链滴

应用程序级别的事件发布和订阅

作者：loogn
原文链接：https://ld246.com/article/1566179408079
来源网站：链滴

认识发布者/订阅者模式

情景：当一个特定的程序事件发生时，程序的其他部分可以得到该事件已经发生的通知。

发布者定义一系列事件，并提供一个注册方法；订阅者向发布者注册，并提供一个可被回调的方法，就是事件处理程序；当事件被触发的时候，订阅者得到通知，而订阅者所提交的所有方法都会被执行。

在C#中，事件可以是类的成员，如果是单个类中的事件并不需要特别的方式来管理，那是语音本身具的功能，但是如果放在整个项目中（很多个类），也就是应用程序级别的事件发布和订阅，就需要一统一的机制来管理了。

简单思路

说到事件，少不了的是委托，因为事件就是委托，而且是多播委托，我们需要一个集合来存储所有注的事件处理程序，并且还需要一个唯一的名字来标识每一个事件，这样才能根据这个名字来触发对应事件，进而执行对应的处理程序。

AppEventService

```
    /// <summary>
    /// 应用程序事件管理器
    /// </summary>
    public class AppEventService
    {
        private readonly ConcurrentDictionary<string, Func<object[], object>> _eventHandlerDi
t =
            new ConcurrentDictionary<string, Func<object[], object>>();

        private IServiceProvider _serviceProvider;


        /// <summary>
        /// 实例化AppEventService
        /// </summary>
        public AppEventService()
        {
        }

        public void SetServiceProvider(IServiceProvider serviceProvider)
        {
            _serviceProvider = serviceProvider;
        }

        /// <summary>
        /// 添加事件处理程序
        /// </summary>
        /// <param name="eventKey"></param>
        /// <param name="func"></param>
        public void AddHandler(string eventKey, Func<object[], object> func)
        {
            _eventHandlerDict.AddOrUpdate(eventKey, func, (key, oldFun) => { return oldFun +=
unc; });
```

原文链接：应用程序级别的事件发布和订阅

```
    }

    /// <summary>
    /// 触发事件
    /// </summary>
    /// <param name="eventKey"></param>
    /// <param name="ps"></param>
    /// <returns></returns>
    public Task Fire(string eventKey, params object[] ps)
    {
        return _eventHandlerDict.TryGetValue(eventKey, out var func)
            ? Task.Run(() => func(ps))
            : Task.CompletedTask;
    }

    /// <summary>
    /// 扫码程序集，注册事件处理程序
    /// </summary>
    /// <param name="assembly"></param>
    public void ScanEventHandler(Assembly assembly)
    {
        foreach (var type in assembly.GetTypes())
        {
            var methodInfos = type.GetMethods(BindingFlags.Static | BindingFlags.Instance | Bi
dingFlags.Public |
                                              BindingFlags.NonPublic ^ BindingFlags.GetProperty ^
                                              BindingFlags.SetProperty);

            foreach (var methodInfo in methodInfos)
            {
                var mehAttr = methodInfo.GetCustomAttribute<AppEventHandlerAttribute>();
                if (mehAttr == null) continue;
                var fun = DynamicMethodHelper.GetExecuteDelegate(methodInfo);

                AddHandler(mehAttr.EventKey,
                    (args) => fun(methodInfo.IsStatic ? null : Activator.CreateInstance(type), args));
            }
        }
    }

    /// <summary>
    /// 扫描应用程序域中的程序集
    /// </summary>
    public void ScanEventHandler()
    {
        foreach (var assembly in AppDomain.CurrentDomain.GetAssemblies())
        {
            ScanEventHandler(assembly);
        }
    }

    /// <summary>
    /// 扫描容器中的服务，注册时间处理程序
    /// </summary>
```

```
        /// <param name="services"></param>
        public void ScanEventHandler(IServiceCollection services)
        {
            foreach (var service in services)
            {
                var methodInfos = service.ServiceType.GetMethods(
                    BindingFlags.Static | BindingFlags.Instance | BindingFlags.Public | BindingFlags.N
nPublic ^
                    BindingFlags.GetProperty ^ BindingFlags.SetProperty);
                foreach (var methodInfo in methodInfos)
                {
                    var mehAttr = methodInfo.GetCustomAttribute<AppEventHandlerAttribute>();
                    if (mehAttr == null) continue;
                    var fun = DynamicMethodHelper.GetExecuteDelegate(methodInfo);

                    AddHandler(mehAttr.EventKey,
                        (args) => fun(methodInfo.IsStatic ? null : _serviceProvider.GetService(service.Se
viceType),
                            args));
                }
            }
        }
    }
```

