我们选择了C++，主要是因为它够艺术、够自由，使用它我们可以实现各种想法，而MFC将多种可灵活使用的功能封装起来，我们岂能忍受这种“黑盒”操作？于是研究分析MFC的核心机制成为必然。

       首先，列出要讲的MFC六大核心机制：

       1、MFC程序的初始化。  
       2、运行时类型识别（RTTI）。  
       3、动态创建。  
       4、永久保存。  
       5、消息映射。  
       6、消息传递。

**MFC程序的初始化过程**

**简单的MFC窗口程序**

       设计一个简单完整MFC程序，产生一个窗口。当然这不能让AppWizard自动为我们生成。我们可以在Win32 Application工程下面那样写：

#include <afxwin.h>

class MyApp : public CWinApp

{

public:

BOOL InitInstance() //②程序入点

{

CFrameWnd \*Frame=new CFrameWnd();//构造框架

m\_pMainWnd=Frame; //将m\_pMainWnd设定为Frame;

Frame->Create(NULL,"最简单的窗口");//建立框架

Frame->ShowWindow(SW\_SHOW); //显示框架

return true; //返回

}

};

MyApp theApp; //①建立应用程序。

       设定链接MFC库，运行，即可看见一个窗口。

       从上面，大家可以看到建立一个MFC窗口很容易，只用两步：一是从CWinApp派生一个应用程序类（这里是MyApp），然后建立应用程序对象（theApp），就可以产生一个自己需要的窗口（即需要什么样就在InitInstance()里创建就行了）。

       整个程序，就改写一个InitInstance()函数，创建那么一个对象（theApp），就是一个完整的窗口程序。这就是“黑盒”操作的魔力！

       在我们正想为微软鼓掌的时候，我们突然觉得心里空荡荡的，我们想知道微软帮我们做了什么事情，而我们想编自己的程序时又需要做什么事情，哪怕在上面几行的程序里面，我们还有不清楚的地方，比如，干嘛有一个m\_pMainWnd指针变量，它从哪里来，又要到哪里去呢？想一想在DOS下编程是多么美妙的一件事呵，我们需要什么变量，就声明什么变量，需要什么样的函数，就编写什么样的函数，或者引用函数库……但是现在我们怎么办？

       我们可以逆向思维一下，MFC要达到这种效果，它是怎么做的呢？首先我们要弄明白，VC++不是一种语言，它就象我们学c语言的时候的一个类似记事本的编辑器（请原谅我的不贴切的比喻），所以，在VC里面我们用的是C++语言编程，C++才是根本（初学者总是以为VC是一门什么新的什么语言，一门比C++先进很多的复杂语言，汗）。说了那么多，我想用一句简单的话概括“MFC黑箱’，就是为我们的程序加入一些固化的‘C++代码’的东西”。

       既然MFC黑箱帮我们加入了代码，那么大家想想它会帮我们加入什么样的代码呢？他会帮我们加入求解一元二次方程的代码吗？当然不会，所以它加入的实际上是每次编写窗口程序必须的，通用的代码。

       再往下想，什么才是通用的呢？我们每次视窗编程都要写WinMain()函数，都要有注册窗口，产生窗口，消息循环，回调函数……即然每次都要的东西，就让它们从我们眼前消失，让MFC帮忙写入！

**手动模拟MFC程序的初始化**

       要知道MFC初始化过程，大家当然可以跟踪执行程序。但这种跟踪很麻烦，我相信大家都会跟踪的晕头转向。本人觉得哪怕你理解了MFC代码，也很容易让人找不着北，我们完全不懂的时候，在成千上万行程序的迷宫中如何能找到出口？

       我们要换一种方法，不如就来重新编写个MFC库吧，哗！大家不要笑，小心你的大牙，我不是疯子（虽然疯子也说自己不疯）。我们要写的就是最简单的MFC类库，就是把MFC宏观上的，理论上的东西写出来。我们要用最简化的代码，简化到刚好能运行。

**1、需要“重写”的MFC库**

       既然，我们这一节写的是MFC程序的初始化过程，上面我们还有了一个可执行的MFC程序。程序中只是用了两个MFC类，一个是CWinApp，另一个是CFrameWnd。当然，还有很多同样重要MFC类如视图类，文档类等等。但在上面的程序可以不用到，所以暂时省去了它（总之是为了简单）。

       好，现在开始写MFC类库吧……唉，面前又有一个大难题，就是让大家背一下MFC层次结构图。天，那张鱼网怎么记得住，但既然我们要理解他，总得知道它是从那里派生出来的吧。

       考虑到大家都很辛苦，那我们看一下上面两个类的父子关系（箭头代表派生）：

CObject->CCmdTarget->CWinThread->CWinApp->自己的重写了InitInstance()的应用程序类。

CObject（同上）->CCmdTarget（同上）->CWnd->CFrameWnd

       看到层次关系图之后，终于可以开始写MFC类库了。按照上面层次结构，我们可以写以下六个类（为了直观，省去了构造函数和析构函数）。

/////////////////////////////////////////////////////////

class CObiect{};//MFC类的基类。

class CCmdTarget : public CObject{};

------------------------------------------------

class CWinThread : public CCmdTarget{};

class CWinApp : public CWinThread{};

------------------------------------------------

class CWnd : public CCmdTarget{};

class CFrameWnd : public CWnd{};

/////////////////////////////////////////////////////////

       大家再想一下，在上面的类里面，应该有什么？大家马上会想到，CWinApp类或者它的基类CCmdTarget里面应该有一个虚函数virtual BOOL InitInstance()，是的，因为那里是程序的入口点，初始化程序的地方，那自然少不了的。可能有些朋友会说，反正InitInstance()在派生类中一定要重载，我不在CCmdTarget或CWinApp类里定义，留待CWinApp的派生类去增加这个函数可不可以。扯到这个问题可能有点越说越远，但我想信C++的朋友对虚函数应该是没有太多的问题的。总的来说，作为程序员如果清楚知道基类的某个函数要被派生类用到，那定义为虚函数要方便很多。

       也有很多朋友问，C++为什么不自动把基类的所有函数定义为虚函数呢，这样可以省了很多麻烦，这样所有函数都遵照派生类有定义的函数就调用派生类的，没定义的就调用基类的，不用写virtual的麻烦多好！其实，很多面向对象的语言都这样做了。但定义一个虚函数要生成一个虚函数表，要占用系统空间，虚函数越多，表就越大，有时得不偿失！这里哆嗦几句，是因为往后要说明的消息映射中大家更加会体验到这一点，好了，就此打往。

       上面我们自己解决了一个问题，就是在CCmdTarge写一个virtual BOOL InitInstance()。

**2、WinMain()函数和CWinApp类**

       大家再往下想，我们还要我们MFC“隐藏”更多的东西：WinMain()函数，设计窗口类，窗口注册，消息循环，回调函数……我们马上想到封装想封装他们。大家似乎隐约地感觉到封装WinMain()不容易，觉得WinMain()是一个特殊的函数，许多时候它代表了一个程序的起始和终结。所以在以前写程序的时候，我们写程序习惯从WinMain()的左大括写起，到右大括弧返回、结束程序。

       我们换一个角度去想，有什么东西可以拿到WinMain()外面去做，许多初学者们，总觉得WinMain()函数是天大的函数，什么函数都好象要在它里面才能真正运行。其实这样了解很片面，甚至错误。我们可以写一个这样的C++程序：

////////////////////////////////////////////////////

#include <iostream.h>

class test{

public:

test(){cout<<"请改变你对main()函数的看法！"<<endl;}

};

test test1;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void main(){}

////////////////////////////////////////////////////

       在上面的程序里，入口的main()函数表面上什么也不做，但程序执行了（注：实际入口函数做了一些我们可以不了解的事情），并输出了一句话（注：全局对象比main()首先运行）。现在大家可以知道我们的WinMain()函数可以什么都不做，程序依然可以运行，但没有这个入口函数程序会报错。

       那么WinMain()函数会放哪个类上面呢，请看下面程序：

#include <afxwin.h>

class MyApp : public CWinApp

{

public:

BOOL InitInstance() //②程序入点

{

AfxMessageBox("程序依然可以运行！");

return true;

}

};

MyApp theApp; //①建立应用程序。

       大家可以看到，我并没有构造框架，而程序却可以运行了——弹出一个对话框（如果没有WinMain()函数程序会报错）。上面我这样写还是为了直观起见，其实我们只要写两行程序：

