**POCO C++库学习和分析 -- 通知和事件 （三）**

**4. 异步通知**

**4.1 NotificationQueue类**

         Poco中的异步通知是通过NotificationQueue类来实现的，同它功能类似还有类PriorityNotificationQueue和TimedNotificationQueue。不同的是PriorityNotificationQueue类中对消息分了优先级，对优先级高的消息优先处理；而TimedNotificationQueue对消息给了时间戳，时间戳早的优先处理，而和其压入队列的时间无关。所以接下来我们主要关注NotificationQueue的实现。  
        事实上NotificationQueue是个非常有趣的类。让我们来看一下它的头文件：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673543)

1. **class** Foundation\_API NotificationQueue
2. /// A NotificationQueue object provides a way to implement asynchronous
3. /// notifications. This is especially useful for sending notifications
4. /// from one thread to another, for example from a background thread to
5. /// the main (user interface) thread.
6. ///
7. /// The NotificationQueue can also be used to distribute work from
8. /// a controlling thread to one or more worker threads. Each worker thread
9. /// repeatedly calls waitDequeueNotification() and processes the
10. /// returned notification. Special care must be taken when shutting
11. /// down a queue with worker threads waiting for notifications.
12. /// The recommended sequence to shut down and destroy the queue is to
13. ///   1. set a termination flag for every worker thread
14. ///   2. call the wakeUpAll() method
15. ///   3. join each worker thread
16. ///   4. destroy the notification queue.
17. {
18. **public**:
19. NotificationQueue();
20. /// Creates the NotificationQueue.
22. ~NotificationQueue();
23. /// Destroys the NotificationQueue.
25. **void** enqueueNotification(Notification::Ptr pNotification);
26. /// Enqueues the given notification by adding it to
27. /// the end of the queue (FIFO).
28. /// The queue takes ownership of the notification, thus
29. /// a call like
30. ///     notificationQueue.enqueueNotification(new MyNotification);
31. /// does not result in a memory leak.
33. **void** enqueueUrgentNotification(Notification::Ptr pNotification);
34. /// Enqueues the given notification by adding it to
35. /// the front of the queue (LIFO). The event therefore gets processed
36. /// before all other events already in the queue.
37. /// The queue takes ownership of the notification, thus
38. /// a call like
39. ///     notificationQueue.enqueueUrgentNotification(new MyNotification);
40. /// does not result in a memory leak.
42. Notification\* dequeueNotification();
43. /// Dequeues the next pending notification.
44. /// Returns 0 (null) if no notification is available.
45. /// The caller gains ownership of the notification and
46. /// is expected to release it when done with it.
47. ///
48. /// It is highly recommended that the result is immediately
49. /// assigned to a Notification::Ptr, to avoid potential
50. /// memory management issues.
52. Notification\* waitDequeueNotification();
53. /// Dequeues the next pending notification.
54. /// If no notification is available, waits for a notification
55. /// to be enqueued.
56. /// The caller gains ownership of the notification and
57. /// is expected to release it when done with it.
58. /// This method returns 0 (null) if wakeUpWaitingThreads()
59. /// has been called by another thread.
60. ///
61. /// It is highly recommended that the result is immediately
62. /// assigned to a Notification::Ptr, to avoid potential
63. /// memory management issues.
65. Notification\* waitDequeueNotification(**long** milliseconds);
66. /// Dequeues the next pending notification.
67. /// If no notification is available, waits for a notification
68. /// to be enqueued up to the specified time.
69. /// Returns 0 (null) if no notification is available.
70. /// The caller gains ownership of the notification and
71. /// is expected to release it when done with it.
72. ///
73. /// It is highly recommended that the result is immediately
74. /// assigned to a Notification::Ptr, to avoid potential
75. /// memory management issues.
77. **void** dispatch(NotificationCenter& notificationCenter);
78. /// Dispatches all queued notifications to the given
79. /// notification center.
81. **void** wakeUpAll();
82. /// Wakes up all threads that wait for a notification.
84. **bool** empty() **const**;
85. /// Returns true iff the queue is empty.
87. **int** size() **const**;
88. /// Returns the number of notifications in the queue.
90. **void** clear();
91. /// Removes all notifications from the queue.
93. **bool** hasIdleThreads() **const**;
94. /// Returns true if the queue has at least one thread waiting
95. /// for a notification.
97. **static** NotificationQueue& defaultQueue();
98. /// Returns a reference to the default
99. /// NotificationQueue.
101. **protected**:
102. Notification::Ptr dequeueOne();
104. **private**:
105. **typedef** std::deque<Notification::Ptr> NfQueue;
106. **struct** WaitInfo
107. {
108. Notification::Ptr pNf;
109. Event             nfAvailable;
110. };
111. **typedef** std::deque<WaitInfo\*> WaitQueue;
113. NfQueue           \_nfQueue;
114. WaitQueue         \_waitQueue;
115. **mutable** FastMutex \_mutex;
116. };

