**POCO C++库学习和分析 -- 通知和事件 ( 二 )**

**2. 通知和事件的总览**

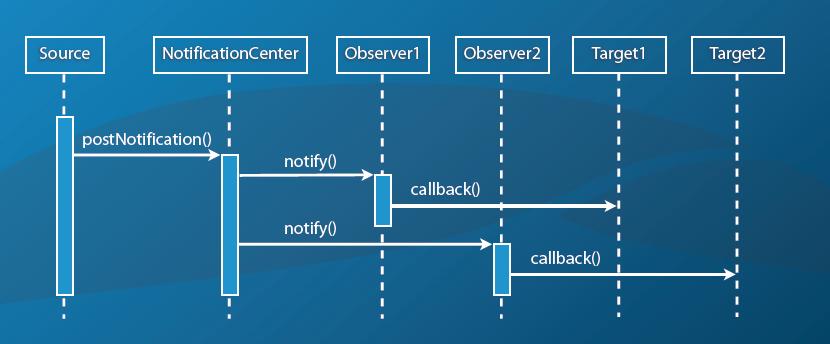
**2.1 相关类信息**

        下面是Poco库和通知、事件相关的类  
        1)  同步通知实现：类Notification和NotificationCenter  
        2)  异步通知实现：类Notification和NotificationQueue  
        3)  事件 Events

**2.2 概述**

        Poco文档上对于通知和事件的区别做了如下描述：  
        1)  通知和事件是Poco库中支持的两种消息通知机制，目的是为了在源对象(source)发生某事件(something)时能够及时通知目的对象(target)  
        2)  使用Poco中的通知，必须注意通知对象(target)也可称观察者(observer)将无法得知事件源的情况。Poco::NotificationCenter和Poco::NotificationQueue是消息传递的中间载体，用来对源(source)和目标(target)进行解耦。  
        3) 如果对象(target)或者说观察者(observer)期望知道事件源的情况，或者想只从某一个确切的源接收事件，可以使用Poco::Event。Poco中的Event同时支持异步和同步消息。  
        看了上面的文档，千万不要以为通知无法获取源对象的信息。事实上，通过对代码的改写，我们也可以使通知支持上述功能。只不过通知是基于消息源角度的设计，在设计时，就认为对于通知者，关注重点并不在消息源，而在消息类型。关于这一点，可以看前面一篇文章[POCO C++库学习和分析 -- 通知和事件（一）](http://blog.csdn.net/arau_sh/article/details/8664372)。

        下图说明了同步消息时，消息发送的流程：



**3. 同步通知**

**3.1 消息**

        所有的通知类都继承自Poco::Notification，其定义如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. class Notification: public RefCountedObject
2. {
3. public:
4. typedef AutoPtr<Notification> Ptr;
5. Notification();
6. virtual std::string name() const;

9. protected:
10. virtual ~Notification();
11. };

class Notification: public RefCountedObject

{

public:

typedef AutoPtr<Notification> Ptr;

Notification();

virtual std::string name() const;

protected:

virtual ~Notification();

};

        从定义看，我们可以发现其从RefCountedObject类继承，也就是说其是一个引用计数对象。作为从RefCountedObject中继承的引用计数对象，毫无疑问在其在Poco中使用是和AutoPtr类配合的，完成堆对象的自动回收。关于AutoPtr的介绍，可以看前面的文章[POCO C++库学习和分析 -- 内存管理(一)](http://blog.csdn.net/arau_sh/article/details/8631621)。

        使用时，我们可以从Notification继承，以便实现自己的通知，并且在通知类中可以放置我们想的任意数据，但是Poco中的Notification继承类不支持值语义操作，也就是说不支持拷贝构造函数(copy constructor)和赋值构造函数(assignment)。要注意的是这个限制并不是由于Notification继承类本身的限制导致的不支持，我们完全可以为其实现拷贝构造函数和赋值构造函数。这个限制是使用继承类的时候，为了实现动态对象的自动回收，消息的中介Poco::NotificationCenter和接收者Observer都使用了Poco::AutoPtr去传递和接收数据造成的。所以所有的Notification继承类对象都在堆上分配，运用时没有必要为其提供拷贝构造函数和赋值构造函数。

