[**设计模式C++学习笔记之一（Strategy策略模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/07/2008796.html)

无意中，从网上下到一本电子书《24种设计模式介绍与6大设计原则》，很好奇这里有24种设计模式，印象中GOF写的《设计模式》（Design Patterns），好像只有23种吧。运行起来一看，还真挺吸引咱的，里面提到的例子都很有趣。很感谢作者写出这样好的例子来，我的目的是把作者提到的例子用C++来实现。写这本书的作者是：cbf4life，更详细的内容及说明可以参考原作者博客：cbf4life.cnblogs.com。

这里只进行简单提示和实现编码。

**1.1.解释**

main()，赵云

CContext，锦囊

IStrategy，策略接口

CBackDoor，策略之一

CGivenGreenLight，策略之二

CBlockEnemy，策略之三

说明：一个策略放到一个锦囊里。当用的时候，找到这个锦囊，从锦囊里拿出策略来使用。

注意：锦囊只是简单的装载和调用策略，锦囊里没有逻辑。策略会有更大的自主权，运行更多的逻辑。

看代码：

**//Context.h**

#pragma once  
#include "IStrategy.h"  
class CContext  
{  
public:  
    CContext(IStrategy \*pStrategy);  
    ~CContext(void);  
    void Operate(void);  
private:  
    IStrategy \*m\_pStrategy;  
};

**//Context.cpp**

#include "StdAfx.h"  
#include "Context.h"  
CContext::CContext(IStrategy \*pStrategy)  
{  
    this->m\_pStrategy = pStrategy;  
}  
CContext::~CContext(void)  
{  
    delete this->m\_pStrategy;  
}  
void CContext::Operate(void)  
{  
    this->m\_pStrategy->Operate();  
}

**//IStrategy.h**

#pragma once  
  
class IStrategy  
{  
public:  
    IStrategy(void);  
    virtual ~IStrategy(void);  
    virtual void Operate(void) = 0;  
};

**//BackDoor.h**

#pragma once  
#include "istrategy.h"  
class CBackDoor :public IStrategy  
{  
public:  
    CBackDoor(void);  
    ~CBackDoor(void);  
    void Operate(void);  
};

//BackDoor.cpp

#include "StdAfx.h"  
#include "BackDoor.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CBackDoor::CBackDoor(void)  
{  
}  
CBackDoor::~CBackDoor(void)  
{  
}  
void CBackDoor::Operate(void)  
{  
    cout << "找乔国老帮忙，让吴国太给孙权施加压力" << endl;  
}

**//GivenGreenLight.h**

#pragma once  
#include "istrategy.h"  
class CGivenGreenLight :public IStrategy  
{  
public:  
    CGivenGreenLight(void);  
    ~CGivenGreenLight(void);  
    void Operate(void);  
};

//GivenGreenList.cpp

#include "StdAfx.h"  
#include "GivenGreenLight.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CGivenGreenLight::CGivenGreenLight(void)  
{  
}  
CGivenGreenLight::~CGivenGreenLight(void)  
{  
}  
void CGivenGreenLight::Operate(void)  
{  
    cout << "求吴国太开个绿灯，放行！" << endl;  
}

**//BlockEnemy.h**

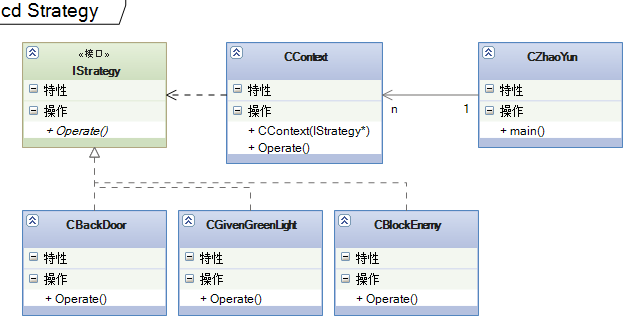
#pragma once  
#include "istrategy.h"  
class CBlockEnemy : public IStrategy  
{  
public:  
    CBlockEnemy(void);  
    ~CBlockEnemy(void);  
    void Operate(void);  
};

//BlockEnemy.cpp

#include "StdAfx.h"  
#include "BlockEnemy.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CBlockEnemy::CBlockEnemy(void)  
{  
}  
CBlockEnemy::~CBlockEnemy(void)  
{  
}  
void CBlockEnemy::Operate()  
{  
    cout << "孙夫人断后，挡住追兵" << endl;  
}

**//Strategy.cpp**

#include "stdafx.h"  
#include "Context.h"  
#include "BackDoor.h"  
#include "GivenGreenLight.h"  
#include "BlockEnemy.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    CContext \*pContext;  
  
    cout << "\14\n\n\n\n\17" << endl;  
    cout << "----------刚刚到吴国的时候拆第一个----------" << endl;  
    pContext = new CContext(new CBackDoor());  
    pContext->Operate();  
    delete pContext;  
  
    cout << "\14\n\n\n\n\17" << endl;  
    cout << "----------刘备乐不思蜀了，拆第二个了----------" << endl;  
    pContext = new CContext(new CGivenGreenLight());  
    pContext->Operate();  
    delete pContext;  
  
    cout << "\14\n\n\n\n\17" << endl;  
    cout << "----------孙权的小兵追了，咋办？拆第三个----------" << endl;  
    pContext = new CContext(new CBlockEnemy());  
    pContext->Operate();  
    delete pContext;  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



一个锦囊只能装一个妙计，赵云可以有多个锦囊。属于对象行为型模式。很简单的一个模式了，貌似这24个里面，这是最简单的了。也好，先用一个简单容易的开头，都说万事开头难，找个简单的开始，鼓励自己坚持学下来，就会有收获。博客也会起到这个作用吧，鼓励自己。上图不是单纯的抽象出来的模式类图，而只是描述了代码里用到的类之间的关系图。真正的抽象类图，只有策略接口和**一个**策略实现类，还有CContext类以及Client。

[**设计模式C++学习笔记之二（Proxy代理模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/08/2009965.html)

代理，一看名字就知道这只是个中介而已，真实的执行者在代理的后面呢。cbf4life在他的书里提的例子也很有趣，更详细的内容及说明可以参考原作者博客：cbf4life.cnblogs.com。现在贴代码，以方便随用随取。

**2.1.解释**

main()，西门庆

IKindWomen，接口

CWangPo，代理

CPanJinLian，实际执行者之一

CJiaShi，实际执行者之二

说明：代理和实际执行者派生于**共同的接口**，代理拥有实际执行者的实例。代理的每一个函数（接口的实现函数），直接调用实际执行者的对应接口函数。

注意：代理只是简单的装载，然后调用实际执行者的函数。

看代码：

**//IKindWomen.h**

#pragma once  
class IKindWomen  
{  
public:  
    IKindWomen(void);  
    virtual ~IKindWomen(void);  
    virtual void MakeEyesWithMan() = 0;  
    virtual void HappyWithMan() = 0;  
};

**//WangPo.h**

#pragma once  
#include "ikindwomen.h"  
class CWangPo :public IKindWomen  
{  
public:  
    CWangPo(IKindWomen \*pKindWomen);  
    ~CWangPo(void);  
    void HappyWithMan(void);  
    void MakeEyesWithMan(void);  
private:  
    IKindWomen \*m\_pKindWomen;  
};

//WangPo.cpp

#include "StdAfx.h"  
#include "WangPo.h"  
CWangPo::CWangPo(IKindWomen \*pKindWomen)  
{  
    this->m\_pKindWomen = pKindWomen;  
}  
CWangPo::~CWangPo(void)  
{  
    delete this->m\_pKindWomen;  
}  
void CWangPo::HappyWithMan()  
{  
    this->m\_pKindWomen->HappyWithMan();  
}  
void CWangPo::MakeEyesWithMan(void)  
{  
    this->m\_pKindWomen->MakeEyesWithMan();  
}

**//PanJinLian.h**

#pragma once  
#include "ikindwomen.h"  
class CPanJinLian :public IKindWomen  
{  
public:  
    CPanJinLian(void);  
    ~CPanJinLian(void);  
    void HappyWithMan(void);  
    void MakeEyesWithMan(void);  
};

**//PanJinLian.cpp**

#include "StdAfx.h"  
#include "PanJinLian.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CPanJinLian::CPanJinLian(void)  
{  
}  
CPanJinLian::~CPanJinLian(void)  
{  
}  
void CPanJinLian::HappyWithMan(void)  
{  
    cout << " 潘金莲和男人做那个... ... " << endl;  
}  
void CPanJinLian::MakeEyesWithMan(void)  
{  
    cout << " 潘金莲抛媚眼 " << endl;  
}

**//JiaShi.h**

#pragma once  
#include "ikindwomen.h"  
class CJiaShi :public IKindWomen  
{  
public:  
    CJiaShi(void);  
    ~CJiaShi(void);  
    void HappyWithMan(void);  
    void MakeEyesWithMan(void);  
};

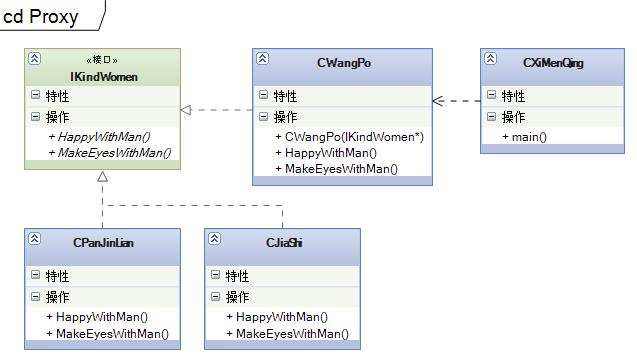
//JiaShi.cpp

#include "StdAfx.h"  
#include "JiaShi.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CJiaShi::CJiaShi(void)  
{  
}  
CJiaShi::~CJiaShi(void)  
{  
}  
void CJiaShi::HappyWithMan(void)  
{  
    cout << " 贾氏和男人做那个... ... " << endl;  
}  
void CJiaShi::MakeEyesWithMan(void)  
{  
    cout << " 贾氏抛媚眼 " << endl;  
}

**//Proxy.cpp**  
#include "stdafx.h"  
#include "WangPo.h"  
#include "PanJinLian.h"  
#include "JiaShi.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
void DoPanJinLian()  
{  
    CWangPo \*pWangPo;  
    // 西门庆想找潘金莲，让王婆来安排。  
    pWangPo = new CWangPo(new CPanJinLian());  
    pWangPo->MakeEyesWithMan();  
    pWangPo->HappyWithMan();  
    delete pWangPo;  
}  
void DoJiaShi()  
{  
    CWangPo \*pWangPo;  
    // 西门庆想找贾氏，让王婆来安排。  
    pWangPo = new CWangPo(new CJiaShi());  
    pWangPo->MakeEyesWithMan();  
    pWangPo->HappyWithMan();  
    delete pWangPo;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    // 西门庆想找潘金莲  
    DoPanJinLian();

    // 西门庆想找贾氏  
    DoJiaShi();

    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
  
    system("pause");  
    return 0;  
}



看起来代理模式的结构和策略模式类似，都是由一个类来装载接口的一个实例，策略模式是CContext来装载，代理模式是CWangPo来装载。CContext不是从IStrategy派生，所以不需要实现IStrategy接口函数，而CWangPo是从IKindWomen派生的所以CWangPo很清楚CPanJinLian和CJiaShi的接口函数。这就是代理，代理人知道被代理人能干的事情即函数，所以代理人可以成为中介。

代理模式可以很好的将前后端分开，实现了松散耦合。代理模式属于结构型模式。上图仍然是例子中用到的类相关图，并不是代理模式的抽象类图。这样的类图更容易理解代理模式。抽象类图当然具有更高的抽象层次，但不利于理解。

[**设计模式C++学习笔记之三（Singleton单例模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/09/2010876.html)

单例模式看起来也蛮简单的，就是在系统中只允许产生这个类的一个实例，既然这么简单，就直接贴代码了。更详细的内容及说明可以参考原作者博客：cbf4life.cnblogs.com。

###### 3.1.解释

main()，大臣

CEmperor，需要单例的类

说明：很多大臣拜见的皇帝，只有一个。体现在面向对象方面，CEmperor定义一个静态指针，和一个静态函数，私有化构造函数、析构函数、构造函数复制、重载赋值语句。

注意：线程安全，采用互斥体的方式实现。

看代码：

**//Emperor.h**

#pragma once

#include <iostream>

using std::cout;

using std::endl;

using std::string;

class CEmperor

{

public:

static CEmperor \* GetInstance();

static void ReleaseInstance();

void EmperorInfo(void);

void SetEmperorTag(string tag);

private:

CEmperor(void);

virtual ~CEmperor(void);

CEmperor(const CEmperor&);

CEmperor& operator=(const CEmperor&);

static CEmperor \*m\_pEmperor;

static HANDLE m\_pMutex;

string m\_EmperorTag;

class CGarbo

{

public:

CGarbo()

{

cout << "Create Garbo" << endl;

}

~CGarbo()

{

cout << "Destroy Garbo" << endl;

if (NULL != m\_pEmperor)

{

WaitForSingleObject(m\_pMutex, INFINITE);

if (NULL != m\_pEmperor)

{

cout << "Remove instance" << endl;

delete m\_pEmperor;

m\_pEmperor = NULL;

}

ReleaseMutex(m\_pMutex);

}

if (NULL != m\_pMutex)

{

cout << "Delete mutex" << endl;

CloseHandle(m\_pMutex);

m\_pMutex = NULL;

}

}

};

static CGarbo m\_Garbo;

};

**//Emperor.cpp**

#include "StdAfx.h"

#include "Emperor.h"

#include <iostream>

using std::cout;

using std::endl;

using std::string;

CEmperor\* CEmperor::m\_pEmperor = NULL;

HANDLE CEmperor::m\_pMutex = CreateMutex(NULL, FALSE, NULL);

CEmperor::CGarbo CEmperor::m\_Garbo;

CEmperor::CEmperor(void)

{

cout << "Create CEmperor Instance" << endl;

}

CEmperor::~CEmperor(void)

{

cout << "Destroy CEmperor Instance and release its resource" << endl;

}

void CEmperor::EmperorInfo(void)

{

char msgBuffer[50] = { 0 };

sprintf\_s(msgBuffer, 50, "皇ê帝?某3某3某3... ...(%s).", m\_EmperorTag.c\_str());

string msg(msgBuffer);

cout << msg.c\_str() << endl;

}

CEmperor\* CEmperor::GetInstance()

{

if (NULL == m\_pEmperor)

{

WaitForSingleObject(m\_pMutex, INFINITE);

if (NULL == m\_pEmperor)

m\_pEmperor = new CEmperor();

ReleaseMutex(m\_pMutex);

}

return m\_pEmperor;

}

void CEmperor::ReleaseInstance()

{

if (NULL != m\_pEmperor)

{

WaitForSingleObject(m\_pMutex, INFINITE);

if (NULL != m\_pEmperor)

{

delete m\_pEmperor;

m\_pEmperor = NULL;

}

ReleaseMutex(m\_pMutex);

}

}

void CEmperor::SetEmperorTag( string tag )

{

m\_EmperorTag = tag;

}

**//Singleton.cpp**

#include "stdafx.h"

#include "Emperor.h"

void DoIt()

{

CEmperor \*pEmperor1 = CEmperor::GetInstance();

pEmperor1->SetEmperorTag("95");

pEmperor1->EmperorInfo();

CEmperor \*pEmperor2 = CEmperor::GetInstance();

pEmperor2->EmperorInfo();

CEmperor \*pEmperor3 = CEmperor::GetInstance();

pEmperor3->EmperorInfo();

CEmperor \*pEmperor4 = pEmperor3;

pEmperor4->EmperorInfo();

CEmperor \*pEmperor5 = NULL;

pEmperor5 = pEmperor4;

pEmperor5->EmperorInfo();

CEmperor::ReleaseInstance();

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

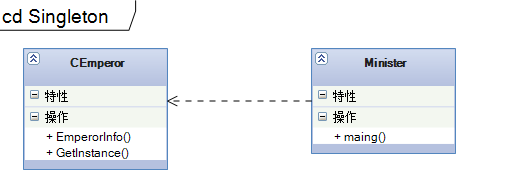
DoIt();

\_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);

\_CrtDumpMemoryLeaks();

return 0;

}

[[](http://images.cnblogs.com/cnblogs_com/wanggary/201104/201104092230199087.png)](http://images.cnblogs.com/cnblogs_com/wanggary/201104/201104092230199087.png)

单例模式比较简单，但在项目中使用的时候，需要明确只调用CEmperor的GetInstance函数来获取实例。在C#里  
有更简单的方法，那就是声明只读的静态变量，比C++简单多了。  
但C++更吸引人们去研究，这就是软件研发的乐趣吧。

[**设计模式C++学习笔记之五（Factory Method工厂方法模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/11/2013033.html)

工厂方法模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。这个例子更详细的内容及说明可以参考原作者博客：cbf4life.cnblogs.com。贴代码了。

**5.1.解释**

main()，女娲

IHuman，产品接口

CYellowHuman，产品之一

CWhiteHuman，产品之二

CBlackHuman，产品之三

IHumanFactory，工厂接口

CYellowHumanFactory，工厂之一

CWhiteHumanFactory，工厂之二

CBlackHumanFactory，工厂之三

看代码：

**//IHuman.h**

#pragma once  
class IHuman  
{  
public:  
    IHuman(void)  
    {  
    }  
    virtual ~IHuman(void)  
    {  
    }  
    virtual void Laugh() = 0;  
    virtual void Cry() = 0;  
    virtual void Talk() = 0;  
};

**//YellowHuman.h**

#pragma once  
#include "ihuman.h"  
class CYellowHuman :public IHuman  
{  
public:  
    CYellowHuman(void);  
    ~CYellowHuman(void);  
    void Laugh();  
    void Cry();  
    void Talk();  
};

**//YellowHuman.cpp**

#include "StdAfx.h"  
#include "YellowHuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CYellowHuman::CYellowHuman(void)  
{  
}  
CYellowHuman::~CYellowHuman(void)  
{  
}  
void CYellowHuman::Cry()  
{  
    cout << "黄色人种会哭" << endl;  
}  
void CYellowHuman::Laugh()  
{  
    cout << "黄色人种会大笑，幸福呀！" << endl;  
}  
void CYellowHuman::Talk()  
{  
    cout << "黄色人种会说话，一般说的都是双字节" << endl;  
}

**//WhiteHuman.h**

#pragma once  
#include "ihuman.h"  
class CWhiteHuman :public IHuman  
{  
public:  
    CWhiteHuman(void);  
    ~CWhiteHuman(void);  
    void Laugh();  
    void Cry();  
    void Talk();  
};

**//WhiteHuman.cpp**

#include "StdAfx.h"  
#include "WhiteHuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CWhiteHuman::CWhiteHuman(void)  
{  
}  
CWhiteHuman::~CWhiteHuman(void)  
{  
}  
void CWhiteHuman::Cry()  
{  
    cout << "白色人种会哭" << endl;  
}  
void CWhiteHuman::Laugh()  
{  
    cout << "白色人种会大笑，侵略的笑声" << endl;  
}  
void CWhiteHuman::Talk()  
{  
    cout << "白色人种会说话，一般都是单字节" << endl;  
}

**//BlackHuman.h**

#pragma once  
#include "ihuman.h"  
class CBlackHuman :  
    public IHuman  
{  
public:  
    CBlackHuman(void);  
    ~CBlackHuman(void);  
    void Laugh();  
    void Cry();  
    void Talk();  
};

**//BlackHuman.cpp**

#include "StdAfx.h"  
#include "BlackHuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CBlackHuman::CBlackHuman(void)  
{  
}  
CBlackHuman::~CBlackHuman(void)  
{  
}  
void CBlackHuman::Cry()  
{  
    cout << "黑人会哭" << endl;  
}  
void CBlackHuman::Laugh()  
{  
    cout << "黑人会笑" << endl;  
}  
void CBlackHuman::Talk()  
{  
    cout << "黑人可以说话，一般人听不懂" << endl;  
}

**//IHumanFactory.h**

#pragma once  
#include "IHuman.h"  
class IHumanFactory  
{  
public:  
    IHumanFactory(void)  
    {  
    }  
    virtual ~IHumanFactory(void)  
    {  
    }  
    virtual IHuman \* CreateHuman() = 0;  
};  
**//YellowHuman.h**

#pragma once  
#include "ihumanfactory.h"  
class CYellowHumanFactory :public IHumanFactory  
{  
public:  
    CYellowHumanFactory(void);  
    ~CYellowHumanFactory(void);  
    virtual IHuman \* CreateHuman(void);  
};

**//YellowHumanFactory.cpp**

#include "StdAfx.h"  
#include "YellowHumanFactory.h"  
#include "YellowHuman.h"  
CYellowHumanFactory::CYellowHumanFactory(void)  
{  
}  
CYellowHumanFactory::~CYellowHumanFactory(void)  
{  
}  
IHuman \* CYellowHumanFactory::CreateHuman( void )  
{  
    return new CYellowHuman();  
}  
**//WhiteHuman.h**

