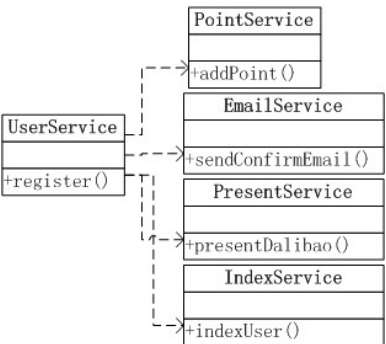


Spring事件驱动模型

事件驱动模型简介

- 事件驱动模型也就是我们常说的观察者，或者发布-订阅模型；理解它的几个关键点：
- 1. 首先是一种对象间的一对多的关系；最简单的如交通信号灯，信号灯是目标（一方），行人注视着信号灯（多方）；
 - 2. 当目标发送改变（发布），观察者（订阅者）就可以接收到改变；
 - 3. 观察者如何处理（如行人如何走，是快走/慢走/不走，目标不会管的），目标无需干涉；所以就松散耦合了它们之间的关系。

接下来先看一个用户注册的例子：



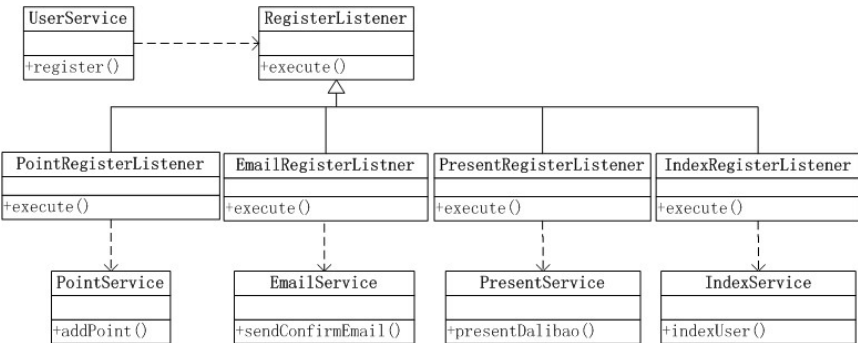
用户注册成功后，需要做这么多事：

- 1、加积分
 - 2、发确认邮件
 - 3、如果是游戏帐户，可能赠送游戏大礼包
 - 4、索引用户数据
-

问题：

- 1. UserService和其他Service耦合严重，增删功能比较麻烦；
- 2. 有些功能可能需要调用第三方系统，如增加积分/索引用户，速度可能比较慢，此时需要异步支持；这个如果使用Spring，可以轻松解决，后边再介绍；

从如上例子可以看出，应该使用一个观察者来解耦这些Service之间的依赖关系，如图：



增加了一个Listener来解耦UserService和其他服务，即注册成功后，只需要通知相关的监听器，不需要关系它们如何处理。增删功能非常容易。

这就是一个典型的事件处理模型/观察者，解耦目标对象和它的依赖对象，目标只需要通知它的依赖对象，具体怎么处理，依赖对象自己决定。比如是异步还是同步，延迟还是非延迟等。

上边其实也使用了DIP（依赖倒置原则），依赖于抽象，而不是具体。

还是就是使用了IoC思想，即以前主动去创建它依赖的Service，现在只是被动等待别人注册进来。

其他的例子还有如GUI中的按钮和动作的关系，按钮和动作本身都是一种抽象，每个不同的按钮的动作可能不一样；如“文件->新建”打开新建窗口；点击“关闭”按钮关闭窗口等等。

主要目的是：松散耦合对象间的一对多的依赖关系，如按钮和动作的关系；

2013年7月

日	一	二	三	四	五	六
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

公告



导航

- BlogJava
- 首页
- 发新随笔
- 发新文章
- 联系
- 聚合XML
- 管理

常用链接

- 我的随笔
- 我的评论
- 我的参与
- 最新评论

留言簿(1)

- 给我留言
- 查看公开留言
- 查看私人留言

随笔档案(152)

