**POCO C++库学习和分析 -- 通知和事件 （四）**

**5. 事件**

        Poco中的事件和代理概念来自于C#。对于事件的使用者，也就是调用方来说，用法非常的简单。

**5.1 从例子说起**

        首先让我们来看一个同步事件例子，然后再继续我们的讨论：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. #include "Poco/BasicEvent.h"
2. #include "Poco/Delegate.h"
3. #include <iostream>
5. **using** Poco::BasicEvent;
6. **using** Poco::Delegate;
8. **class** Source
9. {
10. **public**:
11. BasicEvent<**int**> theEvent;
12. **void** fireEvent(**int** n)
13. {
14. theEvent(**this**, n);
15. // theEvent.notify(this, n); // alternative syntax
16. }
17. };
19. **class** Target
20. {
21. **public**:
22. **void** onEvent(**const** **void**\* pSender, **int**& arg)
23. {
24. std::cout << "onEvent: " << arg << std::endl;
25. }
26. };
28. **int** main(**int** argc, **char**\*\* argv)
29. {
30. Source source;
31. Target target;
32. source.theEvent += Poco::delegate(&target, &Target::onEvent);
33. source.fireEvent(42);
34. source.theEvent -= Poco::delegate(&target, &Target::onEvent);
36. **return** 0;
37. }

        从上面的代码里，我们可以清晰的看到几个部分，数据源Source，事件BasicEvent<T>，目标对象Target。

        其中source.theEvent += Poco::delegate(&target,  &Target::onEvent)完成了，目标向数据源事件注册的过程。大家都知道在C++中，程序运行是落实到类的实例的，看一下消息传递的过程，Poco是如何解决这个问题。target是目标对象实例，Target::onEvent目标对象处理事件的函数入口地址。source.fireEvent(42)触发事件运行，其定义为:

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. **void** fireEvent(**int** n)
2. {
3. theEvent(**this**, n);
4. // theEvent.notify(this, n); // alternative syntax
5. }

        theEvent(this, n)中存在两个参数，其中n为Target::onEvent(const void\* pSender, int& arg)处理函数的参数，可理解为消息或者事件内容；this给出了触发源实例的信息。  
ok。这样消息的传递流程出来了。消息源实例的地址，消息内容，目标实例地址，目标实例类的处理函数入口地址。使用者填入上述信息就可以传递消息了。相当简单。  
  
        而对于事件的开发者，如何实现上述功能。这是另外一码事，用C++实现这么一个功能还是挺复杂的一件事。看一下使用语言的方式，想一下用到的C++技术：  
        1. +=/-= 重载

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. source.theEvent += Poco::delegate(&target, &Target::onEvent);