#include <afxwin.h>

CWinApp theApp; //整个程序只构造一个CWinApp类对象，程序就可以运行！

       所以说，只要我们构造了CWinApp对象，就可以执行WinMain()函数。我们马上相信WinMain()函数是在CWinApp类或它的基类中，而不是在其他类中。其实这种看法是错误的，我们知道编写C++程序的时候，不可能让你在一个类中包含入口函数，WinMain()是由系统调用，跟我们的平时程序自身调用的函数有着本质的区别。我们可以暂时简单想象成，当CWinApp对象构造完的时候，WinMain()跟着执行。

       现在大家明白了，大部分的“通用代码（我们想封装隐藏的东西）”都可以放到CWinApp类中，那么它又是怎样运行起来的呢？为什么构造了CWinApp类对象就“自动”执行那么多东西。

       大家再仔细想一下，CWinApp类对象构造之后，它会“自动”执行自己的构造函数。那么我们可以把想要“自动”执行的代码放到CWinApp类的构造函数中。

       那么CWinApp类可能打算这样设计（先不计较正确与否）：

class CWinApp : public CWinThead{

public:

virtual BOOL InitInstance(); //解释过的程序的入点

CWinApp ::CWinApp(){ //构造函数

////////////////////////

WinMain(); //这个是大家一眼看出的错误

Create(); //设计、创建、更新显示窗口

Run(); //消息循环

//////////////////////

}

};

       写完后，大家又马上感觉到似乎不对，WinMain()函数在这里好象真的一点用处都没有，并且能这样被调用吗（请允许我把手按在圣经上声明一下：WinMain()不是普通的函数，它要肩负着初始化应用程序，包括全局变量的初始化，是由系统而不是程序本身调用的，WinMain()返回之后，程序就结束了，进程撤消）。再看Create()函数，它能确定设计什么样的窗口，创建什么样的窗口吗？如果能在CWinApp的构造函数里确定的话，我们以后设计MFC程序时窗口就一个样，这样似乎不太合理。

       回过头来，我们可以让WinMain()函数一条语句都不包含吗？不可以，我们看一下WinMain() 函数的四个参数：

WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

       其中第一个参数指向一个实例句柄，我们在设计WNDCLASS的时候一定要指定实例句柄。我们窗口编程，肯定要设计窗口类。所以，WinMain()再简单也要这样写：

int WinMain(HINSTANCE hinst, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{ hInstance=hinst }

        既然实例句柄要等到程序开始执行才能知道，那么我们用于创建窗口的Create()函数也要在WinMain()内部才能执行（因为如果等到WinMain()执行完毕后，程序结束，进程撤消，当然Create()也不可能创建窗口）。

       再看Run()（消息循环）函数，它能在WinMain()函数外面运行吗？众所周知，消息循环就是相同的那么几句代码，但我们也不要企图把它放在WinMain()函数之外执行。

       所以我们的WinMain()函数可以像下面这样写：

WinMain(……)

{

……窗口类对象执行创建窗口函数……

……程序类对象执行消息循环函数……

}

       对于WinMain()的问题，得总结一下，我们封装的时候是不可以把它封装到CWinApp类里面，但由于WinMain()的不变性（或者说有规律可循），MFC完全有能力在我们构造CWinApp类对象的时候，帮我们完成那几行代码。

       转了一个大圈，我们仿佛又回到了SDK编程的开始。但现在我们现在能清楚地知道，表面上MFC与SDK编程截然不同，但实质上MFC只是用类的形式封装了SDK函数，封装之后，我们在WinMain()函数中只需要几行代码，就可以完成一个窗口程序。我们也由此知道了应如何去封装应用程序类（CWinApp）和主框架窗口类（CFrameWnd）。下面把上开始设计这两个类。

**3、MFC库的“重写”**

       为了简单起见，我们忽略这两个类的基类和派生类的编写，可能大家会认为这是一种很不负责任的做法，但本人觉得这既可减轻负担，又免了大家在各类之间穿来穿去，更好理解一些（我们在关键的地方作注明）。还有，我把全部代码写在同一个文件中，让大家看起来不用那么吃力，但这是最不提倡的写代码方法，大家不要学哦！

#include <windows.h>

HINSTANCE hInstance;

class CFrameWnd

{

HWND hwnd;

public:

CFrameWnd(); //也可以在这里调用Create()

virtual ~CFrameWnd();

int Create(); //类就留意这一个函数就行了！

BOOL ShowWnd();

};

class CWinApp1

{

public:

CFrameWnd\* m\_pMainWnd;//在真正的MFC里面

//它是CWnd指针，但这里由于不写CWnd类

//只要把它写成CFrameWnd指针

CWinApp1\* m\_pCurrentWinApp;//指向应用程序对象本身

CWinApp1();

virtual ~CWinApp1();

virtual BOOL InitInstance();//MFC原本是必须重载的函数，最重要的函数！！！！

virtual BOOL Run();//消息循环

};

CFrameWnd::CFrameWnd(){}

CFrameWnd::~CFrameWnd(){}

int CFrameWnd::Create() //封装创建窗口代码

{

WNDCLASS wndcls;

wndcls.style=0;

wndcls.cbClsExtra=0;

wndcls.cbWndExtra=0;

wndcls.hbrBackground=(HBRUSH)GetStockObject(WHITE\_BRUSH);

wndcls.hCursor=LoadCursor(NULL,IDC\_CROSS);

wndcls.hIcon=LoadIcon(NULL,IDC\_ARROW);

wndcls.hInstance=hInstance;

wndcls.lpfnWndProc=DefWindowProc;//默认窗口过程函数。

//大家可以想象成MFC通用的窗口过程。

wndcls.lpszClassName="WindowClassName";

wndcls.lpszMenuName=NULL;

RegisterClass(&wndcls);

hwnd=CreateWindow("WindowClassName","WindowInstanceTitle",WS\_OVERLAPPEDWINDOW,0,0,600,400,NULL,NULL,hInstance,NULL);

return 0;

}

BOOL CFrameWnd::ShowWnd()//显示更新窗口

{

ShowWindow(hwnd,SW\_SHOWNORMAL);

UpdateWindow(hwnd);

return 0;

}

/////////////

CWinApp1::CWinApp1()

{

m\_pCurrentWinApp=this;

}

CWinApp1::~CWinApp1(){}

//以下为InitInstance()函数，MFC中要为CWinApp的派生类改写，

//这里为了方便理解，把它放在CWinApp类里面完成！

//你只要记住真正的MFC在派生类改写此函数就行了。

BOOL CWinApp1::InitInstance()

{

m\_pMainWnd=new CFrameWnd;

m\_pMainWnd->Create();

m\_pMainWnd->ShowWnd();

return 0;

}

BOOL CWinApp1::Run()//////////////////////封装消息循环

{

MSG msg;

while(GetMessage(&msg,NULL,0,0))

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return 0;

} //////////////////////////////////////////////////////封装消息循环

CWinApp1 theApp; //应用程序对象（全局）

int WINAPI WinMain( HINSTANCE hinst, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

hInstance=hinst;

CWinApp1\* pApp=theApp.m\_pCurrentWinApp;

//真正的MFC要写一个全局函数AfxGetApp，以获取CWinApp指针。

pApp->InitInstance();

pApp->Run();

return 0;

}

       代码那么长，实际上只是写了三个函数，一是CFrameWnd类的Create()，第二个是CWinApp类的InitInstance()和Run()。在此特别要说明的是InitInstance()，真正的MFC中，那是我们跟据自己构造窗口的需要，自己改写这个函数。

       大家可以看到，封装了上面两个类以后，在入口函数WinMain中就写几行代码，就可以产生一个窗口程序。在MFC中，因为WinMain函数就是固定的那么几行代码，所以MFC绝对可以帮我们自动完成（MFC的特长就是帮我们完成有规律的代码），也因此我们创建MFC应用程序的时候，看不到WinMain函数。