**3.2 消息的发送者 source**

        类NotificationCenter类扮演了一个消息源的角色。下面是它的定义：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. class NotificationCenter
2. {
3. public:
4. NotificationCenter();
5. ~NotificationCenter();
7. void addObserver(const AbstractObserver& observer);
8. void removeObserver(const AbstractObserver& observer);
10. void postNotification(Notification::Ptr pNotification);
12. **bool** hasObservers() const;
13. std::**size\_t** countObservers() const;
15. static NotificationCenter& defaultCenter();
17. private:
18. typedef SharedPtr<AbstractObserver> AbstractObserverPtr;
19. typedef std::vector<AbstractObserverPtr> ObserverList;
21. ObserverList  \_observers;
22. mutable Mutex \_mutex;
23. };

class NotificationCenter

{

public:

NotificationCenter();

~NotificationCenter();

void addObserver(const AbstractObserver& observer);

void removeObserver(const AbstractObserver& observer);

void postNotification(Notification::Ptr pNotification);

bool hasObservers() const;

std::size\_t countObservers() const;

static NotificationCenter& defaultCenter();

private:

typedef SharedPtr<AbstractObserver> AbstractObserverPtr;

typedef std::vector<AbstractObserverPtr> ObserverList;

ObserverList \_observers;

mutable Mutex \_mutex;

};

        从定义可以看出它是一个目标对象的集合std::vector<SharedPtr<AbstractObserver>>\_observers。  
        通过调用函数addObserver(const AbstractObserver& observer)，可以完成目标对象的注册过程。调用函数removeObserver()则可以完成反注册。而函数postNotification是一个消息传递的过程，其定义如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. void NotificationCenter::postNotification(Notification::Ptr pNotification)
2. {
3. poco\_check\_ptr (pNotification);

6. ScopedLockWithUnlock<Mutex> lock(\_mutex);
7. ObserverList observersToNotify(\_observers);
8. lock.unlock();
9. for (ObserverList::iterator it = observersToNotify.begin(); it !=

12. observersToNotify.end(); ++it)
13. {
14. (\*it)->notify(pNotification);
15. }
16. }

void NotificationCenter::postNotification(Notification::Ptr pNotification)

{

poco\_check\_ptr (pNotification);

ScopedLockWithUnlock<Mutex> lock(\_mutex);

ObserverList observersToNotify(\_observers);

lock.unlock();

for (ObserverList::iterator it = observersToNotify.begin(); it !=

observersToNotify.end(); ++it)

{

(\*it)->notify(pNotification);

}

}

        从它的实现看，只是简单遍历\_observers对象，并最终通过AbstractObserver->notify()把消息发送给通知对象。同时为了避免长时间占用\_observers对象，在发送消息时，复制了一份。

        当使用者调用postNotification函数时，毫无疑问，消息被触发。

**3.3 消息的接收者 target**

        消息产生后，最终都要求被发送给合适的处理者。在C++中，处理者一定是一个对象，而处理即意味着行为，在C++中意味着类的成员函数，也就是说最终的处理要落实到类的对象实例的函数指针上。在Poco中，AbstractObserver可以理解成对象类的一个代理，它是一个纯虚类，定义了接收对象的接口。它的定义如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. class AbstractObserver
2. {
3. public:
4. AbstractObserver();
5. AbstractObserver(const AbstractObserver& observer);
6. virtual ~AbstractObserver();
8. AbstractObserver& operator = (const AbstractObserver& observer);

11. virtual void notify(Notification\* pNf) const = 0;
12. virtual **bool** equals(const AbstractObserver& observer) const = 0;
13. virtual **bool** accepts(Notification\* pNf) const = 0;
14. virtual AbstractObserver\* clone() const = 0;
15. virtual void disable() = 0;
16. };

class AbstractObserver

{

public:

AbstractObserver();

AbstractObserver(const AbstractObserver& observer);

virtual ~AbstractObserver();

AbstractObserver& operator = (const AbstractObserver& observer);

virtual void notify(Notification\* pNf) const = 0;

virtual bool equals(const AbstractObserver& observer) const = 0;

virtual bool accepts(Notification\* pNf) const = 0;

virtual AbstractObserver\* clone() const = 0;

virtual void disable() = 0;