- 2013年7月 (3)
- 2013年4月 (2)
- 2012年7月 (4)
- 2012年6月 (8)
- 2012年5月 (1)
- 2012年4月 (116)
- 2012年3月 (5)
- 2012年2月 (8)
- 2012年1月 (5)

我的收藏

- a
- b (rss)
- www.javady.com
- www.javady.com

搜索

搜索

最新评论XML

- 1. re: Spring MVC 3 深入总结
- 评论内容较长,点击标题查看
- --zuidaima
- 2. re: [原]java传统集合的一些弊病以及解决办法[未登录]
- 你不是用的迭代么，用迭代器来移除元素不就好了。这也能称之为弊端？我觉得这是你代码写的有问题！！
- --胡言乱语
- 3. re: [原]web项目测试方法总结
- 犯得上发个爱的方式公司的
- --阿三地方

如何实现呢？**面向接口编程（即面向抽象编程）**，而非**面向实现**。即按钮和动作可以定义为接口，这样它俩的依赖是最小的（如在Java中，没有比接口更抽象的了）。

有朋友会问，我刚开始学的时候也是这样：抽象类不也行吗？记住一个原则：**接口目的是抽象，抽象类目的是复用**；所以如果接触过servlet/servlets2/spring等框架，大家都应该知道：

- Servlet<-----GenericServlet<-----HttpServlet<-----我们自己的
- Action<-----ActionSupport<-----我们自己的
- DaoInterface<-----xxDaoSupport<-----我们自己的

从上边大家应该能体会出接口、抽象类的主要目的了。现在想想其实很简单。

在Java中接口还有一个非常重要的好处：**接口是可以多实现的，类/抽象类只能单继承，所以使用接口可以非常容易扩展新功能（还可以实现所谓的mixin），类/抽象类办不到。**

Java GUI事件驱动模型/观察者

扯远了，再来看看Java GUI世界里的**事件驱动模型**吧：

如果写过AWT/Swing程序，应该知道其所有组件都继承自java.awt.Component抽象类，其内部提供了addXXXListener(XXXListener l)注册监听器的方法，即Component与实际动作之间依赖于XXXListener抽象。

比如获取焦点事件，很多组件都可以有这个事件，是我们知道组件获取到焦点后需要一个处理，虽然每个组件如何处理是特定的（具体的），但我们可以抽象一个FocusListener，让所有具体实现它然后提供具体动作，这样组件只需依赖于FocusListener抽象，而不是具体。

还有如java.awt.Button，提供了一个addActionListener(ActionListener l)，用于注册点击后触发的ActionListener实现。

组件是一个抽象类，其好处主要是复用，比如复用这些监听器的触发及管理。

JavaBean规范的事件驱动模型/观察者

JavaBean规范提供了JavaBean的PropertyEditorSupport及PropertyChangeListener支持。

PropertyEditorSupport就是目标，而PropertyChangeListener就是监听器，大家可以google搜索下，具体网上有很多例子。

Java提供的事件驱动模型/观察者抽象

JDK内部直接提供了观察者模式的抽象：

目标：java.util.Observable，提供了目标需要的关键抽象：addObserver/deleteObserver/notifyObservers()等，具体请参考javadoc。

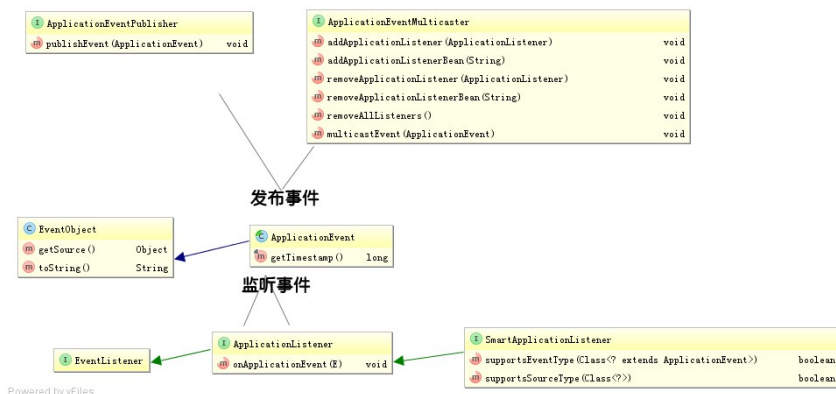
观察者：java.util.Observer，提供了观察者需要的主要抽象：update(Observable o, Object arg)，此处还提供了一种推模型（目标主动把数据通过arg推到观察者）/拉模型（目标需要根据o自己去拉数据，arg为null）。