分享源自：<http://blog.csdn.net/liyi268/article/details/297875>

**运行时类型识别（RTTI）**

**typeid运算子**

       运行时类型识别（RTTI）即是程序执行过程中知道某个对象属于某个类，我们平时用C++编程接触的RTTI一般是编译器的RTTI，即是在新版本的VC++编译器里面选用“使能RTTI”，然后载入typeinfo.h文件，就可以使用一个叫typeid()的运算子，它的地位与在C++编程中的sizeof()运算子类似的地方（包含一个头文件，然后就有一个熟悉好用的函数）。typdid()关键的地方是可以接受两个类型的参数：一个是类名称，一个是对象指针。所以我们判别一个对象是否属于某个类就可以象下面那样：

if （typeid (ClassName)== typeid（\*ObjectName））{

((ClassName\*)ObjectName)->Fun();

}

       像上面所说的那样，一个typeid()运算子就可以轻松地识别一个对象是否属于某一个类，但MFC并不是用typeid()的运算子来进行动态类型识别，而是用一大堆令人费解的宏。很多学员在这里很疑惑，好象MFC在大部分地方都是故作神秘。使们大家编程时很迷惘，只知道在这里加入一组宏，又在那儿加入一个映射，而不知道我们为什么要加入这些东东。

       其实，早期的MFC并没有typeid()运算子，所以只能沿用一个老办法。我们甚至可以想象一下，如果MFC早期就有template（模板）的概念，可能更容易解决RTTI问题。

       所以，我们要回到“古老”的年代，想象一下，要完成RTTI要做些什么事情。就好像我们在一个新型（新型到我们还不认识）电器公司里面，我们要识别哪个是电饭锅，哪个是电磁炉等等，我们要查看登记的各电器一系列的信息，我们才可以比较、鉴别，那个东西是什么！

**CRuntimeClass链表的设计**

       要登记一系列的消息并不是一件简单的事情，大家可能首先想到用数组登记对象。但如果用数组，我们要定义多大的数组才好呢，大了浪费空间，小了更加不行。所以我们要用另一种数据结构——链表。因为链表理论上可大可小，可以无限扩展。

       链表是一种常用的数据结构，简单地说，它是在一个对象里面保存了指向下一个同类型对象的指针。我们大体可以这样设计我们的类：

struct CRuntimeClass

{

//……类的名称等一切信息……

CRuntimeClass \* m\_pNextClass;//指向链表中下一CRuntimeClass对象的指针

};

       链表还应该有一个表头和一个表尾，这样我们在查链表中各对象元素的信息的时候才知道从哪里查起，到哪儿结束。我们还要注明本身是由哪能个类派生。所以我们的链表类要这样设计：

struct CRuntimeClass

{

//……类的名称等一切信息……

CRuntimeClass \* m\_pBaseClass;//指向所属的基类。

CRuntimeClass \* m\_pNextClass;//定义表尾的时候只要定义此指针为空就可以 了。

static CRuntimeClass\* pFirstClass;//这里表头指针属于静态变量，因为我们知道static变量在内存中只初始化一次，就可以为各对象所用！保证了各对象只有一个表头。

};

       有了CRuntimeClass结构后，我们就可以定义链表了：

static CRuntimeClass classCObject={NULL,NULL}; //这里定义了一个CRuntimeClass对象，因为classCObject无基类，所以m\_pBaseClass为NULL。因为目前只有一个元素（即目前没有下一元素），所以m\_pNextClass为NULL（表尾）。

       至于pFirstClass（表头），大家可能有点想不通，它到什么地方去了。因为我们这里并不想把classCObject作为链表表头，我们还要在前面插入很多的CRuntimeClass对象，并且因为pFirstClass为static指针，即是说它不是属于某个对象，所以我们在用它之前要先初始化：

CRuntimeClass\* CRuntimeClass::pFirstClass=NULL;

       现在我们可以在前面插入一个CRuntimeClass对象，插入之前我得重要申明一下：如果单纯为了运行时类型识别，我们未必用到m\_pNextClass指针（更多是在运行时创建时用），我们关心的是类本身和它的基类。这样，查找一个对象是否属于一个类时，主要关心的是类本身及它的基类。

CRuntimeClass classCCmdTarget={ &classCObject, NULL};

CRuntimeClass classCWnd={ &classCCmdTarget ,NULL };

CRuntimeClass classCView={ &classCWnd , NULL };

       好了，上面只是仅仅为一个指针m\_pBaseClass赋值（MFC中真正CRuntimeClass有多个成员变量和方法），就连接成了链表。假设我们现在已全部构造完成自己需要的CRuntimeClass对象，那么，这时候应该定义表头。即要用pFirstClass指针指向我们最后构造的CRuntimeClass对象——classCView。

CRuntimeClass::pFirstClass=&classCView;

       现在链表有了，表头表尾都完善了，问题又出现了，我们应该怎样访问每一个CRuntimeClass对象？要判断一个对象属于某类，我们要从表头开始，一直向表尾查找到表尾，然后才能比较得出结果吗。肯定不是这样！

**类中构造CRuntimeClass对象**

       大家可以这样想一下，我们构造这个链表的目的，就是构造完之后，能够按主观地拿一个CRuntimeClass对象和链表中的元素作比较，看看其中一个对象是否属于你指定的类。这样，我们需要有一个函数，一个能返回自身类型名的函数GetRuntimeClass()。

       上面简单地说了一下链表的过程，但单纯有这个链表是没有任何意义。回到MFC中来，我们要实现的是在每个需要有RTTI能力的类中构造一个CRuntimeClass对象，比较一个类是否属于某个CRuntimeClass对象的时候，实际上只是比较CRuntimeClass对象。

       如何在各个类之中插入CRuntimeClass对象，并且指定CRuntimeClass对象的内容及CRuntimeClass对象的链接，这里起码有十行的代码才能完成。在每个需要有RTTI能力的类设计中都要重复那十多行代码是一件乏味的事情，也容易出错，所以MFC用了两个宏代替这些工作，即DECLARE\_DYNAMIC(类名)和IMPLEMENT\_DYNAMIC(类名，基类名)。从这两个宏我们可以看出在MFC名类中的CRuntimeClass对象构造连接只有类名及基类名的不同！

       到此，可能会有朋友问：为什么要用两个宏，用一个宏不可以代换CRuntimeClass对象构造连接吗？个人认为肯定可以，因为宏只是文字代换的游戏而已。但我们在编程之中，头文件与源文件是分开的，我们要在头文件头声明变量及方法，在源文件里实具体实现。即是说我们要在头文件中声明：

public:

static CRuntimeClass classXXX //XXX为类名

virtual CRuntime\* GetRuntimeClass() const;

       然后在源文件里实现：

CRuntimeClass\* XXX：：classXXX={……}；

CRuntime\* GetRuntimeClass() const;

{ return &XXX::classXXX;}//这里不能直接返回&classXXX，因为static变量是类拥有而不是对象拥有。

              我们一眼可以看出MFC中的DECLARE\_DYNAMIC(类名)宏应该这样定义：

#define DECLARE\_DYNAMIC(class\_name) public: static CRuntimeClass class##class\_name; virtual CRuntimeClass\* GetRuntimeClass() const;

其中##为连接符，可以让我们传入的类名前面加上class，否则跟原类同名，大家会知道产生什么后果。

       有了上面的DECLARE\_DYNAMIC(类名)宏之后，我们在头文件里写上一句

DECLARE\_DYNAMIC(XXX)

       宏展开后就有了我们想要的：

public:

static CRuntimeClass classXXX //XXX为类名

virtual CRuntime\* GetRuntimeClass() const;

       对于IMPLEMENT\_DYNAMIC(类名，基类名)，看来也不值得在这里代换文字了，大家知道它是知道回事，宏展开后为我们做了什么，再深究真是一点意义都没有！  
有了此链表之后，就像有了一张存放各类型的网，我们可以轻而易举地RTTI。

**IsKindOf函数**

       CObject有一个函数BOOL IsKindOf(const CRuntimeClass\* pClass) const;，被它以下所有派生类继承。

       此函数实现如下：

BOOL CObject::IsKindOf(const CRuntimeClass\* pClass) const

{

CRuntimeClass\* pClassThis=GetRuntimeClass();//获得自己的CRuntimeClass对象指针。

while(pClassThis!=NULL)

{

if(pClassThis==pClass) return TRUE;

pClassThis=pClassThis->m\_pBaseClass;//这句最关键，指向自己基类，再回头比较，一直到尽头m\_pBaseClass为NULL结束。

}

return FALSE;