#pragma once  
#include "ihumanfactory.h"  
class CWhiteHumanFactory :  
    public IHumanFactory  
{  
public:  
    CWhiteHumanFactory(void);  
    ~CWhiteHumanFactory(void);  
    virtual IHuman \* CreateHuman(void);  
};  
  
**//WhiteHumanFactory.cpp**

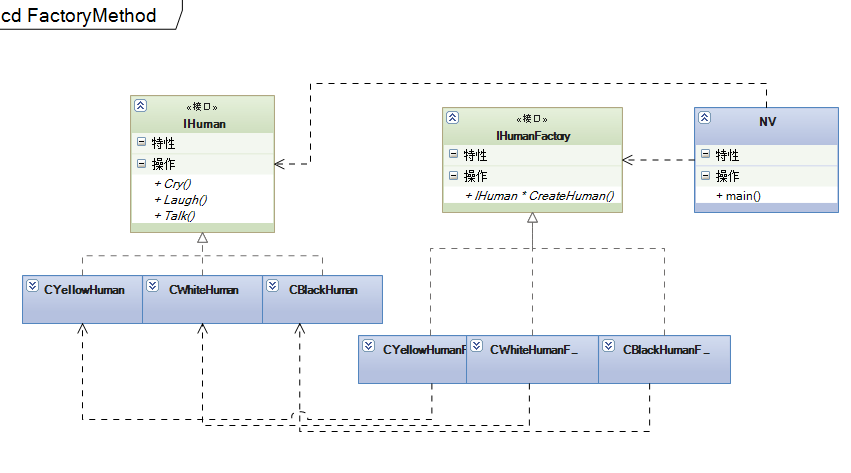
#include "StdAfx.h"  
#include "WhiteHumanFactory.h"  
#include "WhiteHuman.h"  
CWhiteHumanFactory::CWhiteHumanFactory(void)  
{  
}  
CWhiteHumanFactory::~CWhiteHumanFactory(void)  
{  
}  
IHuman \* CWhiteHumanFactory::CreateHuman( void )  
{  
    return new CWhiteHuman();  
}  
**//BlackHuman.h**

#pragma once  
#include "ihumanfactory.h"  
class CBlackHumanFactory :  
    public IHumanFactory  
{  
public:  
    CBlackHumanFactory(void);  
    ~CBlackHumanFactory(void);  
    virtual IHuman \* CreateHuman();  
};  
**//BlackHumanFactory.cpp**

#include "StdAfx.h"  
#include "BlackHumanFactory.h"  
#include "BlackHuman.h"  
CBlackHumanFactory::CBlackHumanFactory(void)  
{  
}  
CBlackHumanFactory::~CBlackHumanFactory(void)  
{  
}  
IHuman \* CBlackHumanFactory::CreateHuman()  
{  
    return new CBlackHuman();  
}

**//FactoryMethod.cpp**

**// FactoryMethod.cpp : 定义控制台应用程序的入口点。  
//**  
#include "stdafx.h"  
#include "IHuman.h"  
#include "YellowHuman.h"  
#include "WhiteHuman.h"  
#include "BlackHuman.h"  
#include "SimpleHumanFactory.h"  
#include "StandardHumanFactory.h"  
#include "IHumanFactory.h"  
#include "YellowHumanFactory.h"  
#include "WhiteHumanFactory.h"  
#include "BlackHumanFactory.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
void DoFactoryMethod1()  
{  
    cout << "----------第一批人是这样的：黄种人工厂来生产黄种人" << endl;  
    IHumanFactory \*pHumanFactory = new CYellowHumanFactory();  
    IHuman \*pHuman = pHumanFactory->CreateHuman();  
    pHuman->Cry();  
    pHuman->Laugh();  
    pHuman->Talk();  
    delete pHuman;  
    delete pHumanFactory;  
}  
void DoFactoryMethod2()  
{  
    cout << "----------第二批人是这样的：白种人工厂来生产白种人" << endl;  
    IHumanFactory \*pHumanFactory = new CWhiteHumanFactory();  
    IHuman \*pHuman = pHumanFactory->CreateHuman();  
    pHuman->Cry();  
    pHuman->Laugh();  
    pHuman->Talk();  
    delete pHuman;  
    delete pHumanFactory;  
}  
void DoFactoryMethod3()  
{  
    cout << "----------第一批人是这样的：黑种人工厂来生产黑种人" << endl;  
    IHumanFactory \*pHumanFactory = new CBlackHumanFactory();  
    IHuman \*pHuman = pHumanFactory->CreateHuman();  
    pHuman->Cry();  
    pHuman->Laugh();  
    pHuman->Talk();  
    delete pHuman;  
    delete pHumanFactory;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //工厂方法  
    cout << "----------工厂方法：" << endl;  
    DoFactoryMethod1();  
    DoFactoryMethod2();  
    DoFactoryMethod3();  
      
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



工厂方法属于创建型模式，适合于产品不太多的情况。产品如果很多，则需要用抽象工厂来实现。

[**设计模式C++学习笔记之五（Factory Method工厂方法模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/11/2013033.html)

工厂方法模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。这个例子更详细的内容及说明可以参考原作者博客：cbf4life.cnblogs.com。贴代码了。

**5.1.解释**

main()，女娲

IHuman，产品接口

CYellowHuman，产品之一

CWhiteHuman，产品之二

CBlackHuman，产品之三

IHumanFactory，工厂接口

CYellowHumanFactory，工厂之一

CWhiteHumanFactory，工厂之二

CBlackHumanFactory，工厂之三

看代码：

**//IHuman.h**

#pragma once  
class IHuman  
{  
public:  
    IHuman(void)  
    {  
    }  
    virtual ~IHuman(void)  
    {  
    }  
    virtual void Laugh() = 0;  
    virtual void Cry() = 0;  
    virtual void Talk() = 0;  
};

**//YellowHuman.h**

#pragma once  
#include "ihuman.h"  
class CYellowHuman :public IHuman  
{  
public:  
    CYellowHuman(void);  
    ~CYellowHuman(void);  
    void Laugh();  
    void Cry();  
    void Talk();  
};

**//YellowHuman.cpp**

#include "StdAfx.h"  
#include "YellowHuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CYellowHuman::CYellowHuman(void)  
{  
}  
CYellowHuman::~CYellowHuman(void)  
{  
}  
void CYellowHuman::Cry()  
{  
    cout << "黄色人种会哭" << endl;  
}  
void CYellowHuman::Laugh()  
{  
    cout << "黄色人种会大笑，幸福呀！" << endl;  
}  
void CYellowHuman::Talk()  
{  
    cout << "黄色人种会说话，一般说的都是双字节" << endl;  
}

**//WhiteHuman.h**

#pragma once  
#include "ihuman.h"  
class CWhiteHuman :  
    public IHuman  
{  
public:  
    CWhiteHuman(void);  
    ~CWhiteHuman(void);  
    void Laugh();  
    void Cry();  
    void Talk();  
};

**//WhiteHuman.cpp**

#include "StdAfx.h"  
#include "WhiteHuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CWhiteHuman::CWhiteHuman(void)  
{  
}  
CWhiteHuman::~CWhiteHuman(void)  
{  
}  
void CWhiteHuman::Cry()  
{  
    cout << "白色人种会哭" << endl;  
}  
void CWhiteHuman::Laugh()  
{  
    cout << "白色人种会大笑，侵略的笑声" << endl;  
}  
void CWhiteHuman::Talk()  
{  
    cout << "白色人种会说话，一般都是单字节" << endl;  
}

**//BlackHuman.h**

#pragma once  
#include "ihuman.h"  
class CBlackHuman :  
    public IHuman  
{  
public:  
    CBlackHuman(void);  
    ~CBlackHuman(void);  
    void Laugh();  
    void Cry();  
    void Talk();  
};

**//BlackHuman.cpp**

#include "StdAfx.h"  
#include "BlackHuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CBlackHuman::CBlackHuman(void)  
{  
}  
CBlackHuman::~CBlackHuman(void)  
{  
}  
void CBlackHuman::Cry()  
{  
    cout << "黑人会哭" << endl;  
}  
void CBlackHuman::Laugh()  
{  
    cout << "黑人会笑" << endl;  
}  
void CBlackHuman::Talk()  
{  
    cout << "黑人可以说话，一般人听不懂" << endl;  
}

**//IHumanFactory.h**

#pragma once  
#include "IHuman.h"  
class IHumanFactory  
{  
public:  
    IHumanFactory(void)  
    {  
    }  
    virtual ~IHumanFactory(void)  
    {  
    }  
    virtual IHuman \* CreateHuman() = 0;  
};  
**//YellowHuman.h**

#pragma once  
#include "ihumanfactory.h"  
class CYellowHumanFactory :public IHumanFactory  
{  
public:  
    CYellowHumanFactory(void);  
    ~CYellowHumanFactory(void);  
    virtual IHuman \* CreateHuman(void);  
};

**//YellowHumanFactory.cpp**

#include "StdAfx.h"  
#include "YellowHumanFactory.h"  
#include "YellowHuman.h"  
CYellowHumanFactory::CYellowHumanFactory(void)  
{  
}  
CYellowHumanFactory::~CYellowHumanFactory(void)  
{  
}  
IHuman \* CYellowHumanFactory::CreateHuman( void )  
{  
    return new CYellowHuman();  
}  
**//WhiteHuman.h**

#pragma once  
#include "ihumanfactory.h"  
class CWhiteHumanFactory :public IHumanFactory  
{  
public:  
    CWhiteHumanFactory(void);  
    ~CWhiteHumanFactory(void);  
    virtual IHuman \* CreateHuman(void);  
};  
  
**//WhiteHumanFactory.cpp**

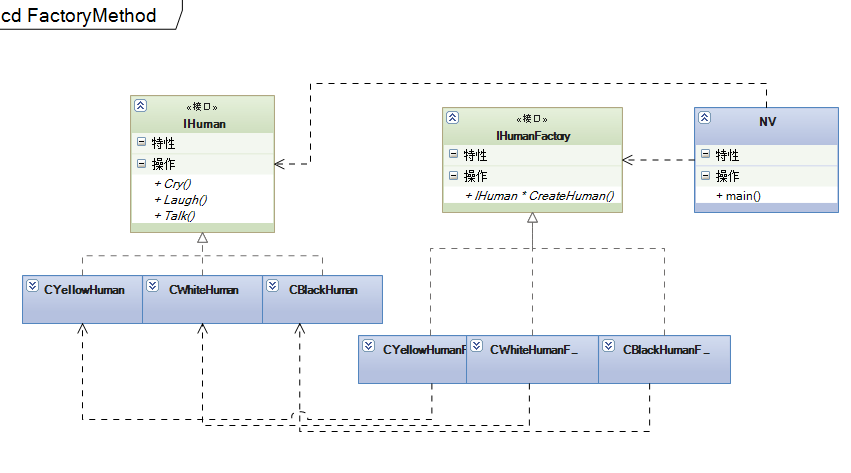
#include "StdAfx.h"  
#include "WhiteHumanFactory.h"  
#include "WhiteHuman.h"  
CWhiteHumanFactory::CWhiteHumanFactory(void)  
{  
}  
CWhiteHumanFactory::~CWhiteHumanFactory(void)  
{  
}  
IHuman \* CWhiteHumanFactory::CreateHuman( void )  
{  
    return new CWhiteHuman();  
}  
**//BlackHuman.h**

#pragma once  
#include "ihumanfactory.h"  
class CBlackHumanFactory :  
    public IHumanFactory  
{  
public:  
    CBlackHumanFactory(void);  
    ~CBlackHumanFactory(void);  
    virtual IHuman \* CreateHuman();  
};  
**//BlackHumanFactory.cpp**

#include "StdAfx.h"  
#include "BlackHumanFactory.h"  
#include "BlackHuman.h"  
CBlackHumanFactory::CBlackHumanFactory(void)  
{  
}  
CBlackHumanFactory::~CBlackHumanFactory(void)  
{  
}  
IHuman \* CBlackHumanFactory::CreateHuman()  
{  
    return new CBlackHuman();  
}

**//FactoryMethod.cpp**

**// FactoryMethod.cpp : 定义控制台应用程序的入口点。  
//**  
#include "stdafx.h"  
#include "IHuman.h"  
#include "YellowHuman.h"  
#include "WhiteHuman.h"  
#include "BlackHuman.h"  
#include "SimpleHumanFactory.h"  
#include "StandardHumanFactory.h"  
#include "IHumanFactory.h"  
#include "YellowHumanFactory.h"  
#include "WhiteHumanFactory.h"  
#include "BlackHumanFactory.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
void DoFactoryMethod1()  
{  
    cout << "----------第一批人是这样的：黄种人工厂来生产黄种人" << endl;  
    IHumanFactory \*pHumanFactory = new CYellowHumanFactory();  
    IHuman \*pHuman = pHumanFactory->CreateHuman();  
    pHuman->Cry();  
    pHuman->Laugh();  
    pHuman->Talk();  
    delete pHuman;  
    delete pHumanFactory;  
}  
void DoFactoryMethod2()  
{  
    cout << "----------第二批人是这样的：白种人工厂来生产白种人" << endl;  
    IHumanFactory \*pHumanFactory = new CWhiteHumanFactory();  
    IHuman \*pHuman = pHumanFactory->CreateHuman();  
    pHuman->Cry();  
    pHuman->Laugh();  
    pHuman->Talk();  
    delete pHuman;  
    delete pHumanFactory;  
}  
void DoFactoryMethod3()  
{  
    cout << "----------第一批人是这样的：黑种人工厂来生产黑种人" << endl;  
    IHumanFactory \*pHumanFactory = new CBlackHumanFactory();  
    IHuman \*pHuman = pHumanFactory->CreateHuman();  
    pHuman->Cry();  
    pHuman->Laugh();  
    pHuman->Talk();  
    delete pHuman;  
    delete pHumanFactory;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //工厂方法  
    cout << "----------工厂方法：" << endl;  
    DoFactoryMethod1();  
    DoFactoryMethod2();  
    DoFactoryMethod3();  
      
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



工厂方法属于创建型模式，适合于产品不太多的情况。产品如果很多，则需要用抽象工厂来实现.

[**设计模式C++学习笔记之六（Facade门面模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/12/2014186.html)

Facade门面模式，也是比较常用的一种模式，基本上所有软件系统中都会用到。 GOF 在《设计模式》一书中给出如下定义：为子系统中的一组接口提供一个一致的界面， Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。简单说，就是将复杂的逻辑封装起来，对外公开简单的接口，由客户程序调用。这里举了一个发送邮件的例子，我们理解为电子邮件吧，普通的邮件应该不需要告诉邮局，我们写的信件内容（呵呵有点较真了）。这个例子更详细的内容及说明可以参考原作者博客：cbf4life.cnblogs.com。

**6.1.解释**

main()，客户

ILetterProcess，接口

CLetterProcessImpl，信件处理的4个函数

CLetterPolice，警察

CModenPostOffice，邮局

说明：邮局对外只有一个窗口，接收信件内容和邮件地址。对内调用邮件处理的4个函数。将复杂逻辑封装在邮局的里面，当需要增加警察来检查信件时，只需在邮局内增加警察检查信件的方法。

注意：将复杂逻辑封装起来，对外只有一个简单的接口。

看代码：

**//ILetterProcess.h**

#pragma once  
#include <iostream>  
using std::string;  
class ILetterProcess  
{  
public:  
    ILetterProcess(void);  
    virtual ~ILetterProcess(void);  
    virtual void WriteContext(string context) = 0;  
    virtual void FillEnvelope(string address) = 0;  
    virtual void LetterIntoEnvelope() = 0;  
    virtual void SendLetter() = 0;  
};

**//ILetterProcess.cpp**

#include "StdAfx.h"  
#include "ILetterProcess.h"  
ILetterProcess::ILetterProcess(void)  
{  
}  
ILetterProcess::~ILetterProcess(void)  
{  
}

**//LetterprocessImpl.h**

#pragma once  
#include "iletterprocess.h"  
class CLetterProcessImpl :public ILetterProcess  
{  
public:  
    CLetterProcessImpl(void);  
    ~CLetterProcessImpl(void);  
  
    void WriteContext(string context);  
    void FillEnvelope(string address);  
    void LetterIntoEnvelope();  
    void SendLetter();  
};

**//LetterProcessImpl.cpp**

#include "StdAfx.h"  
#include "LetterProcessImpl.h"  
#include <iostream>  
using std::string;  
using std::cout;  
using std::endl;  
CLetterProcessImpl::CLetterProcessImpl(void)  
{  
}  
CLetterProcessImpl::~CLetterProcessImpl(void)  
{  
}  
void CLetterProcessImpl::WriteContext(string context)  
{  
    cout << "填写信的内容... ..." << endl;  
}  
void CLetterProcessImpl::FillEnvelope(string address)  
{  
    cout << "填写收件人地址及姓名... ..." << endl;  
}  
void CLetterProcessImpl::LetterIntoEnvelope()  
{  
    cout << "把信放到信封中..." << endl;  
}  
void CLetterProcessImpl::SendLetter()  
{  
    cout << "邮递信件..." << endl;  
}

**//ModenPostOffice.h**

#pragma once  
#include "ILetterProcess.h"  
#include "LetterProcessImpl.h"  
#include "LetterPolice.h"  
#include <iostream>  
using std::string;  
class CModenPostOffice  
{  
public:  
    CModenPostOffice(void);  
    ~CModenPostOffice(void);  
    void SendLetter(string context, string address);  
private:  
    ILetterProcess \*m\_pLetterProcess;  
    CLetterPolice \*m\_pLetterPolice;  
};

**//ModenPostOffice.cpp**

#include "StdAfx.h"  
#include "ModenPostOffice.h"  
CModenPostOffice::CModenPostOffice(void)  
{  
    this->m\_pLetterProcess = new CLetterProcessImpl();  
    this->m\_pLetterPolice = new CLetterPolice();  
}  
CModenPostOffice::~CModenPostOffice(void)  
{  
    delete m\_pLetterProcess;  
    delete m\_pLetterPolice;  
}  
void CModenPostOffice::SendLetter( string context, string address )  
{  
    //帮忙写信  
    m\_pLetterProcess->WriteContext(context);  
    //写好信封  
    m\_pLetterProcess->FillEnvelope(address);  
    //警察要检查信件了  
    m\_pLetterPolice->CheckLetter(m\_pLetterProcess);  
    //把信放到信封中  
    m\_pLetterProcess->LetterIntoEnvelope();  
    //邮递信件  
    m\_pLetterProcess->SendLetter();  
}

**//LetterPolice.h**

#pragma once  
#include "ILetterProcess.h"  
class CLetterPolice  
{  
public:  
    CLetterPolice(void);  
    ~CLetterPolice(void);  
    void CheckLetter(ILetterProcess \*pLetterProcess);  
};

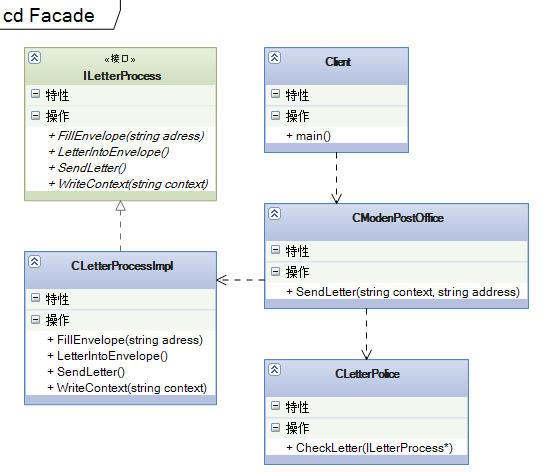
**//LetterPolice.cpp**

#include "StdAfx.h"  
#include "LetterPolice.h"  
CLetterPolice::CLetterPolice(void)  
{  
}  
CLetterPolice::~CLetterPolice(void)  
{  
}  
void CLetterPolice::CheckLetter( ILetterProcess \*pLetterProcess )  
{

    //检查信件，此处省略一万字。  
    return;  
}

**//Facade.cpp**

#include "stdafx.h"  
#include "ILetterProcess.h"  
#include "LetterProcessImpl.h"  
#include "ModenPostOffice.h"  
#include<iostream>  
using std::string;  
using std::cout;  
using std::endl;  
void DoItByPostOffice()  
{  
    CModenPostOffice modenPostOffice;  
    string context = "Hello, It's me, do you know who I am? I'm your old lover. I'd like to ... ...";  
    string address = "Happy Road No. 666, Beijing City, China";  
    modenPostOffice.SendLetter(context, address);  
}  
void DoItYourself()  
{  
    ILetterProcess \*pLetterProcess = new CLetterProcessImpl();  
    pLetterProcess->WriteContext("Hello, It's me, do you know who I am? I'm your old lover. I'd like to ... ...");  
    pLetterProcess->FillEnvelope("Happy Road No. 666, Beijing City, China");  
    pLetterProcess->LetterIntoEnvelope();  
    pLetterProcess->SendLetter();  
    delete pLetterProcess;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //现在的调用方式。对于客户来说确实简单多了。  
    //如需要增加逻辑，例如让警察来检查邮件。可以在邮局里完成这项工作。  
    DoItByPostOffice();  
      