因为网上介绍的非常多了，请google搜索了解如何使用这个抽象及推/拉模型的优缺点。

接下来是我们的重点：spring提供的事件驱动模型。

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

具体代表者是：ApplicationEvent：

- 其继承自JDK的EventObject，JDK要求所有事件将继承它，并通过source得到事件源，比如我们的AWT事件体系也是继承自它；
- 系统默认提供了如下ApplicationEvent事件实现：

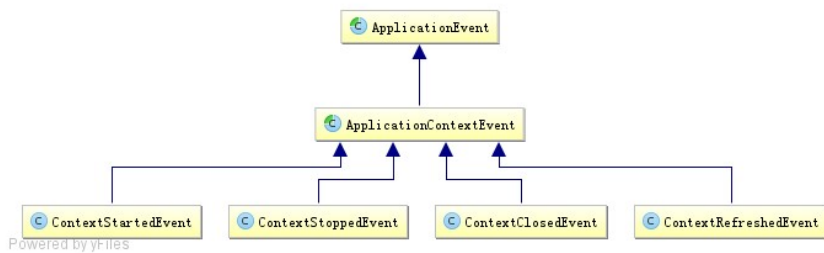
- 4. re: Spring事件驱动模型
- 评论内容较长,点击标题查看

--源代码

- 5. re: android截取屏幕图片
- 啊哈，楼主，还有权限哈~我随便说的，没有什么别的意思~

--无巾帽须眉

Powered by: 博客园
模板提供：沪江博客
Copyright ©2015 陈雨晨



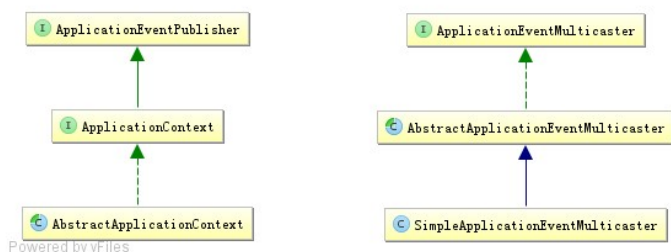
只有一个ApplicationContextHolderEvent，表示ApplicationContextHolder容器事件，且其又有如下实现：

- ContextStartedEvent：ApplicationContextHolder启动后触发的事件；（目前版本没有任何作用）
- ContextStoppedEvent：ApplicationContextHolder停止后触发的事件；（目前版本没有任何作用）
- ContextRefreshedEvent：ApplicationContextHolder初始化或刷新完成后触发的事件；（容器初始化完成后调用）
- ContextClosedEvent：ApplicationContextHolder关闭后触发的事件；（如web容器关闭时会自动会触发spring容器的关闭，如果是普通java应用，需要调用ctx.registerShutdownHook();注册虚拟机关闭时的钩子才行）

注：org.springframework.context.support.AbstractApplicationContext抽象类实现了Lifecycle的start和stop回调并发布ContextStartedEvent和ContextStoppedEvent事件；但是无任何实现调用它，所以目前无任何作用。

目标（发布事件者）

具体代表者是：ApplicationEventPublisher及ApplicationEventMulticaster，系统默认提供了如下实现：



1、ApplicationContext接口继承了ApplicationEventPublisher，并在AbstractApplicationContext实现了具体代码，实际执行是委托给ApplicationEventMulticaster（可以认为是多播）：

Java代码 ☆

```

1. public void publishEvent(ApplicationEvent event) {
2.     //省略部分代码
3. }
4. getApplicationEventMulticaster().multicastEvent(event);
5. if (this.parent != null) {
6.     this.parent.publishEvent(event);
7. }
8. }
  
