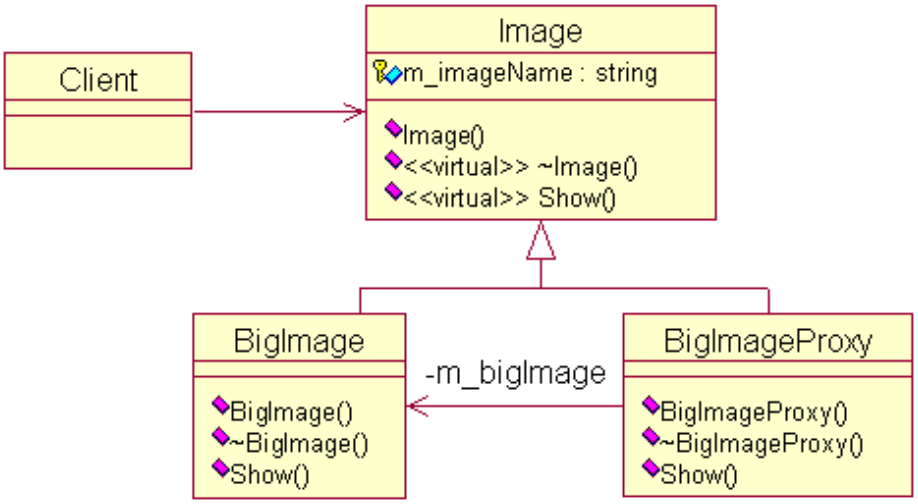
設計模式C++實現（８）——代理模式

星期六, 2013 12月 14, 1:05 上午

軟件領域中的設計模式為開發人員提供了一種使用專家設計經驗的有效途徑。設計模式中運用了面向對象編程語言的重要特性：封裝、繼承、多態，真正領悟設計模式的精髓是可能一個漫長的過程，需要大量實踐經驗的積累。最近看設計模式的書，對於每個模式，用C++寫了個小例子，加深一下理解。主要參考《大話設計模式》和《設計模式:可復用面向對象軟件的基礎》（DP）兩本書。本文介紹代理模式的實現。

[DP]上的定義：為其他對象提供一種代理以控制對這個對象的訪問。有四種常用的情況：（1）遠程代理，（2）虛代理，（3）保護代理，（4）智能引用。本文主要介紹虛代理和智能引用兩種情況。

考慮一個可以在文檔中嵌入圖形對象的文檔編輯器。有些圖形對象的創建開銷很大。但是打開文檔必須很迅速，因此我們在打開文檔時應避免一次性創建所有開銷很大的對象。這裡就可以運用代理模式，在打開文檔時，並不打開圖形對象，而是打開圖形對象的代理以替代真實的圖形。待到真正需要打開圖形時，仍由代理負責打開。這是[DP]一書上的給的例子。下面給出代理模式的UML圖。



簡單實現如下：

```
class Image
{
public:
    Image(string name): m_imageName(name) {}
    virtual ~Image() {}
    virtual void Show() {}
protected:
    string m_imageName;
};
class BigImage: public Image
{
public:
    BigImage(string name):Image(name) {}
    ~BigImage() {}
    void Show() { cout<<"Show big image : "<<m_imageName<<endl; }
};
class BigImageProxy: public Image
{
private:
```

```
20.     BigImage *m_bigImage;
21. public:
22.     BigImageProxy(string name):Image(name),m_bigImage(0) {}
23.     ~BigImageProxy() { delete m_bigImage; }
24.     void Show()
25.     {
26.         if(m_bigImage == NULL)
27.             m_bigImage = new BigImage(m_imageName);
28.         m_bigImage->Show();
29.     }
30. };
```

客戶調用：

```
1. int main()
2. {
3.     Image *image = new BigImageProxy("proxy.jpg"); //代理
4.     image->Show(); //需要時由代理負責打開
5.     delete image;
6.     return 0;
7. }
```

在這個例子屬於虛代理的情況，下面給兩個智能引用的例子。一個是C++中的auto_ptr，另一個是smart_ptr。自己實現了一下。先給出auto_ptr的代碼實現：

```
1. template<class T>
2. class auto_ptr {
3. public:
4.     explicit auto_ptr(T *p = 0): pointee(p) {}
5.     auto_ptr(auto_ptr<T>& rhs): pointee(rhs.release()) {}
6.     ~auto_ptr() { delete pointee; }
7.     auto_ptr<T>& operator=(auto_ptr<T>& rhs)
8.     {
9.         if (this != &rhs) reset(rhs.release());
10.        return *this;
11.    }
12.    T& operator*() const { return *pointee; }
13.    T* operator->() const { return pointee; }
14.    T* get() const { return pointee; }
15.    T* release()
16.    {
17.        T *oldPointee = pointee;
18.        pointee = 0;
19.        return oldPointee;
20.    }
21.    void reset(T *p = 0)
22.    {
23.        if (pointee != p) {
24.            delete pointee;
25.            pointee = p;
26.        }
27.        }
28. private:
29.    T *pointee;
30. };
```

閱讀上面的代碼，我們可以發現 auto_ptr 類就是一個代理，客戶只需操作auto_prt的對象，而不需要與被代理的指針pointee打交道。auto_ptr 的好處在於為動態分配的對象提供異常安全。因為它用一個對象存儲需要被自動釋放的資源，然後依靠對象的析構函數來釋放資源。這樣客戶就不需要關注資源的釋放，由auto_ptr 對象自動完成。實現中的一個關鍵就是重載瞭解引用操作符和箭頭操作

符，從而使得auto_ptr的使用與真實指針類似。

我們知道C++中沒有垃圾回收機制，可以通過智能指針來彌補，下面給出智能指針的一種實現，採用了引用計數的策略。

```
1.  template <typename T>
2.  class smart_ptr
3.  {
4.  public:
5.      smart_ptr(T *p = 0): pointee(p), count(new size_t(1)) { }  //初始的計數值為1
6.      smart_ptr(const smart_ptr &rhs): pointee(rhs.pointee), count(rhs.count) { ++*count; } //
    拷貝構造函數，計數加1
7.      ~smart_ptr() { decr_count(); }              //析構，計數減1，減到0時進行垃圾回收，即釋放空
    間
8.      smart_ptr& operator= (const smart_ptr& rhs) //重載賦值操作符
9.      {
10.         //給自身賦值也對，因為如果自身賦值，計數器先減1，再加1，並未發生改變
11.         ++*count;
12.         decr_count();
13.         pointee = rhs.pointee;
14.         count = rhs.count;
15.         return *this;
16.     }
17.     //重載箭頭操作符和解引用操作符，未提供指針的檢查
18.     T *operator->() { return pointee; }
19.     const T *operator->() const { return pointee; }
20.     T &operator*() { return *pointee; }
21.     const T &operator*() const { return *pointee; }
22.     size_t get_refcount() { return *count; } //獲得引用計數器值
23. private:
24.     T *pointee;        //實際指針，被代理
25.     size_t *count;     //引用計數器
26.     void decr_count() //計數器減1
27.     {
28.         if(--*count == 0)
29.         {
30.             delete pointee;
31.             delete count;
32.         }
33.     }
34. };
```