    //原来的调用方式。  
    DoItYourself();  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
  
    return 0;  
}



上面的图仍然只是类图而已，用于帮助理解代码中类之间的关系，而不是模式的抽象类图。CModenPostOffice封装了复杂的处理逻辑，对外只有SendLetter这个函数接口。使客户程序容易了解到想要做什么，应该告诉邮局什么内容，邮局才能正确的工作。

目前为止，还都是比较简单的模式，越到后面越复杂。我的理解是当没有模式的时候，大家编写代码也需要考虑扩展性、伸缩性、稳定性等等。那个时候大家写程序都是自己在摸索经验，逐渐的才意识到程序应该可以适应需求的变化。于是总结很多方法来，让程序既能适应变化，又有一定的可靠性。这使得编程序更有趣，也更抽象了。所以软件开发就是提炼和抽象的过程。类似于哲学的提炼，从特殊到一般。

[**设计模式C++学习笔记之七（AbstractFactory抽象工厂模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/13/2015288.html)

抽象工厂，提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。对于工厂方法来说，抽象工厂可实现一系列产品的生产，抽象工厂更注重产品的组合。

看代码：

**7.1.解释**

main()，女娲

IHuman，产品接口

CYellowHuman，抽象产品之一

CYellowFemaleHuman，具体产品之一

CYellowMaleHuman，具体产品之二

CWhiteHuman，抽象产品之二

CWhiteFemaleHuman，具体产品之三

CWhiteMaleHuman，具体产品之四

CBlackHuman，抽象产品之三

CBlackFemaleHuman，具体产品之五

CBlackMaleHuman，具体产品之六

IHumanFactory，抽象工厂

CStandardHumanFactory，抽象工厂基类（此类可有可无）

CFemaleHumanFactory，工厂之一

CMaleHumanFactory，工厂之二

**//IHuman.h**

#pragma once  
class IHuman  
{  
public:  
  
    IHuman(void)  
    {  
    }  
  
    virtual ~IHuman(void)  
    {  
    }  
  
    virtual void Laugh() = 0;  
    virtual void Cry() = 0;  
    virtual void Talk() = 0;  
    virtual void Sex() = 0;  
};

**//YellowHuman.h**

#pragma once  
#include "ihuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CYellowHuman :public IHuman  
{  
public:  
    CYellowHuman(void)  
    {  
    }  
    ~CYellowHuman(void)  
    {  
    }  
    void Laugh()  
    {  
        cout << "黄色人种会大笑，幸福呀！" << endl;  
    }  
    void Cry()  
    {  
        cout << "黄色人种会哭" << endl;  
    }  
    void Talk()  
    {  
        cout << "黄色人种会说话，一般说的都是双字节" << endl;  
    }  
    virtual void Sex() = 0;  
};  
**//YellowFemaleHuman.h**

#pragma once  
#include "yellowhuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CYellowFemaleHuman :public CYellowHuman  
{  
public:  
    CYellowFemaleHuman(void)  
    {  
    }  
    ~CYellowFemaleHuman(void)  
    {  
    }  
    void Sex()  
    {  
        cout << "该黄种人的性别为女..." << endl;  
    }  
};

**//YellowMaleHuman.h**

#pragma once  
#include "yellowhuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CYellowMaleHuman :  
    public CYellowHuman  
{  
public:  
    CYellowMaleHuman(void)  
    {  
    }  
    ~CYellowMaleHuman(void)  
    {  
    }  
    void Sex()  
    {  
        cout << "该黄种人的性别为男..." << endl;  
    }  
};  
**//WhiteHuman.h**

#pragma once  
#include "ihuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CWhiteHuman :public IHuman  
{  
public:  
    CWhiteHuman(void)  
    {  
    }  
    ~CWhiteHuman(void)  
    {  
    }  
    void Laugh()  
    {  
        cout << "白色人种会大笑，侵略的笑声" << endl;  
    }  
    void Cry()  
    {  
        cout << "白色人种会哭" << endl;  
    }  
    void Talk()  
    {  
        cout << "白色人种会说话，一般都是单字节" << endl;  
    }  
    virtual void Sex() = 0;  
};

**//WhiteFemaleHuman.h**

#pragma once  
#include "whitehuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CWhiteFemaleHuman :public CWhiteHuman  
{  
public:  
    CWhiteFemaleHuman(void)  
    {  
    }  
    ~CWhiteFemaleHuman(void)  
    {  
    }  
    void Sex()  
    {  
        cout << "该白种人的性别为女..." << endl;  
    }  
};

**//WhiteMaleHuman.h**

#pragma once  
#include "whitehuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CWhiteMaleHuman :public CWhiteHuman  
{  
public:  
    CWhiteMaleHuman(void)  
    {  
    }  
    ~CWhiteMaleHuman(void)  
    {  
    }  
    void Sex()  
    {  
        cout << "该白种人的性别为男..." << endl;  
    }  
};

**//BlackHuman.h**

#pragma once  
#include "ihuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CBlackHuman :public IHuman  
{  
public:  
    CBlackHuman(void)  
    {  
    }  
    ~CBlackHuman(void)  
    {  
    }  
    void Laugh()  
    {  
        cout << "黑人会笑" << endl;  
    }  
    void Cry()  
    {  
        cout << "黑人会哭" << endl;  
    }  
    void Talk()  
    {  
        cout << "黑人可以说话，一般人听不懂" << endl;  
    }  
  
    virtual void Sex() = 0;  
};  
**//BlackFemaleHuman.h**

#pragma once  
#include "blackhuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CBlackFemaleHuman :public CBlackHuman  
{  
public:  
    CBlackFemaleHuman(void)  
    {  
    }  
    ~CBlackFemaleHuman(void)  
    {  
    }  
    void Sex()  
    {  
        cout << "该黑种人的性别为女..." << endl;  
    }  
};  
**//BlackMaleHuman.h**

#pragma once  
#include "blackhuman.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
class CBlackMaleHuman :  
    public CBlackHuman  
{  
public:  
    CBlackMaleHuman(void)  
    {  
    }  
    ~CBlackMaleHuman(void)  
    {  
    }  
    void Sex()  
    {  
        cout << "该黑种人的性别为男..." << endl;  
    }  
};

**//IHumanFactory.h**

#pragma once  
#include "IHuman.h"  
class IHumanFactory  
{  
public:  
    IHumanFactory(void)  
    {  
    }  
    virtual ~IHumanFactory(void)  
    {  
    }  
    virtual IHuman \* CreateYellowHuman() = 0;  
    virtual IHuman \* CreateWhiteHuman() = 0;  
    virtual IHuman \* CreateBlackHuman() = 0;  
};

**//StandardHumanFactory.h**

#pragma once  
#include "ihumanfactory.h"  
#include "IHuman.h"  
template<class T>  
class CStandardHumanFactory :public IHumanFactory  
{  
public:  
    CStandardHumanFactory(void)  
    {  
    }  
    ~CStandardHumanFactory(void)  
    {  
    }  
    IHuman \* CreateHuman()  
    {  
        return new T;  
    }  
};

**//MaleHumanFactory.h**

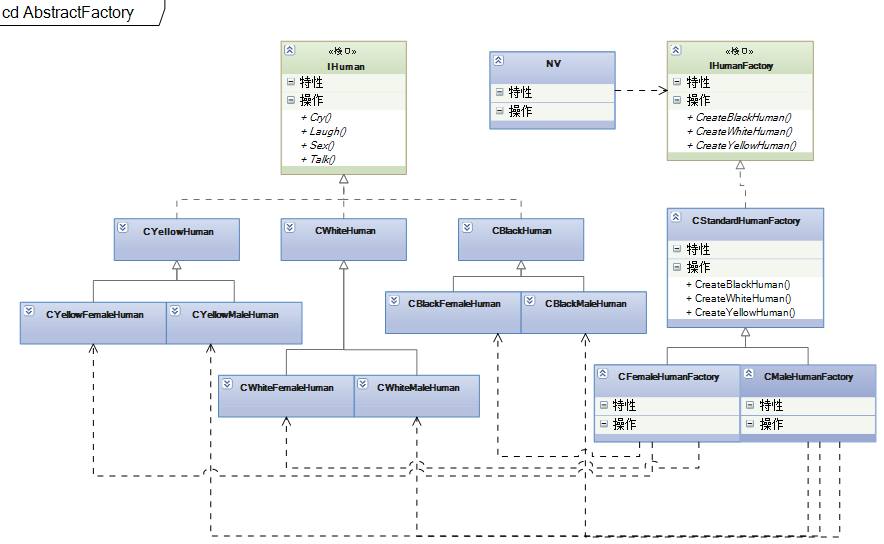
#pragma once  
#include "standardhumanfactory.h"  
#include "IHumanFactory.h"  
template<class T>  
class CMaleHumanFactory :public CStandardHumanFactory<T>  
{  
public:  
    CMaleHumanFactory(void);  
    ~CMaleHumanFactory(void);  
    IHuman \* CreateYellowHuman();  
    IHuman \* CreateWhiteHuman();  
    IHuman \* CreateBlackHuman();  
};

**//MaleHumanFactory.cpp**

#include "StdAfx.h"  
#include "MaleHumanFactory.h"  
template<class T>  
CMaleHumanFactory<T>::CMaleHumanFactory(void)  
{  
}  
template<class T>  
CMaleHumanFactory<T>::~CMaleHumanFactory(void)  
{  
}  
template<class T>  
IHuman \* CMaleHumanFactory<T>::CreateYellowHuman()  
{  
    return CreateHuman();  
}  
template<class T>  
IHuman \* CMaleHumanFactory<T>::CreateWhiteHuman()  
{  
    return CreateHuman();  
}  
template<class T>  
IHuman \* CMaleHumanFactory<T>::CreateBlackHuman()  
{  
    return CreateHuman();  
}

**//FemaleHumanFactory.h**

#pragma once  
#include "standardhumanfactory.h"  
template<class T>  
class CFemaleHumanFactory :public CStandardHumanFactory<T>  
{  
public:  
    CFemaleHumanFactory(void)  
    {  
    }  
    ~CFemaleHumanFactory(void)  
    {  
    }  
    IHuman \* CreateYellowHuman()  
    {  
        return CreateHuman();  
    }  
    IHuman \* CreateWhiteHuman()  
    {  
        return CreateHuman();  
    }  
    IHuman \* CreateBlackHuman()  
    {  
        return CreateHuman();  
    }  
};

**//AbstractFactory.cpp**  
#include "stdafx.h"  
#include "IHuman.h"  
#include "IHumanFactory.h"  
#include "FemaleHumanFactory.h"  
#include "MaleHumanFactory.h"  
#include "MaleHumanFactory.cpp"  
#include "YellowFemaleHuman.h"  
#include "YellowMaleHuman.h"  
#include "WhiteFemaleHuman.h"  
#include "WhiteMaleHuman.h"  
#include "BlackFemaleHuman.h"  
#include "BlackMaleHuman.h"  
void DoIt()  
{  
    IHumanFactory \*pFemaleHumanFactory = new CFemaleHumanFactory<CYellowFemaleHuman>();  
    IHuman \*pYellowFemaleHuman = pFemaleHumanFactory->CreateYellowHuman();  
    pYellowFemaleHuman->Cry();  
    pYellowFemaleHuman->Laugh();  
    pYellowFemaleHuman->Talk();  
    pYellowFemaleHuman->Sex();  
    delete pYellowFemaleHuman;  
    delete pFemaleHumanFactory;  
  
    IHumanFactory \*pMaleHumanFactory = new CMaleHumanFactory<CYellowMaleHuman>();  
    IHuman \*pYellowMaleHuman = pMaleHumanFactory->CreateYellowHuman();  
    pYellowMaleHuman->Cry();  
    pYellowMaleHuman->Laugh();  
    pYellowMaleHuman->Talk();  
    pYellowMaleHuman->Sex();  
    delete pYellowMaleHuman;  
    delete pMaleHumanFactory;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    DoIt();  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}  


以上类图是本例子中所用到的类的相关图，有助于理解程序实现。抽象工厂属于创建型模式。主要用于生产一系列产品，每一个工厂也可以由单件来实现。这里使用模板方式来传递参数，使代码更简洁，但Java或.Net里面的反射方式，在C++还是不能真正实现。

[**设计模式C++学习笔记之八（Adapter适配器模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/13/2015335.html)

适配器模式，使用之处比较特殊，不属于常规设计模式，主要用于不同系统之间的处理。是将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。也是一个比较简单的模式，直接上代码了。

看代码：

**8.1.解释**

main()，主程序

IUserInfo，本系统内接口

CUserInfo，本系统内实现类

IOuterUser，外系统接口

COuterUser，外系统实现类

COuterUserInfo，本系统内适配类

说明：COuterUserInfo实现IUserInfo接口，将外部系统实现类COuterUser转换成本系统内的接口IUserInfo。使用外部数据跟使用本系统内部数据一样。

注意：COuterUserInfo继承了IUserInfo，如果同时继承了COuterUser则是类适配器。如果COuterUserInfo只是使用了COuterUser则是对象适配器。

**//IUserInfo.h//系统内部的实体接口**

#pragma once  
#include <iostream>  
using std::string;  
class IUserInfo  
{  
public:  
    IUserInfo(void)  
    {  
    }  
    virtual ~IUserInfo(void)  
    {  
    }  
    virtual string GetUserName() = 0;  
    virtual string GetHomeAddress() = 0;  
    virtual string GetMobileNumber() = 0;  
    virtual string GetOfficeTelNumber() = 0;  
    virtual string GetJobPosition() = 0;  
    virtual string GetHomeTelNumber() = 0;  
};

**//UserInfo.h//系统内部实体类**

#pragma once  
#include "iuserinfo.h"  
#include <iostream>  
using std::string;  
class CUserInfo :public IUserInfo  
{  
public:  
    CUserInfo(void);  
    ~CUserInfo(void);  
    string GetUserName();  
    string GetHomeAddress();  
    string GetMobileNumber();  
    string GetOfficeTelNumber();  
    string GetJobPosition();  
    string GetHomeTelNumber();  
};

**//UserInfo.cpp**

#include "StdAfx.h"  
#include "UserInfo.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
CUserInfo::CUserInfo(void)  
{  
}  
CUserInfo::~CUserInfo(void)  
{  
}  
string CUserInfo::GetUserName()  
{  
    cout << "姓名叫做..." << endl;  
    return "0";  
}  
string CUserInfo::GetHomeAddress()  
{  
    cout << "这里是员工的家庭地址..." << endl;  
    return "0";  
}  
string CUserInfo::GetMobileNumber()  
{  
    cout << "这个人的手机号码是0000..." << endl;  
    return "0";  
}  
string CUserInfo::GetOfficeTelNumber()  
{  
    cout << "办公室电话是..." << endl;  
    return "0";  
}  
string CUserInfo::GetJobPosition()  
{  
    cout << "这个人的职位是BOSS..." << endl;  
    return "0";  
}  
string CUserInfo::GetHomeTelNumber()  
{  
    cout << "员工的家庭电话是..." << endl;  
    return "0";  
}

**//IOuterUser.h//外部系统实体接口**

#pragma once  
#include "OuterUserBaseInfo.h"  
#include "OuterUserHomeInfo.h"  
#include "OuterUserOfficeInfo.h"  
class IOuterUser  
{  
public:  
    IOuterUser(void)  
    {  
    }  
    ~IOuterUser(void)  
    {  
    }  
    COuterUserBaseInfo \* GetUserBaseInfo();  
    COuterUserHomeInfo \* GetUserHomeInfo();  
    COuterUserOfficeInfo \* GetUserOfficeInfo();  
};

**//OuterUser.h//外部系统实体类**

#pragma once  
#include "OuterUserBaseInfo.h"  
#include "OuterUserHomeInfo.h"  
#include "OuterUserOfficeInfo.h"  
class IOuterUser  
{  
public:  
    IOuterUser(void)  
    {  
    }  
    ~IOuterUser(void)  
    {  
    }  
    COuterUserBaseInfo \* GetUserBaseInfo();  
    COuterUserHomeInfo \* GetUserHomeInfo();  
    COuterUserOfficeInfo \* GetUserOfficeInfo();  
};

**//OuterUser.cpp**

#include "StdAfx.h"  
#include "OuterUser.h"  
#include "OuterUserBaseInfo.h"  
#include "OuterUserHomeInfo.h"  
#include "OuterUserOfficeInfo.h"  
COuterUser::COuterUser(void)  
{  
}  
COuterUser::~COuterUser(void)  
{  
}  
COuterUserBaseInfo \* COuterUser::GetUserBaseInfo()  
{  
    return new COuterUserBaseInfo();  
}  
COuterUserHomeInfo \* COuterUser::GetUserHomeInfo()  
{  
    return new COuterUserHomeInfo();  
}  
COuterUserOfficeInfo \* COuterUser::GetUserOfficeInfo()  
{  
    return new COuterUserOfficeInfo();  
}

**//OuterUserBaseInfo.h**

#pragma once  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
class COuterUserBaseInfo  
{  
public:  
    COuterUserBaseInfo(void)  
    {  
    }  
    ~COuterUserBaseInfo(void)  
    {  
    }  
    string GetUserName()  
    {  
        cout << "姓名叫做..." << endl;  
        return "0";  
    }  
    string GetMobileNumber()  
    {  
        cout << "这个人的手机号码是0001..." << endl;  
        return "0";  
    }  
};

**//OuterUserHomeInfo.h**

#pragma once  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
class COuterUserHomeInfo  
{  
public:  
    COuterUserHomeInfo(void)  
    {  
    }  
    ~COuterUserHomeInfo(void)  
    {  
    }  
    string GetHomeAddress()  
    {  
        cout << "这里是员工的家庭地址..." << endl;  
        return "0";  
    }  
    string GetHomeTelNumber()  
    {  
        cout << "员工的家庭电话是..." << endl;  
        return "0";  
    }  
};

**//OuterUserOfficeInfo.h**

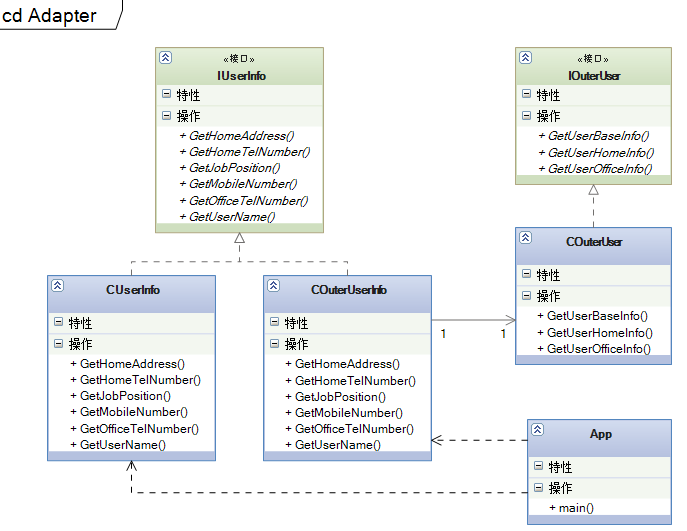
#pragma once  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
class COuterUserOfficeInfo  
{  
public:  
    COuterUserOfficeInfo(void)  
    {  
    }  
    ~COuterUserOfficeInfo(void)  
    {  
    }  
    string GetOfficeTelNumber()  
    {  
        cout << "办公室电话是..." << endl;  
        return "0";  
    }  
    string GetJobPosition()  
    {  
        cout << "这个人的职位是BOSS..." << endl;  
        return "0";  
    }  
};

**//OuterUserInfo.h//由IUserInfo接口派生的实体类，并引入外部系统实体的实例**

#pragma once  
#include "iuserinfo.h"  
#include "OuterUser.h"  
#include <iostream>  
using std::string;  
class COuterUserInfo :  
    public IUserInfo  
{  
public:  
    COuterUserInfo(void);  
    ~COuterUserInfo(void);  
    string GetUserName();  
    string GetHomeAddress();  
    string GetMobileNumber();  
    string GetOfficeTelNumber();  
    string GetJobPosition();  
    string GetHomeTelNumber();  
private:  
    COuterUser \*m\_pOuterUser;  
};  
**//OuterUserInfo.cpp**

#include "StdAfx.h"  
#include "OuterUserInfo.h"  
#include "OuterUserBaseInfo.h"  
#include "OuterUserHomeInfo.h"  
#include "OuterUserOfficeInfo.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
COuterUserInfo::COuterUserInfo(void)  
{  
    m\_pOuterUser = new COuterUser();  
}  
COuterUserInfo::~COuterUserInfo(void)  
{  
    delete m\_pOuterUser;  
}  
string COuterUserInfo::GetUserName()  
{  
    COuterUserBaseInfo \*pBaseInfo = m\_pOuterUser->GetUserBaseInfo();  
    pBaseInfo->GetUserName();  
    delete pBaseInfo;  
    pBaseInfo = NULL;  
    return "0";  
}  
string COuterUserInfo::GetHomeAddress()  
{  
    COuterUserHomeInfo \*pHomeInfo = m\_pOuterUser->GetUserHomeInfo();  
    pHomeInfo->GetHomeAddress();  
    delete pHomeInfo;  
    pHomeInfo = NULL;  
    return "0";  
}  
string COuterUserInfo::GetMobileNumber()  
{  
    COuterUserBaseInfo \*pBaseInfo = m\_pOuterUser->GetUserBaseInfo();  
    pBaseInfo->GetMobileNumber();  
    delete pBaseInfo;  
    pBaseInfo = NULL;  
    return "0";  
}  
string COuterUserInfo::GetOfficeTelNumber()  
{  
    COuterUserOfficeInfo \*pOfficeInfo = m\_pOuterUser->GetUserOfficeInfo();  
    pOfficeInfo->GetOfficeTelNumber();  
    delete pOfficeInfo;  
    pOfficeInfo = NULL;  
    return "0";  
}  
string COuterUserInfo::GetJobPosition()  
{  
    COuterUserOfficeInfo \*pOfficeInfo = m\_pOuterUser->GetUserOfficeInfo();  
    pOfficeInfo->GetJobPosition();  
    delete pOfficeInfo;  
    pOfficeInfo = NULL;  
    return "0";  
}  
string COuterUserInfo::GetHomeTelNumber()  
{  
    COuterUserHomeInfo \*pHomeInfo = m\_pOuterUser->GetUserHomeInfo();  
    pHomeInfo->GetHomeTelNumber();  
    delete pHomeInfo;  
    pHomeInfo = NULL;  
    return "0";  
}  
**//Adapter.cpp//使用方法**