};

        所有的接收者代理类都从类AbstractObserver继承，因为真实接收者的类类型未定，所以接受者代理类只能由模板技术去实现。这解决了处理时第一个问题。第二个问题是，不同的类处理函数可以不同。可以拥有一个，或多个参数，可以拥有或没有返回值。而编译器编译时，必须指定函数指针的类型。解决方法即把处理函数的类型固定下来。  
        在Poco库的内部实现Observer类和NObserver类中，其定义分别是：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. void (C::\*Callback)(N\*);
2. void (C::\*Callback)(const AutoPtr<N> &);

void (C::\*Callback)(N\*);

void (C::\*Callback)(const AutoPtr<N> &);

        函数调用时分别带了一个参数。这其实解决了所有的问题，因为一个参数可以是结构体，供使用传入传出所需的值。当然我们也可以从AbstractObserver继承实现自己的目标代理类，这样我们可以定义自己所需要的函数类型。让我们来看一下Observer定义，NObserver的分析类似。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. template <class C, class N>
2. class Observer: public AbstractObserver
3. {
4. public:
5. typedef void (C::\*Callback)(N\*);

8. Observer(C& object, Callback method):
9. \_pObject(&object),
10. \_method(method)
11. {
12. }
14. Observer(const Observer& observer):
15. AbstractObserver(observer),
16. \_pObject(observer.\_pObject),
17. \_method(observer.\_method)
18. {
19. }
21. ~Observer()
22. {
23. }
25. Observer& operator = (const Observer& observer)
26. {
27. if (&observer != this)
28. {
29. \_pObject = observer.\_pObject;
30. \_method  = observer.\_method;
31. }
32. return \*this;
33. }
35. void notify(Notification\* pNf) const
36. {
37. Poco::Mutex::ScopedLock lock(\_mutex);

40. if (\_pObject)
41. {
42. N\* pCastNf = dynamic\_cast<N\*>(pNf);
43. if (pCastNf)
44. {
45. pCastNf->duplicate();
46. (\_pObject->\*\_method)(pCastNf);
47. }
48. }
49. }
51. **bool** equals(const AbstractObserver& abstractObserver) const
52. {
53. const Observer\* pObs = dynamic\_cast<const Observer\*>(&abstractObserver);
54. return pObs && pObs->\_pObject == \_pObject && pObs->\_method == \_method;
55. }

58. **bool** accepts(Notification\* pNf) const
59. {
60. return dynamic\_cast<N\*>(pNf) != 0;
61. }
63. AbstractObserver\* clone() const
64. {
65. return new Observer(\*this);
66. }
68. void disable()
69. {
70. Poco::Mutex::ScopedLock lock(\_mutex);
72. \_pObject = 0;
73. }
75. private:
76. Observer();

79. C\*       \_pObject;
80. Callback \_method;
81. mutable Poco::Mutex \_mutex;
82. };

template <class C, class N>

class Observer: public AbstractObserver

{

public:

typedef void (C::\*Callback)(N\*);

Observer(C& object, Callback method):

\_pObject(&object),

\_method(method)

{

}

Observer(const Observer& observer):

AbstractObserver(observer),

\_pObject(observer.\_pObject),

\_method(observer.\_method)

{

}

~Observer()

{

}

Observer& operator = (const Observer& observer)

{

if (&observer != this)

{

\_pObject = observer.\_pObject;

\_method = observer.\_method;

}

return \*this;

}

void notify(Notification\* pNf) const

{

Poco::Mutex::ScopedLock lock(\_mutex);

if (\_pObject)

{

N\* pCastNf = dynamic\_cast<N\*>(pNf);

if (pCastNf)

{

pCastNf->duplicate();

(\_pObject->\*\_method)(pCastNf);

}

}

}

bool equals(const AbstractObserver& abstractObserver) const

{

const Observer\* pObs = dynamic\_cast<const Observer\*>(&abstractObserver);

return pObs && pObs->\_pObject == \_pObject && pObs->\_method == \_method;

}

bool accepts(Notification\* pNf) const

{

return dynamic\_cast<N\*>(pNf) != 0;

