

85 | 开源实战四（中）：剖析Spring框架中用来支持扩展的两种设计模式

王争 · 设计模式之美



上一节课中，我们学习了 Spring 框架背后蕴藏的一些经典设计思想，比如约定优于配置、低侵入松耦合、模块化轻量级等等。我们可以将这些设计思想借鉴到其他框架开发中，在大的设计层面提高框架的代码质量。这也是我们在专栏中讲解这部分内容的原因。

除了上一节课中讲到的设计思想，实际上，可扩展也是大部分框架应该具备的一个重要特性。所谓的框架可扩展，我们之前也提到过，意思就是，框架使用者在不修改框架源码的情况下，基于扩展点定制扩展新的功能。


前面在理论部分，我们也讲到，常用来实现扩展特性的设计模式有：观察者模式、模板模式、职责链模式、策略模式等。今天，我们再剖析 Spring 框架为了支持可扩展特性用的 2 种设计模式：观察者模式和模板模式。

话不多说，让我们正式开始今天的学习吧！

观察者模式在 Spring 中的应用

在前面我们讲到，Java、Google Guava 都提供了观察者模式的实现框架。Java 提供的框架比较简单，只包含 `java.util.Observable` 和 `java.util.Observer` 两个类。Google Guava 提供的框架功能比较完善和强大：通过 `EventBus` 事件总线来实现观察者模式。实际上，Spring 也提供了观察者模式的实现框架。今天，我们就再来讲一讲它。

Spring 中实现的观察者模式包含三部分：Event 事件（相当于消息）、Listener 监听者（相当于观察者）、Publisher 发送者（相当于被观察者）。我们通过一个例子来看下，Spring 提供的观察者模式是怎么使用的。代码如下所示：


 复制代码

```
1 // Event事件
2 public class DemoEvent extends ApplicationEvent {
3     private String message;
4
5     public DemoEvent(Object source, String message) {
6         super(source);
7     }
8
9     public String getMessage() {
10         return this.message;
11     }
12 }
13
14 // Listener监听者
15 @Component
16 public class DemoListener implements ApplicationListener<DemoEvent> {
17     @Override
18     public void onApplicationEvent(DemoEvent demoEvent) {
19         String message = demoEvent.getMessage();
20         System.out.println(message);
21     }
22 }
23
24 // Publisher发送者
25 @Component
26 public class DemoPublisher {
27     @Autowired
28     private ApplicationContext applicationContext;
29
30     public void publishEvent(DemoEvent demoEvent) {
31         this.applicationContext.publishEvent(demoEvent);
```

```
32     }  
33 }
```

从代码中，我们可以看出，框架使用起来并不复杂，主要包含三部分工作：定义一个继承 `ApplicationEvent` 的事件（`DemoEvent`）；定义一个实现了 `ApplicationListener` 的监听器（`DemoListener`）；定义一个发送者（`DemoPublisher`），发送者调用 `ApplicationContext` 来发送事件消息。

其中，`ApplicationEvent` 和 `ApplicationListener` 的代码实现都非常简单，内部并不包含太多属性和方法。实际上，它们最大的作用是做类型标识之用（继承自 `ApplicationEvent` 的类是事件，实现 `ApplicationListener` 的类是监听器）。

 复制代码

```
1 public abstract class ApplicationEvent extends EventObject {  
2     private static final long serialVersionUID = 7099057708183571937L;  
3     private final long timestamp = System.currentTimeMillis();  
4  
5     public ApplicationEvent(Object source) {  
6         super(source);  
7     }  
8  
9     public final long getTimestamp() {  
10         return this.timestamp;  
11     }  
12 }  
13  
14 public class EventObject implements java.io.Serializable {  
15     private static final long serialVersionUID = 5516075349620653480L;  
16     protected transient Object source;  
17  
18     public EventObject(Object source) {  
19         if (source == null)  
20             throw new IllegalArgumentException("null source");  
21         this.source = source;  
22     }  
23  
24     public Object getSource() {  
25         return source;  
26     }  
27  
28     public String toString() {  
29         return getClass().getName() + "[source=" + source + "];"
```

```


30     }
31 }
32
33 public interface ApplicationListener<E extends ApplicationEvent> extends EventLis
34     void onApplicationEvent(E var1);
35 }

