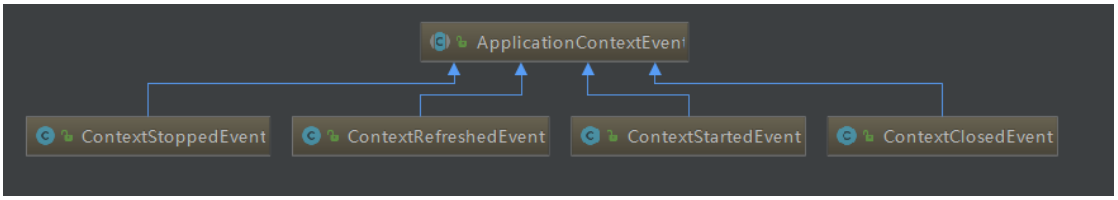
1. 使用Spring 事件

Spring事件体系包括三个组件：事件，事件监听器，事件广播器。

事件

Spring内置事件

内置事件中由系统内部进行发布，只需注入监听器



Event	说明
ContextRefreshedEvent	当容器被实例化或refreshed时发布. 如调用refresh()方法，此处的实例化是指所有的bean都已被加载,后置处理器都被激活,所有单例bean都已被实例化，所有的容器对象都已准备好可使用. 如果容器支持热重载,则refresh可以被触发多次(XmlWebApplicatonContext支持热刷新,而GenericApplicationContext则不支持)
ContextStartedEvent	当容器启动时发布,即调用start()方法，已启用意味着所有的Lifecycle bean都已显式接收到了start信号
ContextStoppedEvent	当容器停止时发布,即调用stop()方法，即所有的Lifecycle bean都已显式接收到了stop信号 ，关闭的容器可以通过start()方法重启
ContextClosedEvent	当容器关闭时发布,即调用close方法，关闭意味着所有的单例bean都已被销毁.关闭的容器不能被重启或refresh
RequestHandledEvent	这只在使用spring的DispatcherServlet时有效,当一个请求被处理完成时发布

自定义事件

事件类需要继承ApplicationEvent，代码如下：

```
1
2  /***
3   * @Author 徐庶 QQ:1092002729
4   * @Slogan 致敬大师，致敬未来的你
5   * 事件
6   */
7  public class BigEvent extends ApplicationEvent {
8
9   private String name;
10
11   public BigEvent(Object source, String name) {
12   super(source);
13   this.name = name;
14   }
15
```

```
 public String getName() {
 return name;
 }
}
```

这里为了简单测试，所以写的很简单。

事件类是一种很简单的pojo，除了需要继承ApplicationEvent也没什么了，这个类有一个构造方法需要super。

事件监听器

事件监听器-基于接口

```
@Component
public class HelloEventListener implements ApplicationListener<OrderEvent> {

    @Override
    public void onApplicationEvent(OrderEvent event) {
 if(event.getName().equals("减库存")){
 System.out.println("减库存.......");
 }
    }
}
```

事件监听器需要实现ApplicationListener接口，这是个泛型接口，泛型类类型就是事件类型，其次需要是spring容器托管的bean，所以这里加了@component，只有一个方法，就是onApplicationEvent。

事件监听器-基于注解

```
@Component
public class OrderEventListener {
 @EventListener(OrderEvent.class)
 public void onApplicationEvent(OrderEvent event) {
 if(event.getName().equals("减库存")){
 System.out.println("减库存.......");
 }
 }

}
```

