   事件为类型定义了外发接口，C#的事件是建立在委托的基础上的，委托为事件处理器提供了类型安全的函数签名。

    委托要比事件的使用范围广泛，我们可以把事件看做是一种经过了封装的委托，专门用于事件驱动模型。你可以在客户代码中直接调用委托来激发委托指向的函数，而事件不可以,你只能在服务端调用事件，在客户端调用事件是会引发编译错误的。我们来看下面的程序。

**[csharp]** [view plain](http://blog.csdn.net/yin554393109/article/details/38108645" \o "view plain) [copy](http://blog.csdn.net/yin554393109/article/details/38108645" \o "copy" \t "http://blog.csdn.net/yin554393109/article/details/_blank) [IMG_256](https://code.csdn.net/snippets/434912" \o "在CODE上查看代码片" \t "http://blog.csdn.net/yin554393109/article/details/_blank)[IMG_257](https://code.csdn.net/snippets/434912/fork" \o "派生到我的代码片" \t "http://blog.csdn.net/yin554393109/article/details/_blank)

1. **public** **class** EventTest
2. {
3. **public** **delegate** **int** Add(**int** value1, **int** value2);
4. **public** **event** Add AddHandler;
5. **public** Add AddDelegate;
7. **public** **void** OutputAddResult(**int** value1, **int** value2)
8. {
9. AddHandler(value1, value2);
10. }
11. }
13. **public** **class** ClientTest
14. {
15. **private** EventTest m\_EventTest;
17. **public** ClientTest()
18. {
19. m\_EventTest = **new** EventTest();
20. m\_EventTest.AddHandler += **new** EventTest.Add(m\_EventTest\_AddHandler);
21. m\_EventTest.AddDelegate = AddDelegate;
22. //the line below will cause compile error.
23. //m\_EventTest.AddHandler(1, 1);
24. m\_EventTest.AddDelegate(1, 1);
25. }
27. **private** **int** m\_EventTest\_AddHandler(**int** value1, **int** value2)
28. {
29. **return** value1 + value2;
30. }
32. **private** **int** AddDelegate(**int** value1, **int** value2)
33. {
34. **return** value1 + value2;
35. }
36. }

    上述代码也说明对于委托，你不但可以安排谁是它的调用函数，还可以直接调用它；而对于事件，你是不能直接调用的，只能通过某些操作触发。

    .NET针对Event类型的变量，定义了add和remove两个访问器，类似于普通属性中的get和set，通过add和remove，我们可以使用“+=” 或者 “-=”来注册事件或者解除事件。关于add和remove，是由编译器自动为我们生成的，在实际编写代码时，我们应该声明共有事件，然后让编译器来为我们创建add和remove访问器。

    在定义事件或者事件所在的类型中，是不需要知道潜在的客户调用方的信息的，即事件是只能够在服务器端调用，在客户端进行注册实现，但是服务器端是无需知道客户端的信息的，这两者是松耦合的。这里所说的服务器端和客户端，分别表示声明事件的类型和注册事件的类型。

    当我们的类型包含的事件比较多时，仍然采取为每一个事件定义个一个字段的方式，就变的不可取了，这时，我们可以定义一个事件的容器，在运行时，动态的创建事件对象。其中.NET框架内核在Windows控件子系统中包含有这方面的做法示例。

    我们可以查看下面的代码，使用了容器的方式来保存事件的具体信息。

**[csharp]** [view plain](http://blog.csdn.net/yin554393109/article/details/38108645" \o "view plain) [copy](http://blog.csdn.net/yin554393109/article/details/38108645" \o "copy" \t "http://blog.csdn.net/yin554393109/article/details/_blank) [IMG_258](https://code.csdn.net/snippets/434912" \o "在CODE上查看代码片" \t "http://blog.csdn.net/yin554393109/article/details/_blank)[IMG_259](https://code.csdn.net/snippets/434912/fork" \o "派生到我的代码片" \t "http://blog.csdn.net/yin554393109/article/details/_blank)

1. **public** **class** Logger
2. {
3. **private** **static** System.ComponentModel.EventHandlerList
4. Handlers = **new** System.ComponentModel.EventHandlerList();
6. **static** **public** **void** AddLogger(
7. **string** system, AddMessageEventHandler ev )
8. {
9. Handlers[ system ] = ev;
10. }
12. **static** **public** **void** RemoveLogger( **string** system )
13. {
14. Handlers[ system ] = **null**;
15. }
17. **static** **public** **void** AddMsg ( **string** system, **int** priority,  **string** msg )
18. {
19. **if** ( ( system != **null** ) && ( system.Length > 0 ) )
20. {
21. AddMessageEventHandler l =
22. Handlers[ system ] **as** AddMessageEventHandler;
24. LoggerEventArgs args = **new** LoggerEventArgs(
25. priority, msg );
26. **if** ( l != **null** )
27. l ( **null**, args );
29. // The empty string means receive all messages:
30. l = Handlers[ "" ] **as** AddMessageEventHandler;
31. **if** ( l != **null** )
32. l( **null**, args );
33. }
34. }
35. }

    上述代码会在EventHandlerList集合中存储各个事件处理器，当客户代码关联到一个特定的子系统（或者说Key值）时，新的事件对象就会被创建。对于同一个Key值，其后的请求会返回相同的事件对象，因为容器是一个静态容器。如果我们的类型在其接口中包含有大量的时间，那么我们就应该采用这种事件容易的方式，当客户代码真正关联事件处理器时，我们才会创建事件成员。

    总结：我们使用事件来定义类型中的外发接口，任意数量的客户对象都可以将自己的处理器注册到事件上，然后处理它们，这些客户对象不需要在编译时存在，事件也不必非有订阅者才可以正常工作。在C#中使用事件可以对发送者和可能的通知接收者进行解耦，发送者完全可以独立于接收者进行开发。