```

在前面讲到观察者模式的时候，我们提到，观察者需要事先注册到被观察者（JDK 的实现方式）或者事件总线（EventBus 的实现方式）中。那在 Spring 的实现中，观察者注册到了哪里呢？又是如何注册的呢？

我想你应该猜到了，我们把观察者注册到了 ApplicationContext 对象中。这里的 ApplicationContext 就相当于 Google EventBus 框架中的“事件总线”。不过，稍微提醒一下，ApplicationContext 这个类并不只是为观察者模式服务的。它底层依赖 BeanFactory（IOC 的主要实现类），提供应用启动、运行时的上下文信息，是访问这些信息的最顶层接口。

实际上，具体到源码来说，ApplicationContext 只是一个接口，具体的代码实现包含在它的实现类 AbstractApplicationContext 中。我把跟观察者模式相关的代码，摘抄到了下面。你只需要关注它是如何发送事件和注册监听者就好，其他细节不需要细究。

 复制代码

```

1 public abstract class AbstractApplicationContext extends ... {
2     private final Set<ApplicationListener<?>> applicationListeners;
3
4     public AbstractApplicationContext() {
5         this.applicationListeners = new LinkedHashSet();
6         //...
7     }
8
9     public void publishEvent(ApplicationEvent event) {
10         this.publishEvent(event, (ResolvableType)null);
11     }
12
13     public void publishEvent(Object event) {
14         this.publishEvent(event, (ResolvableType)null);
15     }
16
17     protected void publishEvent(Object event, ResolvableType eventType) {

```

```

18     //...
19     Object applicationEvent;
20     if (event instanceof ApplicationEvent) {
21         applicationEvent = (ApplicationEvent)event;
22     } else {
23         applicationEvent = new PayloadApplicationEvent(this, event);
24         if (eventType == null) {
25             eventType = ((PayloadApplicationEvent)applicationEvent).getResolvableType
26         }
27     }
28
29     if (this.earlyApplicationEvents != null) {
30         this.earlyApplicationEvents.add(applicationEvent);
31     } else {
32         this.getApplicationEventMulticaster().multicastEvent(
33             (ApplicationEvent)applicationEvent, eventType);
34     }
35
36     if (this.parent != null) {
37         if (this.parent instanceof AbstractApplicationContext) {
38             ((AbstractApplicationContext)this.parent).publishEvent(event, eventType);
39         } else {
40             this.parent.publishEvent(event);
41         }
42     }
43 }
44
45 public void addApplicationListener(ApplicationListener<?> listener) {
46     Assert.notNull(listener, "ApplicationListener must not be null");
47     if (this.applicationEventMulticaster != null) {
48         this.applicationEventMulticaster.addApplicationListener(listener);
49     } else {
50         this.applicationListeners.add(listener);
51     }
52 }
53
54 public Collection<ApplicationListener<?>> getApplicationListeners() {
55     return this.applicationListeners;
56 }
57
58 protected void registerListeners() {
59     Iterator var1 = this.getApplicationListeners().iterator();
60
61     while(var1.hasNext()) {
62         ApplicationListener<?> listener = (ApplicationListener)var1.next();    thi
63     }
64
65     String[] listenerBeanNames = this.getBeanNamesForType(ApplicationListener.class);
66     String[] var7 = listenerBeanNames;