#include "stdafx.h"  
#include "IOuterUser.h"  
#include "IUserInfo.h"  
#include "UserInfo.h"  
#include "OuterUserInfo.h"  
void DoIt()  
{  
    IUserInfo \*pYourGirl = new CUserInfo();  
    for(int i = 0; i < 101; i += 20)  
    {  
        pYourGirl->GetMobileNumber();  
    }  
    delete pYourGirl;  
}  
void NowDoIt()  
{  
    IUserInfo \*pYourGirl = new COuterUserInfo();  
  
    for(int i = 0; i < 101; i += 20)  
    {  
        pYourGirl->GetMobileNumber();  
    }  
  
    delete pYourGirl;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    DoIt();  
  
    NowDoIt();  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



适配器模式属于结构型模式，当出现数据接口不一致的情况下，才会使用到。例如，之前做过的一个系统，所有的组织结构都是由另一个系统来维护，但我们开发的系统需要用到这些组织结构，并且在我们的系统里组织结构是只读的。因为系统设计的原因，所以两个系统的结构设计并不一样，正好应用到了适配器模式。

[设计模式C++学习笔记之九（Template Method模板方法模式）](http://www.cnblogs.com/wanggary/archive/2011/04/15/2016619.html)

模板模式也是相当简单的一种模式，而且是比较常用的。模板模式是定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。cbf4life举的例子是悍马车模型，更详细的例子说明可以参考原作者博客：cbf4life.cnblogs.com。这里只提出C++的设计和实现方法，以方便随用随取。

9.1.解释

main()，客户

CHummerModel，悍马模型

CHummerH1Model，悍马模型型号1

CHummerH2Model，悍马模型型号2

说明：在CHummerModel声明Start、Engineboom、Alarm、Stop虚函数，由派生类实现。基类的Run负责组织逻辑，分别调用这几个派生类实现的函数。

注意：基类中的Run应该禁止派生类覆盖。

看代码：

**//HummerModel.h**

#pragma once  
class CHummerModel  
{  
public:  
    CHummerModel(void);  
    virtual ~CHummerModel(void);  
    void Run();  
protected:  
    virtual void Start() = 0;  
    virtual void Stop() = 0;  
    virtual void Alarm() = 0;  
    virtual void EngineBoom() = 0;  
    virtual bool IsAlarm();  
};

**//HummerModel.cpp**

#include "StdAfx.h"  
#include "HummerModel.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHummerModel::CHummerModel(void)  
{  
}  
CHummerModel::~CHummerModel(void)  
{  
}  
void CHummerModel::Run()  
{  
    //先发动汽车  
    Start();  
    //引擎开始轰鸣  
    EngineBoom();  
    //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭  
    if (IsAlarm())  
        Alarm();  
  
    //到达目的地就停车  
    Stop();  
}  
bool CHummerModel::IsAlarm()  
{  
    //钩子方法，默认喇叭是会响的  
    return true;  
}

//HummerH1Model.h

#pragma once  
#include "hummermodel.h"  
class CHummerH1Model :public CHummerModel  
{  
public:  
    CHummerH1Model(void);  
    ~CHummerH1Model(void);  
    void SetAlarm(bool tag);  
    void Start();  
    void Stop();  
    void Alarm();  
    void EngineBoom();  
    bool IsAlarm();  
private:  
    bool m\_isAlarm;  
};  
**//HummerH1Model.cpp**

#include "StdAfx.h"  
#include "HummerH1Model.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHummerH1Model::CHummerH1Model(void)  
{  
    m\_isAlarm = true;  
}  
CHummerH1Model::~CHummerH1Model(void)  
{  
}  
void CHummerH1Model::Start()  
{  
    cout << "悍马H1发动..." << endl;  
}  
void CHummerH1Model::Stop()  
{  
    cout << "悍马H1停车..." << endl;  
}  
void CHummerH1Model::Alarm()  
{  
    cout << "悍马H1鸣笛" << endl;  
}  
void CHummerH1Model::EngineBoom()  
{  
    cout << "悍马H1引擎声音是这样...." << endl;  
}  
bool CHummerH1Model::IsAlarm()  
{  
    return this->m\_isAlarm;  
}  
void CHummerH1Model::SetAlarm( bool tag )  
{  
    this->m\_isAlarm = tag;  
}

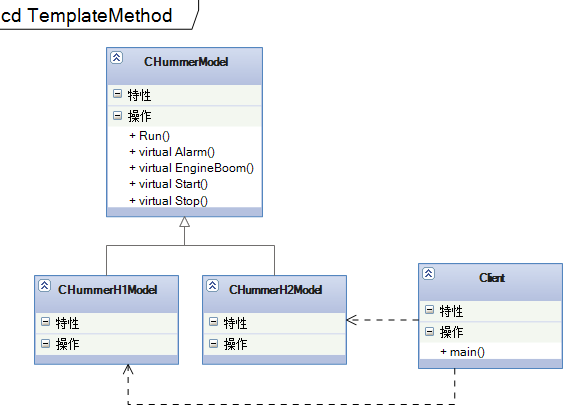
**//HummerH2Model.h**

#pragma once  
#include "hummermodel.h"  
class CHummerH2Model :public CHummerModel  
{  
public:  
    CHummerH2Model(void);  
    ~CHummerH2Model(void);  
    void Start();  
    void Stop();  
    void Alarm();  
    void EngineBoom();  
    bool IsAlarm();  
};  
**//HummerH2Model.cpp**

#include "StdAfx.h"  
#include "HummerH2Model.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHummerH2Model::CHummerH2Model(void)  
{  
}  
CHummerH2Model::~CHummerH2Model(void)  
{  
}  
void CHummerH2Model::Start()  
{  
    cout << "悍马H2发动..." << endl;  
}  
void CHummerH2Model::Stop()  
{  
    cout << "悍马H2停车..." << endl;  
}  
void CHummerH2Model::Alarm()  
{  
    cout << "悍马H2鸣笛" << endl;  
}  
void CHummerH2Model::EngineBoom()  
{  
    cout << "悍马H2引擎声音是这样...." << endl;  
}  
bool CHummerH2Model::IsAlarm()  
{  
    return false;  
}

**//TemplateMethod.cpp**

#include "stdafx.h"  
#include "HummerModel.h"  
#include "HummerH1Model.h"  
#include "HummerH2Model.h"  
#include <crtdbg.h>  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //客户开着H1型号，出去遛弯了  
    CHummerModel \*ph1 = new CHummerH1Model();  
    ph1->Run();  
    delete ph1;  
  
**//客户开H2型号，出去玩耍了**  
    CHummerModel \*ph2 = new CHummerH2Model();  
    ph2->Run();  
    delete ph2;  
  
    //客户开着H1型号，出去遛弯了，并且不让喇叭响  
    CHummerH1Model \*ph11 = new CHummerH1Model();  
    ph11->SetAlarm(false);  
    ph11->Run();  
    delete ph11;  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



由基类的Run来实现基本固定的逻辑，而把不同的部分封装在子类里。Run应该不允许子类覆盖。模板方法属于行为型模式。比较简单，也比较常用。

[设计模式C++学习笔记之十（Builder建造者模式）](http://www.cnblogs.com/wanggary/archive/2011/04/15/2017615.html)

建造者模式，将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。一段晦涩难懂的文字，实现创建不同表示的方法就是给创建的过程传入创建的参数。详细的还是看代码吧。

10.1.解释

main()，客户

CCarModel，产品模型

CBenzModel，奔驰模型

CBMWModel，宝马模型

ICarBuilder，建造者接口

CBenzBuilder，奔驰建造者

CBMWBuilder，宝马建造者

CDirector，导演

说明：CCarModel实现模板方法，Builder负责开始建造产品。建造产品时，构建的顺序由Director或main决定。

注意：建造者模式和抽象工厂非常类似。建造者更重视产品建造时的逻辑顺序，而抽象工厂更重视生产出不同型号的产品，抽象工厂不关心顺序。

看代码：

**// Builder.cpp**  
#include "stdafx.h"  
#include "CarModel.h"  
#include "BenzModel.h"  
#include "BMWModel.h"  
#include "BenzBuilder.h"  
#include "BMWBuilder.h"  
#include "Director.h"  
#include <vector>  
#include <iostream>  
using std::vector;  
using std::string;  
using std::cout;  
using std::endl;

void DoBenzRun()　　//没有使用模式时，需要把步骤一条一条的传入模型。  
{  
    cout << "----------生成奔驰模型----------" << endl;  
    CBenzModel \*pBenz = new CBenzModel();  
    vector<string> seq;  
    seq.push\_back("engine boom");//客户要求run的时候先发动引擎  
    seq.push\_back("start");//启动起来  
    seq.push\_back("stop");//开了一段就停下来  
  
    pBenz->SetSequence(&seq);  
    pBenz->Run();  
    delete pBenz;  
}

**//使用模式后，由benzBuilder和bmwBuilder来生成，并且使用同样的创建顺序。**  
void DoBuilder()　　　　  
{  
    cout << "----------用同一个顺序，生成模型----------" << endl;  
    vector<string> seq;  
    seq.push\_back("engine boom");  
    seq.push\_back("start");  
    seq.push\_back("stop");  
  
    CBenzBuilder benzBuilder;  
    benzBuilder.SetSequence(&seq);  
    CBenzModel \*pBenz = dynamic\_cast<CBenzModel\*>(benzBuilder.GetCarModel());  
    pBenz->Run();  
  
    CBMWBuilder bmwBuilder;  
    bmwBuilder.SetSequence(&seq);  
    CBMWModel \*pBmw = dynamic\_cast<CBMWModel\*>(bmwBuilder.GetCarModel());  
    pBenz->Run();  
}

**//使用指导者来封装创建的逻辑，把创建的顺序内聚在指导者类里面。**  
void DoDirector()  
{  
    cout << "----------批量生成模型----------" << endl;  
    CDirector director;  
  
    //1W辆A类型的奔驰车  
    for(int i = 0; i < 2; i++)  
        director.GetABenzModel()->Run();  
  
    //100W辆B类型的奔驰车  
    for(int i = 0; i < 2; i++)  
        director.GetBBenzModel()->Run();  
  
    //1000W辆C类型的宝马车  
    for(int i = 0; i < 2; i++)  
        director.GetCBMWModel()->Run();  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    DoBenzRun();  
  
    DoBuilder();  
  
    DoDirector();  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}

**//CarModel.h**

#pragma once  
#include <vector>  
#include <iostream>  
using std::vector;  
using std::string;  
class CCarModel  
{  
public:  
    CCarModel(void);  
    virtual ~CCarModel(void);  
    void Run();  
    void SetSequence(vector<string> \*pSeq);  
protected:  
    virtual void Start() = 0;  
    virtual void Stop() = 0;  
    virtual void Alarm() = 0;  
    virtual void EngineBoom() = 0;  
private:  
    vector<string> \* m\_pSequence;  
};

**//CarModel.cpp**

#include "StdAfx.h"  
#include "CarModel.h"  
#include <vector>  
#include <iostream>  
using std::vector;  
using std::string;  
CCarModel::CCarModel(void)  
{  
}  
CCarModel::~CCarModel(void)  
{  
}  
void CCarModel::SetSequence(vector<string> \*pSeq)  
{  
    m\_pSequence = pSeq;  
}  
void CCarModel::Run()  
{  
    vector<string>::const\_iterator it = m\_pSequence->begin();  
    for (; it < m\_pSequence->end(); ++it)  
    {  
        string actionName = \*it;  
        if(actionName.compare("start") == 0)  
        {  
            Start();  
        }  
        else if(actionName.compare("stop") == 0)  
        {  
            Stop();  
        }  
        else if(actionName.compare("alarm") == 0)  
        {  
            Alarm();  
        }  
        else if(actionName.compare("engine boom") == 0)  
        {  
            EngineBoom();  
        }  
    }  
}

**//BenzModel.h**

#pragma once  
#include "carmodel.h"  
class CBenzModel :public CCarModel  
{  
public:  
    CBenzModel(void);  
    ~CBenzModel(void);  
protected:  
    void Start();  
    void Stop();  
    void Alarm();  
    void EngineBoom();  
};

**//BenzModel.cpp**

#include "StdAfx.h"  
#include "BenzModel.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CBenzModel::CBenzModel(void)  
{  
}  
CBenzModel::~CBenzModel(void)  
{  
}  
void CBenzModel::Start()  
{  
    cout << "奔驰发动..." << endl;  
}  
void CBenzModel::Stop()  
{  
    cout << "奔驰停车..." << endl;  
}  
void CBenzModel::Alarm()  
{  
    cout << "奔驰鸣笛" << endl;  
}  
void CBenzModel::EngineBoom()  
{  
    cout << "奔驰引擎声音是这样...." << endl;  
}

**//BMWModel.h**

#pragma once  
#include "carmodel.h"  
class CBMWModel :public CCarModel  
{  
public:  
    CBMWModel(void);  
    ~CBMWModel(void);  
protected:  
    void Start();  
    void Stop();  
    void Alarm();  
    void EngineBoom();  
};

**//BMWModel.cpp**

#include "StdAfx.h"  
#include "BMWModel.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CBMWModel::CBMWModel(void)  
{  
}  
CBMWModel::~CBMWModel(void)  
{  
}  
void CBMWModel::Start()  
{  
    cout << "宝马发动..." << endl;  
}  
void CBMWModel::Stop()  
{  
    cout << "宝马停车..." << endl;  
}  
void CBMWModel::Alarm()  
{  
    cout << "宝马鸣笛" << endl;  
}  
void CBMWModel::EngineBoom()  
{  
    cout << "宝马引擎声音是这样...." << endl;  
}

**//ICarBuilder.h**

#pragma once  
#include "CarModel.h"  
#include <iostream>  
#include <vector>  
using std::string;  
using std::vector;  
class ICarBuilder  
{  
public:  
    ICarBuilder(void)  
    {  
    }  
    virtual ~ICarBuilder(void)  
    {  
    }  
    virtual void SetSequence(vector<string> \*pseq) = 0;  
    virtual CCarModel \* GetCarModel() = 0;  
};  
**//BenzBuilder.h**

#pragma once  
#include "icarbuilder.h"  
#include "CarModel.h"  
#include <iostream>  
#include <vector>  
using std::string;  
using std::vector;  
class CBenzBuilder :public ICarBuilder  
{  
public:  
    CBenzBuilder(void);  
    ~CBenzBuilder(void);  
    void SetSequence(vector<string> \*pSeq);  
    CCarModel \* GetCarModel();  
private:  
    CCarModel \*m\_pBenz;  
};  
**//BenzBuilder.cpp**

#include "StdAfx.h"  
#include "BenzBuilder.h"  
#include "BenzModel.h"  
CBenzBuilder::CBenzBuilder(void)  
{  
    m\_pBenz = new CBenzModel();  
}  
CBenzBuilder::~CBenzBuilder(void)  
{  
    delete m\_pBenz;  
}  
void CBenzBuilder::SetSequence(vector<string> \*pSeq)  
{  
    m\_pBenz->SetSequence(pSeq);  
}  
CCarModel \* CBenzBuilder::GetCarModel()  
{  
    return m\_pBenz;  
}

**//BMWBuilder.h**

#pragma once  
#include "icarbuilder.h"  
#include "CarModel.h"  
#include <iostream>  
#include <vector>  
using std::string;  
using std::vector;  
class CBMWBuilder :  
    public ICarBuilder  
{  
public:  
    CBMWBuilder(void);  
    ~CBMWBuilder(void);  
    void SetSequence(vector<string> \*pSeq);  
    CCarModel \* GetCarModel();  
private:  
    CCarModel \*m\_pBMW;  
};

**//BMWBuilder.cpp**

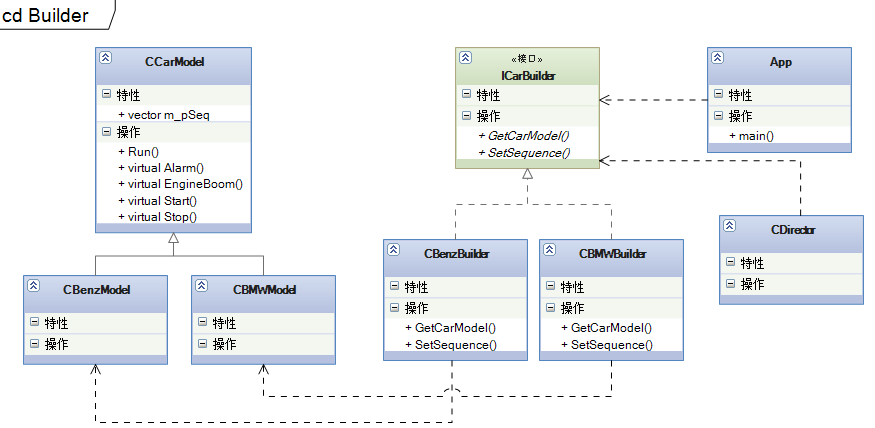
#include "StdAfx.h"  
#include "BMWBuilder.h"  
#include "BMWModel.h"  
CBMWBuilder::CBMWBuilder(void)  
{  
    m\_pBMW = new CBMWModel();  
}  
CBMWBuilder::~CBMWBuilder(void)  
{  
    delete m\_pBMW;  
}  
void CBMWBuilder::SetSequence( vector<string> \*pSeq )  
{  
    m\_pBMW->SetSequence(pSeq);  
}  
CCarModel \* CBMWBuilder::GetCarModel()  
{  
    return m\_pBMW;  
}

**//Director.h**

#pragma once  
#include "BenzModel.h"  
#include "BMWModel.h"  
#include "BenzBuilder.h"  
#include "BMWBuilder.h"  
#include <vector>  
using std::vector;  
class CDirector  
{  
public:  
    CDirector(void);  
    ~CDirector(void);  
    CBenzModel \* GetABenzModel();  
    CBenzModel \* GetBBenzModel();  
    CBMWModel \* GetCBMWModel();  
    CBMWModel \* GetDBMWModel();  
private:  
    vector<string> \* m\_pSeqence;  
    CBenzBuilder \* m\_pBenzBuilder;  
    CBMWBuilder \* m\_pBMWBuilder;  
};

**//Director.cpp**

#include "StdAfx.h"  
#include "Director.h"  
CDirector::CDirector(void)  
{  
    m\_pBenzBuilder = new CBenzBuilder();  
    m\_pBMWBuilder = new CBMWBuilder();  
    m\_pSeqence = new vector<string>();  
}  
CDirector::~CDirector(void)  
{  
    delete m\_pBenzBuilder;  
    delete m\_pBMWBuilder;  
    delete m\_pSeqence;  
}  
CBenzModel \* CDirector::GetABenzModel()  
{  
    m\_pSeqence->clear();  
    m\_pSeqence->push\_back("start");  
    m\_pSeqence->push\_back("stop");  
    m\_pBenzBuilder->SetSequence(m\_pSeqence);  
    return dynamic\_cast<CBenzModel\*>(m\_pBenzBuilder->GetCarModel());  
}  
CBenzModel \* CDirector::GetBBenzModel()  
{  
    m\_pSeqence->clear();  
    m\_pSeqence->push\_back("engine boom");  
    m\_pSeqence->push\_back("start");  
    m\_pSeqence->push\_back("stop");  
    m\_pBenzBuilder->SetSequence(m\_pSeqence);  
    return dynamic\_cast<CBenzModel\*>(m\_pBenzBuilder->GetCarModel());  
}  
CBMWModel \* CDirector::GetCBMWModel()  
{  
    m\_pSeqence->clear();  
    m\_pSeqence->push\_back("alarm");  
    m\_pSeqence->push\_back("start");  
    m\_pSeqence->push\_back("stop");  
    m\_pBMWBuilder->SetSequence(m\_pSeqence);  
    return static\_cast<CBMWModel\*>(m\_pBMWBuilder->GetCarModel());  
}  
CBMWModel \* CDirector::GetDBMWModel()  
{  
    m\_pSeqence->clear();  
    m\_pSeqence->push\_back("start");  
    m\_pBenzBuilder->SetSequence(m\_pSeqence);  
    return dynamic\_cast<CBMWModel\*>(m\_pBMWBuilder->GetCarModel());  
}



