# [EventBus 事件总线之个人理解](https://www.cnblogs.com/hfdel/p/7172914.html)

**用例：假设公司发布了一个公告 需要通过短信 和 邮件分别2种方式 通知员工**

**1:首先我们建立领域模型**

[IMG_256](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 领域核心基类

/// </summary>

public abstract class Core

{

public string Id { set; get; } = Guid.NewGuid().ToString();

}

public interface ICore

{

}

[IMG_257](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

**2:消息模型**

[IMG_258](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 通知的领域模型

/// </summary>

public class Notice : Core

{

/// <summary>

/// 通知内容

/// </summary>

public string Message { set; get; }

/// <summary>

/// 通知发送时间

/// </summary>

public DateTime DateTime { set; get; } = DateTime.Now;

}

[IMG_259](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

这个时候我们会想到 建立2个服务类 一个是SmsService 和 EmailService服务 分别用来发送短信和Email

[IMG_260](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

public class EmailService

{

public EmailService() { }

public EmailService(Entity.Notice notice) => Console.WriteLine($"邮件通知:{notice.Message} 发送时间:{notice.DateTime}");

}

[IMG_261](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

[IMG_262](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

public class SmsService

{

public SmsService() { }

public SmsService(Entity.Notice notice) => Console.WriteLine($"短信通知:{notice.Message} 发送时间:{notice.DateTime}");

}

[IMG_263](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

[IMG_264](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

static void Main(string[] args)

{

Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);

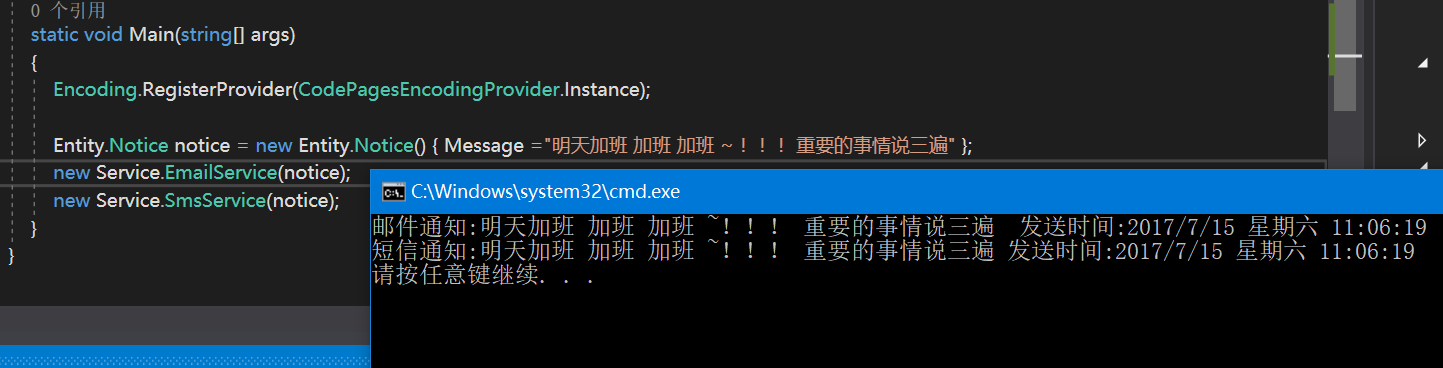
Entity.Notice notice = new Entity.Notice() { Message ="明天加班 加班 加班 ~！！！ 重要的事情说三遍" };

new Service.EmailService(notice);

new Service.SmsService(notice);

}

[IMG_265](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

Encoding.RegisterProvider(CodePagesEncodingProvider.Instance)   
这里主要是因为.NET CORE中文输出会导致乱码 需要加上编码配置。也可以换上非Core平台  
看看运行后的效果  
  
  
运行后的效果似乎已经满足了我们的需求 公司公告分别以2种方式发送出去了  
这样就带来了一个问题 如果未来社交发展需要多平台发送通知呢。。假设这里有还有QQ 默默 探探 钉钉 微信 等等  
这一系列的消息推送方式 那我们的代码里是不是这样 ？

[IMG_267](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

Entity.Notice notice = new Entity.Notice() { Message ="明天加班 加班 加班 ~！！！ 重要的事情说三遍" };

new Service.EmailService(notice);

new Service.SmsService(notice);

new Service.QQService(notice);

new Service.WeiXinService(notice);

new Service.MomoService(notice);

new Service.DingDService(notice);

new Service.TanTanService(notice);

