Spring系列-5 事件机制

原创 Ewen Seong 已于 2024-04-05 16:41:00 修改 阅读量2.6k 收藏 8 点赞数 1

分类专栏： Spring系列 文章标签： spring java 后端

Spring系列 专栏收录该内容 19 订阅 13 篇文章

背景：

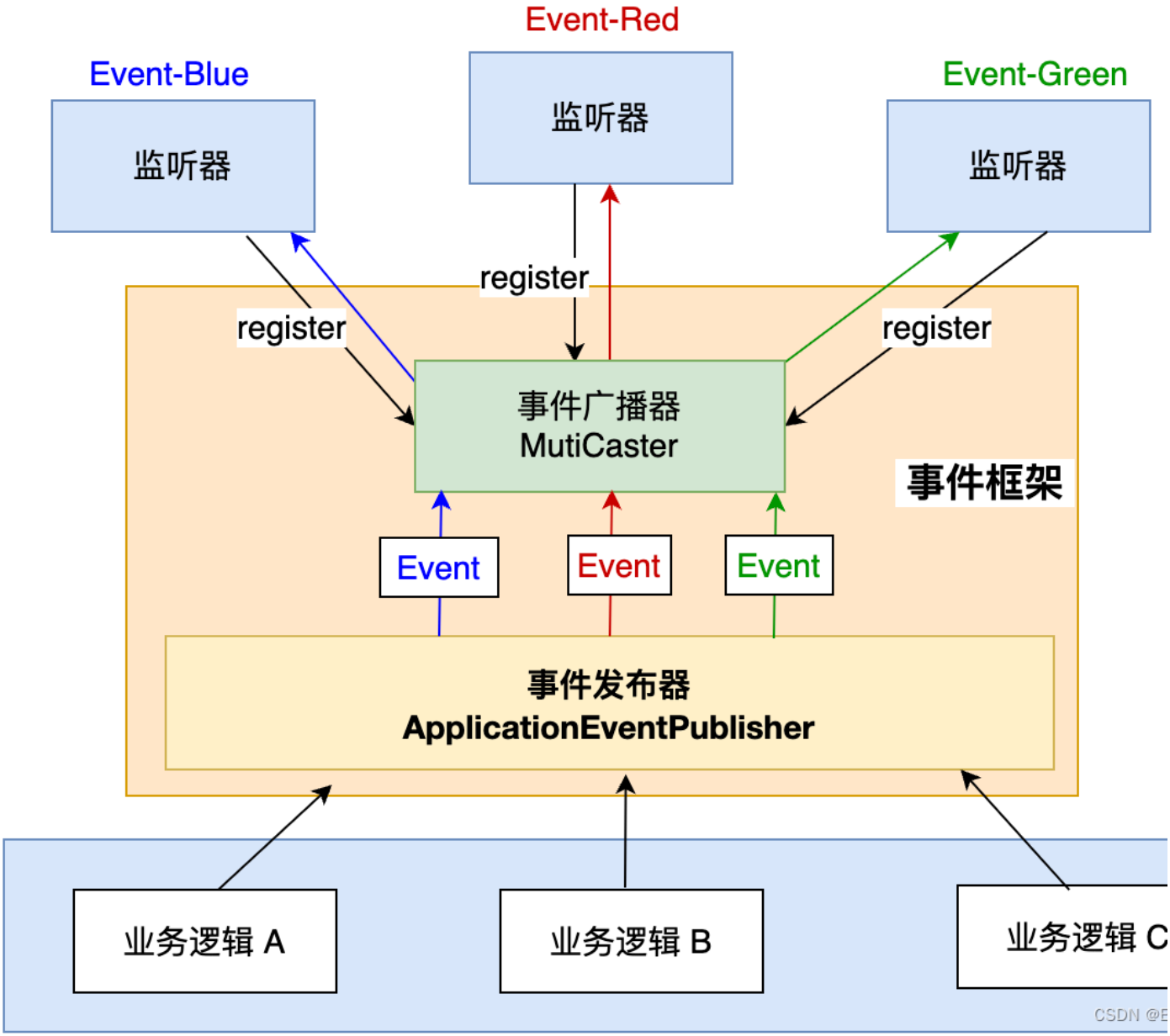
本文介绍Spring的事件机制，包括使用方式、注意事项以及实现原理，重心在于介绍事件机制的实现原理。介绍原理时会参考 Spring源码 ，对这部分者请订阅Spring系列。

