



下载APP



## 07 | Spring事件常见错误

2021-05-05 傅健

Spring编程常见错误50例

[进入课程 >](#)**讲述：傅健**

时长 17:12 大小 15.76M



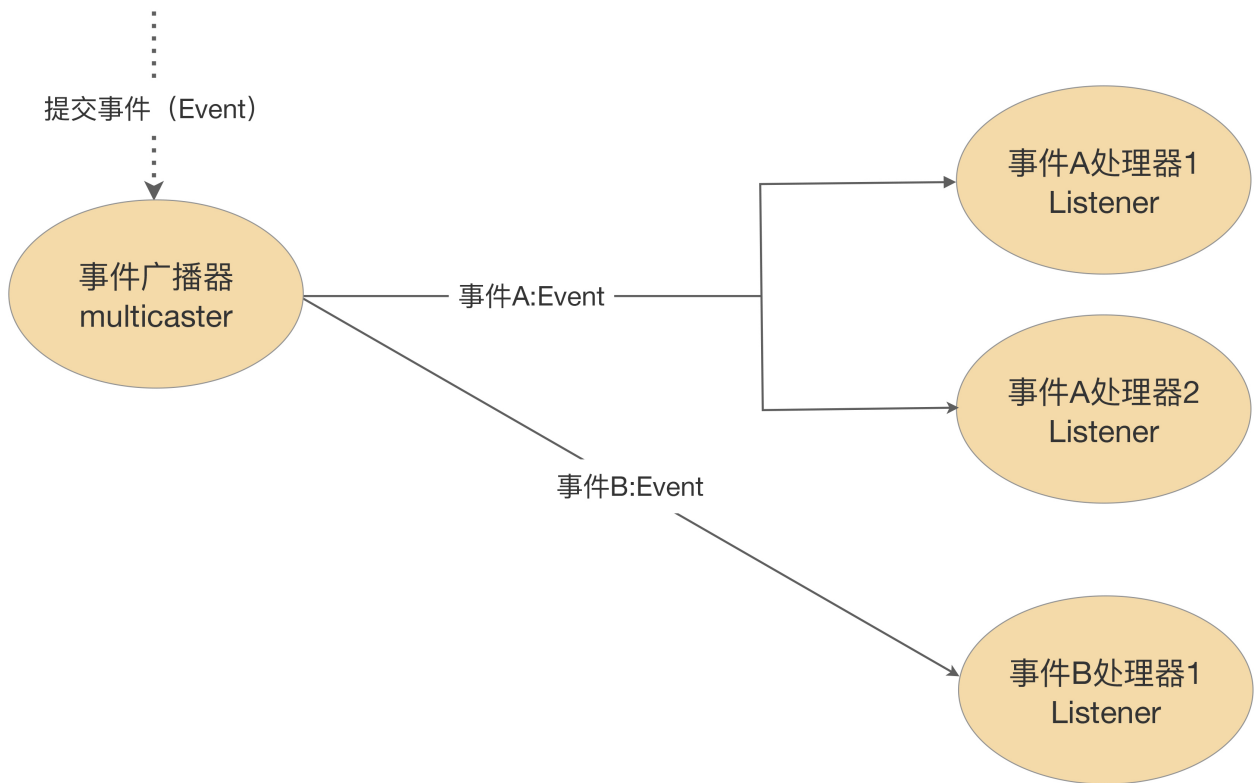
你好，我是傅健，这节课我们聊聊 Spring 事件上的常见错误。

前面的几讲中，我们介绍了 Spring 依赖注入、AOP 等核心功能点上的常见错误。而作为 Spring 的关键功能支撑，Spring 事件是一个相对独立的点。或许你从没有在自己的项目中使用过 Spring 事件，但是你一定见过它的相关日志。而且在未来的编程实践中，你会发现，一旦你用上了 Spring 事件，往往完成的都是一些有趣的、强大的功能，例如动态配置。那么接下来我就来讲讲 Spring 事件上都有哪些常见的错误。