        2. 仿函式

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

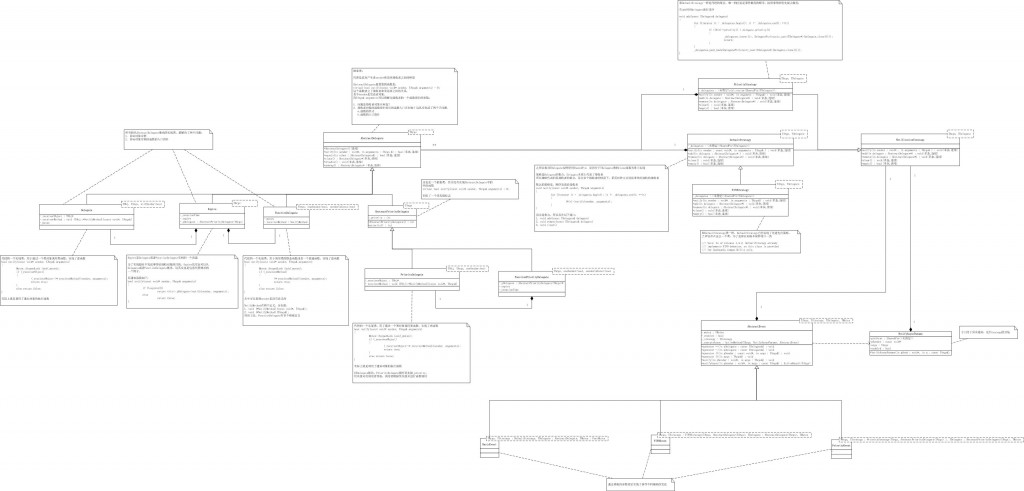
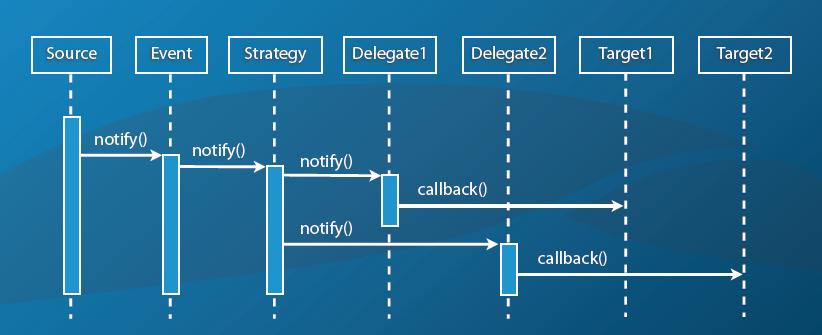
1. theEvent(**this**, n);

        3. 模板  
        开发者是不应该限定使用者发送消息的类以及接受消息类的类型的，因此C++中能够完成此功能的技术只有模板了。关于模板编程还想聊上几句。STL的特点在于算法和数据结构的分离，这个其实也是泛型编程的特点。如果把使用者对于类的应用过程看做算法过程的话，就可以对这个过程进行泛型编程。同时应该注意的是，算法和数据结构是存在关联的，这是隐含在泛型编程中的，能够使用某种算法的数据结构一定是符合该种算法要求的。  
        就拿Poco中事件的委托Delegate来说，目标对象处理事件的函数入口是存在某种假设的。Poco中假设入口函数必须是如下形式之一：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. **void** (TObj::\*NotifyMethod)(**const** **void**\*, TArgs&);
2. **void** (TObj::\*NotifyMethod)(TArgs&);
3. **void** (\*NotifyMethod)(**const** **void**\*, TArgs&);
4. **void** (\*NotifyMethod)(**void**\*, TArgs&);

**5.2 事件的实现**

        下面一张图是Poco中Event的类图：  
  
  
        下面另一张图是Poco中Event流动的过程：  
  
  
  
  
        从图上看实现事件的类被分成了几类：  
        1) Delegate：

             AbstractDelegate,Delegate,Expire,FunctionDelegate,AbstractPriorityDelegate,PriorityDelegate,FunctionPriorityDelegate：  
        2) Strategy：  
             NotificationStrategy，PriorityStrategy，DefaultStrategy，FIFOStrategy  
        3) Event：  
            AbstractEvent，PriorityEvent，FIFOEvent，BasicEvent  
  