专题刚开始，后续会不断更新，预计每周末更新一篇。

1.事件机制

> 本文介绍的事件机制指 Spring框架 提供的事件能力，不涉及Java和Guava的事件机制。

事件机制由事件广播器、事件监听器、事件及事件发布器等组件及其行为组合形成，如下图所示：



组件间的交互流程为：

(1) 监听器向事件广播器注册(订阅)，携带关心的事件信息；

Ewen Seong 已关注

(2) 用户通过事件发布器向事件广播器发送事件；
(3) 事件广播器获取相关的监听器列表，并向监听器分别发送通知；
(4) 监听器收到事件后，进行响应和处理；

上述过程中，发送消息的业务逻辑与事件监听器逻辑处于相互独立的状态；因此，事件机制可用于代码解耦，使得原本处于相互调用的依赖关系转向依架。

> 《Java 设计模式》
> 观察者模式：定义对象之间的一种一对多的依赖关系，使得每当一个对象的状态发生改变时其相关的依赖对象皆得到通知并被自动更新。

事件机制本质上是观察者模式的一种实现：事件广播器与事件监听器处于一对多的依赖关系，当对象状态变化时—即事件到了广播器，广播器会获取相并对其进行通知。

2.使用方式

Spring为事件机制提供了友好的API，前后共提供两种使用方式：基于接口和基于注解。

2.1 事件广播器：

可以使用Spring框架默认的广播器，也可以自定义，参考3.2章节。

2.2 事件监听器：

基于ApplicationListener接口方式：
监听器需要显式实现ApplicationListener接口以及指定事件类型，并实现onApplicationEvent接口：

```
@Slf4j
@Component
public class MyInterfaceEventListener implements ApplicationListener<MyApplicationEvent> {
    @Override
    public void onApplicationEvent(MyApplicationEvent event) {
        LOGGER.info("[Interface] event is {}.", event);
    }
}
```

其中事件类型必须为ApplicationEvent的子类，此时为MyApplicationEvent：

```
public class MyApplicationEvent extends ApplicationEvent {
    public MyApplicationEvent(Object source) {
        super(source);
    }
}
```

基于@EventListener注解方式

```
@Slf4j
@Component
public class MyAnnotationEventListener {
    @EventListener(ObjectEvent.class)
    public void onObjectEvent(ObjectEvent objectEvent) {
        LOGGER.info("[Annotation] event is {}.", objectEvent);
    }
}
```

在方法上添加@EventListener注解实现监听功能；其中：@EventListener(ObjectEvent.class)可以简化为@EventListener。相比接口使用方式，注解监需要继承ApplicationEvent(可以是任意类型)。

2.3 事件发布：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class EventTest {
    @Autowired
    private ApplicationEventPublisher publisher;

    @Test
    public void test() {
```

```
        publisher.publishEvent(new ObjectEvent().setName("test-object-event-1"));
        publisher.publishEvent(new MyApplicationEvent("test-application-event-2"));
    }

}
```

测试用例的输出结果：

```
rent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
st.EventTest                 : Started EventTest in 9.774 seconds (JVM running for 11.36)
.MyAnnotationEventListener   : [Annotation] event is ObjectEvent(name=test-object-event-1).
.MyInterfaceEventListener    : [Interface] event is com.seong.model.MyApplicationEvent[source=test-application-
```

2.4 使用事项：

Spring事件机制的实现依赖于IOC容器，因此需要保证所有监听器都被注入到IOC容器中。另外，使用Spring框架提供的事件机制时需要注意：事件广
当前线程同步执行，因此会阻塞整个事务。
根据业务场景不同可以选择使用**章节3.2**中方式为其配置线程池，也可以选择对监听方法添加异步注解@Async，如下所示：

```
@Slf4j
@Component
public class MyInterfaceEventListener implements ApplicationListener<MyApplicationEvent> {
    @Override
    @Async
    public void onApplicationEvent(MyApplicationEvent event) {
        LOGGER.info("[Interface] event is {}.", event);
    }
}
```

> 关于**@Async**注解的实现原理不是本文的主线，不在此介绍，将在**Spring系列**专题的后续文章中介绍，敬请期待。

3.实现原理

3.1 事件监听器

Spring框架基于注解的事件机制和基于接口的事件机制的实现原理不同，这里分别进行介绍。

基于接口的实现原理：

基于ApplicationListener接口实现事件机制依赖于ApplicationListenerDetector的能力。

ApplicationListenerDetector类作用：