### 案例 1：试图处理并不会抛出的事件



Spring 事件的设计比较简单。说白了，就是监听器设计模式在 Spring 中的一种实现，参考下图：



从图中我们可以看出，Spring 事件包含以下三大组件。

1. 事件 (Event)：用来区分和定义不同的事件，在 Spring 中，常见的如 `ApplicationEvent` 和 `AutoConfigurationImportEvent`，它们都继承于 `java.util.EventObject`。
2. 事件广播器 (Multicaster)：负责发布上述定义的事件。例如，负责发布 `ApplicationEvent` 的 `ApplicationEventMulticaster` 就是 Spring 中一种常见的广播器。
3. 事件监听器 (Listener)：负责监听和处理广播器发出的事件，例如 `ApplicationListener` 就是用来处理 `ApplicationEventMulticaster` 发布的 `ApplicationEvent`，它继承于 JDK 的 `EventListener`，我们可以看下它的定义来验证这个结论：

```
public interface ApplicationListener<E extends ApplicationEvent> extends
EventListener {
    void onApplicationEvent(E event);
}
```

当然，虽然在上述组件中，任何一个都是缺一不可的，但是功能模块命名不见得完全贴合上述提及的关键字，例如发布 `AutoConfigurationImportEvent` 的广播器就不含有 `Multicaster` 字样。它的发布是由 `AutoConfigurationImportSelector` 来完成的。

对这些基本概念和实现有了一定的了解后，我们就可以开始解析那些常见的错误。闲话少说，我们先来看下面这段基于 Spring Boot 技术栈的代码：

[复制代码](#)

```
1 @Slf4j
2 @Component
3 public class MyContextStartedEventListener implements ApplicationListener<Cont
4
5     public void onApplicationEvent(final ContextStartedEvent event) {
6         log.info("{} received: {}", this.toString(), event);
7     }
8
9 }
```

很明显，这段代码定义了一个监听器 `MyContextStartedEventListener`，试图拦截 `ContextStartedEvent`。因为在很多 Spring 初级开发者眼中，Spring 运转的核心就是一个 `Context` 的维护，那么启动 Spring 自然会启动 `Context`，于是他们是很期待出现类似下面的日志的：

```
2021-03-07 07:08:21.197 INFO 2624 --- [nio-8080-exec-1]
c.s.p.l.e.MyContextStartedEventListener :
com.spring.puzzle.class7.example1.MyContextStartedEventListener@d33d5a
received:
org.springframework.context.event.ContextStartedEvent[source=org.springfra
mework.boot.web.servlet.context.AnnotationConfigServletWebServerApplication
Context@19b56c0, started on Sun Mar 07 07:07:57 CST 2021]
```

但是当我们启动 Spring Boot 后，会发现并不会拦截到这个事件，如何理解这个错误呢？

## 案例解析

在 Spring 事件运用上，这是一个常见的错误，就是不假思索地认为一个框架只要定义了一个事件，那么一定会抛出来。例如，在本案例中，`ContextStartedEvent` 就是 Spring 内

置定义的事件，而 Spring Boot 本身会创建和运维 Context，表面看起来这个事件的抛出是必然的，但是这个事件一定会在 Spring Boot 启动时抛出来么？

答案明显是否定的，我们首先看下要抛出这个事件需要调用的方法是什么？在 Spring Boot 中，这个事件的抛出只发生在一处，即位于方法 `AbstractApplicationContext#start` 中。

[复制代码](#)

```
1 @Override
2 public void start() {
3     getLifecycleProcessor().start();
4     publishEvent(new ContextStartedEvent(this));
5 }
```

也就是说，只有上述方法被调用，才会抛出 `ContextStartedEvent`，但是这个方法在 Spring Boot 启动时会被调用么？我们可以查看 Spring 启动方法中围绕 Context 的关键方法调用，代码如下：

[复制代码](#)

```
1 public ConfigurableApplicationContext run(String... args) {
2     //省略非关键代码
3     context = createApplicationContext();
4     //省略非关键代码
5     prepareContext(context, environment, listeners, applicationArguments, pr
6     refreshContext(context);
7     //省略非关键代码
8     return context;
9 }
```

我们发现围绕 Context、Spring Boot 的启动只做了两个关键工作：创建 Context 和 Refresh Context。其中 Refresh 的关键代码如下：

[复制代码](#)

```
1 protected void refresh(ApplicationContext applicationContext) {
2     Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);
3     ((AbstractApplicationContext) applicationContext).refresh();
4 }
```

