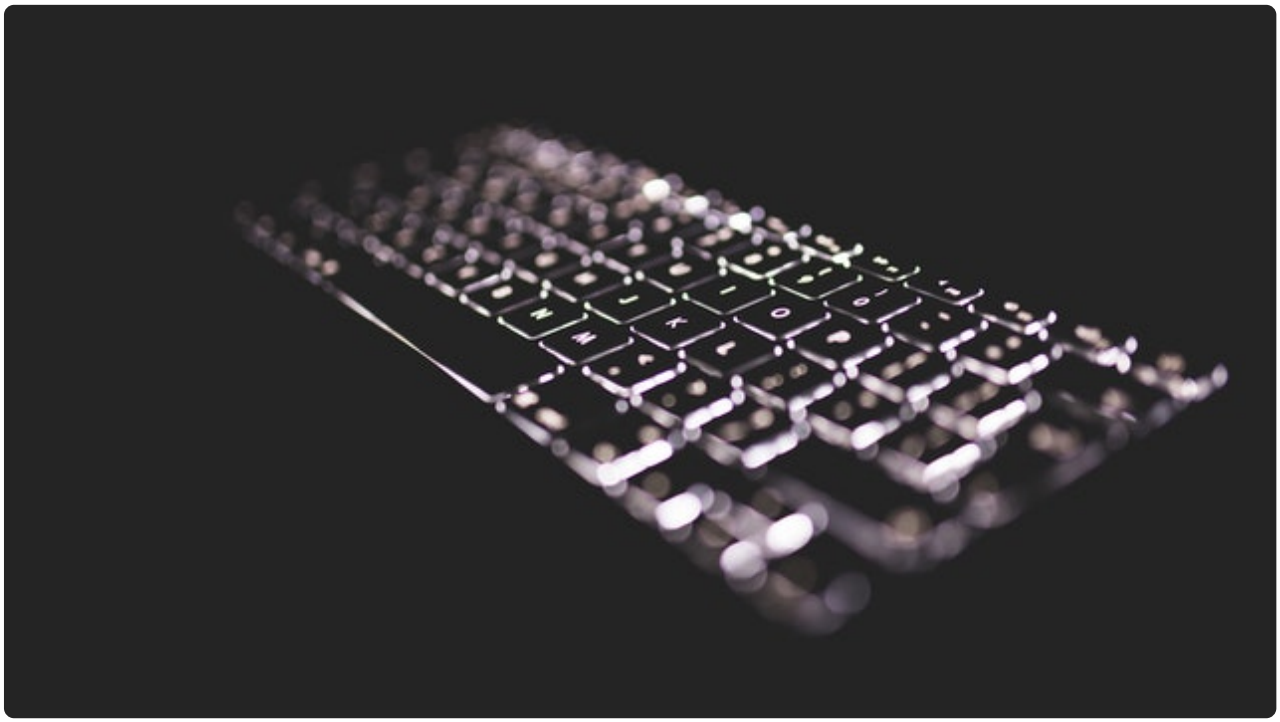
11 spring IoC容器中事件event消息的发送和接收内部工作原理揭秘

更新时间：2020-08-04 12:11:42



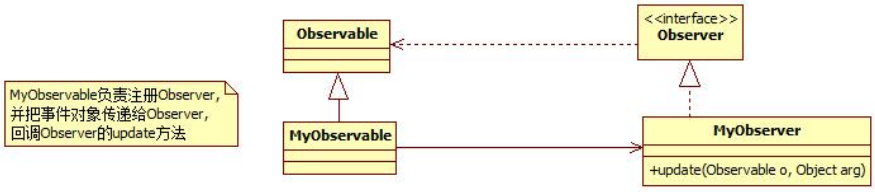
> 先相信你自己，然后别人才会相信你。——屠格涅夫

背景

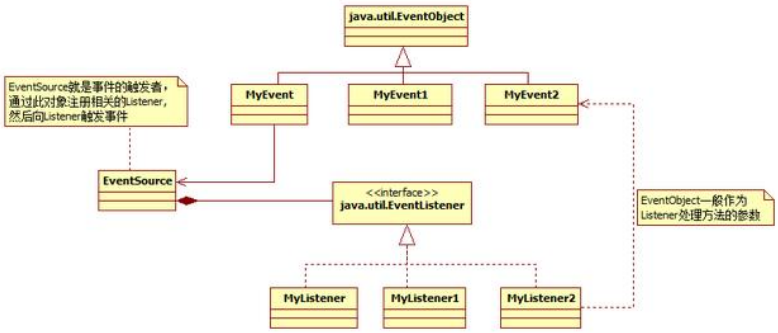
消息机制是使用消息通知的方式，解耦生产者与消费者。编程上体现的是职责分割，使得消息处理的扩展性得到增强，符合设计原则中的单一职责以及开闭原则。