```
public class ApplicationListenerDetector implements DestructionAwareBeanPostProcessor, MergedBeanDefinitionPostProcessor {

    private final transient Map<String, Boolean> singletonNames = new ConcurrentHashMap<>(256);

    @Override
    public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?> beanType, String beanName) {
        if (ApplicationListener.class.isAssignableFrom(beanType)) {
            this.singletonNames.put(beanName, beanDefinition.isSingleton());
        }
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) {
        if (bean instanceof ApplicationListener) {
            Boolean flag = this.singletonNames.get(beanName);
            if (Boolean.TRUE.equals(flag)) {
                this.applicationContext.addApplicationListener((ApplicationListener<?>) bean);
            } else if (Boolean.FALSE.equals(flag)) {
                this.singletonNames.remove(beanName);
            }
        }
        return bean;

```

```
    }
}
```

从类的继承关系分析：ApplicationListenerDetector是MergedBeanDefinitionPostProcessor接口实现类，即ApplicationListenerDetector被注入到容器的beanPostProcessor后，后续引入的Bean在完成实例化后且属性设置前会调用ApplicationListenerDetector的 `postProcessMergedBeanDefinition` 方法；MergedBeanDefinitionPostProcessor又是BeanPostProcessor接口继承者，即ApplicationListenerDetector也是BeanPostProcessor，当后续引入的Be设置、初始化后会调用 `postProcessAfterInitialization` 方法。这块不熟的读者可以参考：Spring系列-1启动流程

接下来按照流程执行顺序介绍一下这两个方法的作用：
当Bean对象完成实例化后，调用 `postProcessMergedBeanDefinition` 方法：

```
@Override
public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?> beanType, String beanName) {
    if (ApplicationListener.class.isAssignableFrom(beanType)) {
        this.singletonNames.put(beanName, beanDefinition.isSingleton());
    }
}
```

将实现ApplicationListener接口的Bean及其是否单例信息存入singletonNames中。

当Bean对象完成属性注入、初始化后调用 `postProcessAfterInitialization` 方法：

```
@Override
public Object postProcessAfterInitialization(Object bean, String beanName) {
    if (bean instanceof ApplicationListener) {
        Boolean flag = this.singletonNames.get(beanName);
        if (Boolean.TRUE.equals(flag)) {
            this.applicationContext.addApplicationListener((ApplicationListener<?>) bean);
        } else if (Boolean.FALSE.equals(flag)) {
            this.singletonNames.remove(beanName);
        }
    }
    return bean;
}
```

将ApplicationListener实现类的Bean对象通过 `this.applicationContext.addApplicationListener((ApplicationListener<?>) bean);` 添加到Spring容器性及注册到事件广播器上：

```
public void addApplicationListener(ApplicationListener<?> listener) {
    if (this.applicationEventMulticaster != null) {
        // 广播器不为空时-直接向广播器注册
        this.applicationEventMulticaster.addApplicationListener(listener);
    }
    // 添加到Spring容器中的applicationListeners属性中，后续整体向广播器注册
    // 防止，此时applicationEventMulticaster为空
    this.applicationListeners.add(listener);
}
```

这里IF判断存在的根源是 ApplicationListenerDetector在Spring容器启动过程中被注册到容器的BeanPostProcessor中的时机。

Spring容器引入ApplicationListenerDetector：

容器启动过程中在不同阶段会进行共计两次注册，这里需要结合Spring容器的启动流程进行介绍。

```
    // 该部分省略了try-catch异常处理逻辑、初始化国际化、扩展接口等，主要用来说明主体流程
    @Override
public void refresh() throws BeansException, IllegalStateException {
    // 创建Beanfactory并加载BeanDefinition等操作
    ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

    // 对beanFactory的预处理
    //【⚠向Spring容器注册BeanPostProcessor：ApplicationListenerDetector】
    prepareBeanFactory(beanFactory);

    // 调用所有的BeanFactoryPostProcessor
    invokeBeanFactoryPostProcessors(beanFactory);