很明显，Spring 启动最终调用的是 `AbstractApplicationContext#refresh`，并不是 `AbstractApplicationContext#start`。在这样的残酷现实下，`ContextStartedEvent` 自然不会被抛出，不抛出，自然也不可能被捕获。所以这样的错误也就自然发生了。

## 问题修正

针对这个案例，有了源码的剖析，我们可以很快找到问题发生的原因，但是修正这个问题还要去追溯我们到底想要的是什么？我们可以分两种情况来考虑。

### 1. 假设我们是误读了 `ContextStartedEvent`。

针对这种情况，往往是因为我们确实想在 Spring Boot 启动时拦截一个启动事件，但是我们粗略扫视相关事件后，误以为 `ContextStartedEvent` 就是我们想要的。针对这种情况，我们只需要把监听事件的类型修改成真正发生的事件即可，例如在本案例中，我们可以修正如下：

[复制代码](#)

```
1 @Component
2 public class MyContextRefreshedEventListener implements ApplicationListener<Co
3
4     public void onApplicationEvent(final ContextRefreshedEvent event) {
5         log.info("{} received: {}", this.toString(), event);
6     }
7
8 }
```

我们监听 `ContextRefreshedEvent` 而非 `ContextStartedEvent`。

`ContextRefreshedEvent` 的抛出可以参考方法

`AbstractApplicationContext#finishRefresh`，它本身正好是 Refresh 操作中的一步。

[复制代码](#)

```
1 protected void finishRefresh() {
2     //省略非关键代码
3     initLifecycleProcessor();
4     // Propagate refresh to lifecycle processor first.
5     getLifecycleProcessor().onRefresh();
6     // Publish the final event.
7     publishEvent(new ContextRefreshedEvent(this));
8     //省略非关键代码
9 }
```

## 2. 假设我们就是想要处理 ContextStartedEvent。

这种情况下，我们真的需要去调用 `AbstractApplicationContext#start` 方法。例如，我们可以使用下面的代码来让这个事件抛出：

[复制代码](#)

```
1 @RestController
2 public class HelloWorldController {
3
4     @Autowired
5     private AbstractApplicationContext applicationContext;
6
7     @RequestMapping(path = "publishEvent", method = RequestMethod.GET)
8     public String notifyEvent(){
9         applicationContext.start();
10        return "ok";
11    };
12 }
```

我们随便找一处来 `Autowired` 一个 `AbstractApplicationContext`，然后直接调用其 `start()` 就能让事件抛出来。

很明显，这种抛出并不难，但是作为题外话，我们可以思考下为什么要去调用 `start()` 呢？`start()` 本身在 Spring Boot 中有何作用？

如果我们去翻阅这个方法，我们会发现 `start()` 是 `org.springframework.context.Lifecycle` 定义的方法，而它在 Spring Boot 的默认实现中是去执行所有 `Lifecycle Bean` 的启动方法，这点可以参考 `DefaultLifecycleProcessor#startBeans` 方法来验证：

[复制代码](#)

```
1 private void startBeans(boolean autoStartupOnly) {
2     Map<String, Lifecycle> lifecycleBeans = getLifecycleBeans();
3     Map<Integer, LifecycleGroup> phases = new HashMap<>();
4     lifecycleBeans.forEach((beanName, bean) -> {
5         if (!autoStartupOnly || (bean instanceof SmartLifecycle && ((SmartLifecycle
6             int phase = getPhase(bean);
7             LifecycleGroup group = phases.get(phase);
```



```
8         if (group == null) {
9             group = new LifecycleGroup(phase, this.timeoutPerShutdownPhase, li
10             phases.put(phase, group);
11         }
12         group.add(beanName, bean);
13     }
14 });
15 if (!phases.isEmpty()) {
16     List<Integer> keys = new ArrayList<>(phases.keySet());
17     Collections.sort(keys);
18     for (Integer key : keys) {
19         phases.get(key).start();
20     }
21 }
22 }
```

说起来比较抽象，我们可以去写一个 Lifecycle Bean，代码如下：

[复制代码](#)

```
1 @Component
2 @Slf4j
3 public class MyLifeCycle implements Lifecycle {
4
5     private volatile boolean running = false;
6
7     @Override
8     public void start() {
9         log.info("lifecycle start");
10        running = true;
11    }
12
13    @Override
14    public void stop() {
15        log.info("lifecycle stop");
16        running = false;
17    }
18
19    @Override
20    public boolean isRunning() {
21        return running;
22    }
23
24 }
```


