如图 4 所示，创建一个简单广播器，传入 beanFactory 主要用于后期获取 BeanFactory 中的 Listener，然后将广播器单例 bean 注入到 Spring IOC 容器中

```
// 创建一个简单广播器，传入beanFactory主要用于后期获取BeanFactory中的Listener
this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
// 将广播器单例bean注入到Spring IOC容器中
beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticaster);
```

图 4 initApplicationEventMulticaster 核心代码

registerListeners 是用来检查和注册监听器的，如图 5 所示，此方法主要将容器中所有声明的 ApplicationListener 的 Bean 对象加载到广播器中，并且发送在广播器和 Bean 准备期间所需要发送的事件。

```
// Register statically specified listeners first.
for (ApplicationListener<?> listener : getApplicationListeners()) {
    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) {
    getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
}

// Publish early application events now that we finally have a multicaster...
// earlyApplicationEvents 是容器刚刚初始化所构建的Event链表，里面主要保存在广播器够着之前
// 需要发放的Event
Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
this.earlyApplicationEvents = null;
if (earlyEventsToProcess != null) {
    for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
        getApplicationEventMulticaster().multicastEvent(earlyEvent);
    }
}
```

图 5 registerListeners

如图 6 所示，将容器中初始化好的 Spring Bean 中包含@EventListener 注解的方法和 Bean 包裹到一个新通用的 ApplicationListener 子类（即 ApplicationListenerMethodAdapter 中），

```
private void processBean(final String beanName, final Class<?> targetType) {
    if (!this.nonAnnotatedClasses.contains(targetType) &&
            AnnotationUtils.isCandidateClass(targetType, EventListener.class) &&
            !isSpringContainerClass(targetType)) {
        Map<Method, EventListener> annotatedMethods = null;
        try {
            annotatedMethods = MethodIntrospector.selectMethods(targetType,
                    (MethodIntrospector.MetadataLookup<EventListener>) method ->
                            AnnotatedElementUtils.findMergedAnnotation(method, EventListener.class));
        }
        catch (Throwable ex) {
        }
        if (CollectionUtils.isEmpty(annotatedMethods)) {
            this.nonAnnotatedClasses.add(targetType);
        }
        else {
            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.getType(beanName));
                        ApplicationListener<?> applicationListener =
                                factory.createApplicationListener(beanName, targetType, methodToUse);
                        if (applicationListener instanceof ApplicationListenerMethodAdapter) {
                            ((ApplicationListenerMethodAdapter) applicationListener).init(context, this.evaluator);
                        }
                        context.addApplicationListener(applicationListener);
                        break;
                    }
                }
            }

        }
    }
}
```

图 6 processBean

ApplicationListenerMethodAdapter 类，其继承自 ApplicationListener 实现 onApplicationEvent 方法，传入参数：String beanName, Class<?> targetClass, Method method（@EventListener 的目标方法）。如图 7 所示，它会实现 processEvent 方法，方法体中的 handleResult 会将目标方法执行的结果继续当做 Event 发送出去。

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

图 7 processEvent

5、总结

本节课进行了 Spring Event 事件通知机制的原理分析，通过三个部分给大家展开介绍：首先通过 Spring Event 简单使用的例子了解 Spring Event 的要素，然后对其发送消息以及接收消息的流程进行描述，最后针对源码中重要的三个函数进行了深入的分析。

下节课会在登录验证的基础上，考虑如何通过异步化的方式将登录、发优惠券操作进行解耦。下期见，拜拜。