        我们取Delegate，DefaultStrategy，BasicEvent来分析，其他的只是在它们的基础上加了一些修饰，流程类似。  
  
        Delegate类定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. **template** <**class** TObj, **class** TArgs, **bool** withSender = **true**>
2. **class** Delegate: **public** AbstractDelegate<TArgs>
3. {
4. **public**:
5. **typedef** **void** (TObj::\*NotifyMethod)(**const** **void**\*, TArgs&);
7. Delegate(TObj\* obj, NotifyMethod method):
8. \_receiverObject(obj),
9. \_receiverMethod(method)
10. {
11. }
13. Delegate(**const** Delegate& delegate):
14. AbstractDelegate<TArgs>(delegate),
15. \_receiverObject(delegate.\_receiverObject),
16. \_receiverMethod(delegate.\_receiverMethod)
17. {
18. }
20. ~Delegate()
21. {
22. }
24. Delegate& operator = (**const** Delegate& delegate)
25. {
26. **if** (&delegate != **this**)
27. {
28. **this**->\_receiverObject = delegate.\_receiverObject;
29. **this**->\_receiverMethod = delegate.\_receiverMethod;
30. }
31. **return** \***this**;
32. }
34. **bool** notify(**const** **void**\* sender, TArgs& arguments)
35. {
36. Mutex::ScopedLock lock(\_mutex);
37. **if** (\_receiverObject)
38. {
39. (\_receiverObject->\*\_receiverMethod)(sender, arguments);
40. **return** **true**;
41. }
42. **else** **return** **false**;
43. }
45. **bool** equals(**const** AbstractDelegate<TArgs>& other) **const**
46. {
47. **const** Delegate\* pOtherDelegate = **reinterpret\_cast**<**const** Delegate\*>(other.unwrap());
48. **return** pOtherDelegate && \_receiverObject == pOtherDelegate->\_receiverObject && \_receiverMethod == pOtherDelegate->\_receiverMethod;
49. }
51. AbstractDelegate<TArgs>\* clone() **const**
52. {
53. **return** **new** Delegate(\***this**);
54. }
56. **void** disable()
57. {
58. Mutex::ScopedLock lock(\_mutex);
59. \_receiverObject = 0;
60. }
62. **protected**:
63. TObj\*        \_receiverObject;
64. NotifyMethod \_receiverMethod;
65. Mutex        \_mutex;

68. **private**:
69. Delegate();
70. };

        我们可以看到Delegate类中存储了目标类实例的指针\_receiverObject，同时存储了目标类处理函数的入口地址\_receiverMethod，当初始化Delegate实例时，参数被带进。  
Delegate类中处理事件的函数为bool notify(const void\* sender, TArgs& arguments)，这是一个虚函数. 如果去看它的实现的话，它最终调用了目标类处理函数

(\_receiverObject->\*\_receiverMethod)(sender, arguments)。如果用简单的话来描述Delegate的作用，那就是目标类的代理。

        在Poco中对于Delegate提供了模板函数delegate，来隐藏Delegate对象的创建，其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. **template** <**class** TObj, **class** TArgs>
2. **static** Delegate<TObj, TArgs, **true**> delegate(TObj\* pObj, **void** (TObj::\*NotifyMethod)(**const** **void**\*, TArgs&))
3. {
4. **return** Delegate<TObj, TArgs, **true**>(pObj, NotifyMethod);
5. }

        在来看DefaultStrategy类，其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. **template** <**class** TArgs, **class** TDelegate>
2. **class** DefaultStrategy: **public** NotificationStrategy<TArgs, TDelegate>
3. /// Default notification strategy.
4. ///
5. /// Internally, a std::vector<> is used to store
6. /// delegate objects. Delegates are invoked in the
7. /// order in which they have been registered.
8. {
9. **public**:
10. **typedef** SharedPtr<TDelegate>         DelegatePtr;
11. **typedef** std::vector<DelegatePtr>     Delegates;
12. **typedef** **typename** Delegates::iterator Iterator;
14. **public**:
15. DefaultStrategy()
16. {
17. }
19. DefaultStrategy(**const** DefaultStrategy& s):
20. \_delegates(s.\_delegates)
21. {
22. }
24. ~DefaultStrategy()
25. {
26. }
28. **void** notify(**const** **void**\* sender, TArgs& arguments)
29. {
30. **for** (Iterator it = \_delegates.begin(); it != \_delegates.end(); ++it)
31. {
32. (\*it)->notify(sender, arguments);
33. }
34. }
36. **void** add(**const** TDelegate& delegate)
37. {
38. \_delegates.push\_back(DelegatePtr(**static\_cast**<TDelegate\*>(delegate.clone())));
39. }
41. **void** remove(**const** TDelegate& delegate)
42. {
43. **for** (Iterator it = \_delegates.begin(); it != \_delegates.end(); ++it)
44. {
45. **if** (delegate.equals(\*\*it))
46. {
47. (\*it)->disable();
48. \_delegates.erase(it);
49. **return**;
50. }
51. }
52. }
54. DefaultStrategy& operator = (**const** DefaultStrategy& s)
55. {
56. **if** (**this** != &s)
57. {
58. \_delegates = s.\_delegates;
59. }
60. **return** \***this**;
61. }
63. **void** clear()
64. {
65. **for** (Iterator it = \_delegates.begin(); it != \_delegates.end(); ++it)
66. {
67. (\*it)->disable();
68. }
69. \_delegates.clear();
70. }
72. **bool** empty() **const**
73. {
74. **return** \_delegates.empty();
75. }
77. **protected**:
78. Delegates \_delegates;
79. };