当我们再次运行 Spring Boot 时，只要执行了 AbstractApplicationContext 的 start()，就会输出上述代码定义的行为：输出 LifeCycle start 日志。

通过这个 Lifecycle Bean 的使用，AbstractApplicationContext 的 start 要做的事，我们就清楚多了。它和 Refresh() 不同，Refresh() 是初始化和加载所有需要管理的 Bean，而 start 只有在有 Lifecycle Bean 时才有被调用的价值。那么我们自定义 Lifecycle Bean 一般是用来做什么呢？例如，可以用它来实现运行中的启停。这里不再拓展，你可以自己做更深入的探索。

通过这个案例，我们搞定了第一类错误。而从这个错误中，我们也得出了一个启示：**当一个事件拦截不了时，我们第一个要查的是拦截的事件类型对不对，执行的代码能不能抛出它。**把握好这点，也就事半功倍了。

## 案例 2：监听事件的体系不对

通过案例 1 的学习，我们可以保证事件的抛出，但是抛出的事件就一定能被我们监听到么？我们再来看这样一个案例，首先上代码：

 复制代码

```
1 @Slf4j
2 @Component
3 public class MyApplicationEnvironmentPreparedEventListener implements Applicat
4
5     public void onApplicationEvent(final ApplicationEnvironmentPreparedEvent e
6         log.info("{} received: {}", this.toString(), event);
7     }
8
9 }
```

这里我们试图处理 ApplicationEnvironmentPreparedEvent。期待出现拦截事件的日志如下：

```
2021-03-07 09:12:08.886 INFO 27064 --- [ restartedMain]
licationEnvironmentPreparedEventListener :
com.spring.puzzle.class7.example2.MyApplicationEnvironmentPreparedEventList
ener@2b093d received:
org.springframework.boot.context.event.ApplicationEnvironmentPreparedEvent[
source=org.springframework.boot.SpringApplication@122b9e6]
```

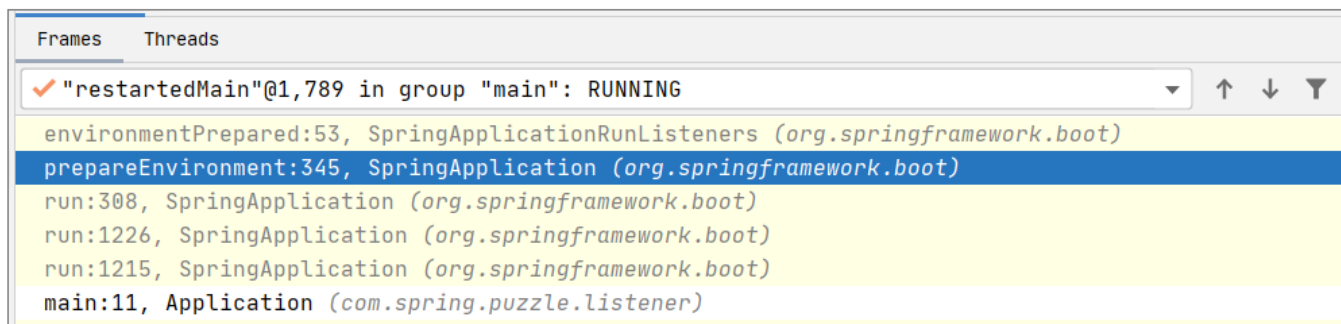


有了案例 1 的经验，首先我们就可以查看下这个事件的抛出会不会存在问题。这个事件在 Spring 中是由 `EventPublishingRunListener#environmentPrepared` 方法抛出，代码如下：

[复制代码](#)

```
1 @Override
2 public void environmentPrepared(ConfigurableEnvironment environment) {
3     this.initialMulticaster
4         .multicastEvent(new ApplicationEnvironmentPreparedEvent(this.applicat
5 }
```

现在我们调试下代码，你会发现这个方法在 Spring 启动时一定经由 `SpringApplication#prepareEnvironment` 方法调用，调试截图如下：



表面上看，既然代码会被调用，事件就会抛出，那么我们在最开始定义的监听器就能处理，但是我们真正去运行程序时会发现，效果和案例 1 是一样的，都是监听器的处理并不执行，即拦截不了。这又是为何？