[IMG_268](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

 这样写显然看起来是一个非常蛋疼的事。仔细想想这一系列的消息推送不变的是 事件消息源 也就是Notice对象

 并且所有发送消息的方法都是被动的接收这个对象 , 这样我们就可以把2者的关系理解成 消息源是发布者,

具体处理消息发送的是订阅者,从而我们换一个思路去改造之前的方法

[IMG_269](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

public interface IEventHandler<T> where T : Entity.Core

{

/// <summary>

/// 订阅对象的具体实现

/// </summary>

/// <param name="entity"></param>

void Handler(T entity);

}

[IMG_270](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

定义一个泛型接口,所有订阅者必须实现这个接口

[IMG_271](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// Email形式处理

/// </summary>

public class EmailEventHandler : IEventHandler<Entity.Notice>

{

public void Handler(Notice notice)

{

Console.WriteLine($"邮件通知:{notice.Message} 发送时间:{notice.DateTime}");

}

}

[IMG_272](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

[IMG_273](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 短信形式处理

/// </summary>

public class SmsEventHandler : IEventHandler<Entity.Notice>

{

public void Handler(Notice notice)

{

Console.WriteLine($"短信通知:{notice.Message} 发送时间:{notice.DateTime}");

} }

[IMG_274](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

这是改造的第一步,改造的目的是让它能以一种自动的方式处理,而不像之前一样需要一个一个对象的new出来,

做到能像看电视一样,只要你打开电视就能收看到传输过来的画面

定义一个消息总线的接口 用来管理 订阅者的注册 以及消息的发布

[IMG_275](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

public interface IBus

{

/// <summary>

/// 默认订阅所有事件消息 启动时调用

/// </summary>

void SubscribeAll();

/// <summary>

/// 订阅

/// </summary>

/// <param name="type"></param>

/// <param name="data"></param>

void Subscribe(Type type, object data);

/// <summary>

/// 发布订阅

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="eneity"></param>

void Publish<T>(T eneity) where T : Entity.Core;

}

[IMG_276](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

定义一个BusManager来实现 IBus

[IMG_277](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

private BusManager()

{

if (\_instance == null)

Bus = new BusService();

}

private static object \_lock = new object();

private static BusManager \_instance;

public static IBus Instance

{

get

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

\_instance = new BusManager();

}

}

return \_instance;

}

}

[IMG_278](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

[IMG_279](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 主要是针对 事件源和具体的事件处理注册关系。ConcurrentDictionary保证了线程的安全

/// </summary>

private static ConcurrentDictionary<Type, List<object>> \_dicHandlers = new ConcurrentDictionary<Type, List<object>>();

[IMG_280](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

[IMG_281](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 判断type是否是abs的实现类或者子类/

/// .net Core中对反射进行了单独的处理 反射后一些详细属性都要通过GetTypeInfo获取 原对象只保留了一些基本属性

/// </summary>

/// <param name="type"></param>

/// <param name="abs"></param>

/// <returns></returns>

private bool IsAssignableFrom(Type type, Type abs)

{

/// if ((abs.GetTypeInfo().IsAbstract || abs.GetTypeInfo().IsInterface) && abs.IsAssignableFrom(type))

return true;

else

{

if (type.GetInterfaces().Any(o => o.GetTypeInfo().IsGenericType && o.GetGenericTypeDefinition() == abs))

return true;

}

return false;

}

/// <summary>

/// 判断2个类型是否相同

/// </summary>

private Func<object, object, bool> \_Equals = (o1, o2) =>

{

return o1.GetType() == o2.GetType();

};

[IMG_282](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

上述方法都是为了实现IBus接口所做的铺垫 如有.Net Core反射疑问的自行Bing

[IMG_283](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

public void Subscribe(Type type, object data)

{

lock (\_lock)

{

if (\_dicHandlers.ContainsKey(type))

{

var \_handlers = \_dicHandlers[type];

if (!\_handlers.Any(o => \_Equals(o, data)))

\_handlers.Add(data);

}

else

{

\_dicHandlers[type] = new List<object>() { data };