```

```

67     int var3 = listenerBeanNames.length;
68
69     for(int var4 = 0; var4 < var3; ++var4) {
70         String listenerBeanName = var7[var4];
71         this.getApplicationEventMulticaster().addApplicationListenerBean(listenerBe
72     }
73
74     Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
75     this.earlyApplicationEvents = null;
76     if (earlyEventsToProcess != null) {
77         Iterator var9 = earlyEventsToProcess.iterator();
78
79         while(var9.hasNext()) {
80             ApplicationEvent earlyEvent = (ApplicationEvent)var9.next();
81             this.getApplicationEventMulticaster().multicastEvent(earlyEvent);
82         }
83     }
84 }
85 }

```

从上面的代码中，我们发现，真正的消息发送，实际上是通过 `ApplicationEventMulticaster` 这个类来完成的。这个类的源码我只摘抄了最关键的一部分，也就是 `multicastEvent()` 这个消息发送函数。不过，它的代码也并不复杂，我就不多解释了。这里我稍微提示一下，它通过线程池，支持异步非阻塞、同步阻塞这两种类型的观察者模式。

 复制代码

```

1  public void multicastEvent(ApplicationEvent event) {
2      this.multicastEvent(event, this.resolveDefaultEventType(event));
3  }
4
5  public void multicastEvent(final ApplicationEvent event, ResolvableType eventType
6      ResolvableType type = eventType != null ? eventType : this.resolveDefaultEventT
7      Iterator var4 = this.getApplicationListeners(event, type).iterator();
8
9      while(var4.hasNext()) {
10         final ApplicationListener<?> listener = (ApplicationListener)var4.next();
11         Executor executor = this.getTaskExecutor();
12         if (executor != null) {
13             executor.execute(new Runnable() {
14                 public void run() {
15                     SimpleApplicationEventMulticaster.this.invokeListener(listener, event);
16                 }
17             });
18         } else {

```

```
19         this.invokeListener(listener, event);
20     }
21 }
22
23 }
```

借助 Spring 提供的观察者模式的骨架代码，如果我们要在 Spring 下实现某个事件的发送和监听，只需要做很少的工作，定义事件、定义监听器、往 ApplicationContext 中发送事件就可以了，剩下的工作都由 Spring 框架来完成。实际上，这也体现了 Spring 框架的扩展性，也就是在不需要修改任何代码的情况下，扩展新的事件和监听。


模板模式在 Spring 中的应用

刚刚讲的是观察者模式在 Spring 中的应用，现在我们再讲下模板模式。

我们来看下一下经常在面试中被问到的一个问题：请你说下 Spring Bean 的创建过程包含哪些主要的步骤。这其中就涉及模板模式。它也体现了 Spring 的扩展性。利用模板模式，Spring 能让用户定制 Bean 的创建过程。

Spring Bean 的创建过程，可以大致分为两大步：对象的创建和对象的初始化。

对象的创建是通过反射来动态生成对象，而不是 new 方法。不管是哪种方式，说白了，总归还是调用构造函数来生成对象，没有什么特殊的。对象的初始化有两种实现方式。一种是在类中自定义一个初始化函数，并且通过配置文件，显式地告知 Spring，哪个函数是初始化函数。我举了一个例子解释一下。如下所示，在配置文件中，我们通过 init-method 属性来指定初始化函数。

 复制代码

```
1 public class DemoClass {
2     //...
3
4     public void initDemo() {
5         //...初始化...
6     }
7 }
8
```

```
9 // 配置：需要通过init-method显式地指定初始化方法
10 <bean id="demoBean" class="com.xzg.cd.DemoClass" init-method="initDemo"></bean>
```

这种初始化方式有一个缺点，初始化函数并不固定，由用户随意定义，这就需要 Spring 通过反射，在运行时动态地调用这个初始化函数。而反射又会影响代码执行的性能，那有没有替代方案呢？