建造者模式属于创建型模式，主要关注创建的顺序，不同的顺序，生产的产品略有不同。

[**设计模式C++学习笔记之十一（Bridge桥梁模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/17/2019067.html)

桥梁模式，将抽象部分与它的实现部分分离，使它们都可以独立地变化。实现分离的办法就是增加一个类，

**11.1.解释**

main()，客户

IProduct，产品接口

CHouse，房子

CIPod，ipod

CClothes，服装

CNewCorp，桥梁类，MakeMoney()是桥梁方法

CNewHouseCorp，只能生产房子，所以构造函数是CHouse\*

CShanZhaiCorp，什么赚钱就生产什么，所以构造函数是IProduct\*

说明：客户直接使用CNewHouseCorp和CShanZhaiCorp类，在main()函数里构造产品，然后传到这两个类里。这两个类的MakeMoney()函数，先调用基类的MakeMoney()，然后分别执行各自的逻辑。

注意：CNewCorp起到了桥梁的作用。可以分别增加产品和公司。

看代码：

**//NewCorp.h**

#pragma once  
#include "IProduct.h"  
class CNewCorp  
{  
public:  
    CNewCorp(IProduct \*pproduct);  
    virtual ~CNewCorp(void);  
    void MakeMoney();  
private:  
    IProduct \*m\_pProduct;  
};

**//NewCorp.cpp**

#include "StdAfx.h"  
#include "NewCorp.h"  
CNewCorp::CNewCorp( IProduct \*pproduct )  
{  
    this->m\_pProduct = pproduct;  
}  
CNewCorp::~CNewCorp(void)  
{  
}  
void CNewCorp::MakeMoney()  
{  
    //每个公司都是一样，先生产  
    this->m\_pProduct->BeProducted();  
  
    //然后销售  
    this->m\_pProduct->BeSelled();  
}

**//NewHouseCorp.h**

#pragma once  
#include "newcorp.h"  
#include "House.h"  
class CNewHouseCorp :  
    public CNewCorp  
{  
public:  
    CNewHouseCorp(CHouse \*pHouse);  
    ~CNewHouseCorp(void);  
    void MakeMoney();  
};

**//NewHouseCorp.cpp**

#include "StdAfx.h"  
#include "NewHouseCorp.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CNewHouseCorp::CNewHouseCorp(CHouse \*pHouse) : CNewCorp(pHouse)  
{  
}  
CNewHouseCorp::~CNewHouseCorp(void)  
{  
}  
void CNewHouseCorp::MakeMoney()  
{  
    this->CNewCorp::MakeMoney();  
    cout << "房地产公司赚大钱了..." << endl;  
}

**//ShanZhaiCorp.h**

#pragma once  
#include "newcorp.h"  
#include "IProduct.h"  
class CShanZhaiCorp :  
    public CNewCorp  
{  
public:  
    CShanZhaiCorp(IProduct \*pproduct);  
    ~CShanZhaiCorp(void);  
    void MakeMoney();  
};

**//ShanZhaiCorp.cpp**

#include "StdAfx.h"  
#include "ShanZhaiCorp.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CShanZhaiCorp::CShanZhaiCorp(IProduct \*pproduct) : CNewCorp(pproduct)  
{  
}  
CShanZhaiCorp::~CShanZhaiCorp(void)  
{  
}  
void CShanZhaiCorp::MakeMoney()  
{  
    this->CNewCorp::MakeMoney();  
    cout << "我赚钱呀..." << endl;  
}

**//IProduct.h**

#pragma once  
class IProduct  
{  
public:  
    IProduct(void)  
    {  
    }  
    virtual ~IProduct(void)  
    {  
    }  
    virtual void BeProducted() = 0;  
    virtual void BeSelled() = 0;  
};

**//House.h**

#pragma once  
#include "iproduct.h"  
class CHouse :  
    public IProduct  
{  
public:  
    CHouse(void);  
    ~CHouse(void);  
    void BeProducted();  
    void BeSelled();  
};

**//House.cpp**

#include "StdAfx.h"  
#include "House.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHouse::CHouse(void)  
{  
}  
CHouse::~CHouse(void)  
{  
}  
void CHouse::BeProducted()  
{  
    cout << "生产出的房子是这个样子的..." << endl;  
}  
void CHouse::BeSelled()  
{  
    cout << "生产出的房子卖出去了..." << endl;  
}

**//Clothes.h**

#pragma once  
#include "iproduct.h"  
class CClothes :  
    public IProduct  
{  
public:  
    CClothes(void);  
    ~CClothes(void);  
    void BeProducted();  
    void BeSelled();  
};

**//Clothes.cpp**

#include "StdAfx.h"  
#include "Clothes.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CClothes::CClothes(void)  
{  
}  
CClothes::~CClothes(void)  
{  
}  
void CClothes::BeProducted()  
{  
    cout << "生产出的衣服是这个样子的..." << endl;  
}  
void CClothes::BeSelled()  
{  
    cout << "生产出的衣服卖出去了..." << endl;  
}

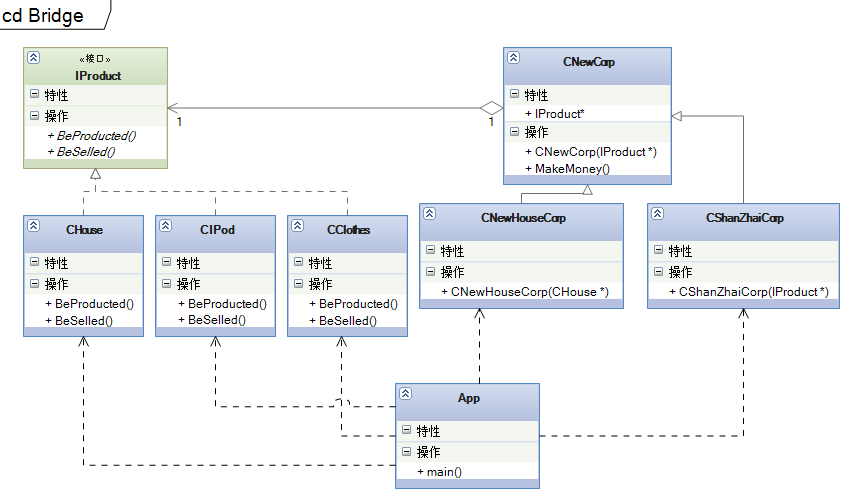
**//IPod.h**

#pragma once  
#include "iproduct.h"  
class CIPod :public IProduct  
{  
public:  
    CIPod(void);  
    ~CIPod(void);  
    void BeProducted();  
    void BeSelled();  
};

**//IPod.cpp**

#include "StdAfx.h"  
#include "IPod.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CIPod::CIPod(void)  
{  
}  
CIPod::~CIPod(void)  
{  
}  
void CIPod::BeProducted()  
{  
    cout << "生产出的ipod是这个样子的..." << endl;  
}  
void CIPod::BeSelled()  
{  
    cout << "生产出的ipod卖出去了..." << endl;  
}

**//**

**// Bridge.cpp : 定义控制台应用程序的入口点。  
//**  
  
#include "stdafx.h"  
#include "ClothesCorp.h"  
#include "NewHouseCorp.h"  
#include "Clothes.h"  
#include "IPod.h"  
#include "ShanZhaiCorp.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
void DoNewRun1()  
{  
    cout << "----------房地产公司是这样运行的----------" << endl;  
    CHouse house;  
    CNewHouseCorp newHouseCorp(&house);  
    newHouseCorp.MakeMoney();  
    cout << endl;  
  
    cout << "----------山寨公司是这样运行的----------" << endl;  
    CClothes clothes;  
    CShanZhaiCorp shanZhaiCorp(&clothes);  
    shanZhaiCorp.MakeMoney();  
    cout << endl;  
}  
  
void DoNewRun2()  
{  
    cout << "----------房地产公司是这样运行的----------" << endl;  
    CHouse house;  
    CNewHouseCorp newHouseCorp(&house);  
    newHouseCorp.MakeMoney();  
    cout << endl;  
  
    cout << "----------山寨公司是这样运行的----------" << endl;  
    CIPod ipod;  
    CShanZhaiCorp shanZhaiCorp(&ipod);  
    shanZhaiCorp.MakeMoney();  
    cout << endl;  
}  
  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //只有两家公司，一家是房地产公司，另一家公司是衣服赚钱就生产衣服  
    DoNewRun1();  
  
    //只有两家公司，一家是房地产公司，另一家公司是ipod赚钱就生产ipod  
    DoNewRun2();  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}  


桥梁模式（又可以称做桥接模式）属于结构型模式。CNewCorp实现了桥梁（桥接）的作用。

[**设计模式C++学习笔记之十二（Command命令模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/18/2020230.html)

命令模式，将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。应该是一个比较简单的模式了。

12.1.解释

main()，客户

CInvoker，命令接收者，如项目经理

IGroup，执行者接口

CRequirementGroup，实际执行者之一

CPageGroup，实际执行者之二

CCodePage，实际执行者之三

ICommand，命令接口

CAddRequirementCommand，Execute函数，将调用CRequirementGroup的多个命令。来组合执行用户发出的命令。

CDeletePageCommand，同上

... ... 其它命令。

说明：客户只需要知道向Invoker发出命令（多个命令），而不是将命令直接传达给具体的执行者。当然，客户是需要知道都有什么命令的。

注意：客户只发命令，不需要知道由谁来执行和怎么执行，体现出高内聚的特点。用户在发出命令后，是允许撤回的，所以可以增加一个命令“Undo ”，Undo是状态的变更。

看代码：

**//Invoker.h**

#pragma once  
#include "ICommand.h"  
class CInvoker  
{  
public:  
    CInvoker(void);  
    ~CInvoker(void);  
    void SetCommand(ICommand \*pcommand);  
    void Action();  
private:  
    ICommand \*m\_pCommand;  
};

**//Invoker.cpp**

#include "StdAfx.h"  
#include "Invoker.h"  
CInvoker::CInvoker(void)  
{  
}  
CInvoker::~CInvoker(void)  
{  
}  
void CInvoker::SetCommand( ICommand \*pcommand )  
{  
    this->m\_pCommand = pcommand;  
}  
void CInvoker::Action()  
{  
    this->m\_pCommand->Execute();  
}

**//IGroup.h**

#pragma once  
class IGroup  
{  
public:  
    IGroup(void)  
    {  
    }  
    virtual ~IGroup(void)  
    {  
    }  
    virtual void Find() = 0;  
    virtual void Add() = 0;  
    virtual void Delete() = 0;  
    virtual void Change() = 0;  
    virtual void Plan() = 0;  
};

**//RequirementGroup.h**

#pragma once  
#include "igroup.h"  
class CRequirementGroup :  
    public IGroup  
{  
public:  
    CRequirementGroup(void);  
    ~CRequirementGroup(void);  
    void Find();  
    void Add();  
    void Delete();  
    void Change();  
    void Plan();  
};

**//RequirementGroup.cpp**

#include "StdAfx.h"  
#include "RequirementGroup.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CRequirementGroup::CRequirementGroup(void)  
{  
}  
CRequirementGroup::~CRequirementGroup(void)  
{  
}  
void CRequirementGroup::Find()  
{  
    cout << "找到需求组..." << endl;  
}  
void CRequirementGroup::Add()  
{  
    cout << "客户要求增加一项需求..." << endl;  
}  
void CRequirementGroup::Delete()  
{  
    cout << "要求删除一项需求..." << endl;  
}  
void CRequirementGroup::Change()  
{  
    cout << "客户要求修改一项需求..." << endl;  
}  
void CRequirementGroup::Plan()  
{  
    cout << "客户要求需求变更计划..." << endl;  
}  
**//PageGroup.h**

#pragma once  
#include "igroup.h"  
class CPageGroup :  
    public IGroup  
{  
public:  
    CPageGroup(void);  
    ~CPageGroup(void);  
    void Find();  
    void Add();  
    void Delete();  
    void Change();  
    void Plan();  
};

**//PageGroup.cpp**

#include "StdAfx.h"  
#include "PageGroup.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CPageGroup::CPageGroup(void)  
{  
}  
CPageGroup::~CPageGroup(void)  
{  
}  
void CPageGroup::Find()  
{  
    cout << "找到美工组..." << endl;  
}  
void CPageGroup::Add()  
{  
    cout << "客户要求增加一个页面..." << endl;  
}  
void CPageGroup::Delete()  
{  
    cout << "客户要求删除一个页面..." << endl;  
}  
void CPageGroup::Change()  
{  
    cout << "客户要求修改一个页面..." << endl;  
}  
void CPageGroup::Plan()  
{  
    cout << "客户要求页面变更计划..." << endl;  
}  
**//CodeGroup.h**

#pragma once  
#include "igroup.h"  
class CCodeGroup :  
    public IGroup  
{  
public:  
    CCodeGroup(void);  
    ~CCodeGroup(void);  
    void Find();  
    void Add();  
    void Delete();  
    void Change();  
    void Plan();  
};

**//CodeGroup.cpp**

#include "StdAfx.h"  
#include "CodeGroup.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CCodeGroup::CCodeGroup(void)  
{  
}  
CCodeGroup::~CCodeGroup(void)  
{  
}  
void CCodeGroup::Find()  
{  
    cout << "找到代码组..." << endl;  
}  
void CCodeGroup::Add()  
{  
    cout << "客户要求增加一项功能..." << endl;  
}  
void CCodeGroup::Delete()  
{  
    cout << "客户要求删除一项功能..." << endl;  
}  
void CCodeGroup::Change()  
{  
    cout << "客户要求修改一项功能..." << endl;  
}  
void CCodeGroup::Plan()  
{  
    cout << "客户要求代码变更计划..." << endl;  
}  
**//ICommand.h**

#pragma once  
#include "RequirementGroup.h"  
#include "PageGroup.h"  
#include "CodeGroup.h"  
class ICommand  
{  
public:  
    ICommand(void)  
    {  
        m\_prg = new CRequirementGroup();  
        m\_ppg = new CPageGroup();  
        m\_pcg = new CCodeGroup();  
    }  
    virtual ~ICommand(void)  
    {  
        delete m\_prg;  
        delete m\_ppg;  
        delete m\_pcg;  
    }  
    virtual void Execute() = 0;  
protected:  
    CRequirementGroup \*m\_prg;  
    CPageGroup \*m\_ppg;  
    CCodeGroup \*m\_pcg;  
};  
**//AddRequirementCommand.h**

#pragma once  
#include "icommand.h"  
class CAddRequirementCommand :  
    public ICommand  
{  
public:  
    CAddRequirementCommand(void);  
    ~CAddRequirementCommand(void);  
    void Execute();  
};

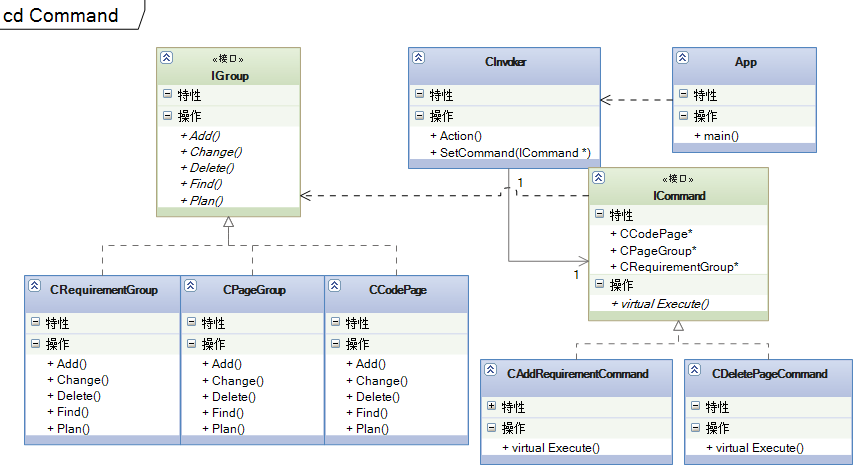
**//AddRequirementCommand.cpp**

#include "StdAfx.h"  
#include "AddRequirementCommand.h"  
CAddRequirementCommand::CAddRequirementCommand(void)  
{  
}  
CAddRequirementCommand::~CAddRequirementCommand(void)  
{  
}  
void CAddRequirementCommand::Execute()  
{  
    //执行增另一项需求的命令  
    this->ICommand::m\_prg->Find();  
  
    //增加一份需求  
    this->ICommand::m\_prg->Add();  
  
    //给出计划  
    this->ICommand::m\_prg->Plan();  
}  
**//DeletePageCommand.h**

#pragma once  
#include "icommand.h"  
class CDeletePageCommand :  
    public ICommand  
{  
public:  
    CDeletePageCommand(void);  
    ~CDeletePageCommand(void);  
    void Execute();  
};  
//DeletePageCommand.cpp

#include "StdAfx.h"  
#include "DeletePageCommand.h"  
CDeletePageCommand::CDeletePageCommand(void)  
{  
}  
CDeletePageCommand::~CDeletePageCommand(void)  
{  
}  
void CDeletePageCommand::Execute()  
{  
    //执行增另一项需求的命令  
    this->ICommand::m\_ppg->Find();  
  
    //增加一份需求  
    this->ICommand::m\_ppg->Delete();  
  
    //给出计划  
    this->ICommand::m\_ppg->Plan();  
}  
//**Command.cpp**

#include "stdafx.h"  
#include "IGroup.h"  
#include "CodeGroup.h"  
#include "PageGroup.h"  
#include "RequirementGroup.h"  
#include "Invoker.h"  
#include "AddRequirementCommand.h"  
#include "DeletePageCommand.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
void DoIt()  
{  
    cout << "----------客户想增加一个需求----------" << endl;  
    IGroup \*rg = new CRequirementGroup();  
    rg->Find();  
    rg->Add();  
    rg->Plan();  
    delete rg;  
    cout << endl;  
  
    cout << "----------客户又想修改一个页面----------" << endl;  
    IGroup \*pg = new CPageGroup();  
    pg->Find();  
    pg->Add();  
    pg->Plan();  
    delete pg;  
    cout << endl;  
  
    cout << "----------客户又想删除一个功能----------" << endl;  
    IGroup \*cg = new CCodeGroup();  
    cg->Find();  
    cg->Add();  
    cg->Plan();  
    delete cg;  
    cout << endl;  
}  
  
void DoNew()  
{  
    cout << "----------客户觉得烦了，希望只找一个人，并告诉他要做什么----------" << endl;  
    cout << "----------客户要求增加一项需求----------" << endl;  
    CInvoker gary;  
    ICommand \*pcommand = new CAddRequirementCommand();  
    gary.SetCommand(pcommand);  
    gary.Action();  
    delete pcommand;  
    cout << endl;  
  
    //客户想要改动只需要找CInvoker就可以了。  
    cout << "----------客户要求删除一个页面----------" << endl;  
    CInvoker ricky;  
    ICommand \*pcommand2 = new CDeletePageCommand();  
    ricky.SetCommand(pcommand2);  
    ricky.Action();  
    delete pcommand2;  
    cout << endl;  
}  
  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //客户原来的运行流程  
    DoIt();  
  
    //客户觉得麻烦了，每次改动都要找不同的组，谈不同的事  
    //客户只想找一个人，告诉他要做什么就可以，不想关心由哪几个组来做和怎么做  
    DoNew();  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



记得曾经给系统中增加Timesheet的小功能，在这里面就用到了命令模式，当时也只是练练手，因为命令模式只适用于变化不是很多的场合，因为一个命令就定义为一个ICommand实现类，这样的话，对ICommand派生类的数量增长可能会难以控制。上图是代码实现命令模式时，用到的相关类图。

学习需要坚持，同时也是痛苦的，谁都想每天下了班，回到家里休息一下，看看电视什么的。但我要坚持下去，我要不停的鼓励自己。并完成自己的学习计划。加油！

[**设计模式C++学习笔记之十三（Decorator装饰模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/18/2020254.html)

装饰模式，动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

**13.1.解释**

main()，老爸

ISchoolReport，成绩单接口

CFourthGradeSchoolReport，四年级成绩单

ReportDecorator，成绩单装饰器基类

HighScoreDecorator，最高分装饰器

SortDecorator，班级排名装饰器

说明：对“四年级成绩单”进行装饰，ReportDecorator必然有一个private变量指向ISchoolReport。

注意：

看代码：

**// Decorator.cpp//主程序**  
#include "stdafx.h"  
#include "ISchoolReport.h"  
#include "FouthGradeSchoolReport.h"  
#include "SugarFouthGradeSchoolReport.h"  
#include "HighScoreDecorator.h"  
#include "SortDecorator.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
void DoIt()  
{  
    ISchoolReport \*psr = new CSugarFouthGradeSchoolReport();  
    psr->Report();//看成绩单  
    psr->Sign("老三");//很开心，就签字了  
    delete psr;  
}  
void DoNew()  
{  
    cout << "----------分部分进行装饰----------" << endl;  
    ISchoolReport \*psr = new CFouthGradeSchoolReport();//原装成绩单  
    //  
    ISchoolReport \*pssr = new CSortDecorator(psr);//又加了成绩排名的说明  
    ISchoolReport \*phsr = new CHighScoreDecorator(pssr);//加了最高分说明的成绩单  
    phsr->Report();//看成绩单  
    phsr->Sign("老三");//很开心，就签字了  
      