Java 提供了两种解决方式：观察者模式和监听器模式。

观察者模式：观察者（Observer）相当于事件监听者，被观察者（Observable）相当于事件源和事件，执行逻辑时通知 Observer 即可触发 Observer 的 update，同时可传被观察者和参数。



监听器模式：事件源经过事件的封装传给监听器，当事件源触发事件后，监听器接收到事件对象可以回调事件的方法。



监听的整个处理过程是这样的：事件源可以注册事件监听器对象，并可以向事件监听器对象发送事件对象.事件发生后，事件源将事件对象发给已经注册的所有事件监听器。

那么 Spring 中是怎么实现消息机制的呢？

Spring 的消息机制

Spring 系统消息使用了监听器模式，为什么使用监听器而不是观察者模式呢？

猜测缘由：

观察者模式的缺点是其中的每个 Observer 接收的消息都是一样的，内部需要通过类似 instanceof 的方式来区分不同类型的消息。而监听器模式中监听器有不同的类型，监视不同类型的消息，因此不用区分消息类型。

Spring 的系统事件消息还实现了同步和异步的两种不同方式。原理后面内容中会讲解到。

Spring 实现消息传递机制主要依赖 3 个对象：事件，事件源，事件监听器。

1. Spring 事件

Spring 事件一般继承自 ApplicationEvent 对象，常用系统事件有：

ContextRefreshedEvent

ContextStartedEvent

ContextStoppedEvent

ContextClosedEvent

开发者也可以通过继承 ApplicationEvent 自行定义。

2. Spring 事件源：

Spring 事件源就是触发事件的源头，不同的事件源会触发不同的事件类型。

ApplicationEventPublisher 是抽象层，定义了 publishEvent(ApplicationEvent event) 方法，具体实现在公共抽象类 AbstractApplicationContext 中：

finishRefresh() 调用 publishEvent(new ContextRefreshedEvent(this));

doClose() 调用 publishEvent(new ContextClosedEvent(this));

start() 调用 publishEvent(new ContextStartedEvent(this));

stop() 调用 publishEvent(new ContextStoppedEvent(this));

3. 事件监听器：

事件监听器负责监听事件源发出的事件。一个事件监听器通常实现 java.util.EventListener 这个标识接口。监听器对象随后会根据事件对象内的相应方法响应这个事件。

ApplicationListener 继承了 EventListener，并在 AbstractApplicationContext.java#registerListeners() 注册了监听器：

```
getApplicationEventMulticaster().addApplicationListener(listener);
```

其中 ApplicationEventMulticaster 管理多个 ApplicationListener，并发送消息给 ApplicationListener。

Spring 事件示例

上面说了那么多，下面我们通过一个自定义的 Spring 事件来看看 Spring 消息是如何传递的？

自定义事件消息：

```
private static class MyEvent extends ApplicationEvent{
    private final String msg;

    public MyEvent (Object source, String msg) {
        super(source);
        this.msg = msg;
    }

    public String getMsg () {
        return msg;
    }
}
```

消息发送者：

```
    private static class MyEvenPublisherBean{
        @Autowired
        ApplicationEventPublisher publisher;

        public void sendMsg(String msg){
            publisher.publishEvent(new MyEvent(this, msg));
            System.out.println("publish event :"+msg);

        }

    }
```

消息接收者：

```
private static class AListenerBean {

    @EventListener
    public void onMyEvent (MyEvent event) {
        System.out.println("event received: "+event.getMsg());
        System.out.println("source: "+event.getSource());
    }
}
```

测试类（包含上面所有类）：

```
public class CustomEventWithApplicationEvent {

    @Bean
    AListenerBean listenerBean () {
        return new AListenerBean();
    }

    @Bean
    MyEvenPublisherBean publisherBean () {
        return new MyEvenPublisherBean();
    }


    public static void main (String[] args) {

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(
                                CustomEventWithApplicationEvent.class);
        MyEvenPublisherBean bean = context.getBean(MyEvenPublisherBean.class);
        bean.sendMsg("A test message");

    }
```

上面代码我们自定义了 3 个对象，消息事件 MyEvent，消息发送者 MyEvenPublisherBean 和消息接收者 AListenerBean，实现了 Spring 消息发送，并用实例测试消息发送及消息接收并打印的过程。

Spring 事件消息工作原理

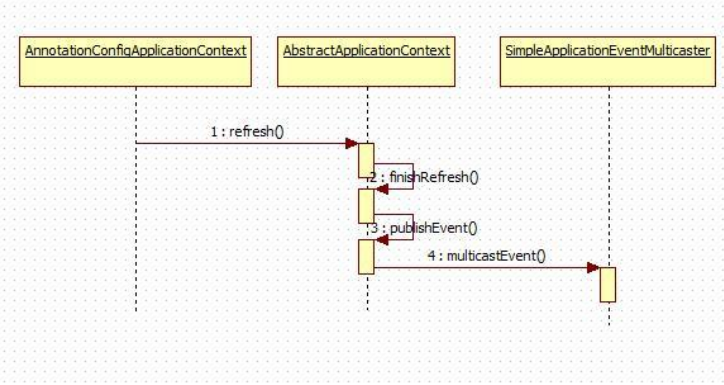
debug 上述程序，其中，

```
AnnotationConfigApplicationContext context =
```

```
new AnnotationConfigApplicationContext(CustomEventWithApplicationEvent.class);
```