Spring 提供了另外一个定义初始化函数的方法，那就是让类实现 InitializingBean 接口。这个接口包含一个固定的初始化函数定义（afterPropertiesSet() 函数）。Spring 在初始化 Bean 的时候，可以直接通过 bean.afterPropertiesSet() 的方式，调用 Bean 对象上的这个函数，而不需要使用反射来调用了。我举个例子解释一下，代码如下所示。

 复制代码

```
1 public class DemoClass implements InitializingBean{
2     @Override
3     public void afterPropertiesSet() throws Exception {
4         //...初始化...
5     }
6 }
7
8 // 配置：不需要显式地指定初始化方法
9 <bean id="demoBean" class="com.xzg.cd.DemoClass"></bean>
```


尽管这种实现方式不会用到反射，执行效率提高了，但业务代码（DemoClass）跟框架代码（InitializingBean）耦合在了一起。框架代码侵入到了业务代码中，替换框架的成本就变高了。所以，我并不是太推荐这种写法。

实际上，在 Spring 对 Bean 整个生命周期的管理中，还有一个跟初始化相对应的过程，那就是 Bean 的销毁过程。我们知道，在 Java 中，对象的回收是通过 JVM 来自动完成的。但是，我们可以在将 Bean 正式交给 JVM 垃圾回收前，执行一些销毁操作（比如关闭文件句柄等等）。

销毁过程跟初始化过程非常相似，也有两种实现方式。一种是通过配置 destroy-method 指定类中的销毁函数，另一种是让类实现 DisposableBean 接口。因为 destroy-method、

DisposableBean 跟 init-method、InitializingBean 非常相似，所以，这部分我们就不详细讲解了，你可以自行研究下。

实际上，Spring 针对对象的初始化过程，还做了进一步的细化，将它拆分成了三个小步骤：初始化前置操作、初始化、初始化后置操作。其中，中间的初始化操作就是我们刚刚讲的那部分，初始化的前置和后置操作，定义在接口 BeanPostProcessor 中。BeanPostProcessor 的接口定义如下所示：

 复制代码

```
1 public interface BeanPostProcessor {  
2     Object postProcessBeforeInitialization(Object var1, String var2) throws BeansEx  
3  
4     Object postProcessAfterInitialization(Object var1, String var2) throws BeansExc  
5 }
```

我们再来看下，如何通过 BeanPostProcessor 来定义初始化前置和后置操作？

我们只需要定义一个实现了 BeanPostProcessor 接口的处理器类，并在配置文件中像配置普通 Bean 一样去配置就可以了。Spring 中的 ApplicationContext 会自动检测在配置文件中实现了 BeanPostProcessor 接口的所有 Bean，并把它们注册到 BeanPostProcessor 处理器列表中。在 Spring 容器创建 Bean 的过程中，Spring 会逐一去调用这些处理器。

