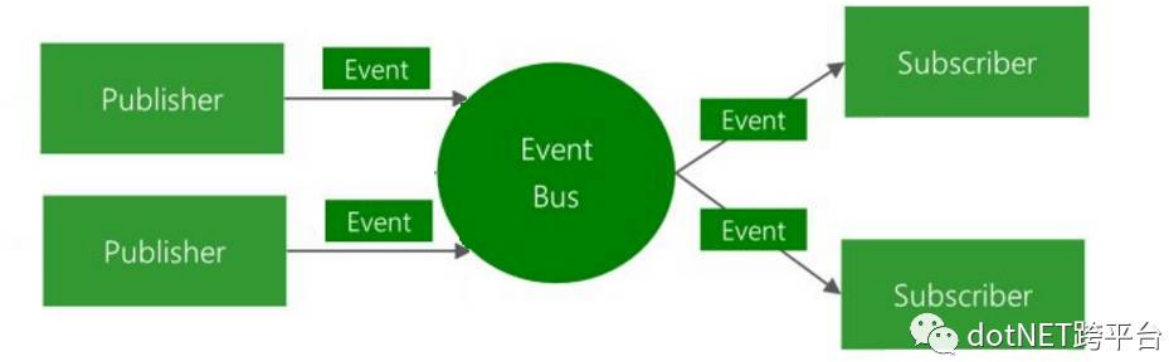
事件总线（Event Bus）知多少

2017-06-11 dotNET跨平台

1. 引言

事件总线这个概念对你来说可能很陌生，但提到观察者（发布-订阅）模式，你也许就很熟悉。事件总线是对发布-订阅模式的一种实现。它是一种集中式事件处理机制，允许不同的组件之间进行彼此通信而又不需要相互依赖，达到一种解耦的目的。

我们来看看事件总线的处理流程：



了解了事件总线的基本概念和处理流程，下面我们就来分析下如何去实现事件总线。

2.回归本质

在动手实现事件总线之前，我们还是要追本溯源，探索一下事件的本质和发布订阅模式的实现机制。

2.1.事件的本质

我们先来探讨一下事件的概念。都是读过书的，应该都还记得记叙文的六要素：时间、地点、人物、事件（起因、经过、结果）。

> 我们拿注册的案例，来解释一下。

> 用户输入用户名、邮箱、密码后，点击注册，输入无误校验通过后，注册成功并发送邮件给用户，要求用户进行邮箱验证激活。

这里面就涉及了两个主要事件：

1. 注册事件：起因是用户点击了注册按钮，经过是输入校验，结果是是否注册成功。
2. 发送邮件事件：起因是用户使用邮箱注册成功需要验证邮箱，经过是邮件发送，结果是邮件是否发送成功。

其实这六要素也适用于我们程序中事件的处理过程。开发过WinForm程序的都知道，我们在做UI设计的时候，从工具箱拖入一个注册按钮（btnRegister），双击它，VS就会自动帮我们生成如下代码：

```
void btnRegister_Click(object sender, EventArgs e){ // 事件的处理}
```

其中object sender指代发出事件的对象，这里也就是button对象；EventArgs e 事件参数，可以理解为对事件的描述 ，它们可以统称为**事件源**。其中的代码逻辑，就是对事件的处理。我们可以统称为**事件处理**。

说了这么多，无非是想透过现象看本质：**事件是由事件源和事件处理组成**。

2.2. 发布订阅模式

> 定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。 ——发布订阅模式

发布订阅模式主要有两个角色：

- 发布方（Publisher）：也称为被观察者，当状态改变时负责通知所有订阅者。
- 订阅方（Subscriber）：也称为观察者，订阅事件并对接收到的事件进行处理。