该类中_eventHandlerDict来存储注册的事件处理程序，有几个添加或者批量扫描事件处理程序的方，另外一个就是根据唯一标识触发事件的方法Fire，并且可以出入所需的参数。还有一个_serviceProvider字段，是为了更好的结合Ioc框架使用，后面有说明。

其中DynamicMethodHelper.GetExecuteDelegate方法会把一个MethodInfo动态生成一个委托，为了提高效率，实现如下：

```
public static Func<object, object[], object> GetExecuteDelegate(MethodInfo methodInfo)
        {
            // parameters to execute
            ParameterExpression instanceParameter =
                Expression.Parameter(typeof(object), "instance");
            ParameterExpression parametersParameter =
                Expression.Parameter(typeof(object[]), "parameters");

            // build parameter list
            List<Expression> parameterExpressions = new List<Expression>();
            ParameterInfo[] paramInfos = methodInfo.GetParameters();
            for (int i = 0; i < paramInfos.Length; i++)
            {
                // (Ti)parameters[i]
                BinaryExpression valueObj = Expression.ArrayIndex(
                    parametersParameter, Expression.Constant(i));
                UnaryExpression valueCast = Expression.Convert(
                    valueObj, paramInfos[i].ParameterType);

                parameterExpressions.Add(valueCast);
            }

            // non-instance for static method, or ((TInstance)instance)
            Expression instanceCast = methodInfo.IsStatic ? null :
```

```
            Expression.Convert(instanceParameter, methodInfo.ReflectedType);

        // static invoke or ((TInstance)instance).Method
        MethodCallExpression methodCall = Expression.Call(
            instanceCast, methodInfo, parameterExpressions);

        // ((TInstance)instance).Method((T0)parameters[0], (T1)parameters[1], ...)
        if (methodCall.Type == typeof(void))
        {
            Expression<Action<object, object[]>> lambda =
                Expression.Lambda<Action<object, object[]>>(
                    methodCall, instanceParameter, parametersParameter);

            Action<object, object[]> execute = lambda.Compile();
            return (instance, parameters) =>
            {
                execute(instance, parameters);
                return null;
            };
        }
        else
        {
            UnaryExpression castMethodCall = Expression.Convert(
                methodCall, typeof(object));
            Expression<Func<object, object[], object>> lambda =
                Expression.Lambda<Func<object, object[], object>>(
                    castMethodCall, instanceParameter, parametersParameter);

            return lambda.Compile();
        }
    }
```

AppEventHandlerAttribute

```
/// <summary>
/// 标记方法是appEvent的事件处理程序
/// </summary>
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = false)]
public class AppEventHandlerAttribute : Attribute
{
    public AppEventHandlerAttribute(string eventKey)
    {
        EventKey = eventKey;
    }
    /// <summary>
    /// key
    /// </summary>
    public string EventKey { get; set; }
}
```

我们批量扫描的时候，会判断方法是否有此特性，有特性的视为事件处理程序，EventKey就是我们上说的唯一标识。

基本用法

在普通的项目中，没Ioc的话，上面的代码基本就可以用了，应用程序开始的时候，执行扫描或添加间处理程序（前提是方法上有AppEventHandlerAttribute特性），然后再适当的时候使用AppEventservice.Fire触发事件就行了，如果要更好的在ioc框架中使用，我们再提供额外的方法。

Ioc中使用

```
namespace Microsoft.Extensions.DependencyInjection
{
    public static class AppEventExtensions
    {
        /// <summary>
        /// 注册应用程序域中所有有AppService特性的类
        /// </summary>
        /// <param name="services"></param>
        public static void AddAppEvents(this IServiceCollection services)
        {
            AppEventService appEventService = new AppEventService();
            appEventService.ScanEventHandler(services);
            services.AddSingleton(appEventService);
        }
    }
}
```

按照惯例，我们在IServiceCollection添加AddAppEvents扩展方法，以单例方式注入AppEventServie。

```
namespace Microsoft.AspNetCore.Builder
{
    public static class AppEventBuilderExtensions
    {
        public static void UseAppEvents(this IApplicationBuilder app)
        {
            app.ApplicationServices.GetService<AppEventService>().SetServiceProvider(app.Applications);
        }
    }
}
```

按照惯例，我们在IApplicationBuilder中添加UseAppEvents扩展方法。

然后我们就可以使用常规的方法来使用AppEventService的功能了。

后记

实在没啥东西，但是的确在应用程序级别解耦了，这个功能没有考虑返回值的问题，我感觉也不需要虑返回值的问题，如果你感觉需要返回值，直接调用就好了，使用这个模式反倒不合适了。没有事务障，不过如果需要，可以结合工作单元模式来实现，或者在方法参数中传递工作单元。