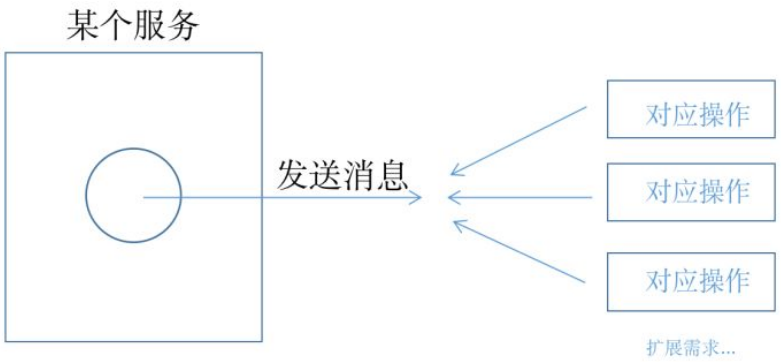
事件发布操作

事件发布方式很简单

```
applicationContext.publishEvent(new HelloEvent(this,"lgb"));
```

然后调用方法就能看到

```
2020-9-22 19:08:00.052  INFO 284928 --- [nio-5577-exec-3] l.b.e.c.s.event.HelloEventListener       : r
eceive lgb say hello!
```

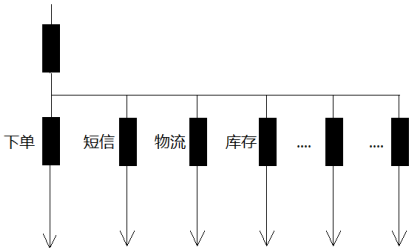


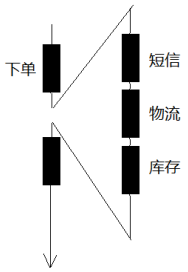
疑问

1. 同样的事件能有多个监听器 -- 可以的
2. 事件监听器一定要写一个类去实现吗 -- 其实是可以不需要的，spring有个注解@EventListener，修饰在方法上，稍后给出使用方法
3. 事件监听操作和发布事件的操作是同步的吗？ -- 是的，所以如果有事务，监听操作也在事务内
4. 可以作为异步处理吗？ --可以 看源码有解释。：

```
@Bean(name = "applicationEventMulticaster")
public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
 SimpleApplicationEventMulticaster eventMulticaster
 = new SimpleApplicationEventMulticaster();

 eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
 return eventMulticaster;
}
```