        从定义可以看到NotificationQueue类管理了两个deque容器。其中一个是WaitInfo对象的deque，另一个是Notification对象的deque。而WaitInfo一对一的对应了一个消息对象pNf和事件对象nfAvailable，毫无疑问Event对象是用来同步多线程的。让我们来看看它如何实现。  
        NotificationQueue实现的巧妙之处就在于WaitInfo由消费者动态创建，消费者线程通过函数Notification\* waitDequeueNotification()获取消息，其函数定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673543)

1. Notification\* NotificationQueue::waitDequeueNotification()
2. {
3. Notification::Ptr pNf;
4. WaitInfo\* pWI = 0;
5. {
6. FastMutex::ScopedLock lock(\_mutex);
7. pNf = dequeueOne();
8. **if** (pNf) **return** pNf.duplicate();
9. pWI = **new** WaitInfo;
10. \_waitQueue.push\_back(pWI);
11. }
12. pWI->nfAvailable.wait();
13. pNf = pWI->pNf;
14. **delete** pWI;
15. **return** pNf.duplicate();
16. }
18. Notification::Ptr NotificationQueue::dequeueOne()
19. {
20. Notification::Ptr pNf;
21. **if** (!\_nfQueue.empty())
22. {
23. pNf = \_nfQueue.front();
24. \_nfQueue.pop\_front();
25. }
26. **return** pNf;
27. }

        消费者线程首先从Notification对象的deque中获取消息，如果消息获取不为空，则直接返回处理，如果消息为空，则创建一个新的WaitInfo对象，并压入WaitInfo对象的  
deque。 消费者线程开始等待，直到生产者通知有消息的存在，然后再从WaitInfo对象中取出消息，返回处理。当消费者线程能从Notification对象的deque中获取到消息时，说明消费者处理消息的速度要比生成者低；反之则说明消费者处理消息的速度要比生成者高。  
  
        让我们再看一下生产者的调用函数void NotificationQueue::enqueueNotification(Notification::Ptr pNotification)，其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673543)

1. **void** NotificationQueue::enqueueNotification(Notification::Ptr pNotification)
2. {
3. poco\_check\_ptr (pNotification);
4. FastMutex::ScopedLock lock(\_mutex);
5. **if** (\_waitQueue.empty())
6. {
7. \_nfQueue.push\_back(pNotification);
8. }
9. **else**
10. {
11. WaitInfo\* pWI = \_waitQueue.front();
12. \_waitQueue.pop\_front();
13. pWI->pNf = pNotification;
14. pWI->nfAvailable.set();
15. }
16. }

        生产者线程首先判断WaitInfo对象的deque是否为空，如果不为空，说明存在消费者线程等待，则从deque中获取一个WaitInfo对象，灌入Notification消息，释放信号量激活消费者线程；而如果为空，说明目前说有的消费者线程都在工作，则把消息暂时存入Notification对象的deque，等待消费者线程有空时处理。  
        整个处理过程中对于\_mutex对象的处理是非常小心的，\_waitQueue不被使用则释放。OK，整个流程结束，消息源和目标被解耦。

**4.2 一个异步通知的例子**

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8673543)