        哦，明白了，DefaultStrategy是一组委托的集合，内部存在的\_delegates定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. std::vector<SharedPtr<TDelegate>>  \_delegate

        DefaultStrategy可以被理解成一组目标的代理。在DefaultStrategy的notify函数中，我们可以设定，当一个事件发生，要送给多个目标时，所采取的策略。NotificationStrategy，PriorityStrategy，DefaultStrategy，FIFOStrategy之间的区别也就在于此。  
  
        最后来看一下BasicEvent类。它的定义是：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. **template** <**class** TArgs, **class** TMutex = FastMutex>
2. **class** BasicEvent: **public** AbstractEvent <
3. TArgs, DefaultStrategy<TArgs, AbstractDelegate<TArgs> >,
4. AbstractDelegate<TArgs>,
5. TMutex
6. >
7. /// A BasicEvent uses the DefaultStrategy which
8. /// invokes delegates in the order they have been registered.
9. ///
10. /// Please see the AbstractEvent class template documentation
11. /// for more information.
12. {
13. **public**:
14. BasicEvent()
15. {
16. }
18. ~BasicEvent()
19. {
20. }
22. **private**:
23. BasicEvent(**const** BasicEvent& e);
24. BasicEvent& operator = (**const** BasicEvent& e);
25. };
27. AbstractEvent定义为：
28. **template** <**class** TArgs, **class** TStrategy, **class** TDelegate, **class** TMutex = FastMutex>
29. **class** AbstractEvent
30. {
31. **public**:
32. AbstractEvent():
33. \_executeAsync(**this**, &AbstractEvent::executeAsyncImpl),
34. \_enabled(**true**)
35. {
36. }
38. AbstractEvent(**const** TStrategy& strat):
39. \_executeAsync(**this**, &AbstractEvent::executeAsyncImpl),
40. \_strategy(strat),
41. \_enabled(**true**)
42. {
43. }
45. **virtual** ~AbstractEvent()
46. {
47. }
49. **void** operator += (**const** TDelegate& aDelegate)
50. {
51. **typename** TMutex::ScopedLock lock(\_mutex);
52. \_strategy.add(aDelegate);
53. }
55. **void** operator -= (**const** TDelegate& aDelegate)
56. {
57. **typename** TMutex::ScopedLock lock(\_mutex);
58. \_strategy.remove(aDelegate);
59. }
61. **void** operator () (**const** **void**\* pSender, TArgs& args)
62. {
63. notify(pSender, args);
64. }
66. **void** operator () (TArgs& args)
67. {
68. notify(0, args);
69. }
71. **void** notify(**const** **void**\* pSender, TArgs& args)
72. {
73. Poco::ScopedLockWithUnlock<TMutex> lock(\_mutex);
75. **if** (!\_enabled) **return**;
77. TStrategy strategy(\_strategy);
78. lock.unlock();
79. strategy.notify(pSender, args);
80. }
82. ActiveResult<TArgs> notifyAsync(**const** **void**\* pSender, **const** TArgs& args)
83. {
84. NotifyAsyncParams params(pSender, args);
85. {
86. **typename** TMutex::ScopedLock lock(\_mutex);
88. params.ptrStrat = SharedPtr<TStrategy>(**new** TStrategy(\_strategy));
89. params.enabled  = \_enabled;
90. }
91. ActiveResult<TArgs> result = \_executeAsync(params);
92. **return** result;
93. }
94. // .......
96. **protected**:
97. **struct** NotifyAsyncParams
98. {
99. SharedPtr<TStrategy> ptrStrat;
100. **const** **void**\* pSender;
101. TArgs       args;
102. **bool**        enabled;
104. NotifyAsyncParams(**const** **void**\* pSend, **const** TArgs& a):ptrStrat(), pSender(pSend), args(a), enabled(**true**)
105. {
106. }
107. };
109. ActiveMethod<TArgs, NotifyAsyncParams, AbstractEvent> \_executeAsync;
111. TArgs executeAsyncImpl(**const** NotifyAsyncParams& par)
112. {
113. **if** (!par.enabled)
114. {
115. **return** par.args;
116. }