```

我们常用的ApplicationContext都继承自AbstractApplicationContext，如ClassPathXmlApplicationContext、XmlWebApplicationContext等。所以自动拥有这个功能。

2、ApplicationContext自动到本地容器里找一个名字为“”的ApplicationEventMulticaster实现，如果没有自己new一个SimpleApplicationEventMulticaster。其中SimpleApplicationEventMulticaster发布事件的代码如下：

Java代码 ☆

```

1. public void multicastEvent(final ApplicationEvent event) {
2.     for (final ApplicationListener listener : getApplicationListeners(event)) {
3.         Executor executor = getTaskExecutor();
4.         if (executor != null) {
5.             executor.execute(new Runnable() {
6.                 public void run() {
7.                     listener.onApplicationEvent(event);
8.                 }
9.             });
10.        }
11.        else {
12.            listener.onApplicationEvent(event);
13.        }
14.    }
15. }
  
```

大家可以看到如果给它一个executor（java.util.concurrent.Executor），它就可以异步支持发布事件了。佛则就是通过发送。

所以我们发送事件只需要通过ApplicationContext.publishEvent即可，没必要再创建自己的实现了。除非有必要。

监听器

具体代表者是：ApplicationListener

- 其继承自JDK的EventListener，JDK要求所有监听器将继承它，比如我们的AWT事件体系也是继承自它；
- ApplicationListener接口：

Java代码 ☆

```
1. public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {
2.     void onApplicationEvent(E event);
3. }
```

其只提供了onApplicationEvent方法，我们需要在该方法实现内部判断事件类型来处理，也没有提供按顺序触发监听器的语义，所以Spring提供了另一个接口，SmartApplicationListener：

Java代码 


```
1. public interface SmartApplicationListener extends ApplicationListener<ApplicationEvent>, Ordered {
2.     //如果实现支持该事件类型 那么返回true
3.     boolean supportsEventType(Class<? extends ApplicationEvent> eventType);
4.
5.     //如果实现支持“目标”类型，那么返回true
6.     boolean supportsSourceType(Class<?> sourceType);
7.
8.     //顺序，即监听器执行的顺序，值越小优先级越高
9.     int getOrder();
10. }
```

该接口可方便实现去判断支持的事件类型、目标类型，及执行顺序。

Spring事件机制的简单例子

本例子模拟一个给多个人发送内容（类似于报纸新闻）的例子。

1、定义事件


Java代码 

```
1. package com.sishuok.hello;
2. import org.springframework.context.ApplicationEvent;
3. public class ContentEvent extends ApplicationEvent {
4.     public ContentEvent(final String content) {
5.         super(content);
6.     }
7. }
```

非常简单，如果用户发送内容，只需要通过构造器传入内容，然后通过getSource即可获取。

2、定义无序监听器


之所以说无序，类似于AOP机制，顺序是无法确定的。

Java代码 

```
1. package com.sishuok.hello;
2. import org.springframework.context.ApplicationEvent;
3. import org.springframework.context.ApplicationListener;
4. import org.springframework.stereotype.Component;
5. @Component
6. public class Lisilistener implements ApplicationListener<ApplicationEvent> {
7.     @Override
8.     public void onApplicationEvent(final ApplicationEvent event) {
9.         if(event instanceof ContentEvent) {
10.             System.out.println("李四收到了新的内容: " + event.getSource());
11.         }
12.     }
13. }
```

- 1、使用@Component注册Bean即可；
- 2、在实现中需要判断event类型是ContentEvent才可以处理；

更简单的办法是通过泛型指定类型，如下所示

Java代码 

```
1. package com.sishuok.hello;
2. import org.springframework.context.ApplicationListener;
3. import org.springframework.stereotype.Component;
4. @Component
5. public class ZhangsanListener implements ApplicationListener<ContentEvent> {
6.     @Override
7.     public void onApplicationEvent(final ContentEvent event) {
8.         System.out.println("张三收到了新的内容: " + event.getSource());
9.     }
10. }
```