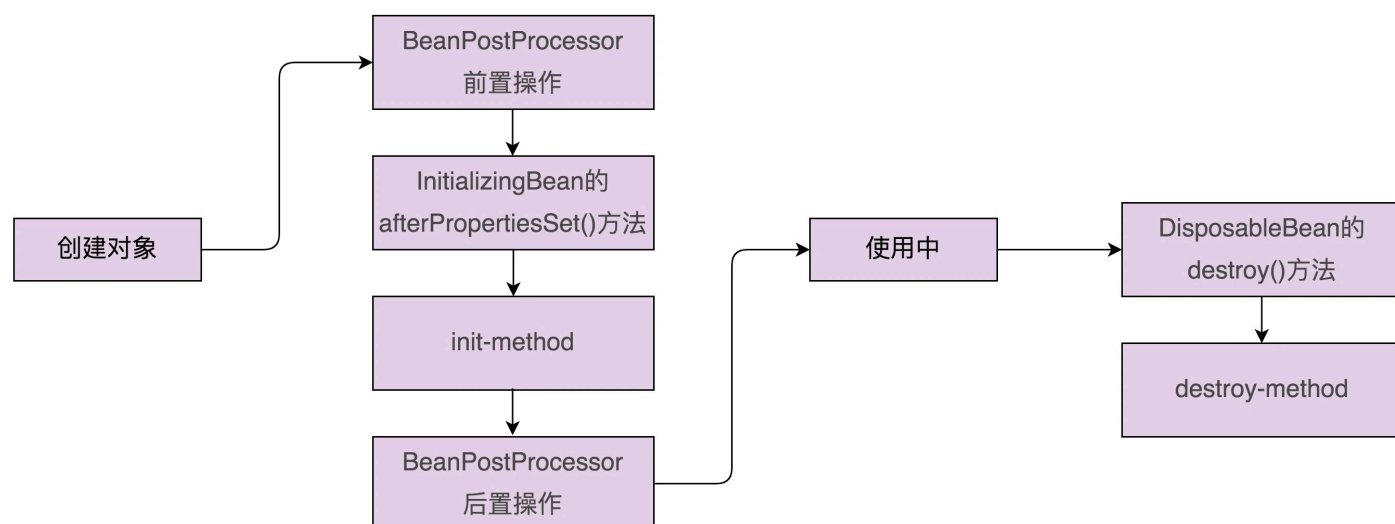
通过上面的分析，我们基本上弄清楚了 Spring Bean 的整个生命周期（创建加销毁）。针对这个过程，我画了一张图，你可以结合着刚刚讲解一块看下。

1.创建

2.初始化

3.使用

4.销毁



不过，你可能会说，这里哪里用到了模板模式啊？模板模式不是需要定义一个包含模板方法的抽象模板类，以及定义子类实现模板方法吗？

实际上，这里的模板模式的实现，并不是标准的抽象类的实现方式，而是有点类似我们前面讲到的 Callback 回调的实现方式，也就是将要执行的函数封装成对象（比如，初始化方法封装成 InitializingBean 对象），传递给模板（BeanFactory）来执行。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天我讲到了 Spring 中用到的两种支持扩展的设计模式，观察者模式和模板模式。

其中，观察者模式在 Java、Google Guava、Spring 中都有提供相应的实现代码。在平时的项目开发中，基于这些实现代码，我们可以轻松地实现一个观察者模式。

Java 提供的框架比较简单，只包含 `java.util.Observable` 和 `java.util.Observer` 两个类。Google Guava 提供的框架功能比较完善和强大，可以通过 EventBus 事件总线来实现观察者模式。Spring 提供了观察者模式包含 Event 事件、Listener 监听者、Publisher 发送者三部

分。事件发送到 ApplicationContext 中，然后，ApplicationConext 将消息发送给事先注册好的监听者。

除此之外，我们还讲到模板模式在 Spring 中的一个典型应用，那就是 Bean 的创建过程。Bean 的创建包含两个大的步骤，对象的创建和对象的初始化。其中，对象的初始化又可以分解为 3 个小的步骤：初始化前置操作、初始化、初始化后置操作。

课堂讨论

在 Google Guava 的 EventBus 实现中，被观察者发送消息到事件总线，事件总线根据消息的类型，将消息发送给可匹配的观察者。那在 Spring 提供的观察者模式的实现中，是否也支持按照消息类型匹配观察者呢？如果能，它是如何实现的？如果不能，你有什么方法可以让它支持吗？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

Spring框架中的观察者模式和模板模式是支持扩展的设计模式。观察者模式包含Event事件、Listener监听者、Publisher发送者三部分，通过ApplicationContext实现事件的发送和监听。模板模式在Spring中用于定制Bean的创建过程，包括对象的创建和初始化。Spring还提供了Bean的销毁过程管理。在Bean的整个生命周期中，Spring通过BeanPostProcessor接口定义了初始化前置和后置操作。这些设计模式的应用体现了Spring框架的扩展性和灵活性。文章还讨论了观察者模式在Google Guava的EventBus实现中的消息类型匹配，以及对Spring提供的观察者模式的消息类型匹配的探讨。整体而言，本文深入探讨了Spring框架中观察者模式和模板模式的应用，以及相关技术细节，对于想要深入了解Spring框架设计模式的读者具有一定的参考价值。