    // 向Spring容器注册BeanPostProcessor
```

```
14
15      //【⚠向Spring容器注册BeanPostProcessor：ApplicationListenerDetector】
16      registerBeanPostProcessors(beanFactory);
17
18      // 初始化事件广播器
19      initApplicationEventMulticaster();
20
21      // 注册ApplicationContext中的监听器
22      registerListeners();
23
24      // 实例化所有的非Lazy单例Bean对象
25      finishBeanFactoryInitialization(beanFactory);
26  }
```

> 代码中注释的**【向Spring容器注册BeanPostProcessor】**表示将BeanPostProcessor实现类的Bean对象添加到IOC容器的beanPostProcessors属
> **final List beanPostProcessors = new CopyOnWriteArrayList();**

后续Bean的初始化前后会依次调用 `beanPostProcessors` 中的BeanPostProcessor对象的接口，这里不再赘述，**Spring系列**专题文章中有详细介绍。
如前文所述：ApplicationListenerDetector是MergedBeanDefinitionPostProcessor(和BeanDefinitionPostProcessor)接口实现类，该类的核心功能发生在
的初始化完成后。对于业务使用者而言，在 `finishBeanFactoryInitialization(beanFactory);` 过程；也就是说，只要保证【向Spring容器注册
BeanPostProcessor】操作在 `finishBeanFactoryInitialization(beanFactory);` 之前执行即可。
这里可以深究一下为什么需要注册两次，这里对这部分细节不关心的读者可以选择跳过。

在 `prepareBeanFactory(beanFactory)` 阶段【向Spring容器注册BeanPostProcessor】：

```
1  protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
2      // 省略其他无关代码 ...
3
4      // Register early post-processor for detecting inner beans as ApplicationListeners.
5      beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));
6  }
```

在准备阶段执行，是为了控制ApplicationListenerDetector的影响范围，保证后续引入的ApplicationListener类型的Bean对象不会丢失，包括Spring自身
下图所示：

以及业务引入的(作为ApplicationListener，同时也是BeanFactoryPostProcessor或者BeanPostProcessor接口的实现类)对象。在 `registerListeners()`
就是这些监听器。

在 `registerBeanPostProcessors(beanFactory)` 阶段【向Spring容器注册BeanPostProcessor】：

```
1  public static void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory, AbstractApplicationContext application
2      // 简化后的伪代码：
3      registerBeanPostProcessors(beanFactory, "其他所有BeanPostProcessor");
4
5      // Re-register post-processor for detecting inner beans as ApplicationListeners,
6      // moving it to the end of the processor chain (for picking up proxies etc).
7      beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));
8  }
```

如注释：再次注册的目的是为了将ApplicationListenerDetector放在所有BeanPostProcessor最后，以获取代理对象等。因为Spring AOP也是基于
BeanPostProcessor，ApplicationListenerDetector需要注册代理之后的ApplicationListener对象，所以Detect操作需要在代理操作完成后执行，即放在
方案。

基于注解的实现原理：

基于注解的实现原理依赖EventListenerMethodProcessor和DefaultEventListenerFactory的能力。

EventListenerMethodProcessor类作用：

```
1
2  public class EventListenerMethodProcessor
3          implements SmartInitializingSingleton, ApplicationContextAware, BeanFactoryPostProcessor {
4
5      @Override
6      public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
7
```

```
    @Override
    public void afterSingletonsInstantiated() { // ...}
}
```

从类的继承关系分析：EventListenerMethodProcessor是SmartInitializingSingleton和BeanFactoryPostProcessor接口实现类：(1) 在容器启动之初调用BeanFactoryPostProcessor的postProcessBeanFactory接口；(2) 在所有的Bean初始化完成后执行SmartInitializingSingleton的afterSingletonsInstantia
接下来按照流程执行顺序介绍一下这两个方法的作用：
容器启动之初，调用 `postProcessBeanFactory` 方法：

```
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    this.beanFactory = beanFactory;
    Map<String, EventListenerFactory> beans = beanFactory.getBeansOfType(EventListenerFactory.class, false, false);
    List<EventListenerFactory> factories = new ArrayList<>(beans.values());
    AnnotationAwareOrderComparator.sort(factories);
    this.eventListenerFactories = factories;
}
```

该部分核心逻辑是初始化EventListenerFactory工厂，默认为DefaultEventListenerFactory. 一般业务侧并不会实现EventListenerFactory，即 `postProcessBeanFactory` 可以理解为从IOC容器中获取DefaultEventListenerFactory对象并复制给eventListenerFactories属性.
在所有的Bean初始化完成后执行SmartInitializingSingleton的afterSingletonsInstantiated接口：