    //先装饰哪个不重要，顺序已经在装饰内部确定好，但一定要调用最后一个装饰器的接口。  
    //ISchoolReport \*phsr = new CHighScoreDecorator(psr);//加了最高分说明的成绩单  
    //ISchoolReport \*pssr = new CSortDecorator(phsr);//又加了成绩排名的说明  
    //pssr->Report();//看成绩单  
    //pssr->Sign("老三");//很开心，就签字了  
  
    delete pssr;  
    delete phsr;  
    delete psr;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //在装饰之前，可以用继承的办法，来进行简单的修饰  
    DoIt();  
  
    //但如果需要修饰的项目太多呢？或者装饰的项目不是固定的，继承显然会变得更复杂  
    DoNew();  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}

**//ISchoolReport.h**

#pragma once  
#include <iostream>  
using std::string;  
class ISchoolReport  
{  
public:  
    ISchoolReport(void)  
    {  
    }  
    virtual ~ISchoolReport(void)  
    {  
    }  
    virtual void Report() = 0;  
    virtual void Sign(string name) = 0;  
};

**//FouthGradeSchoolReport.h**

#pragma once  
#include "ischoolreport.h"  
class CFouthGradeSchoolReport :public ISchoolReport  
{  
public:  
    CFouthGradeSchoolReport(void);  
    ~CFouthGradeSchoolReport(void);  
    void Report();  
    void Sign(string name);  
};

**//FouthGradeSchoolReport.cpp**

#include "StdAfx.h"  
#include "FouthGradeSchoolReport.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
CFouthGradeSchoolReport::CFouthGradeSchoolReport(void)  
{  
}  
CFouthGradeSchoolReport::~CFouthGradeSchoolReport(void)  
{  
}  
void CFouthGradeSchoolReport::Report()  
{  
    cout << "尊敬的XXX家长：" << endl;  
    cout << "......" << endl;  
    cout << "语文62  数学65  体育98  自然63" << endl;  
    cout << "......" << endl;  
    cout << "                家长签名：" << endl;  
}  
void CFouthGradeSchoolReport::Sign(string name)  
{  
    cout << "家长签名为：" << name.c\_str() << endl;  
}

**//ReportDecorator.h**

#pragma once  
#include "ischoolreport.h"  
class CReportDecorator :public ISchoolReport  
{  
public:  
    CReportDecorator(ISchoolReport \*psr);  
    virtual ~CReportDecorator(void);  
    void Report();  
    void Sign(string name);  
private:  
    ISchoolReport \*m\_pSchoolReport;  
};

**//ReportDecorator.cpp**

#include "StdAfx.h"  
#include "ReportDecorator.h"  
#include <iostream>  
using std::string;  
CReportDecorator::CReportDecorator(ISchoolReport \*psr)  
{  
    this->m\_pSchoolReport = psr;  
}  
CReportDecorator::~CReportDecorator(void)  
{  
}  
void CReportDecorator::Report()  
{  
    this->m\_pSchoolReport->Report();  
}  
void CReportDecorator::Sign( string name )  
{  
    this->m\_pSchoolReport->Sign(name);  
}

**//HighScoreDecorator.h**

#pragma once  
#include "reportdecorator.h"  
#include "ISchoolReport.h"  
class CHighScoreDecorator :public CReportDecorator  
{  
public:  
    CHighScoreDecorator(ISchoolReport \*psr);  
    ~CHighScoreDecorator(void);  
    void Report();  
private:  
    void ReportHighScore();  
};

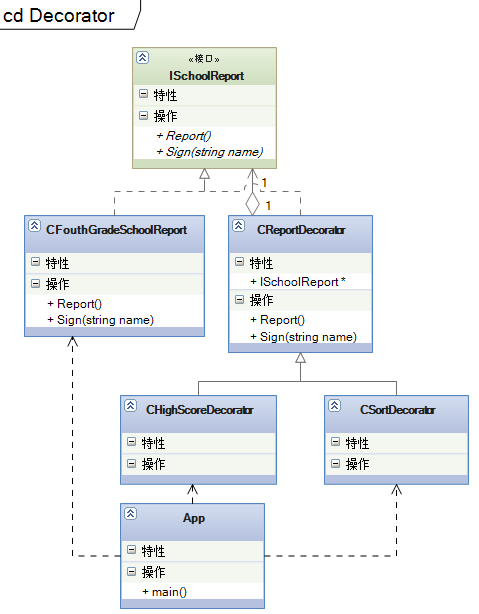
**//HighScoreDecorator.cpp**

#include "StdAfx.h"  
#include "HighScoreDecorator.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHighScoreDecorator::CHighScoreDecorator( ISchoolReport \*psr ) : CReportDecorator(psr)  
{  
}  
CHighScoreDecorator::~CHighScoreDecorator(void)  
{  
}  
void CHighScoreDecorator::Report()  
{  
    this->ReportHighScore();  
    this->CReportDecorator::Report();  
}  
void CHighScoreDecorator::ReportHighScore()  
{  
    cout << "这次考试语文最高是75， 数学是78， 自然是80" << endl;  
}

**//SortDecorator.h**

#pragma once  
#include "reportdecorator.h"  
#include "ISchoolReport.h"  
class CSortDecorator :public CReportDecorator  
{  
public:  
    CSortDecorator(ISchoolReport \*psr);  
    ~CSortDecorator(void);  
    void Report();  
private:  
    void ReportSort();  
};  
**//SortDecorator.cpp**

#include "StdAfx.h"  
#include "SortDecorator.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CSortDecorator::CSortDecorator( ISchoolReport \*psr ) : CReportDecorator(psr)  
{  
}  
CSortDecorator::~CSortDecorator(void)  
{  
}  
void CSortDecorator::ReportSort()  
{  
    cout << "我是排名第38名..." << endl;  
}  
void CSortDecorator::Report()  
{  
    this->CReportDecorator::Report();  
    this->ReportSort();  
}



这也是一个比较简单的模式，属于行为型模式。

[**设计模式C++学习笔记之十四（Iterator迭代器模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/19/2021600.html)

**14.1.解释**

概念：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

main()，客户

IProject，产品接口

CProject，产品类

IIterator，迭代器接口

IProjectIterator，产品迭代器接口

CProjectIterator，产品迭代器实现类

说明：CProject实现产品类，能够返回一个迭代器的指针。这个迭代器将封装产品类里的一个数组。所以迭代器在运行Next函数时，可以遍历这个数组的所有元素。

简单来说，就是用代码实现vector<int>::iterator或vector<int>::const\_iterator。

我们来看代码：

**//IProject.h**

#pragma once  
#include "IProjectIterator.h"  
#include <iostream>  
using std::string;  
class IProject  
{  
public:  
    IProject(void)  
    {  
    }  
    virtual ~IProject(void)  
    {  
    }  
    virtual void Add(string name, int num, int cost) = 0;  
    virtual string GetProjectInfo() = 0;  
    virtual IProjectIterator\* GetIterator() = 0;  
    virtual void Erase() = 0;  
};

**//Project.h**

#pragma once  
#include "iproject.h"  
#include "IProjectIterator.h"  
#include <iostream>  
#include <vector>  
using std::string;  
using std::vector;  
class CProject :public IProject  
{  
public:  
    CProject(void);  
    CProject(string name, int num, int cost);  
    ~CProject(void);  
    string GetProjectInfo();  
    void Add(string name, int num, int cost);  
    IProjectIterator \* GetIterator();  
    void Erase();  
private:  
    string m\_name;  
    int m\_num;  
    int m\_cost;  
    vector<IProject\*> m\_projectList;  
};

**//Project.cpp**

#include "StdAfx.h"  
#include "Project.h"  
#include "..\CommonDeclare\Convert.h"  
#include "ProjectIterator.h"  
#include <iostream>  
#include <vector>  
using std::string;  
using std::vector;  
CProject::CProject( void )  
{  
    m\_name = "";  
    m\_num = 0;  
    m\_cost = 0;  
}  
CProject::CProject(string name, int num, int cost) :m\_name(name), m\_num(num), m\_cost(cost)  
{  
}  
CProject::~CProject(void)  
{  
}  
string CProject::GetProjectInfo()  
{  
    string info = "";  
    info.append("项目名称是：");  
    info.append(this->m\_name);  
    info.append("\t项目人数：");  
    info.append(CConvert::ToString(m\_num));  
    info.append("\t项目费用：");  
    info.append(CConvert::ToString(m\_cost));  
    return info;  
}  
void CProject::Add( string name, int num, int cost )  
{  
    this->m\_projectList.push\_back(new CProject(name, num, cost));  
}  
IProjectIterator \* CProject::GetIterator()  
{  
    return new CProjectIterator(this->m\_projectList);  
}  
void CProject::Erase()  
{  
    vector<IProject\*>::reverse\_iterator projectDelIt = m\_projectList.rbegin();  
    for (; projectDelIt != m\_projectList.rend(); projectDelIt++)  
    {  
        delete (\*projectDelIt);  
        (\*projectDelIt) = NULL;  
    }  
    m\_projectList.clear();  
}

**//IIterator.h**

#pragma once  
class IProject;  
class IIterator  
{  
public:  
    IIterator(void)  
    {  
    }  
    virtual ~IIterator(void)  
    {  
    }  
    virtual bool HasNext() = 0;  
    virtual IProject \* Next() = 0;  
};

**//IProjectIterator.h**

#pragma once  
#include "iiterator.h"  
class IProject;  
class IProjectIterator :  
    public IIterator  
{  
public:  
    IProjectIterator(void)  
    {  
    }  
    virtual ~IProjectIterator(void)  
    {  
    }  
    virtual bool HasNext() = 0;  
    virtual IProject \* Next() = 0;  
};

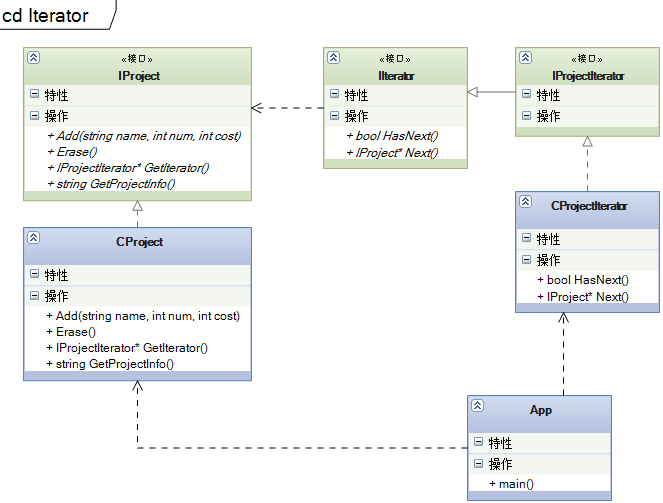
**//ProjectIterator.h**

#pragma once  
#include "iprojectiterator.h"  
#include "IProject.h"  
#include <vector>  
using std::vector;  
class CProjectIterator :public IProjectIterator  
{  
public:  
    CProjectIterator(vector<IProject \*> pl);  
    ~CProjectIterator(void);  
    bool HasNext();  
    IProject \* Next();  
private:  
    vector<IProject \*> m\_projectList;  
    size\_t m\_currentItem;  
};

**//ProjectIterator.cpp**

#include "StdAfx.h"  
#include "ProjectIterator.h"  
CProjectIterator::CProjectIterator(vector<IProject \*> pl) : m\_projectList(pl)  
{  
    m\_currentItem = 0;  
}  
CProjectIterator::~CProjectIterator(void)  
{  
}  
bool CProjectIterator::HasNext()  
{  
    bool b = true;  
    if (m\_currentItem >= m\_projectList.size())  
        b = false;  
    return b;  
}  
IProject \* CProjectIterator::Next()  
{  
    IProject \*pp = m\_projectList.at(m\_currentItem ++);  
    return pp;  
}

**// Iterator.cpp**  
#include "stdafx.h"  
#include "IProject.h"  
#include "Project.h"  
#include "..\CommonDeclare\Convert.h"  
#include "ProjectIterator.h"  
#include <iostream>  
#include <vector>  
using std::vector;  
using std::cout;  
using std::endl;  
void DoIt()  
{  
    cout << "----------未使用迭代模式----------" << endl;  
    vector<IProject\*> projectList;  
  
    projectList.push\_back(new CProject("星球大战项目", 10, 100000));  
    projectList.push\_back(new CProject("扭转时空项目", 100, 10000000));  
    projectList.push\_back(new CProject("超人改造项目", 10000, 1000000000));

    for (int i = 4; i < 6; i ++)  
    {  
        string name = "";  
        name.append("第");  
        name.append(CConvert::ToString(i));  
        name.append("个项目");  
        projectList.push\_back(new CProject(name, i \* 5, i \* 1000000));  
    }  
  
    vector<IProject\*>::const\_iterator projectIt = projectList.begin();  
    for (; projectIt != projectList.end(); projectIt++)  
        cout << (\*projectIt)->GetProjectInfo().c\_str() << endl;  
  
    vector<IProject\*>::reverse\_iterator projectDelIt = projectList.rbegin();  
    for (; projectDelIt != projectList.rend(); projectDelIt++)  
    {  
        delete (\*projectDelIt);  
        (\*projectDelIt) = NULL;  
    }  
    projectList.clear();  
}  
void DoNew()  
{  
    cout << "----------使用迭代模式----------" << endl;  
    IProject \*pproject = new CProject();  
    pproject->Add("星球大战项目", 10, 100000);  
    pproject->Add("扭转时空项目", 100, 10000000);  
    pproject->Add("超人改造项目", 10000, 1000000000);  
  
    for (int i = 4; i < 6; i ++)  
    {  
        string name = "";  
        name.append("第");  
        name.append(CConvert::ToString(i));  
        name.append("个项目");  
        pproject->Add(name, i \* 5, i \* 1000000);  
    }  
  
    IProjectIterator \*pprojectIt = pproject->GetIterator();  
    while(pprojectIt->HasNext())  
    {  
        IProject \*p = dynamic\_cast<IProject\*>(pprojectIt->Next());  
        cout << p->GetProjectInfo().c\_str() << endl;  
    }  
    delete pprojectIt;  
    pprojectIt = NULL;  
    pproject->Erase();  
    delete pproject;  
    pproject = NULL;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //使用Iterator模式之前  
    DoIt();  
  
    //使用Iterator  
    DoNew();  
  
   \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF\_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}  


这个模式，可能有一点绕，需要再仔细的思考一番。本来希望把模式的工作方式，用最简单易懂的语言来总结，但其实这个总结更难。简单的模式还容易些理解，也容易总结，复杂的就难了。打算在之后，学习数据结构的过程中，都总结出这样一句话来。很期待后面对数据结构的学习，发现自己越来越爱学习了。

[**设计模式C++学习笔记之十五（Composite组合模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/19/2021638.html)

**15.1.解释**

概念：将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得用户对单个对象和组合的使用具有一致性。

main()，客户

CCorpNode，抽象基类，实现基本信息

CBranchNode，树枝节点，实现Addordinate()函数和GetSubordinate()函数

CLeafNode，叶子节点，IsLeaf属性总是“true”

说明：组合模式主要是实现在CBranchNode对象里增加对其它对象的数组，如vector<CCorpNode\*>，数组里可以存放CBranchNode和CLeafNode对象。这样方便进行遍历操作。

注意：组合模式有透明组合模式和安全组合模式。透明组合模式是将Addordinate和GetSubordinate这两个函数也抽象到CCorpNode基类里，这增加了操作叶子节点的难度，更易出现逻辑问题。所以尽量使用安全模式。

这个简单了，可以想像一下TreeView和TreeNode基本上是这个意思了，将很多数据组织在一块。

看代码：

//CorpNode.h

#pragma once  
#include <iostream>  
using std::string;  
class CCorpNode  
{  
public:  
    CCorpNode();  
    CCorpNode(string \_name, string \_pos, int \_salary);  
    virtual ~CCorpNode(void);  
    virtual string GetInfo();  
    void SetParent(CCorpNode \*\_pParent);  
    CCorpNode \* GetParent();  
    virtual bool IsLeaf() = 0;  
private:  
    string m\_name;  
    string m\_position;  
    int m\_salary;  
protected:  
    bool m\_isLeaf;  
    CCorpNode \*m\_pParent;  
};

//CorpNode.cpp

#include "StdAfx.h"  
#include "CorpNode.h"  
#include "..\CommonDeclare\Convert.h"  
CCorpNode::CCorpNode(void)  
{  
    m\_name = "";  
    m\_position = "";  
    m\_salary = 0;  
}  
CCorpNode::CCorpNode(string \_name, string \_pos, int \_salary) : m\_name(\_name), m\_position(\_pos), m\_salary(\_salary)  
{  
}  
CCorpNode::~CCorpNode(void)  
{  
}  
string CCorpNode::GetInfo()  
{  
    string info = "";  
    info.append("姓名：");  
    info.append(this->m\_name);  
    info.append("\t职位：");  
    info.append(this->m\_position);  
    info.append("\t薪水：");  
    info.append(CConvert::ToString(this->m\_salary));  
    return info;  
}  
void CCorpNode::SetParent( CCorpNode \*\_parent )  
{  
    this->m\_pParent = \_parent;  
}  
CCorpNode \* CCorpNode::GetParent()  
{  
    return this->m\_pParent;  
}

//BranchNode.h

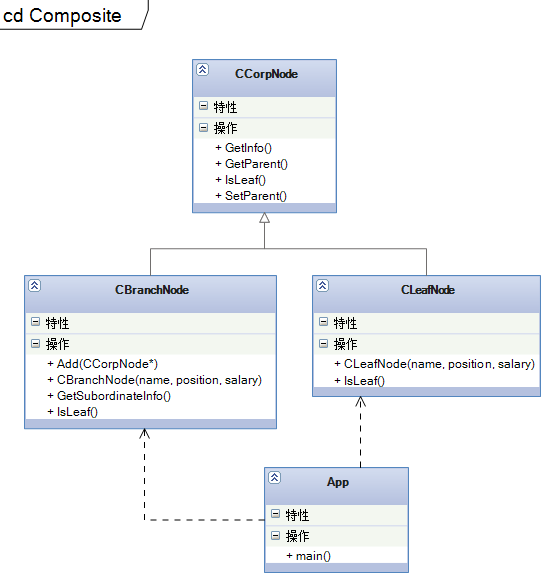
#pragma once  
#include "corpnode.h"  
#include "CorpNode.h"  
#include <vector>  
#include <iostream>  
using std::vector;  
using std::string;  
class CBranchNode :public CCorpNode  
{  
public:  
    CBranchNode(void);  
    CBranchNode(string name, string pos, int salary);  
    ~CBranchNode(void);  
    void Add(CCorpNode \*pcorpNode);  
    vector<CCorpNode\*> GetSubordinateInfo();  
    bool IsLeaf();  
private:  
    vector<CCorpNode\*> m\_subordinateList;  
};  
**//BranchNode.cpp**

#include "StdAfx.h"  
#include "BranchNode.h"  
CBranchNode::CBranchNode(void)  
{  
    m\_isLeaf = false;  
}  
CBranchNode::CBranchNode( string name, string pos, int salary ) : CCorpNode(name, pos, salary)  
{  
    m\_isLeaf = false;  
}  
CBranchNode::~CBranchNode(void)  
{  
}  
void CBranchNode::Add( CCorpNode \*pcorpNode )  
{  
    pcorpNode->SetParent(this);  
    m\_subordinateList.push\_back(pcorpNode);  
}  
vector<CCorpNode\*> CBranchNode::GetSubordinateInfo()  
{  
    return this->m\_subordinateList;  
}  
bool CBranchNode::IsLeaf()  
{  
    return m\_isLeaf;  
}

**//LeafNode.h**

#pragma once  
#include "corpnode.h"  
class CLeafNode :public CCorpNode  
{  
public:  
    CLeafNode(void);  
    CLeafNode(string name, string pos, int salary);  
    ~CLeafNode(void);  
    bool IsLeaf();  
};  
**//LeafNode.cpp**

#include "StdAfx.h"  
#include "LeafNode.h"  
CLeafNode::CLeafNode(void)  
{  
    m\_isLeaf = true;  
}  
CLeafNode::CLeafNode( string name, string pos, int salary ) : CCorpNode(name, pos, salary)  
{  
    m\_isLeaf = true;  
}  
CLeafNode::~CLeafNode(void)  
{  
}  
bool CLeafNode::IsLeaf()  
{  
    return m\_isLeaf;  
}



曾经开发过一款Gantt图的控件，采用的就是这种模式，有GanttView和GanttNode两个类，IGanttNode一个接口，GanttScale标尺类。GanttView负责显示，GanttNode是实现类，只有一个GanttNode类来实现数据的组合，IGanttNode指示了一个IGanttNodeCollection属性来记录所有下级结点的集合。算是实践中的一个应用吧。

[**设计模式C++学习笔记之十六（Observer观察者模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/20/2022975.html)