}

AbstractObserver\* clone() const

{

return new Observer(\*this);

}

void disable()

{

Poco::Mutex::ScopedLock lock(\_mutex);

\_pObject = 0;

}

private:

Observer();

C\* \_pObject;

Callback \_method;

mutable Poco::Mutex \_mutex;

};

        Observer中存在一个类实例对象的指针\_pObject，以及对应函数入口地址\_method。其处理函数为notify。这里注意两点：  
        1. 使用了dynamic\_cast转换，这意味着接受者处理的消息是向下继承的。如果一个对象订购了Poco::Notification，那么它将接受到所有继承自Poco::Notification的消息。

        2. 调用了pCastNf->duplicate(),增加了引用计数，这意味着处理者在处理函数中必须相应的去调用pCastNf->release(),去释放引用计数。在这里我倒是没有搞明白，为什么要调用duplicate()，在我看来不调用也完全可以。可能是为了照顾引用计数对象的语义，即引用计数对象的所有权发生了改变，从NotificationCenter对象独占转变成为了真实处理类对象和NotificationCenter对象共同拥有所有权。

**3.4 一个使用例子**

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/arau_sh/article/details/8673459)

1. #include "Poco/NotificationCenter.h"
2. #include "Poco/Notification.h"
3. #include "Poco/Observer.h"
4. #include "Poco/NObserver.h"
5. #include "Poco/AutoPtr.h"
6. #include <iostream>
7. using Poco::NotificationCenter;
8. using Poco::Notification;
9. using Poco::Observer;
10. using Poco::NObserver;
11. using Poco::AutoPtr;
12. class BaseNotification: public Notification
13. {
14. };
15. class SubNotification: public BaseNotification
16. {
17. };

20. class Target
21. {
22. public:
23. void handleBase(BaseNotification\* pNf)
24. {
25. std::cout << "handleBase: " << pNf->name() << std::endl;
26. pNf->release(); // we got ownership, so we must release
27. }
28. void handleSub(const AutoPtr<SubNotification>& pNf)
29. {
30. std::cout << "handleSub: " << pNf->name() << std::endl;
31. }
32. };

35. **int** main(**int** argc, **char**\*\* argv)
36. {
37. NotificationCenter nc;
38. Target target;
39. nc.addObserver(
40. Observer<Target, BaseNotification>(target, &Target::handleBase)
41. );
42. nc.addObserver(
43. NObserver<Target, SubNotification>(target, &Target::handleSub)
44. );
45. nc.postNotification(new BaseNotification);
46. nc.postNotification(new SubNotification);
47. nc.removeObserver(
48. Observer<Target, BaseNotification>(target, &Target::handleBase)
49. );
50. nc.removeObserver(
51. NObserver<Target, SubNotification>(target, &Target::handleSub)
52. );
53. return 0;
54. }

#include "Poco/NotificationCenter.h"

#include "Poco/Notification.h"

#include "Poco/Observer.h"

#include "Poco/NObserver.h"

#include "Poco/AutoPtr.h"

#include <iostream>

using Poco::NotificationCenter;

using Poco::Notification;

using Poco::Observer;

using Poco::NObserver;

using Poco::AutoPtr;

class BaseNotification: public Notification

{

};

class SubNotification: public BaseNotification

{

};

class Target

{

public:

void handleBase(BaseNotification\* pNf)

{

std::cout << "handleBase: " << pNf->name() << std::endl;

pNf->release(); // we got ownership, so we must release

}

void handleSub(const AutoPtr<SubNotification>& pNf)

{

std::cout << "handleSub: " << pNf->name() << std::endl;

}

};

int main(int argc, char\*\* argv)

{

NotificationCenter nc;

Target target;

nc.addObserver(

Observer<Target, BaseNotification>(target, &Target::handleBase)

);

nc.addObserver(

NObserver<Target, SubNotification>(target, &Target::handleSub)

);

nc.postNotification(new BaseNotification);

nc.postNotification(new SubNotification);

nc.removeObserver(

Observer<Target, BaseNotification>(target, &Target::handleBase)

);

nc.removeObserver(

NObserver<Target, SubNotification>(target, &Target::handleSub)

);

return 0;

}

**3.5 最后给出同步通知的类图**