```
public void afterSingletonsInstantiated() {
    ConfigurableListableBeanFactory beanFactory = this.beanFactory;
    String[] beanNames = beanFactory.getBeanNamesForType(Object.class);
    for (String beanName : beanNames) {
        Class<?> type = AutoProxyUtils.determineTargetClass(beanFactory, beanName);
        processBean(beanName, type);
    }
}
```

该接口逻辑比较简单，对IOC容器中的所有Bean对象获取beanName和type后调用 `processBean(beanName, type)` 方法。下面是processBean方法的主体

```
// 省略异常逻辑和日志部分，并简化逻辑体现主体功能
//引入`nonAnnotatedClasses`属性是为了减少不必要的判断;

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 = annotatedMethods = MethodIntrospector.selectMethods(targetType,
                (MethodIntrospector.MetadataLookup<EventListener>) method ->
                        AnnotatedElementUtils.findMergedAnnotation(method, EventListener.class));

        if (CollectionUtils.isEmpty(annotatedMethods)) {
            this.nonAnnotatedClasses.add(targetType);
            return;
        }
        // Non-empty set of methods
        ConfigurableApplicationContext context = this.applicationContext;
        List<EventListenerFactory> factories = this.eventListenerFactories;
        for (Method method : annotatedMethods.keySet()) {
            Method methodToUse = AopUtils.selectInvocableMethod(method, context.getType(beanName));
            ApplicationListener<?> applicationListener =
                    factory.createApplicationListener(beanName, targetType, methodToUse);
            ((ApplicationListenerMethodAdapter) applicationListener).init(context, this.evaluator);
            context.addApplicationListener(applicationListener);
            break;
        }
    }
}
```

调用 `MethodIntrospector.selectMethods` 接口遍历指定类型的所有方法，对于每个方法根据 `AnnotatedElementUtils.findMergedAnnotation(method, EventListener.class)` 获得@EventListener注解合并的属性值(@EventListener不允许重复注解)。
接口返回Map<Method, EventListener>类型的annotatedMethods集合对象，其中key为方法对象，value为注解在方法上的@EventListener解析生成的

如果annotatedMethods为空表明该类中没有方法被@EventListener注解—直接返回；否则遍历annotatedMethods对象，并通过DefaultEventListenerF个被注解的方法生成一个ApplicationListenerMethodAdapter类型的ApplicationListener事件监听器，在初始化后注册到Spring容器：

```
for (Method method : annotatedMethods.keySet()) {
    // 找到一个可以被调用的目标方法Method，如果本身可调用-直接返回
    Method methodToUse = AopUtils.selectInvocableMethod(method, context.getType(beanName));
    // 使用事件工厂创建事件对象
    ApplicationListener<?> applicationListener =
            factory.createApplicationListener(beanName, targetType, methodToUse);
    // 初始化：赋值
    ((ApplicationListenerMethodAdapter) applicationListener).init(context, this.evaluator);
    // ⚠注册到Spring容器
    context.addApplicationListener(applicationListener);
    break;
}
```

这里，再次进入Spring容器的 `addApplicationListener` 方法：

```
public void addApplicationListener(ApplicationListener<?> listener) {
    Assert.notNull(listener, "ApplicationListener must not be null");
    if (this.applicationEventMulticaster != null) {
        this.applicationEventMulticaster.addApplicationListener(listener);
    }
    this.applicationListeners.add(listener);
}
```

上述代码 `context.addApplicationListener(applicationListener);` 发生在所有非Lazy单例Bean被初始化后，即applicationEventMulticaster已在 `initApplicationEventMulticaster()` 步骤被初始化过—不为空：此时同时向事件广播器进行了注册。

DefaultEventListenerFactory类作用：

上述过程涉及DefaultEventListenerFactory为每个被@EventListener注解的方法生产ApplicationListenerMethodAdapter对象的过程：

```
public class DefaultEventListenerFactory implements EventListenerFactory, Ordered {
    //... 省略无关逻辑
    @Override
    public ApplicationListener<?> createApplicationListener(String beanName, Class<?> type, Method method) {
        return new ApplicationListenerMethodAdapter(beanName, type, method);
    }
}
```

ApplicationListenerMethodAdapter类可以简化如下：