}

       说到这里，运行时类型识别（RTTI）算是完成了。

来自：<http://blog.csdn.net/liyi268/article/details/304175>

**动态创建**

       MFC中很多地方都使用了动态创建技术。动态创建就是在程序运行时创建指定类的对象。例如MFC的单文档程序中，文档模板类的对象就动态创建了框架窗口对象、文档对象和视图对象。动态创建技术对于希望了解MFC底层运行机制的朋友来说，非常有必要弄清楚。

**不需要手动实例化对象的疑惑**

       MFC编程入门时，一般人都会有这样的疑惑：MFC中几个主要的类不需要我们设计也就罢了，为什么连实例化对象都不用我们来做？我们认为本该是：需要框架的时候，亲手写上CFrameWnd myFrame；需要视的时候，亲自打上CView myView;……。

       但MFC不给我们这个机会，致使我们错觉窗口没有实例化就弹出来了！但大伙想了一下，可能会一拍脑门，认为简单不过：MFC自动帮我们完成CView myView之流的代码不就行了么！其实不然，写MFC程序的时候，我们几乎要对每个大类进行派生改写。换句话说，MFC并不知道我们打算怎样去改写这些类，当然也不打算全部为我们“静态”创建这些类了。即使静态了创建这些类也没有用，因为我们从来也不会直接利用这些类的实例干什么事情。我们只知道，想做什么事情就往各大类里塞，不管什么变量、方法照塞，塞完之后，我们似乎并未实例化对象，程序就可以运行！

**CRuntimeClass链表**

       要做到把自己的类交给MFC，MFC就用同一样的方法，把不同的类一一准确创建，我们要做些什么事情呢？同样地，我们要建立链表，记录各类的关键信息，在动态创建的时候找出这些信息，就象上一节RTTI那样！我们可以设计一个类：

struct CRuntimeClass{

LPCSTR m\_lpszClassName; //类名指针

CObject\* (PASCAL \*m\_pfnCreateObject)(); //创建对象的函数的指针

CRuntimeClass\* m\_pBaseClass; //讲RTTI时介绍过

CRuntimeClass\* m\_pNextClass; //指向链表的下一个元素(许多朋友说上一节讲RTTI时并没有用到这个指针，我原本以为这样更好理解一些，因为没有这个指针，这个链表是无法连起来，而m\_pBaseClass仅仅是向基类走，在MFC的树型层次结构中m\_pBaseClass是不能遍历的)

CObject\* CreateObject(); //创建对象

static CRuntimeClass\* PASCAL Load(); //遍历整个类型链表，返回符合动态创建的对象。

static CRuntimeClass\* pFirstClass; //类型链表的头指针

};

      一下子往结构里面塞了那么多的东西，大家可以觉得有点头晕。至于CObject\* (PASCAL \*m\_pfnCreateObject)();，这定义函数指针的方法，大家可能有点陌生。函数指针在C++书籍里一般被定为选学章节，但MFC还是经常用到此类的函数，比如我们所熟悉的回调函数。简单地说m\_pfnCreateObject即是保存了一个函数的地址，它将会创建一个对象。即是说，以后，m\_pfnCreateObject指向不同的函数，我们就会创建不同类型的对象。

      有函数指针，我们要实现一个与原定义参数及返回值都相同一个函数，在MFC中定义为：

static CObject\* PASCAL CreateObject(){return new XXX};//XXX为类名。类名不同，我们就创建不同的对象。

       由此，我们可以如下构造CRuntimeClass到链表（伪代码）：

CRuntimeClass classXXX={

类名，

……，

XXX::CreateObject()， //m\_pfnCreateObject指向的函数

RUNTIME\_CLASS(基类名)， // RUNTIME\_CLASS宏可以返回CRuntimeClass对象指针。

NULL //m\_pNextClass暂时为空，最后会我们再设法让它指向旧链表表头。

};

       这样，我们用函数指针m\_pfnCreateObject（指向CreateObject函数)，就随时可new新对象了。并且大家留意到，我们在设计CRuntimeClass类对象的时候，只有类名（和基类名）的不同（我们用XXX代替的地方），其它的地方一样，这正是我们想要的，因为我们动态创建也象RTTI那样用到两个宏，只要传入类名和基类作宏参数，就可以满足条件。

       即是说，我们类说明中使用DECLARE\_DYNCREATE（CLASSNMAE）宏和在类的实现文件中使用IMPLEMENT\_DYNCREATE（CLASSNAME，BASECLASS）宏来为我们加入链表，至于这两个宏怎么为我们建立一个链表，我们自己可以玩玩文字代换的游戏，在此不一一累赘。但要说明的一点就是：动态创建宏xxx\_DYNCREATE包含了RTTI宏，即是说， xxx\_DYNCREATE是xxx\_DYNAMIC的“增强版”。

       到此，我们有必要了解一下上节课没有明讲的m\_pNextClass指针。因为MFC层次结构是树状的，并不是直线的。如果我们只有一个m\_pBaseClass指针，它只会沿着基类上去，会漏掉其它分支。在动态创建时，必需要检查整个链表，看有多少个要动态创建的对象，即是说要从表头（pFirstClass）开始一直遍历到表尾（m\_pNextClass=NULL），不能漏掉一个CRuntimeClass对象。

       所以每当有一个新的链表元素要加入链表的时候，我们要做的就是使新的链表元素成为表头，并且m\_pNextClass指向原来链表的表头，即像下面那样（当然，这些不需要我们操心，是RTTI宏帮助我们完成的）：

pNewClass->m\_pNextClass=CRuntimeClass::pFirstClass;//新元素的m\_pNextClass指针指向想加入的链表的表头。

CRuntimeClass::pFirstClass=pNewClass;//链表的头指针指向刚插入的新元素。

       好了，有了上面的链表，我们就可以分析动态创建了。

**动态创建的步骤**

       有了一个包含类名，函数指针，动态创建函数的链表，我们就可以知道应该按什么步骤去动态创建了：

       1、获得一要动态创建的类的类名（假设为A）。

       2、将A跟链表里面每个元素的m\_lpszClassName指向的类名作比较。

       3、若找到跟A相同的类名就返回A所属的CRuntimeClass元素的指针。

       4、判断m\_pfnCreateObject是否有指向创建函数，有则创建对象，并返回该对象。

       代码演示如下（以下两个函数都是CRuntimeClass类函数）：

///////////////以下为根据类名从表头向表尾查找所属的CRuntimeClass对象////////////

CRuntimeClass\* PASCAL CRuntimeClass::Load()

{

char szClassXXX[64];

CRuntimeClass\* pClass;

cin>>szClassXXX; //假定这是我们希望动态创建的类名

for(pClass=pFirstClass;pClass!=NULL;pClass=pClass->m\_pNextClass)

{

if(strcmp(szClassXXX,pClass->m\_lpszClassName)==0)

return pClass;

}

return NULL;

}

///////////根据CRuntimeClass创建对象///////////

CObject\* CRuntimeClass::CreateObject()

{

if(m\_pfnCreateObject==NULL) return NULL;

CObject \*pObject;

pObject=(\* m\_pfnCreateObject)(); //函数指针调用

return pObject;

}

       有了上面两个函数，我们在程序执行的时候调用，就可以动态创建对象了。

**简单实现动态创建**

       我们还可以更简单地实现动态创建，大家注意到，就是在我们的程序类里面有一个RUNTIME\_CLASS(class\_name)宏，这个宏在MFC里定义为：

RUNTIME\_CLASS(class\_name) ((CRuntimeClass\*)(&class\_name::class##class\_name))

       作用就是得到类的RunTime信息，即返回class\_name所属CRuntimeClass的对象。在我们的应用程序类(CMyWinApp)的InitInstance()函数下面的CSingleDocTemplate函数中，有：

RUNTIME\_CLASS(CMyDoc),

RUNTIME\_CLASS(CMainFrame), // main SDI frame window

RUNTIME\_CLASS(CMyView)

       构造文档模板的时候就用这个宏得到文档、框架和视的RunTime信息。有了RunTime信息，我们只要一条语句就可以动态创建了，如：

classMyView->CreateObject(); //对象直接调用用CRuntimeClass本身的CreateObject()

**总结**

       最后再总结和明确下动态创建的具体步骤：

       1、定义一个不带参数的构造函数（默认构造函数）；因为我们是用CreateObject()动态创建，它只有一条语句就是return new XXX，不带任何参数。所以我们要有一个无参构造函数。