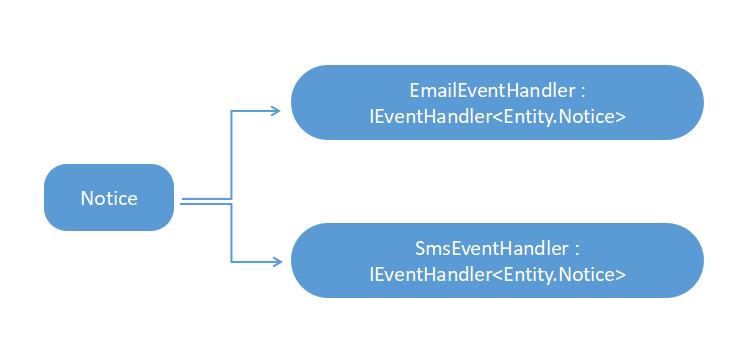
}

}

}

[IMG_284](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

实现的单个对象的加载 Key为具体的发布对象类型, Value是具体的订阅者的行为实现集合,这里是多个订阅者的实现



ConcurrentDictionary 里的关系

[IMG_286](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 初始化 默认的所有实现都订阅事件

/// </summary>

public void SubscribeAll()

{

///加载程序集,具体的你也可以加载项目路径下所有的dll或者exe

var assembly = Assembly.Load(new AssemblyName("ConsoleApp2"));

assembly.GetTypes().Where(x => x.GetTypeInfo().IsClass && !x.GetTypeInfo().IsAbstract && !x.GetTypeInfo().IsInterface).ToList().ForEach(x =>

{

if (IsAssignableFrom(x, typeof(IEventHandler<>)))

{

///反射创建对象

var entity = Activator.CreateInstance(x);

///获得到泛型参数的类型

var key = x.GetInterfaces().FirstOrDefault().GetGenericArguments().FirstOrDefault();

this.Subscribe(key, entity);

}

});

}

[IMG_287](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

事件源的发布

[IMG_288](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

/// <summary>

/// 事件源的发布

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="eneity"></param>

public void Publish<T>(T eneity) where T : Core

{

var type = eneity.GetType();

if (\_dicHandlers.ContainsKey(type) && \_dicHandlers[type] != null)

{

\_dicHandlers[type].ForEach(o =>

{

var eve = o as IEventHandler<T>;

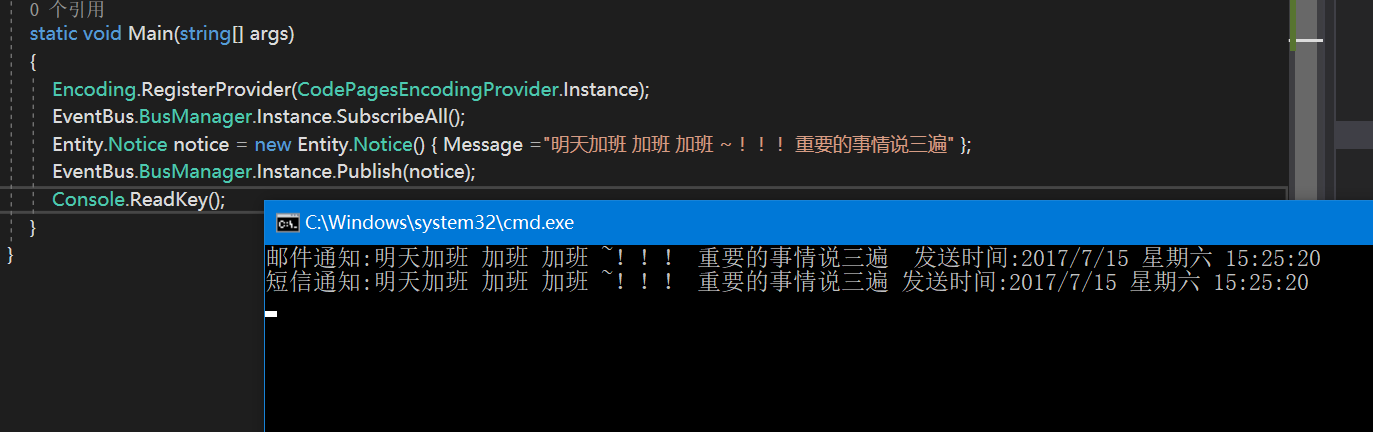
eve.Handler(eneity);

});

}

}

[IMG_289](https://www.cnblogs.com/hfdel/p/javascript:void(0);)

上面的所有代码基本上就完成了一个简单的事件总线驱动的模式，  
试试最终的运行效果。  


是不是感觉很神奇,这就是单纯自己理解的事件总线模式。