发布订阅模式有两种实现方式：

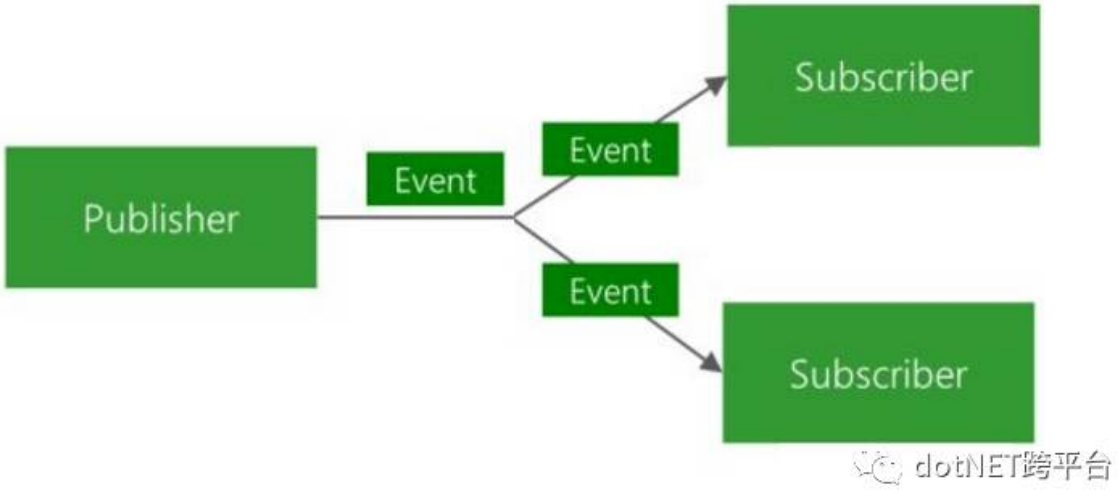
- 简单的实现方式：由Publisher维护一个订阅者列表，当状态改变时循环遍历列表通知订阅者。
- 委托的实现方式：由Publisher定义事件委托，Subscriber实现委托。

总的来说，发布订阅模式中有两个关键字，通知和更新。

被观察者状态改变通知观察者做出相应更新。

解决的是当对象改变时需要通知其他对象做出相应改变的问题。

如果画一个图来表示这个流程的画，图形应该是这样的：



3 实现发布订阅模式

相信通过上面的解释，对事件和发布订阅模式有了一个大概的印象。都说理论要与实践相结合，所以我们还是动动手指敲敲代码比较好。

我将以『观察者模式』来钓鱼这个例子为基础，通过重构的方式来完善一个更加通用的发布订阅模式。

先上代码：

```
/// <summary>/// 鱼的品类枚举/// </summary>public enum FishType
{
        鲫鱼,
        鲤鱼,
        黑鱼,
        青鱼,
        草鱼,
        鲈鱼
}
```

钓鱼竿的实现：

```
/// <summary>
///        鱼竿（被观察者）
/// </summary>
public class FishingRod
{       public delegate void FishingHandler(FishType type); //声明委托
       public event FishingHandler FishingEvent; //声明事件

       public void ThrowHook(FishingMan man)        {
              Console.WriteLine("开始下钩！");

       //用随机数模拟鱼咬钩，若随机数为偶数，则为鱼咬钩
              if (new Random().Next() % 2 == 0)
              {                    var type = (FishType) new Random().Next(0, 5);
                     Console.WriteLine("铃铛：叮叮叮，鱼儿咬钩了");
   if (FishingEvent != null)
                           FishingEvent(type);
              }
       }
}
```

垂钓者：

```
/// <summary>///        垂钓者（观察者）/// </summary>public class FishingMan{
  public FishingMan(string name)       {
             Name = name;
      }       public string Name { get; set; }       public int FishCount { get;
set; }       /// <summary>
       /// 垂钓者自然要有鱼竿啊
       /// </summary>
       public FishingRod FishingRod { get; set; }       public void Fishing()
  {            this.FishingRod.ThrowHook(this);
       }       public void Update(FishType type)       {
              FishCount++;
```

```
            Console.WriteLine("{0}：钓到一条[{2}]，已经钓到{1}条鱼了！", Name,
FishCount, type);
        }
}
```

场景类也很简单：

```
//1、初始化鱼竿var fishingRod = new FishingRod();//2、声明垂钓者var jeff = new
FishingMan("圣杰");//3.分配鱼竿jeff.FishingRod = fishingRod;//4、注册观察者
fishingRod.FishingEvent += jeff.Update;//5、循环钓鱼while (jeff.FishCount < 5)
{
        jeff.Fishing();
        Console.WriteLine("--------------------");       //睡眠5s
        Thread.Sleep(5000);
}
```

代码很简单，相信你一看就明白。但很显然这个代码实现仅适用于当前这个钓鱼场景，假如有其他场景也想使用这个模式，我们还需要重新定义委托，重新定义事件处理，岂不很累。本着”Don't repeat yourself“的原则，我们要对其进行重构。

结合我们对事件本质的探讨，事件是由事件源和事件处理组成。针对我们上面的案例来说，`public delegate void FishingHandler(FishType type);`这句代码就已经说明了事件源和事件处理。事件源就是`FishType type`，事件处理自然是注册到`FishingHandler`上面的委托实例。