触发初始化事件消息，此时没有 refresh 消息。

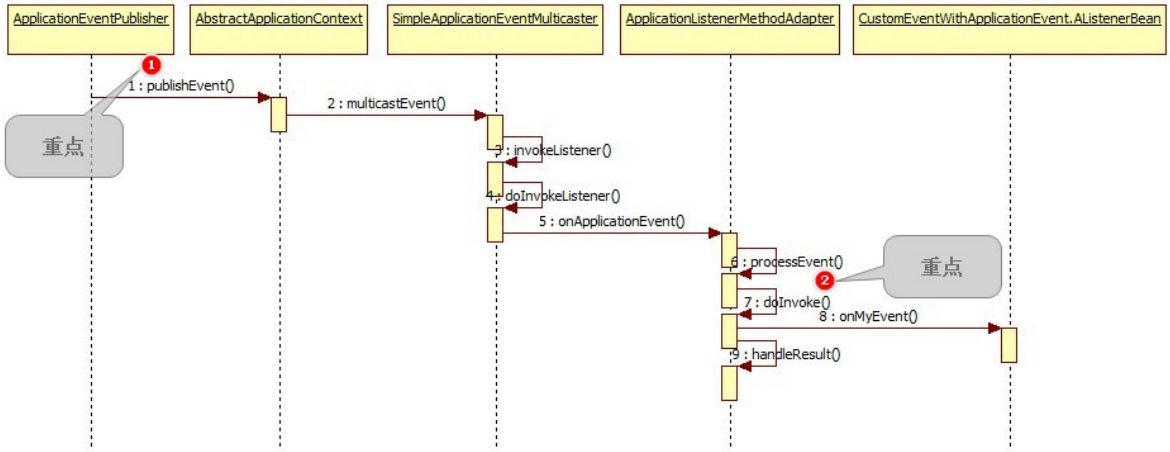
- 初始化时的 ContextRefreshedEvent：




```
MyEvenPublisherBean bean = context.getBean(MyEvenPublisherBean.class);
bean.sendMsg("A test message");
```

触发 sendMsg 方法，进而触发 publish 消息。

Spring 消息触发的完整过程：



其中重点的有两个，一个是发布事件：

```
/**
 * Publish the given event to all listeners.
 * @param event the event to publish (may be an {@link ApplicationEvent}
 * or a payload object to be turned into a {@link PayloadApplicationEvent})
 * @param eventType the resolved event type, if known
 * @since 4.2
 */
protected void publishEvent(Object event, @Nullable ResolvableType eventType) {
    Assert.notNull(event, "Event must not be null");

    // Decorate event as an ApplicationEvent if necessary
    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);
        }
    }
}
```

另一个触发 listener 的监听：

```
/**
 * Invoke the event listener method with the given argument values.
 */
@Nullable
protected Object doInvoke(Object... args) {
    Object bean = getTargetBean();
    // Detect package-protected NullBean instance through equals(null) check
    if (bean.equals(null)) {
        return null;
    }

    ReflectionUtils.makeAccessible(this.method);
    try {
        return this.method.invoke(bean, args);
    }
    catch (IllegalArgumentException ex) {
        assertTargetBean(this.method, bean, args);
        throw new IllegalStateException(getInvocationErrorMessage(bean, ex.getMessage(), args), ex);
    }
    catch (IllegalAccessException ex) {
        throw new IllegalStateException(getInvocationErrorMessage(bean, ex.getMessage(), args), ex);
    }
    catch (InvocationTargetException ex) {
        // Throw underlying exception
        Throwable targetException = ex.getTargetException();
        if (targetException instanceof RuntimeException) {
            throw (RuntimeException) targetException;
        }
        else {
            String msg = getInvocationErrorMessage(bean, "Failed to invoke event listener method", args);
            throw new UndeclaredThrowableException(targetException, msg);
        }
    }
}

/**
```

触发的方法后执行 listenerbean 的方法：

```
    private static class AListenerBean {

        @EventListener
        public void onMyEvent (MyEvent event) {
            System.out.println("event received: "+event.getMsg());
            System.out.println("source: "+event.getSource());
        }
    }
```

注意点：

前面提到，Spring 的系统事件消息还实现了同步和异步的两种不同方式。通过 debug 代码可以看到，异步方式通过后台线程池执行。

其中 1 是异步方式，2 是同步方式。

总结

Spring 的系统消息主要通过监听器模式实现的，监听器的三要素分别为事件，发布者，监听者。Spring 内部通过反射实现消息的监听。

}

10 扩展之Spring MVC中如何实现国际化i18n

12 @Autowired是如何工作的？ – Spring注解源码深度揭秘