       2、类说明中使用DECLARE\_DYNCREATE（CLASSNMAE）宏；和在类的实现文件中使用IMPLEMENT\_DYNCREATE（CLASSNAME，BASECLASS）宏；这个宏完成构造CRuntimeClass对象，并加入到链表中。

       3、使用时先通过宏RUNTIME\_CLASS得到类的RunTime信息，然后使用CRuntimeClass的成员函数CreateObject创建一个该类的实例。

       4、CObject\* pObject = pRuntimeClass->CreateObject();//完成动态创建。

来自：<http://blog.csdn.net/liyi268/article/details/310895>

**永久保存（串行化）**

       永久保存（串行化）是MFC的重要内容，可以用一句简明直白的话来形容其重要性：弄懂它以后，你就越来越像个程序员了！

       如果我们的程序不需要永久保存，那几乎可以肯定是一个小玩儿。那怕我们的记事本、画图等小程序，也需要保存才有真正的意义。

       对于MFC的很多地方我不甚满意，总觉得它喜欢拿一组低能而神秘的宏来故弄玄虚，但对于它的连续存储（serialize）机制，却是我十分钟爱的地方。在此，可让大家感受到面向对象的幸福。

       MFC的连续存储（serialize）机制俗称串行化。“在你的程序中尽管有着各种各样的数据，serialize机制会象流水一样按顺序存储到单一的文件中，而又能按顺序地取出，变成各种不同的对象数据。”不知我在说上面这一句话的时候，大家有什么反应，可能很多朋友直觉是一件很简单的事情，只是说了一个“爽”字就没有下文了。

**串行化原理的讨论**

       要实现象流水一样存储其实是一个很大的难题。试想，在我们的程序里有各式各样的对象数据。如画图程序中，里面设计了点类，矩形类，圆形类等等，它们的绘图方式及对数据的处理各不相同，用它们实现了成百上千的对象之后，如何存储起来？不想由可，一想头都大了：我们要在程序中设计函数store()，在我们单击“文件/保存”时能把各对象往里存储。那么这个store()函数要神通广大，它能清楚地知道我们设计的是什么样的类，产生什么样的对象。大家可能并不觉得这是一件很困难的事情，程序有能力知道我们的类的样子，对象也不过是一块初始化了存储区域罢了。就把一大堆对象“转换”成磁盘文件就行了。

       即使上面的存储能成立，但当我们单击“文件/打开”时，程序当然不能预测用户想打开哪个文件，并且当打开文件的时候，要根据你那一大堆垃圾数据new出数百个对象，还原为你原来存储时的样子，你又该怎么做呢？

       试想，要是我们有一个能容纳各种不同对象的容器，这样，用户用我们的应用程序打开一个磁盘文件时，就可以把文件的内容读进我们程序的容器中。把磁盘文件读进内存，然后识别它“是什么对象”是一件很难的事情。首先，保存过程不像电影的胶片，把景物直接映射进去，然后，看一下胶片就知道那是什么内容。可能有朋友说它象录像磁带，拿着录像带我们看不出里面变化的磁场信号，但经过录像机就能把它还原出来。

       其实不是这样的，比如保存一个矩形，程序并不是把矩形本身按点阵存储到磁盘中，因为我们绘制矩形的整个过程只不过是调用一个GDI函数罢了。它保存只是坐标值、线宽和某些标记等。程序面对“00 FF”这样的东西，当然不知道它是一个圆或是一个字符！

       拿刚才录像带的例子，我们之所以能最后放映出来，前提我们知道这对象是“录像带”，即确定了它是什么类对象。如果我们事先只知道它“里面保存有东西，但不知道它是什么类型的东西”，这就导致我们无法把它读出来。拿录像带到录音机去放，对录音机来说，那完全是垃圾数据。即是说，要了解永久保存，要对动态创建有深刻的认识。

       现在大家可以知道困难的根源了吧。我们在写程序的时候，会不断创造新的类，构造新的对象。这些对象，当然是旧的类对象（如MyDocument）从未见过的。那么，我们如何才能使文档对象可以保存自己新对象呢，又能动态创建自己新的类对象呢？

       许多朋友在这个时候想起了CObject这个类，也想到了虚函数的概念。于是以为自己“大致了解”串行化的概念。他们设想：“我们设计的MyClass（我们想用于串行化的对象）全部从CObject类派生，CObject类对象当然是MyDocument能认识的。”这样就实现了一个目的：本来MyDocument不能识别我们创建的MyClass对象，但它能识别CObject类对象。由于MyClass从CObject类派生，构造的新类对象“是一个CObject”，所以MyDocument能把我们的新对象当作CObiect对象读出。或者根据书本上所说的：打开或保存文件的时候，MyDocument会调用Serialize()，MyDocument的Serialize()函会呼叫我们创建类的Serialize函数[即是在MyDocument Serialize()中调用：m\_pObject->Serialize()，注意：在此m\_pObject是CObject类指针，它可以指向我们设计的类对象]。最终结果是MyDocument的读出和保存变成了我们创建的类对象的读出和保存，这种认识是不明朗的。

       有意思还有，在网上我遇到几位自以为懂了Serialize的朋友，居然不约而同的犯了一个很低级得让人不可思议的错误。他们说：Serialize太简单了！Serialize()是一个虚函数，虚函数的作用就是“优先派生类的操作”。所以MyDocument不实现Serialize()函数，留给我们自己的MyClass对象去调用Serialize()……真是哭笑不得，我们创建的类MyClass并不是由MyDocument类派生，Serialize()函数为虚在MyDocument和MyClass之间没有任何意义。MyClass产生的MyObject对象仅仅是MyDocument的一个成员变量罢了。

       话说回来，由于MyClass从CObject派生，所以CObject类型指针能指向MyClass对象，并且能够让MyClass对象执行某些函数（特指重载的CObject虚函数），但前提必须在MyClass对象实例化了，即在内存中占领了一块存储区域之后。不过，我们的问题恰恰就是在应用程序随便打开一个文件，面对的是它不认识的MyClass类，当然实例化不了对象。

       幸好我们在上一节课中懂得了动态创建。即想要从CObject派生的MyClass成为可以动态创建的对象只要用到DECLARE\_DYNAMIC/IMPLEMENT\_DYNAMIC宏就可以了（注意：最终可以Serialize的对象仅仅用到了DECLARE\_SERIAL/IMPLEMENT\_SERIAL宏，这是因为DECLARE\_SERIAL/IMPLEMENT\_SERIAL包含了DECLARE\_DYNAMIC/IMPLEMENT\_DYNAMIC宏）。

**深入理解串行化**

       从解决上面的问题中，我们可以分步理解了：

       1、Serialize的目的：让MyDocument对象在执行打开/保存操作时，能读出（构造）和保存它不认的MyClass类对象。

       2、MyDocument对象在执行打开/保存操作时会调用它本身的Serialize()函数。但不要指望它会自动保存和读出我们的MyClass类对象。这个问题很容易解决，如下即可：

MyDocument::Serialize（）{

// 在此函数调用MyClass类的Serialize（）就行了！即

MyObject.Serialize（）;

}

       3、我们希望MyClass对象为可以动态创建的对象，所以要求在MyClass类中加上DECLARE\_DYNAMIC/IMPLEMENT\_DYNAMIC宏。

       但目前的Serialize机制还很抽象。我们仅仅知道了表面上的东西，实际又是如何的呢？下面作一个简单深刻的详解。

       先看一下我们文档类的Serialize()：

void CMyDoc::Serialize(CArchive& ar)

{

if (ar.IsStoring())

{

// TODO: add storing code here

}

else

{

// TODO: add loading code here

}

}

       目前这个子数什么也没做（没有数据的读出和写入），CMyDoc类正等待着我们去改写这个函数。现在假设CMyDoc有一个MFC可识别的成员变量m\_MyVar，那么函数就可改写成如下形式：

void CMyDoc::Serialize(CArchive& ar)

{

if (ar.IsStoring()) //读写判断

{

ar<<m\_MyVar; //写

}

else

{

ar>>m\_MyVar; //读

}

}

       许多网友问：自己写的类（即MFC未包含的类）为什么不行？我们在CMyDoc里包含自写类的头文件MyClass.h，这样CMyDoc就认识MyDoc类对象了。这是一般常识性的错误，MyDoc类认识MyClass类对象与否并没有用，关键是CArchive类，即对象ar不认识MyClass（当然你梦想重写CArchive类当别论）。“>>”、“<<”都是CArchive重载的操作符。上面ar>>m\_MyVar说白即是在执行一个以ar和m\_MyVar 为参数的函数，类似于function(ar,m\_MyVar)罢了。我们当然不能传递一个它不认识的参数类型，也因此不会执行function(ar,m\_MyObject)了。