3、定义有序监听器

实现SmartApplicationListener接口即可。

Java代码 

```
1. package com.sishuok.hello;
2. import org.springframework.context.ApplicationEvent;
3. import org.springframework.context.event.SmartApplicationListener;
4. import org.springframework.stereotype.Component;
```

```
5.
6. @Component
7. public class WangwuListener implements SmartApplicationListener {
8.
9.     @Override
10.    public boolean supportsEventType(final Class<? extends ApplicationEvent> eventType) {
11.
12.        return eventType == ContentEvent.class;
13.    }
14.    @Override
15.    public boolean supportsSourceType(final Class<?> sourceType) {
16.        return sourceType == String.class;
17.    }
18.    @Override
19.    public void onApplicationEvent(final ApplicationEvent event) {
20.        System.out.println("王五在孙六之前收到新的内容: " + event.getSource());
21.    }
22.    @Override
23.    public int getOrder() {
24.        return 1;
25.    }
26. }
```

Java代码 ☆

```
1. package com.sishuok.hello;
2. import org.springframework.context.ApplicationEvent;
3. import org.springframework.context.event.SmartApplicationListener;
4. import org.springframework.stereotype.Component;
5.
6. @Component
7. public class SunliuListener implements SmartApplicationListener {
8.
9.     @Override
10.    public boolean supportsEventType(final Class<? extends ApplicationEvent> eventType) {
11.
12.        return eventType == ContentEvent.class;
13.    }
14.    @Override
15.    public boolean supportsSourceType(final Class<?> sourceType) {
16.        return sourceType == String.class;
17.    }
18.
19.    @Override
20.    public void onApplicationEvent(final ApplicationEvent event) {
21.        System.out.println("孙六在王五之后收到新的内容: " + event.getSource());
22.    }
23.
24.    @Override
25.    public int getOrder() {
26.        return 2;
27.    }
28. }
```

1. supportsEventType : 用于指定支持的事件类型，只有支持的才调用onApplicationEvent；
2. supportsSourceType : 支持的目标类型，只有支持的才调用onApplicationEvent；
3. getOrder : 即顺序，越小优先级越高

4、测试

4.1、配置文件

Java代码 ☆

```
1. <context:component-scan base-package="com.sishuok"/>
```

就一句话，自动扫描注解Bean。

4.2、测试类

Java代码 ☆

```
1. package com.sishuok;
2. import com.sishuok.hello.ContentEvent;
3. import org.junit.Test;
4. import org.junit.runner.RunWith;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.context.ApplicationContext;
7. import org.springframework.test.context.ContextConfiguration;
8. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
9.
10. @RunWith(SpringJUnit4ClassRunner.class)
11. @ContextConfiguration(locations={"classpath:spring-config-hello.xml"})
12. public class HelloIT {
13.
14.     @Autowired
15.     private ApplicationContext applicationContext;
16.     @Test
17.     public void testPublishEvent() {
```

```
18.         applicationContext.publishEvent(new ContentEvent("今年是龙年的博客更新了"));
19.     }
20.
21. }
```

接着会输出：

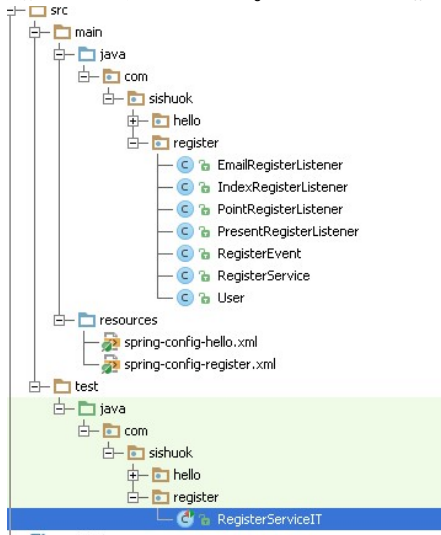
Java代码 ☆

1. 王五在孙六之前收到新的内容：今年是龙年的博客更新了
2. 孙六在王五之后收到新的内容：今年是龙年的博客更新了
3. 李四收到了新的内容：今年是龙年的博客更新了
4. 张三收到了新的内容：今年是龙年的博客更新了