```
public class ApplicationListenerMethodAdapter implements GenericApplicationListener {
    public ApplicationListenerMethodAdapter(String beanName, Class<?> targetClass, Method method) {
        this.beanName = beanName;
        // 对方法进行桥接处理
        this.method = BridgeMethodResolver.findBridgedMethod(method);
        //...
    }

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

当onApplicationEvent接口被调用时，ApplicationListenerMethodAdapter对象依次调用processEvent、doInvoke方法，通过反射调用目标方法。

3.2 事件广播器

Spring容器启动时在AbstractApplicationContext的refresh()方法中存在初始化事件广播器步骤：

```java
@Override
public void refresh() throws BeansException, IllegalStateException {
    //...
    // ⚠初始化事件广播器
    initApplicationEventMulticaster();
    //...
    // 注册监听器
    registerListeners();
    // 实例化所有非Lazy单例Bean
    finishBeanFactoryInitialization(beanFactory);
}
```

进入 `initApplicationEventMulticaster()` 方法：

```java
protected void initApplicationEventMulticaster() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (beanFactory.containsLocalBean("applicationEventMulticaster")) {
        this.applicationEventMulticaster =
                beanFactory.getBean("applicationEventMulticaster", ApplicationEventMulticaster.class);
    } else {
        this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
        beanFactory.registerSingleton("applicationEventMulticaster", this.applicationEventMulticaster);
    }
}
```

逻辑比较简单，如果IOC工厂没有" `applicationEventMulticaster` "名称的Bean对象，就new一个SimpleApplicationEventMulticaster对象并注册到IOC容
需要注意：Spring默认创建的SimpleApplicationEventMulticaster对象的内置线程池为空：

```java
private Executor taskExecutor;

// 构造函数未对taskExecutor属性进行设置
public SimpleApplicationEventMulticaster(BeanFactory beanFactory) {
    setBeanFactory(beanFactory);
}
```

业务上可以通过注入Bean的方式进行，可以参考开源代码：

示例4: simpleApplicationEventMulticaster

▲ 点赞 2 ▼

```
import org.springframework.context.event.SimpleApplicationEventMulticaster; //导入依
package包/类
@Bean(name = "applicationEventMulticaster")
public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
    SimpleApplicationEventMulticaster eventMulticaster = new SimpleApplicationEvent
ticaster();

    taskExecutor = new ThreadPoolTaskExecutor();
    taskExecutor.setThreadNamePrefix("asyncEventExecutor-");
    taskExecutor.setCorePoolSize(4);
    taskExecutor.initialize();

    eventMulticaster.setTaskExecutor(taskExecutor);
    return eventMulticaster;
}
```

开发者ID:jeperon，项目名称:freqtrade-java，代码行数:13，代码来源:FreqTradeConfiguration.java

示例5: simpleApplicationEventMulticaster

▲ 点赞 2 ▼

```
import org.springframework.context.event.SimpleApplicationEventMulticaster; //导入依
package包/类
@Bean(name = "applicationEventMulticaster")
public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
    multicasterExecutor.initialize();
    SimpleApplicationEventMulticaster eventMulticaster = new SimpleApplicationEvent
ticaster();
    eventMulticaster.setTaskExecutor(multicasterExecutor);
    return eventMulticaster;
}
```

开发者ID:FlowCI，项目名称:flow-platform，代码行数:8，代码来源:AppConfig.java

CSD

SimpleApplicationEventMulticaster类功能介绍

核心是对外通过监听器注册/注销接口、事件发布的接口

事件发布接口

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

Ewen Seong 已关注

```
        if (executor != null) {
            executor.execute(() -> invokeListener(listener, event));
        } else {
            invokeListener(listener, event);
        }
    }
}
```

如上所示，对外提供两个重载的multicastEvent方法，接受ApplicationEvent类型的参数以及接受ApplicationEvent和ResolvableType组合的参数；Reso
如果为空，框架会使用 `new ResolvableType(clazz)` 为ApplicationEvent实现类生成对应的ResolvableType对象。其中，核心处理逻辑在 `void multicas
ApplicationEvent event, @Nullable ResolvableType eventType)` 中：
根据**事件类型**以及**事件对象**获取相关(匹配、感兴趣)的**监听器对象集合**，并遍历该集合调用invokeListener方法；另外，如果用户为SimpleApplicationEventMulticaster的taskExecutor属性设置类线程池对象，则将调用invokeListener方法的逻辑提交给taskExecutor线程池对象处理。
跟进invokeListener(ApplicationListener<?> listener, ApplicationEvent event)：