## 案例解析

实际上，这是在 Spring 事件处理上非常容易犯的一个错误，即监听的体系不一致。通俗点说，就是“驴头不对马嘴”。我们首先来看下关于 `ApplicationEnvironmentPreparedEvent` 的处理，它相关的两大组件是什么？

1. 广播器：这个事件的广播器是 `EventPublishingRunListener` 的 `initialMulticaster`，代码参考如下：

[复制代码](#)

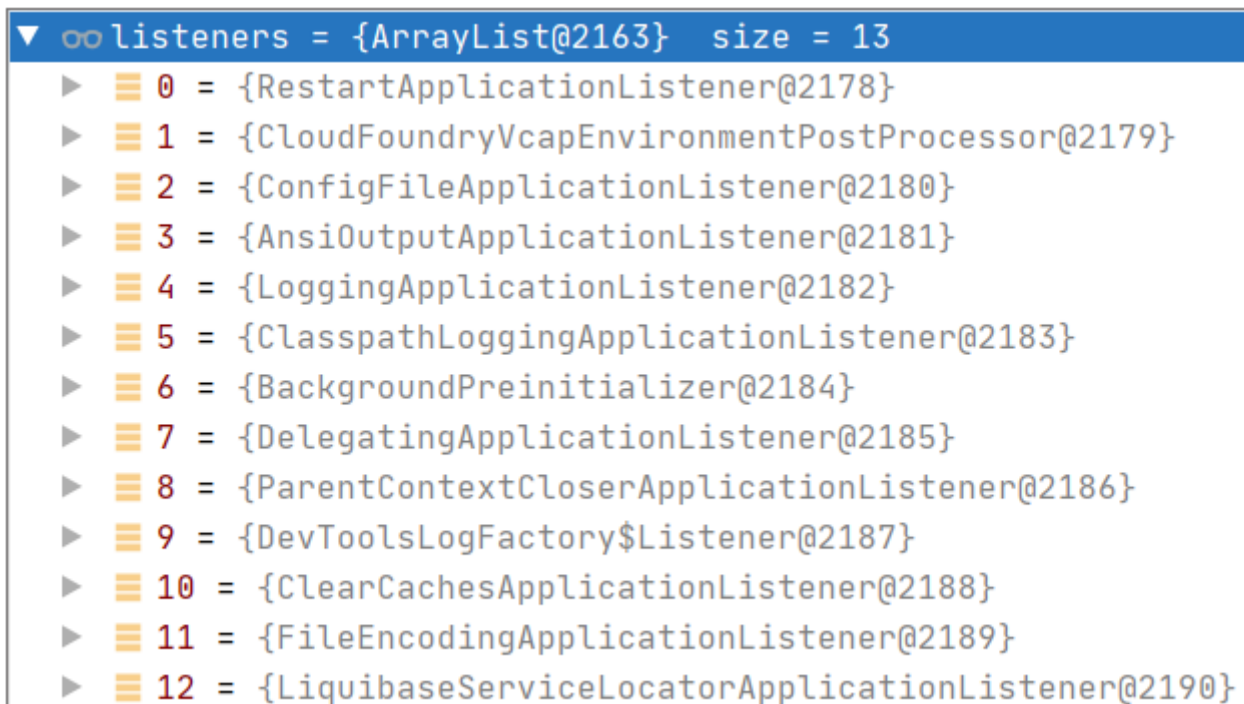
```
1 public class EventPublishingRunListener implements SpringApplicationRunListene
2     //省略非关键代码
```

```
3     private final SimpleApplicationEventMulticaster initialMulticaster;
4
5     public EventPublishingRunListener(SpringApplication application, String[] a
6         //省略非关键代码
7         this.initialMulticaster = new SimpleApplicationEventMulticaster();
8         for (ApplicationListener<?> listener : application.getListeners()) {
9             this.initialMulticaster.addApplicationListener(listener);
10        }
11    }
12 }
```

2. 监听器：这个事件的监听器同样位于 `EventPublishingRunListener` 中，获取方式参考关键代码行：

```
this.initialMulticaster.addApplicationListener(listener);
```