问题找到了，很显然是我们的事件源和事件处理不够抽象，所以不能通用，下面咱们就来动手改造。

3.1. 提取事件源

事件源应该至少包含事件发生的时间和触发事件的对象。

我们提取`IEventData`接口来封装事件源：

```
/// <summary>/// 定义事件源接口，所有的事件源都要实现该接口/// </summary>public interface IEventData{      /// <summary>
      /// 事件发生的时间
      /// </summary>
      DateTime EventTime { get; set; }      /// <summary>
      /// 触发事件的对象
      /// </summary>
      object EventSource { get; set; }
}
```

自然我们应该给一个默认的实现EventData：

```
/// <summary>/// 事件源：描述事件信息，用于参数传递/// </summary>public class EventData : IEventData{      /// <summary>
      /// 事件发生的时间
      /// </summary>
      public DateTime EventTime { get; set; }      /// <summary>
      /// 触发事件的对象
      /// </summary>
      public Object EventSource { get; set; }      public EventData()      {
            EventTime = DateTime.Now;
      }
}
```

针对Demo，扩展事件源如下：

```
public class FishingEventData : EventData{      public FishType FishType { get; set; }      public FishingMan FisingMan { get; set; }
}
```

完成后，我们就可以去把在FishingRod声明的委托参数类型改为FishingEventData类型了，即public delegate void FishingHandler(FishingEventData eventData); //声明委托；

然后修改FishingMan的Update方法按委托定义的参数类型修改即可，代码我就不放了，大家自行脑补。

到这一步我们就统一了事件源的定义方式。

3.2.提取事件处理器

事件源统一了，那事件处理也得加以限制。比如如果随意命名事件处理方法名，那在进行事件注册的时候还要去按照委托定义的参数类型去匹配，岂不麻烦。

我们提取一个IEventHandler接口：

```
/// <summary>
/// 定义事件处理器公共接口，所有的事件处理都要实现该接口
/// </summary>
public interface IEventHandler
{
}
```

事件处理要与事件源进行绑定，所以我们再来定义一个泛型接口：

```
/// <summary>
/// 泛型事件处理器接口
/// </summary>
/// <typeparam name="TEventData"></typeparam>
public interface IEventHandler<TEventData> : IEventHandler where TEventData :
IEventData
{       /// <summary>
        /// 事件处理器实现该方法来处理事件
        /// </summary>
        /// <param name="eventData"></param>
        void HandleEvent(TEventData eventData);
}
```

你可能会纳闷，为什么先定义了一个空接口？这里就留给自己思考吧。

至此我们就完成了事件处理的抽象。我们再继续去改造我们的Demo。我们让FishingMan实现IEventHandler接口，然后修改场景类中将fishingRod.FishingEvent += jeff.Update;改为fishingRod.FishingEvent += jeff.HandleEvent;即可。代码改动很简单，同样在此略去。

至此你可能觉得我们完成了对Demo的改造。但事实上呢，我们还要弄清一个问题——如果这个FishingMan订阅的有其他的事件，我们该如何处理？

聪颖如你，你立马想到了可以**通过事件源来进行区分处理**。

```
public class FishingMan : IEventHandler<IEventData>{      //省略其他代码
      public void HandleEvent(IEventData eventData)
      {            if (eventData is FishingEventData)
            {                  //do something
            }            if(eventData is XxxEventData)
            {                  //do something else
            }
      }
}
```

至此，这个模式实现到这个地步基本已经可以通用了。

4. 实现事件总线

通用的发布订阅模式不是我们的目的，我们的目的是一个集中式的事件处理机制，且各个模块之间相互不产生依赖。那我们如何做到呢？同样我们还是一步一步的进行分析改造。

4.1.分析问题

思考一下，每次为了实现这个模式，都要完成以下三步：

1. 事件发布方定义事件委托
2. 事件订阅方定义事件处理逻辑
3. 显示的订阅事件

虽然只有三步，但这三步已经很繁琐了。而且事件发布方和事件订阅方还存在着依赖（体现在订阅者要显示的进行事件的注册和注销上）。而且当事件过多时，直接在订阅者中实现IEventHandler接口处理多个事件逻辑显然不太合适，违法单一职责原则。这里就暴露了三个问题：

