enable_shared_from_this模板类使用完全解析

作者：hahaya
日期：2013-11-28

以前都没有用过enable_shared_from_this模板类，虽然经常遇到但是也没怎么去关注，今天抽时间好好学习了下enable_shared_from_this模板类，发现在使用shared_ptr模板类和enable_shared_from_this模板类时有许多陷阱的，故记录于此。

什么时候该使用enable_shared_from_this模板类

在看下面的例子之前，简单说下使用背景，单有一个类，某个函数需要返回当前对象的指针，我们返回的是shared_ptr，为什么使用智能指针呢，这是因为：当我们使用智能指针管理资源时，必须统一使用智能指针，而不能再某些地方使用智能指针，某些地方使用原始指针，否则不能保持智能指针的语义，从而产生各种错误。好了，介绍完背景，看下面的一段小程序：

```
1. #include <iostream>
2. #include <boost/shared_ptr.hpp>
3. class Test
4. {
5. public:
6.     //析构函数
7.     ~Test() { std::cout << "Test Destructor." << std::endl; }
8.     //获取指向当前对象的指针
9.     boost::shared_ptr<Test> GetObject()
10.     {
11.         boost::shared_ptr<Test> pTest(this);
12.         return pTest;
13.     }
14. };
15. int main(int argc, char *argv[])
16. {
17.     {
18.         boost::shared_ptr<Test> p( new Test( ));
19.         boost::shared_ptr<Test> q = p->GetObject();
20.     }
21.     return 0;
22. }
```

程序输出：

```
1.      Test Destructor.
2.      Test Destructor.
```

从上面的输出你发现了什么，很明显的发现只创建new了一个Test对象，但是却调用了两次析构函数，这对程序来说肯定是一个灾难。为什么会出现这种情况呢？main函数中的`boost::shared_ptr<Test> p( new Test( ));`将shared_ptr中引用计数器的值设置为1，而在GetObject函数中又通过`boost::shared_ptr<Test> pTest(this);`又将shared_ptr中的引用计数器的值增加了1，故在析构时一个Test对象被析构了两次。即产生这个错误的原因是通过同一个Test指针对象创建了多个shared_ptr，这是绝对禁止的。同时这也提醒我们在使用shared_ptr时一定不能通过同一个指针对象创建一个以上的shared_ptr对象。那么有什么方法从一个类的成员函数中获取当前对象的shared_ptr呢，其实方法很简单：只需要该类继承至enable_shared_from_this模板类,然后在需要shared_prt的地方调用enable_shared_from_this模板类的成员函数shared_from_this()即可，下面是改进后的代码：

```
1. #include <iostream>
2. #include <boost/enable_shared_from_this.hpp>
3. #include <boost/shared_ptr.hpp>
4. class Test : public boost::enable_shared_from_this<Test>        //改进1
5. {
6. public:
7.      //析构函数
8.      ~Test() { std::cout << "Test Destructor." << std::endl; }
9.      //获取指向当前对象的指针
10.     boost::shared_ptr<Test> GetObject()
11.     {
12.         return shared_from_this();      //改进2
13.     }
14. };
15. int main(int argc, char *argv[])
16. {
17.     {
18.         boost::shared_ptr<Test> p( new Test( ));
19.         boost::shared_ptr<Test> q = p->GetObject();
20.     }
21.     return 0;
22. }
```

程序输出：

```
1.      Test Destructor.
```

从输出对象只被析构了一次，这是我们想要的结果，因此enable_shared_from_this模板类的作用是：用

来作为一个基类，它允许从一个成员函数中获得一个当前对象的shared_ptr。那么enable_shared_from_this模板类到底是如何工作的了？请看下文分解~