如果继续查看代码，我们会发现这个事件的监听器就存储在 `SpringApplication#Listeners` 中，调试下就可以找出所有的监听器，截图如下：



```
listeners = {ArrayList@2163} size = 13
0 = {RestartApplicationListener@2178}
1 = {CloudFoundryVcapEnvironmentPostProcessor@2179}
2 = {ConfigFileApplicationListener@2180}
3 = {AnsiOutputApplicationListener@2181}
4 = {LoggingApplicationListener@2182}
5 = {ClasspathLoggingApplicationListener@2183}
6 = {BackgroundPreinitializer@2184}
7 = {DelegatingApplicationListener@2185}
8 = {ParentContextCloserApplicationListener@2186}
9 = {DevToolsLogFactory$Listener@2187}
10 = {ClearCachesApplicationListener@2188}
11 = {FileEncodingApplicationListener@2189}
12 = {LiquibaseServiceLocatorApplicationListener@2190}
```

从中我们可以发现并不存在我们定义的

`MyApplicationEnvironmentPreparedEventListener`，这是为何？

还是查看代码，当 `Spring Boot` 被构建时，会使用下面的方法去寻找上述监听器：

```
setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
```

而上述代码最终寻找 Listeners 的候选者，参考代码

SpringFactoriesLoader#loadSpringFactories 中的关键行：

```
// 下面的 FACTORIES_RESOURCE_LOCATION 定义为 "META-INF/spring.factories"  
classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
```

我们可以寻找下这样的文件（spring.factories），确实可以发现类似的定义：

 复制代码

```
1 org.springframework.context.ApplicationListener=\n2 org.springframework.boot.ClearCachesApplicationListener,\n3 org.springframework.boot.builder.ParentContextCloserApplicationListener,\n4 org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor,\n5 //省略其他监听器
```

说到这里，相信你已经意识到本案例的问题所在。我们定义的监听器并没有被放置在 META-INF/spring.factories 中，实际上，我们的监听器监听的体系是另外一套，其关键组件如下：


1. 广播器：即 AbstractApplicationContext#applicationEventMulticaster;
2. 监听器：由上述提及的 META-INF/spring.factories 中加载的监听器以及扫描到的 ApplicationListener 类型的 Bean 共同组成。

这样比较后，我们可以得出一个结论：**我们定义的监听器并不能监听到 initialMulticaster 广播出的 ApplicationEnvironmentPreparedEvent。**

## 问题修正

现在就到了解决问题的时候了，我们可以把自定义监听器注册到 initialMulticaster 广播体系中，这里提供两种方法修正问题。

1. 在构建 Spring Boot 时，添加 MyApplicationEnvironmentPreparedEventListener:

 复制代码

```
1 @SpringBootApplication
2 public class Application {
3     public static void main(String[] args) {
4         MyApplicationEnvironmentPreparedEventListener myApplicationEnvironment
5         SpringApplication springApplication = new SpringApplicationBuilder(App
6         springApplication.run(args);
7     }
8 }
```

2. 使用 META-INF/spring.factories, 即在 /src/main/resources 下面新建目录 META-INF, 然后新建一个对应的 spring.factories 文件:

 复制代码

```
1 org.springframework.context.ApplicationListener=\
2 com.spring.puzzle.listener.example2.MyApplicationEnvironmentPreparedEventListe
```


通过上述两种修改方式, 即可完成事件的监听, 很明显第二种方式要优于第一种, 至少完全用原生的方式去解决, 而不是手工实例化一个

MyApplicationEnvironmentPreparedEventListener。这点还是挺重要的。

反思这个案例的错误, 结论就是**对于事件一定要注意“驴头”（监听器）对上“马嘴”（广播）**。

### 案例 3：部分事件监听器失效

通过前面案例的解析, 我们可以确保事件在合适的时机被合适的监听器所捕获。但是理想总是与现实有差距, 有些时候, 我们可能还会发现部分事件监听器一直失效或偶尔失效。这里我们可以写一段代码来模拟偶尔失效的场景, 首先我们完成一个自定义事件和两个监听器, 代码如下:

 复制代码

```
1 public class MyEvent extends ApplicationEvent {
2     public MyEvent(Object source) {
3         super(source);
4     }
5 }
6
7 @Component
8 @Order(1)
```

```
9 public class MyFirstEventListener implements ApplicationListener<MyEvent> {
10
11     Random random = new Random();
12
13     @Override
14     public void onApplicationEvent(MyEvent event) {
15         log.info("{} received: {}", this.toString(), event);
16         //模拟部分失效
17         if(random.nextInt(10) % 2 == 1)
18             throw new RuntimeException("exception happen on first listener");
19     }
20 }
21
22 @Component
23 @Order(2)
24 public class MySecondEventListener implements ApplicationListener<MyEvent> {
25     @Override
26     public void onApplicationEvent(MyEvent event) {
27         log.info("{} received: {}", this.toString(), event);
28     }
29 }
30 }
```

这里监听器 `MyFirstEventListener` 的优先级稍高，且执行过程中会有 50% 的概率抛出异常。然后我们再写一个 `Controller` 来触发事件的发送：

[复制代码](#)

```
1 @RestController
2 @Slf4j
3 public class HelloWorldController {
4
5     @Autowired
6     private AbstractApplicationContext applicationContext;
7
8     @RequestMapping(path = "publishEvent", method = RequestMethod.GET)
9     public String notifyEvent(){
10         log.info("start to publish event");
11         applicationContext.publishEvent(new MyEvent(UUID.randomUUID()));
12         return "ok";
13     };
14 }
```

完成这些代码后，我们就可以使用 <http://localhost:8080/publishEvent> 来测试监听器的接收和执行了。观察测试结果，我们会发现监听器 `MySecondEventListener` 有一半的概率并没有接收到任何事件。可以说，我们使用了最简化的代码模拟出了部分事件监听器


偶尔失效的情况。当然在实际项目中，抛出异常这个根本原因肯定不会如此明显，但还是可以借机举一反三的。那么如何理解这个问题呢？

## 案例解析

这个案例非常简易，如果你稍微有些开发经验的话，大概也能推断出原因：处理器的执行是顺序执行的，在执行过程中，如果一个监听器执行抛出了异常，则后续监听器就得不到被执行的机会了。这里我们可以通过 Spring 源码看下事件是如何被执行的？

具体而言，当广播一个事件，执行的方法参考

`SimpleApplicationEventMulticaster#multicastEvent(ApplicationEvent):`

 复制代码

```
1 @Override
2 public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType
3     ResolvableType type = (eventType != null ? eventType : resolveDefaultEventT
4     Executor executor = getTaskExecutor());
5     for (ApplicationListener<?> listener : getApplicationListeners(event, type)
6         if (executor != null) {
7             executor.execute(() -> invokeListener(listener, event));
8         }
9         else {
10             invokeListener(listener, event);
11         }
12     }
13 }
```

上述方法通过 Event 类型等信息调用 `getApplicationListeners` 获取了具有执行资格的所有监听器（在本案例中，即为 `MyFirstEventListener` 和 `MySecondEventListener`），然后按顺序去执行。最终每个监听器的执行是通过 `invokeListener()` 来触发的，调用的是接口方法 `ApplicationListener#onApplicationEvent`。执行逻辑可参考如下代码：

 复制代码

```
1 protected void invokeListener(ApplicationListener<?> listener, ApplicationEven
2     ErrorHandler errorHandler = getErrorHandler();
3     if (errorHandler != null) {
4         try {
5             doInvokeListener(listener, event);
6         }
7         catch (Throwable err) {
8             errorHandler.handleError(err);
9         }
10    }
```



```
9      }
10     }
11     else {
12         doInvokeListener(listener, event);
13     }
14 }
15
16 private void doInvokeListener(ApplicationListener listener, ApplicationEvent e
17     try {
18         listener.onApplicationEvent(event);
19     }
20     catch (ClassCastException ex) {
21         //省略非关键代码
22     }
23     else {
24         throw ex;
25     }
26 }
27 }
```


这里我们并没有去设置什么 `org.springframework.util.ErrorHandler`，也没有绑定什么 `Executor` 来执行任务，所以针对本案例的情况，我们可以看出：**最终事件的执行是由同一个线程按顺序来完成的，任何一个报错，都会导致后续的监听器执行不了。**