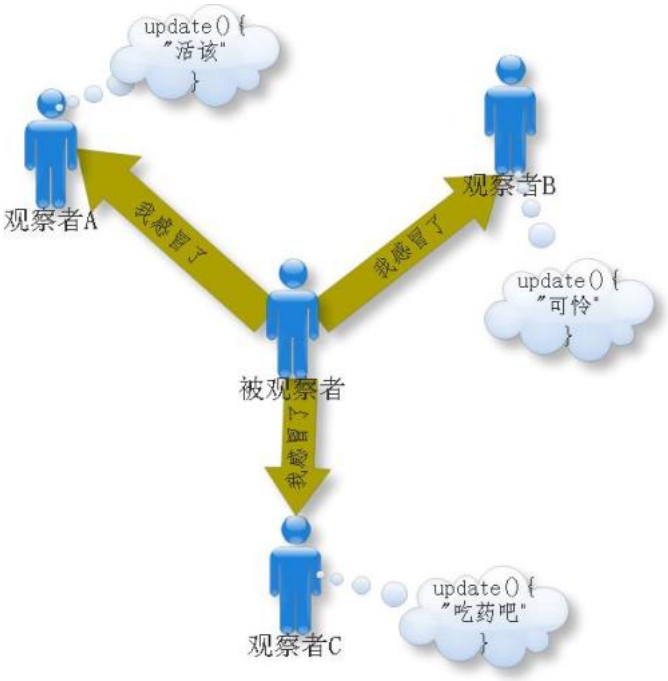


2. Spring事件原理

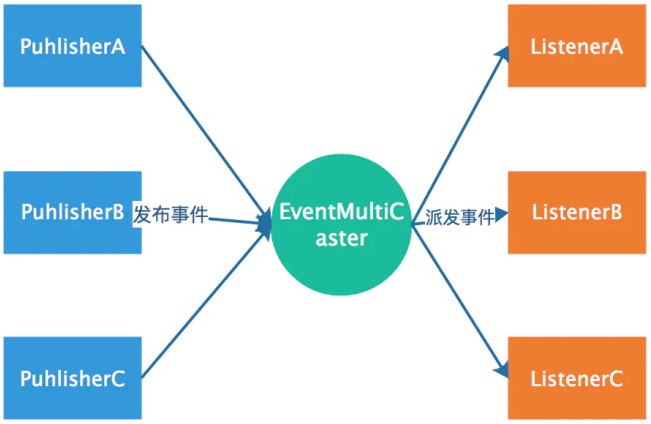
原理：观察者模式

spring的事件监听有三个部分组成：

- **事件**（ApplicationEvent）负责对应相应监听器 事件源发生某事件是特定事件监听器被触发的原因。
- **监听器(**ApplicationListener) 对应于观察者模式中的**观察者**。监听器监听特定事件,并在内部定义了事件发生后的响应逻辑。
- **事件发布器**（ApplicationEventMulticaster ）对应于观察者模式中的**被观察者/主题， 负责通知观察者** 对外提供发布事件和增删事件监听器的接口,维护事件和事件监听器之间的映射关系,并在事件发生时负责通知相关监听器。



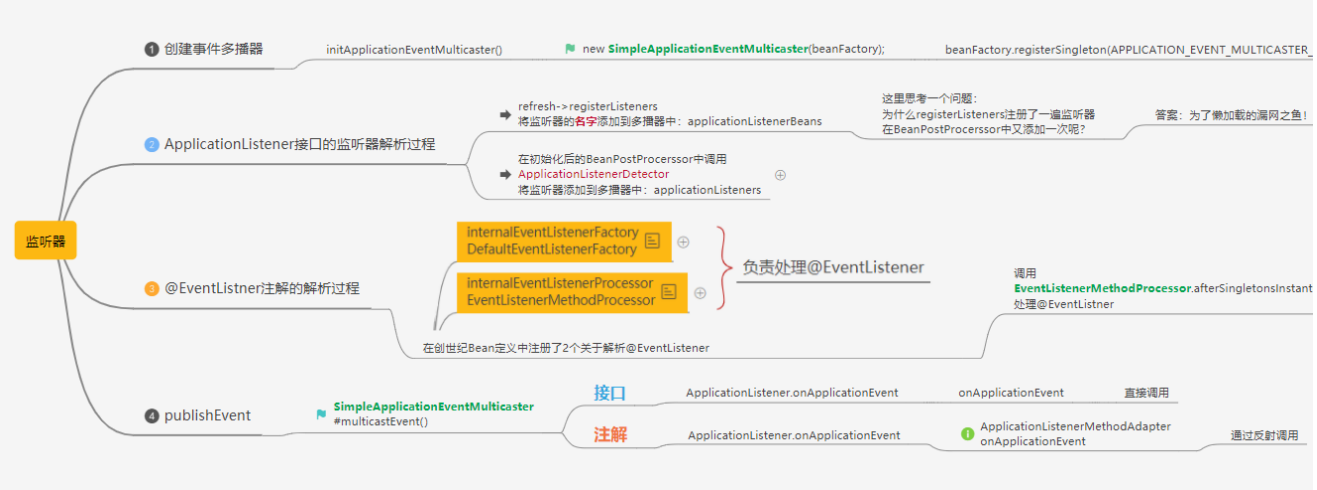
Spring事件机制是观察者模式的一种实现，但是除了发布者和监听者者两个角色之外，还有一个EventMultiCaster的角色负责把事件转发给监听者，工作流程如下：



Spring事件机制

也就是说上面代码中发布者调用`applicationEventPublisher.publishEvent(msg);`是会将事件发送给了EventMultiCaster， 而后由EventMultiCaster注册着所有的Listener，然后根据事件类型决定转发给那个Listener。