1. 如何精简步骤？
2. 如何解除发布方与订阅方的依赖？
3. 如何避免在订阅者中同时处理多个事件逻辑？

带着问题思考，我们就会更接近真相。

想要精简步骤，那我们需要寻找共性。共性就是事件的本质，也就是我们针对事件源和事件处理提取出来的两个接口。

想要解除依赖，那就要在发布方和订阅方之间添加一个中介。

想要避免订阅者同时处理过多事件逻辑，那我们就把事件逻辑的处理提取到订阅者外部。

思路有了，下面我们就来实施吧。

4.2.解决问题

本着先易后难的思想，我们下面就来解决以上问题。

4.2.1. 实现IEventHandler

我们先解决上面的第三个问题：如何避免在订阅者中同时处理多个事件逻辑？

自然是针对不同的事件源IEventData实现不同的IEventHandler。改造后的钓鱼事件处理逻辑如下：

```
/// <summary>/// 钓鱼事件处理/// </summary>public class FishingEventHandler : IEventHandler<FishingEventData>
{       public void HandleEvent(FishingEventData eventData)       {
            eventData.FishingMan.FishCount++;
```

```
            Console.WriteLine("{0}：钓到一条[{2}]，已经钓到{1}条鱼了！",
                  eventData.FishingMan.Name, eventData.FishingMan.FishCount,
eventData.FishType);

      }
}
```

这时我们就可以移除在FishingMan中实现的IEventHandler接口了。

然后将事件注册改为fishingRod.FishingEvent += new FishingEventHandler().HandleEvent;即可。

4.2.2. 统一注册事件

上一个问题的解决，有助于我们解决第一个问题：如何精简流程？

为什么呢，因为我们是根据事件源定义相应的事件处理的。也就是我们之前说的可以根据事件源来区分事件。

然后呢？反射，我们可以通过反射来进行事件的统一注册。

在FishingRod的构造函数中使用反射，统一注册实现了IEventHandler<FishingEventData>类型的实例方法HandleEvent：

```
public FishingRod()
{
      Assembly assembly = Assembly.GetExecutingAssembly();

      foreach (var type in assembly.GetTypes())
      {            if (typeof(IEventHandler).IsAssignableFrom(type))//判断当前类
型是否实现了IEventHandler接口
            {
                  Type handlerInterface =
type.GetInterface("IEventHandler`1");//获取该类实现的泛型接口
```

```
                    Type eventDataType = handlerInterface.GetGenericArguments()
[0]; // 获取泛型接口指定的参数类型

                    //如果参数类型是FishingEventData，则说明事件源匹配
                    if (eventDataType.Equals(typeof(FishingEventData)))
                    {                           //创建实例
                        var handler = Activator.CreateInstance(type) as
IEventHandler<FishingEventData>;                            //注册事件
                        FishingEvent += handler.HandleEvent;
                    }
                }
            }
}
```

这样，我们就可以移出场景类中的显示注册代码`fishingRod.FishingEvent += new FishingEventHandler().HandleEvent;`。

4.2.3. 解除依赖

如何解除依赖呢？其实答案就在本文的两张图上，仔细对比我们可以很直观的看到，Event Bus就相当于一个介于Publisher和Subscriber中间的桥梁。它隔离了Publlisher和Subscriber之间的直接依赖，接管了所有事件的发布和订阅逻辑，并负责事件的中转。

