**EC-Central中利用消息发布订阅机制解除对第三方系统依赖**

1. **发布订阅模式简介**

Pub/Sub模式的核心在于有一个Publisher（发布者）和多个Subscriber（订阅者），当发布者发布一个消息时，所有的订阅了该消息的订阅者都会得到消息的通知；而这其中最大的特点是发布者不用关心有哪些订阅者的存在，也不用关心哪些订阅者订阅了哪些消息，更不用关心订阅者在收到消息后具体会做什么处理；这样订阅者是可以灵活的添加、移除，而不会影响到发布者的程序代码；

所以Pub/Sub模式会涉及到如下几种角色：

Publisher（发布者）

Message（消息）

Subscriber（订阅者）

Subscription List（消息订阅关系列表）

1. **EC-Central中采用发布订阅机制的目的**

在EC-Central系统中，我们除了完成本身所覆盖的电商标准流程外，可能还会涉及到与第三方系统之间的交互。如果是第三方系统需要通知或调用我们的服务，那么我们只需增加相关的Service即可，是不会依赖第三方系统的；但是如果是我们需要主动地去通知第三方系统，比如调用第三方系统的某个Service，那么必然的会导致我们的EC-Central系统中需要写相关的代码，从而产生了对第三方系统的依赖，而这个依赖关系又不是稳定的，可能不同的定制化项目实施时所需要的第三方系统是不一样的，那么这将会导致我们EC-Central产品的不稳定。

所以我们采用Pub/Sub模式的方式，来解决这个问题。

从上面对Pub/Sub模式的简介可以看出，该模式用来解除Publisher（发布者）与Subscriber（订阅者）之间的耦合关系。我们的EC-Central系统的角色就类似于Publisher，然后针对电商流程中的一些核心点，我们定义一系列的标准的Message，并在这些地方做一个消息发布的动作。剩下的就是外部第三方系统来订阅我们的标准消息了；当然外部第三方系统对相关消息的订阅以及接收到消息后需要做出的处理逻辑，这些代码还是需要我们自己来写，但是这部分代码就被剥离在了EC-Central标准产品之外，更加的独立和清晰，而不是和我们的标准电商流程混杂在一起了；这样当外部第三方系统发生变化时，我们也只需要对这部分代码进行修改、重写即可，而我们的标准电商流程的代码逻辑不受任何影响了。通过接口、多态、反射等技术实现手段，我们也可以更灵活的来增加或移除第三方系统的订阅关系。

1. **EC-Central中发布订阅机制具体的使用方式**

(1). 识别出需要发布消息的业务点，并定义出标准的消息类

在Service端项目解决方案的3PartService目录下有一个ECCentral.Service.EventMessage的工程，专门用来存放所定义的标准消息类；标准消息类需要继承ECCentral.Service.Utility.IEventMessage接口；

|  |
| --- |
| public class SampleMessage : IEventMessage  {  ……  } |

(2). 在需要发布消息的业务点，进行消息的发布动作

在BizProcessor中的相关地方，调用ECCentral.Service.Utility.EventPublisher.Publish<TMsg>( TMsg message)方法，来发布消息，其中泛型参数TMsg就为(1)里定义的标准消息，TMsg必须从ECCentral.Service.Utility.IEventMessage接口继承；

|  |
| --- |
| SampleMessage msg = new SampleMessage();  ……  EventPublisher.Publish<SampleMessage>(msg); |

(3). 定义订阅者，实现订阅后的处理逻辑

在Service端项目解决方案的3PartService目录下有一个ECCentral.Service.EventConsumer的工程，专门用来存放消息的订阅者（或者叫消息的消费使用者，所以用的Consumer这个单词）；定义的订阅者必须要实现接口ECCentral.Service.Utility.IConsumer<TMsg>，其中泛型参数TMsg就为(1)里定义的标准消息，TMsg必须从ECCentral.Service.Utility.IEventMessage接口继承，这样也就表示了该订阅者将只能用来订阅TMsg类型的消息；在接口方法void HandleEvent(T eventMessage)里完成对消息的处理逻辑，比如调用第三方的WebService或者发送SSB消息等动作；

|  |
| --- |
| public class SampleMessageConsumer : IConsumer<SampleMessage>  {  public void HandleEvent(SampleMessage\_1 eventMessage)  {  …… //TODO:做本Consumer的业务处理  }  } |

(4). 注册添加消息和订阅者之间的消息订阅关系

在ECCentral.Service.WebHost下的Configuration目录下有个EventConsumer.config的配置文件，在该文件里注册消息和订阅者的关系：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <subscription>  <event type="ECCentral.Service.EventMessage.SampleMessage\_1, ECCentral.Service.EventMessage">  <consumer type="ECCentral.Service.EventConsumer.SampleMessageConsumer\_A, ECCentral.Service.EventConsumer" />  <consumer type="ECCentral.Service.EventConsumer.SampleMessageConsumer\_B, ECCentral.Service.EventConsumer" />  </event>  <event type="ECCentral.Service.EventMessage.SampleMessage\_2, ECCentral.Service.EventMessage">  <consumer type="ECCentral.Service.EventConsumer.SampleMessageConsumer\_C, ECCentral.Service.EventConsumer" />  </event>  </subscription> |

其中每个<event></event>节点表示一个消息，其下的<consumer />节点表示一个对该消息的订阅者。该订阅者必须实现IConsumer<TMsg>接口，这个泛型TMsg必须为<event></event>节点上的type属性所指定的类型。

1. **补充说明**
2. 上面介绍了在EC-Central中通过4点步骤来使用发布订阅模式，以解决对第三方系统的耦合问题；这4点中最重要的也是最关键的，是第1、2两点，因为我们的EC-Central系统是不知道有哪些外部系统的订阅者，所以也是不知道它们的具体的对消息数据格式的需求，所以在第1点定义消息时，需要根据当前中蛋的情况，来考虑一个比较通用的消息格式，当然我们也不能过度考虑，毕竟消息结构还是可以在以后定制化有需要时再添加字段来扩展的，这个修改成本不大；另外一个就是上面的第2点，需要在哪些地方需要做消息的发布动作，这个我们现在也无法预知将来的需求，所以我们也就按照当前中蛋网的需求，来设计这些发布点就OK了，等将来发生了其他的具体需求后，我们再在相应地方增加对应消息的发布动作即可，这个修改成本也不大，可以接受的。
3. 我们的消息是使用的.Net的class类型来标示的，所以一个消息类型，就只能代表一个业务动作含义，我们要避免说2个或更多个业务动作来公用同一个消息类型，这样会倒在我们的Consumer来收到消息通知时无法识别是发生了哪一个业务操作。所以即使是2个或更多个业务动作需要的消息的数据格式（类的属性）完全一样，那么也必须分开定义出2个或更多个.Net的class类型来（这里可以通过A消息类型继承B消息类型的方式，来尽量减少相同的重复代码，但必须要定义出A和B两个类型来）。
4. 当前是为了中蛋实施的方便，所以直接在Service端的解决方案里添加了ECCentral.Service.EventConsumer工程，在将来的产品定制化过程中，这个部分可能会从EC-Central产品的解决方案里移除，而是作为单独的一个产品配件项目存在；在具体实施过程中，通过配置文件配置，来将我们的第三方系统的配件整合部署到我们的产品里（类似于插件式的模式）。在将来这些第三方系统的配件本身也是可以复用的模块，可以单独的进行维护和更新。