119. NotifyAsyncParams params = par;
120. TArgs retArgs(params.args);
121. params.ptrStrat->notify(params.pSender, retArgs);
122. **return** retArgs;
123. }
125. TStrategy \_strategy; /// The strategy used to notify observers.
126. **bool**      \_enabled;  /// Stores if an event is enabled. Notfies on disabled events have no effect
127. /// but it is possible to change the observers.
128. **mutable** TMutex \_mutex;
130. **private**:
131. AbstractEvent(**const** AbstractEvent& other);
132. AbstractEvent& operator = (**const** AbstractEvent& other);
133. };

        从AbstractEvent类中，我们看到AbstractEvent类中存在了一个TStrategy的对象\_strategy。接口上则重载了+=函数，用来把所需的目标对象加入\_strategy中，完成注册功能。重载了operator ()，用于触发事件。  
        于是同步事件所有的步骤便被串了起来。

**5.2 异步事件**

        理解了同步事件后，让我们来看异步事件。这还是让我们从一个例子说起：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673557)

1. #include "Poco/BasicEvent.h"
2. #include "Poco/Delegate.h"
3. #include "Poco/ActiveResult.h"
4. #include <iostream>
6. **using** Poco::BasicEvent;
7. **using** Poco::Delegate;
8. **using** Poco::ActiveResult;
10. **class** TargetAsync
11. {
12. **public**:
13. **void** onAsyncEvent(**const** **void**\* pSender, **int**& arg)
14. {
15. std::cout << "onAsyncEvent: " << arg <<  " Current Thread Id is :" << GetCurrentThreadId() << " "<< std::endl;
16. **return**;
17. }
18. };
20. **template**<**typename** RT> **class** Source
21. {
22. **public**:
23. BasicEvent<**int**> theEvent;
24. ActiveResult<RT> AsyncFireEvent(**int** n)
25. {
26. **return** ActiveResult<RT> (theEvent.notifyAsync(**this**, n));
27. }
28. };
30. **int** main(**int** argc, **char**\*\* argv)
31. {
32. Source<**int**> source;
33. TargetAsync target;
34. std::cout <<  "Main Thread Id is :" << GetCurrentThreadId() << " " << std::endl;
35. source.theEvent += Poco::delegate(&target, &TargetAsync::onAsyncEvent);
36. ActiveResult<**int**> Targs = source.AsyncFireEvent(43);
37. Targs.wait();
38. std::cout << "onEventAsync: " << Targs.data() << std::endl;
39. source.theEvent -= Poco::delegate(&target, &TargetAsync::onAsyncEvent);
41. **return** 0;
42. }

        例子里可以看出，同同步事件不同的是，触发事件时，我们调用的是notifyAsync接口。在这个接口里，NotifyAsyncParams对象被创建，并被交由一个主动对象\_executeAsync执行。关于主动对象ActiveMethod的介绍，可以从前面的文章[POCO C++库学习和分析 -- 线程 （四）](http://blog.csdn.net/arau_sh/article/details/8618571)中找到。  
  
  
（版权所有，转载时请注明作者和出处  http://blog.csdn.net/arau\_sh/article/details/8673557）