       【注：这里我们可以用指针。让MyClass从Cobject派生，一切又起了质的变化，假设我们定义了：MyClass \*pMyClass = new MyClass;因为MyClass从CObject派生，根据虚函数原理，pMyClass也是一个CObject\*，即pMyClass指针是CArchive类可认识的。所以执行上述function(ar, pMyClass)，即ar << pMyClass是没有太多的问题（在保证了MyClass对象可以动态创建的前提下）。】  
   
       回过头来，如果想让MyClass类对象能Serialize，就得让MyClass从CObject派生，Serialize()函数在CObject里为虚，MyClass从CObject派生之后就可以根据自己的要求去改写它，像上面改写CMyDoc::Serialize()方法一样。这样MyClass就得到了属于MyClass自己特有的Serialize()函数。

       现在，程序就可以这样写：

#include “MyClass.h”

……

void CMyDoc::Serialize(CArchive& ar)

{

//在此调用MyClass重写过的Serialize()

m\_MyObject.Serialize(ar); // m\_MyObject为MyClass实例

}

         至此，串行化工作就算完成了，简单直观的讲：从CObject派生自己的类，重写Serialize()。在此过程中，我刻意安排：在没有用到DECLARE\_SERIAL / IMPLEMENT\_SERIAL宏，也没有用到CArray等模板类的前提下就完成了串行化的工作。我看过某些书，总是一开始就讲DECLARE\_SERIAL / IMPLEMENT\_SERIAL宏或马上用CArray模板，让读者觉得串行化就是这两个东西，导致许多朋友因此找不着北。

**CArchive**

       最后再补充讲解一下有些抽象的CArchive。我们先看以下程序（注：以下程序包含动态创建等，请包含DECLARE\_SERIAL / IMPLEMENT\_SERIAL宏）

void MyClass::Serialize(CArchive& ar)

{

if (ar.IsStoring()) //读写判断

{

ar<< m\_pMyVar; //问题：ar 如何把m\_pMyVar所指的对象变量保存到磁盘？

}

else

{

pMyClass = new MyClass; //准备存储空间

ar>> m\_pMyVar;

}

}

       为回答上面的问题，即“ar<<XXX”的问题，这里给出一段模拟CArchive的代码。

       “ar<<XXX”是执行CArchive的运算符<<的重载。ar和XXX都是该重载函数中的一参数而已。函数大致如下：

CArchive& operator<<( CArchive& ar, const CObject\* pOb)

{

…………

//以下为CRuntimeClass链表中找到、识别pOb资料。

CRuntimeClass\* pClassRef = pOb->GetRuntimeClass();

//保存pClassRef即类信息（略）

((CObject\*)pOb)->Serialize();//保存MyClass数据

…………

}

       从上面可以看出，因为Serialize()为虚函数，即“ar<<XXX”的结果是执行了XXX所指向对象本身的Serialize()。对于“ar>>XXX”，虽然不是“ar<<XXX”逆过程，大家可能根据动态创建和虚函数的原理料想到它。

       至此，永久保存算是写完了。在此过程中，我一直努力用最少的代码，详尽的解释来说明问题。以前我为本课题写过一个版本，并在几个论坛上发表过，但不知怎么在网上遗失（可能被删除）。所以这篇文章是我重写的版本。记得第一个版本中，我是对DECLARE\_SERIAL / IMPLEMENT\_SERIAL和可串行化的数组及链表对象说了许多。这个版本中我对DECLARE\_SERIAL / IMPLEMENT\_SERIAL其中奥秘几乎一句不提，目的是让大家能找到中心，有更简洁的永久保存的概念，我觉得这种感觉很好！

来自：<http://blog.csdn.net/liyi268/article/details/623367>

**消息映射和命令传递**

       作为C++程序员，我们总是希望自己程序的所有代码都是自己写出来的，如果使用了其他的一些库，也总是千方百计想弄清楚其中的类和函数的原理，否则就会感觉不踏实。所以，我们对于在进行MFC视窗程序设计时经常要用到的消息机制也不满足于会使用，而是希望能理解个中道理。本文就为大家剖析MFC消息映射和命令传递的原理。

**理解MFC消息机制的必要性**

       说到消息，在MFC中，“最熟悉的神秘”可以说是消息映射了，那是我们刚开始接触MFC时就要面对的东西。有过SDK编程经验的朋友转到MFC编程的时候，一下子觉得什么都变了样。特别是窗口消息及对消息的处理跟以前相比，更是风马牛不相及的。如文档不是窗口，是怎样响应命令消息的呢？

       初次用MFC编程，我们只会用MFC ClassWizard为我们做大量的东西，最主要的是添加消息响应。记忆中，如果是自已添加消息响应，我们应何等的小心翼翼，对BEGIN\_MESSAGE\_MAP()……END\_MESSAGE\_MAP()更要奉若神灵。它就是一个魔盒子，把我们的咒语放入恰当的地方，就会发生神奇的力量，放错了，自己的程序就连“命”都没有。

       据说，知道得太多未必是好事。我也曾经打算不去理解这神秘的区域，觉得编程的时候知道自己想做什么就行了。MFC外表上给我们提供了东西，直观地说，不但给了我个一个程序的外壳，更给我们许多方便。微软的出发点可能是希望达到“傻瓜编程”的结果，试想，谁不会用ClassWizard？大家知道，Windows是基于消息的，有了ClassWizard，你又会添加类，又会添加消息，那么你所学的东西似乎学到头了。于是许多程序员认为“我们没有必要走SDK的老路，直接用MFC编程，新的东西通常是简单、直观、易学……”。

       到你真正想用MFC编程的时候，你会发觉光会ClassWizard的你是多么的愚蠢。MFC不是一个普通的类库，普通的类库我们完全可以不理解里面的细节，只要知道这些类库能干什么，接口参数如何就万事大吉。如string类，操作顺序是定义一个string对象，然后修改属性，调用方法。但对于MFC，并不是在你的程序中写上一句“#include <MFC.h>”，然后就使用MFC类库的。

       MFC是一块包着糖衣的牛骨头。你很轻松地写出一个单文档窗口，在窗口中间打印一句“I love MFC!”，然后，恶梦开始了……想逃避，打算永远不去理解MFC内幕？门都没有！在MFC这个黑暗神秘的洞中，即使你打算摸着石头前行，也注定找不到出口。对着MFC这块牛骨头，微软温和、民主地告诉你“你当然可以选择不啃掉它，咳咳……但你必然会因此而饿死！”

**MFC消息机制与SDK的不同**

       消息映射与命令传递体现了MFC与SDK的不同。在SDK编程中，没有消息映射的概念，它有明确的回调函数，通过一个switch语句去判断收到了何种消息，然后对这个消息进行处理。所以，在SDK编程中，会发送消息和在回调函数中处理消息就差不多可以写SDK程序了。

       在MFC中，看上去发送消息和处理消息比SDK更简单、直接，但可惜不直观。举个简单的例子，如果我们想自定义一个消息，SDK是非常简单直观的，用一条语句：SendMessage(hwnd,message/\*一个大于或等于WM\_USER的数字\*/,wparam,lparam)，之后就可以在回调函数中处理了。但MFC就不同了，因为你通常不直接去改写窗口的回调函数，所以只能亦步亦趋对照原来的MFC代码，把消息放到恰当的地方。这确实是一样很痛苦的劳动。

       要了解MFC消息映射原理并不是一件轻松的事情。我们可以逆向思维，想象一下消息映射为我们做了什么工作。MFC在自动化给我们提供了很大的方便，比如，所有的MFC窗口都使用同一窗口过程，即所有的MFC窗口都有一个默认的窗口过程。不像在SDK编程中，要为每个窗口类写一个窗口过程。

**消息队列**

对于每个用MFC开发的GUI程序，他们都有一个CMy\*\*\*App,该类继承自CWinApp，而CWinApp继承自CWinThread, CWinApp是一个GUI线程，系统会为其维护一个THREADINFO结构，