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

这里给予用户在框架内部处理异常的能力，而不是简单向外抛出；用法同taskExecutor，即为SimpleApplicationEventMulticaster对象的errorHandler属
ErrorHandler对象。
跟进doInvokeListener(ApplicationListener listener, ApplicationEvent event)：

```
// 省略try-catch逻辑
private void doInvokeListener(ApplicationListener listener, ApplicationEvent event) {
    listener.onApplicationEvent(event);
}
```

逻辑较为简单：调用监听器的onApplicationEvent方法。

监听器注册/注销接口

监听器注册与注销是互为相反的操作，因此这里仅展示注册逻辑：

```
@Override
public void addApplicationListener(ApplicationListener<?> listener) {
    synchronized (this.retrievalMutex) {
        Object singletonTarget = AopProxyUtils.getSingletonTarget(listener);
        if (singletonTarget instanceof ApplicationListener) {
            this.defaultRetriever.applicationListeners.remove(singletonTarget);
        }
        this.defaultRetriever.applicationListeners.add(listener);
        this.retrieverCache.clear();
    }
}

@Override
public void addApplicationListenerBean(String listenerBeanName) {
    synchronized (this.retrievalMutex) {
        this.defaultRetriever.applicationListenerBeans.add(listenerBeanName);
        this.retrieverCache.clear();
    }
}
```


接受ApplicationListener对象或beanName作为参数，将其分别添加到defaultRetriever属性的applicationListeners和applicationListenerBeans中；可以
SimpleApplicationEventMulticaster的defaultRetriever属性的内部结构：

```
public abstract class AbstractApplicationEventMulticaster implements ApplicationEventMulticaster, ... {
    private final ListenerRetriever defaultRetriever = new ListenerRetriever(false);
}

private class ListenerRetriever {
    // 直接存放ApplicationListener对象的集合
    public final Set<ApplicationListener<?>> applicationListeners = new LinkedHashSet<>();

    // 存放ApplicationListener对象的beanName的集合
    public final Set<String> applicationListenerBeans = new LinkedHashSet<>();
}
```

注意框架使用两个属性保存监听器信息；这是为了照顾到已完成实例化的Bean对象和部分BeanDefintion未被实例化的场景。
存放的方式决定着读取的方式，当获取全量监听器或者根据事件类型和事件对象获取相关的监听器时，需要从这两个属性中获取并集。以前者为例：

```
public Collection<ApplicationListener<?>> getApplicationListeners() {
    List<ApplicationListener<?>> allListeners = new ArrayList<>(
            this.applicationListeners.size() + this.applicationListenerBeans.size());
    // ⚠1.添加applicationListeners集合中的所有监听器对象
    allListeners.addAll(this.applicationListeners);

    if (!this.applicationListenerBeans.isEmpty()) {
        BeanFactory beanFactory = getBeanFactory();
        for (String listenerBeanName : this.applicationListenerBeans) {
            // ⚠2.从IOC容器中根据beanName获取监听器对象
            ApplicationListener<?> listener = beanFactory.getBean(listenerBeanName, ApplicationListener.class);
            // ⚠3.取并集
            if (this.preFiltered || !allListeners.contains(listener)) {
                allListeners.add(listener);
            }
        }
    }

    // ⚠4.排序
    if (!this.preFiltered || !this.applicationListenerBeans.isEmpty()) {
        AnnotationAwareOrderComparator.sort(allListeners);
    }
    return allListeners;
}
```

3.3 事件发布器

Spring容器提供的事件发布器接口如下所示；

```
package org.springframework.context;

@FunctionalInterface
public interface ApplicationEventPublisher {
    default void publishEvent(ApplicationEvent event) {
        publishEvent((Object) event);
    }

    void publishEvent(Object event);
}
```

如代码所示，核心方法为 `void publishEvent(Object event);`
AbstractApplicationContext对ApplicationEventPublisher接口提供了实现：

```
@Override
public void publishEvent(Object event) {
    publishEvent(event, null);
}
```

跟进protected void publishEvent(Object event, @Nullable ResolvableType eventType)：

```java
protected void publishEvent(Object event, @Nullable ResolvableType eventType) {
    ApplicationEvent applicationEvent;
    if (event instanceof ApplicationEvent) {
        applicationEvent = (ApplicationEvent) event;
    } else {
        applicationEvent = new PayloadApplicationEvent<>(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);
        }
    }
}
```