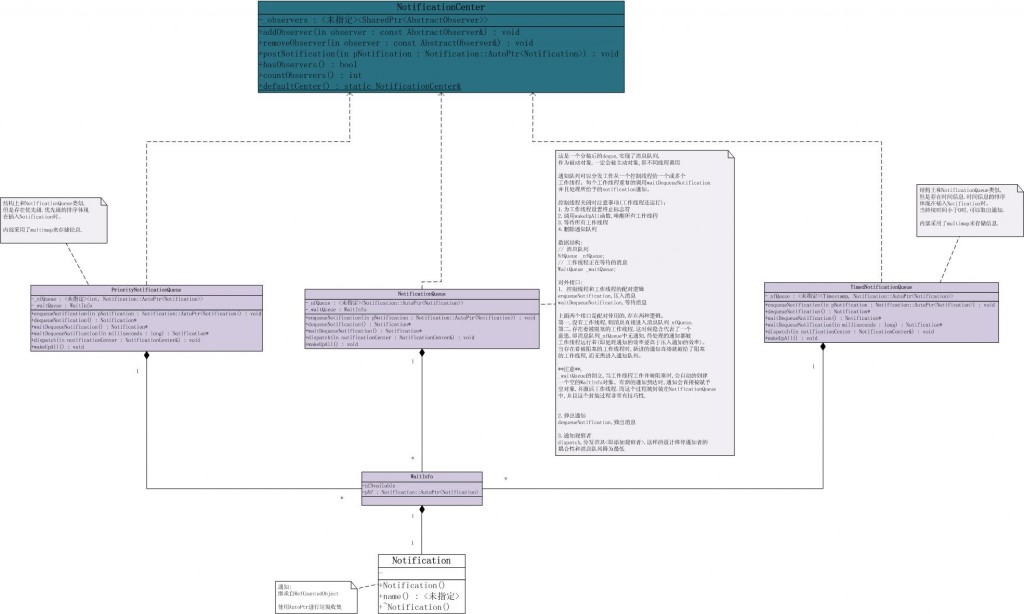
1. #include "Poco/Notification.h"
2. #include "Poco/NotificationQueue.h"
3. #include "Poco/ThreadPool.h"
4. #include "Poco/Runnable.h"
5. #include "Poco/AutoPtr.h"
6. **using** Poco::Notification;
7. **using** Poco::NotificationQueue;
8. **using** Poco::ThreadPool;
9. **using** Poco::Runnable;
10. **using** Poco::AutoPtr;
11. **class** WorkNotification: **public** Notification
12. {
13. **public**:
14. WorkNotification(**int** data): \_data(data) {}
15. **int** data() **const**
16. {
17. **return** \_data;
18. }
19. **private**:
20. **int** \_data;
21. };

24. **class** Worker: **public** Runnable
25. {
26. **public**:
27. Worker(NotificationQueue& queue): \_queue(queue) {}
28. **void** run()
29. {
30. AutoPtr<Notification> pNf(\_queue.waitDequeueNotification());
31. **while** (pNf)
32. {
33. WorkNotification\* pWorkNf =
34. **dynamic\_cast**<WorkNotification\*>(pNf.get());
35. **if** (pWorkNf)
36. {
37. // do some work
38. }
39. pNf = \_queue.waitDequeueNotification();
40. }
41. }
42. **private**:
43. NotificationQueue& \_queue;
44. };

47. **int** main(**int** argc, **char**\*\* argv)
48. {
49. NotificationQueue queue;
50. Worker worker1(queue); // create worker threads
51. Worker worker2(queue);
52. ThreadPool::defaultPool().start(worker1); // start workers
53. ThreadPool::defaultPool().start(worker2);
54. // create some work
55. **for** (**int** i = 0; i < 100; ++i)
56. {
57. queue.enqueueNotification(**new** WorkNotification(i));
58. }
59. **while** (!queue.empty()) // wait until all work is done
60. Poco::Thread::sleep(100);
61. queue.wakeUpAll(); // tell workers they're done
62. ThreadPool::defaultPool().joinAll();
63. **return** 0;
64. }

**4.3 异步通知的类图**

        最后给出异步通知的类图:



（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8673543>）