**16.1.解释**

概念：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

main()，

IObservable，被观察者接口

CHanFeiZiObservable，被观察者韩非子

IObserver，观察者接口

CLiSiObserver，观察者李斯

CZhouSiObserver观察者周斯

说明：将观察者聚集到被观察者韩非子身边，韩非子的每一个举动都会通知给观察者，如李斯或周斯。

注意：最多允许一个对象既是观察者也是被观察者。就像数据库中的触发器一样，成为一个复杂的链就很难维护了。观察者类似于委托的处理方式。

**//IObservable.h**

#pragma once  
#include "IObserver.h"  
#include <iostream>  
using std::string;  
class IObservable  
{  
public:  
    IObservable(void)  
    {  
    }  
    virtual ~IObservable(void)  
    {  
    }  
    virtual void AddObserver(IObserver \*pObserver) = 0;  
    virtual void DeleteObserver(IObserver \*pObserver) = 0;  
    virtual void NotifyObservers(string context) = 0;  
};

**//HanFeiziObservable.h**

#pragma once  
#include "iobservable.h"  
#include "IObserver.h"  
#include <vector>  
using std::vector;  
class CHanFeiziObservable :public IObservable  
{  
public:  
    CHanFeiziObservable(void);  
    ~CHanFeiziObservable(void);  
    void AddObserver(IObserver \*pObserver);  
    void DeleteObserver(IObserver \*pObserver);  
    void NotifyObservers(string context);  
    void HaveBreakfast();  
    void HaveFun();  
private:  
    vector<IObserver\*> m\_observerList;  
    typedef vector<IObserver\*>::const\_iterator ObserverList\_C\_iterator;  
};

**//HanFeiziObservable.cpp**

#include "StdAfx.h"  
#include "HanFeiziObservable.h"  
#include <iostream>  
using std::string;  
using std::cout;  
using std::endl;  
CHanFeiziObservable::CHanFeiziObservable(void)  
{  
}  
CHanFeiziObservable::~CHanFeiziObservable(void)  
{  
}  
void CHanFeiziObservable::AddObserver( IObserver \*pObserver )  
{  
    m\_observerList.push\_back(pObserver);  
}  
void CHanFeiziObservable::DeleteObserver( IObserver \*pObserver )  
{  
    ObserverList\_C\_iterator it = m\_observerList.begin();  
    for (; it != m\_observerList.end(); it++)  
    {  
        string name = (\*it)->GetName();  
        if (name.compare(pObserver->GetName()) == 0)  
        {  
            //找到了删除。  
        }  
    }  
}  
void CHanFeiziObservable::NotifyObservers( string context )  
{  
    ObserverList\_C\_iterator it = m\_observerList.begin();  
    for (; it != m\_observerList.end(); it ++)  
    {  
        (\*it)->Update(context);  
    }  
}  
void CHanFeiziObservable::HaveBreakfast()  
{  
    cout << "韩非子：开始吃饭了..." << endl;  
  
    this->NotifyObservers("韩非子在吃饭");  
}  
void CHanFeiziObservable::HaveFun()  
{  
    cout << "韩非子：开始娱乐了..." << endl;  
  
    this->NotifyObservers("韩非子在娱乐");  
}  
**//IObserver.h**

#pragma once  
#include <iostream>  
using std::string;  
class IObserver  
{  
public:  
    IObserver(string \_name)  
    {  
        this->m\_name = \_name;  
    }  
    virtual ~IObserver(void)  
    {  
    }  
    virtual void Update(string context) = 0;  
    virtual string GetName() = 0;//为c++单独增加的函数，用于删除时查找观察者。  
protected:  
    string m\_name;  
};

**//LiSiObserver.h**

#pragma once  
#include "iobserver.h"  
#include <iostream>  
using std::string;  
class CLiSiObserver :  
    public IObserver  
{  
public:  
    CLiSiObserver(void);  
    ~CLiSiObserver(void);  
    void Update(string context);  
    string GetName();  
private:  
    void ReportToQinShiHuang(string report);  
};

**//LiSiObserver.cpp**

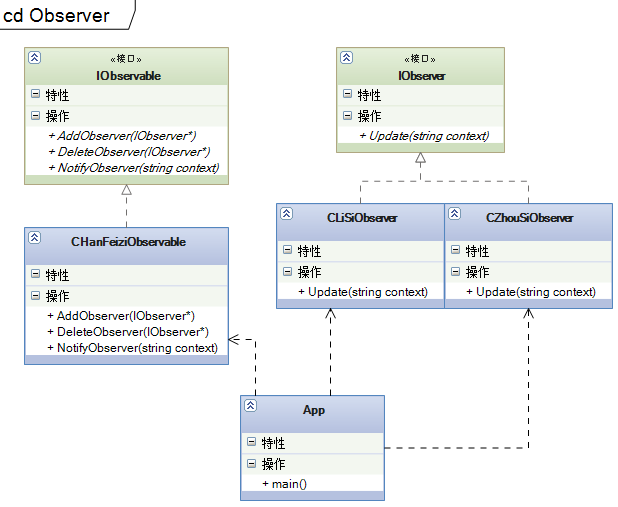
#include "StdAfx.h"  
#include "LiSiObserver.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
CLiSiObserver::CLiSiObserver(void) : IObserver("李斯")  
{  
}  
CLiSiObserver::~CLiSiObserver(void)  
{  
}  
void CLiSiObserver::Update( string context )  
{  
    cout << "李斯：观察到韩非子活动，开始向老板汇报了..." << endl;  
    this->ReportToQinShiHuang(context);  
    cout << "李斯：汇报完毕，秦老板赏给他两个萝卜吃吃..." << endl;  
}  
void CLiSiObserver::ReportToQinShiHuang( string report )  
{  
    cout << "李斯：报告，秦老板！韩非子有活动了--->" << report.c\_str() << endl;  
}  
string CLiSiObserver::GetName()  
{  
    return m\_name;  
}  
**//ZhouSiObserver.h**

#pragma once  
#include "iobserver.h"  
#include <iostream>  
using std::string;  
class CZhouSiObserver :public IObserver  
{  
public:  
    CZhouSiObserver(void);  
    ~CZhouSiObserver(void);  
    void Update(string context);  
    string GetName();  
private:  
    void Cry(string report);  
};

**//ZhouSiObserver.cpp**

#include "StdAfx.h"  
#include "ZhouSiObserver.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
CZhouSiObserver::CZhouSiObserver(void) : IObserver("周斯")  
{  
}  
CZhouSiObserver::~CZhouSiObserver(void)  
{  
}  
void CZhouSiObserver::Update( string context )  
{  
    cout << "周斯：观察到韩非子活动，自己也开始活动了..." << endl;  
    this->Cry(context);  
    cout << "周斯：真真的哭列了..." << endl;  
}  
void CZhouSiObserver::Cry( string report )  
{  
    cout << "周斯：为因" << report.c\_str() << ", ————所以我悲伤呀！" << endl;  
}  
string CZhouSiObserver::GetName()  
{  
    return m\_name;  
}

**// Observer.cpp**  
#include "stdafx.h"  
#include "HanFeiZi.h"  
#include "LiSi.h"  
#include "HanFeiZiNew.h"  
#include "HanFeiziObservable.h"  
#include "LiSiObserver.h"  
#include "ZhouSiObserver.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
using std::string;  
  
void DoNew()  
{  
    //IHanFeiZi.h, HanFeiZiNew.h, ILiSi.h, LiSi.h  
   // cout << "----------用新的方法试试----------" << endl;  
  
    //CHanFeiZiNew hanfeizi;  
  
    //hanfeizi.HaveBreakfast();  
  
    //hanfeizi.HaveFun();  
}  
  
  
void DoNewNew()  
{  
    //IObservable.h, HanfeiziObservable.h, IObserver.h, LiSiObserver.h  
    cout << "----------用更新的方法再试试----------" << endl;  
    IObserver \*pLiSi = new CLiSiObserver();  
    IObserver \*pZhouSi = new CZhouSiObserver();  
  
    CHanFeiziObservable \*pHanFeiZi = new CHanFeiziObservable();  
  
    pHanFeiZi->AddObserver(pLiSi);  
    pHanFeiZi->AddObserver(pZhouSi);  
    pHanFeiZi->HaveBreakfast();  
  
    delete pLiSi;  
    pLiSi = NULL;  
    delete pHanFeiZi;  
    pHanFeiZi = NULL;  
}  
  
  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //比较原始的方法，用线程来观察。  
    //DoIt();  
  
    //把李斯这个类聚集到韩非子这个类上，这样的话耦合度太高了，还是用更抽象的方式。  
    DoNew();  
  
    //更抽象的方式，想要观察韩非子的人多了去了，不可能只允许李斯观察。  
    DoNewNew();  
  
  
    \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}



观察者模式属于行为型模式。

[**设计模式C++学习笔记之十七（Chain of Responsibility责任链模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/20/2022988.html)

**17.1.解释**

概念：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

main()，客户

IWomen，发出请求者接口

CWomen，请求者实现类

CHandler，处理请求抽象类

CFather，处理请求实现类，责任链之一

CHusband，处理请求实现类，责任链之二

CSon，处理请求实现类，责任链之三

说明：CHandler抽象类负责聚合责任链之中的其它处理对象，用SetNext来建立这个责任链。HandleMessage在处理请求时，会判断是否是自己要处理的请求，如果是则直接处理。如果不是，则查找下一个责任链上的处理对象，找到了则由下一个处理。

**//IWomen.h**

#pragma once  
#include <iostream>  
using std::string;  
class IWomen  
{  
public:  
    IWomen(void)  
    {  
    }  
    virtual ~IWomen(void)  
    {  
    }  
    virtual int GetType() = 0;  
    virtual string GetRequest() = 0;  
};

**//Women.h**

#pragma once  
#include "iwomen.h"  
#include <iostream>  
using std::string;  
class CWomen :  
    public IWomen  
{  
public:  
    CWomen(int \_type, string \_request);  
    ~CWomen(void);  
    int GetType();  
    string GetRequest();  
private:  
    int m\_type;  
    string m\_request;  
};

**//Women.cpp**

#include "StdAfx.h"  
#include "Women.h"  
CWomen::CWomen( int \_type, string \_request )  
{  
    this->m\_type = \_type;  
    switch (this->m\_type)  
    {  
    case 1:  
        this->m\_request.append("女儿的请求是：");  
        this->m\_request.append(\_request);  
        break;  
    case 2:  
        this->m\_request.append("妻子的请求是：");  
        this->m\_request.append(\_request);  
        break;  
    case 3:  
        this->m\_request.append("母亲的请求是：");  
        this->m\_request.append(\_request);  
        break;  
    }  
}  
CWomen::~CWomen(void)  
{  
}  
int CWomen::GetType()  
{  
    return m\_type;  
}  
string CWomen::GetRequest()  
{  
    return m\_request;  
}  
**//Handler.h**

#pragma once  
#include "IWomen.h"  
class CHandler  
{  
public:  
    CHandler(int \_level);  
    virtual ~CHandler(void);  
    void HandleMessage(IWomen \*pwomen);  
    void SetNext(CHandler \*phandler);  
    virtual void Response(IWomen \*pwomen) = 0;  
private:  
    int m\_level;  
    CHandler \*m\_pNextHandler;  
};

**//Handler.cpp**

#include "StdAfx.h"  
#include "Handler.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHandler::CHandler(int \_level ) : m\_level(\_level)  
{  
    m\_pNextHandler = NULL;  
}  
CHandler::~CHandler(void)  
{  
}  
void CHandler::HandleMessage( IWomen \*pwomen )  
{  
    if (pwomen->GetType() == this->m\_level)  
    {  
        this->Response(pwomen);  
    }  
    else  
    {  
        if(this->m\_pNextHandler != NULL)  
            this->m\_pNextHandler->HandleMessage(pwomen);  
        else  
            cout << "----------没地方请示了，不做处理！----------" << endl;  
    }  
}  
void CHandler::SetNext( CHandler \*phandler )  
{  
    m\_pNextHandler = phandler;  
}

**//Father.h**

#pragma once  
#include "handler.h"  
#include "IWomen.h"  
class CFather :public CHandler  
{  
public:  
    CFather(void);  
    ~CFather(void);  
    void Response(IWomen \*pwomen);  
};

**//Father.cpp**

#include "StdAfx.h"  
#include "Father.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CFather::CFather(void) : CHandler(1)  
{  
}  
CFather::~CFather(void)  
{  
}  
void CFather::Response( IWomen \*pwomen )  
{  
    cout << "女儿向父亲请示：" << endl;  
    cout << pwomen->GetRequest().c\_str() << endl;  
    cout << "父亲的答复是：同意" << endl;  
}

**//Husband.h**

#pragma once  
#include "handler.h"  
#include "IWomen.h"  
class CHusband :public CHandler  
{  
public:  
    CHusband(void);  
    ~CHusband(void);  
    void Response(IWomen \*pwomen);  
};

**//Husband.cpp**

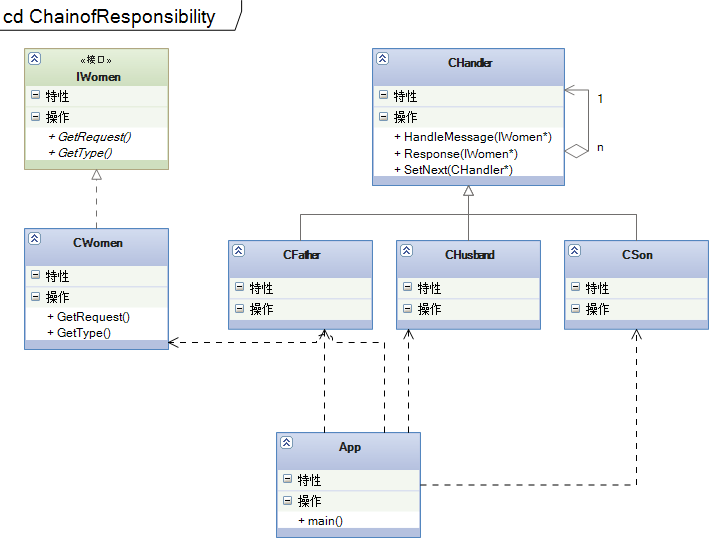
#include "StdAfx.h"  
#include "Husband.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CHusband::CHusband(void) : CHandler(2)  
{  
}  
CHusband::~CHusband(void)  
{  
}  
void CHusband::Response( IWomen \*pwomen )  
{  
    cout << "妻子向丈夫请示：" << endl;  
    cout << pwomen->GetRequest().c\_str() << endl;  
    cout << "丈夫的答复是：同意" << endl;  
}

**//Son.h**

#pragma once  
#include "handler.h"  
#include "IWomen.h"  
class CSon :public CHandler  
{  
public:  
    CSon(void);  
    ~CSon(void);  
    void Response(IWomen \*pwomen);  
};

**//Son.cpp**

#include "StdAfx.h"  
#include "Son.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
CSon::CSon(void) : CHandler(3)  
{  
}  
CSon::~CSon(void)  
{  
}  
void CSon::Response( IWomen \*pwomen )  
{  
    cout << "母亲向儿子请示：" << endl;  
    cout << pwomen->GetRequest().c\_str() << endl;  
    cout << "儿子的答复是：同意" << endl;  
}

**// ChainofResponsibility.cpp**  
  
#include "stdafx.h"  
#include "INormalWomen.h"  
#include "NormalWomen.h"  
#include "INormalHandler.h"  
#include "NormalFather.h"  
#include "NormalHusband.h"  
#include "NormalSon.h"  
  
#include "IWomen.h"  
#include "Women.h"  
#include "Handler.h"  
#include "Father.h"  
#include "Husband.h"  
#include "Son.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
void DoIt()  
{  
    cout << "----------原来的处理方式----------" << endl;  
    //INormalWomen.h, NormalWomen.h, INormalHandler.h, NormalFather.h, NormalHusband.h, NormalSon.h  
    INormalWomen \*pwomen = new CNormalWomen(1, "我要出去逛街");  
    INormalHandler \*pfather = new CNormalFather();  
    INormalHandler \*phusband = new CNormalHusband();  
    INormalHandler \*pson = new CNormalSon();  
    if (pwomen->GetType() == 1)  
    {  
        cout << "女儿向父亲请示：" << endl;  
        pfather->HandleMessage(pwomen);  
    }  
    else if (pwomen->GetType() == 2)  
    {  
        cout << "妻子向丈夫请示：" << endl;  
        phusband->HandleMessage(pwomen);  
    }  
    else if (pwomen->GetType() == 3)  
    {  
        cout << "母亲向儿子请示：" << endl;  
        pson->HandleMessage(pwomen);  
    }  
    else  
    {  
        //什么也不做  
    }  
    delete pwomen;  
    delete pfather;  
    delete phusband;  
    delete pson;  
}  
void DoNew()  
{  
    cout << "----------使用模式后的处理方式----------" << endl;  
    //Handler.h, Handler.cpp, IWomen.h, Women.h, Women.cpp, Father.h, Father,cpp, Husband.h, Husband.cpp, Son.h, Son.cpp  
    IWomen \*pwomen1 = new CWomen(1, "我要出去逛街");  
    IWomen \*pwomen2 = new CWomen(2, "我要出去吃饭");  
    IWomen \*pwomen3 = new CWomen(3, "我也要出去吃饭");  
    IWomen \*pwomen4 = new CWomen(4, "我也要出去逛街");  
  
    CHandler \*pfather = new CFather();  
    CHandler \*phusband = new CHusband();  
    CHandler \*pson = new CSon();  
  
    pfather->SetNext(phusband);  
    phusband->SetNext(pson);  
  
    pfather->HandleMessage(pwomen1);  
    pfather->HandleMessage(pwomen2);  
    pfather->HandleMessage(pwomen3);  
    pfather->HandleMessage(pwomen4);  
  
    delete pfather;  
    delete phusband;  
    delete pson;  
    delete pwomen1;  
    delete pwomen2;  
    delete pwomen3;  
    delete pwomen4;  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    //反面  
    DoIt();  
    //要实现逻辑判断，即女性的请求先发送到父亲类，父亲类一看是自己要处理的，就回应进行处理。如果女儿已经出嫁了，那就把这个请求转发到女婿类来处理。依此类推，形成了一个责任链。  
    DoNew();  
    return 0;  
}  


[**设计模式C++学习笔记之十八（Visitor访问者模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/21/2024099.html)

18.1.解释

概念：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

main()，客户

IVisitor，访问者接口

CBaseVisitor，访问者实现类

CEmployee，被访问者抽象类

CCommonEmployee，被访问者实现类之一

CManager，被访问者实现类之二

说明：A接受B的访问，B主动的执行访问动作。

注意：和观察者的区别是，被观察者要执行一个动作，然后主动发送通知给观察者。访问者模式是由访问者主动发出的动作。

看代码：

[复制代码](javascript:void(0);)

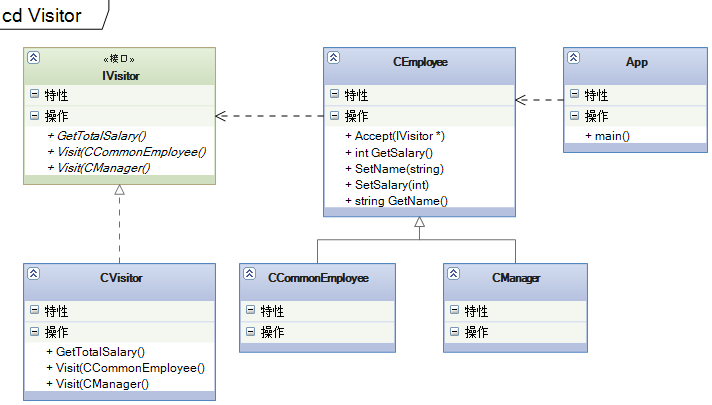
**//IVisitor.h**  
#pragma once  
  
class CCommonEmployee;  
class CManager;  
class IVisitor  
{  
public:  
  
 IVisitor(void)  
 {  
 }  
  
 virtual ~IVisitor(void)  
 {  
 }  
 virtual void Visit(CCommonEmployee commonEmployee) = 0;  
 virtual void Visit(CManager manager) = 0;  
 virtual int GetTotalSalary() = 0;  
};  
  
**//BaseVisitor.h**  
#pragma once  
#include "ivisitor.h"  
#include "CommonEmployee.h"  
#include "Manager.h"  
#include <iostream>  
using std::string;  
class CBaseVisitor :public IVisitor  
{  
public:  
 CBaseVisitor(void);  
 ~CBaseVisitor(void);  
 void Visit(CCommonEmployee commonEmployee);  
 void Visit(CManager manager);  
 int GetTotalSalary();  
private:  
 string GetBasicInfo(CEmployee \*pemployee);  
 string GetManagerInfo(CManager manager);  
 string GetCommonEmployee(CCommonEmployee employee);  
 static const int MANAGER\_COEFFICENT = 5;  
 static const int COMMONEMPLOYEE\_COEFFICENT = 2;  
 int m\_commonTotal;  
 int m\_managerTotal;  
 void CalCommonSalary(int salary);  
 void CalManagerSalary(int salary);  
};  
  
**//BaseVisitor.cpp**  
#include "StdAfx.h"  
#include "..\CommonDeclare\Convert.h"  
#include "BaseVisitor.h"  
#include <iostream>  
using std::string;  
using std::cout;  
using std::endl;  
  
CBaseVisitor::CBaseVisitor(void)  
{  
 m\_commonTotal = 0;  
 m\_managerTotal = 0;  
}  
  