 消息队列包含在一个叫THREADINFO的结构中，有四个队列：

[?](http://my.oschina.net/myspaceNUAA/blog/64682?p=1)

|  |  |
| --- | --- |
| 1  2  3  4 | Sent Message Queue     发送消息队列 该队列保存其他程序通过SendMessage给该线程发送的消息  Posted Message Queue   登记消息队列 该队列保存其他队列通过PostMessage给该线程发送的消息  Visualized Input Queue 输入消息队列 保存系统队列分发过来的消息，比如鼠标或者键盘的消息  Reply Message Queue    应答消息队列 保存向窗体发送消息后的结果，比如sendMessage操作结束后，接收消息方会发送一个Reply消息给发送方的Reply队列中，以唤醒发送队列。 |

**MFC消息映射原理**

       对于消息映射，最直截了当地猜想是：消息映射就是用一个数据结构把“消息”与“响应消息函数名”串联起来。这样，当窗口感知消息发生时，就对结构查找，找到相应的消息响应函数执行。其实这个想法也不能简单地实现：我们每个不同的MFC窗口类，对同一种消息，有不同的响应方式。即是说，对同一种消息，不同的MFC窗口会有不同的消息响应函数。

       这时，大家又想了一个可行的方法。我们设计窗口基类（CWnd）时，我们让它对每种不同的消息都来一个消息响应，并把这个消息响应函数定义为虚函数。这样，从CWnd派生的窗口类对所有消息都有了一个空响应，我们要响应一个特定的消息就重载这个消息响应函数就可以了。但这样做的结果，一个几乎什么也不做的CWnd类要有几百个“多余”的函数，哪怕这些消息响应函数都为纯虚函数，每个CWnd对象也要背负着一个巨大的虚拟表，这也是得不偿失的。

       许多朋友在学习消息映射时苦无突破，其原因是一开始就认为MFC的消息映射的目的是为了替代SDK窗口过程的编写——这本来没有理解错。但他们还有多一层的理解，认为既然是替代“旧”的东西，那么MFC消息映身应该是更高层次的抽象、更简单、更容易认识。但结果是，如果我们不通过ClassWizard工具，手动添加消息是相当迷茫的一件事。

       所以，我们在学习MFC消息映射时，首先要弄清楚：消息映射的目的，不是为是更加快捷地向窗口过程添加代码，而是一种机制的改变。如果不想改变窗口过程函数，那么应该在哪里进行消息响应呢？许多朋友一知半解地认为：我们可以用HOOK技术，抢在消息队列前把消息抓取，把消息响应提到窗口过程的外面。再者，不同的窗口，会有不同的感兴趣的消息，所以每个MFC窗口都应该有一个表把感兴趣的消息和相应消息响应函数连系起来。然后得出——消息映射机制执行步骤是：当消息发生，我们用HOOK技术把本来要发送到窗口过程的消息抓获，然后对照一下MFC窗口的消息映射表，如果是表里面有的消息，就执行其对应的函数。

       当然，用HOOK技术，我们理论上可以在不改变窗口过程函数的情况下，可以完成消息响应。MFC确实是这样做的，但实际操作起来可能跟你的想象差别很大。

       现在我们来编写消息映射表，我们先定义一个结构，这个结构至少有两个项：一是消息ID，二是响应该消息的函数。如下：

struct AFX\_MSGMAP\_ENTRY

{

UINT nMessage; //感兴趣的消息

AFX\_PMSG pfn; //响应以上消息的函数指针

}

       当然，只有两个成员的结构连接起来的消息映射表是不成熟的。Windows消息分为标准消息、控件消息和命令消息，每类型的消息都是包含数百不同ID、不同意义、不同参数的消息。我们要准确地判别发生了何种消息，必须再增加几个成员。还有，对于AFX\_PMSG pfn，实际上等于作以下声明：

       void (CCmdTarget::\*pfn)();    // 提示：AFX\_PMSG为类型标识，具体声明是：typedef void (AFX\_MSG\_CALL CCmdTarget::\*AFX\_PMSG)(void);

       pfn是一个不带参数和返回值的CCmdTarget类型成员函数指针，只能指向CCmdTarget类中不带参数和返回值的成员函数，这样pfn更为通用，但我们响应消息的函数许多需要传入参数的。为了解决这个矛盾，我们还要增加一个表示参数类型的成员。当然，还有其它……

       最后，MFC我们消息映射表成员结构如下定义：

struct AFX\_MSGMAP\_ENTRY

{

UINT nMessage; //Windows 消息ID

UINT nCode; // 控制消息的通知码

UINT nID; //命令消息ID范围的起始值

UINT nLastID; //命令消息ID范围的终点

UINT nSig; // 消息的动作标识

AFX\_PMSG pfn;

};

       有了以上消息映射表成员结构，我们就可以定义一个AFX\_MSGMAP\_ENTRY类型的数组，用来容纳消息映射项。定义如下：

       AFX\_MSGMAP\_ENTRY \_messageEntries[];

       但这样还不够，每个AFX\_MSGMAP\_ENTRY数组，只能保存着当前类感兴趣的消息，而这仅仅是我们想处理的消息中的一部分。对于一个MFC程序，一般有多个窗口类，里面都应该有一个AFX\_MSGMAP\_ENTRY数组。

       我们知道，MFC还有一个消息传递机制，可以把自己不处理的消息传送给别的类进行处理。为了能查找各下MFC对象的消息映射表，我们还要增加一个结构，把所有的AFX\_MSGMAP\_ENTRY数组串联起来。于是，我们定义了一个新结构体：

struct AFX\_MSGMAP

{

const AFX\_MSGMAP\* pBaseMap; //指向别的类的AFX\_MSGMAP对象

const AFX\_MSGMAP\_ENTRY\* lpEntries; //指向自身的消息表

};

       之后，在每个打算响应消息的类中声明这样一个变量：AFX\_MSGMAP messageMap，让其中的pBaseMap指向基类或另一个类的messageMap，那么将得到一个AFX\_MSGMAP元素的单向链表。这样，所有的消息映射信息形成了一张消息网。

       当然，仅有消息映射表还不够，它只能把各个MFC对象的消息、参数与相应的消息响应函数连成一张网。为了方便查找，MFC在上面的类中插入了两个函数（其中theClass代表当前类）：

       一个是\_GetBaseMessageMap()，用来得到基类消息映射的函数。函数原型如下：

const AFX\_MSGMAP\* PASCAL theClass::\_GetBaseMessageMap() /

{ return &baseClass::messageMap; } /

      另一个是GetMessageMap() ，用来得到自身消息映射的函数。函数原型如下：

const AFX\_MSGMAP\* theClass::GetMessageMap() const /

{ return &theClass::messageMap; } /

       有了消息映射表之后，我们得讨论到问题的关键，那就是消息发生以后，其对应的响应函数如何被调用。大家知道，所有的MFC窗口，都有一个同样的窗口过程——AfxWndProc(…)。在这里顺便要提一下的是，看过MFC源代码的朋友都得，从AfxWndProc函数进去，会遇到一大堆曲折与迷团，因为对于这个庞大的消息映射机制，MFC要做的事情很多，如优化消息，增强兼容性等，这一大量的工作，有些甚至用汇编语言来完成，对此，我们很难深究它。所以我们要省略大量代码，理性地分析它。

       对已定型的AfxWndProc来说，对所有消息，最多只能提供一种默认的处理方式。这当然不是我们想要的。我们想通过AfxWndProc最终执行消息映射网中对应的函数。那么，这个执行路线是怎么样的呢？

       从AfxWndProc下去，最终会调用到一个函数OnWndMsg。请看代码：

LRESULT CALLBACK AfxWndProc(HWND hWnd,UINT nMsg,WPARAM wParam, LPARAM lParam)

{

……

CWnd\* pWnd = CWnd::FromHandlePermanent(hWnd); //把对句柄的操作转换成对CWnd对象。

Return AfxCallWndProc(pWnd,hWnd,nMsg,wParam,lParam);

}

       把对句柄的操作转换成对CWnd对象是很重要的一件事，因为AfxWndProc只是一个全局函数，当然不知怎么样去处理各种windows窗口消息，所以它聪明地把处理权交给windows窗口所关联的MFC窗口对象。

       现在，大家几乎可以想象得到AfxCallWndProc要做的事情，不错，它当中有一句：

pWnd->WindowProc(nMsg,wParam,lParam);