© 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。

全部留言 (14)

最新 精选



zz

2020-06-05

看到源码中有这么多的if else，瞬间给了自己一些信心。

共 7 条评论 >

👍 36



Heaven

2020-05-18

看了下源码,其流程可以从

图片: <https://uploader.shimo.im/f/fZuIVWFIIWQnnRFq.png>

推送Event时候,去发送Event开始走

主要就是这个

在此方法中,会调用getApplicationListeners(event,eventType)函数

图片: <https://uploader.shimo.im/f/3mZZvSBhmc8CXLnX.png>

在这个方法中,会获取到对应的所有监听者,如何获取到的,会先通过一个锁来从一个名为retrieverCache的map中尝试获取到对应的监听者

如果拿不到,会进入到retrieveApplicationListeners()这个函数之中

图片: <https://uploader.shimo.im/f/GFvS2QEKGIMctZrc.png>

在这个方法中,会在add返回的结果的时候,会调用一个方法supportsEvent(),这才是真正进行匹配的方法

图片: <https://uploader.shimo.im/f/102la9Toqlw5ZOyq.png>

匹配事件和源类型是否一致,一致才算做可以发送

共 1 条评论 >

👍 10



悟光

2020-05-18

支持按照消息类型匹配观察者, 最终调用 SimpleApplicationEventMulticaster 类的multicastEvent方法通过反射匹配类型。根据配置采用异步还是同步的监听方式。

```
public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType) {
```

```
    ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
```

```
    Executor executor = getTaskExecutor();
```

```
    for (ApplicationListener<?> listener : getApplicationListeners(event, type)) {
```

```
        if (executor != null) {
```

```
            executor.execute(() -> invokeListener(listener, event));
```

```
        }
```

```
    else {
```

```
        invokeListener(listener, event);
```

```
    }
```

```
}
```

```
}
```



👍 10



test

2020-05-18

用反射获取的type



8



Monday

2020-05-20

定义了一个bean同时实现了InitializingBean, BeanPostProcessor, DisposableBean, 发现方法跟老师最后一张图的不一致:

- 1、顺序是构造器、afterPropertiesSet、postProcessBeforeInitialization、postProcessAfterInitialization、destroy
- 2、postProcessBeforeInitialization、postProcessAfterInitialization这两个方法交替执行了N次

共 1 条评论 >



6



松小鼠

2020-05-18

昨天刚好在隔壁小马哥那里看到了，两个课一起听，侧重点不同，都很重要啊

共 3 条评论 >



5



Tobias

2020-07-27

Spring 提供的观察者模式是支持按照消息类型匹配观察者。getApplicationListeners(event, type) 方法会根据eventtype 找到对应的listeners. getApplicationListeners(event, type) 通过反射找到 event 以及event的子类 对应的listeners.



3



tonyli

2022-09-19 来自中国香港

使用模板模式定义了一系列步骤的骨架，是各类框架的根本设计模式。



2



握了个大蚂蚱

2020-10-12

- 1.实现InitializingBean的初始化方法，也是约定优于配置的一个体现，只不过不是覆盖默认值而是实现init-method的一个前置方法afterPropertiesSet。
- 2.实现InitializingBean的初始化方法和自己指定init-method相比，侵入性更高，所以不太推

荐。可以用注解版的@Bean(initMethod = "xx")来指定初始化方法，或者使用JSR250中的@PostConstruct标注在初始化方法上来让程序回调。



1



剑八

2020-07-05

spring中的refresh是一个模板方法：

大致有：注册beanFactoryPostProcessor，beanPostProcessor，读取bean definition，创建并初始化bean,等

共 1 条评论 >



1