源码流程：



Spring在ApplicationContext接口的抽象实现类AbstractApplicationContext中完成了事件体系的搭建。
AbstractApplicationContext拥有一个applicationEventMulticaster成员变量，
applicationEventMulticaster提供了容器监听器的注册表。
AbstractApplicationContext在refresh()这个容器启动方法中搭建了事件的基础设施,其中
AbstractApplicationContext的refresh方法实现如下:

```
@Override
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()) {
logger.warn("Exception encountered during context initialization - " +
"cancelling refresh attempt: " + ex);
}

// Destroy already created singletons to avoid dangling resources.
destroyBeans();

// Reset 'active' flag.
cancelRefresh(ex);

// Propagate exception to caller.
throw ex;
}

finally {
// Reset common introspection caches in Spring's core, since we
// might not ever need metadata for singleton beans anymore...
resetCommonCaches();
}
}
```

```
}
```

1 事件广播器的初始化

```
/**
 * Initialize the ApplicationEventMulticaster.
 * Uses SimpleApplicationEventMulticaster if none defined in the context.
 * @see org.springframework.context.event.SimpleApplicationEventMulticaster
 */
protected void initApplicationEventMulticaster() {
 ConfigurableListableBeanFactory beanFactory = getBeanFactory();
 if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
 this.applicationEventMulticaster =
  beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, ApplicationEventMulticaster.class);
  if (logger.isDebugEnabled()) {
  logger.debug("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster + "]");
  }
  }
  else {
  this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
  beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticas
ter);
  if (logger.isDebugEnabled()) {
  logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
  APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
  "': using default [" + this.applicationEventMulticaster + "]");
  }
  }
}
```

用户可以在配置文件中为容器定义一个自定义的事件广播器，只要实现ApplicationEventMulticaster就可以了，Spring会通过 反射的机制将其注册成容器的事件广播器，如果没有找到配置的外部事件广播器，Spring自动使用 SimpleApplicationEventMulticaster作为事件广播器。

2 注册事件监听器

```
/**
 * Add beans that implement ApplicationListener as listeners.
 * Doesn't affect other listeners, which can be added without being beans.
 */
protected void registerListeners() {
 // 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...
 Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
```

```
20  this.earlyApplicationEvents = null;
21  if (earlyEventsToProcess != null) {
22  for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
23  getApplicationEventMulticaster().multicastEvent(earlyEvent);
24  }
25  }
26 }
```

Spring根据反射机制，使用ListableBeanFactory的getBeansOfType方法，从BeanDefinitionRegistry中找出所有实现 org.springframework.context.ApplicationListener的Bean，将它们注册为容器的事件监听器，实际的操作就是将其添加到事件广播器所提供的监听器注册表中。

3 发布事件

跟着 finishRefresh();方法进入publishEvent(new ContextRefreshedEvent(this));方法如下:

```
1  /**
2   * Publish the given event to all listeners.
3   * @param event the event to publish (may be an {@link ApplicationEvent}
4   * or a payload object to be turned into a {@link PayloadApplicationEvent})
5   * @param eventType the resolved event type, if known
6   * @since 4.2
7   */
8  protected void publishEvent(Object event, ResolvableType eventType) {
9   Assert.notNull(event, "Event must not be null");
10  if (logger.isTraceEnabled()) {
11  logger.trace("Publishing event in " + getDisplayName() + ": " + event);
12  }
13
14  // Decorate event as an ApplicationEvent if necessary
15  ApplicationEvent applicationEvent;
16  if (event instanceof ApplicationEvent) {
17  applicationEvent = (ApplicationEvent) event;
18  }
19  else {
20  applicationEvent = new PayloadApplicationEvent<Object>(this, event);
21  if (eventType == null) {
22  eventType = ((PayloadApplicationEvent)applicationEvent).getResolvableType();
23  }
24  }
25
26  // Multicast right now if possible - or lazily once the multicaster is initialized
27  if (this.earlyApplicationEvents != null) {
28  this.earlyApplicationEvents.add(applicationEvent);
29  }
30  else {
31  getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
32  }
33
34  // Publish event via parent context as well...
35  if (this.parent != null) {
36  if (this.parent instanceof AbstractApplicationContext) {
37  ((AbstractApplicationContext) this.parent).publishEvent(event, eventType);
38  }
39  else {
```

```
 this.parent.publishEvent(event);
 }
 }
}
```

在AbstractApplicationContext的publishEvent方法中， Spring委托ApplicationEventMulticaster将事件通知给所有的事件监听器.