方法的主线逻辑较为清晰：
[1] 如果入参event的类型不是ApplicationEvent，使用PayloadApplicationEvent进行封装；
[2] 从applicationEventMulticaster属性获取事件广播器，并调用广播事件接口；
[3] 如果Spring容器的父容器不为空，调用父容器的事件发布接口进行事件发布；

技术派Spring事件监听机制及原理_spring事件监听原理
Spring事件监听机制是Spring框架中的一种重要技术,允许组件之间进行松耦合通信。通过使用事件监听机制,应用程序的各个组件可以在其他组件不直接引用的情况下,相互

Spring中的事件机制 cgl_dong
Spring中的事件机制 Spring对事件机制也提供了支持，一个事件被发布后，被对应的监听器监听到，执行对应方法。 Spring内已经提供了许多事件，ApplicationEvent可以

Spring 事件机制 Rebel_z
事件事件发布事件监听器。如何实现一个事件机制，应用的场景，搭配 @Async 注解实现异步的操作等等。希望对大家有所帮助。如何定义一个事件新增一个类，继承我们

Spring事件Event详解_spring event事件
一、Spring的事件机制核心组件 二、Spring事件的基本步骤 三、ApplicationEventMulticaster 四、Spring event事件使用场景 Spring框架提供了事件驱动的编程模型,使开发

spring-5.3.9-dist.zip
在Spring 5.3.9这个版本中，我们能看到一系列的改进和优化。 Spring 框架的核心组件包括IoC（Inversion of Control，控制反转）容器、AOP（Aspect Oriented Programm

Tomcat和Spring中的事件机制深入讲解
总结来说，无论是Tomcat还是Spring，事件机制都是为了在组件间提供一种灵活的通知机制，使得系统在特定情况下能够自动调用合适的处理代码，降低了组件之间的依赖

spring-framework-5.1.8.RELEASE.zip
在5.1.8.RELEASE这个版本中，Spring框架提供了一系列的增强和改进，旨在提高开发效率和应用的可维护性。 首先，让我们深入了解Spring的依赖注入（DI）机制。DI是

spring-framework-6.0.11.zip
9. **性能优化**：通过一系列的内部重构和优化，Spring Framework 6.0.11在启动速度、内存占用和处理能力上都有显著提升，为大型企业级应用提供更强健的基础。 10.

Spring基于事件驱动模型的订阅发布模式代码实例详解 u014746965
转载地址：http://blog.csdn.net/yaerfeng/article/details/27683813 Spring提供的事件驱动模型/观察者抽象 首先看一下Spring提供的事件驱动模型体系图： 事件 具体代表者

Spring的事件机制 weixin_34234823
当把一个事件发布到Spring提供的ApplicationContext中，被监听器侦测到，就会执行对应的处理方法。 事件本身 事件是一个自定义的类，需要继承Spring提供的Applicatio

Spring中的事件处理机制 qq_42372935的
Spring中的事件处理机制

Spring 中的事件机制 rickiyang
说到事件机制，可能脑海中最先浮现的就是日常使用的各种 listener，listener去监听事件源，如果被监听的事件有变化就会通知listener，从而针对变化做相应的动作。这些

spring 事件机制 梵
【事件派发器】 * 1）、容器启动：refresh(); * 2）、initApplicationEventMulticaster();初始化ApplicationEventMulticaster； * 1）、先去容器中找有没有id="a

Spring的事件机制详解 weixin_34380781
同步事件和异步事件 同步事件:在一个线程里,按顺序执行业务,做完一件事再去做下一件事. 异步事件:在一个线程里,做一个事的同事,可以另起一个新的线程执行另一件事

spring-boot-starter-rxtx
spring-boot-starter-rxtx是基于Spring Boot框架的一个启动器，用于与Java串口通信库RXTX进行集成。通过集成RXTX库，spring-boot-starter-rxtx可以提供在Java程序中对

Ewen Seong
码龄6年 暂无认证

83	3464	1万+	16万+	
原创	周排名	总排名	访问	等级
1685	2546	800	31	817
积分	粉丝	获赞	评论	收藏

热门文章

分类专栏

最新评论

最新文章

2024年 35篇 2023年 18篇
2022年 17篇 2021年 13篇

目录