一个简单的测试例子就演示完毕，而且我们使用spring的事件机制去写相关代码会非常简单。

Spring事件机制实现之前提到的注册流程

具体请下载源代码参考com.sishuok.register包里的代码。此处贴一下源码结构：



这里讲解一下Spring对异步事件机制的支持，实现方式有两种：

1、全局异步

即只要是触发事件都是以异步执行，具体配置（spring-config-register.xml）如下：

Java代码 ☆

```
1. <task:executor id="executor" pool-size="10" />
2. <!-- 名字必须是applicationEventMulticaster和messageSource是一样的，默认找这个名字的对象 -->
3. <!-- 名字必须是applicationEventMulticaster，因为AbstractApplicationContext默认找个 -->
4. <!-- 如果找不到就new一个，但不是异步调用而是同步调用 -->
5. <bean id="applicationEventMulticaster" class="org.springframework.context.event.SimpleApplicationEventMulticaster">
6.     <!-- 注入任务执行器 这样就实现了异步调用（缺点是全局的，要么全部异步，要么全部同步（删除这个属性即是同步） -->
7.     <property name="taskExecutor" ref="executor"/>
8. </bean>
```

通过注入taskExecutor来完成异步调用。具体实现可参考之前的代码介绍。这种方式的缺点很明显：要么大家都是异步，要么大家都不是。所以不推荐使用这种方式。

2、更灵活的异步支持

spring3提供了@Async注解来完成异步调用。此时我们可以使用这个新特性来完成异步调用。不仅支持异步调用，还支持简单的任务调度，比如我的项目就去掉Quartz依赖，直接使用spring3这个新特性，具体可参考spring-config.xml。

2.1、开启异步调用支持

Java代码 ☆

```
1. <!-- 开启@AspectJ AOP代理 -->
2. <aop:aspectj-autoproxy proxy-target-class="true"/>
3.
4. <!-- 任务调度器 -->
5. <task:scheduler id="scheduler" pool-size="10"/>
6.
7. <!-- 任务执行器 -->
8. <task:executor id="executor" pool-size="10"/>
9.
10. <!--开启注解调度支持 @Async @Scheduled-->
```

```
11. <task:annotation-driven executor="executor" scheduler="scheduler" proxy-target-class="true" />
```

2.2、配置监听器让其支持异步调用

Java代码 ☆

```
1. @Component
2. public class EmailRegisterListener implements ApplicationListener<RegisterEvent> {
3.     @Async
4.     @Override
5.     public void onApplicationEvent(final RegisterEvent event) {
6.         System.out.println("注册成功, 发送确认邮件给: " + ((User)event.getSource()).getUsername());
7.     }
8. }
```

使用@Async注解即可, 非常简单。

这样不仅可以支持通过调用, 也支持异步调用, 非常的灵活, 实际应用推荐大家使用这种方式。

通过如上, 大体了解了Spring的事件机制, 可以使用该机制非常简单的完成如注册流程, 而且对于比较耗时的调用, 可以直接使用Spring自身的异步支持来优化。

event.rar (10.5 KB)
下载次数: 28

jQuery MiniUI, 企业级Web开发

500%效率提升, 强大UI组件库! 支持java,.net,php,兼容ie6+



发表于 2013-07-11 16:16 陈雨晨 阅读(3430) 评论(1) 编辑 收藏

评论

re: Spring事件驱动模型

最代码转自: Spring基于事件驱动模型的订阅发布模式代码实例详解, 地址: <http://www.zuidaima.com/share/1791499571923968.htm>

最代码 评论于 2014-04-22 23:24 回复 更多评论

[新用户注册](#) [刷新评论列表](#)

找优秀程序员, 就在博客园

标题

姓名

主页

验证码 *

8945

内容(请不要发表任何与政治相关的内容)

☒ Remember Me?

提交

登录

[使用Ctrl+Enter键可以直接提交]