Event Bus终于要粉墨登场了！！！

分析一下，如果EventBus要接管所有事件的发布和订阅，那它则需要有一个容器来记录事件源和事件处理。那又如何触发呢？有了事件源，我们就自然能找到绑定的事件处理逻辑，通过反射触发。代码如下：

```
/// <summary>/// 事件总线/// </summary>

public class EventBus{
```

```
  public static EventBus Default => new EventBus();      /// <summary>
      /// 定义线程安全集合
      /// </summary>
      private readonly ConcurrentDictionary<Type, List<Type>>
_eventAndHandlerMapping;      public EventBus()      {
            _eventAndHandlerMapping = new ConcurrentDictionary<Type, List<Type>>
();
            MapEventToHandler();
      }      /// <summary>
      ///通过反射，将事件源与事件处理绑定
      /// </summary>
      private void MapEventToHandler()      {
            Assembly assembly = Assembly.GetEntryAssembly();

     foreach (var type in assembly.GetTypes())
            {                   if
(typeof(IEventHandler).IsAssignableFrom(type))//判断当前类型是否实现了
IEventHandler接口
                  {
                        Type handlerInterface =
type.GetInterface("IEventHandler`1");//获取该类实现的泛型接口
                        if (handlerInterface != null)
                        {
                              Type eventDataType =
handlerInterface.GetGenericArguments()[0]; // 获取泛型接口指定的参数类型

                              if
(_eventAndHandlerMapping.ContainsKey(eventDataType))
                              {
                                    List<Type> handlerTypes =
_eventAndHandlerMapping[eventDataType];
                                    handlerTypes.Add(type);
                                    _eventAndHandlerMapping[eventDataType] =
handlerTypes;
                              }                                   else
```

```
                                {                                       var
handlerTypes = new List<Type> { type };
                                        _eventAndHandlerMapping[eventDataType] =
handlerTypes;
                                }
                            }
                        }
                    }
        }       /// <summary>
        /// 手动绑定事件源与事件处理
        /// </summary>
        /// <typeparam name="TEventData"></typeparam>
        /// <param name="eventHandler"></param>
        public void Register<TEventData>(Type eventHandler)
        {
                List<Type> handlerTypes =
_eventAndHandlerMapping[typeof(TEventData)];

      if (!handlerTypes.Contains(eventHandler))
                {
                        handlerTypes.Add(eventHandler);
                        _eventAndHandlerMapping[typeof(TEventData)] = handlerTypes;
                }
        }       /// <summary>
        /// 手动解除事件源与事件处理的绑定
        /// </summary>
        /// <typeparam name="TEventData"></typeparam>
        /// <param name="eventHandler"></param>
        public void UnRegister<TEventData>(Type eventHandler)
        {
                List<Type> handlerTypes =
_eventAndHandlerMapping[typeof(TEventData)];

              if (handlerTypes.Contains(eventHandler))
               {
```

```
                handlerTypes.Remove(eventHandler);
                _eventAndHandlerMapping[typeof(TEventData)] = handlerTypes;
            }
        }       /// <summary>
        /// 根据事件源触发绑定的事件处理
        /// </summary>
        /// <typeparam name="TEventData"></typeparam>
        /// <param name="eventData"></param>
        public void Trigger<TEventData>(TEventData eventData)

            where TEventData : IEventData
        {
            List<Type> handlers = _eventAndHandlerMapping[eventData.GetType()];
            if (handlers != null && handlers.Count > 0)
            {                   foreach (var handler in handlers)
                {
                    MethodInfo methodInfo =
handler.GetMethod("HandleEvent");                           if (methodInfo != null)
                    {                               object obj =
Activator.CreateInstance(handler);
                        methodInfo.Invoke(obj, new object[] { eventData
});
                    }
                }
            }
        }
}
```

事件总线主要定义三个方法，注册、取消注册、事件触发。还有一点就是我们在构造函数中通过反射去进行事件源和事件处理的绑定。

代码注释已经很清楚了，这里就不过多解释了。

下面我们就来修改Demo，修改FishingRod的事件触发：

```
/// <summary>/// 下钩/// </summary>

public void ThrowHook(FishingMan man){
       Console.WriteLine("开始下钩！");

//用随机数模拟鱼咬钩，若随机数为偶数，则为鱼咬钩
       if (new Random().Next() % 2 == 0)
       {              var a = new Random(10).Next();

  var type = (FishType)new Random().Next(0, 5);
             Console.WriteLine("铃铛：叮叮叮，鱼儿咬钩了");

     if (FishingEvent != null)
             {                    var eventData = new FishingEventData() { FishType
= type, FishingMan = man };

        //FishingEvent(eventData);//不再需要通过事件委托触发
                     EventBus.Default.Trigger<FishingEventData>(eventData);//直接通
过事件总线触发即可
             }
       }
}
```

至此，事件总线的雏形已经形成！

5.事件总线的总结

通过上面一步一步的分析和实践，发现事件总线也不是什么高深的概念，只要我们自己善于思考，勤于动手，也能实现自己的事件总线。

根据我们的实现，大概总结出以下几条：

1. 事件总线维护一个事件源与事件处理的映射字典；
2. 通过单例模式，确保事件总线的唯一入口；
3. 利用反射完成事件源与事件处理的初始化绑定；

4. 提供统一的事件注册、取消注册和触发接口。

最后，以上事件总线的实现只是一个雏形，还有很多潜在的问题。有兴趣的不妨思考完善一下，我也会继续更新完善，尽情期待！

参考资料

ABP EventBus

DDD~领域事件与事件总线

DDD事件总线的实现

原文地址：http://www.cnblogs.com/sheng-jie/p/6970091.html