## 问题修正

怎么解决呢？好办，我提供两种方案给你。

### 1. 确保监听器的执行不会抛出异常。

既然我们使用多个监听器，我们肯定是希望它们都能执行的，所以我们一定要保证每个监听器的执行不会被其他监听器影响。基于这个思路，我们修改案例代码如下：

 复制代码

```
1 @Component
2 @Order(1)
3 public class MyFirstEventListener implements ApplicationListener<MyEvent> {
4     @Override
5     public void onApplicationEvent(MyEvent event) {
6         try {
7             // 省略事件处理相关代码
8         } catch (Throwable throwable) {
9             //write error/metric to alert
10        }
```

```
11     }  
12 }  
13
```

## 2. 使用 org.springframework.util.ErrorHandler。

通过上面的案例解析，我们发现，假设我们设置了一个 ErrorHandler，那么就可以用这个 ErrorHandler 去处理掉异常，从而保证后续事件监听器处理不受影响。我们可以使用下面的代码来修正问题：

[复制代码](#)

```
1 SimpleApplicationEventMulticaster simpleApplicationEventMulticaster = applicat  
2     simpleApplicationEventMulticaster.setErrorHandler(TaskUtils.LOG_AND_SUPPRE
```

其中 LOG\_AND\_SUPPRESS\_ERROR\_HANDLER 的实现如下：

[复制代码](#)

```
1 public static final ErrorHandler LOG_AND_SUPPRESS_ERROR_HANDLER = new LoggingE  
2  
3 private static class LoggingErrorHandler implements ErrorHandler {  
4  
5     private final Log logger = LogFactory.getLog(LoggingErrorHandler.class);  
6  
7     @Override  
8     public void handleError(Throwable t) {  
9         logger.error("Unexpected error occurred in scheduled task", t);  
10    }  
11 }
```

对比下方案 1，使用 ErrorHandler 有一个很大的优势，就是我们不需要在某个监听器中都重复类似下面的代码了：

[复制代码](#)

```
1 try {  
2     //省略事件处理过程  
3 }catch(Throwable throwable){  
4     //write error/metric to alert  
5 }
```

这么看的话，其实 Spring 的设计还是很全面的，它考虑了各种各样的情况。但是 Spring 使用者往往都不会去了解其内部实现，这样就会遇到各种各样的问题。相反，如果你对其实现有所了解的话，也对常见错误有一个感知，则大概率是可以快速避坑的，项目也可以运行得更加平稳顺畅。

## 重点回顾

今天我们粗略地了解了 Spring 事件处理的基本流程。其实，抛开 Spring 框架，我们去设计一个通用的事件处理框架，常常也会犯这三种错误：

1. 误读事件本身含义；
2. 监听错了事件的传播系统；
3. 事件处理之间互相影响，导致部分事件处理无法完成。

这三种错误正好对应了我们这节课讲解的三个案例。

此外，在 Spring 事件处理过程中，我们也学习到了监听器加载的特殊方式，即使用 SPI 的方式直接从配置文件 META-INF/spring.factories 中加载。这种方式或者说思想非常值得你去学习，因为它在许多 Java 应用框架中都有所使用，例如 Dubbo，就是使用增强版的 SPI 来配置编解码器的。

## 思考题

在案例 3 中，我们提到默认的事件执行是在同一个线程中执行的，即事件发布者使用的线程。参考如下日志佐证这个结论：

```
2021-03-09 09:10:33.052 INFO 18104 --- [nio-8080-exec-1]
c.s.p.listener.HelloWorldController : start to publish event
2021-03-09 09:10:33.055 INFO 18104 --- [nio-8080-exec-1]
c.s.p.l.example3.MyFirstEventListener :
com.spring.puzzle.class7.example3.MyFirstEventListener@18faf0 received:
com.spring.puzzle.class7.example3.MyEvent[source=df42b08f-8ee2-44df-a957-
d8464ff50c88]
```

通过日志可以看出，事件的发布和执行使用的都是 `nio-8080-exec-1` 线程，但是在事件比较多时，我们往往希望事件执行得更快些，或者希望事件的执行可以异步化不影响主线程。此时应该怎么做呢？

期待在留言区看到你的回复，我们下节课见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | Spring AOP常见错误（下）

## 精选留言 (1)

写留言



哦吼掉了

2021-05-06

思考题：在SimpleApplicationEventMulticaster实例化的时候，设置属性。或者使用@PostConstructor注解。在想有没有啥其他的更优雅的方式？

@Bean

```
public SimpleApplicationEventMulticaster testMulticaster(SimpleApplicationEventMulticaster caster) {...
```

展开 ∨