4 Spring默认的事件广播器SimpleApplicationEventMulticaster

```
@Override
public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {
 ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
 for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {
 Executor executor = getTaskExecutor();
 if (executor != null) {
 executor.execute(new Runnable() {
 @Override
 public void run() {
  invokeListener(listener, event);
 }
 });
 }
 else {
 invokeListener(listener, event);
 }
 }
}

/**
 * Invoke the given listener with the given event.
 * @param listener the ApplicationListener to invoke
 * @param event the current event to propagate
 * @since 4.1
 */
@SuppressWarnings({"unchecked", "rawtypes"})
protected void invokeListener(ApplicationListener listener, ApplicationEvent event) {
 ErrorHandler errorHandler = getErrorHandler();
 if (errorHandler != null) {
 try {
 listener.onApplicationEvent(event);
 }
 catch (Throwable err) {
 errorHandler.handleError(err);
 }
 }
 else {
 try {
 listener.onApplicationEvent(event);
 }
 catch (ClassCastException ex) {
 // Possibly a lambda-defined listener which we could not resolve the generic event type for
 LogFactory.getLog(getClass()).debug("Non-matching event type for listener: " + listener, ex);
 }
```

```
 }
}

```

遍历注册的每个监听器，并启动来调用每个监听器的onApplicationEvent方法。由于SimpleApplicationEventMulticaster的taskExecutor的实现类是SyncTaskExecutor，因此，事件监听器对事件的处理，是同步进行的。

从代码可以看出，applicationContext.publishEvent()方法，需要同步等待各个监听器处理完之后，才返回。

也就是说，Spring提供的事件机制，默认是同步的。如果想用异步的，可以自己实现ApplicationEventMulticaster接口，并在Spring容器中注册id为applicationEventMulticaster的Bean。例如下面所示:

```
public class AsyncApplicationEventMulticaster extends AbstractApplicationEventMulticaster {
 private TaskExecutor taskExecutor = new SimpleAsyncTaskExecutor();

 public void setTaskExecutor(TaskExecutor taskExecutor) {
 this.taskExecutor = (taskExecutor != null ? taskExecutor : new SimpleAsyncTaskExecutor());
 }

 protected TaskExecutor getTaskExecutor() {
 return this.taskExecutor;
 }

 @SuppressWarnings("unchecked")
 public void multicastEvent(final ApplicationEvent event) {
 for (Iterator<ApplicationListener> it = getApplicationListeners().iterator(); it.hasNext();) {
 final ApplicationListener listener = it.next();
 getTaskExecutor().execute(new Runnable() {
 public void run() {
 listener.onApplicationEvent(event);
 }
 });
 }
 }
}
```

spring配置：

```
 @Bean(name = "applicationEventMulticaster")
public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
 SimpleApplicationEventMulticaster eventMulticaster
 = new SimpleApplicationEventMulticaster();

 //ThreadPoolTaskExecutor
 eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
 return eventMulticaster;
}
```

Spring发布事件之后，所有注册的事件监听器，都会收到该事件，因此，事件监听器在处理事件时，需要先判断该事件是否是自己关心的。

Sping事件体系所使用的设计模式是：观察者模式。ApplicationListener是观察者接口，接口中定义了onApplicationEvent方法，该方法的作用是对ApplicationEvent事件进行处理。

问题：

Spring是怎样避免读取到不完整Bean的?

怎么样可以在所有Bean创建完后做扩展代码?

请介绍下Spring事件监听器的原理。

文档：Spring事件监听机制
链接：http://note.youdao.com/noteshare?id=42d15ea5ab2072b4a354441ce9080eb9&sub=wcp1597739566991276