CBaseVisitor::~CBaseVisitor(void)  
{  
}  
  
void CBaseVisitor::Visit(CCommonEmployee commonEmployee)  
{  
 cout << this->GetCommonEmployee(commonEmployee).c\_str() << endl;  
 this->CalCommonSalary(commonEmployee.GetSalary());  
}  
  
void CBaseVisitor::Visit(CManager manager)  
{  
 cout << this->GetManagerInfo(manager).c\_str() << endl;  
 this->CalManagerSalary(manager.GetSalary());  
}  
  
string CBaseVisitor::GetBasicInfo(CEmployee \*pemployee)  
{  
 string info = "";  
 info.append("姓名：");  
 info.append(pemployee->GetName());  
 info.append("\t");  
 info.append("性别：");  
 info.append(CConvert::ToString(pemployee->GetSex()));  
 info.append("\t");  
 info.append("薪水：");  
 info.append(CConvert::ToString(pemployee->GetSalary()));  
 info.append("\t");  
 return info;  
}  
  
string CBaseVisitor::GetManagerInfo(CManager manager)  
{  
 string basicInfo = this->GetBasicInfo(&manager);  
 string otherInfo = "";  
 otherInfo.append("业绩：");  
 otherInfo.append(manager.GetPerformance());  
 otherInfo.append("\t");  
 basicInfo.append(otherInfo);  
 return basicInfo;  
}  
  
string CBaseVisitor::GetCommonEmployee(CCommonEmployee employee)  
{  
 string basicInfo = this->GetBasicInfo(&employee);  
 string otherInfo = "";  
 otherInfo.append("工作：");  
 otherInfo.append(employee.GetJob());  
 otherInfo.append("\t");  
 basicInfo.append(otherInfo);  
 return basicInfo;  
}  
  
int CBaseVisitor::GetTotalSalary()  
{  
 return this->m\_commonTotal + this->m\_managerTotal;  
}  
  
void CBaseVisitor::CalCommonSalary(int salary)  
{  
 this->m\_commonTotal += salary;  
}  
  
void CBaseVisitor::CalManagerSalary(int salary)  
{  
 this->m\_managerTotal += salary;  
}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

**//Employee.h**  
#pragma once  
#include "IVisitor.h"  
#include <iostream>  
using std::string;  
class CEmployee  
{  
public:  
static int MALE;  
static int FEMALE;  
CEmployee(void);  
virtual ~CEmployee(void);  
string GetName();  
void SetName(string name);  
int GetSalary();  
void SetSalary(int v);  
int GetSex();  
void SetSex(int v);  
virtual void Accept(IVisitor \*pVisitor) = 0;  
private:  
string m\_name;  
int m\_salary;  
int m\_sex;  
};  
  
**//Employee.cpp**  
#include "StdAfx.h"  
#include "Employee.h"  
int CEmployee::MALE = 0;  
int CEmployee::FEMALE = 1;  
  
CEmployee::CEmployee(void)  
{  
}  
  
CEmployee::~CEmployee(void)  
{  
}  
  
string CEmployee::GetName()  
{  
return m\_name;  
}  
  
void CEmployee::SetName( string name )  
{  
m\_name = name;  
}  
  
int CEmployee::GetSalary()  
{  
return m\_salary;  
}  
  
void CEmployee::SetSalary( int v )  
{  
m\_salary = v;  
}  
  
int CEmployee::GetSex()  
{  
return m\_sex;  
}  
  
void CEmployee::SetSex( int v )  
{  
m\_sex = v;  
}  
  
**//Manager.h**  
#pragma once  
#include "employee.h"  
#include "IVisitor.h"  
#include <iostream>  
using std::string;  
class CManager :  
public CEmployee  
{  
public:  
CManager(void);  
~CManager(void);  
string GetPerformance();  
void SetPerformance(string per);  
void Accept(IVisitor \*pVisitor);  
protected:  
string GetOtherInfo();  
private:  
string m\_performance;  
};  
  
**//Manager.cpp**  
#include "StdAfx.h"  
#include "Manager.h"  
#include <iostream>  
using std::string;  
  
CManager::CManager(void)  
{  
this->m\_performance = "";  
}  
  
CManager::~CManager(void)  
{  
}  
  
string CManager::GetPerformance()  
{  
return m\_performance;  
}  
  
void CManager::SetPerformance(string per)  
{  
this->m\_performance = per;  
}  
  
string CManager::GetOtherInfo()  
{  
string info = "";  
info.append("业绩：");  
info.append(this->m\_performance);  
info.append("\t");  
return info;  
}  
  
void CManager::Accept(IVisitor \*pVisitor)  
{  
pVisitor->Visit(\*this);  
}  
  
**// Visitor.cpp : 定义控制台应用程序的入口点。**  
#include "stdafx.h"  
#include "Employee.h"  
#include "CommonEmployee.h"  
#include "Manager.h"  
#include "BaseVisitor.h"  
#include "..\CommonDeclare\Convert.h"  
#include <vector>  
#include <iostream>  
using std::vector;  
using std::cout;  
using std::endl;  
  
void MockEmployee(vector<CEmployee\*> \*pvce)  
{  
CCommonEmployee \*pZhangSan = new CCommonEmployee();  
pZhangSan->SetJob("编写Java程序，绝对的蓝领、苦工加搬运工");  
pZhangSan->SetName("张三");  
pZhangSan->SetSalary(1800);  
pZhangSan->SetSex(CEmployee::MALE);  
pvce->push\_back(pZhangSan);  
  
CCommonEmployee \*pLiSi = new CCommonEmployee();  
pLiSi->SetJob("页面美工，审美素质太不流行了！");  
pLiSi->SetName("李四");  
pLiSi->SetSalary(1900);  
pLiSi->SetSex(CEmployee::FEMALE);  
pvce->push\_back(pLiSi);  
  
CManager \*pWangWu = new CManager();  
pWangWu->SetPerformance("基本上是负值，但是我会拍马屁呀");  
pWangWu->SetName("王五");  
pWangWu->SetSalary(1900);  
pWangWu->SetSex(CEmployee::FEMALE);  
pvce->push\_back(pWangWu);  
}  
  
void DoIt()  
{  
vector<CEmployee\*> vce;  
MockEmployee(&vce);  
vector<CEmployee\*>::const\_iterator readIt = vce.begin();  
  
CBaseVisitor visitor;  
for (; readIt != vce.end(); readIt ++)  
{  
(\*readIt)->Accept(&visitor);  
}  
cout << "本公司的月工资总额是：" << CConvert::ToString(visitor.GetTotalSalary()).c\_str() << endl;  
  
vector<CEmployee\*>::reverse\_iterator delIt = vce.rbegin();  
for (; delIt != vce.rend(); delIt++)  
{  
delete (\*delIt);  
(\*delIt) = NULL;  
}  
vce.clear();  
}  
  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
DoIt();  
return 0;  
}

[复制代码](javascript:void(0);)



访问者模式属于行为型模式。访问者模式是由访问者主动发出的动作。

[**设计模式C++学习笔记之十九（State状态模式）**](http://www.cnblogs.com/wanggary/archive/2011/04/21/2024117.html)

**19.1.解释**

概念：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

main()，客户

CLiftState，电梯状态抽象类

CCloseingState，电梯门关闭

COpenningState，电梯门打开

CRunningState，电梯运行

CStoppingState，电梯停止

CContext，电梯的控制面板

说明：CContext保持电梯的状态，并提供操作的接口函数。当函数被调用时，CContext直接调用当前状态的相应函数。由状态的接口函数来确定是否可以执行这个动作，以及修改状态为执行这个动作后的状态。

看代码：第一块是不使用模式的做法，第二块是使用模式的做法，在main()函数里会有调用的方式。

[复制代码](javascript:void(0);)

//ILift.h  
#pragma once  
class ILift  
{  
public:  
  
 ILift(void)  
 {  
 }  
  
 virtual ~ILift(void)  
 {  
 }  
 static const int OPENING\_STATE = 1;  
 static const int CLOSING\_STATE = 2;  
 static const int RUNNING\_STATE = 3;  
 static const int STOPPING\_STATE = 4;  
 virtual void SetState(int state) = 0;  
 virtual void Open() = 0;  
 virtual void Close() = 0;  
 virtual void Run() = 0;  
 virtual void Stop() = 0;  
};  
  
//Lift.h  
#pragma once  
#include "ilift.h"  
class CLift :public ILift  
{  
public:  
 CLift(void);  
 ~CLift(void);  
 void SetState(int state);  
 void Open();  
 void Close();  
 void Run();  
 void Stop();  
private:  
 int m\_state;  
 void OpenWithoutLogic();  
 void CloseWithoutLogic();  
 void RunWithoutLogic();  
 void StopWithoutLogic();  
};  
  
//Lift.cpp  
#include "StdAfx.h"  
#include "Lift.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
CLift::CLift(void)  
{  
 this->m\_state = 0;  
}  
  
CLift::~CLift(void)  
{  
}  
  
void CLift::SetState(int state)  
{  
 this->m\_state = state;  
}  
  
void CLift::Open()  
{  
 switch(this->m\_state)  
 {  
 case OPENING\_STATE:  
 break;  
 case CLOSING\_STATE:  
 this->OpenWithoutLogic();  
 this->SetState(OPENING\_STATE);  
 break;  
 case RUNNING\_STATE:  
 break;  
 case STOPPING\_STATE:  
 this->OpenWithoutLogic();  
 this->SetState(OPENING\_STATE);  
 break;  
 }  
}  
  
void CLift::Close()  
{  
 switch(this->m\_state)  
 {  
 case OPENING\_STATE:  
 this->CloseWithoutLogic();  
 this->SetState(CLOSING\_STATE);  
 break;  
 case CLOSING\_STATE:  
 break;  
 case RUNNING\_STATE:  
 break;  
 case STOPPING\_STATE:  
 break;  
 }  
}  
  
void CLift::Run()  
{  
 switch(this->m\_state)  
 {  
 case OPENING\_STATE:  
 break;  
 case CLOSING\_STATE:  
 this->RunWithoutLogic();  
 this->SetState(RUNNING\_STATE);  
 break;  
 case RUNNING\_STATE:  
 break;  
 case STOPPING\_STATE:  
 this->RunWithoutLogic();  
 this->SetState(RUNNING\_STATE);  
 break;  
 }  
}  
  
void CLift::Stop()  
{  
 switch(this->m\_state)  
 {  
 case OPENING\_STATE:  
 break;  
 case CLOSING\_STATE:  
 this->StopWithoutLogic();  
 this->SetState(CLOSING\_STATE);  
 break;  
 case RUNNING\_STATE:  
 this->StopWithoutLogic();  
 this->SetState(CLOSING\_STATE);  
 break;  
 case STOPPING\_STATE:  
 break;  
 }  
}  
  
void CLift::OpenWithoutLogic()  
{  
 cout << "电梯门开启..." << endl;  
}  
  
void CLift::CloseWithoutLogic()  
{  
 cout << "电梯门关闭..." << endl;  
}  
  
void CLift::RunWithoutLogic()  
{  
 cout << "电梯上下跑起来..." << endl;  
}  
  
void CLift::StopWithoutLogic()  
{  
 cout << "电梯停止了..." << endl;  
}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

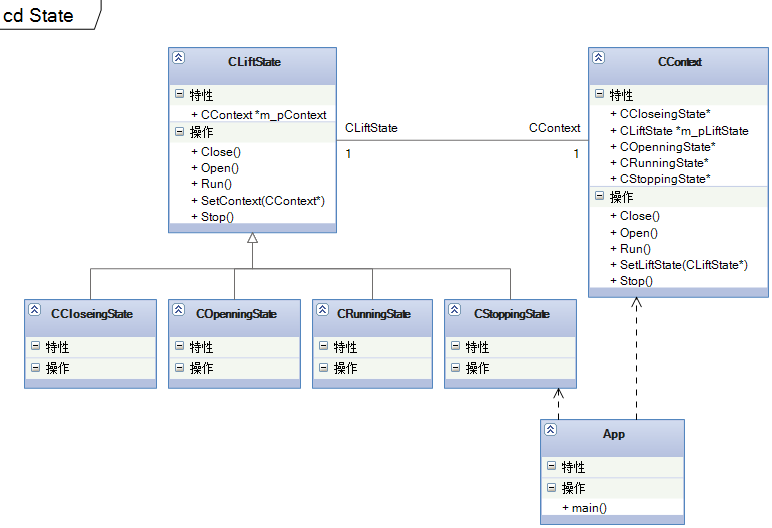
//LiftState.h  
#pragma once  
class CContext;  
class CLiftState  
{  
public:  
 CLiftState(void);  
 virtual ~CLiftState(void);  
 void SetContext(CContext \*pContext);  
 virtual void Open() = 0;  
 virtual void Close() = 0;  
 virtual void Run() = 0;  
 virtual void Stop() = 0;  
protected:  
 CContext \*m\_pContext;  
};  
  
//LiftState.cpp  
#include "StdAfx.h"  
#include "LiftState.h"  
  
CLiftState::CLiftState(void)  
{  
}  
  
CLiftState::~CLiftState(void)  
{  
}  
  
void CLiftState::SetContext( CContext \*pContext )  
{  
 m\_pContext = pContext;  
}  
  
//CloseingState.h  
#pragma once  
#include "liftstate.h"  
class CCloseingState :  
 public CLiftState  
{  
public:  
 CCloseingState(void);  
 ~CCloseingState(void);  
 void Open();  
 void Close();  
 void Run();  
 void Stop();  
};  
  
//CloseingState.cpp  
#include "StdAfx.h"  
#include "CloseingState.h"  
#include "Context.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
CCloseingState::CCloseingState(void)  
{  
}  
  
CCloseingState::~CCloseingState(void)  
{  
}  
  
void CCloseingState::Open()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pOpenningState);  
 this->CLiftState::m\_pContext->GetLiftState()->Open();  
}  
  
void CCloseingState::Close()  
{  
 cout << "电梯门关闭..." << endl;  
}  
  
void CCloseingState::Run()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pRunningState);  
 this->CLiftState::m\_pContext->GetLiftState()->Run();  
}  
  
void CCloseingState::Stop()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pStoppingState);  
 this->CLiftState::m\_pContext->GetLiftState()->Stop();  
}  
  
//OpenningState.h  
#pragma once  
#include "liftstate.h"  
class COpenningState :public CLiftState  
{  
public:  
 COpenningState(void);  
 ~COpenningState(void);  
 void Open();  
 void Close();  
 void Run();  
 void Stop();  
};  
  
//OpenningState.cpp  
#include "StdAfx.h"  
#include "OpenningState.h"  
#include "Context.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
COpenningState::COpenningState(void)  
{  
}  
  
COpenningState::~COpenningState(void)  
{  
}  
  
void COpenningState::Open()  
{  
 cout << "电梯门开启..." << endl;  
}  
  
void COpenningState::Close()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pCloseingState);  
 this->CLiftState::m\_pContext->GetLiftState()->Close();  
}  
  
void COpenningState::Run()  
{  
 //do nothing  
}  
  
void COpenningState::Stop()  
{  
 //do nothing  
}  
  
//RunningState.h  
#pragma once  
#include "liftstate.h"  
class CRunningState :  
 public CLiftState  
{  
public:  
 CRunningState(void);  
 ~CRunningState(void);  
 void Open();  
 void Close();  
 void Run();  
 void Stop();  
};  
  
//RunningState.cpp  
#include "StdAfx.h"  
#include "RunningState.h"  
#include "Context.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
CRunningState::CRunningState(void)  
{  
}  
  
CRunningState::~CRunningState(void)  
{  
}  
  
void CRunningState::Open()  
{  
 //do nothing  
}  
  
void CRunningState::Close()  
{  
 //do nothing  
}  
  
void CRunningState::Run()  
{  
 cout << "电梯上下跑..." << endl;  
}  
  
void CRunningState::Stop()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pStoppingState);  
 this->CLiftState::m\_pContext->GetLiftState()->Stop();  
}  
  
//StoppingState.h  
#pragma once  
#include "liftstate.h"  
class CStoppingState :  
 public CLiftState  
{  
public:  
 CStoppingState(void);  
 ~CStoppingState(void);  
 void Open();  
 void Close();  
 void Run();  
 void Stop();  
};  
  
//StoppingState.cpp  
#include "StdAfx.h"  
#include "StoppingState.h"  
#include "Context.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
CStoppingState::CStoppingState(void)  
{  
}  
  
CStoppingState::~CStoppingState(void)  
{  
}  
  
void CStoppingState::Open()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pOpenningState);  
 this->CLiftState::m\_pContext->GetLiftState()->Open();  
}  
  
void CStoppingState::Close()  
{  
 //do nothing  
}  
  
void CStoppingState::Run()  
{  
 this->CLiftState::m\_pContext->SetLiftState(CContext::pRunningState);  
 this->CLiftState::m\_pContext->GetLiftState()->Run();  
}  
  
void CStoppingState::Stop()  
{  
 cout << "电梯停止了..." << endl;  
}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//Contex.h  
#pragma once  
#include "LiftState.h"  
#include "OpenningState.h"  
#include "CloseingState.h"  
#include "RunningState.h"  
#include "StoppingState.h"  
class CContext  
{  
public:  
 CContext(void);  
 ~CContext(void);  
 static COpenningState \*pOpenningState;  
 static CCloseingState \*pCloseingState;  
 static CRunningState \*pRunningState;  
 static CStoppingState \*pStoppingState;  
 CLiftState \* GetLiftState();  
 void SetLiftState(CLiftState \*pLiftState);  
 void Open();  
 void Close();  
 void Run();  
 void Stop();  
private:  
 CLiftState \*m\_pLiftState;  
};  
  
//Context.cpp  
#include "StdAfx.h"  
#include "Context.h"  
COpenningState\* CContext::pOpenningState = NULL;  
CCloseingState\* CContext::pCloseingState = NULL;  
CRunningState\* CContext::pRunningState = NULL;  
CStoppingState\* CContext::pStoppingState = NULL;  
  
CContext::CContext(void)  
{  
 m\_pLiftState = NULL;  
 pOpenningState = new COpenningState();  
 pCloseingState = new CCloseingState();  
 pRunningState = new CRunningState();  
 pStoppingState = new CStoppingState();  
}  
  
CContext::~CContext(void)  
{  
 delete pOpenningState;  
 pOpenningState = NULL;  
 delete pCloseingState;  
 pCloseingState = NULL;  
 delete pRunningState;  
 pRunningState = NULL;  
 delete pStoppingState;  
 pStoppingState = NULL;  
}  
  
CLiftState \* CContext::GetLiftState()  
{  
 return m\_pLiftState;  
}  
  
void CContext::SetLiftState(CLiftState \*pLiftState)  
{  
 this->m\_pLiftState = pLiftState;  
 this->m\_pLiftState->SetContext(this);  
}  
  
void CContext::Open()  
{  
 this->m\_pLiftState->Open();  
}  
  
void CContext::Close()  
{  
 this->m\_pLiftState->Close();  
}  
  
void CContext::Run()  
{  
 this->m\_pLiftState->Run();  
}  
  
void CContext::Stop()  
{  
 this->m\_pLiftState->Stop();  
}  
  
// State.cpp : 定义控制台应用程序的入口点。  
#include "stdafx.h"  
#include "ILift.h"  
#include "Lift.h"  
#include "Context.h"  
#include "OpenningState.h"  
#include "CloseingState.h"  
#include "RunningState.h"  
#include "StoppingState.h"  
#include <iostream>  
using std::cout;  
using std::endl;  
  
void DoIt()  
{  
 //ILift.h, Lift.h, Lift.cpp  
 ILift \*pLift = new CLift();  
 pLift->SetState(ILift::STOPPING\_STATE);//电梯的初始条件是停止状态。  
 pLift->Open();//首先是电梯门开启，人进去  
 pLift->Close();//然后电梯门关闭  
 pLift->Run();//再然后，电梯跑起来，向上或者向下  
 pLift->Stop();//最后到达目的地，电梯停下来  
 delete pLift;  
}  
  
  
void DoNew()  
{  
 //LiftState.h, LiftState.cpp, OpenningState.h, CloseingState.h, RunningState.h, StoppingState.h  
 //Context.h, Context.cpp  
 CContext context;  
 CCloseingState closeingState;  
 context.SetLiftState(&closeingState);  
 context.Close();  
 context.Open();  
 context.Run();  
 context.Stop();  
}  
  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
 cout << "----------使用模式之前----------" << endl;  
 DoIt();  
 cout << "----------使用模式之后----------" << endl;  
 DoNew();  
  
 \_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF | \_CRTDBG\_ALLOC\_MEM\_DF);  
 \_CrtDumpMemoryLeaks();  
 return 0;  
}

[复制代码](javascript:void(0);)



状态模式也是行为型模式。写了好多天了，才发现如何在页面上插入代码。这样方便多了，看起来也清楚多了。博客园也许需要开个入门贴子，免得新来博客园的童鞋都不大会用。这几天还在学习数据结构中的排序，入排序入手来学习数据结构还是比较容易的。上学的时候只顾着考试了，该学的也都学了，但考完试就全忘掉，已经成为了一种习惯。现在再补回来。鼓励自己加油！一定要做一个学习型的程序员。