enable_shared_from_this模板类实现

打开enable_shared_from_this.hpp文件，会发现enable_shared_from_this模板类的实现如下：

```
1. template<class T> class enable_shared_from_this
2. {
3. protected:
4.     enable_shared_from_this() BOOST_NOEXCEPT
5.     {
6.     }
7.     enable_shared_from_this(enable_shared_from_this const &)
   BOOST_NOEXCEPT
8.     {
9.     }
10.    enable_shared_from_this & operator=(enable_shared_from_this const &)
   BOOST_NOEXCEPT
11.    {
12.        return *this;
13.    }
14.    ~enable_shared_from_this() BOOST_NOEXCEPT // ~weak_ptr<T> newer
   throws, so this call also must not throw
15.    {
16.    }
17. public:
18.    shared_ptr<T> shared_from_this()
19.    {
20.        shared_ptr<T> p( weak_this_ );
21.        BOOST_ASSERT( p.get() == this );
22.        return p;
23.    }
24.    shared_ptr<T const> shared_from_this() const
25.    {
26.        shared_ptr<T const> p( weak_this_ );
27.        BOOST_ASSERT( p.get() == this );
28.        return p;
29.    }
30. public: // actually private, but avoids compiler template friendship
   issues
31.    // Note: invoked automatically by shared_ptr; do not call
32.    template<class X, class Y> void _internal_accept_owner(
   shared_ptr<X> const * ppx, Y * py ) const
```

```
33.     {
34.         if( weak_this_.expired() )
35.         {
36.             weak_this_ = shared_ptr<T>( *ppx, py );
37.         }
38.     }
39. private:
40.     mutable weak_ptr<T> weak_this_;
41. };
```

从enable_shared_from_this模板类的实现文件中我们可以很容易的发现我们只能使用返回shared_ptr的shared_from_this()和返回shared_ptr的shared_from_this()，因为这两个版本的shared_from_this()是public权限的，还有一个public权限的是*internal_accept_owner函数，但是注释中已经明显指出不能调用这个函数，这个函数会被shared_ptr自动调用，*internal_accept_owner函数用来初始化enable_shared_from_this模板类中的唯一成员变量weak_ptr weak_this。*而shared_from_this()中是通过将weak_ptr weak_this*转化成shared_ptr和shared_ptr来返回的，因此在使用shared_from_this()之前需要先初始化weak_ptr weak_this*对象，而weak_ptr weak_this*对象是在_internal_accept_owner函数中进行的初始化，也就是说先需要创建shared_ptr对象。即在使用shared_from_this()函数之前，应该先初始化对象的基类enable_shared_from_this,接着再初始化对象，最后初始化shared_ptr。正因为有这个特点所以会出现以下常见的错误：

使用enable_shared_from_this常见错误

先来看情形1：

```
1. class Test : public boost::enable_shared_from_this<Test>
2. {
3.     Test() { boost::shared_ptr<Test> pTest = shared_from_this(); }
4. };
```

这种用法明显是错的，虽然对象的基类enable_shared_from_this类的构造函数已经被调用，但是shared_ptr的构造函数并没有被调用，因此weak_ptr weak_this_并没有被初始化，所以这时调用shared_from_this()是错误的。

接着我们来看情形2：

```
1. class Test : public boost::enable_shared_from_this<Test>
2. {
3.     void func() { boost::shared_ptr<Test> pTest = shared_from_this(); }
4. };
5. int main()
6. {
```

```
7.      Test test;
8.      test.func();     //错误
9.      Test pTest = new Test;
10.     pTest->func(); //错误
11. }
```

同样这种做法也是错误的，和情形1同样的原因shared_ptr的构造函数并没有被调用，因此weak_ptr weak_this_并没有被初始化。

正确的做法应该是：

```
1. class Test : public boost::enable_shared_from_this<Test>
2. {
3.      void func() { boost::shared_ptr<Test> pTest = shared_from_this(); }
4. };
5. int main()
6. {
7.      shared_ptr<Test> pTest( new Test() );
8.      pTest->func();
9. }
```

`shared_ptr<Test> pTest( new Test() );`这句话依次执行的顺序是：1 调用enable_shared_from_this的构造函数。2 调用Test的构造函数。 3 调用shared_ptr的构造函数初始化weak_ptr weak_this_。最后才能通过func()函数使用shared_from_this函数。

从上面的错误中我们知道在使用enable_shared_from_this类中的shared_from_this()函数时应该注意：
1. 不能在对象的构造函数中使用shared_from_this()函数。
2. 先需要调用enable_shared_from_this类的构造函数，接着调用对象的构造函数，最后需要调用shared_ptr类的构造函数初始化enable_shared_from_this的成员变量weak_this_。然后才能使用shared_from_this()函数。
3. 如何程序中使用了智能指针shared_ptr,则程序中统一使用智能指针，不能使用原始指针，以免出现错误。

作者：hahaya
出处：http://hahaya.github.com/use-enable-shared-from-this

分类： blog 标签： boost