       到此，MFC窗口过程函数变成了自己的一个成员函数。WindowProc是一个虚函数，我们甚至可以通过改写这个函数去响应不同的消息，当然，这是题外话。

       WindowProc会调用到CWnd对象的另一个成员函数OnWndMsg，下面看看大概的函数原型是怎么样的：

BOOL CWnd::OnWndMsg(UINT message,WPARAM wParam,LPARAM lParam,LRESULT\* pResult)

{

if(message==WM\_COMMAND)

{

OnCommand(wParam,lParam);

……

}

if(message==WM\_NOTIFY)

{

OnCommand(wParam,lParam,&lResult);

……

}

const AFX\_MSGMAP\* pMessageMap;

pMessageMap=GetMessageMap();

const AFX\_MSGMAP\_ENTRY\* lpEntry;

/\*以下代码作用为：用AfxFindMessageEntry函数从消息入口pMessageMap处查找指定消息，如果找到，返回指定消息映射表成员的指针给lpEntry。然后执行该结构成员的pfn所指向的函数\*/

if((lpEntry=AfxFindMessageEntry(pMessageMap->lpEntries,message,0,0)!=NULL)

{

lpEntry->pfn();/\*注意：真正MFC代码中没有用这一条语句。上面提到，不同的消息参数代表不同的意义和不同的消息响应函数有不同类型的返回值。而pfn是一个不带参数的函数指针，所以真正的MFC代码中，要根据对象lpEntry的消息的动作标识nSig给消息处理函数传递参数类型。这个过程包含很复杂的宏代换，大家在此知道：找到匹配消息，执行相应函数就行！\*/

}

}

**MFC命令传递**

       在上面的代码中，大家看到了OnWndMsg能根据传进来的消息参数，查找到匹配的消息和执行相应的消息响应。但这还不够，我们平常响应菜单命令消息的时候，原本属于框架窗口（CFrameWnd）的WM\_COMMAND消息，却可以放到视对象或文档对象中去响应。其原理如下：

       我们看上面函数OnWndMsg原型中看到以下代码：

if(message==WM\_COMMAND)

{

OnCommand(wParam,lParam);

……

}

       即对于命令消息，实际上是交给OnCommand函数处理。而OnCommand是一个虚函数，即WM\_COMMAND消息发生时，最终是发生该消息所对应的MFC对象去执行OnCommand。比如点框架窗口菜单，即向CFrameWnd发送一个WM\_COMMAND，将会导致CFrameWnd::OnCommand(wParam,lParam)的执行。且看该函数原型：

BOOL CFrameWnd::OnCommand(WPARAM wParam,LPARAM lParam)

{

……

return CWnd:: OnCommand(wParam,lParam);

}

       可以看出，它最后把该消息交给CWnd:: OnCommand处理。再看：

BOOL CWnd::OnCommand(WPARAM wParam,LPARAM lParam)

{

……

return OnCmdMsg(nID,nCode,NULL,NULL);

}

       这里包含了一个C++多态性很经典的问题。在这里，虽然是执行CWnd类的函数，但由于这个函数在CFrameWnd:: OnCmdMsg里执行，即当前指针是CFrameWnd类指针，再有OnCmdMsg是一个虚函数，所以如果CFrameWnd改写了OnCommand，程序会执行CFrameWnd::OnCmdMsg(…)。

       对CFrameWnd::OnCmdMsg(…)函数的原理扼要分析如下：

BOOL CFrameWnd:: OnCmdMsg(…)

{

CView pView = GetActiveView();//得到活动视指针。

if(pView-> OnCmdMsg(…))

return TRUE; //如果CView类对象或其派生类对象已经处理该消息，则返回。

……//否则，同理向下执行，交给文档、框架、及应用程序执行自身的OnCmdMsg。

}

       到此，CFrameWnd:: OnCmdMsg完成了把WM\_COMMAND消息传递到视对象、文档对象及应用程序对象实现消息响应。

       写了这么多，我们已经清楚了MFC消息映射与命令传递的大致过程。

        一个按钮点击事件的过程如下：  
CWinThread::PumpMessage -> CWnd::PretranslateMessage -> CWnd::WalkPreTranslateMessate -> CD1Dlg::PreTranslateMessage -> CDialog::PreTranslateMessage -> CWnd::PreTranslateInput   -> CWnd::IsDialogMessageA -> USER32内核 - > AfxWndProcBase -> AfxWndProc -> AfxCallWndProc -> CWnd::WindowProc -> CWnd::OnWndMsg -> CWnd::OnCommand -> CDialog::OnCmdMsg -> CCmdTarget::OnCmdMsg -> \_AfxDispatchCmdMsg -> CD1Dlg::OnButton1()

**MFC消息映射宏**

       现在，我们来看MFC“神秘代码”，会发觉好看多了。

       先看DECLARE\_MESSAGE\_MAP()宏，它在MFC中定义如下：

#define DECLARE\_MESSAGE\_MAP() /

private: /

static const AFX\_MSGMAP\_ENTRY \_messageEntries[]; /

protected: /

static AFX\_DATA const AFX\_MSGMAP messageMap; /

virtual const AFX\_MSGMAP\* GetMessageMap() const; /

       可以看出DECLARE\_MESSAGE\_MAP()定义了我们熟悉的两个结构和一个函数，显而易见，这个宏为每个需要实现消息映射的类提供了相关变量和函数。

       现在集中精力来看一下BEGIN\_MESSAGE\_MAP，END\_MESSAGE\_MAP和ON\_COMMAND三个宏，它们在MFC中定义如下（其中ON\_COMMAND与另外两个宏并没有定义在同一个文件中，把它放到一起是为了好看）：

#define BEGIN\_MESSAGE\_MAP(theClass, baseClass) /

const AFX\_MSGMAP\* theClass::GetMessageMap() const /

{ return &theClass::messageMap; } /

AFX\_COMDAT AFX\_DATADEF const AFX\_MSGMAP theClass::messageMap = /

{ &baseClass::messageMap, &theClass::\_messageEntries[0] }; /

AFX\_COMDAT const AFX\_MSGMAP\_ENTRY theClass::\_messageEntries[] = /

{ /

#define ON\_COMMAND(id, memberFxn) /

{ WM\_COMMAND, CN\_COMMAND, (WORD)id, (WORD)id, AfxSig\_vv, (AFX\_PMSG)&memberFxn },

#define END\_MESSAGE\_MAP() /

{0, 0, 0, 0, AfxSig\_end, (AFX\_PMSG)0 } /

}; /

      一下子看三个宏觉得有点复杂，但这仅仅是复杂，公式性的文字代换并不是很难。且看下面例子，假设我们框架中有一菜单项为“Test”，即定义了如下宏：

BEGIN\_MESSAGE\_MAP(CMainFrame, CFrameWnd)

ON\_COMMAND(ID\_TEST, OnTest)

END\_MESSAGE\_MAP()

       那么宏展开之后得到如下代码：

const AFX\_MSGMAP\* CMainFrame::GetMessageMap() const

{ return &CMainFrame::messageMap; }

///以下填入消息表映射信息

const AFX\_MSGMAP CMainFrame::messageMap =

{ &CFrameWnd::messageMap, &CMainFrame::\_messageEntries[0] };

//下面填入保存着当前类感兴趣的消息，可填入多个AFX\_MSGMAP\_ENTRY对象

const AFX\_MSGMAP\_ENTRY CMainFrame::\_messageEntries[] =

{

{ WM\_COMMAND, CN\_COMMAND, (WORD)ID\_TEST, (WORD)ID\_TEST, AfxSig\_vv, (AFX\_PMSG)&OnTest }, // 加入的ID\_TEST消息参数

{0, 0, 0, 0, AfxSig\_end, (AFX\_PMSG)0 } //本类的消息映射的结束项

};

      大家知道，要完成ID\_TEST消息映射，还要定义和实现OnTest函数。即在头文件中写afx\_msg void OnTest()并在源文件中实现它。根据以上所学的东西，我们知道了当ID为ID\_TEST的命令消息发生，最终会执行到我们写的OnTest函数。

       至此，MFC六大关键技术写完了。其中写得最难的是消息映射与命令传递，除了技术复杂之外，最难的是有许多避不开的代码。为了大家看得轻松一点，我把那繁杂的宏放在文章最后，希望能给你阅读带来方便。